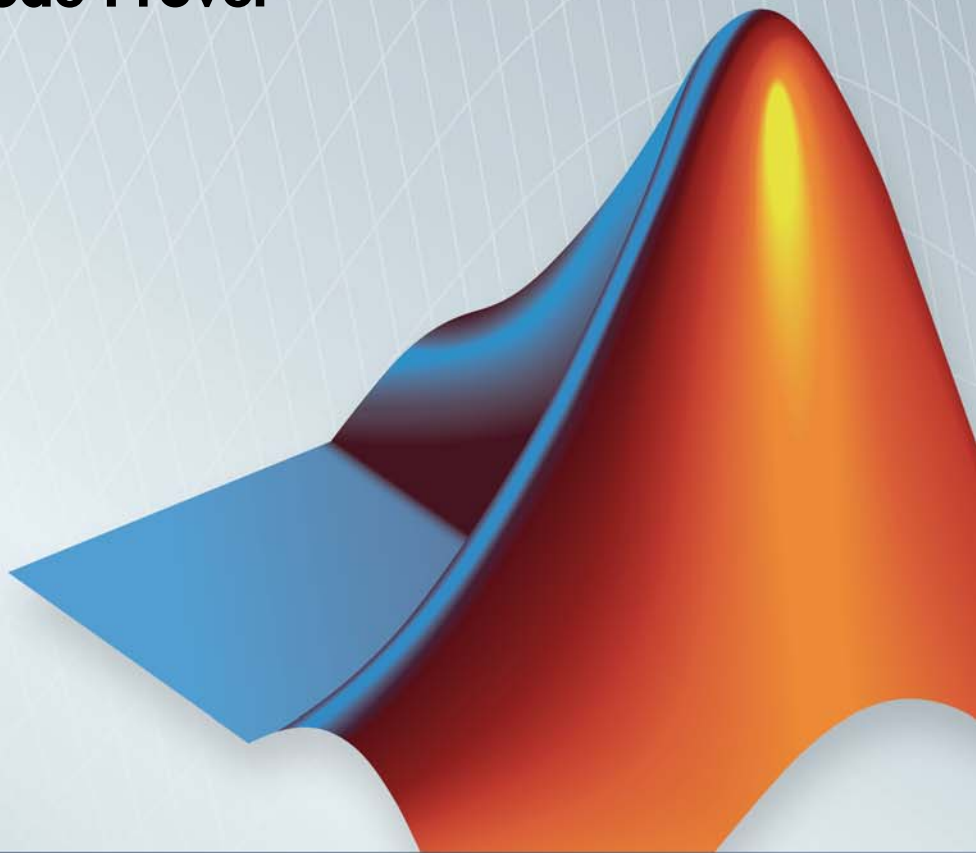


Polyspace® Code Prover™

User's Guide

R2014a



MATLAB® & SIMULINK®



How to Contact MathWorks



www.mathworks.com
comp.soft-sys.matlab
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

For contact information about worldwide offices, see the MathWorks Web site.

Polyspace® Code Prover™ User's Guide

© COPYRIGHT 2013–2014 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)

Introduction to Polyspace Products

1

Polyspace Code Prover Product Description	1-2
Key Features	1-2
Polyspace Verification	1-3
Polyspace Verification	1-3
Value of Polyspace Verification	1-3
How Polyspace Verification Works	1-6
What is Static Verification	1-6
Exhaustiveness	1-7
Related Products	1-8
Polyspace Products for Verifying Ada Code	1-8
Polyspace Bug Finder	1-8
Tool Qualification and Certification	1-9

How to Use Polyspace Software

2

Polyspace Verification and the Software Development Cycle	2-2
Software Quality and Productivity	2-2
Best Practices for Verification Workflow	2-3
Analysis and Review Criteria	2-4
Choose Robustness or Contextual Verification	2-4
Choose Coding Rules	2-5
Choose Strict or Permissive Verification	2-6
Define Software Quality Levels	2-7

Implement Process for Verification	2-9
Overview of the Polyspace Process	2-9
Define Process to Meet Your Goals	2-11
Apply Process to Assess Code Quality	2-12
Improve Your Verification Process	2-12
Sample Workflows for Polyspace Verification	2-13
Overview of Verification Workflows	2-13
Software Developers and Testers – Standard Development Process	2-14
Software Developers and Testers – Rigorous Development Process	2-17
Quality Engineers – Code Acceptance Criteria	2-21
Quality Engineers – Certification/Qualification	2-24
Model-Based Design Users — Verifying Generated Code ..	2-25
Project Managers — Integrating Polyspace Verification with Configuration Management Tools	2-29

Setting Up Project: Basic Steps

3

What is a Project?	3-2
What is a Project Template?	3-3
Create New Project	3-4
Add Source Files and Include Folders	3-6
Specify Results Folder	3-8
Specify Analysis Options	3-10
Specify Options in User Interface	3-10
Specify Options from DOS and UNIX Command Line	3-11
Specify Options from MATLAB Command Line	3-11
Save Analysis Options as Project Template	3-13

Specify Text Editor	3-16
----------------------------------	-------------

Setting Up Project : Advanced Steps

4

Create Projects Automatically from Your Build	
System	4-2
Create Project in User Interface	4-2
Create Project from DOS and UNIX Command Line	4-3
Create Project from MATLAB Command Line	4-4
Requirements for Project Creation from Build	
Systems	4-6
Create Multiple Modules	4-7
Create Multiple Analysis Option Configurations	4-8
Customize Results Folder Location and Name	4-9
Specify Functions Not Called by Generated Main	4-10
Specify Options to Match Your Quality Goals	4-11
Choose Contextual Verification Options for C Code	4-11
Choose Contextual Verification Options for C++ Code	4-12
Choose Strict or Permissive Verification Options	4-14
Set Up Project to Check Coding Rules	4-16
Polyspace Coding Rules Checker Overview	4-16
Check Compliance with MISRA C Coding Rules	4-16
Check Compliance with C++ Coding Rules	4-17
Set Up Project to Automatically Test Orange	4-19
Polyspace Automatic Orange Tester	4-19
Enable Automatic Orange Tester	4-19

Setting Up Project: Additional Information

5

Create Projects Using Visual Studio Information	5-2
Use Visual Studio Project	5-2
Trace Visual Studio Build	5-3
Cannot create project from Visual Studio build	5-6
Storage of Polyspace Preferences	5-7

Emulating Your Runtime Environment

6

Set Up a Target	6-2
Target & Compiler Overview	6-2
Specify Target and Compiler	6-3
Predefined Target Processor Specifications	6-3
Modify Predefined Target Processor Attributes	6-6
Define Generic Target Processors	6-7
Common Generic Targets	6-9
View or Modify Existing Generic Targets	6-10
Delete Generic Target	6-11
Compile Operating System Dependent Code	6-12
Address Alignment	6-19
Ignore or Replace Keywords Before Compilation	6-20
Language Extensions	6-23
Verify Keil or IAR Dialects	6-23
Gather Compilation Options Efficiently	6-31
Verify C Application Without a “Main”	6-33
Main Generator Overview	6-33
Automatically Generate a Main	6-33
Manually Generate a Main	6-35
Specify Call Sequence	6-36
Specify Functions Not Called by Generated Main	6-37
Main Generator Assumptions	6-38

Polyspace C++ Class Analyzer	6-39
Why Provide a Class Analyzer	6-39
How the Class Analyzer Works	6-40
Sources Verified	6-40
Architecture of Generated Main	6-40
Class Verification Log File	6-41
Characteristics of Class and Log File Messages	6-42
Behavior of Global Variables and Members	6-42
Methods and Class Specifics	6-45
Simple Class	6-47
Simple Inheritance	6-49
Multiple Inheritance	6-50
Abstract Classes	6-51
Virtual Inheritance	6-52
Other Types of Classes	6-53
Data Range Specifications (DRS)	6-55
Specify Data Ranges Using DRS Template	6-56
Specify Data Ranges Using Existing DRS Configuration	6-58
Edit Existing DRS Configuration	6-59
Remove Non Applicable Entries from DRS File	6-60
Specify Data Ranges Using Text Files	6-61
DRS Text File Format	6-62
Tips for Creating DRS Text Files	6-63
Example DRS Text File	6-63
Perform Efficient Module Testing with DRS	6-65
Reduce Oranges with DRS	6-67
Why Is DRS Most Effective on Module Testing?	6-68
Example	6-68
DRS Configuration Settings	6-71

Variable Scope	6-76
DRS Support for Structures	6-78
DRS Support for Union Members	6-78
XML Format of DRS File	6-80
Syntax Description — XML Elements	6-80
Valid Modes and Default Values	6-85

Preparing Source Code for Verification

7

Stubbing Overview	7-3
When to Provide Function Stubs	7-4
Manual stubs	7-5
Provide Stubs for Functions	7-6
Stubbing Examples	7-7
Example: Specification	7-7
Example: Colored Source Code	7-8
Automatic Stubbing Behavior for C++	
Pointer/Reference	7-10
Specify Functions to Stub Automatically	7-12
Special Characters in Function Names	7-12
Function Syntax for C++	7-12
Constrain Data with Stubbing	7-14
Add Precision Constraints Using Stubs	7-14
Default Behavior of Global Data	7-15
Constraining the Data	7-16
Apply the Technique	7-16
Integer Example	7-16
Recode Specific Functions	7-17

Default and Alternative Behavior for Stubbing	7-20
Function Pointer Cases	7-22
Stub Functions with Variable Argument Number	7-23
Stub Standard Library Functions	7-25
Check Variable Ranges with <code>assert</code>	7-26
Check Global Variable Ranges with Global Assert	7-27
Model Variables External to Application	7-29
External Variables	7-30
Volatile Variables	7-31
Absolute Addresses	7-32
Data Rules	7-33
Definitions and Declarations	7-34
Definition	7-34
Declaration	7-34
Prepare Code for Built-In Functions	7-35
Overview	7-35
Stubs of <code>stl</code> Functions	7-35
Stubs of <code>libc</code> Functions	7-35
Prepare Multitasking Code	7-38
Polyspace Software Assumptions	7-38
Model Synchronous Tasks	7-39
Model Interruptions and Asynchronous Events and Tasks	7-41
Are Interruptions Maskable or Preemptive?	7-43
Shared Variables	7-45

Mailboxes	7-49
Atomicity (Can Instruction be Interrupted by Another?) ..	7-52
Priorities	7-53
Comment Code for Known Defects	7-54
Comment Syntax for Marking Known Defects	7-58
Syntax Examples: Runtime Errors	7-61
Syntax Examples: Coding Rule Violations	7-61
Check Acronyms for Code Comments	7-62
Types Promotion	7-64
Unsigned Integers Promoted to Signed Integers	7-64
Promotions Rules in Operators	7-65
Example	7-65
Ignoring Assembly Code	7-67
Ignoring Assembly Code — Overview	7-67
When to Ignore Assembly Code	7-67
Automatic Stubbing of Single Function	7-70
Automatic Stubbing of List of Functions	7-70
Directives #asm and #endasm	7-72
If Verification Fails to Parse asm Code	7-72
Local Variables in Functions with Assembly Code	7-73
Loss of Precision Using memset and memcpy	7-75
Avoid memset and memcpy for Structure Initialization ..	7-76

Running a Verification

8

Types of Verification	8-2
Select Analysis Options Configuration	8-3

Check for Compilation Problems	8-4
Start Local Verification	8-6
Start Remote Verification	8-7
Stop Verification	8-8
Stop Remote Verification	8-8
Stop Local Verification	8-8
Phases of Verification	8-9
Run Verification Unit-by-Unit	8-10
Verify All Modules in Project	8-11
Manage Previous Verifications With Polyspace Metrics	8-12
Manage Remote Verifications	8-15
Monitor Progress of Verification	8-16
Run Verification from Command Line	8-17
Manage Remote Analyses at the Command Line	8-18
Modularization of Large Applications	8-20
Partition Application into Modules	8-21
Choose Number of Modules for Application	8-24
Partition Application Using Batch Command	8-27
Basic Options	8-27
Constrain Module Complexity During Partitioning	8-28
Control Naming of Result Folders	8-30

Troubleshooting Verification Problems

9

View error information when verification stops	9-3
View Error Information in Project Manager	9-3
View Error Information in Log File	9-3
Troubleshoot compiler and linking errors	9-6
Obtain system information for technical support	9-7
Information Required	9-7
How to Obtain Required Information	9-7
Header file location not specified	9-8
Message	9-8
Possible Cause	9-8
Solution	9-8
Polyspace software cannot find the server	9-9
Message	9-9
Possible Cause	9-9
Solution	9-9
Errors due to disk defragmentation and antivirus	
software	9-10
Message	9-10
Possible Cause	9-10
Solution	9-10
Software runs out of memory during report	
generation	9-12
Message	9-12
Possible Cause	9-12
Solution	9-12

Compilation Error Overview	9-13
Running multiple Polyspace processes	9-14
Troubleshoot using preprocessed files	9-15
Preprocessed Files	9-15
Troubleshoot Using Preprocessed Files	9-15
Examples	9-15
Check Compilation Before Verification	9-20
Syntax Error	9-21
Message	9-21
Code Used	9-21
Solution	9-21
Undeclared Identifier	9-22
Message	9-22
Code Used	9-22
Solution	9-22
Unknown Prototype	9-23
Message	9-23
Code Used	9-23
Solution	9-23
No Such File or Folder	9-24
Messages	9-24
Code Used	9-24
Solution	9-24
#error directive	9-25
Message	9-25
Code Used	9-25
Solution	9-25
Class, Array, Struct or Union is Too Large	9-26
Messages	9-26
Code Used	9-26
Solution	9-26

Unsupported Non-ANSI Keywords (C)	9-27
Initialization of Global Variables (C++)	9-29
Double Declarations of Standard Template Library Functions	9-30
Large Static Initializer	9-31
Compilation Messages Described in This Section	9-32
C++ Dialect Issues	9-33
ISO versus Default Dialects	9-33
CFront2 and CFront3 Dialects	9-35
Visual Dialects	9-36
GNU Dialect	9-38
C Link Errors	9-42
Link Error Overview (C)	9-42
Function: Procedure Multiply Defined	9-43
Function: Wrong Argument Type	9-43
Function: Wrong Argument Number	9-44
Function: Wrong Return Type	9-45
Variable: Wrong Type	9-45
Variable: Signed/Unsigned	9-46
Variable: Different Qualifier	9-46
Variable: Array Against Variable	9-47
Variable: Wrong Array Size	9-47
Missing Required Prototype for varargs	9-48
C++ Link Errors	9-49
STL Library C++ Stubbing Errors	9-49
Lib C Stubbing Errors	9-50
Standard Library Function Stubbing Errors	9-53
Conflicts Between Library Functions and Polyspace Stubs	9-53
_polyspace_stdstubs.c Compilation Errors	9-53
Troubleshooting Approaches for Standard Library Function Stubs	9-55

Restart with the -I option	9-55
Replace Automatic Stubbing with Include Files	9-56
Create _polyspace_stdstubs.c File with Required Includes	9-57
Provide .c file Containing Prototype Function	9-58
Ignore _polyspace_stdstubs.c	9-59
Automatic Stubbing Errors	9-60
Three Types of Error Messages	9-60
Unknown Prototype Error	9-60
Parameter -entry-points Error	9-60
Reduce Verification Time	9-62
Factors Affecting Verification Time	9-62
Techniques to Improve Verification Performance	9-62
Tune Polyspace Parameters	9-65
Subdivide Code	9-66
Reduce Procedure Complexity	9-76
Reduce Task Complexity	9-78
Reduce Variable Complexity	9-78
Choose Lower Precision	9-79
Storage of Temporary Files	9-80

Reviewing Verification Results

10

Open Remote Verification Results	10-4
Download Remote Verification Results From Command Line	10-5
Open Unit-by-Unit Verification Results	10-6
Open Local Verification Results	10-7
Search Results in Results Manager	10-8

Set Character Encoding Preferences	10-12
Open Results for Generated Code	10-15
Manually Create the Code Generator Text File	10-15
Review Results Progressively	10-16
Assign Review Status to Result	10-18
Review Methodologies	10-24
Organize Results Using Predefined Methodologies ...	10-26
Organize Results Using Custom Methodologies	10-30
Organize Results Using Filters and Groups	10-34
View Call Sequence for Checks	10-42
View Call Tree for Functions	10-44
View Callers and Callees of a Function	10-45
Navigate Call Tree	10-47
View Access Graph for Global Variables	10-49
Customize Review Status	10-50
Use Range Information in Results Manager	10-55
View Pointer Information in Results Manager	10-60
View Probable Cause for Checks	10-61
Check Colors	10-64
Source Code Colors	10-65

Results Manager Overview	10-66
Results Summary	10-67
Source	10-71
Source	10-71
Dashboard	10-80
Check Details	10-86
Error Call Graph	10-86
Check Review	10-87
Check Review	10-87
Review Statistics	10-87
Call Hierarchy	10-91
Variable Access	10-94
Red Checks	10-102
Gray Checks	10-103
Gray Checks	10-103
Common Causes for Gray Checks	10-103
Orange Checks	10-105
Orange Check Identified as Potential Errors	10-105
Color Sequence of Checks	10-109
Defects from Code Integration	10-113
Defects in Unprotected Shared Data	10-114
Defects Related to Pointers	10-115
Messages on Dereferences	10-115
Variables in Structures (C)	10-117

Global Variables	10-118
Initializing Global Variables	10-118
Using Global Variables	10-119
Dataflow Verification	10-120
Results Folder	10-121
ALL Subfolder	10-121
Polyspace-Doc Subfolder	10-122
Polyspace-Instrumented Subfolder	10-123
Reusing Review Comments	10-124
Import Review Comments from Previous	
Verifications	10-125
Import Comments from Previous Verifications	10-125
Automatically Import Comments from Last Verification ..	10-125
Automatically Import Comments During Command-Line	
Verification	10-126
View Checks and Comments Report	10-127
Generate Report After Verification Automatically	10-129
Generate Report After Verification Manually	10-130
Generate Report from Command Line	10-132
-template <i>path</i>	10-132
-format <i>type</i>	10-132
-help or -h	10-132
-output-name <i>filename</i>	10-132
-results-dir <i>folder_paths</i>	10-132
Open Verification Report	10-134
Customize Verification Report	10-135

What is an Orange Check?	11-3
Sources of Orange Checks	11-7
Orange Checks from Code	11-7
Orange Checks from Verification Limitations	11-9
Do I Have Too Many Orange Checks?	11-12
Manage Orange Checks	11-13
Overview: Reducing Orange Checks	11-14
Apply Coding Rules to Reduce Orange Checks	11-15
Use Generated Code	11-16
Improve Verification Precision	11-17
Specify Multitasking Behavior	11-21
Effects of Application Code Size	11-22
Overview: Reviewing Orange Checks	11-23
Define Your Review Methodology	11-24
Perform Selective Review	11-25
View Sources of Orange Checks	11-28
Prioritize Orange Check Review	11-30
Refine Data Range Specifications	11-33

Perform Exhaustive Review	11-37
Exhaustive Orange Review Methodology	11-37
Inconclusive Verification and Code Complexity	11-38
Resolving Orange Checks Caused by Imprecise Approximation	11-39
 Automatic Orange Tester Overview	 11-40
 How the Automatic Orange Tester Works	 11-42
Limitations of Dynamic Testing	11-42
 Select the Automatic Orange Tester	 11-44
 Start the Automatic Orange Tester Manually	 11-45
 Review Test Results After Manual Run	 11-48
AOT Results	11-48
Results	11-48
Log	11-49
 Refine Data Ranges with Automatic Orange Tester ...	 11-50
 Save and Reuse Your Configuration	 11-53
 Export Data Ranges for Polyspace Verification	 11-54
 Polyspace-Instrumented Folder	 11-55
 Technical Limitations	 11-56
Unsupported Polyspace Options	11-56
Options with Restrictions	11-56
Unsupported C Routines	11-56

Rule Checking	12-2
Custom Naming Convention Rules	12-3
Polyspace MISRA C and MISRA AC AGC Checkers ...	12-10
Software Quality Objective Subsets (C)	12-11
Rules in SQ0-Subset1	12-11
Rules in SQ0-Subset2	12-13
Software Quality Objective Subsets (AC AGC)	12-16
Rules in SQ0-Subset1	12-16
Rules in SQ0-Subset2	12-16
MISRA C:2004 Coding Rules	12-18
Supported MISRA C:2004 Rules	12-18
MISRA C:2004 Rules Not Checked	12-56
Polyspace MISRA C++ Checker	12-59
Software Quality Objective Subsets (C++)	12-60
SQO Subset 1 – Direct Impact on Selectivity	12-60
SQO Subset 2 – Indirect Impact on Selectivity	12-63
MISRA C++ Coding Rules	12-69
Supported MISRA C++ Coding Rules	12-69
MISRA C++ Rules Not Checked	12-89
Polyspace JSF C++ Checker	12-95
JSF C++ Coding Rules	12-96
Supported JSF C++ Coding Rules	12-96
JSF++ Rules Not Checked	12-121

13

Activate Coding Rules Checker 13-2

Select Specific MISRA or JSF Coding Rules 13-6

Create Custom Coding Rules 13-8

Format of Custom Coding Rules File 13-10

Exclude Files from Rules Checking 13-12

Allow Custom Pragma Directives 13-13

Specify Boolean Types 13-14

Review Coding Rule Violations 13-15

Apply Coding Rule Violation Filters 13-17

Generate Coding Rules Report 13-18

Software Quality with Polyspace Metrics

14

Software Quality with Polyspace Metrics 14-3

Set Up Verification to Generate Metrics 14-5

 Specify Automatic Verification 14-5

Open Polyspace Metrics 14-12

Organize Polyspace Metrics Projects 14-14

Protect Access to Project Metrics	14-16
Web Browser Support	14-18
What You Can Do with Polyspace Metrics	14-19
Review Overall Progress	14-20
Display Metrics for Single Project Version	14-24
Create File Module and Specify Quality Level	14-25
Compare Project Versions	14-27
Review New Findings	14-28
Review Results	14-29
Save Review Comments	14-31
Fix Defects	14-32
Review Code Complexity	14-33
Customize Software Quality Objectives	14-34
SQO Levels	14-36
SQO Level 1	14-36
SQO Level 2	14-41
SQO Level 3	14-41
SQO Level 4	14-41
SQO Level 5	14-42
SQO Level 6	14-42
SQO Exhaustive	14-42
Coding Rules Sets	14-44
Coding Rules Set 1	14-44

Coding Rules Set 2	14-46
Run-Time Checks Sets	14-48
Run-Time Checks Set 1	14-48
Run-Time Checks Set 2	14-49
Run-Time Checks Set 3	14-50
Status Acronyms	14-52
Code Metrics	14-53
Run-Time Checks	14-61
Number of Lines of Code Calculation	14-63
Administer Results Repository	14-64
Administer Repository Through Web Browser	14-64
Administer Repository From Command Line	14-65
Backup Results Repository	14-66

Configure Model for Code Analysis

15

Model Configuration for Code Generation and Analysis	15-2
Configure Simulink Model	15-3
Recommended Model Settings for Code Analysis	15-5
Check Simulink Model Settings	15-7
Check Simulink Model Settings Before Code Generation	15-8

Check Simulink Model Settings Before Analysis	15-10
Annotate Blocks for Known Errors or Coding-Rule Violations	15-12

Model Link for Polyspace Code Prover

16

Install Polyspace Plug-In for Simulink	16-2
Specify Signal Ranges	16-3
Specify Signal Range through Source Block Parameters ..	16-3
Specify Signal Range through Base Workspace	16-5
Annotate Code to Justify Polyspace Checks	16-8
Configure Data Range Settings	16-10
Main Generation for Model Verification	16-13
Embedded Coder Considerations	16-15
Subsystems	16-15
Default Options	16-15
Data Range Specification	16-16
Recommended Polyspace options for Verifying Generated Code	16-16
Hardware Mapping Between Simulink and Polyspace	16-21
TargetLink Considerations	16-22
TargetLink Support	16-22
Subsystems	16-22
Default Options	16-22
Data Range Specification	16-23
Lookup Tables	16-24
Code Generation Options	16-24
Generate and Verify Code with Configured Model	16-25

View Results in Polyspace Code Prover	16-27
Identify Errors in Simulink Models	16-29

Configure Code Analysis Options

17

Polyspace Configuration for Generated Code	17-2
Include Handwritten Code	17-3
Specify Remote Analysis	17-5
Configure Analysis Depth for Referenced Models	17-6
Specify Location of Results	17-7
Check Coding Rules Compliance	17-8
Configure Polyspace Options from Simulink	17-10
Configure Polyspace Project Properties	17-11
Create a Polyspace Configuration File Template	17-12
Specify Header Files for Target Compiler	17-15
Open Polyspace Results Automatically	17-16
Remove Polyspace Options From Simulink Model	17-17

Run Polyspace on Generated Code

18

Specify Type of Analysis to Perform	18-2
Run Analysis for Embedded Coder	18-5
Run Analysis for TargetLink	18-7
Monitor Progress	18-8
Local Analyses	18-8
Remote Batch Analyses	18-8

Using Polyspace Software in the Eclipse IDE

19

Install Polyspace Plug-In for Eclipse	19-2
Install Polyspace Plug-In for Eclipse IDE	19-2
Uninstall Polyspace Plug-In for Eclipse IDE	19-4
Verify Code in the Eclipse IDE	19-5
Code Verification in the Eclipse IDE	19-5
Create an Eclipse Project	19-5
Set Up Polyspace Verification with Eclipse Editor	19-6
Start Verification from Eclipse Editor	19-7
Review Verification Results from Eclipse Editor	19-7

Using Polyspace Software in Visual Studio

20

Install Polyspace Add-In for Visual Studio	20-2
Install Polyspace Add-In for Visual Studio	20-2
Uninstall Polyspace Add-In for Visual Studio	20-3

Verify Code in Visual Studio	20-4
Code Verification in Visual Studio	20-4
Create Visual Studio Project	20-4
Verify Code in Visual Studio	20-6
Monitor Verification in Visual Studio	20-14
Review Verification Results in Visual Studio	20-16

Glossary

Introduction to Polyspace Products

- “Polyspace® Code Prover™ Product Description” on page 1-2
- “Polyspace Verification” on page 1-3
- “How Polyspace Verification Works” on page 1-6
- “Related Products” on page 1-8
- “Tool Qualification and Certification” on page 1-9

Polyspace Code Prover Product Description

Prove the absence of run-time errors in software

Polyspace® Code Prover™ proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in C and C++ source code. It produces results without requiring program execution, code instrumentation, or test cases. Polyspace Code Prover uses static analysis and abstract interpretation based on formal methods. You can use it on handwritten code, generated code, or a combination of the two. Each operation is color-coded to indicate whether it is free of run-time errors, proven to fail, unreachable, or unproven.

Polyspace Code Prover also displays range information for variables and function return values, and can prove which variables exceed specified range limits. Results can be published to a dashboard to track quality metrics and ensure conformance with software quality objectives. Polyspace Code Prover can be integrated into build systems for automated verification.

Support for industry standards is available through IEC Certification Kit (for IEC 61508 and ISO 26262) and DO Qualification Kit (for DO-178).

Key Features

- Proven absence of certain run-time errors in C and C++ code
- Color-coding of run-time errors directly in code
- Calculation of range information for variables and function return values
- Identification of variables that exceed specified range limits
- Quality metrics for tracking conformance with software quality objectives
- Web-based dashboard providing code metrics and quality status
- Guided review-checking process for classifying results and run-time error status
- Graphical display of variable reads and writes

Polyspace Verification

In this section...
“Polyspace Verification” on page 1-3
“Value of Polyspace Verification” on page 1-3

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 1-4
- “Decrease Development Time” on page 1-4
- “Improve the Development Process” on page 1-5

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.

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

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.

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 Products for Verifying Ada Code” on page 1-8
“Polyspace® Bug Finder™” on page 1-8

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/>

Polyspace Bug Finder

For information about Polyspace Bug Finder™, see

[http://www.mathworks.com/products/polyspace-bug-finder/.](http://www.mathworks.com/products/polyspace-bug-finder/)

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).

How to Use Polyspace Software

- “Polyspace Verification and the Software Development Cycle” on page 2-2
- “Analysis and Review Criteria” on page 2-4
- “Implement Process for Verification” on page 2-9
- “Sample Workflows for Polyspace Verification” on page 2-13

Polyspace Verification and the Software Development Cycle

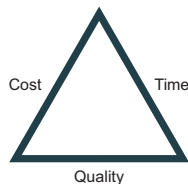
In this section...

“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.

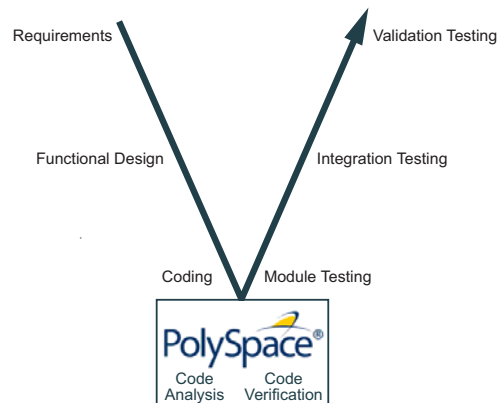
Polyspace verification allows a different process. Polyspace verification can support both productivity improvement and quality improvement at the same time. However, you must balance the aims of these activities.

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.

Analysis and Review Criteria

In this section...
“Choose Robustness or Contextual Verification” on page 2-4
“Choose Coding Rules” on page 2-5
“Choose Strict or Permissive Verification” on page 2-6
“Define Software Quality Levels” on page 2-7

Choose Robustness or Contextual Verification

Before using Polyspace products to verify your code, you must decide what type of software verification you want to perform. There are two approaches to code verification that result in slightly different workflows:

- **Robustness Verification** – Prove that the software does not generate run-time errors for all verification conditions.
- **Contextual Verification** – Prove that the software does not generate run-time errors under normal working conditions.

Note Some verification processes may incorporate both robustness and contextual verification. For example, developers may perform robustness verification on individual files early in the development cycle, while writing the code. Later, the team may perform contextual verification on larger software components.

Robustness Verification

Robustness verification proves that the software does not generate run-time errors under all verification conditions, including “abnormal” conditions for which it was not designed. This can be thought of as “worst case” verification.

By default, Polyspace software assumes you want to perform robustness verification. In a robustness verification, Polyspace software:

- Assumes function inputs are full range

- Initializes global variables to full range
- Automatically stubs missing functions

Although this approach checks the software for all conditions, it can lead to *orange checks* (unproven code) in your results. You must then manually inspect these orange checks in accordance with your software quality goals.

Contextual Verification

Contextual verification proves that the software 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.

When performing contextual verification, you use Polyspace options to reduce the number of orange checks. For example, you can:

- Use Data Range Specifications (DRS) to specify the ranges for your variables, thereby limiting the verification to these cases. For more information, see “Data Range Specifications (DRS)” on page 6-55.
- Create a detailed main program to model the call sequence, instead of using the default main generator. For more information, see “Verify C Application Without a “Main”” on page 6-33.
- Provide manual stubs that emulate the behavior of missing functions, instead of using the default automatic stubs. For more information, see “When to Provide Function Stubs” on page 7-4.

Choose Coding Rules

Coding rules are one of the most efficient means to improve both the quality of your code, and the quality of your verification results.

If your development team observes certain coding rules, the number of orange checks (unproven code) in your verification results will be reduced substantially. This means that there is less to review, and that the remaining checks are more likely to represent actual bugs. This can make the cost of bug detection much lower.

Polyspace software can check that your code complies with specified coding rules. Before starting code verification, you should consider implementing coding rules, and choose which rules to enforce.

For more information, see “Activate Coding Rules Checker” on page 13-2.

Choose Strict or Permissive Verification

While defining the quality goals for your application, you should determine which of these options you want to use.

Options that make verification more strict include:

- **Detect overflows on signed and unsigned (-scalar-overflow-checks)** — Verification is more strict with overflowing computations on unsigned integers.
- **Do not consider all global variables to be initialized (-no-def-init-glob)** — Verification treats global variables as non-initialized. If a global variable is read before it is written to, the verification generates a red check.
- **-Wall** — Specifies that C compliance warnings are written to the log file during compilation.
- **-strict** — Specifies strict verification mode, which is equivalent to using the `-Wall` and `-no-automatic-stubbing` options simultaneously.

Options that make verification more permissive include:

- **Dialect (-dialect)** — Verification allows syntax associated with the IAR and Keil dialects.
- **Ignore overflowing computations on constants (-ignore-constant-overflows)** — Verification is permissive with overflowing computations on constants.
- **Allow negative operand for left shifts (-allow-negative-operand-in-shift)** — Verification allows a shift operation on a negative number.

- **Enable pointer arithmetic across fields**
(-allow-ptr-arith-on-struct) — Enables navigation within a structure or union from one field to another.

For more information on these options, see “Analysis Options for C Code”.

Define Software Quality Levels

The software quality level you define determines which Polyspace options you use, and which results you must review.

You define the quality levels for your application, from level QL-1 (lowest) to level QL-4 (highest). Each quality level consists of a set of software quality criteria that represent a certain quality threshold. For example:

Software Quality Levels

Criteria	Software Quality Levels			
	QL1	QL2	QL3	QL4
Document static information	X	X	X	X
Enforce coding rules with direct impact on selectivity	X	X	X	X
Review all red checks	X	X	X	X
Review all gray checks	X	X	X	X
Review first criteria level for orange checks		X	X	X
Review second criteria level for orange checks			X	X
Enforce coding rules with indirect impact on selectivity			X	X
Perform dataflow analysis			X	X
Review third criteria level for orange checks				X

In the example above, the quality criteria include:

- **Static Information** – Includes information about the application architecture, the structure of each module, and all files. Full verification of your application requires the documentation of static information.
- **Coding rules** – Polyspace software can check that your code complies with specified coding rules. The section “Apply Coding Rules to Reduce Orange Checks” on page 11-15 defines two sets of coding rules – a first set with direct impact on the selectivity of the verification, and a second set with indirect impact on selectivity.
- **Red checks** – Represent errors that occur every time the code is executed.
- **Gray checks** – Represent unreachable code.
- **Orange checks** – Indicate unproven code, meaning a run-time error may occur.
- **Dataflow analysis** – Identifies errors such as non-initialized variables and variables that are written but not read. This can include inspection of:
 - Application call tree
 - Read/write accesses to global variables
 - Shared variables and their associated concurrent access protection

Implement Process for Verification

In this section...
“Overview of the Polyspace Process” on page 2-9
“Define Process to Meet Your Goals” on page 2-11
“Apply Process to Assess Code Quality” on page 2-12
“Improve Your Verification Process” on page 2-12

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 3-10.
- Setting review criteria in the Results Manager perspective for consistent review of results. For more information, see “Organize Results Using Custom Methodologies” on page 10-30.

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 “Assign Review Status to Result” on page 10-18.
- 3** Saving review comments for each component, so they are available for future review. See “Import Review Comments from Previous Verifications” on page 10-125.
- 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 “Orange Check Management”.

Sample Workflows for Polyspace Verification

In this section...

“Overview of Verification Workflows” on page 2-13

“Software Developers and Testers – Standard Development Process” on page 2-14

“Software Developers and Testers – Rigorous Development Process” on page 2-17

“Quality Engineers – Code Acceptance Criteria” on page 2-21

“Quality Engineers – Certification/Qualification” on page 2-24

“Model-Based Design Users — Verifying Generated Code” on page 2-25

“Project Managers — Integrating Polyspace Verification with Configuration Management Tools” on page 2-29

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.

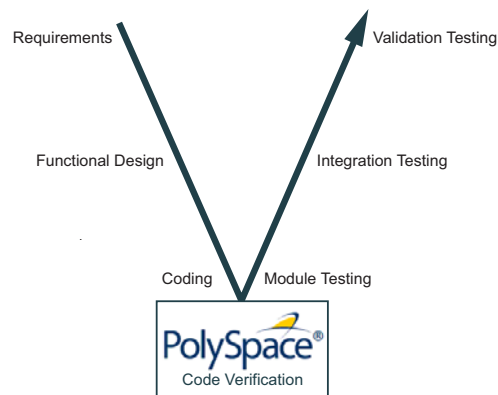
Quality

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 than 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.

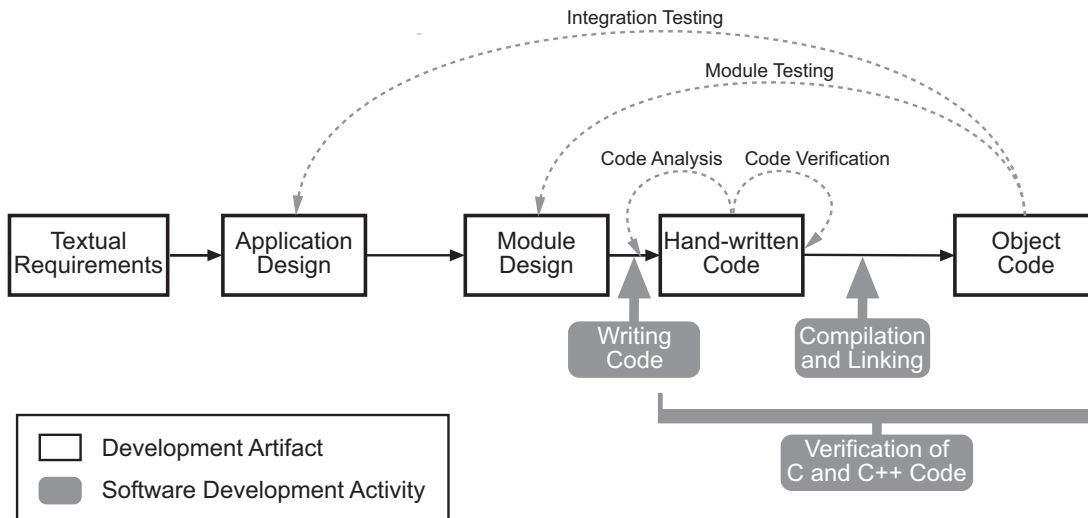
Quality

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:

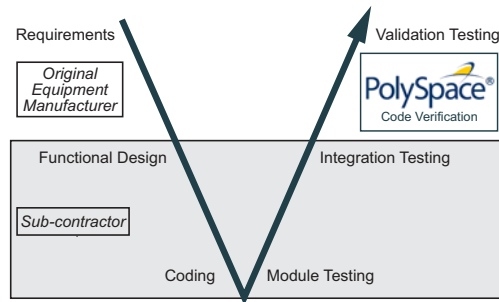
- You may be satisfied with zero red checks.
- In addition to zero red checks, you may want to conduct an exhaustive orange check review.

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 “Define Software Quality Levels” on page 2-7.

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 “Analysis and Review Criteria” on page 2-4.
- 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 2-11.
- 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 Software Quality Levels” on page 2-7).

- 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-178C qualification.

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

For information on using Polyspace products within an IEC 61508 certification environment, see the *IEC Certification Kit: Verification of C and C++ Code Using Polyspace Products*.

For information on using Polyspace products within an DO-178C qualification environment, see the *DO Qualification Kit: Polyspace Client/Server for C/C++ Tool Qualification Plan*.

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

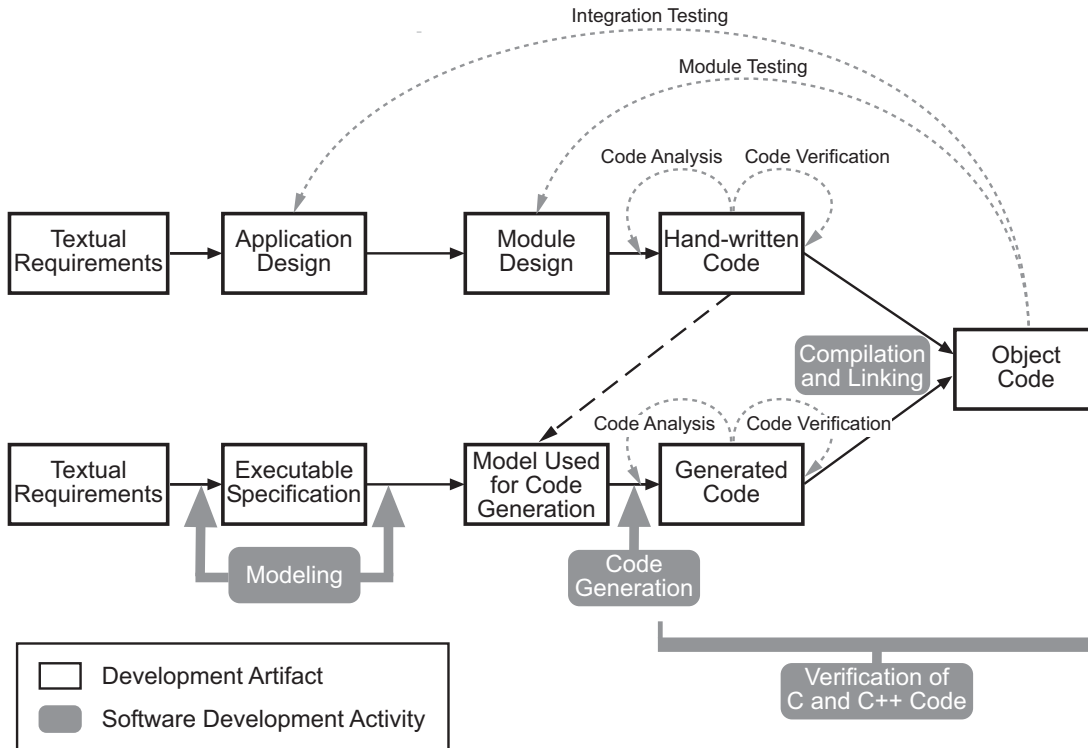
The goal of Polyspace verification is to improve the quality of the software by identifying implementation issues in the code, and proving that the code is both semantically and logically correct.

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 Verification and Validation™ 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	<ul style="list-style-type: none">• Incorrect Scaling• Unknown calibrations• Untested data ranges	<ul style="list-style-type: none">• Overflows/Underflows• Division by zero• Square root of negative numbers
Memory corruption	<ul style="list-style-type: none">• Incorrect array specification in state machines• Incorrect legacy code (look-up tables)	<ul style="list-style-type: none">• Out of bound array indexes• Pointer arithmetic
Data truncation	<ul style="list-style-type: none">• Unexpected data flow	<ul style="list-style-type: none">• Overflows/Underflows• Wrap-around
Logic errors	<ul style="list-style-type: none">• Unreachable states• Incorrect Transitions	<ul style="list-style-type: none">• 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”

Setting Up Project: Basic Steps

- “What is a Project?” on page 3-2
- “What is a Project Template?” on page 3-3
- “Create New Project” on page 3-4
- “Add Source Files and Include Folders” on page 3-6
- “Specify Results Folder” on page 3-8
- “Specify Analysis Options” on page 3-10
- “Save Analysis Options as Project Template” on page 3-13
- “Specify Text Editor” on page 3-16

What is a Project?

In Polyspace software, a project is a named set of parameters for your software project's source files. A project includes:

- Source files
- Include folders
- One or more configurations, specifying a set of analysis options
- One or more modules, each of which include:
 - Source (specific set of source files)
 - Configuration (specific set of analysis options)
 - Results

Use the Project Manager perspective to create and modify a project.

What is a Project Template?

A **Project Template** is a predefined set of analysis options for a specific compilation environment. When creating a new project, you have the option to:

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

Polyspace software provides predefined templates for common compilers such as IAR, Kiel, and VxWorks Aonix, Rational, and Greenhills. For additional templates, see [Polyspace Compiler Templates](#) .

- Set analysis options manually. You can save your options to a custom template and reuse them later. For more information, see “[Save Analysis Options as Project Template](#)” on page 3-13 .

Create New Project

This example shows how to create a new project in Polyspace Code Prover. Before you create a project, you must know:

- Location of source files
- Location of include files
- Location where verification results will be stored

For these locations, it is convenient to create three subfolders under a common project folder. For instance, under the folder `polyspace_project`, you can create `sources`, `includes` and `results`.

1 Select **File > New Project...**

2 In the Project – Properties dialog box, enter the following information:

- **Project name**
- **Location:** Folder where you will store the project file with extension `.psprj`. You can use this file to open an existing project.

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

- **Project language**

If you want to use a template, select the **Use template** check box. Then, click **Next**.

3 Select the template for your compiler. If your compiler does not appear in the list of predefined templates, select **Baseline**. You can then start with a generic template. Click **Next**.

4 Add source files and include folders to your project.

- Navigate to the location where you stored your source files. Select the source files for your project. Click **Add Source Files**.

- The software automatically adds the standard include files to your project. To use custom include files, navigate to the folder containing your include files. Click **Add Include Folders**.

5 Click **Finish**.

The new project opens in the **Project Browser**.

6 Save the project. Select **File > Save** or enter **Ctrl+S**.

Related Examples

- “Add Source Files and Include Folders” on page 3-6

Concepts


- “What is a Project?” on page 3-2

Add Source Files and Include Folders

This example shows how to add source files and include folders to an existing project.

Add Sources and Includes

1 In the **Project Browser**, select your project.

2 Click the **Add source** icon .

3 Add source files and include folders to your project.

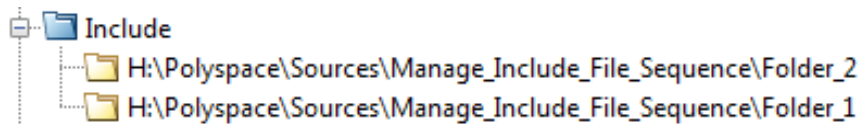
- Navigate to the location where you stored your source files. Select the source files for your project. Click **Add Source Files**.
- The software adds standard include files to your project. To use custom include files, navigate to the folder containing your include files. Click **Add Include Folders**.

4 Click **Finish**.



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, it is convenient to change the order of include folders instead of reorganizing the contents of your folders. For a particular include file name, the software includes the file in the first include folder under **Project_Name > Include**.

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



To change the order of include folders:

- 1 In the **Project Browser**, expand the **Include** folder.
- 2 Select the include folder that you want to move.
- 3 To move the folder, click either  or  on the Project Browser toolbar.

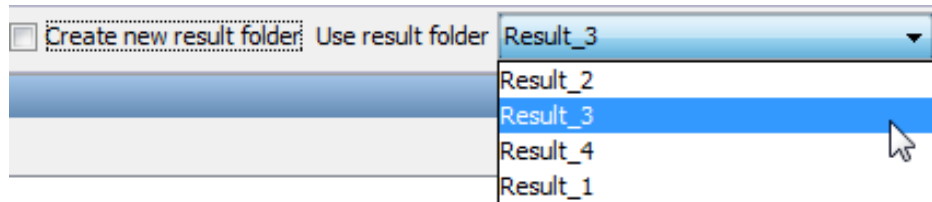
Related Examples

- “Specify Results Folder” on page 3-8
- “Create New Project” on page 3-4

Specify Results Folder

This example shows how to specify a results folder. By default, the software creates a new results folder for each analysis. Before starting an analysis, you can choose to overwrite an existing results folder. For example, if you stopped an analysis before completion and want to restart it, you can overwrite a results folder.

- To create a new folder, in the Project Manager toolbar, select the **Create new result folder** box.
 - By default, the new folder is created in *Project_folder / Module_name*. *Project_folder* is the project location you specified when creating a new project.
 - You can also create a parent folder for storing your results. Select **Options > 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 will be created under this parent folder.
- To overwrite an existing folder that is open in the **Project Browser**, clear the **Create new result folder** box. In the **Overwrite result folder** drop-down list, select the folder that you want to use.



- To overwrite an existing folder not open in the **Project Browser**, right-click **Results**. Select **Choose a Result folder**. Select the folder where you want your results stored.
- To specify a results folder from the command line, use the `-results-dir` option, followed by the full path to the folder inside " ".

When you start the verification, the software saves the results in the specified folder.

See Also `-results-dir`

Related Examples

- “Customize Results Folder Location and Name” on page 4-9

Specify Analysis Options

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

In this section...

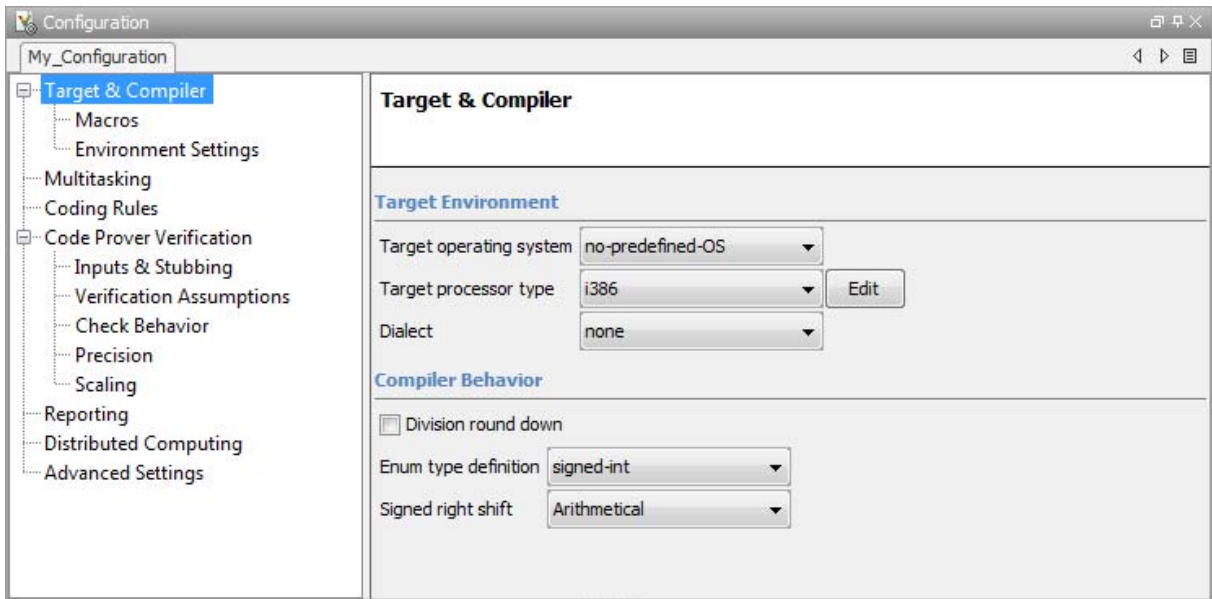
“Specify Options in User Interface” on page 3-10

“Specify Options from DOS and UNIX Command Line” on page 3-11

“Specify Options from MATLAB Command Line” on page 3-11

Specify Options in User Interface

In the Polyspace Project Manager perspective, use the **Configuration** pane.



For instance:

- To specify the target processor, select **Target & Compiler** in the **Configuration** tree view. Select your processor from the **Target processor type** drop-down list.
- To specify verification precision, select **Code Prover Verification > Precision**. Select a number from the **Precision level** drop-down list.

Specify Options from DOS and UNIX Command Line

At the DOS or UNIX[®] command-line, append 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 `-0` 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 -02
```

Specify Options from MATLAB Command Line

At the MATLAB[®] command-line, enter analysis options and their values as string arguments to the `polyspaceCodeProver` function. For instance:

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

```
polyspaceCodeProver('-sources','file.c','-lang','c','-target','m68k')
```

- To specify verification precision, use the `-0` option. For instance, to set precision level to 2 for your source file `file.c`, enter:

```
polyspaceCodeProver('-sources','file.c','-lang','c','-02')
```

See Also `polyspaceCodeProver`

Related Examples

- “Save Analysis Options as Project Template” on page 3-13

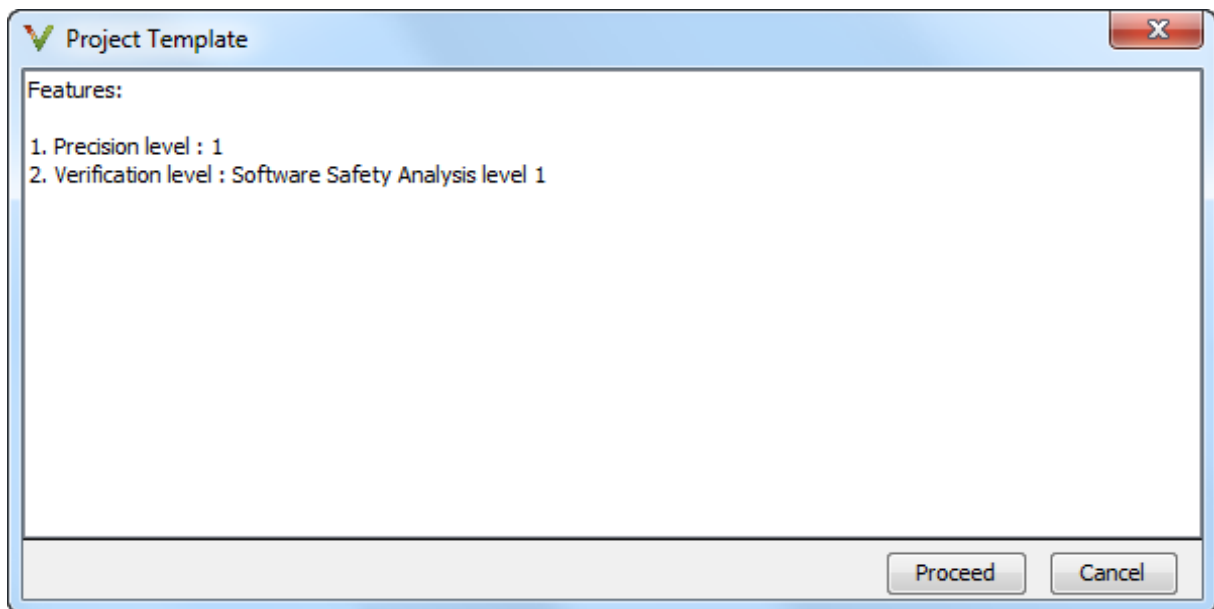
Concepts

- “Analysis Options for C Code”
- “Analysis Options for C++ Code”

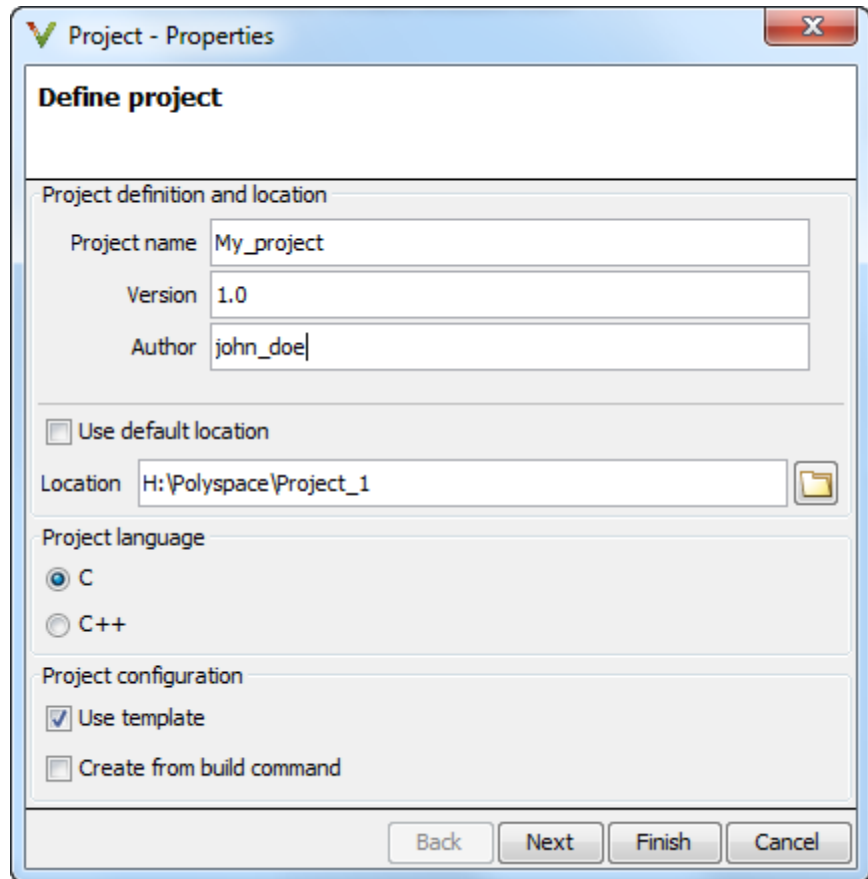
Save Analysis Options as Project Template


This example shows how to save analysis options for use in other projects. Once you have configured analysis options for a project, you can save the configuration as a **Project Template**. You can use this saved configuration to automatically set up analysis options for other projects.

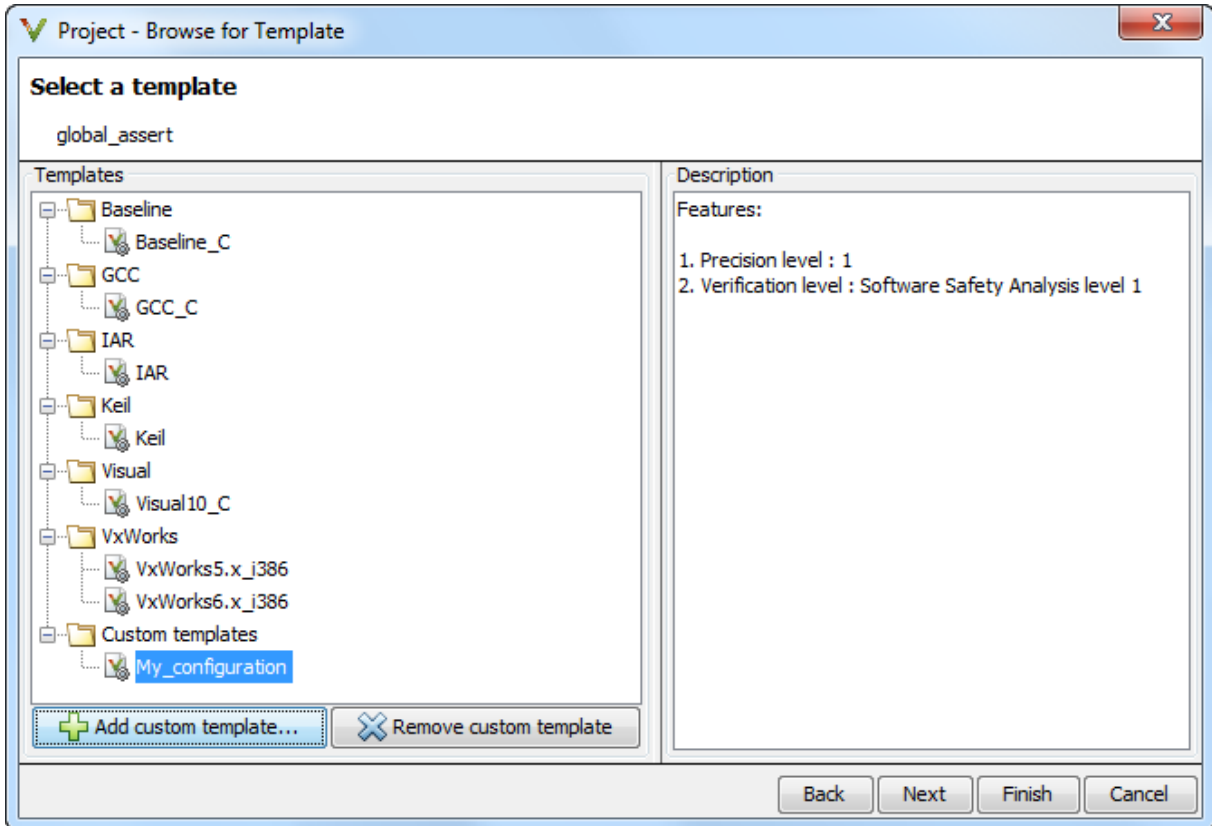
- To create a **Project Template** from an open project:
 - 1 Right-click the configuration that you want to use, and then select **Save As Template**.
 - 2 Enter a description for the template, then click **Proceed**. Save your Template file.



- When you create a new project, to use a saved template:
 - 1 Under **Project configuration**, check the **Use template** box. Click **Next**.



- 2 Select . 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.



Related Examples

- “Specify Analysis Options” on page 3-10

Concepts

- “Analysis Options for C Code”
- “Analysis Options for C++ Code”

Specify Text Editor

This example shows how to change the default text editor for opening source files from the Polyspace interface. Polyspace uses WordPad as the default editor in Windows®. It uses vi as the default editor in Linux®.

- 1** Select **Options > Preferences**.
- 2** On the Polyspace Preferences dialog box, select the **Editors** tab.
- 3** In the **Text editor** field, specify the path to your text editor. For example:

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

- 4** 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 Summary** pane and select **Open Source File**, 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.

- 5** Click **OK**.

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

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

Setting Up Project : Advanced Steps

- “Create Projects Automatically from Your Build System” on page 4-2
- “Requirements for Project Creation from Build Systems” on page 4-6
- “Create Multiple Modules” on page 4-7
- “Create Multiple Analysis Option Configurations” on page 4-8
- “Customize Results Folder Location and Name” on page 4-9
- “Specify Functions Not Called by Generated Main” on page 4-10
- “Specify Options to Match Your Quality Goals” on page 4-11
- “Set Up Project to Check Coding Rules” on page 4-16
- “Set Up Project to Automatically Test Orange” on page 4-19

Create Projects Automatically from Your Build System

In this section...
“Create Project in User Interface” on page 4-2
“Create Project from DOS and UNIX Command Line” on page 4-3
“Create Project from MATLAB Command Line” on page 4-4

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.

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.

Create Project in User Interface

- 1** Select **File > New Project**.
- 2** On the Project – Properties dialog box, 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	Specify: <ul style="list-style-type: none"> Name of your build automation script. Example:make -B Full path to an executable such as Visual Studio®. Example:"C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\VCExpress.exe".
Specify working directory for running build command	Specify the directory from which you run your build automation script.
Add advanced configuration options	Specify additional options for advanced tasks such as incremental build. For the full list of options, see the <code>-options</code> value argument for <code>polyspaceConfigure</code> .

4 Click .

- 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.

5 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**.

Create Project from DOS and UNIX Command Line

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 -B targetName buildOptions
```

- Create an options file. You can then use the options file to run verification 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 -B targetName buildOptions
```

Use the options file to run verification:

```
polyspace-code-prover-nodesktop -options-file myOptions
```

For more information on advanced options for `polyspace-configure`, see the `-options` value argument for `polyspaceConfigure`.

Create Project from MATLAB Command Line

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 -B targetName buildOptions
```

- Create an options file. You can then use the options file to run verification 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 -B targetName buildOptions
```

Use the options file to run verification:

```
polyspaceCodeProver -options-file myOptions
```

For more information, see `polyspaceConfigure`.

Related Examples

- “Trace Visual Studio Build” on page 5-3

Concepts

- “Requirements for Project Creation from Build Systems” on page 4-6

Requirements for Project Creation from Build Systems

For `polyspaceConfigure` to correctly trace your build and gather all your source files:

- Your compiler must be called locally for a clean build.
- Your compiler configuration must be available to Polyspace. The compilers currently supported are:
 - Visual C++[®] compiler
 - gcc
 - clang

If your compiler does not meet these requirements, try the following:

- 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` option to force a clean build.
- If your compiler configuration is not available to Polyspace:
 - Write a compiler configuration file in a specific format. Use the option `-compiler-configuration configurationFileName` to provide the configuration file `configurationFileName`. You can find existing configuration files in `matlabroot\polyspace\configure\compiler_configuration\`.
 - Contact MathWorks Technical Support. For more information, see “Obtain system information for technical support” on page 9-7.
- If you use a compiler cache such as `ccache` or a distributed build system such as `distmake`, `polyspaceconfigure` cannot trace your build. You must deactivate them.

See Also `polyspaceConfigure`

Related Examples

- “Create Projects Automatically from Your Build System” on page 4-2


Create Multiple Modules

A Polyspace Code Prover project can contain multiple modules. With each of these modules, you can analyze a specific set of source files using a specific set of analysis options.

When you create a module, the software creates a project configuration with default option values. You can modify these values. In addition, you can create multiple configurations in each module, allowing you to change analysis options each time you run an analysis.

To create a new module in your project:

1 In the Project Browser, select your project.

2 On the Project Browser toolbar, click .

You see a second module, `Module_2`, in the Project Browser tree.

3 In the project **Source** folder, right-click the files that you want to add to the module. From the context menu, select **Copy Source File to > Module_2**.

The software displays these files in the **Source** folder of `Module_2`.

If you have twenty or more modules in your project, when you select **Copy Source File to**, the Select Modules dialog box opens. From the module list, choose the required modules. Then click **Select**.

Note You can also drag source files from a project into the Source folder of a module.

Create Multiple Analysis Option Configurations

Each Polyspace project can contain multiple *configurations*. Each of these configurations specifies a specific set of analysis options. Using multiple configurations allows you to analyze a set of source files multiple times using different analysis options for each run.

To create a new configuration in your project:

- 1** In the **Project Browser**, select a module.
- 2** Right-click the **Configuration** folder in the module. From the context menu, select **Create New Configuration**.

In the **Project Browser**, the software displays a new configuration *project_name_1*. To rename the configuration, double-click it.

- 3** On the **Configuration** pane, specify the analysis options for the new configuration.
- 4** To save your project with the new settings, select **File > Save**.
- 5** After verification, to see the configuration you used, right-click your results on the **Project Browser**. From the context menu, select **Open Configuration**.

If you are viewing the results in the Results Manager, to see the configuration you used, click the  icon on the Results Manager toolbar.

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

Concepts

- “Analysis Options for C Code”

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 From the Polyspace toolbar, select **Options > 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 require a subfolder, select the **Add a subfolder using the project name** check box. 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*.

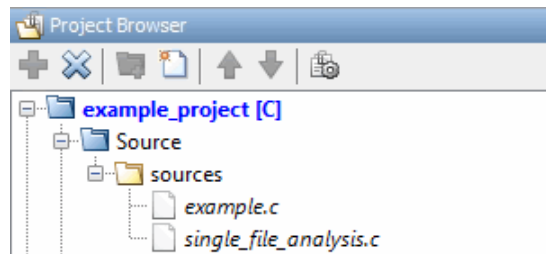
Specify Functions Not Called by Generated Main

If you select the option **Verify module**, Polyspace generates a main function if one does not exist. You can choose which functions are called by the generate main.

For instance, you can specify source files that the main generator ignores. Functions defined in these source files are not called by the generated main and not verified unless they are called elsewhere in your code.

- 1 In the **Project Browser**, expand the **Source** tree to find the source file you want the main generator to ignore.
- 2 Right click the file, and select **Define As > Not Called by Generated Main**.

The files ignored by the main generator appear in italics in the **Source** tree of the project.



- 3 To undo your action, right-click the file again. Select **Define As > Called by Generated Main**.

Specify Options to Match Your Quality Goals

In this section...

“Choose Contextual Verification Options for C Code” on page 4-11

“Choose Contextual Verification Options for C++ Code” on page 4-12

“Choose Strict or Permissive Verification Options” on page 4-14

Choose Contextual Verification Options for C Code

Polyspace software performs robustness verification by default. If you want to perform contextual verification, there are several options you can use to provide context for data ranges, function call sequence, and stubbing.

For more information on robustness and contextual verification, see “Choose Robustness or Contextual Verification” on page 2-4.

Note If you are aware of run-time errors in your code but still want to run a verification, you can annotate your code so that these known errors are highlighted in the Results Manager perspective. For more information, see “Comment Code for Known Defects” on page 7-54.

To specify contextual verification for your project:

- 1** In the Project Manager perspective, on the **Configuration > Code Prover Verification** pane, select **Verify module**.
- 2** To set ranges on variables, use the following options:
 - **Code Prover Verification > Variables to initialize** — Specifies how the generated main initializes global variables.
With cyclic systems, for example, code generated from Simulink models, you configure **Calibration variables** and **Input variables**.
 - **Code Prover Verification > Inputs & Stubbing > Variable/function range setup** — Activates the DRS

option, allowing you to set specific data ranges for a list of global variables.

3 To specify function call sequence, use the following options:

- **Code Prover Verification > Initialization functions** (-functions-called-before-main) — Specifies an initialization function called after initialization of global variables but before the main loop.
- **Code Prover Verification > Functions to call** — Specifies how the generated main calls functions.

With cyclic systems, you configure the options **Initialization functions** (-functions-called-before-loop), **Cyclic functions**, and **Termination functions**.

4 To control stubbing behavior, use the following options:

- **Code Prover Verification > Inputs & Stubbing > No automatic stubbing** — If you select this option, the software does not automatically stub functions. The software list the functions to be stubbed and stops the verification.
- **Code Prover Verification > Inputs & Stubbing > Functions to stub** — Specify the functions that you want Polyspace to stub.

For more information about these options, see “Analysis Options for C Code”.

Choose Contextual Verification Options for C++ Code

Polyspace software performs robustness verification by default. If you want to perform contextual verification, there are several options you can use to provide context for data ranges, function call sequence, and stubbing.

For more information on robustness and contextual verification, see “Choose Robustness or Contextual Verification” on page 2-4.

Note If you are aware of run-time errors in your code but still want to run a verification, you can annotate your code so that these known errors are highlighted in the Results Manager perspective. For more information, see “Comment Code for Known Defects” on page 7-54.

To specify contextual verification for your project:

- 1** In the Project Manager perspective, on the **ConfigurationCode Prover Verification** pane, select **Verify module**.
- 2** To set ranges on variables, use the following options:
 - **Code Prover Verification > Variables to initialize** — Specifies how the generated main initializes global variables.
 - **Code Prover Verification > Inputs & Stubbing > Variable/function range setup** — Activates the DRS option, allowing you to set specific data ranges for a list of global variables.
- 3** To specify function call sequence, use the following options:
 - **Code Prover Verification > Initialization functions (-functions-called-before-main)** — Specifies an initialization function called after initialization of global variables but before the main loop.
 - **Code Prover Verification > Functions to call** — Specifies how the generated main calls functions.
- 4** To control stubbing behavior, use the following options:
 - **Code Prover Verification > Inputs & Stubbing > No automatic stubbing** — If you select this option, the software does not automatically stub functions. The software list the functions to be stubbed and stops the verification.
 - **Code Prover Verification > Inputs & Stubbing > No STL stubs** — Stub Standard Library functions using standard rules instead of using Polyspace implementation.

- **Code Prover Verification > Inputs & Stubbing > Functions to stub** — Specify the functions that you want Polyspace to stub.

For more information on these options, see “Analysis Options for C++ Code” .

Choose Strict or Permissive Verification Options

Polyspace software provides several options that allow you to customize the strictness of the verification. You should set these options to match the quality goals for your application.

Note If you are aware of run-time errors in your code but still want to run a verification, you can annotate your code so that these known errors are highlighted in the Results Manager perspective. For more information, see “Comment Code for Known Defects” on page 7-54.

Use the following options to make verification more strict:

- **Code Prover Verification > Inputs & Stubbing > Ignore default initialization of global variables** — Verification treats global variables as non-initialized, causing a red error if the global variables are read before being written to.
- **Code Prover Verification > Check Behavior > Detect overflows** — Stricter checks for overflowing computations on unsigned integers.

Use the following options to make verification more permissive:

- **Target & Compiler > Dialect** — Verification allows syntax associated with the IAR and Keil dialects.
- **Code Prover Verification > Checks Behavior > Ignore overflowing computations on constants** — Verification is permissive with overflowing computations on constants.
- **Code Prover Verification > Checks Behavior > Allow negative operand for left shifts** — Verification allows a shift operation on a negative number.

- **Code Prover Verification > Checks Behavior > Enable pointer arithmetic across fields** — Enables navigation within a structure or union from one field to another.

For more information on these options, see “Analysis Options for C Code”.

Set Up Project to Check Coding Rules

In this section...
“Polyspace Coding Rules Checker Overview” on page 4-16
“Check Compliance with MISRA C Coding Rules” on page 4-16
“Check Compliance with C++ Coding Rules” on page 4-17

Polyspace Coding Rules Checker Overview

Polyspace software can check that your code complies with established C or C++ coding standards. The coding rules checker can check MISRA C, MISRA C++ or JSF C++ coding standards.²

The Polyspace coding rules checker enables Polyspace software to provide messages when coding rules are not respected. Most messages are reported during the compile phase of a verification.

Note The Polyspace MISRA checker is based on MISRA C:2004, which also incorporates MISRA-C Technical Corrigendum 1 (<http://www.misra-c.com>).

The Polyspace MISRA C++ checker is based on MISRA C++:2008 – “Guidelines for the use of the C++ language in critical systems.” For more information on these coding standards, see <http://www.misra-cpp.com>.

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

For more information on these coding standards, see

http://www.jsf.mil/downloads/documents/JSF_AV_C++_Coding_Standards_Rev_C.doc.

Check Compliance with MISRA C Coding Rules

To check MISRA C compliance, you set an option in your project before running a verification. Polyspace software finds the violations during the compile phase of a verification. When you have addressed all MISRA C violations, you run the verification again.

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

To set the MISRA C checking option:

- 1** In the Project Manager perspective, select the **Configuration > Coding Rules** pane.
- 2** Select the **Check MISRA C rules** check box.
- 3** Specify coding rule checker options, for example:
 - MISRA C rules to check
 - Custom rules
 - Files to exclude from the checking
 - Data types that you want Polyspace to consider as Boolean

Note For more information on using the MISRA C checker, see “Activate Coding Rules Checker” on page 13-2.

Check Compliance with C++ Coding Rules

To check coding rules compliance, you set an option in your project before running a verification. Polyspace software finds the violations during the compile phase of a verification. When you have addressed all coding rules violations, you run the verification again.

To set up coding rules checking:

- 1** In the Project Manager perspective, select the **Configuration > Coding Rules** node.
- 2** Select the **Check MISRA C++ rules** or **Check JSF C++ rules** check box.
- 3** Specify other options:
 - MISRA or JSF C++ rules to check
 - Custom rules
 - Files to exclude from the checking

Note For more information on using the coding rules checker, see “Activate Coding Rules Checker” on page 13-2.

Set Up Project to Automatically Test Orange

In this section...
“Polyspace Automatic Orange Tester” on page 4-19
“Enable Automatic Orange Tester” on page 4-19

Polyspace Automatic Orange Tester

The Polyspace Automatic Orange Tester performs dynamic stress tests on unproven C code (orange checks) to help you identify potential run-time errors. By default, the Automatic Orange Tester is disabled. If you enable the Automatic Orange Tester:

- The software runs the Automatic Orange Tester at the end of static verification
- You can manually run the Automatic Orange Tester after the verification

For more information, see “Test Orange Checks Automatically”.

Enable Automatic Orange Tester

To enable the Automatic Orange Tester:

- 1** In the Project Manager perspective, select the **Configuration > Advanced Settings** pane.
- 2** Select the **Automatic Orange Tester** check box.
- 3** Specify values for the following options:
 - **Number of automatic tests** — Total number of tests. Default is 500. Software supports maximum of 100,000.
 - **Maximum loop iterations** — Maximum number of iterations allowed before a loop is considered to be an infinite loop. Default is 1000, which is also the maximum value that software supports.
 - **Maximum test time** — Maximum time allowed for each test. Default is 5 seconds. Software supports maximum of 60.

For more information about the Automatic Orange Tester, see “Test Orange Checks Automatically”.

Setting Up Project: Additional Information

- “Create Projects Using Visual Studio Information” on page 5-2
- “Cannot create project from Visual Studio build” on page 5-6
- “Storage of Polyspace Preferences” on page 5-7

Create Projects Using Visual Studio Information

In this section...
“Use Visual Studio Project” on page 5-2
“Trace Visual Studio Build” on page 5-3

Use Visual Studio Project

You can directly create a Polyspace project from a Visual Studio project file with extension `.vcproj`. The Visual Studio import retrieves the following information from a Visual Studio project:

- **Source** files
- **Include** folders
- Some **Target & Compiler** options
- **Preprocessor Macros**

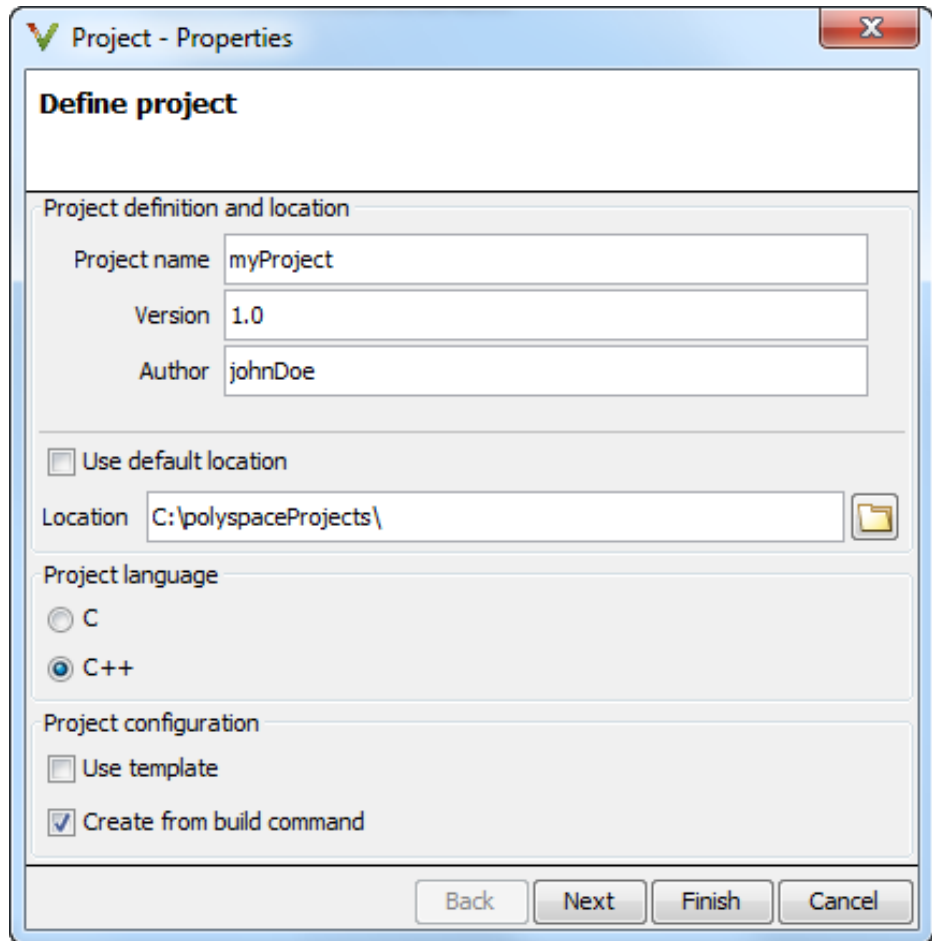
Note For Visual Studio 2010 or Visual Studio 2012, you cannot directly import your project.

- 1** In the Project Manager perspective, select **File > Import Visual Studio Project**.
- 2** In the Import Visual Studio dialog box, specify the **Visual Studio project** that you want to use.
- 3** You can:
 - **Create new Polyspace project:** Enter full path to a new Polyspace project.
 - **Update existing Polyspace project:** The dropdown list contains all projects currently open in the **Project Browser**. Select the project you want to update.
- 4** Click **Import**.

Trace Visual Studio Build

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

- 1** In the Polyspace Project Manager, select **File > New Project**.
- 2** In the Project – Properties window, enter your project information.
 - a** Choose **C++** as **Project Language**.
 - b** Under **Project Configuration**, select **Create from build command** and click **Next**.



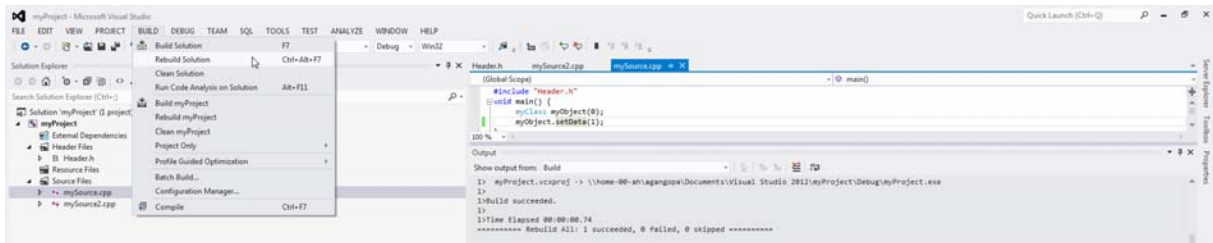
3 In the field **Specify command used for building your source files**, enter the full path to the Visual Studio executable. For instance, "C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\devenv.exe".

4 In the field **Specify working directory for running build command**, enter C:\. Click .

This action opens the Visual Studio environment.

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

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

Related Examples

- “Visual Studio Verification”

Concepts

- “Cannot create project from Visual Studio build” on page 5-6

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
```


Storage of Polyspace Preferences

The software stores the settings that you specify through the Polyspace Preferences dialog box 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 such as johndoe and *\$Release* is the release number such as 2014a.

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

- “Set Up a Target” on page 6-2
- “Verify C Application Without a “Main”” on page 6-33
- “Polyspace C++ Class Analyzer” on page 6-39
- “Data Range Specifications (DRS)” on page 6-55
- “Specify Data Ranges Using DRS Template” on page 6-56
- “Specify Data Ranges Using Existing DRS Configuration” on page 6-58
- “Edit Existing DRS Configuration” on page 6-59
- “Remove Non Applicable Entries from DRS File” on page 6-60
- “Specify Data Ranges Using Text Files” on page 6-61
- “Perform Efficient Module Testing with DRS” on page 6-65
- “Reduce Oranges with DRS” on page 6-67
- “DRS Configuration Settings” on page 6-71
- “Variable Scope” on page 6-76
- “XML Format of DRS File” on page 6-80

Set Up a Target

In this section...
“Target & Compiler Overview” on page 6-2
“Specify Target and Compiler” on page 6-3
“Predefined Target Processor Specifications” on page 6-3
“Modify Predefined Target Processor Attributes” on page 6-6
“Define Generic Target Processors” on page 6-7
“Common Generic Targets” on page 6-9
“View or Modify Existing Generic Targets” on page 6-10
“Delete Generic Target” on page 6-11
“Compile Operating System Dependent Code” on page 6-12
“Address Alignment” on page 6-19
“Ignore or Replace Keywords Before Compilation” on page 6-20
“Language Extensions” on page 6-23
“Verify Keil or IAR Dialects” on page 6-23
“Gather Compilation Options Efficiently” on page 6-31

Target & Compiler Overview

Many applications are designed to run on specific target CPUs and operating systems. The type of CPU determines many data characteristics, such as data sizes and addressing. These factors can determine whether errors occur, for example, overflows.

Since some run-time errors are dependent on the target CPU and operating system, you must specify the type of CPU and operating system used in the target environment before running a verification.

Specify Target and Compiler

In the Project Manager perspective, the **Configuration > Target & Compiler** pane allows you to specify the target environment and compiler behavior for your application.

For example, to specify the target environment for your application:

- 1** From the **Target operating system** drop-down list, select the operating system on which your application is designed to run.
- 2** From the **Target processor type** drop-down list, select the processor on which your application is designed to run.

For detailed specifications of each predefined target processor, see “Predefined Target Processor Specifications” on page 6-3.

Predefined Target Processor Specifications

Polyspace software supports many commonly used processors, as listed in the table below. To specify one of the predefined processors, select it from the **Target processor type** drop-down list.

Predefined Target Processor Specifications

Target	char	short	int	long	long long	float	double	long double	ptr	sign of char	endian	align
i386	8	16	32	32	64	32	64	96	32	signed	Little	32
sparc	8	16	32	32	64	32	64	128	32	signed	Big	64
m68k / ColdFire ³	8	16	32	32	64	32	64	96	32	signed	Big	64
powerpc	8	16	32	32	64	32	64	128	32	unsigned	Big	64
c-167	8	16	16	32	32	32	64	64	16	signed	Little	64
tms320c3x	32	32	32	32	64	32	32	40 ⁴	32	signed	Little	32
sharc21x61	32	32	32	32	64	32	32 [64]	32 [64]	32	signed	Little	32
NEC-V850	8	16	32	32	32	32	32	64	32	signed	Little	32 [16, 8]
hc08 ⁵	8	16	16 [32]	32	32	32	32 [64]	32 [64]	16 ⁶	unsigned	Big	32 [16]
hc12 ⁵	8	16	16 [32]	32	32	32	32 [64]	32 [64]	32 ⁶	signed	Big	32 [16]
mpc5xx ⁵	8	16	32	32	64	32	32 [64]	32 [64]	32	signed	Big	32 [16]
c18	8	16	16	32 [24] ⁷	32	32	32	32	16 [24]	signed	Little	8

3. The M68k family (68000, 68020, etc.) includes the “ColdFire” processor

4. All operations on long double values will be imprecise (that is, shown as orange).

5. Non ANSI C specified keywords and compiler implementation-dependent pragmas and interrupt facilities are not taken into account by this support

6. There are various pointers (near or far) that are 2 bytes (hc08) or 4 bytes (hc12) wide.

7. The c18 target supports the type short long as 24-bits.

Predefined Target Processor Specifications (Continued)

Target	char	short	int	long	long long	float	double	long double	ptr	sign of char	endian	align
x86_64	8	16	32	64 [32] ⁸	64	32	64	96	64	signed	Little	64 [32]
mcpu (Advanced)	8 [16]	8 [16]	16 [32]	32	32 [64]	32	32 [64]	32 [64]	16 [32]	signed	Little	32 [16, 8]

Note The following target processors are supported only for C code verifications: tms320c3x, sharc21x61, NEC-V850, hc08, hc12, mpc5xx, and c18.

After selecting a predefined target, you can modify some default attributes by selecting the browse button to the right of the **Target processor type** drop-down menu. The optional settings for each target are shown in [brackets] in the table.

If your processor is not listed, you can specify a similar processor that shares the same characteristics, or create a generic target processor.

Note If your target processor does not match the characteristics of a processor described above, contact MathWorks technical support for advice.

8. Use option `-long-is-32bits` to support Microsoft C/C++ Win64 target

Modify Predefined Target Processor Attributes

You can modify certain attributes of the predefined target processors. If your specific processor is not listed, you may be able to specify a similar processor and modify its characteristics to match your processor.

Note The settings that you can modify for each target are shown in [brackets] in the Predefined Target Processor Specifications on page 6-4 table.

To modify target processor attributes:

- 1** In the Project Manager perspective, select the **Configuration > Target & Compiler** pane.
- 2** From the **Target processor type** drop-down list, select the target processor that you want to use.
- 3** To the right of the **Target processor type** field, click **Edit**.

The Advanced target options dialog box opens.

- 4** Modify the attributes as required.

For information on each target option, see “Generic target options”.

- 5** Click **OK** to save your changes.

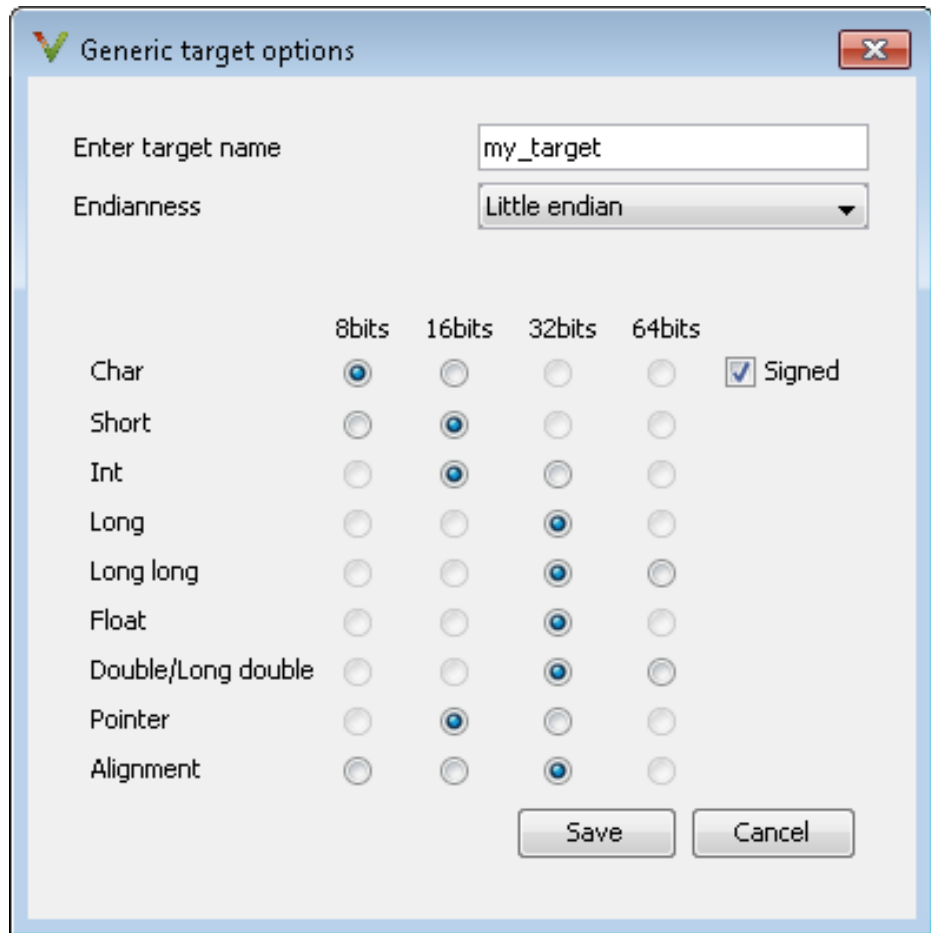
Define Generic Target Processors

If your application is designed for a custom target processor, you can configure many basic characteristics of the target by selecting the selecting the mcpu . . . (Advanced) target, and specifying the characteristics of your processor.

To configure a generic target:

- 1** In the Project Manager perspective, select the **Configuration > Target & Compiler** pane.
- 2** From the **Target processor type** drop-down list, select mcpu . . . (Advanced).

The Generic target options dialog box opens.



- 3 In the **Enter the target name** field, enter a name, for example, MyTarget.
- 4 Specify the parameters for your target, such as the size of basic types, and alignment with arrays and structures.

For example, when the alignment of basic types within an array or structure is 8, the storage assigned to arrays and structures is determined by the size of the individual data objects (without fields and end padding).

Note For information on each target option, see “Generic target options”.

5 Click **Save** to save the generic target options and close the dialog box.

Common Generic Targets

The following tables describe the characteristics of common generic targets.

ST7 (Hiware C compiler : HiCross for ST7)

ST7	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16	32	32	32	32	32	16/32	unsigned	Big
alignment	8	16/8	16/8	32/16/8	32/16/8	32/16/8	32/16/8	32/16/8	32/16/8	N/A	N/A

ST9 (GNU C compiler : gcc9 for ST9)

ST9	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16	32	32	32	64	64	16/64	unsigned	Big
alignment	8	8	8	8	8	8	8	8	8	N/A	N/A

Hitachi H8/300, H8/300L

Hitachi H8/300, H8/300L	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16/32	32	64	32	654	64	16	unsigned	Big
alignment	8	16	16	16	16	16	16	16	16	N/A	N/A

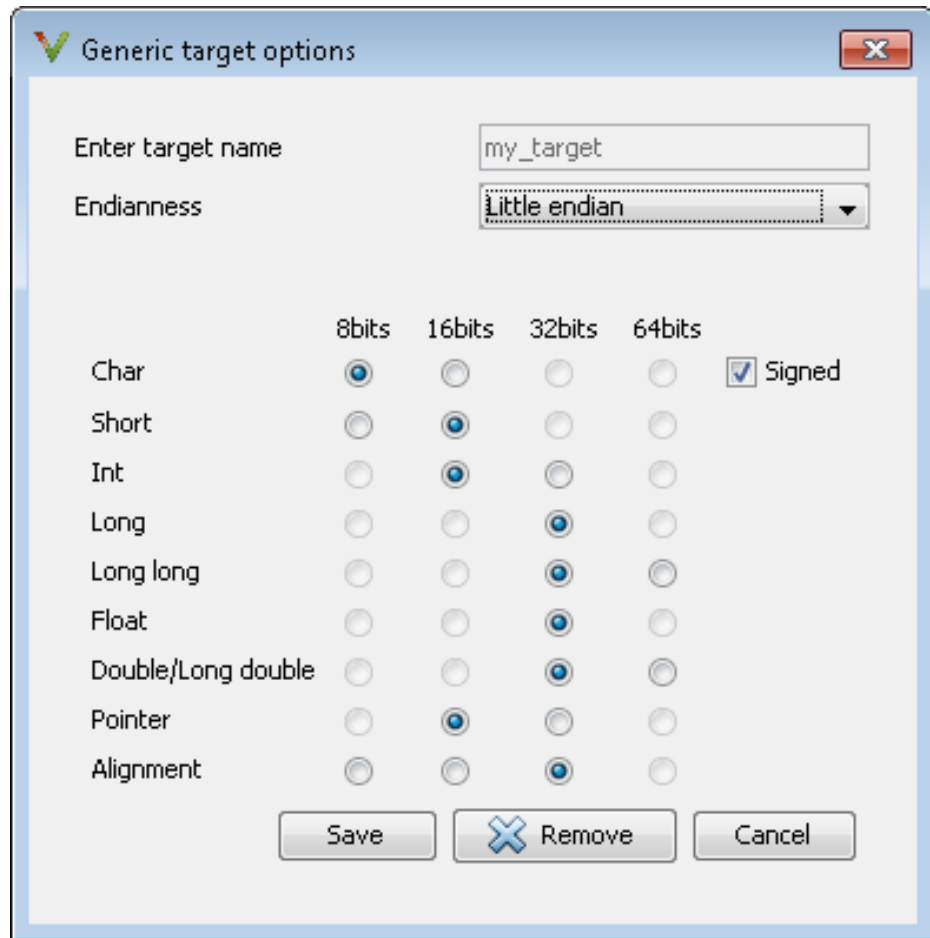
Hitachi H8/300H, H8S, H8C, H8/Tiny

Hitachi H8/300H, H8S, H8C, H8/Tiny	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16/ 32	32	64	32	64	64	32	unsigned	Big
alignment	8	16	32/ 16	32/16	32/16	32/16	32/16	32/16	32/16	N/A	N/A

View or Modify Existing Generic Targets

To view or modify generic targets that you previously created:

- 1** In the Project Manager perspective, select the **Configuration > Target & Compiler** pane.
- 2** From the **Target processor type** drop-down list, select your target, for example, MyTarget.
- 3** Click **Edit**. The Generic target options dialog box opens, displaying your target attributes.



4 If required, specify new attributes for your target. Then click **Save**.

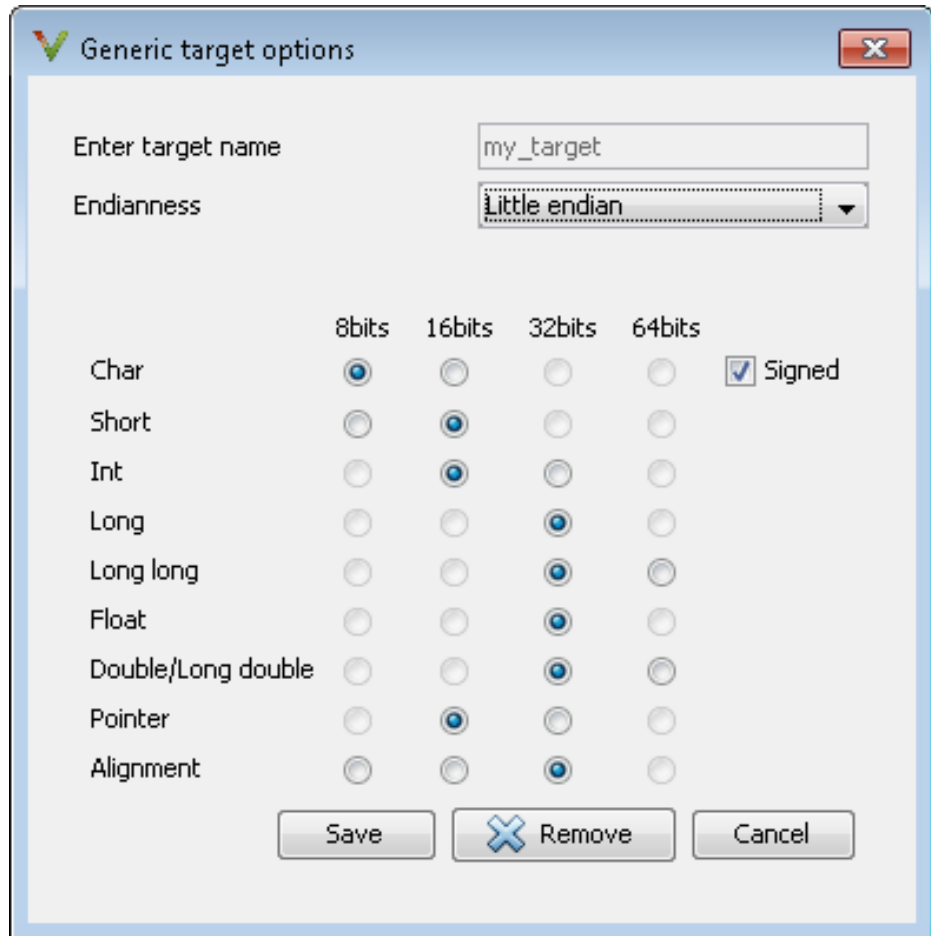
5 Otherwise, click **Cancel**.

Delete Generic Target

To delete a generic target:

1 In the Project Manager perspective, select the **Configuration > Target & Compiler** pane.

- From the **Target processor type** drop-down list, select the target that you want to remove, for example, MyTarget.



- Click **Remove**. The software removes the target from the list.

Compile Operating System Dependent Code

This section describes the options required to compile and verify code designed to run on specific operating systems. It contains the following:

- “Predefined Compilation Flags for C Code” on page 6-13
- “Predefined Compilation Flags for C++ Code” on page 6-15
- “My Target Application Runs on Linux” on page 6-18
- “My Target Application Runs on Solaris” on page 6-18
- “My Target Application Runs on Vxworks” on page 6-18
- “My Target Application Does Not Run on Linux, VxWorks, or Solaris” on page 6-18

Predefined Compilation Flags for C Code

These flags concern the predefined **Target operating system** (-OS-target) option values: no-predefined-OS, Linux, VxWorks, Solaris and Visual.

Target operating system	Compilation flags	–include file and content
no-predefined-OS	-D __STDC__	
Visual	-D __STDC__	-include <product_dir>/cinclude/pst-visual.h
VxWorks	-D __STDC__ -DANSI_PROTOTYPES -DSTATIC= -DCONST=const -D __GNUC__=2 -Dunix -D __unix -D __unix__ -Dsparc -D __sparc -D __sparc__ -Dsun -D __sun -D __sun__ -D __svr4__ -D __SVR4	-include <product_dir>/cinclude/pst-vxworks.h

Target operating system	Compilation flags	-include file and content
Linux	-D __STDC__ -D __GNUC__=2 -D __GNUC_MINOR__=6 -D __GNUC__=2 -D __GNUC_MINOR__=6 -D __ELF__ -D unix -D _unix -D _unix__ -D linux -D _linux -D _linux__ -D i386 -D _i386 -D _i386__ -D i686 -D _i686 -D _i686__ -D pentiumpro -D _pentiumpro -D _pentiumpro__	<product_dir>/cinclude/pst-linux.h
Solaris	-D __STDC__ -D __GNUC__=2 -D __GNUC_MINOR__=8 -D __GNUC__=2 -D __GNUC_MINOR__=8 -D __GCC_NEW_VARARGS__ -D unix -D _unix -D _unix__ -D sparc -D _sparc -D _sparc__ -D sun -D _sun -D _sun__	No -include file mentioned

Target operating system	Compilation flags	-include file and content
	-D__svr4__ -D__SVR4	

Note The use of the `-OS-target` option is equivalent to the following approaches:

- Setting the same `-D` flags manually
- Using the `-include` option on a copied and modified `pst-OS-target.h` file

Predefined Compilation Flags for C++ Code

The following table shown for each `OS-target`, the list of compilation flags defined by default, including pre-include header file (see also `include`):

Target operating system	Compilation flags	-include file	Minimum set of options
Linux	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D__STL_CLASS_PARTIAL_SPECIALIZATION -D__GNU_SOURCE -D__STDC__ -D__ELF__ -Dunix -D__unix -D__unix__ -Dlinux -D__linux__ -D__linux__ -Di386 -D__i386 -D__i386__ -Di686 -D__i686__ -D__i686__ -Dpentiumpro	<product_dir>/ cinclude/ pst-linux.h	polyspace-[desktop-]cpp -OS-target Linux \ -I <polyspace_install>/include/ include-linux \ -I <product_dir>/include/ include-linux/next Where the Polyspace product has been installed in the folder <polyspace_install>

Target operating system	Compilation flags	-include file	Minimum set of options
	-D__pentiumpro -D__pentiumpro__		
VxWorks	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D__STL_CLASS_PARTIAL_SPECIALIZATION -DANSI_PROTOTYPES -DSTATIC= -DCONST=const -D__STDC -D__GNU_SOURCE -Dunix -D__unix -D__unix__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__ -D__svr4 -D__SVR4	<product_dir>/ cinclue/ pstvxworks. h	polyspace-[desktop-].cpp \ -OS-target vxworks \ -I /your_path_to/ Vxworks_include_folders
Visual	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__STRICT_ANSI__ -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D__POSIX_SOURCE -D__STL_CLASS_PARTIAL_SPECIALIZATION	<product_dir>/ cinclue/ pstvisual. h	

Target operating system	Compilation flags	-include file	Minimum set of options
Solaris	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D__STL_CLASS_PARTIAL_SPECIALIZATION -D__GNU_SOURCE -D__STDC -D__GCC_NEW_VARARGS__ -Dunix -D__unix -D__unix__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__ -D__svr4 -D__SVR4		If Polyspace runs on a Linux machine: <pre>polyspace-[desktop-]cpp \ -OS-target Solaris \ -I /your_path_to_solaris_include</pre> If Polyspace runs on a Solaris machine: <pre>polyspace-cpp \ -OS-target Solaris \ -I /usr/include</pre>
no-predefined OS	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__STRICT_ANSI__ -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D__POSIX_SOURCE -D__STL_CLASS_PARTIAL_SPECIALIZATION		<pre>polyspace-[desktop-]cpp \ -OS-target no-predefined-OS \ -I /your_path_to/ MyTarget_include_folders</pre>

Note This list of compiler flags is written in every log file.

My Target Application Runs on Linux

The minimum set of options is as follows:

```
polyspace-code-prover-nodesktop \  
-OS-target Linux \  
-I MATLAB_Install/polyspace/verifier/cxx/include/include-libc \  
  
...
```

If your target application runs on Linux but you are starting your verification from Windows, the minimum set of options is as follows:

```
polyspace-code-prover-nodesktop \  
-OS-target Linux \  
-I MATLAB_Install\polyspace\verifier\cxx\include\include-libc \  
  
...
```

MATLAB_Install is your MATLAB installation folder.

My Target Application Runs on Solaris

If Polyspace software runs on a Linux machine:

```
polyspace-code-prover-nodesktop \  
-OS-target Solaris \  
-I /your_path_to_solaris_include
```

My Target Application Runs on Vxworks

If Polyspace software runs on either a Solaris™ or a Linux machine:

```
polyspace-code-prover-nodesktop \  
-OS-target vxworks \  
-I /your_path_to/Vxworks_include_folders
```

My Target Application Does Not Run on Linux, VxWorks, or Solaris

If Polyspace software does not run on a Linux machine:

```
polyspace-code-prover-nodesktop \  
-
```

```
-OS-target no-predefined-OS \
-I /your_path_to/MyTarget_include_folders
```

Address Alignment

Polyspace software handles address alignment by calculating `sizeof` and alignments. This approach takes into account 3 constraints implied by the ANSI standard which ensure that:

- Global `sizeof` and `offsetof` fields are optimum, that is, as short as possible.
- The alignment of addressable units is respected.
- Global alignment is respected.

Consider the example:


```
struct foo {char a; int b;}
```

- Each field must be aligned; that is, the starting offset of a field must be a multiple of its own size⁹
- So in the example, `char a` begins at offset 0 and its size is 8 bits. `int b` cannot begin at 8 (the end of the previous field) because the starting offset must be a multiple of its own size (32 bits). Consequently, `int b` begins at `offset=32`. The size of the `struct foo` before global alignment is therefore 64 bits.
- The global alignment of a structure is the maximum of the individual alignments of each of its fields;
- In the example, `global_alignment = max (alignment char a, alignment int b) = max (8, 32) = 32`
- The size of a struct must be a multiple of its global alignment. In our case, `b` begins at 32 and is 32 long, and the size of the struct (64) is a multiple of the `global_alignment` (32), so `sizeof` is not adjusted.

9. except in the cases of “double” and “long” on some targets.

Ignore or Replace Keywords Before Compilation

You can ignore noncompliant keywords, for example, `far` or `0x`, which precede an absolute address. The template `myTpl.pl` (listed below) allows you to ignore these keywords:

- 1 Save the listed template as `C:\Polyspace\myTpl.pl`.
- 2 Select the **Configuration > Target & Compiler > Environment Settings** pane.
- 3 To the right of the **Command/script to apply to preprocessed files** field, click  field, 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.

For more information, see `-post-preprocessing-command`.

Content of `myTpl.pl` file

```
#!/usr/bin/perl

#####
# Post Processing template script
#
#####
# Usage from Project Manager GUI:
#
# 1) Linux: /usr/bin/perl PostProcessingTemplate.pl
# 2) Windows: MATLAB_Install\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;

    # Convert current line to lower case
    # $_ =~ tr/A-Z/a-z/;

    # Print the current processed line
    print $OUTFILE $_;
}

```

Perl Regular Expression Summary

```

#####
# 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/;
#
#####
```


Language Extensions

The software allows a verification to accept a subset of common C language constructs and extended keywords, as defined by the C99 standard or supported by many compilers.

By default, the following constructs are accepted:

- 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 software ignores the following extended keywords:

- `near`
- `far`
- `restrict`
- `_attribute_(X)`
- `rom`

Verify Keil or IAR Dialects

Typical embedded control applications frequently read and write port data, set timer registers and read input captures. To deal with this without using

assembly language, some microprocessor compilers have specified special data types like `sfr` and `sbit`. Typical declarations are:

```
sfr A0 = 0x80;
sfr A1 = 0x81;
sfr ADCUP = 0xDE;
sbit EI = 0x80;
```

These declarations reside in header files such as `regxx.h` for the basic 80Cxxx micro processor. The definition of `sfr` in these header files customizes the compiler to the target processor.

When accessing a register or a port, using `sfr` data is then simple, but is not part of standard ANSI C:

```
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) */
}
```

You can verify this type of code using the **Dialect** (`-dialect`) option . This option allows the software to support the Keil or IAR C language extensions even if some structures, keywords, and syntax are not ANSI standard. The following tables summarize what is supported when verifying code that is associated with the Keil or IAR dialects.

The following table summarizes the supported Keil C language extensions:

Example: -dialect keil -sfr-types sfr=8

Type/Language	Description	Example	Restrictions
Type bit	<ul style="list-style-type: none"> An expression to type bit gives values in range [0,1]. Converting an expression in the type, gives 1 if it is not equal to 0, else 0. This behavior is similar to c++ bool type. 	<pre>bit x = 0, y = 1, z = 2; assert(x == 0); assert(y == 1); assert(z == 1); assert(sizeof(bit) == sizeof(int));</pre>	pointers to bits and arrays of bits are not allowed
Type sfr	<ul style="list-style-type: none"> The -sfr-types option defines unsigned types name and size in bits. The behavior of a variable follows a variable of type integral. A variable which overlaps another one (in term of address) will be considered as volatile. 	<pre>sfr x = 0xf0; // declaration of variable x at address 0xF0 sfr16 y = 0x4EEF;</pre> <p>For this example, options need to be:</p> <pre>-dialect keil -sfr-types sfr=8, \ sfr16=16</pre>	sfr and sbit types are only allowed in declarations of external global variables.

Example: -dialect keil -sfr-types sfr=8 (Continued)

Type/Language	Description	Example	Restrictions
Type sbit	<ul style="list-style-type: none"> Each read/write access of a variable is replaced by an access of the corresponding sfr variable access. Only external global variables can be mapped with a sbit variable. Allowed expressions are integer variables, cells of array of int and struct/union integral fields. a variable can also be declared as extern bit in an another file. 	<pre>sfr x = 0xF0; sbit x1 = x ^ 1; // 1st bit of x sbit x2 = 0xF0 ^ 2; // 2nd bit of x sbit x3 = 0xF3; // 3rd bit of x sbit y0 = t[3] ^ 1; /* file1.c */ sbit x = P0 ^ 1; /* file2.c */ extern bit x; x = 1; // set the 1st bit of P0 to 1</pre>	
Absolute variable location	Allowed constants are integers, strings and identifiers.	<pre>int var _at_ 0xF0 int x @ 0xFE ; static const int y @ 0xA0 = 3;</pre>	Absolute variable locations are ignored (even if declared with a #pragma location).

Example: -dialect keil -sfr-types sfr=8 (Continued)

Type/Language	Description	Example	Restrictions
Interrupt functions	A warnings in the log file is displayed when an interrupt function has been found: "interrupt handler detected : <name>" or "task entry point detected : <name>"	<pre>void foo1 (void) interrupt XX = YY using 99 { } void foo2 (void) _ task_ 99 _priority_ 2 { }</pre>	Entry points and interrupts are not taken into account as -entry-points.
Keywords ignored	alien, bdata, far, idata, ebddata, huge, sdata, small, compact, large, reentrant. Defining -D __C51__, keywords large code, data, xdata, pdata and xhuge are ignored.		

The following table summarize the IAR dialect:

Example: -dialect iar -sfr-types sfr=8

Type/Language	Description	Example	Restrictions
Type bit	<ul style="list-style-type: none"> An expression to type bit gives values in range [0,1]. Converting an expression in the type, gives 1 if it is not equal to 0, else 0. This behavior is similar to c++ bool type. If initialized with values 0 or 1, a variable of type bit is a simple variable (like a c++ bool). 	<pre>union { int v; struct { int z; } y; } s; void f(void) { bit y1 = s.y.z . 2; bit x4 = x.4; bit x5 = 0xF0 . 5; y1 = 1; // 2nd bit of s.y.z // is set to 1 };</pre>	pointers to bits and arrays of bits are not allowed

Example: -dialect iar -sfr-types sfr=8 (Continued)

Type/Language	Description	Example	Restrictions
	<ul style="list-style-type: none"> A variable of type bit is a register bit variable (mapped with a bit or a sfr type) 		
Type sfr	<ul style="list-style-type: none"> The -sfr-types option defines unsigned types name and size. The behavior of a variable follows a variable of type integral. A variable which overlaps another one (in term of address) will be considered as volatile. 	<pre>sfr x = 0xf0; // declaration of variable x at address 0xF0</pre>	sfr and sbit types are only allowed in declarations of external global variables.
Individual bit access	<ul style="list-style-type: none"> Individual bit can be accessed without using sbit/bit variables. Type is allowed for integer variables, cells of integer array, and struct/union integral fields. 	<pre>int x[3], y; x[2].2 = x[0].3 + y.1;</pre>	
Absolute variable location	Allowed constants are integers, strings and identifiers.	<pre>int var @ 0xF0; int xx @ 0xFE ; static const int y \ @0xA0 = 3;</pre>	Absolute variable locations are ignored (even if declared with a #pragma location).

Example: -dialect iar -sfr-types sfr=8 (Continued)

Type/Language	Description	Example	Restrictions
Interrupt functions	<ul style="list-style-type: none"> • A warning is displayed in the log file when an interrupt function has been found: "interrupt handler detected : funcname" • A monitor function is a function that disables interrupts while it is executing, and then restores the previous interrupt state at function exit. 	<pre>interrupt [1] \ using [99] void \ foo1(void) { ... }; monitor [3] void \ foo2(void) { ... };</pre>	Entry points and interrupts are not taken into account as -entry-points.
Keywords ignored	saddr, reentrant, reentrant_idata, non_banked, plm, bdata, idata, pdata, code, data, xdata, xhuge, interrupt, __interrupt and __intrinsic		
Unnamed struct/union	<ul style="list-style-type: none"> • Fields of unions/structs with no tag and no name can be accessed without naming their parent struct. • On a conflict between a field of an anonymous struct with other identifiers: <ul style="list-style-type: none"> ▪ with a variable name, field name is hidden ▪ with a field of another 	<pre>union { int x; }; union { int y; struct { int z; }; } @ 0xF0;</pre>	

Example: -dialect iar -sfr-types sfr=8 (Continued)

Type/Language	Description	Example	Restrictions
	<p>anonymous struct at different scope, closer scope is chosen</p> <ul style="list-style-type: none"> with a field of another anonymous struct at same scope: an error "anonymous struct field name <name> conflict" is displayed in the log file. 		
no_init attribute	<ul style="list-style-type: none"> a global variable declared with this attribute is handled like an external variable. It is handled like a type qualifier. 	<pre>no_init int x; no_init union { int y; } @ 0xFE;</pre>	#pragma no_init has no effect

The option `-sfr-types` defines the size of a `sfr` type for the Keil or IAR dialect.

The syntax for an `sfr` element in the list is `type-name=typesize`.

For example:

```
-sfr-types sfr=8,sfr16=16
```

defines two `sfr` types: `sfr` with a size of 8 bits, and `sfr16` with a size of 16-bits. A value `type-name` must be given only once. 8, 16 and 32 are the only supported values for `type-size`.

Note As soon as an `sfr` type is used in the code, you must specify its name and size, even if it is the keyword `sfr`.

Note Many IAR and Keil compilers currently exist that are associated to specific targets. It is difficult to maintain a complete list of those supported.

Gather Compilation Options Efficiently

The code is often tuned for the target (see “Verify Keil or IAR Dialects” on page 6-23). Instead of applying minor changes to the code, create a single `polyspace.h` file that contains all target specific functions and options. The `-include` option can then be used to force the inclusion of the `polyspace.h` file in all source files under verification.

Where there are missing prototypes or conflicts in variable definition, writing the expected definition or prototype within such a header file will yield several advantages.

Direct benefits:

- The error detection is much faster since it will be detected during compilation rather than in the link or subsequent phases.
- The position of the error will be identified more precisely.
- There will be no need to modify original source files.

Indirect benefits:

- The file is automatically included as the very first file in the original `.c` files.
- The file can contain much more powerful macro definitions than simple `-D` options.
- The file is reusable for other projects developed under the same environment.

Example

This is an example of a file that can be used with the `-include` option.

```
// The file may include (say) a standard include file implicitly
// included by the cross compiler

#include <stdlib.h>
#include "another_file.h"

// Generic definitions, reusable from one project to another
#define far
#define at(x)

// A prototype may be positioned here to aid in the solution of
// a link phase conflict between
// declaration and definition. This will allow detection of the
// same error at compilation time instead of at link time.
// Leads to:
// - earlier detection
// - precise localisation of conflict at compilation time
void f(int);

// The same also applies to variables.
extern int x;

// Standard library stubs can be avoided,
// and OS standard prototypes redefined.

#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 a “Main”

In this section...

“Main Generator Overview” on page 6-33

“Automatically Generate a Main” on page 6-33

“Manually Generate a Main” on page 6-35

“Specify Call Sequence” on page 6-36

“Specify Functions Not Called by Generated Main” on page 6-37

“Main Generator Assumptions” on page 6-38

Main Generator Overview

When your application is a function library (API) or a single module, you must provide a main that calls each function because of the execution model used by Polyspace verification. You can either manually provide a main, or use Polyspace software to automatically generate a main.

If you specify **Verify whole application**, the verification stops if the software does not detect a main.

If you specify **Verify module**, the software checks your code for a main:

- If a main exists in your source files, the verification uses that main.
- If a main does not exist, the software generates a main using the options you specify.

Automatically Generate a Main

To automatically generate a main, on the **Configuration > Code Prover Verification** pane, click **Verify module**.

Note If a main exists in your code, then the verification uses this main and disregards the **Verify module** options.

For cyclic program code generated from a Simulink model, the generated main:

- 1 Initializes calibration variables identified by the `-variables-written-before-loop` option.
- 2 Calls initialization functions specified by the `-functions-called-before-loop` option.
- 3 Initializes input variables identified by the `-variables-written-in-loop` option. This initialization of variables is performed for each cycle.
- 4 Calls cyclic functions specified by the `-functions-called-in-loop` option.
- 5 Calls termination functions specified by the option `-functions-called-after-loop`.

For other code, the generated main:

- 1 Initializes variables identified by the `-main-generator-writes-variables` option.
- 2 Calls initialization functions specified by the `-functions-called-before-main` option.
- 3 Calls functions specified by the `-main-generator-calls` option. The order and the number of times that the functions are called is not specified.

Main for Generated Code

The following example shows how to use the main generator options to generate a main for a cyclic program, such as code generated from a Simulink model.

```
init parameters // -variables-written-before-loop
init_fct()     // -functions-called-before-loop

volatile int random = 0;
while(random){ // Start main loop
    init inputs // -variables-written-in-loop
    step_fct() // -functions-called-in-loop
```

```
}  
terminate_fct() // -functions-called-after-loop
```

Manually Generate a Main

Manually generating a main is often preferable to an automatically generated main, because it allows you to provide a more accurate model of the calling sequence to be generated.

To manually define the main:

- 1 Identify the API functions and extract their declarations.
- 2 Create a main containing declarations of a volatile variable for each type that is mentioned in the function prototypes.
- 3 Create a loop with a volatile end condition.
- 4 Inside this loop, create a switch block with a volatile condition.
- 5 For each API function, create a case branch that calls the function using the volatile variable parameters you created.

Consider the following example. Suppose that the API functions are:

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

You should create the following main:

```
void main()  
{  
    volatile int random; /* We need an integer variable as a function  
    parameter */  
    volatile void * volatile ptr; /* We need a void pointer as a function  
    parameter */  
    while (random) {  
        switch (random) {  
            case 1:  
                random = func1(ptr, random); break; /* One API function call */  
            default:  
                func2(random, random); /* Another API function call */  
        }  
    }  
}
```

```
}  
}
```

Specify Call Sequence

Polyspace software verifies functions on the basis that the functions can be called in any order. Consider a scenario where a function `f` is listed before a function `g`. If actions in `f` must be executed before `g` is called, writing a main which calls `f` and `g` in the required order will produce a higher selectivity.

Colored Source Code Example

With default settings, a Polyspace verification will not identify defects in the following example.

```
static char x;  
static int y;  
  
void f(void)  
{  
y = 300;  
}  
  
void g(void)  
{  
x = y; // red or green OVFL?  
}
```

However, if you know the call sequence, you can create a main that calls the functions in the desired order:

```
void main(void)  
{  
f()  
g()  
}
```

If `f` is called first, the assignment `x = y;` generates a red check as assigning 300 to a `char` is incorrect. The assignment statement would be green if `g` were called before `f`.

Specify Functions Not Called by Generated Main

You can specify source files in your project that the main generator will ignore. Functions defined in these source files are not called by the automatically generated main.

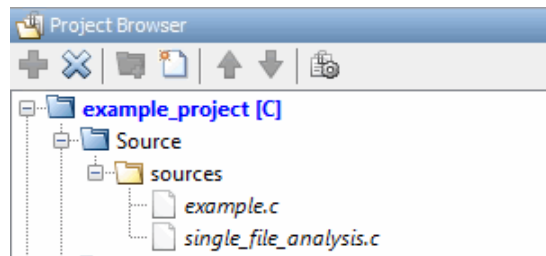
Use this option for files containing function bodies, so that the verification looks for the function body only when the function is called by a primary source file and no body is found.

Note This option applies only to an automatically generated main. Therefore, you must also select the option **Verify module** (-main-generator) for this option to take effect.

To specify source files that the generated main does not call:

- 1 From the Project Browser, in your **Source** folder, select the source files that you want the main generator to ignore.
- 2 Right click any selected file. From the context menu, select **Define As > Not Called by Main Generator**.

The names of ignored files are *italicised*.



Note To specify that a file previously marked **Not Called by Main Generator** should be called, right-click the file in the **Source** folder. From the context menu, select **Regular Source File**.

Main Generator Assumptions

When using the automatic main generator to verify a specific function, the objective is to find problems with the function. To do this, the generated main makes assumptions about parameters so that you can focus on run-time errors (red, gray and orange) that are related to the function.

The main generator makes assumptions about the arguments of called functions to reduce the number of orange checks in the results. Therefore, when you see an orange check in your results, it is likely to be due to the function, not the main.

However, green checks are computed with the same assumptions. Therefore, you should be cautious of green checks involving the main, especially when conducting unit-by-unit verification.

Polyspace C++ Class Analyzer

In this section...

“Why Provide a Class Analyzer” on page 6-39
“How the Class Analyzer Works” on page 6-40
“Sources Verified” on page 6-40
“Architecture of Generated Main” on page 6-40
“Class Verification Log File” on page 6-41
“Characteristics of Class and Log File Messages” on page 6-42
“Behavior of Global Variables and Members” on page 6-42
“Methods and Class Specifics” on page 6-45
“Simple Class” on page 6-47
“Simple Inheritance” on page 6-49
“Multiple Inheritance” on page 6-50
“Abstract Classes” on page 6-51
“Virtual Inheritance” on page 6-52
“Other Types of Classes” on page 6-53

Why Provide a Class Analyzer

One aim of object-oriented languages such as C++ is reusability. A class or a class family is reusable if it is free of defects for all possible uses of the class. The class can be considered free of defects if run-time errors have been removed and the class passes functional tests. The foremost objective when developing code in such a language is to identify and remove as many run-time errors as possible.

Polyspace class analyzer is a tool for removing run-time errors at compilation time. The software will simulate all uses of a class by:

- 1 Creating objects using all constructors (default if none exist).

- 2 Calling all methods (public, static, and protected) of previous objects in every order.
- 3 Calling all methods of the class between time zero and infinity.
- 4 Calling every destructor of previous objects (if they exist).

How the Class Analyzer Works

Polyspace Class Analyzer verifies applications class by class, even if these classes are only partially developed.

The **benefits** of this process include error detection at a very early stage, even if the class is not fully developed, without test cases to write. The process is very simple: provide the class name and the software will verify its robustness.

- Polyspace software generates a “pseudo” main.
- It calls each constructor of the class.
- It then calls each public function from the constructors.
- Each parameter is initialized with full range (i.e., with a random value).
- External variables are assigned random values.

Note Only prototypes of objects (classes, methods, variables, etc.) are required to verify a given class. Missing code is automatically stubbed.

Sources Verified

The sources associated with the verification normally concern public and protected methods of the class. However, sources can also come from inherited classes (fathers) or be the sources of other classes that are used by the class under investigation (friend, etc.).

Architecture of Generated Main

Polyspace software generates the call to each constructor and method of the class. Each method will be analyzed with all constructors. Each parameter is

initialized to random. Note that even if you can get an idea of the architecture of the generated main in the Results Manager perspective, the main is not real. You cannot reuse or compile it.

Consider an example class `MathUtils`. This class contains one constructor, one destructor and seven public methods. The architecture of the generated main is as follows:

```
Generating call to constructor: MathUtils:: MathUtils ()
While (random) {
  If (random) Generating call to function: MathUtils::Pointer_Arithmetic()
  If (random) Generating call to function: MathUtils::Close_To_Zero()
  If (random) Generating call to function: MathUtils::MathUtils()
  If (random) Generating call to function: MathUtils::Recursion_2(int *)
  If (random) Generating call to function: MathUtils::Recursion(int *)
  If (random) Generating call to function: MathUtils::Non_Infinite_Loop()
  If (random) Generating call to function: MathUtils::Recursion_caller()
}
Generating call to destructor: MathUtils::~MathUtils()
```

Note If a class contains more than one constructor, they are called before the “while” statement in an “if then else” statement. This architecture ensures that the verification will evaluate each function method with every constructor.

Class Verification Log File

During a class verification, the list of methods used for the main appears in the log file during the normalization phase of the C++ verification.

You can view the details of what is analyzed in the log file. Consider an example class `MathUtils` with an associated log file:

```
...
* Generating the Main ...
Generating call to function: MathUtils::Pointer_Arithmetic()
Generating call to function: MathUtils::Close_To_Zero()
Generating call to function: MathUtils::MathUtils()
Generating call to function: MathUtils::Recursion_2(int *)
```

```
Generating call to function: MathUtils::Recursion(int *)
Generating call to function: MathUtils::Non_Infinite_Loop()
Generating call to function: MathUtils::~~MathUtils()
Generating call to function: MathUtils::Recursion_caller()
```

If a main is defined in the files being analyzed, you receive a warning:

```
* Warning: a main procedure already exists but will be ignored.
```

Characteristics of Class and Log File Messages

The log file may contain some error messages concerning the class to be analyzed. These messages appear when characteristics of a class are not respected.

- It is not possible to analyze a class that does not exist in the given sources. The verification will halt with the following message:

```
-----
@User Program Error: Argument of option -class-analyzer
must be defined : <name>.
Please correct the program and restart the verifier.
-----
```

- It is not possible to analyze a class that only contains declarations without code. The verification will halt with the following message:

```
-----
@User Program Error: Argument of option -class-analyzer
must contain at least one function : <name>.
Please correct the program and restart the verifier.
-----
```

Behavior of Global Variables and Members

Global Variables

During a class verification, global variables are not considered to be following ANSI Standard anymore if they are defined but not initialized. Remember that ANSI Standard considers, by default, that global variables are initialized to zero.

In a class verification, global variables do not follow standard behaviors:

- Defined variables are initialized to random and then follow the data flow of the code to be analyzed.
- Initialized variables are used with the specified initialized values and then follow the data flow of the code to be analyzed.
- External variables are assigned definitions and initialized to random values.

An example below demonstrates the behaviors of two global variables:

```
1
2 extern int fround(float fx);
3
4 // global variables
5 int globvar1;
6 int globvar2 = 100;
7
8 class Location
9 {
10 private:
11 void calculate_new(void);
12 int x;
13
14 public:
15 // constructor 1
16 Location(int intx = 0) { x = intx; };
17 // constructor 2
18 Location(float fx) { x = fround(fx); };
19
20 void setx(int intx) { x = intx; calculate_new(); };
21 void fsetx(float fx) {
22 int tx = fround(fx);
23 if (tx / globvar1 != 0) // ZDV check is orange
24 {
25 tx = tx / globvar2; // ZDV check is green
26 setx(tx);
27 }
28 };
```

```
29 };
```

In the above example, `globvar1` is defined but not initialized (see line 5), so the check ZDV is orange at line 23. In the same example, `globvar2` is initialized to 100 (see line 6), so the ZDV check is green at line 25.

Data Members of Other Classes

During the verification of a specific class, variable members of other classes, even members of parent classes, are considered to be initialized. They exhibit the following behaviors:

- 1 They may not be considered to be initialized if the constructor of the class is not defined. They are assigned to full range, and then they follow the data flow of the code to be analyzed.
- 2 They are considered to be initialized to the value defined in the constructor if the constructor of the class is defined in the class and is provided for the verification. If the `-class-only` option is applied, the software behaves as though the definition of the constructor is missing (see item 1 above).
- 3 They may be checked as run-time errors if and only if the constructor is defined but does not initialize the member under consideration.

The example below displays the results of a verification of the class `MyClass`. It demonstrates the behavior of a variable member of the class `OtherClass` that was provided without the definition of its constructor. The variable member of `OtherClass` is initialized to random; the check is orange at line 7 and there are possible overflows at line 17 because the range of the return value `wx` is “full range” in the type definition.

```
class OtherClass
{
protected:
    int x;
public:
    OtherClass (int intx);    // code is missing
    int getMember(void) {return x;}; // NIV is warning
};
class MyClass
{
```

```

    OtherClass m_loc;
public:
    MyClass(int intx) : m_loc(0) {};
    void show(void) {
        int wx, wl;
        wx = m_loc.getMember();
        wl = wx*wx + 2;    // Possible overflows because OtherClass
                        // member is assigned to full range
    };
};

```

Methods and Class Specifics

Template

A template class cannot be verified on its own. Polyspace software will only consider a specific instance of a template to be a class that can be analyzed.

Consider `template<class T, class Z> class A { }.`

If we want to analyze template class A with two class parameters T and Z, we have to define a typedef to create an instance of the template with specified specializations for T and Z. In the example below, T represents an int and Z a double:

```

template class A<int, double>;    // Explicit specialisation
typedef class A<int, double> my_template;

```

`my_template` is used as a parameter of the `-class-analyzer` option in order to analyze this instance of template A.

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.”

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 invisible(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 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 all methods in any order many times.

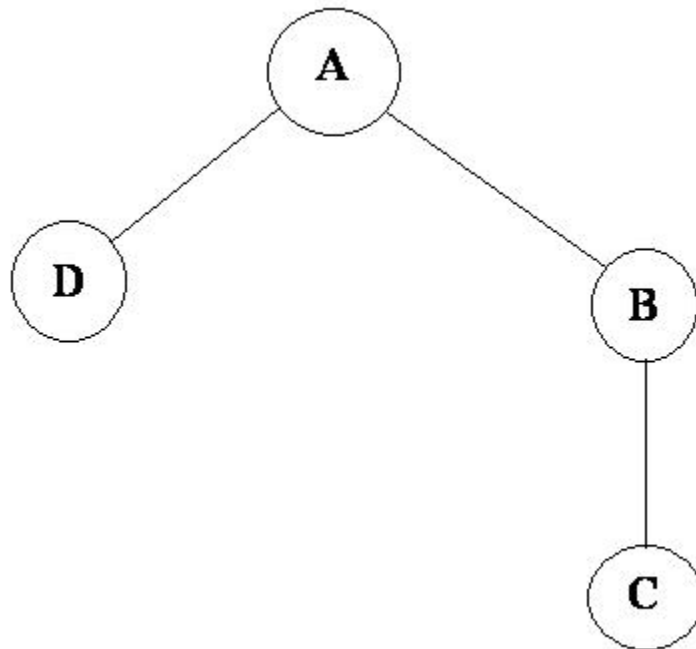
The verification of this class highlights two problems:

- 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.

Simple Inheritance

Consider the following classes:



A is the base class of B and D.

B is the base class of C.

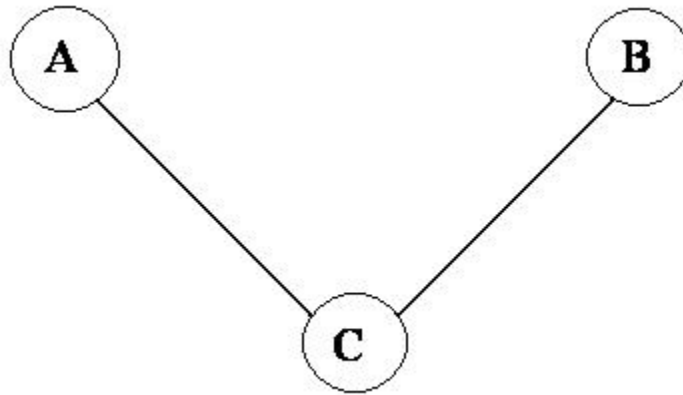
In a case such as this, Polyspace software allows you to run the following verifications:

- 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:



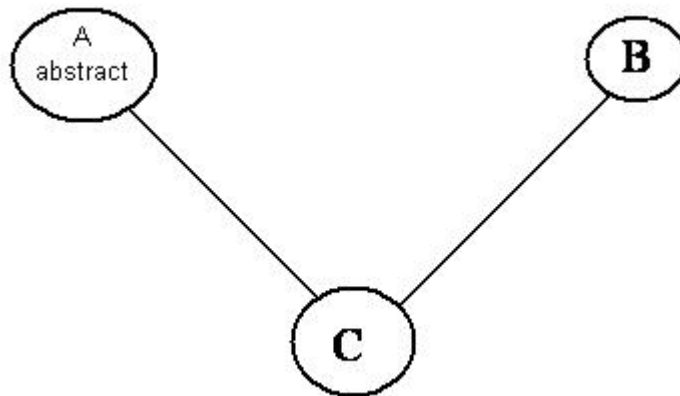
A and B are base classes of C.

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.

Abstract Classes

Consider the following classes:



A is an abstract class

B is a simple class.

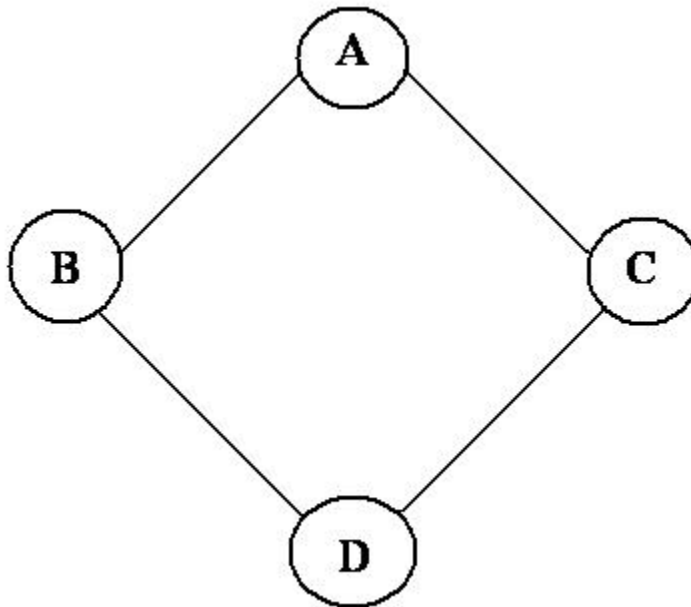
A and B are base classes of C.

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.”

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.

Other Types of Classes

Template Class

A template class can not be analyzed directly. But a class instantiating a template can be analyzed by Polyspace software.

Note If only the template declaration is provided, missing functions' definitions will automatically be stubbed.

Example

```
template<class T > class A {
public:
    T i;
    T geti() {return i;}
    A() : i(1) {}
};
```

You have to define a typedef to create a specialization of the template:

```
template class A<int>;          // Explicit specialization
typedef class A<int> my_template; // complete instance of the template
```

and use option `-class-analyzer my_template`.

The software will analyze a single instance of the template.

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.

Data Range Specifications (DRS)

By default, Polyspace software performs *robustness verification*, proving that the software does not generate run-time errors for all verification conditions. Robustness verification assumes that all data inputs are set to their full range. Therefore, nearly any operation on these inputs could produce an overflow.

The Polyspace Data Range Specifications (DRS) feature allows you to perform *contextual verification*, proving that the software works under normal working conditions. Using DRS, you set constraints on data ranges, and verify the code within these ranges. This can substantially reduce the number of orange checks in the verification results.

You can use DRS to set constraints on:

- Global variables
- Input parameters for user-defined functions called by the main generator
- Return values for stub functions

Note Only one mode can be applied to a global variable.

No checks are added with this option except for `globalassert` mode.

Some warning can be displayed in log file concerning variables when `format` or `type` is not in the scope.

Related Examples

- “Specify Data Ranges Using DRS Template” on page 6-56
- “Specify Data Ranges Using Existing DRS Configuration” on page 6-58
- “Specify Data Ranges Using Text Files” on page 6-61
- “Perform Efficient Module Testing with DRS” on page 6-65
- “Reduce Oranges with DRS” on page 6-67

Specify Data Ranges Using DRS Template

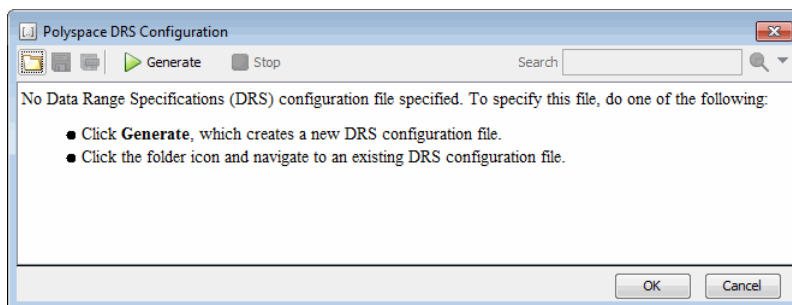
To use the DRS feature, you must provide a list of variables (or functions) and their associated data ranges.

Polyspace software can analyze the files in your project, and generate a DRS template containing all the global variables, user-defined functions, and stub functions for which you can specify data ranges. You can then modify this template to set data ranges.

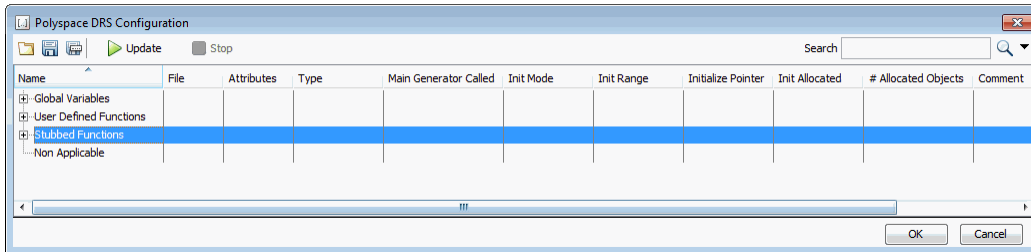
To use a DRS template to set data ranges:

- 1 Open the project for which you want to set data ranges.
- 2 Check that the project contains the source files and include folders that you want to analyze, and specifies the configuration options that you want to use. The software generates a DRS template after compiling the code.
- 3 In the Project Manager perspective, select the **Configuration > Inputs & Stubbing** node.
- 4 To the right of the **Variable/function range setup** field, click the **Edit** button.

The Polyspace DRS Configuration dialog box opens.




- 5 On the toolbar, click **Generate**. The software compiles the project and generates a DRS template, for example, `Bug_Finder_Example-with-MISRA-checker_drs_template.xml`. You can view the DRS values through the Polyspace DRS Configuration dialog box.





Note If the option `-unit-by-unit` is enabled:

- The generated file represents the union of DRS values generated for each unit.
- The DRS file generation functionality is not supported for C++.

6 Specify the data ranges for global variables, user-defined function inputs, and stub-function return values. For more information, see “DRS Configuration Settings” on page 6-71.

7 To save your DRS configuration file, click  (Save DRS).

To save your DRS configuration file to a location that you specify, click  (Save DRS as).

8 If you change your source code, click  Update to generate an updated DRS configuration file. As a result of the source code changes, the updated file might contain entries that no longer apply to your code. You can remove these entries from the file. See “Remove Non Applicable Entries from DRS File” on page 6-60.

9 Click **OK** to close the Polyspace DRS Configuration dialog box. The **Variable/function range setup** field now contains the name of the DRS configuration file. The software uses this DRS configuration file the next time you start a verification.

10 Select **File > Save Project** to save your project settings.

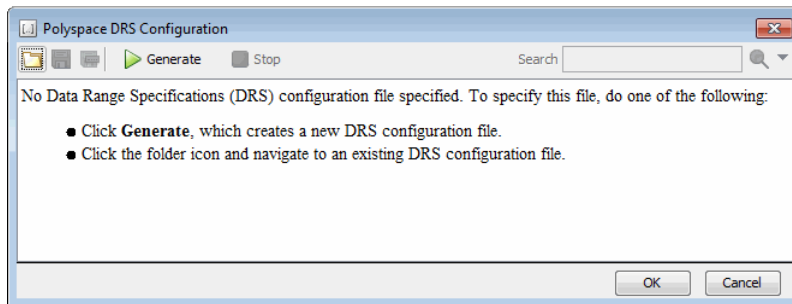
Specify Data Ranges Using Existing DRS Configuration


Once you have created a DRS configuration file for a project, you can reuse the data ranges for subsequent verifications.

To specify an existing DRS configuration file for your project:

- 1 Open the project for which you want to set data ranges.
- 2 In the Project Manager perspective, select the **Configuration > Inputs & Stubbing** node.
- 3 To the right of the **Variable/function range setup** field, click the **Edit** button.

The Polyspace DRS Configuration dialog box opens.



- 4 On the toolbar, click the button .
- 5 In the **Load a DRS file** dialog box, navigate to the folder that contains the required DRS configuration file, and select the file. Then click **Open**. The Load a DRS file dialog box closes.
- 6 In the **Polyspace DRS Configuration dialog** box, click **OK**.
- 7 Select **File > Save Project** to save your project settings, including the DRS file location.

The software uses the specified DRS configuration file the next time you start a verification.

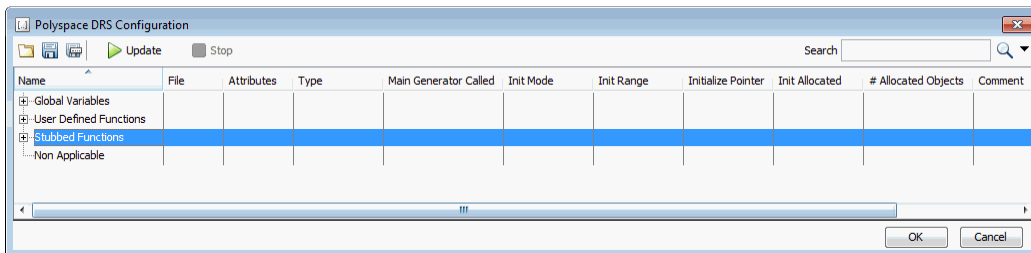
Edit Existing DRS Configuration


Once you have created a DRS configuration file for your project, you can edit the configuration using the Polyspace DRS Configuration dialog box.

To edit an existing DRS configuration:


- 1 Open the project.
- 2 In the Project Manager perspective, select the **Configuration > Inputs & Stubbing** node.
- 3 To the right of the **Variable/function range setup** field, click the **Edit** button.

The Polyspace DRS Configuration dialog box opens.



- 4 Specify the data ranges for global variables, user-defined function inputs, and stub-function return values.
- 5 To save your DRS configuration file, click  (Save DRS),
- 6 Click **OK**, which closes the Polyspace DRS Configuration dialog box.

Remove Non Applicable Entries from DRS File

If you change your source code, you must update your DRS configuration file. From the Polyspace DRS Configuration dialog box, click . The software updates the file, placing DRS entries that no longer apply to your code under the **Non Applicable** node.

You can remove:

- Entries that do not apply:
 - 1 Right-click **Non Applicable**.
 - 2 From the context menu, select **Remove This Node**.
- Entries corresponding to a subnode:
 - 1 Right-click the subnode, for example, **Non_Infinite_loop()**.
 - 2 From the context menu, select **Remove This Node**.

Specify Data Ranges Using Text Files

To use the DRS feature, you must provide a list of variables (or functions) and their associated data ranges.

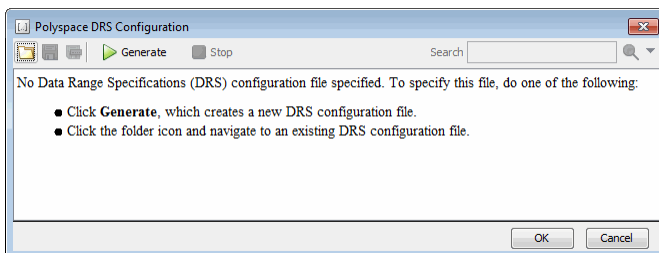
You can specify data ranges using the Polyspace DRS Configuration dialog box (see “Specify Data Ranges Using DRS Template” on page 6-56), or you can provide a text file that contains a list of variables and data ranges.


Note If you used the DRS feature prior to R2010a, you created a text file to specify data ranges. The format of this file has not changed. You can use your existing DRS text file to specify data ranges.

To specify data ranges using a DRS text file:


- 1** Create a DRS text file containing the list of global variables (or functions) and their associated data ranges, as described in “DRS Text File Format” on page 6-62.
- 2** Open the project.
- 3** In the Project Manager perspective, select the **Configuration > Inputs & Stubbing** node.
- 4** To the right of the **Variable/function range setup** field, click the **Edit** button.

The Polyspace DRS Configuration dialog box opens.



- 5 On the toolbar, click the button .
- 6 Navigate to the folder that contains the required DRS text file, and select the file. Then click **Open**.
- 7 In the Polyspace DRS Configuration dialog box, click **OK**.
- 8 Select **File > Save Project** to save your project settings, including the DRS file location.

When you run a verification, the software automatically merges the data ranges in the text file with a DRS template for the project and saves the information in the file `drs-template.xml`, located in your results folder.

Note You can convert your text file to an XML file. On the toolbar of the Polyspace DRS Configuration dialog box, click the button . The software generates an XML version of your text file, which you can edit without affecting your original text file.

DRS Text File Format

The DRS file contains a list of global variables and associated data ranges. The point during verification at which the range is applied to a variable is controlled by the mode keyword: `init`, `permanent`, or `globalassert`.

The DRS file must have the following format:

```
variable_name min_value max_value <init|permanent|globalassert>
```

```
function_name.return min_value max_value permanent
```

- *variable_name* — The name of the global variable.
- *min_value* — The minimum value for the variable.
- *max_value* — The maximum value for the variable.
- `init` — The variable is assigned to the specified range only at initialization, and keeps it until first write.

- `permanent` — The variable is permanently assigned to the specified range. If the variable is assigned outside this range during the program, no warning is provided. Use the `globalassert` mode if you need a warning.
- `globalassert` — After each assignment, an assert check is performed, controlling the specified range. The assert check is also performed at global initialization.
- `function_name` — The name of the stub function.

Tips for Creating DRS Text Files

- You can use the keywords "min" and "max" to denote the minimum and maximum values of the variable type. For example, for the type `long`, min and max correspond to -2^{31} and $2^{31}-1$ respectively.
- You can use hexadecimal values. For example, `x 0x12 0x100 init`.
- Supported column separators are tab, comma, space, or semicolon.
- To insert comments, use shell style "#".
- `init` is the only mode supported for user-defined function arguments.
- `permanent` is the only mode supported for stub function output.
- Function names may be C or C++ functions with blanks or commas. For example, `f(int, int)`.
- Function names can be specified in the short form ("f") as long as no ambiguity exists.
- The function returns either an integral (including enum and bool) or floating point type. If the function returns an integral type and you specify the range as a floating point [`v0.x`, `v1.y`], the software applies the integral interval [`(int)v0-1`, `(int)v1+1`].

Example DRS Text File

In the following example, the global variables are named `x`, `y`, `z`, `w`, and `v`.

```
x 12 100    init
y 0  10000 permanent
z 0  1     globalassert
w min max  permanent
```

```
v 0 max globalassert
arrayOfInt -10 20 init
s1.id 0 max init
array.c2 min 1 init
car.speed 0 350 permanent
bar.return -100 100 permanent

# x is defined between [12;100] at initialization
# y is permanently defined between [0,10000] even any assignment
# z is checked in the range [0;1] after each assignment
# w is volatile and full range on its declaration type
# v is positive and checked after each assignment.
# All cells arrayOfInt are defined between [-10;20] at initialization
# s1.id is defined between [0;2^31-1] at initialisation.
# All cells array[i].c2 are defined between [-2^31;1] at initialization
# Speed of Struct car is permanently defined between 0 and 350 Km/h
# function bar returns -100..100
```

Perform Efficient Module Testing with DRS

DRS allows you to perform efficient static testing of modules. This is accomplished by adding design level information missing in the source-code.

A module can be seen as a black box having the following characteristics:

- Input data are consumed
- Output data are produced
- Constant calibrations are used during black box execution influencing intermediate results and output data.

Using the DRS feature, you can define:

- The nominal range for input data
- The expected range for output data
- The generic specified range for calibrations

These definitions then allow Polyspace software to perform a single static verification that performs two simultaneous tasks:

- answering questions about robustness and reliability
- checking that the outputs are within the expected range, which is a result of applying black-box tests to a module

In this context, you assign DRS keywords according to the type of data (inputs, outputs, or calibrations).

Type of Data	DRS Mode	Effect on Results	Why?	Oranges	Selectivity
Inputs (entries)	permanent	Reduces the number of oranges, (compared with a standard Polyspace verification)	Input data that were full range are set to a smaller range.	↓	↑
Outputs	globalassert	Increases the number of oranges, (compared with a standard Polyspace verification)	More verification is introduced into the code, resulting in both more orange checks and more green checks.	↑	→
Calibration	init	Increases the number of oranges, (compared with a standard Polyspace verification)	Data that were constant are set to a wider range.	↑	↓

Reduce Oranges with DRS

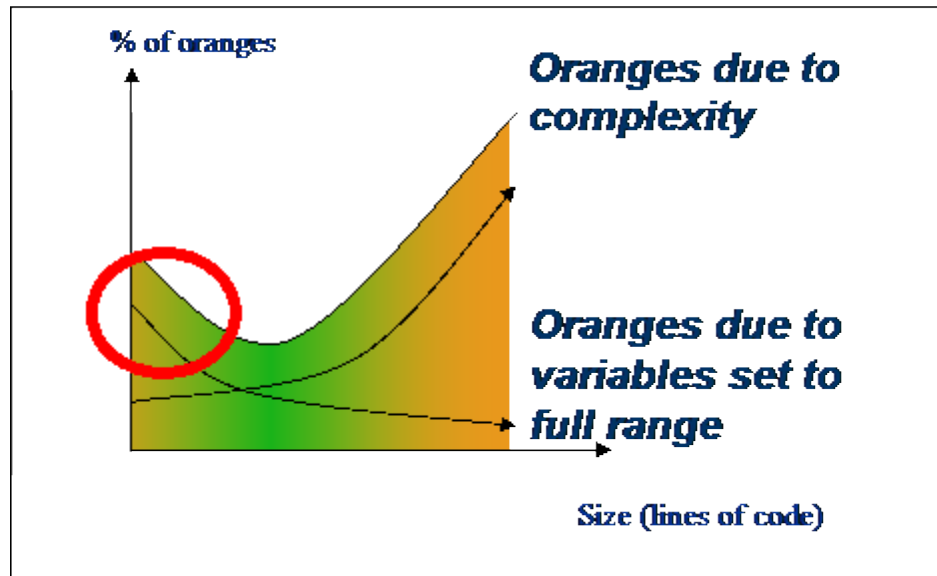
When performing robustness (worst case) verification, data inputs are set to their full range. Therefore, every operation on these inputs, even a simple `one_input + 10` can produce an overflow, as the range of `one_input` varies between the minimum and the maximum of the type.

If you use DRS to restrict the range of `one_input` to the real functional constraints found in its specification, design document, or models, you can reduce the number of orange checks reported for the variable. For example, if you specify that `one_input` can vary between 0 and 10, Polyspace software knows that:

- `one_input + 100` never overflows
- The results of this operation is always between 100 and 110

This not only eliminates the local overflow orange check, but also results in more accuracy in the data. This accuracy is then propagated through the rest of the code.

Using DRS removes the oranges located in the red circle below.



Why Is DRS Most Effective on Module Testing?

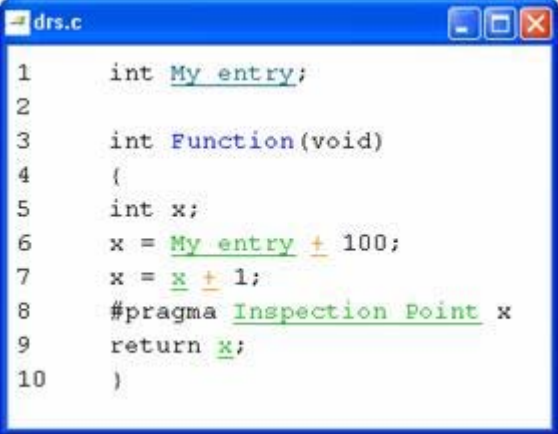
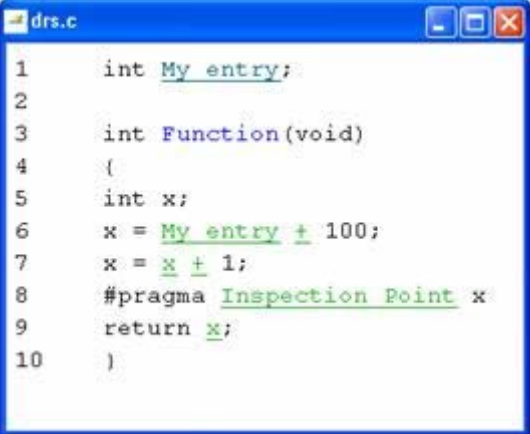
Removing oranges caused by full-range (worst-case) data can drastically reduce the total number of orange checks, especially when used on verifications of small files or modules. However, the number of orange checks caused by code complexity is not effected by DRS. For more information on oranges caused by code complexity, see “Subdivide Code” on page 9-66.

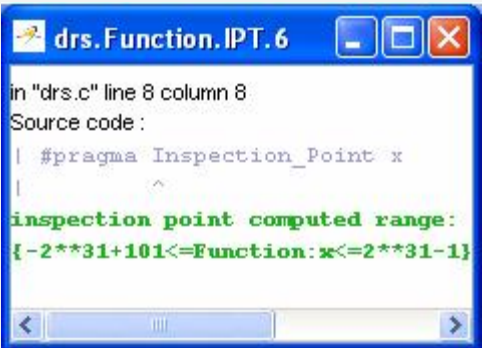
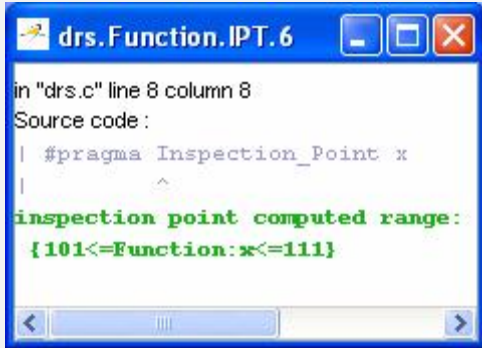
This section describes how DRS reduces oranges on files or modules only.

Example

The following example illustrates how DRS can reduce oranges. Suppose that in the real world, the input “My_entry” can vary between 0 and 10.

Polyspace verification produces the following results: one with DRS and one without.

Without DRS	With DRS – 2 Oranges Removed + Return Statement More Accurate
 <pre> 1 int My_entry; 2 3 int Function(void) 4 { 5 int x; 6 x = My_entry + 100; 7 x = x + 1; 8 #pragma Inspection Point x 9 return x; 10 } </pre>	 <pre> 1 int My_entry; 2 3 int Function(void) 4 { 5 int x; 6 x = My_entry + 100; 7 x = x + 1; 8 #pragma Inspection Point x 9 return x; 10 } </pre>
<ul style="list-style-type: none"> • With “<i>My_entry</i>“ being full range, the addition “+” is orange, • the result “x” is equal to all values between [min+100 max] • Due to previous computations, x+1 can here overflow too, making the addition “+”orange. 	<ul style="list-style-type: none"> • With “<i>My_entry</i>” being bounded to [0,10], the addition “+” is green • the result “x” is equal to [100,110] • Due to previous computations, x+1 can NOT overflow here, making the addition “+” green again.

Without DRS	With DRS – 2 Oranges Removed + Return Statement More Accurate
And the returned result is between [min+101 max]	And the returned result is between [101,111]
 <p>The screenshot shows a debugger window titled "drs.Function.IPT.6". The text inside reads: "in 'drs.c' line 8 column 8", "Source code:", and a code snippet with a pragma. Below the code, it says "inspection point computed range:" followed by the range <code>{-2**31+101<=Function:x<=2**31-1}</code>.</p>	 <p>The screenshot shows a debugger window titled "drs.Function.IPT.6". The text inside reads: "in 'drs.c' line 8 column 8", "Source code:", and a code snippet with a pragma. Below the code, it says "inspection point computed range:" followed by the range <code>{101<=Function:x<=111}</code>.</p>

DRS Configuration Settings

The Polyspace DRS Configuration dialog box allows you to specify data ranges for global variables, user-defined functions, and stub functions in your project.

Column	Settings
Name	<p>Displays the list of variables and functions in your Project for which you can specify data ranges. This Column displays three expandable menu items:</p> <ul style="list-style-type: none"> • Globals – Displays a list of global variables in the Project. • User defined functions – Displays a list of user-defined functions in the Project. Expand a function name to see a list of the input arguments for which you can specify a data range. • Stubbed functions – Displays a list of stub functions in the Project. Expand a function name to see a list of the return values for which you can specify a data range.
File	Displays the name of the source file containing the variable or function.
Attributes	Displays information about the variable or function. For example, static variables display <code>static</code> .
Type	Displays the variable type.
Main Generator Called	<p>Applicable only for user-defined functions. Specifies whether the main generator calls the function:</p> <ul style="list-style-type: none"> • MAIN GENERATOR – Main generator may call this function, depending on the value of the <code>-functions-called-in-loop (C)</code> or <code>-main-generator-calls (C++)</code> parameter. • NO – Main generator will not call this function. • YES – Main generator will call this function.

Column	Settings
Init Mode	<p>Specifies how the software assigns a range to the variable:</p> <ul style="list-style-type: none"> • MAIN GENERATOR – Variable range is assigned depending on the settings of the main generator options <code>-variables-written-before-loop</code> and <code>-no-def-init-glob</code>. (For C++, the options are <code>-main-generator-writes-variables</code>, and <code>-no-def-init-glob</code>.) • IGNORE – Variable is not assigned to any range, even if a range is specified. • INIT – Variable is assigned to the specified range only at initialization, and keeps the range until first write. • PERMANENT – Variable is permanently assigned to the specified range. If the variable is assigned outside this range during the program, no warning is provided. Use the <code>globalassert</code> mode if you need a warning. <p>User-defined functions support only INIT mode.</p> <p>Stub functions support only PERMANENT mode.</p> <p>For C verifications, global pointers support MAIN GENERATOR, IGNORE, or INIT mode.</p> <ul style="list-style-type: none"> • MAIN GENERATOR – Pointer follows the options of the main generator. • IGNORE – Pointer is not initialized • INIT – Specify if the pointer is <code>NULL</code>, and how the pointed object is allocated (Initialize Pointer and Init Allocated options).

Column	Settings
Init Range	<p>Specifies the minimum and maximum values for the variable. You can use the keywords <code>min</code> and <code>max</code> to denote the minimum and maximum values of the variable type. For example, for the type <code>long</code>, <code>min</code> and <code>max</code> correspond to -2^{31} and $2^{31}-1$ respectively.</p> <p>You can also use hexadecimal values. For example: <code>0x12..0x100</code></p>
Initialize Pointer	<p>Applicable only to pointers. Enabled only when you specify Init Mode:INIT.</p> <p>Specifies whether the pointer should be NULL:</p> <ul style="list-style-type: none"> • May-be NULL – The pointer could potentially be a NULL pointer (or not). • Not Null – The pointer is never initialized as a null pointer. • Null – The pointer is initialized as NULL. <hr/> <p>Note Not applicable for C++ projects.</p>
Init Allocated	<p>Applicable only to pointers. Enabled only when you specify Init Mode:INIT.</p> <p>Specifies how the pointed object is allocated:</p> <ul style="list-style-type: none"> • MAIN GENERATOR – The pointed object is allocated by the main generator. • None – Pointed object is not written. • SINGLE – Write the pointed object or the first element of an array. (This setting is useful for stubbed function parameters.) • MULTI – All objects (or array elements) are initialized. <p>See Pointer Examples on page 6-74.</p>

Column	Settings
	<hr/> <p>Note Not applicable for C++ projects.</p> <hr/>
# Allocated Objects	<p>Applicable only to pointers. Specifies how many objects are pointed to by the pointer (the pointed object is considered as an array).</p> <p>Note: The Init Allocated parameter specifies how many allocated objects are actually initialized. See Pointer Examples on page 6-74.</p> <hr/> <p>Note Not applicable for C++ projects.</p> <hr/>
Global Assert	<p>Specifies whether to perform an assert check on the variable at global initialization, and after each assignment.</p>
Global Assert Range	<p>Specifies the minimum and maximum values for the range you want to check.</p>
Comment	<p>Remarks that you enter, for example, justification for your DRS values.</p>

Pointer Examples

For pointer p, **# Allocated objects** = 1, and **Init Allocated** = Single:

```
void f(int *p) {
    int x;
    x = p[0]; // green IDP, green NIV
    x = p[1]; // red IDP: out of bounds
}
```

Note Pointer p may point to any element inside the array.

For pointer `p` (a pointer to `int`), **# Allocated objects** = 3, and **Init Allocated** = MULTI:

```
void f(int *p) {  
    int x;  
    x = p[0]; // green IDP, green NIV  
    x = p[1]; // orange IDP, green NIV  
    x = p[2]; // orange IDP, green NIV  
    x = p[3]; // red IDP: out of bounds  
}
```

Variable Scope

DRS supports variables with external linkages, const variables, extern variables, and defined variables.

Note If you set a data range on a const global variable that is used in another variable declaration (for example as an array size) the variable using the global variable ranged, is not ranged itself.

The following table summarizes possible uses:

	init	permanent	globalassert	comments
Integer	Ok	Ok	Ok	char, short, int, enum, long and long long If you define a range in floating point form, rounding is applied.
Real	Ok	Ok	Ok	float, double and long double If you define a range in floating point form, rounding is applied.
Volatile	No effect	Ok	Full range	Only for int and real

	init	permanent	globalassert	comments
Structure field	Ok	Ok	Ok	Only for int and real fields, including arrays or structures of int or real fields (see below)
Structure field in array	Ok	No effect	No effect	Only when leaves are int or real. Moreover the syntax is the following: <array_name>. <field_name>
Array	Ok	Ok	Ok	Only for int and real fields, including structures or arrays of integer or real fields (see below)
Pointer	Ok (for C) No effect for C++	No effect	No effect	For C, you can specify how the main generator initializes the pointed variable, and how the pointed object is written.
Union field	Ok	No effect	Ok	See “DRS Support for Union Members” on page 6-78.
Complete structure	No effect	No effect	No effect	

	init	permanent	globalassert	comments
Array cell	No effect	No effect	No effect	Example: array[0], array[10] ...
User-defined function arguments	Ok	No effect	No effect	Main generator calls the function with arguments in the specified range
Stubbed function return	No effect	Ok	No effect	Stubbed function returning integer or floating point

Every variable (or function) and associated data range will be written in the log file during the compile phase of verification. If Polyspace software does not support the variable, a warning message is displayed.

Note If you use DRS to set a data range on a const global variable that is used in another variable declaration (for example as an array size), the variable that uses the global variable you ranged is not ranged itself.

DRS Support for Structures

DRS can initialize arrays of structures, structures of arrays, etc., as the long as the last field is explicit (structures of arrays of integers, for example).

However, DRS cannot initialize a structure itself — you can only initialize the fields. For example, "s.x 20 40 init" is valid, but "s 20 40 init" is not (because Polyspace software cannot determine what fields to initialize).

DRS Support for Union Members

In init mode, the software applies the last range in DRS to the union members at the given offset.

In `globalassert` mode, the software checks every `globalassert` in DRS for a given offset within the union at every assignment to the union variable at that offset.

For example:

```
union position {  
  int  sunroof;  
  int  window;  
  int  locks;  
} positionData;
```

DRS:

```
positionData.sunroof 0 100 globalassert  
positionData.window -100 0 globalassert  
positionData.locks -1 1 globalassert
```

An assignment to `positionData.locks` (or other members) will perform assertion checking on the ranges 0 to 100, -100 to 0, and -1 to 1.

XML Format of DRS File

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 6-85.
- **(*****)** — A sub-element of a pointer (i.e. a pointed object) will be taken into account only if `init_pointed` is equal to `SINGLE` or `MULTI`.

<file> Element

Field	Syntax
name	<i>filepath_or_filename</i>
comment	<i>string</i>

<scalar> Element

Field	Syntax
name (**)	<i>name</i>
line (*)	<i>line</i>

Field	Syntax
base_type (*)	intx uintx floatx
Attributes (***)	volatile extern static const
complete_type (*)	<i>type</i>
init_mode	MAIN_GENERATOR IGNORE INIT PERMANENT disabled unsupported
init_modes_allowed (*)	<i>single value (****)</i>
init_range	<i>range</i> disabled unsupported
global_assert	YES NO disabled unsupported
assert_range	<i>range</i> disabled unsupported
comment(*)	<i>string</i>

<pointer> Element

Field	Syntax
Name (**)	<i>name</i>
line (*)	<i>line</i>

Field	Syntax
Attributes (***)	volatile extern static const
complete_type (*)	<i>type</i>
init_mode	MAIN_GENERATOR IGNORE INIT PERMANENT disabled unsupported
init_modes_allowed (*)	<i>single value (****)</i>
initialize_pointer	May be: NULL Not NULL NULL
number_allocated	<i>single value</i> disabled unsupported
init_pointed	MAIN_GENERATOR NONE SINGLE MULTI disabled
comment	<i>string</i>

<array> and <struct> Elements

Field	Syntax
Name (**)	<i>name</i>
line (*)	<i>line</i>
complete_type (*)	<i>type</i>

Field	Syntax
attributes (***)	volatile extern static const
comment	<i>string</i>

<function> Element

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

Valid Modes and Default Values

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
Global variables	Base type	Unqualified/ static/ const scalar	MAIN_ GENERATOR IGNORE INIT PERMANENT	YES NO			Main generator 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			
		Pointed extern scalar	INIT	un- supported			INIT min..max
		Pointed other scalars	MAIN_ GENERATOR INIT	un- supported			MAIN_ GENERATOR dependant

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default	
		Pointed pointer	MAIN_GENERATOR INIT/	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	MAIN_GENERATOR dependant	
		Pointed function	un-supported	un-supported				
Function parameters	Userdef function	Scalar parameters	MAIN_GENERATOR INIT	un-supported			INIT min..max	
		Pointer parameters	MAIN_GENERATOR INIT	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI	
		Other parameters	Refer to parameter type					
	Stubbed function	Scalar parameter	disabled		un-supported			
		Pointer parameters	disabled			disabled	NONE SINGLE MULTI	MULTI
		Pointed parameters	PERMANENT		un-supported			PERMANENT min..max
		Pointed const parameters	disabled		un-supported			
Function return	Userdef function	Return	disabled	un-supported	disabled	disabled		
	Stubbed function	Scalar return	PERMANENT	un-supported			PERMANENT min..max	
		Pointer return	PERMANENT		un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	PERMANENT May be NULL max MULTI

Preparing Source Code for Verification

- “Stubbing Overview” on page 7-3
- “When to Provide Function Stubs” on page 7-4
- “Manual stubs” on page 7-5
- “Provide Stubs for Functions” on page 7-6
- “Stubbing Examples” on page 7-7
- “Automatic Stubbing Behavior for C++ Pointer/Reference” on page 7-10
- “Specify Functions to Stub Automatically” on page 7-12
- “Constrain Data with Stubbing” on page 7-14
- “Default and Alternative Behavior for Stubbing” on page 7-20
- “Function Pointer Cases” on page 7-22
- “Stub Functions with Variable Argument Number” on page 7-23
- “Stub Standard Library Functions” on page 7-25
- “Check Variable Ranges with `assert`” on page 7-26
- “Check Global Variable Ranges with Global Assert” on page 7-27
- “Model Variables External to Application” on page 7-29
- “External Variables” on page 7-30
- “Volatile Variables” on page 7-31
- “Absolute Addresses” on page 7-32
- “Data Rules” on page 7-33

- “Definitions and Declarations” on page 7-34
- “Prepare Code for Built-In Functions” on page 7-35
- “Prepare Multitasking Code” on page 7-38
- “Comment Code for Known Defects” on page 7-54
- “Comment Syntax for Marking Known Defects” on page 7-58
- “Check Acronyms for Code Comments” on page 7-62
- “Types Promotion” on page 7-64
- “Ignoring Assembly Code” on page 7-67
- “Loss of Precision Using `memset` and `memcpy`” on page 7-75
- “Avoid `memset` and `memcpy` for Structure Initialization” on page 7-76

Stubbing Overview

A function stub is a piece of code that models a function whose body is not provided during verification.

Stubs need not model the details of functions. Depending on your requirements, you can:

- Provide the function argument types and return types.
- Provide a bound on the function arguments and return values.
- Provide other details about how the function relates to the rest of the code.

Stubbing allows you to verify code before functions are developed. The more closely your stub models the actual function, the more precise the verification results will be.

Unless you specify the option **Inputs & Stubbing > No automatic stubbing**, Polyspace automatically stubs undefined functions.

When to Provide Function Stubs

By default, Polyspace software automatically stubs undefined functions. Stub functions manually when:

- You note that the automatic stubs do not represent your function arguments and return values. For instance, the automatic stubs can return a broader range of values than you want.
- You want your source code to be complete. If you specify the option **No automatic stubbing**, verification stops if a function is not defined. This behavior allows you to detect undefined functions.
- You want to reduce unproven code. Sometimes, automatic stubs do not provide sufficient information to allow Polyspace to prove presence or absence of run-time errors.
- Your function modifies global variables. Automatic stubs cannot model this behavior.

Manual stubs

If a function `func` represents:

- A timing constraint such as a timer set/reset, a task activation, a delay, or a counter of ticks between two precise locations in the code, stub `func` with an empty action

```
void func(void) {  
}
```

Polyspace takes into account scheduling and interleaving of concurrent execution. Therefore, do not stub functions that set or reset a timer. Declare the variable representing time as volatile.

- An I/O access, such as to a hardware port, a sensor, a read/write of a file, a read of an EEPROM, or a write to a volatile variable, then,
 - You do not need to stub a *write* access. If you want to do so, stub a write access to an empty action (`void func(void)`).
 - Stub *read* accesses to "read all possible values (volatile)".
- A write to a global variable, you may need to consider which procedures or functions write to `func` and why. Do not stub the concerned `func` if:
 - The variable is volatile.
 - The variable is a task list. Such lists are accounted for by default because tasks declared with the `-task` option are automatically modelled as though they have been started. Write `func` manually if:
 - The variable is a regular variable read by other procedures or functions.
 - The variable is a read from a global variable. If you want Polyspace software to detect that the variable is a shared variable, stub a read access. Copy the value into a local variable.

Provide Stubs for Functions

The following example shows a header for a missing function (which might occur, for example, if the code is a subset of a project). The missing function copies the value of the `src` parameter to `dest` so there would be a division by zero, a run-time error..

```
void main(void)
{
    a = 1;
    b = 0;
    a_missing_function(&a, b);
    b = 1 / a;
}
```

Due to the reliance on the software's default stub, the division is shown with an orange warning because `a` is assumed to be anywhere in the full permissible integer range (including 0). If the function is commented out, then the division would be a green `/`. You could only achieve a red `/` with a manual stub.

Default Stubbing	Manual Stubbing	Function Ignored
<pre>void main(void) { a = 1; b = 0; a_missing_function(&a, b); b = 1 / a; // orange division }</pre>	<pre>void a_missing_function (int *x, int y;) { *x = y; } void main(void) { a = 1; b = 0; a_missing_function(&a, b); b = 1 / a; // red division</pre>	<pre>void a_missing_function (int *x, int y;) { } void main(void) { a = 1; b = 0; a_missing_function(&a, b); b = 1 / a; // green division</pre>

Due to the reliance on the software's default stub, the software ignores the assembly code and the division `/` is green. You could only achieve the red division `/` with a manual stub.

Stubbing Examples

The following examples consider the pros and cons of manual and automatic stubbing.

Example: Specification

```
typedef struct _c {
    int cnx_id;
    int port;
    int data;
} T_connection ;

int Lib_connection_create(T_connection *in_cnx) ;
int Lib_connection_open (T_connection *in_cnx) ;
```

File: connection_lib		Function: Lib_connection_create
param in	None	
param in/out	in_cnx	all fields might be changed in case of a success
returns	int	0 : failure of connection establishment 1 : success

Note Default stubbing is suitable here.

Here are the reasons why:

- The content of the *in_cnx* structure might be changed by this function.
- The possible return values of 0 or 1 compared to the full range of an integer wont have much impact on the Run-Time Error aspect. It is unlikely that the results of this operation will be used to compute some mathematical algorithm. It is probably a Boolean status flag and if so is likely to be stored and compared to 0 or 1. Therefore, the default stub does not have a detrimental effect.

File: connection_lib		Function: Lib_connection_open
param in	T_connection *in_cnx	in_cnx->cnx_id is the only parameter used to open the connection, and is a read-only parameter. cnx_id, port and data remain unchanged
param in/out	None	
returns	int	0 : failure of connection establishment 1 : success

Note Default stubbing works here but manual stubbing would give more benefit.

Here are the reasons why:

- For the return value, default stubbing would be applicable as explained in the previous example.
- Since the structure is a read-only parameter, it will be worth creating manually a stub that reflects the behavior of the missing code. Benefits: Polyspace verification will find more red and gray code

Note Even in the examples above, it concerns some C code like; stubs of functions members in classes follow same behavior.

Example: Colored Source Code

```
1     typedef struct _c {
2         int a;
3         int b;
4     } T;
5
6     void send_message(T *);
7     void main(void)
8     {
```



```
9     int i;
10    T x = {10, 20};
11    send_message(&x);
12    i = x.b /x.a; // orange with the default stubbing
13    }
```

Suppose that it is known that `send_message` does not write into its argument. The division by `x.a` will be orange if default stubbing is used, warning of a potential division by zero. A manual stub that accurately reflects the behavior of the missing code will result in a green division instead, thus increasing the selectivity.

Manual stubbing examples for `send_message`:

```
void send_message(T *) {}
```

In this case, an empty function would be a sound manual stub.

Automatic Stubbing Behavior for C++ Pointer/Reference

For parameters of a pointer/reference type, the behavior of automatically stubbed C++ functions differs from the behavior of automatically stubbed C functions. As a result, automatic stubs for C++ do not always write to their arguments.

For C++, the software stubs functions by randomizing the contents of the object passed as actual of the stubbed function, but does not modify the object pointed to by the actual (or by one component of the actual if the latter is a struct/class object or an array).

Consider the following example:

```
extern void stub_def_pointer(struct S *p);
extern void stub_def_array(struct S *p);

int fx = 0, fw = 0;
struct S def = {"-dummy", &fx};
struct S def_array[] = {{ "-foo", &fw } };

assert(*(def.pvar) == 0); // GREEN
stub_def_pointer(&def);
assert(fx == 0);          // GREEN because stubbed stub_def_pointer
                          // does not write *(def.pvar)

assert(*(def_array[0].pvar) == 0); // GREEN
stub_def_array(def_array);
assert(fw == 0);          // GREEN because stubbed stub_def_array
                          // does not write *(def_array[0].pvar)
```

In this situation, you should manually stub the missing routine. For example, you could stub `stub_def_pointer` and `stub_def_array` as follows:

```
volatile int rd;

void stub_def_pointer(struct S *p)
{
    *(p->pvar) = rd; // write the object pointed to by p->pvar
}
```

```
void stub_def_array(struct S *p)
{
    int i = rd;
    for (i; i < rd; i++)
    {
        *(p[i].pvar) = rd; // write the object pointed to
                          // by p[i]->pvar
        i++;
    }
}
```

Using these manual stubs, the verification result become:


```
assert(*(def.pvar) == 0);           // GREEN
stub_def_pointer(&def);
assert(fx == 0);                    // ORANGE

assert(*(def_array[0].pvar) == 0); // GREEN
stub_def_array(def_array);
assert(fw == 0);                    // ORANGE
```

Specify Functions to Stub Automatically

You can specify a list of functions that you want the software to stub automatically.

To specify functions to stub:

- 1 In the Project Manager perspective, select the **Configuration > Code Prover Verification > Inputs & Stubbing** pane.
- 2 To the right of the **Functions to stub** view, click . The software creates a new row.
- 3 In the new row, enter the name of a function that you want to stub.
- 4 For each additional function, repeat steps 2 and 3.
- 5 Save your project.

Special Characters in Function Names

The following special characters are allowed for C functions:

() < > ; _

The following special characters are allowed for C++:

() < > ; _ * & []

Space characters are allowed for C++, but are not allowed for C functions.

Function Syntax for C++

When entering function names, two syntaxes are supported for C++:

- Basic syntax, with extensions for classes and templates:

Function Type	Syntax
Simple function	test
Class method	A::test
Template method	A<T>::test

- Syntax with function arguments, to differentiate overloaded functions. Function arguments are separated with semicolons:

Function Type	Syntax
Simple function	<code>test()</code>
Class method	<code>A::test(int;int)</code>
Template method	<code>A<T>::test(T;T)</code>

Note Overloaded versions of the function will be discarded.

Constrain Data with Stubbing

In this section...
“Add Precision Constraints Using Stubs” on page 7-14
“Default Behavior of Global Data” on page 7-15
“Constraining the Data” on page 7-16
“Apply the Technique” on page 7-16
“Integer Example” on page 7-16
“Recode Specific Functions” on page 7-17

Add Precision Constraints Using Stubs

You can improve the selectivity of your verification by using stubs to indicate that some variables vary within functional ranges instead of the full range of the considered type.

You can apply this approach to:

- Parameters passed to functions.
- Variables that change from one execution to another (mostly globals), for example, calibration data or mission specific data. These variables might be read directly within the code, or read through an API of functions.

If a function returns an integer, default automatic stubbing assumes the function can take any value from the full range of the integer type. This can lead to unproven code (orange checks) in your results. You can achieve more precise results by providing a manual stub that provides external data that is representative of the data expected when the code is implemented.

There are a number of ways to model such data ranges within the code. The following table shows some approaches.

with volatile and assert	with assert and without volatile	without assert, without volatile, without "if"
<pre>#include <assert.h> int stub(void) { volatile int random; int tmp; tmp = random; assert(tmp>=1 && tmp<=10); return</pre>	<pre>#include <assert.h> extern int other_func(void); int stub(void) { int tmp; tmp= other_func(); assert(tmp>=1 && tmp<=10); return }</pre>	<pre>extern int other_func(void); int stub(void) { int tmp; do {tmp= other_func();} while (tmp<1 tmp>10); return tmp; }</pre>

There is no particular advantage to any one of these approaches, except that the assertions in the first two approaches can produce orange checks in your results.

Default Behavior of Global Data

Initially, consider how Polyspace verification handles the verification of global variables.

There is a maximum range of values which may be assigned to each variable as defined by its type. By default, Polyspace verification assigns that full range for each global variable, ensuring that a meaningful verification of such a variable can take place even when the functions that write to it are not included. If a range of values was not considered in these circumstances, such a variable would be assumed to have a value of zero throughout.

Sometimes, to reflect practical use, it is helpful to limit the range of values assigned to some variables. These ranges will be propagated to the whole call tree, and hence will limit the number of “impossible values” that are considered throughout the verification.

This thinking does not just apply to global variables; it is equally appropriate where such a variable is passed as a parameter to a function, or where return values from stubbed functions are under consideration.

To some extent, the effectiveness of this technique is limited by compromises made by Polyspace verification to deal with issues of code complexity. For instance, you cannot assume that all of these ranges will be propagated throughout all function calls. Sometimes, perhaps as a result of complex function interactions or constructions where Polyspace verification is known to be imprecise, the potential value of a variable will assume its full “type” range despite this technique having been applied.

Constraining the Data

Restricting data, such as global variables, to a functional range can be a useful technique if the process can be automated. The technique may not be advantageous if the process requires significant manual effort.

The technique requires:

- A knowledge of the variables and the maximum ranges they may take in practice.
- A data dictionary in electronic format from which the variable names and their minimum and maximum values can be extracted.

Apply the Technique

- 1 Create the range setting stubs:
 - a create 6 functions for each type (8,16 or 32 bits, signed and unsigned)
 - b declare 6 global volatile variables for each type
 - c write the functions which returns sub-ranges (an example follows)
- 2 Gather the initialization of relevant variables into a single procedure
- 3 Call this procedure at the beginning of the main. This should replace existing initialization code.

Integer Example

```
volatile int tmp;  
  
int polyspace_return_range(int min_value, int max_value)
```



```
{
int ret_value;

ret_value = tmp;
assert (ret_value>=min_value && ret_value<=max_value);

return ret_value;
}
void init_all(void)
{
x1 = polyspace_return_range(1,10);
x2 = polyspace_return_range(0,100);
x3 = polyspace_return_range(-10,10);
}

void main(void)
{
init_all();

while(1)
{
if (tmp) function1();
if (tmp) function2();
// ...
}
}
```

Recode Specific Functions

Once data ranges have been specified (above), it may be beneficial to recode some functions in support of them.

Sometimes, perhaps as a result of complex function interactions or constructions where Polyspace verification is known to be imprecise, the potential value of a variable will assume its full “type” range data ranges having been restricted. Recoding those complex functions will address this issue.

Identify in the modules:

- API which read global variables through pointers

Replace this API:

```
typedef struct _points {
    int x,y,nb;
    char *p;
}T;

#define MAX_Calibration_Constant_1 7
char Calibration_Constant_1[MAX_Calibration_Constant_1] = \
{ 1, 50, 75, 87, 95, 97, 100} ;
T Constant_1 = { 0, 0,
    MAX_Calibration_Constant_1,
    &Calibration_Constant_1[0] } ;

int read_calibration(T * in, int index)
{
    if ((index <= in->nb) && (index >=0)) return in->p[index];
}

void interpolation(int i)
{
    int a,b;

    a= read_calibration(&Constant_1,i);
}
```

With this one:

```
char Constant_1 ;

#define read_calibration(in,index) *in

void main(void)
{
    Constant_1 = polyspace_return_range(1, 100);
}

void interpolation(int i)
{
    int a,b;
```

```
a= read_calibration(&Constant_1,i);
}
```

- Points in the source code which expand the data range perceived by Polyspace verification
- Functions responsible for full range data, as shown by the VOA (Value on assignment) check.

if direct access to data is responsible, define the functions as macros.

```
#define read_from_data(param) read_from_data##param
```

```
int read_from_data_my_global1(void)
{ return [a functional range for my_global1]; }
```

```
Char read_from_data_my_global2(void)
{ }
```

- stub complicated algorithms, calibration read accesses and API functions reading global data - as usual. For instance, if an algorithm is iterative - stub it.
- variables
 - where the data range held by each element of an array is the same, replace that array with a single variable.
 - where the data range held by each element of an array differs, separate it into discrete variables.

Default and Alternative Behavior for Stubbing

External functions are assumed to have no effect (read, write) on global variables. Any external function for which this assumption is not valid must be explicitly stubbed.

Consider the example `int f(char *)`;

When verifying this function, there are three options for automatic stubbing, as shown in the following table.

Approach	Worst Case Scenario in Stub
Default automatic stubbing	<pre>int f(char *x) { *x = rand(); return 0; }</pre>
<code>pragma POLYSPACE_WORST</code>	<pre>int f(char *x) { strcpy(x, "the quick brown fox, etc."); return &(x[2]); }</pre>
<code>pragma POLYSPACE_PURE</code>	<pre>int f(char *x) { return strlen(x); }</pre>

If the automatic stub does not accurately model the function using any of these approaches, you can use manual stubbing to achieve more precise results.

PURE and WORST Stubbing Examples

The following table provides examples of stubbing approaches.

Initial Prototype	With pragma POLYSPACE_PURE	With pragma POLYSPACE_WORST	Default Automatic Stubbing
<code>void f1(void);</code>	Do nothing		
<code>int f2 (int u);</code>	Returns $[-2^{31}, 2^{31}-1]$	Returns $[-2^{31}, 2^{31}-1]$ and assumes the ability to write into $(int *) u$	Returns $[-2^{31}, 2^{31}-1]$
<code>int f3 (int *u);</code>			Assumes the ability to write into $*u$ to any depth and returns $[-2^{31}, 2^{31}-1]$
<code>int* f4 (int u);</code>	Returns an absolute address (AA)	Returns AA or $(int *) u$ and assumes the ability to write into $(int *) u$	Returns an absolute address
<code>int* f5 (int *u);</code>	Returns an absolute address	Returns $[-2^{31}, 2^{31}-1]$ and assumes the ability to write into $*u$, to any depth	Assumes the ability to write into $*u$, to any depth and returns an absolute address
<code>void f6 (void (*ptr)(int), param2)</code>	Does nothing	The function pointed to by <code>ptr</code> is called with a full-range random value for the integer. Rules for <code>param2</code> are the same as the preceding rules.	
<code>void f7 (void (*ptr)(param2)</code>		Unless you use the option <code>permissive-stubber</code> , this function is not stubbed. The parameter $(int *)$ associated with the function pointer is too complicated for the software to stub it, and verification stops. You must stub this function manually.	
		Note If $(*ptr)$ contains a pointer as a parameter, it is not stubbed automatically and with <code>permissive-stubber</code> , the function pointer <code>ptr</code> is called with <code>random</code> as a parameter.	

Function Pointer Cases

Function Prototype	Comments
<pre>void _reg(int); int _seq(void *); unsigned char bar(void){ return 0; } void main(void){ unsigned char x=0; _reg(_seq(bar)); }</pre>	<p>Both functions, “_reg” and “_seq”, are automatically stubbed, but the Polyspace software does not exercise the call to the bar function.</p> <p>The function that is a parameter is only called in stubbed functions if the stubbed function prototype contains a function pointer as parameter.</p> <p>Because in this example, the stubbed function is a “void*”, it is not a function pointer.</p>

Stub Functions with Variable Argument Number

Polyspace software can stub most vararg functions. However:

- This stubbing can generate imprecision in pointer verification.
- The stubbing causes a significant increase in complexity and in verification time.

There are three ways that you can deal with this stubbing issue:

- Stub manually
- On every varargs function that you know to be pure, add a `#pragma POLYSPACE_PURE "function_1"`. This action reduces greatly the complexity of pointer verification tenfold.

For example:

```
#pragma POLYSPACE_PURE f

void main(void) {
    int x = 0;
    f(&x);
    assert ( x == 0 ); // Green assertion,
                      //orange without use of #pragma POLYSPACE_PURE
}
```

- Use `#define` to eliminate calls to functions. For example, functions like `printf` generate complexity but are not useful for verification because they only display a message.

For example:

```
#ifdef POLYSPACE
#define example_of_function(format, args...)
#else
void example_of_function(char * format, ...)
#endif
void main(void)
{
    int i = 3;
```

```
    example_of_function("test1 %d", i);  
}
```

```
polyspace-code-prover-nodesktop -D POLYSPACE
```

You can place this kind of line in any `.c` or `.h` file of the verification.

Note Use `#define` only with functions that are pure.

Stub Standard Library Functions

Polyspace provides the file `__polyspace__stdstubs.c`, which stubs functions of the C standard library. During a verification, Polyspace uses the function stubs to generate `STD_LIB` checks. These checks indicate whether the arguments of standard library function calls in your code are valid. See [Invalid use of standard library routine](#).

For more information about how you can use `__polyspace__stdstubs.c`, see [“Standard Library Function Stubbing Errors”](#) on page 9-53.

Check Variable Ranges with assert

`assert` is a macro that aborts a program if the test performed inside the `assert` statement is false.

You can use `assert` to constrain input variables to values within a particular range, for example:

```
#include <stdlib.h>

int random(void);

int return_between_bounds(int min, int max)
{
    int ret; // ret is not initialized
    ret = random(); // ret ~ [-2^31, 2^31-1]
    assert ((min<=ret) && (ret<=max));
    // assert is orange because the condition may or may not
    // be fulfilled
    // ret ~ [min, max] here because all execution paths that don't
    // meet the condition are stopped
    return ret;
}
```

Check Global Variable Ranges with Global Assert

Use the **Global Assert** mode to constrain the range of a global variable. In this mode, Polyspace performs a **Correctness condition** check on each write access to the global variable. After the write access, this check determines whether the variable is within the range that you specified.


1 Run verification on your code. Open the results in the Results Manager perspective.

2 On the **Source** pane, select the **Data Range Configuration** tab.

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 on the **Global Assert** column, select YES.

4 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.

5 To save your specifications, click the  button.

A **Save Data Range Specifications (DRS) as** window opens. Save your entries as an xml file.

6 Return to the Project Manager perspective. On the **Configuration** pane, under **Inputs & Stubbing**, in the **Variable/function range setup** field, enter the full path to the xml file.

Instead of typing the location, you can use the  button to navigate to the location of the .xml file.

7 Rerun the verification and open the results.

For every write access 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.

In a multitasking application, when two or more tasks access the same global variable, if a **Correctness condition** check on a write access in one task turns orange, the **Correctness condition** check on write accesses in all other tasks appear orange. The other orange checks appear even if the other write accesses do not take the variable outside the **Global Assert** range.

See Also

“Variable/function range setup” | Correctness condition

Related Examples

- “Specify Data Ranges Using DRS Template” on page 6-56

Concepts

- “DRS Configuration Settings” on page 6-71

Model Variables External to Application

Express external variables using the keywords `volatile` and `extern`.

- A variable defined with keyword `volatile` can have any value allowed by its type. The value can change at any time, even between two successive memory accesses.
- A variable declared with keyword `extern` and not initialized is presumed to be defined elsewhere.

Concepts

- “External Variables” on page 7-30
- “Volatile Variables” on page 7-31

External Variables

Polyspace verification works on the principle that a global or static external variable could take any value within the range of its type.

```
extern int x;  
void f(void)  
int y;  
y = 1 / x; // orange because x ~ [-2^31, 2^31-1]  
y = 1 / x; // green because x ~ [-2^31 -1] U [1, 2^31-1]
```

For more information on color propagation, refer to “Color Sequence of Checks” on page 10-109.

For external structures containing fields of type “pointer to function”, this principle leads to red errors in the verification results. In this case, the resulting default behavior is that these pointers do not point to any valid function. For meaningful results, you need to define these variables explicitly.

Volatile Variables

Polyspace verification assumes that hardware can assign a value to a volatile variable, but will not de-initialize it. Therefore, NIV checks cannot be red.

```
volatile int x; // x ~ [-2^31, 2^31-1], although x has not been initialised
```

- If x is a global variable, the NIV is green.
- If x is a local variable, the NIV is green if x is initialized by the code, and orange if x has not been initialized by the code.

Absolute Addresses

The content of an absolute address is considered to be potentially uninitialized:

```
int y;

void f1(void) {
#define X (*(int *)0x20000)
  X = 100;
  // Orange ABS_ADDR for address of X
  y = 1 / X;
  // Orange division by zero as X is potentially uninitialized
}

void f2(void) {
  int *p = (int *)0x20000;
  // Orange absolute address
  *p = 100;
  y = 1 / *p;
}
```


Data Rules

Data rules are design rules which dictate how modules and/or files interact with each other.

For instance, consider global variables. It is not always apparent which global variables are produced by a given file, or which global variables are used by that file. The excessive use of global variables can lead to problems in a design. For example:

- File APIs (or functions accessible from outside the file) without procedure parameters.
- The requirement for a formal list of variables which are produced and used, as well as the theoretical ranges they can take as input and/or output values.

Definitions and Declarations

The definition and declaration of a variable are two different but related operations.

Definition

- **for a function:** the body of the function has been written: `int f(void) { return 0; }`
- **for a variable:** a part of memory has been reserved for the variable: `int x;` or `extern int x=0;`

When a variable is not defined, the software considers the variable to be initialized, and to have potentially any value in its full range. For more information, see “External Variables” on page 7-30.

When a function is not defined, it is stubbed automatically.

Declaration

- **for a function:** the prototype: `int f(void);`
- **for an external variable:** `extern int x;`

A declaration provides information about the type of the function or variable. If the function or variable is used in a file where it has not been declared, a compilation error results.

Prepare Code for Built-In Functions

In this section...

“Overview” on page 7-35

“Stubs of stl Functions” on page 7-35

“Stubs of libc Functions” on page 7-35

Overview

Polyspace software stubs functions that are not defined within the verification. Polyspace software provides an accurate stub for the functions defined in the `stl` and in the standard `libc`, taking into account functional aspects of the function.

Stubs of stl Functions

Functions of the `stl` are stubbed by Polyspace software. Using the `-no-stl-stubs` option allows deactivating standard `stl` stubs (not recommended for further possible scaling trouble).

Note Allocation functions found in the code to analyze like `new`, `new[]`, `delete` and `delete[]` are replaced by internal and optimized stubs of `new` and `delete`. A warning is given in the log file when such replace occurs.

Stubs of libc Functions

Functions are declared in the standard list of headers. You can redefine these functions by invalidating the associated set of functions and providing new definitions in your code.

To invalidate standard functions, use:

- `-D POLYSPACE_NO_STANDARD_STUBS` for functions declared in Standard ANSI® headers: `assert.h`, `ctype.h`, `errno.h`, `locale.h`, `math.h`, `setjmp.h` (`setjmp` and `longjmp` functions are partially implemented — see *Polyspace_Install/polyspace/verifier/cxx/cinclude/__polyspace__stdstubs.c*)

signal.h (signal and raise functions are partially implemented — see *Polyspace_Install/polyspace/verifier/cxx/cinclude/__polyspace__stdstubs.c*, *stdio.h*, *stdarg.h*, *stdlib.h*, *string.h*, and *time.h*.

- -D POLYSPACE_STRICT_ANSI_STANDARD_STUBS for functions declared only in *strings.h*, *unistd.h*, and *fcntl.h*.

Note You cannot redefine the following functions that deal with memory allocation: `malloc()`, `calloc()`, `realloc()`, `valloc()`, `alloca()`, `__builtin_malloc()`, and `__builtin_alloca()`.

To invalidate a specific function, use -D `__polyspace_no_function_name`.

For example, if you want to redefine the `fabs()` function:

- For the verification, specify the option -D `__polyspace_no_fabs`.
- In the code, provide your `fabs()` function.

If your **Include** folders contain the standard header files `stdio.h` and `string.h`, Polyspace may recognize your function declarations even if they do not exactly match the standard declarations. For example, you might declare `memset` as:

```
void memset ( void * ptr, unsigned int value, size_t num );
```

instead of:

```
void * memset ( void * ptr, int value, size_t num );
```

In this case, a verification does not generate a compilation error.

If your **Include** folders do not contain `stdio.h` and `string.h`, you can activate this Polyspace feature by specifying the option -D `__polyspace_adapt_types_for_stubs`. If your **Include** folders contain `stdio.h` and `string.h` but you want to deactivate the feature, specify the option -D `__polyspace_static_types_for_stubs`.

Note If your function version differs from the standard function, the internal conversion of parameters and return type during verification may cause a loss of precision.

Prepare Multitasking Code

In this section...
“Polyspace Software Assumptions” on page 7-38
“Model Synchronous Tasks” on page 7-39
“Model Interruptions and Asynchronous Events and Tasks” on page 7-41
“Are Interruptions Maskable or Preemptive?” on page 7-43
“Shared Variables” on page 7-45
“Mailboxes” on page 7-49
“Atomicity (Can Instruction be Interrupted by Another?)” on page 7-52
“Priorities” on page 7-53

Polyspace Software Assumptions

This section describes the default behavior of the Polyspace software. If your code does not conform to these assumptions, before starting verification, you must make minor modifications to the code.

The assumptions are:

- The main procedure must terminate for entry-points (or tasks) to start.
- Tasks or entry-points start after the end of the main procedure without a predefined basis regarding the sequence, priority, or preemption. If an entry-point is seen as dead code, it is because the main procedure contains a red error and therefore does not terminate.
- Verification assumes that there is no atomicity, nor timing constraints.
- Only entry points with `void any_name (void)` as prototype are considered.

Read this entire section before applying the rules described. Some rules are mandatory while other rules allow you to gain selectivity.

Model Synchronous Tasks

In some circumstances, you must adapt your source code to allow synchronous tasks to be taken into account.

Suppose that an application has the following behavior:

- Once every 10 ms: `void tsk_10ms(void);`
- Once every 30 ms: ...
- Once every 50 ms

These tasks do not interrupt each other. They include no infinite loops, and return control to the calling context. For example:

```
void tsk_10ms(void)
{ do_things_and_exit();
  /* it's important it returns control*/
}
```

However, if you specify each entry-point at launch using the option:

```
polyspace-c -entry-points tsk_10ms,tsk_30ms,tsk_50ms
```

then the results are not valid, because each task is called only once.

To address this problem, you must specify that the tasks are purely sequential. You can do this by writing a function to call each of the tasks in the right sequence, and then declaring this new function as a single task entry point.

Solution 1

Write a function that calls the cyclic tasks in the right order; an **exact sequencer**. This sequencer is then specified at launch time as a single task entry point.

This solution requires knowledge of the exact sequence of events.

For example, the sequencer might be:

```
void one_sequential_C_function(void)
{
  while (1) {
    tsk_10ms();
    tsk_10ms();
    tsk_10ms();
    tsk_30ms ();
    tsk_10ms();
    tsk_10ms();
    tsk_50ms ();
  }
}
```

and the associated launching command:

```
polyspace-code-prover-nodesktop -entry-points
one_sequential_C_function
```

Solution 2

Make an **upper approximation sequencer**, taking into account every possible scheduling.

This solution is less precise but quick to code, especially for complicated scheduling:

For example, the sequencer might be:

```
void upper_approx_C_sequencer(void)
{
  volatile int random;
  while (1) {
    if (random) tsk_10ms();
    if (random) tsk_30ms();
    if (random) tsk_50ms();
    if (random) tsk_100ms();
    .....
  }
}
```


and the associated launching command:

```
polyspace-code-prover-nodesktop -entry-points  
upper_approx_C_sequencer
```

Note If this is the only entry-point, then it can be added at the end of the main procedure rather than specified as a task entry point.

Model Interruptions and Asynchronous Events and Tasks

You can adapt your source code to allow Polyspace software to consider both *asynchronous* tasks and *interruptions*. For example:

```
void interrupt_isr_1(void)  
{ ... }
```

Without such an adaptation, interrupt service routines appear as gray (dead code) in the Results Manager perspective. The gray code indicates that this code is not executed and is not taken into account, so interruptions and tasks are ignored by the verification.

The standard execution model is such that the main procedure is executed initially. Only if the main procedure terminates and returns control (i.e. if it is not an infinite loop and has no red errors) do the entry points start, with potential starting sequences being modelled automatically. You can adopt several different approaches to implement the required adaptations.

Solution 1: Where Interrupts (ISRs) Cannot Preempt Each Other

If the following conditions are fulfilled:

- The interrupt functions `it_1` and `it_2` (say) can never interrupt each other.
- Each interrupt can be raised several times, at any time.
- The functions are returning functions, and not infinite loops.

Then these non preemptive interruptions may be grouped into a single function, and that function declared as an entry point.

```
void it_1(void);
void it_2(void);

void all_interruptions_and_events(void)
{ while (1) {
  if (random()) it_1();
  if (random()) it_2();
  ... }
}
```

The associated launching command would be:

```
polyspace-code-prover-nodesktop -entry-points
all_interruptions_and_events
```

Solution 2: Where Interrupts Can Preempt Each Other

If two ISRs can each be interrupted by the other, then:

- Encapsulate each of them in a loop.
- Declare each loop as an entry point.

One approach is to replace the original file with a Polyspace version.

```
original_file.c
void it_1(void)
{
  ... return;
}

void it_2(void)
{
  ... return;
}

void one_task(void)
{
  ... return;
}
```

```
polyspace.c
void polys_it_1(void)
{
    while (1)
    if (random())
        it_1();
}

void polys_it_2(void)
{
    while (1)
    if (random())
        it_2();
}

void polys_one_task(void)
{
    while (1)
    if (random())
        one_task();
}
```

The associated launching command would be:

```
polyspace-code-prover-nodesktop -entry-points
polys_it_1,polys_it_2,polys_one_task
```

Are Interruptions Maskable or Preemptive?

For user interruptions, no *implicit* critical section is defined: you must write them manually.

Sometimes, an application which includes interrupts has a critical section written into its main entry point, but shared data is still flagged as unprotected.

This occurs because Polyspace verification does not distinguish between interrupt service routines and tasks. If you specify an interrupt to be a "-entry-points" entry point, it has the same priority level as the other procedures declared as tasks ("-entry-points" option). Polyspace verification

makes an upper approximation of all scheduling and interleaving, which includes the possibility that the ISR might be interrupted by other tasks. More paths modelled than could happen during execution, but this has no adverse effect on of the results obtained except that more scenarios are considered than could happen during “real life” execution - and the shared data is not seen as being protected.

To address this, the interrupt must be embedded in a specific procedure that uses the same critical section as the interrupt used in the main task. Then, each time this function is called, the task will enter a critical section which will model the behavior of a nonmaskable interruption.

Original files:

```
int shared_x ;

void my_main_task(void)
{
    // ...
    MASK_IT;
    shared_x = 12;
    UMASK_IT;
    // ...
}
int shared_x ;

void interrupt my_real_it(void)
{ /* which is by specification unmaskable */
    shared_x = 100;
}
```

Additional C files required by the verification:

```
extern void my_real_it(void); // declaration required

#define MASK_IT pst_mask_it()
#define UMASK_IT pst_unmask_it()

void pst_mask_it(void); // functions to model critical sections
void pst_unmask_it(void); //
```

```
void other_task (void)
{
    MASK_IT;
    my_real_it();
    UMASK_IT;
}
```

The associated launch command:

```
polyspace-code-prover-nodesktop \
-D interrupt= \
-entry-points my_main_task,other_task \
-critical-section-begin "pst_mask_it:table" \
-critical-section-end "pst_unmask_it:table"
```

Shared Variables

When you launch Polyspace without options, tasks are examined as though concurrent and without assumptions about priorities, sequence order, or timing. Shared variables in this context are considered unprotected, and so are shown as orange in the variable dictionary.

The software uses the following explicit protection mechanisms to protect the variables:

- Critical section
- Mutual exclusion
- “Critical Sections” on page 7-45
- “Mutual Exclusion” on page 7-47
- “Semaphores” on page 7-48
- “Effects of Imprecision on Shared Variable List” on page 7-48

Critical Sections

This is the most common protection mechanism found in applications, and is simple to represent in Polyspace software:

- If one entry-point makes a call to a particular critical section, other entry-points are blocked on the "critical-section-begin" function call until the originating entry-point calls the "critical-section-end" function.
- The code between two critical sections is not atomic.
- The code is a binary semaphore, so there is only one token per label (CS1 in the following example). Unlike many implementations of semaphores, it is not a decrementing counter that can keep track of a number of attempted accesses.

Consider the following example:

Original Code

```
void proc1(void)
{
    MASK_IT;
    x = 12; // X is protected
    y = 100;
    UMASK_IT;
}
void proc2(void)
{
    MASK_IT;
    x = 11; // X is protected
    UMASK_IT;
    y = 101; // Y is not protected
}
```

File Replacing the Original Include File

```
void begin_cs(void);
void end_cs(void);
#define MASK_IT begin_cs()
#define UMASK_IT end_cs()
```

Command Line to Launch Polyspace Verification

```
polyspace-code-prover-nodesktop \
    -entry-point proc1,proc2 \
    -critical-section-begin"begin_cs:label_1" \
```

```
-critical-section-end"end_cs:label_1"
```

Mutual Exclusion

You can implement mutual exclusion between tasks or interrupts while preparing to launch verification.

Suppose there are entry-points which never overlap each other, and that variables are shared by nature.

If entry-points are mutually exclusive, i.e. if they do not overlap in time, you may want the verification to take that into account. Consider the following example:

These entry points cannot overlap:

- t1 and t3
- t2, t3 and t4

These entry-points can overlap:

- t1 and t2
- t1 and t4

Before launching verification, the names of mutually exclusive entry-points are placed on a single line:

```
polyspace-code-prover-nodesktop -temporal-exclusion-file myExclusions.txt  
-entry-points t1,t2,t3,t4
```

The file myExclusions.txt is also required in the current folder. This file contains:

```
t1 t3  
t2 t3 t4
```

Semaphores

Although you can implement the code in C, verification cannot take into account a semaphore system call. However, you can use critical sections to model the behavior of semaphores.

Effects of Imprecision on Shared Variable List

The list of shared variables that Polyspace identifies might contain more than the exact number of shared variables.

Note At a minimum, the list of shared variables contains all global variables or the exact number of shared variables.

Consider the following example.

```
// -entry-points IT_1, IT_2
int C[1];
int D[1];
extern int random(void);
void alias(int* par)
{
    int var;
    var=*par;
}

void IT_1(void)
{
    while (1)
    {
        if (random())
        {
            D[0]=C[0];
            alias(D);
        }
    }
}

void IT_2(void)
```



```

{
while (1)
{
    if (random())
    {
        C[0]=C[0]+1;
        alias(C);
    }
}
}

void main(void)
{
    C[0]=0;
    D[0]=0;
}

```

The variable D is not a shared variable. However, because of array imprecision, Polyspace considers D a shared variable.

Mailboxes

Suppose that an application has several tasks, some of which post messages in a mailbox while other tasks read the messages asynchronously.

This communication mechanism is possible because the OS libraries provide send and receive procedures. The source files will be unavailable because the procedures are part of the OS libraries, but the mechanism needs to be modelled for meaningful verification.

By default, the verification automatically stubs the missing OS send and receive procedures. The stub exhibits the following behavior:

- For `send(char *buffer, int length)`, the content of the buffer is written only when the procedure is called.
- For `receive(char *buffer, int *length)`, each element of the buffer will contain the full range of values for the corresponding data type.

You can use this mechanism and other mechanisms, with different levels of precision.

Let Polyspace software stub automatically

- Quick and easy to code.
- **imprecise** because there is no direct connection between a mailbox sender and receiver. It means that even if the sender is only submitting data within a small range, the full data range for the type(s) will be used for the receiver data

Provide a **real mailbox** mechanism

- Costly (time consuming) to implement.
- Can introduce errors in the stubs.
- Provides little additional benefit when compared to the upper approximation solution.

Provide an **upper approximation of the mailbox**

Models the mechanism so that new read from the mailbox reads **one** of the recently posted messages, but not necessarily the last message.

- Quick and easy to code.
- **gives precise results**

Consider the following detailed implementation of the upper approximation solution:

polyspace_mailboxes.h

```
typedef struct _r {
    int length;
    char content[100];
} MESSAGE;
extern MESSAGE mailbox;
void send(MESSAGE * msg);
void receive(MESSAGE *msg);
```

polyspace_mailboxes.c

```
#include "polyspace_mailboxes.h"

MESSAGE mailbox;

void send(MESSAGE * msg)
{
    volatile int test;
    if (test) mailbox = *msg;
    // a potential write to the mailbox
}

void receive(MESSAGE *msg)
{
    *msg = mailbox;
}
```

Original code

```
#include "polyspace_mailboxes.h"

void t1(void)
{
    MESSAGE msg_to_send;
    int i;
    for (i=0; i<100; i++)
        msg_to_send.content[i] = i;
    msg_to_send.length = 100;
    send(&msg_to_send);
}

void t2(void)
{
    MESSAGE msg_to_read;
    receive (&msg_to_read);
}
```

The verification then proceeds on the assumption that each new read from the mailbox reads a message, but not necessarily the last message.

The associated launching command is:

```
polyspace-c -entry-points t1,t2
```

Atomicity (Can Instruction be Interrupted by Another?)

In computer programming, the term *atomic* describes a unitary action or object that is essentially indivisible, unchangeable, whole, and irreducible. The feature *atomicity* requires the following: in a transaction involving two or more discrete pieces of information, either all pieces are committed or no pieces are committed.

Instructional decomposition

Polyspace verification does not take into account either CPU instruction decomposition or timing considerations.

Polyspace verification assumes that instructions are not atomic except in the case of read and write instructions. Polyspace verification makes an **upper approximation of all scheduling and all interleaving**. There are more paths modelled than could be implemented during execution, but given that **all possible paths are analyzed**, this has no adverse effect on of the results.

Consider a 16-bit target that can manipulate a 32-bit type (an int, for example). In this case, the CPU needs at least two cycles to write to an integer.

Suppose that x is an integer in a multitasking system, with an initial value of 0x0000. Now suppose 0xFF55 is written it. If the operation is not atomic it could be interrupted by another instruction in the middle of the write operation.

- Task 1: Writes 0xFF55 to x.
- Task 2: Interrupts task 1. Depending on the timing, the value of x could be either 0xFF00, 0x0055 or 0xFF55.

Polyspace verification considers write/read instructions atomic, so **task 2 can only read 0xFF55**, even if X is not protected (see “Shared Variables” on page 7-45).

Critical sections

In terms of critical sections, Polyspace does not model the concept of atomicity. A critical section guarantees only that once the function associated with `-critical-section-begin` is called, any other function making use of the same label is blocked. Other functions can still continue to run, even if somewhere else in another task a critical section has been started.

Polyspace verification of run-time errors supposes that there is no conflict when writing the shared variables. Therefore, even if a shared variable is not protected, the run-time error verification is complete and correct.

More information is available in “Critical Sections” on page 7-45.

Priorities

Priorities are not taken into account by Polyspace verification. However, the timing implications of software execution are not relevant to the verification, which is the primary reason for implementing software task prioritization. In addition, priority inversion issues can mean that the software cannot assume that priorities can protect shared variables. For that reason, Polyspace software makes no such assumption.

Although you cannot specify task priorities, priorities **are** taken into account because the default behavior of the software assumes that:

- All task entry points (as defined with the option `-entry-points`) start potentially at the same time;
- The task entry points can interrupt each other, regardless of the sequence of instructions. Therefore, all possible interruptions are accounted for.

If you have two tasks, `t1` and `t2`, in which `t1` has higher priority than `t2`, use the option `-entry-points t1,t2`.

- `t1` interrupts `t2` at any stage of `t2`, which models the behavior at execution time.
- `t2` interrupts `t1` at any stage of `t1`, which models a behavior which (ignoring priority inversion) would never take place during execution. Polyspace verification has made an **upper approximation of all scheduling and interleaving**. There are more paths modelled than could happen during execution, but this does not have an adverse effect on the results.


Comment Code for Known Defects

This example shows how to place comments in your code to mark defects that you are already aware of but do not intend to fix immediately. Using your comments, Polyspace populates the defect **Classification**, **Status** and **Comment** fields on the **Results Summary** pane. After you have placed your comments in your code, you or another reviewer can avoid reviewing the same defect twice. The example uses the following code that is stored in a file `divideByDifference.c`.

```
#include <math.h>
int divideByDifference(int num, int x, int y)
{
    if(x >= y)
        return(num/(abs(x)-abs(y)));
    else
        return 0;
}
```

Verify Source File and Review Results

1 Create a new Polyspace project. Add the file `divideByDifference.c` to the project.

2 Click  to start verification on your project.

The verification uses the default options on the **Configuration** pane. It uses a generated `main` to call the function, `divideByDifference`.

3 Open the verification results. On the **Results Summary** pane, select:

- One of the two orange **Invalid use of standard library routine** errors. On the **Check Details** pane, you can see an error message that the orange error on the `abs` functions can be due to unbounded input values.

Enter the following review information for the error.

Column name	Review Information
Classification	Not a defect
Status	No action planned
Comment	Argument of abs is bounded.

- The orange **Division by Zero** error.

Enter the following review information for the error.

Column name	Review Information
Classification	High
Status	Investigate
Comment	To check if x can be equal to y.

Comment Code for `STD_LIB` error

- 1 On the **Results Summary** pane, right-click one of the orange **Invalid use of standard library routine** errors. Select **Add Pre-Justification to Clipboard**.

This action copies your **Classification**, **Status**, and **Comment** in a form that you can insert in your source code.

- 2 Using the paste option in your text editor, in the file `divideByDifference.c`, paste what you copied just before the line `return(num/(abs(x)-abs(y)));`.

Your source code appears as follows:

```
#include <math.h>
int divideByDifference(int num, int x, int y)
{
    if(x >= y)
        /* polyspace<RTE:STD_LIB:Not a defect:No action planned>
Argument of abs is bounded. */
        return(num/(abs(x)-abs(y)));
    else
        return 0;
```

}

3 Run the verification again. Open your results.

On the **Results Summary** pane, both instances of **Invalid use of standard library routine** on the line `return(num/(abs(x)-abs(y)))`; have the **Classification**, **Status**, and **Comment** that you entered.

Comment Code for OVFL error

1 In the file `divideByDifference.c`, edit the comment that you entered.

Original	Replace with
STD_LIB	STD_LIB,OVFL
Not a defect	Low
Argument of abs is bounded.	Error does not occur for values of x and y.

2 Run the verification again. Open your results.

On the **Results Summary** pane, the **Overflow** and **Invalid use of standard library routine** checks on the line `return(num/(abs(x)-abs(y)))`; have the following review information:

Column name	Review Information
Classification	Low
Status	No action planned
Comment	Error does not occur for values of x and y.

Comment Code for ZDV error

1 On the **Results Summary** pane, right-click the orange **Division by Zero** error. Select **Add Pre-Justification to Clipboard**.

This action copies your **Classification**, **Status**, and **Comment** in a form that you can insert in your source code.

- 2 In the file `divideByDifference.c`, paste what you copied after the already existing comment.

Your source code appears as follows:

```
#include <math.h>
int divideByDifference(int num, int x, int y)
{
    if(x >= y)
        /* polyspace<RTE:STD_LIB,OVFL:Not a defect:
No action planned> Error does not occur for values of x and y. */
        /* polyspace<RTE:ZDV:High:Investigate>
To check if x can be equal to y. */
        return(num/(abs(x)-abs(y)));
    else
        return 0;
}
```

- 3 Run the verification again. Open your results.

On the **Results Summary** pane, the **Division by Zero** error on the line `return(num/(abs(x)-abs(y)));` has the **Classification**, **Status**, and **Comment** that you entered. The other errors retain the earlier review information.

Concepts

- “Comment Syntax for Marking Known Defects” on page 7-58

Comment Syntax for Marking Known Defects

You can place comments in your code to mark defects that you are already aware of but do not intend to fix immediately. Using your comments, Polyspace populates the defect **Classification**, **Status** and **Comment** fields of the **Results Summary** pane. After you have placed your comments in your code, you or another reviewer can avoid reviewing the same defects twice.

To place comments, you can:

- Use the right-click option **Add Pre-Justification To Clipboard** on the **Results Summary** pane.
- Manually enter the comments in a specific syntax just before the line containing the defect.

To comment:

- An individual line of code, use the following syntax:

```
/* polyspace<Defect:Kind1[,Kind2] :  
[Classification] : [Status] >  
[Additional text] */
```

- A section of code, use the following syntax:

```
/* polyspace:begin<Defect:Kind1[,Kind2] :  
[Classification] : [Status] >  
[Additional text] */
```

```
... Code section ...
```

```
/* polyspace:end<Defect:Kind1[,Kind2] : [Classification] : [Status] > */
```

The square brackets *[]* indicate optional information.

Replace	Replace with
<i>Defect</i>	<p>Runtime errors: RTE</p> <hr/> <p>Coding rule violations:</p> <ul style="list-style-type: none"> • MISRA-C ▪ MISRA-AC-AGC ▪ MISRA-CPP ▪ JSF ▪ Custom
<i>Kind1,Kind2,...</i>	<p>Runtime errors:</p> <p>Acronyms for checks such as ZDV, OVFL, etc..</p> <p>If you want the comment to apply to all checks on the following line, specify ALL.</p> <hr/> <p>Coding rule violations:</p> <p>Rule number. For more information, see:</p> <ul style="list-style-type: none"> ▪ “MISRA C:2004 Coding Rules” ▪ “MISRA C++ Coding Rules” ▪ “JSF C++ Coding Rules” ▪ “Custom Naming Convention Rules” <p>If you want the comment to apply to all coding rule violations on the following line, specify ALL.</p>

Replace	Replace with
<i>Classification</i>	<p>Text that indicates the severity of the defect. Enter one of the following:</p> <ul style="list-style-type: none"> • Unset ▪ High ▪ Medium ▪ Low ▪ Not a defect <p>This text populates the Classification column on the Results Summary pane.</p>
<i>Status</i>	<p>Text that indicates how you intend to correct the error in your code. Enter one of the following or any other text:</p> <ul style="list-style-type: none"> ▪ Fix ▪ Improve ▪ Investigate ▪ Justify with annotations ▪ No action planned ▪ Restart with different options ▪ Other ▪ Undecided <p>This text populates the Status column on the Results Summary pane.</p>
<i>Additional text</i>	<p>Any text. This text populates the Comment column on the Results Summary pane.</p>

Syntax Examples: Runtime Errors

- Non terminating call:

```
/* polyspace<RTE: NTC : Low : No Action Planned  
> Known issue */
```

- Division by zero:

```
/* polyspace<RTE: ZDV : High : Fix > Denominator cannot be zero */
```

Syntax Examples: Coding Rule Violations

- MISRA C rule violation:

```
/* polyspace<MISRA-C:6.3 : Low : Justify with annotations> Known issue */
```

- JSF C++ rule violation:

```
/* polyspace<JSF:9 : Low : Justify with annotations> Known issue */
```

Related Examples

Concepts

- “Comment Code for Known Defects” on page 7-54
- “Check Acronyms for Code Comments” on page 7-62

Check Acronyms for Code Comments

You can place comments in your code to mark defects that you are already aware of but do not intend to fix immediately. Using your comments, Polyspace populates the defect **Classification**, **Status** and **Comment** fields of the **Results Summary** pane. After you have placed your comments in your code, you or another reviewer can avoid reviewing the same defects twice.

The following table lists alphabetically the check acronyms that you must use in the comments:

Check	Acronym
Absolute address	ABS_ADDR
C++ specific checks	CPP
Correctness condition	COR
Division by zero	ZDV
Exception handling	EXC
Function returns a value	FRV
Illegally dereferenced pointer	IDP
Initialized return value	IRV
Inspection points	IPT
Invalid use of standard library routine	STD_LIB
Known non-terminating call	k_NTC
Non-initialized local variable	NIVL
Non-initialized pointer	NIP
Non-initialized variable	NIV
Non-null this-pointer in method	NNT
Non-terminating call	NTC
Non-terminating loop	NTL
Object oriented programming	OOP
Out of bounds array index	OBAI

Check	Acronym
Overflow	OVFL
Shift operations	SHF
Unreachable code	UNR
User assertion	ASRT

Related Examples

- “Comment Code for Known Defects” on page 7-54

Concepts

- “Comment Syntax for Marking Known Defects” on page 7-58

Types Promotion

In this section...

“Unsigned Integers Promoted to Signed Integers” on page 7-64

“Promotions Rules in Operators” on page 7-65

“Example” on page 7-65

Unsigned Integers Promoted to Signed Integers

You need to understand the circumstances under which signed integers are promoted to unsigned.

For example, the execution of the following code would produce an assertion failure and a core dump.

```
#include <assert.h>
int f1(void) {
    int x = -2;
    unsigned int y = 5;
    assert(x <= y);
}
```

Implicit promotion explains this behavior. In this example, `x <= y` is implicitly:

```
((unsigned int) x) <= y /* implicit promotion since y is unsigned */
```

A negative cast into unsigned gives a large value. This value can never be `<= 5`, so the assertion can never hold true.

In this second example, consider the range of possible values for `x`:

```
void f2(void)
volatile int random;
unsigned int y = 7;
int x = random;
assert ( x >= -7 && x <= y );

assert (x>=0 && x<=7);
```


The first assertion is orange; it may cause an assert failure. However, given that the range of `x` after the first assertion is **not** `[-7 .. 7]`, but rather `[0 .. 7]`, the second assertion would hold true.

Promotions Rules in Operators

Familiarity with the rules applying to the standard operators of the C language helps you to analyze those orange and **red** checks which relate to overflows on type operations. Those rules are:

- Unary operators operate on the type of the operand.
- Shifts operate on the type of the left operand.
- Boolean operators operate on Booleans.
- Other binary operators operate on a common type. If the types of the two operands are different, they are promoted to the first common type which can represent both of them.
- Be careful of constant types.
- Be careful when verifying a operation between variables of different types without an explicit cast.

Example

Consider the integer promotion aspect of the ANSI C standard (see 6.2.1 in ISO[®]/IEC 9899:1990). On arithmetic operators like `+`, `-`, `*`, `%` and `/`, an integer promotion is applied on both operands. For verification, that can imply an OVFL or a UNFL orange check.

```
2 extern char random_char(void);
3 extern int random_int(void);
4
5 void main(void)
6 {
7   char c1 = random_char();
8   char c2 = random_char();
9   int i1 = random_int();
10  int i2 = random_int();
11
12  i1 = i1 + i2;    // A typical OVFL/UNFL on a + operator
```

```
13 c1 = c1 + c2; // An OVFL/UNFL warning on the c1
14 // assignment [from int32 to int8]
15 }
```

Unlike the addition of two integers at line 12, an implicit promotion is used in the addition of the two chars at line 13. Consider this second “equivalence” example.

```
2 extern char random_char(void);
3
4 void main(void)
5 {
6   char c1 = random_char();
7   char c2 = random_char();
8
9   c1 = (char)((int)c1 + (int)c2); // Warning OVFL: due to
10 // integer promotion
11 }
```

An orange check represents a warning of a potential overflow (OVFL), generated on the (char) cast [from int32 to int8]. A green check represents a verification that the + operator does not produce an overflow (OVFL).

Integer promotion requires that the abstract machine must promote the type of each variable to the integral target size before realizing the arithmetic operation and subsequently adjusting the assignment type. See the preceding equivalence example of a simple addition of two *char*.

Integer promotion respects the size hierarchy of basic types:

- *char* (*signed* or *not*) and *signed short* are promoted to *int*.
- *unsigned short* is promoted to *int* only if *int* can represent all possible values of an *unsigned short*. If that is not the case (because of a 16-bit target, for example) then *unsigned short* is promoted to *unsigned int*.
- Other types such as *(un)signed int*, *(un)signed long int*, and *(un)signed long long int* promote themselves.

Ignoring Assembly Code

In this section...

“Ignoring Assembly Code — Overview” on page 7-67

“When to Ignore Assembly Code” on page 7-67

“Automatic Stubbing of Single Function” on page 7-70

“Automatic Stubbing of List of Functions” on page 7-70

“Directives #asm and #endasm” on page 7-72

“If Verification Fails to Parse asm Code” on page 7-72

“Local Variables in Functions with Assembly Code” on page 7-73

Ignoring Assembly Code – Overview

Polyspace verification is designed for C code. By default, Polyspace ignores assembly code during verification.

In cases, where Polyspace does not ignore assembly code by default, use the command line options `-asm-begin` and `-asm-end` to specify the beginning and end of assembly code sections.

When to Ignore Assembly Code

Ignoring assembly instructions can change the behavior of the code. For example, a write access to a shared variable can be written in assembly code. If this write access is ignored, the verification may produce inaccurate results. In such cases, use stubbing, which applies to functions as well as to stubbed instructions. For more information, see “Stubbing Overview” on page 7-3.

Ignoring assembly code is an acceptable approach when the ignored assembly instructions have no impact on the remainder of the function. See “When to Provide Function Stubs” on page 7-4.

The following code illustrates this approach.

```
int f(void)
{
```

```
asm ("% reg val; mtmsr val;");
asm("\tmove.w #$2700,sr");
asm("\ttrap #7");
asm(" stw r11,0(r3) ");
assert (1); // is green
return 1;
}

int other_ignored6(void)
{
#define A_MACRO(bus_controller_mode) \
__asm__ volatile("nop"); \
__asm__ volatile("nop"); \
__asm__ volatile("nop"); \
__asm__ volatile("nop"); \
__asm__ volatile("nop"); \
__asm__ volatile("nop")
assert (1); // is green
A_MACRO(x);
assert (1); // is green
return 1;
}

int pragma_ignored(void)
{
#pragma asm
SRST
#pragma endasm
assert (1); // is green
}

int other_ignored2(void)
{
asm "% reg val; mtmsr val;";
asm mtmsr val;
assert (1); // is green
asm ("px = pm(0,%2); \
%0 = px1; \
%1 = px2;"
: "=d" (data_16), "=d" (data_32)
```

```

        : "y" ((UI_32 pm *)ram_address):
"px");
    assert (1); // is green
}

int other_ignored1(void)
{
    __asm
    {MOV R8,R8
    MOV R8,R8
    MOV R8,R8
    MOV R8,R8
    MOV R8,R8}
    assert (1); // is green
}

int GNUC_include (void)
{
    extern int __P (char *__pattern, int __flags,
    int (*__errfunc) (char *, int),
    unsigned *__pglob) __asm__ ("glob64");
    __asm__ ("rorw $8, %w0" \
    : "=r" (__v) \
    : "0" ((guint16) (val)));
    __asm__ ("st g14,%0" : "=m" (*(AP)));
    __asm__(" \
    : "=r" (__t.c) \
    : "0" (((union { int i, j; } *) (AP))++)->i));
    assert (1); // is green
    return (int) 3 __asm__("% reg val");
}

int other_ignored3(void)
{
    __asm {ldab 0xffff,0;trapdis;};
    __asm {ldab 0xffff,1;trapdis;};
    assert (1); // is green
    __asm__ ("% reg val");
    __asm__ ("mtmsr val");
    assert (1); // is green
}

```

```
    return 2;
}

int other_ignored4(void)
{
    asm {
        port_in: /* byte = port_in(port); */
        mov EAX, 0
        mov EDX, 4[ESP]
        in AL, DX
        ret
        port_out: /* port_out(byte,port); */
        mov EDX, 8[ESP]
        mov EAX, 4[ESP]
        out DX, AL
        ret }
    assert (1); // is green
}
```

Automatic Stubbing of Single Function

The software automatically stubs a function that is preceded by `asm`, even if a body is defined.

```
asm int h(int tt)           // function h is stubbed even if body is defined
{
    % reg val;              // ignored
    mtmsr val;              // ignored
    return 3;                // ignored
};

void f(void) {
    int x;
    x = h(3);                // x is full-range
}
```

Automatic Stubbing of List of Functions

The functions that you specify through the following pragma are stubbed automatically, even if function bodies are defined:

```
#pragma inline_asm(List of functions)
```

The following code provides examples:

```
#pragma inline_asm(ex1, ex2)
    // The functions ex1 and ex2 are
    // stubbed, even if their bodies are defined

int ex1(void)
{
    % reg val;
    mtmsr val;
    return 3;                // ignored
};

int ex2(void)
{
    % reg val;
    mtmsr val;
    assert (1);             // ignored
    return 3;
};

#pragma inline_asm(ex3) // the definition of ex3 is ignored

int ex3(void)
{
    % reg val;
    mtmsr val;             // ignored
    return 3;
};

void f(void) {
    int x;

    x = ex1();             // ex1 is stubbed : x is full-range
    x = ex2();             // ex2 is stubbed : x is full-range
    x = ex3();             // ex3 is stubbed : x is full-range
}
```

Directives `#asm` and `#endasm`

By default, the software disregards assembly code that lies between `#asm` and `#endasm`.

```
void test(void)
{
    #asm
        mov _as:pe, reg
        jre _nop
    #endasm
    int r;
    r=0;
    r++;
}
```

If Verification Fails to Parse `asm` Code

Occasionally the software might not automatically ignore an assembly code section, for example, when the code section is compiler-specific. In this case, use the options `-asm-begin` and `-asm-end`.

Consider the following code.

```
1 int x=12;
2
3 void f(void)
4 {
5 #pragma will_be_ignored
6 x =0;
7 x= 1/x;           // no color is displayed
8                   // not even C code
9 #pragma was_ignored
10 x++;
11 x=15;
12 }
13
14 void main (void)
15 {
16 int y;
```



```

17 f();
18 y = 1/x + 1 / (x-15); // Red ZDV, x is equal to 15
19
20 }

```

The verification ignores text or code placed between the two `#pragma` statements if you specify the following options:

```
-asm-begin will_be_ignored -asm-end was_ignored
```

This approach allows an unsupported assembly code section to be ignored without changing the meaning of the original code.

Local Variables in Functions with Assembly Code

In functions containing assembly code, the software treats local variables that are not explicitly initialized as potentially initialized variables.

Consider the following function.

```

1 inline int f(void) {
2   int r;
3   asm("mov 4%0,%eax:::m"(r));
4   return r; // orange NIVL (red NIVL before 12a) because r is not initialized
5 }

```

The software treats `r` as a potentially initialized variable. Verification generates an orange NIVL check for `r`.

Consider another function.

```

1 int dummy(void) {
2   int g,h;
3   h = g * 2; // orange NIVL for g (red NIVL before 12a)
4   h = 2; // h is assigned the value 2
5   asm("int $0x3");
6   asm("mov 4%0,%eax:::m"(g));
7   asm("movss 4%0,%xmm1:::m");
8   return h; // value returned is 2
9 }

```

In line 3, the variable `g` is not initialized. Verification:

- Generates an orange NIVL check for `g`.
- Assigns a full-range value to `g`.

Loss of Precision Using memset and memcpy

Polyspace verifies uses of `memset` and `memcpy` as full-range values for all possibilities except zero. This interpretation of values can cause a loss of precision for data not initialized to zero. `memset` and `memcpy` view structures as a block of contiguous data. Problems can occur because of “padding” between fields, especially if the fields are of different data types. Additionally, reading fields with `memcpy` might result in different values depending on the endianness of the machine. For example, consider two variables `tab` and `glob`:

```
char tab[4] = {0, 0, 0, 1}; //4 bytes of memory
int glob; //also 4 bytes of memory
memcpy(&glob, tab, 4);
```

Under different machines, the `memcpy` function gives `glob` different values.

Machine Environment	Value of glob
Big endian (i.e. Sparc)	1
Little endian (i.e. x86)	16777216 (0x01000000)

These machine dependencies cause Polyspace to see the values as full range and prevent it from proving the validity of the data values. Therefore, objects modified by `memset` or `memcpy` might result in an orange check.

To learn how to improve Polyspace precision when initializing with `memset`, see “Avoid `memset` and `memcpy` for Structure Initialization” on page 7-76.

Avoid `memset` and `memcpy` for Structure Initialization

The most common use of `memset` is initialization of an object (array, structure, etc.). When initializing to zero, Polyspace can validate the value of the data as exactly zero. The software assumes full range for other values (i.e. $[-2^{31}, 2^{31}-1]$ for an `int`). However, a better way to do the initialization is to replace the call to `memset` by initializing with curly braces: `S another_struct = {2,1,0}`. With this initialization, Polyspace recognizes precisely what the structure fields are and their values. You can use the syntax `{ }` only in a declaration statement.

The following example summarizes how Polyspace sees objects when modified with `memset` and `memcpy`. The comments show the value of `x` depending on how the data is initialized:

```
#include <string.h>

typedef struct {
    char a;
    int b;
    char c;
} S;

S s;
int glob;

void main() {
    int x;

    // initialize
    S struct_2 = {1,1,1};
    char tab[4] = {0,0,0,1};

    // test of memcpy
    memcpy(&glob, tab, 4);    // array of char to int
    x = glob;                // x is full range ~ [-2^31, 2^31-1]

    // test of memset
    memset(&s, 1, sizeof(S));
    x = s.b;                // x is full range ~ [-2^31, 2^31-1]
```

```
memset(&s, 0, sizeof(S));  
x = s.b;           // x = 0  
  
s = struct_2;  
x = s.b;           // x = 1  
}
```

To learn why `memset` and `memcpy` cause a loss of precision in Polyspace verification, see “Loss of Precision Using `memset` and `memcpy`” on page 7-75.

Running a Verification

- “Types of Verification” on page 8-2
- “Select Analysis Options Configuration” on page 8-3
- “Check for Compilation Problems” on page 8-4
- “Start Local Verification” on page 8-6
- “Start Remote Verification” on page 8-7
- “Stop Verification” on page 8-8
- “Phases of Verification” on page 8-9
- “Run Verification Unit-by-Unit” on page 8-10
- “Verify All Modules in Project” on page 8-11
- “Manage Previous Verifications With Polyspace Metrics” on page 8-12
- “Manage Remote Verifications” on page 8-15
- “Monitor Progress of Verification” on page 8-16
- “Run Verification from Command Line” on page 8-17
- “Manage Remote Analyses at the Command Line” on page 8-18
- “Modularization of Large Applications” on page 8-20
- “Partition Application into Modules” on page 8-21
- “Choose Number of Modules for Application” on page 8-24
- “Partition Application Using Batch Command” on page 8-27

Types of Verification

You can run a local or remote verification.

Verification type	How to specify verification	Use when
Remote	Select Distributed Computing > Batch check box.	Source files are large (more than 800 lines of code including comments), and execution time of verification is long.
	Select Distributed Computing > Add to results repository check box.	You want to generate Polyspace Metrics. Through Polyspace Metrics, you can manage verifications and monitor quality over a project lifecycle.
Local	Clear Distributed Computing > Batch check box.	Source files are small, and execution time of verification is short.

Select Analysis Options Configuration

Each module in the Project Browser can contain multiple configurations, with each configuration specifying a set of analysis options. This allows you to verify the same source files multiple times using different analysis options for each verification.

If you have created multiple configurations, you must choose a configuration before starting a verification.

To specify the configuration for a verification:

- 1** In the **Project Browser**, select the module you want to run.
- 2** In the **Configuration** folder of the module, right-click the configuration that you want to use. Select **Set as Default**.


When you start the verification, the software uses the analysis options from this configuration.

For more information, see “Specify Analysis Options” on page 3-10.

Check for Compilation Problems

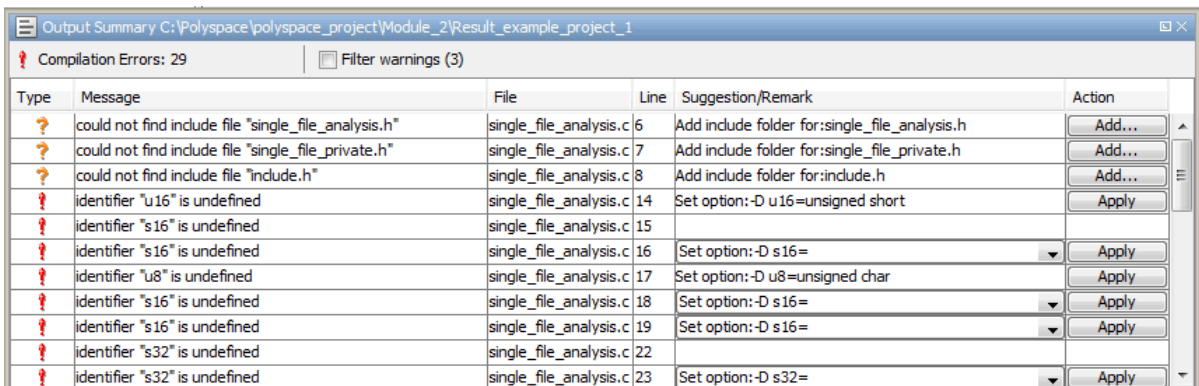
During a verification, if the Compilation Assistant detects compilation errors, the verification stops and the software displays errors and possible solutions on the **Output Summary**.

To check your project for compilation problems:

- 1 Select **Options > Preferences**.
- 2 In the Polyspace Preferences dialog box, click the **Project and Results Folder** tab.
- 3 Select the **Use Compilation Assistant** check box. Then click **OK**.
- 4 On the Project Manager toolbar, click  .

The software compiles your code and checks for errors, and reports the results on the **Output Summary** tab.

- 5 Select a **Suggestion/Remark** cell to see a list of possible solutions for the problem.



Output Summary C:\Polyspace\polyspace_project\Module_2\Result_example_project_1

Compilation Errors: 29 Filter warnings (3)

Type	Message	File	Line	Suggestion/Remark	Action
?	could not find include file "single_file_analysis.h"	single_file_analysis.c	6	Add include folder for:single_file_analysis.h	Add...
?	could not find include file "single_file_private.h"	single_file_analysis.c	7	Add include folder for:single_file_private.h	Add...
?	could not find include file "include.h"	single_file_analysis.c	8	Add include folder for:include.h	Add...
!	identifier "u16" is undefined	single_file_analysis.c	14	Set option:-D u16=unsigned short	Apply
!	identifier "s16" is undefined	single_file_analysis.c	15		
!	identifier "s16" is undefined	single_file_analysis.c	16	Set option:-D s16=	Apply
!	identifier "u8" is undefined	single_file_analysis.c	17	Set option:-D u8=unsigned char	Apply
!	identifier "s16" is undefined	single_file_analysis.c	18	Set option:-D s16=	Apply
!	identifier "s16" is undefined	single_file_analysis.c	19	Set option:-D s16=	Apply
!	identifier "s32" is undefined	single_file_analysis.c	22		
!	identifier "s32" is undefined	single_file_analysis.c	23	Set option:-D s32=	Apply

In this example, you can either add the missing include files, or set options to compile the code without the missing include files:

- Select **Apply** to set the selected option for your project. The software automatically sets the option.
- Select **Add** to add suggested include folders to your project. The Add Source Files and Include Folders dialog box opens, allowing you to add additional include folders.


When you have addressed compilation problems, run the verification again.

The Compilation Assistant is automatically disabled if you specify one of the following options:

- `-unit-by-unit`
- `-post-preprocessing-command`

Start Local Verification

To start a verification on your local computer:


- 1** In the Project Manager perspective, from the **Project Browser** view, select the module you want to verify.
- 2** Select the **Configuration > Distributed Computing** pane.
- 3** By default, the **Batch** check box is not selected. However, if this check box is selected, you must clear the check box.
- 4** On the Project Manager toolbar, click  .

You can monitor the progress of the verification through the **Progress Monitor**, **Full Log**, and **Output Summary** tabs. See “Monitor Progress of Verification” on page 8-16.

Start Remote Verification

Before you run a remote verification, you must set up a server for this purpose. For more information, see “Set Up Remote Verification and Analysis”.

To start a remote verification:

- 1** In the Project Manager perspective, from the **Project Browser** pane, select the module you want to verify.
- 2** Select the **Configuration > Distributed Computing** pane.
- 3** Select the **Batch** check box. The software runs the verification on your computer cluster with batch commands.
- 4** On the Project Manager toolbar, click  .

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 8-9.

Note If you see the message `Verification process failed`, click **OK**. For more information on errors related to remote verification, see “Polyspace software cannot find the server” on page 9-9.

By default, the software also selects the **Add results to repository** check box, which enables the generation of Polyspace Metrics. If you clear this check box, the software does not generate Polyspace Metrics but downloads results automatically when the verification is complete.

To monitor progress and manage the verification, see “Manage Remote Verifications” on page 8-15.


Stop Verification

In this section...

“Stop Remote Verification” on page 8-8

“Stop Local Verification” on page 8-8

Stop Remote Verification

- 1 On the Polyspace Code Prover toolbar, click .
- 2 In the Polyspace Queue Manager, right-click your verification. From the context menu, select **Remove From Queue**.

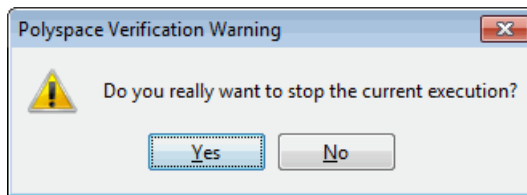
For more information, see “Manage Previous Verifications With Polyspace Metrics” on page 8-12.

Stop Local Verification

To stop a local verification:

- 1 On the Project Manager toolbar, click the **Stop** button.

A warning dialog box opens.



- 2 Click **Yes**. The verification stops, and results are incomplete. If you start another verification, the verification starts from the beginning.

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 if the Polyspace software does not find a main and you have selected the **Verify module** option. For more information about generating a main, see:
 - “Verify module” — C verification
 - “Verify module” — C++ verification
- 3** Analyzing the code for run-time errors and generating color-coded results.

Run Verification Unit-by-Unit


When you run a remote verification, you can create a separate verification job for each source file in the project. Each file is compiled, sent to the head node of your computer cluster, and verified individually. You can view verification results for the entire project, or for individual units.

To run a unit-by-unit verification:

- 1** In the Project Manager perspective, select the **Configuration > Distributed Computing** pane.
- 2** Select the **Batch** check box.
- 3** Select the **Configuration > Code Prover Verification** pane.
- 4** Select the **Run unit by unit verification** check box.

The **Unit by unit common source** files view is now visible.

- 5** You can create a list of common files to include with each unit verification:

- a** Click . The software creates a new row.
- b** In the new row, enter the full path to a common file. For example, `C:\Polyspace\polyspace_project\includes\include.h`.

Repeat steps a and b until you have created your list of common files. These files are compiled once, and then linked to each unit before verification. Functions that are not included in this list are stubbed.

- 6** Save your project.
- 7** On the Project Manager toolbar, click **Run**.

Verify All Modules in Project

You can have many modules within a project, each module containing a set of source files and an active configuration.

To verify all modules in a project:

- 1** In the Project Manager perspective, from the Project Browser, select the project for which you want to run verifications.
- 2** Select **Run > Run All**.

The software verifies each module as an individual job. For information on the verification process, see “Phases of Verification” on page 8-9.

Note If the verification fails, go to “Troubleshooting in Polyspace Code Prover”.

Manage Previous Verifications With Polyspace Metrics

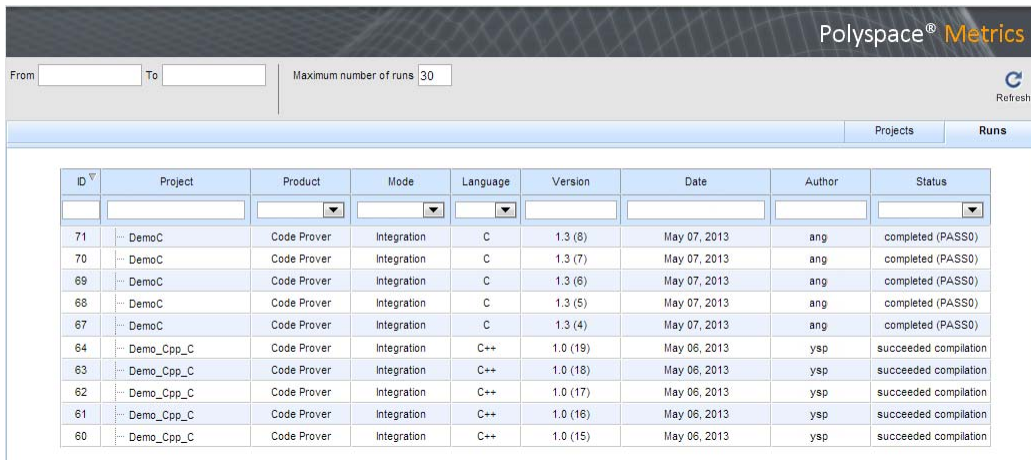
Use the **Runs** view of Polyspace Metrics to administer previous remote verifications. For example, you can:

- Delete verification results from the results repository.
- Set or change the password for projects.

To open the **Runs** view of Polyspace Metrics, in the address bar of your Web browser, enter the following URL:

protocol://ServerName:PortNumber

- *protocol* is either http (default) or https.
- *ServerName* is the name or IP address of your Polyspace Metrics server.
- *PortNumber* is the Web server port number (default 8080).



ID	Project	Product	Mode	Language	Version	Date	Author	Status
71	DemoC	Code Prover	Integration	C	1.3 (8)	May 07, 2013	ang	completed (PASS0)
70	DemoC	Code Prover	Integration	C	1.3 (7)	May 07, 2013	ang	completed (PASS0)
69	DemoC	Code Prover	Integration	C	1.3 (6)	May 07, 2013	ang	completed (PASS0)
68	DemoC	Code Prover	Integration	C	1.3 (5)	May 07, 2013	ang	completed (PASS0)
67	DemoC	Code Prover	Integration	C	1.3 (4)	May 07, 2013	ang	completed (PASS0)
64	Demo_Cpp_C	Code Prover	Integration	C++	1.0 (19)	May 06, 2013	ysp	succeeded compilation
63	Demo_Cpp_C	Code Prover	Integration	C++	1.0 (18)	May 06, 2013	ysp	succeeded compilation
62	Demo_Cpp_C	Code Prover	Integration	C++	1.0 (17)	May 06, 2013	ysp	succeeded compilation
61	Demo_Cpp_C	Code Prover	Integration	C++	1.0 (16)	May 06, 2013	ysp	succeeded compilation
60	Demo_Cpp_C	Code Prover	Integration	C++	1.0 (15)	May 06, 2013	ysp	succeeded compilation

To perform a task:

- 1 Right-click your verification.
- 2 From the context menu, select your task.

The following table describes the tasks that you can perform.

Task	Details
Rename	Available only for Project and Version . When you select this menu item, the text becomes editable. Enter your new project name or version number. Then press Return .
Delete Run from Repository	Remove verification from Polyspace Metrics results repository.
Go to Metrics Page	Open the Polyspace Metrics Summary view of the verification.
Change/Set Password	Control access to the metrics for the project by specifying a password. See “Protect Access to Project Metrics” on page 14-16.


In the **Runs** view, you can use Polyspace Metrics controls to specify the list of verifications displayed.

Control	Details
From	If you click the field, the software displays a calendar. Use this calendar to select the start date for your list.
To	If you click the field, the software displays a calendar. Use this calendar to select the end date of for your list.
Maximum number of runs	Specify the maximum number of verifications that you want to display. The default is 30.
ID	If you enter a numeric string in the field, the software displays verifications with IDs that contain this string.
Project	If you enter a string in the field, the software displays verifications with project names that contain this string.

Control	Details
Product	Polyspace Metrics displays results from Polyspace Bug Finder analyses and Polyspace Code Prover verifications. To display only verifications, from the drop-down list, select Code Prover.
Mode	Use the drop-down list to select verifications that are either Integration or Unit By Unit. By default, both verification types are displayed.
Language	Use the drop-down list to select language type. By default, verifications for all language types are displayed.
Version	If you enter a string in the field, the software displays verifications with version numbers that contain this string.
Date	If you enter a string in the field, the software displays verifications with dates that contain this string.
Author	If you enter a string in the field, the software displays verifications with author names that contain this string.
Status	Use the drop-down list to select verifications with a specific status, for example, completed (PASS4).


Manage Remote Verifications

You can manage your verification through the Polyspace Queue Manager:

- 1** On the Polyspace Code Prover toolbar, click .
- 2** In the Polyspace Queue Manager, 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.
 - **Remove From Queue** — Remove verification from the submission queue.

Monitor Progress of Verification

To monitor the progress of a remote verification, open the verification log:

- 1 On the Polyspace Code Prover toolbar, click .
- 2 In the Polyspace Queue Manager, right-click your verification.
- 3 From the context menu, select **View Log File**.

To monitor the progress of a local verification, use the following tabs in the Project Manager perspective of Polyspace Code Prover:

- **Progress Monitor** — A blue progress bar indicates the current phase of the verification. The tab also displays the time and percentage completed for each phase.
- **Full Log** — This tab displays messages, errors, and statistics for all phases of the verification. To search for a term, in the **Search** field, enter the required term. Click the up arrow or down arrow to move sequentially through occurrences of this term.
- **Output Summary** — Displays compile phase messages and errors. To search for a term, in the **Search** field, enter the required term. Click the up or down arrow to move sequentially through occurrences of the term.

At the end of a local verification, the **Verification Statistics** tab displays statistics, for example, code coverage and check distribution.

Run Verification from Command Line

Use the following command to run a local verification:

```
MATLAB_Install\polyspace\bin\polyspace-code-prover-nodesktop  
[options]
```

Use the following command to run a remote verification:

```
MATLAB_Install\polyspace\bin\polyspace-code-prover-nodesktop  
-batch -scheduler NodeHost | MJSName@NodeHost [options]
```

- *MATLAB_Install* is your MATLAB installation folder, for example:

```
C:\Program Files\MATLAB\R2013b
```

- *NodeHost* is the name of the computer that hosts the head node of your MDCS cluster.
- *MJSName* is the name of the MATLAB Job Scheduler (MJS) on the head node host.

Note Before you run a remote verification, you must set up a server for this purpose. For more information, see “Set Up Remote Verification and Analysis”.

You can also run verifications from the MATLAB Command Window using the `polyspaceCodeProver` command. For information about this command, in the MATLAB Command Window, enter:

```
polyspaceCodeProver(' -help');
```

Manage Remote Analyses at the Command Line

To manage remote analyses, use this command:

```
MATLAB_Install\polyspace\bin\polyspace-jobs-manager
action [options]
           [-scheduler schedulerOption]
```

MATLAB_Install is your MATLAB installation folder, for example:

```
C:\Program Files\MATLAB\R2014a
```

schedulerOption is one of the following:

- Name of the computer that hosts the head node of your MDCS 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 “Clusters and Cluster Profiles”

If you do not specify a job scheduler, `polyspace-jobs-manager` uses the scheduler specified in the **Polyspace Preferences > Server Configuration > Job scheduler host name**.

The following table lists the possible action commands to manage jobs on the scheduler.

Action	Options	Task
<code>listjobs</code>	None	Generate a list of Polyspace jobs on the scheduler. For each job, the software produces the following information: <ul style="list-style-type: none"> • ID — Verification or analysis identifier. • AUTHOR — Name of user that submitted job.

Action	Options	Task
		<ul style="list-style-type: none"> • APPLICATION — Name of Polyspace product, for example, Polyspace Code Prover or Polyspace Bug Finder. • LOCAL_RESULTS_DIR — Results folder on local computer, specified through the Options > Preferences > Server Configuration tab. • WORKER — Local computer from which job was submitted. • STATUS — Status of job, for example, running and completed. • DATE — Date on which job was submitted. • LANG — Language of submitted source code.
download	-job <i>ID</i> -results-folder <i>FolderPath</i>	Download results of analysis with specified ID to folder specified by <i>FolderPath</i> .
getlog	-job <i>ID</i>	Open log for job with specified ID.
remove	-job <i>ID</i>	Remove job with specified ID.

Modularization of Large Applications

The source code within your project may represent a single application. In this case, you might want to analyze all of the code together. However, if the application is extremely large, the verification might 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 — see “Partition Application into Modules” on page 8-21 and “Partition Application Using Batch Command” on page 8-27.
- Specify the number of modules in a trade-off between verification speed and precision — see “Choose Number of Modules for Application” on page 8-24.

Polyspace Model Link products do not support modularization of applications.

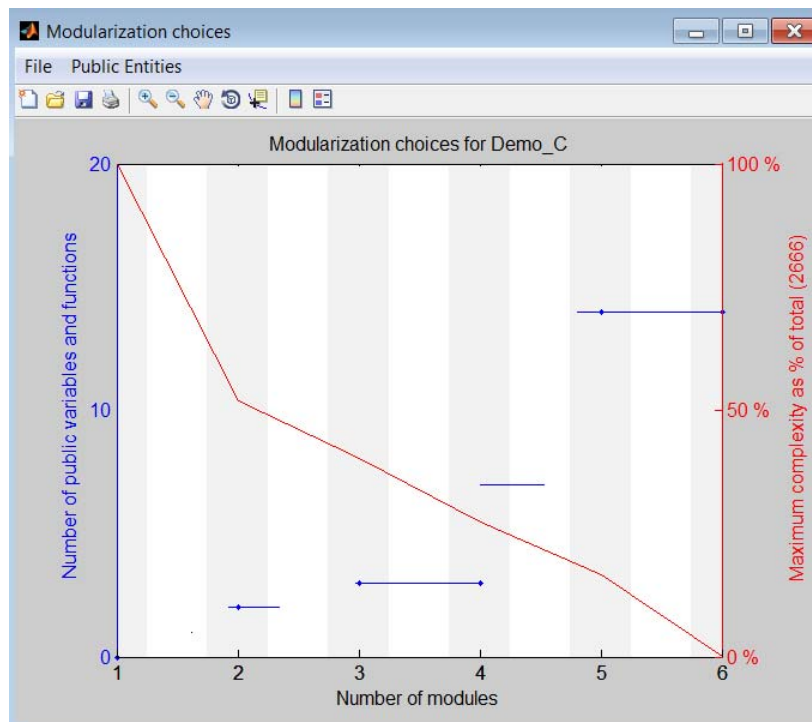
You can carry out faster analysis with a larger number of small modules. However, with more modules, greater cross-module referencing is required during verification, which results in a loss of precision.

Note During partitioning, the software automatically minimizes cross-module references.

Partition Application into Modules

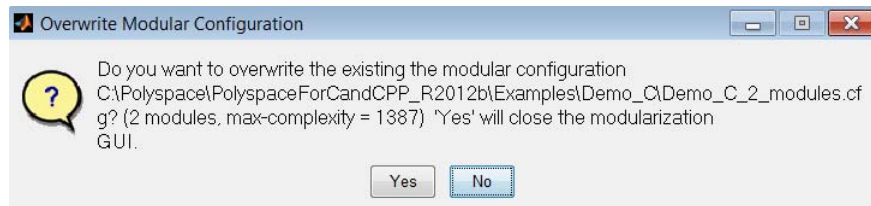
To partition your application into modules:

- 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 From the Project Manager toolbar, select **Run > Run Modularize**. The software analyzes your application code and displays two plots in a new Modularization choices window.

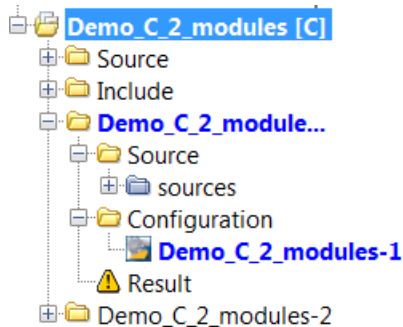


The plots show the following information:

- 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. See “Choose Number of Modules for Application” on page 8-24. In this example, a suitable number is 2 or 4.
 - 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 — with the precision and verification levels that you require. The configuration (.psprj) file for each module specifies the default values:

- **Precision level** — 2
- **Verification level** — Software Safety Analysis level 4

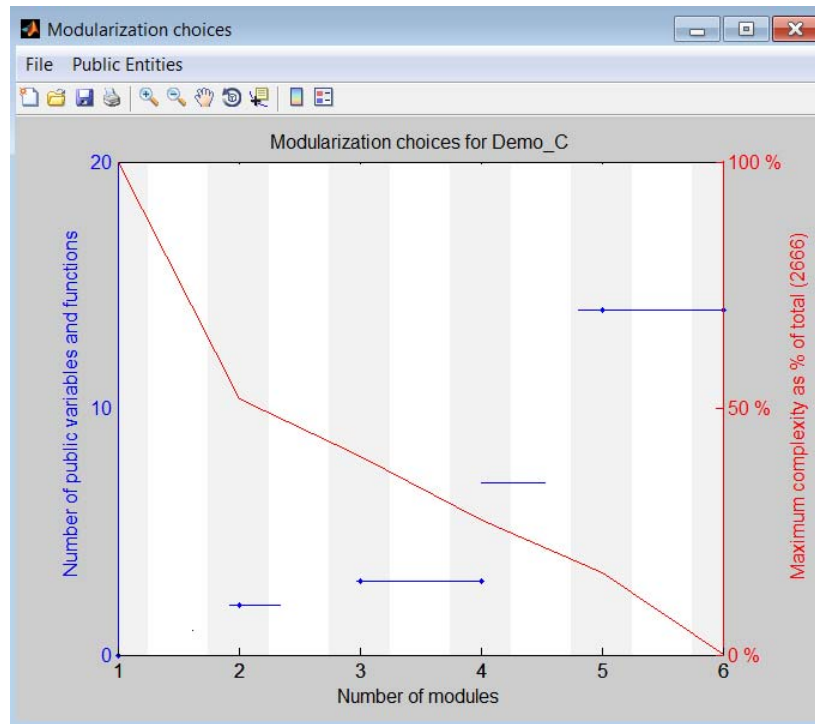
You can change these values through the **Configuration > Precision** pane.

Choose Number of Modules for Application

Use the Modularizing choices window to select the number of partitioned modules. The number of partitioned modules that you choose involves a trade-off between the following:

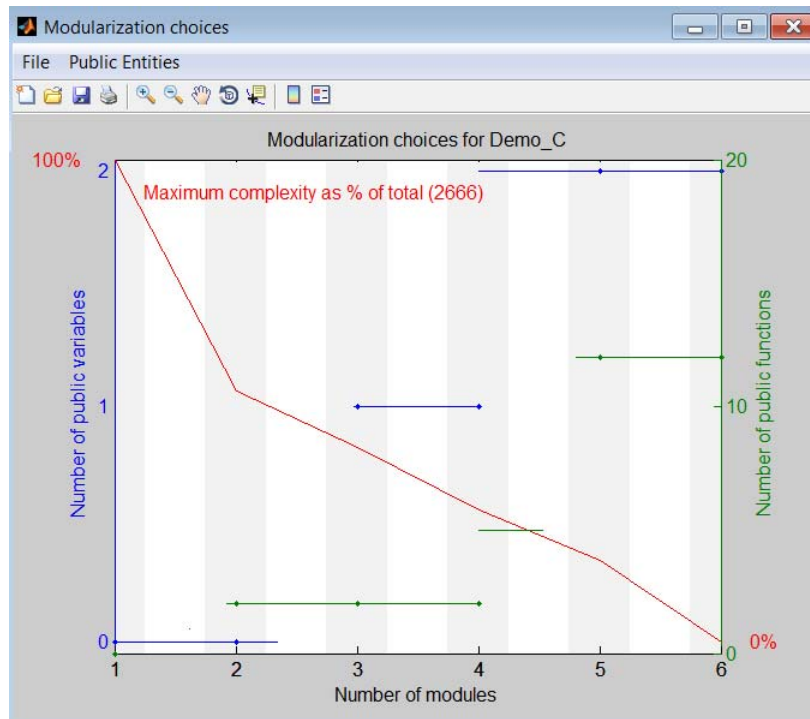
- Time — The smaller the maximum complexity, the shorter the time required for verification. This time saving is even greater if the different modules are verified in parallel.
- Precision — The smaller the number of public variables and functions, the greater the precision of the verification.

Select a number just after a big drop in maximum complexity and before a big increase in the number of public functions and variables. In the following example, you must click the gray region associated with either 2 (just after a big drop in maximum complexity) or 4 (before a big increase in 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:

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



- 2 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.

Partition Application Using Batch Command

In this section...

“Basic Options” on page 8-27

“Constrain Module Complexity During Partitioning” on page 8-28

“Control Naming of Result Folders” on page 8-30

“Forbid Cycles in Module Dependence Graph” on page 8-30

Basic Options

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

```
polyspace-modularize [target_folder] {options}
```

This table describes the basic options that you can use.

Option	Description
<i>target_folder</i>	Folder that contains the results of the initial run that processes source files. Default is the folder from which you run <code>polyspace-modularize</code> .
<code>-o <i>output_folder</i></code>	Output folder for partitioned application. Default is the folder from which you run <code>polyspace-modularize</code> .
<code>-gui <i>max_n</i></code>	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 <i>max_n</i> .

Option	Description
-matlab <i>max_n</i>	<p>If data cache for Modularizing choices window does not exist, create cache <i>project_name_max_n.m</i>. Cache enables faster display of Modularizing choices window.</p> <p><i>project_name</i> is the value used by -prog option.</p> <p><i>max_n</i> is the limit for the maximum number of modules that you can select.</p> <p>No action if cache already exists.</p>
-modules <i>n</i>	<p>Partition application into <i>n</i> modules. Identical to clicking the gray region associated with <i>n</i> in the Modularizing choices window.</p>
-max-complexity <i>max_c</i>	<p>Partitions application into modules with reference to specified maximum complexity <i>max_c</i>. 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 <i>max_c</i>.</p> <p>If you make <i>max_c</i> sufficiently large, the software generates only one module, which is identical to the original, unpartitioned application.</p>

Constrain Module Complexity During Partitioning

Each Polyspace verification produces two "module dependence graph" files in *target_folder/ALL/*:

- *project_name.mdg* — Created early in verification, even for very large applications.
- *project_name_IL.mdg* — Similar to *project_name.mdg*, but based on alias analysis and generated later in verification.

You can partition your application provided an earlier verification has generated the following files in *target_folder*:

- ALL/*project_name*.mdg
- ALL/ SRC/_original.txt
- options
- sources_list.txt

By default, the software uses *project_name*.mdg when partitioning an application. However, in some cases, using *project_name_IL*.mdg might generate more precise results. To specify *project_name_IL*.mdg, run the following command:

```
polyspace-modularize IL
```

Note The -IL option does not support C++.

If you specify the -IL option, then the software computes modules applying the constraint that the complexity of a function is always 1. In addition, using the options:

- -gui *n* and -matlab *n* generates a file named *project_name_IL_n.m*.
- -max-complexity *max_c* generates a file named *project_name_n_modules-IL.psprj*.

n is the number of modules. The results folder for the *i*th module is *project_name_n_modules-IL-i*.

To force all functions to have a complexity of 1 without specifying the -IL option, run the following command:

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

Control Naming of Result Folders

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

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

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

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

If you do not specify `-stem`, then the default string *stem_format* has the form *project_nameCCkk_modules*:

- *CC* is `_IL_` when you use `-IL`, but `_` otherwise.
- *kk* is `@` when you use `-max-complexity` or `#` when you use the Polyspace verification environment.

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

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

Forbid Cycles in Module Dependence Graph

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

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

Troubleshooting Verification Problems

- “View error information when verification stops” on page 9-3
- “Troubleshoot compiler and linking errors” on page 9-6
- “Obtain system information for technical support” on page 9-7
- “Header file location not specified” on page 9-8
- “Polyspace software cannot find the server” on page 9-9
- “Errors due to disk defragmentation and antivirus software” on page 9-10
- “Software runs out of memory during report generation” on page 9-12
- “Compilation Error Overview” on page 9-13
- “Running multiple Polyspace processes” on page 9-14
- “Troubleshoot using preprocessed files” on page 9-15
- “Check Compilation Before Verification” on page 9-20
- “Syntax Error” on page 9-21
- “Undeclared Identifier” on page 9-22
- “Unknown Prototype” on page 9-23
- “No Such File or Folder” on page 9-24
- “#error directive” on page 9-25
- “Class, Array, Struct or Union is Too Large” on page 9-26
- “Unsupported Non-ANSI Keywords (C)” on page 9-27
- “Initialization of Global Variables (C++)” on page 9-29

- “Double Declarations of Standard Template Library Functions” on page 9-30
- “Large Static Initializer” on page 9-31
- “Compilation Messages Described in This Section” on page 9-32
- “C++ Dialect Issues” on page 9-33
- “C Link Errors” on page 9-42
- “C++ Link Errors” on page 9-49
- “Standard Library Function Stubbing Errors” on page 9-53
- “Automatic Stubbing Errors” on page 9-60
- “Reduce Verification Time” on page 9-62
- “Storage of Temporary Files” on page 9-80

View error information when verification stops

If verification stops, you can view error information in the Project Manager interface or in the log file.

In this section...

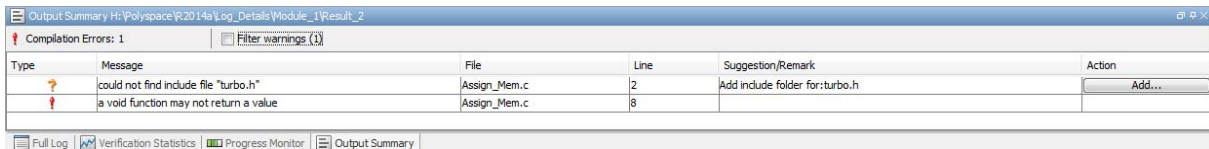
“View Error Information in Project Manager” on page 9-3

“View Error Information in Log File” on page 9-3

View Error Information in Project Manager

- 1 View the errors on the **Output Summary** tab.
- 2 If you have the **Compilation Assistant** on, to fix the 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 through the **Output Summary** tab.



- 3 To open the source code at the line containing the error, double-click the message.
- 4 For more information, right-click the message. From the context menu, select **Open Preprocessed File**.

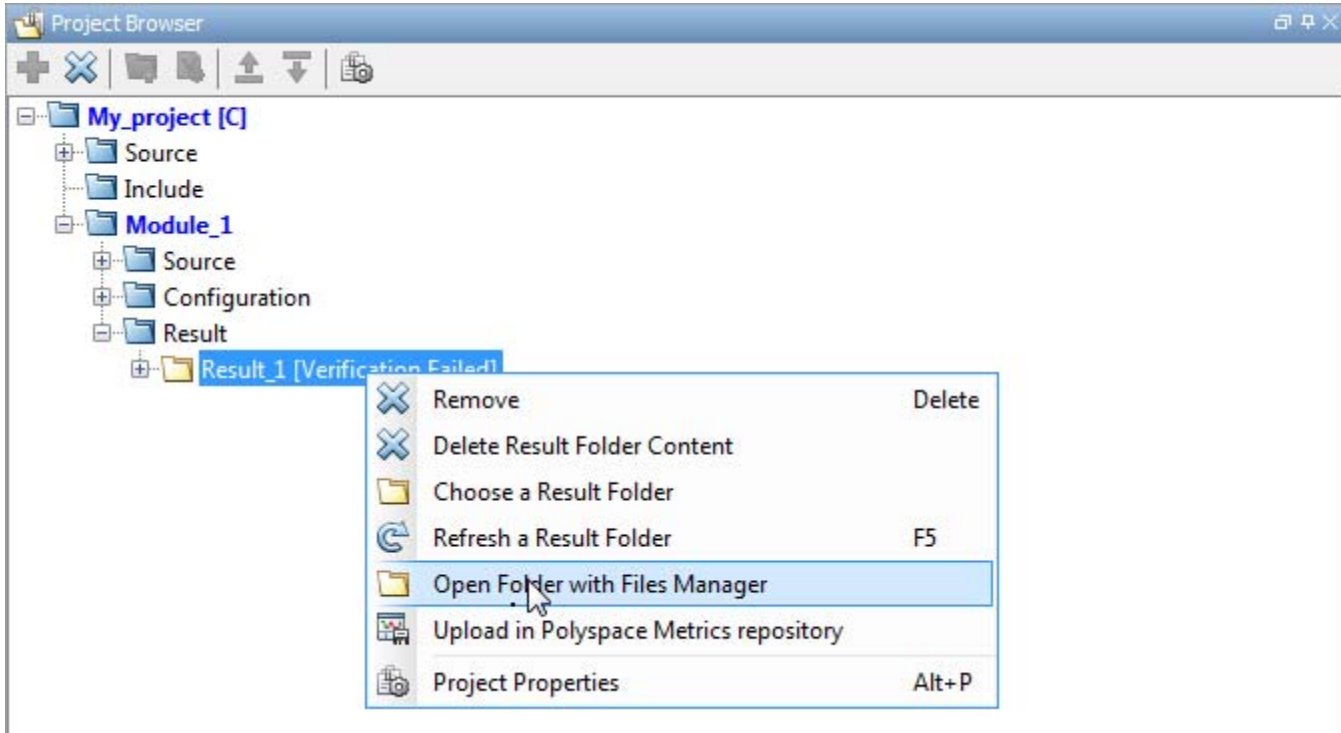
The `.ci` file opens. The Polyspace software uses this file to compile the source file. The file contents can help you understand the compilation error.

- 5 To search the log, enter search terms in the **Search** box.

View Error Information in Log File

You can view errors directly in the log file. The log file is in `Results_folder`.

- 1 To open the Results folder in your file browser, right-click the result folder name on the **Project Browser** pane. From the context menu, select **Open Folder with Files Manager**.



- 2 To view the errors, scroll through the verification log file, starting at the end and working back.

The following example shows sample log file information. The error has occurred because the C++ option `-class-analyzer arg` was used, but the verification 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 compiler and linking errors

When you obtain an error message related to compilation or linking, try:

- Checking whether the error message is related to the dialect that you specified. To specify a different dialect:
 - In the user interface, choose a dialect on the **Configuration** pane. In the **Configuration** tree view, select **Target & Compiler**. From the **Dialect** drop-down list, select an option.
 - At the command line, use the `-dialect` option.

For more information, see “Dialect”.

- Checking whether the error message is related to stubbing of standard library functions. To avoid this stubbing by Polyspace implementations:
 - For C: Use option `-D POLYSPACE_NO_STANDARD_STUBS` or `-D POLYSPACE_STRICT_ANSI_STANDARD_STUBS`.
 - For C++: Use option `-no-stl-stubs`.

For more information, see “Prepare Code for Built-In Functions” on page 7-35.

- Checking the preprocessed files with extension `.ci` to view:
 - Expanded headers.
 - Expanded macros.
 - Active branch of `#ifdef` conditional statement.

For more information, see “Troubleshoot using preprocessed files” on page 9-15.

Obtain system information for technical support

When you enter a support request, you must provide your system information.

In this section...
“Information Required” on page 9-7
“How to Obtain Required Information” on page 9-7

Information Required

- Hardware configuration
- Operating system
- Polyspace and MATLAB licenses
- Specific version numbers for Polyspace products
- Installed Bug Report patches

How to Obtain Required Information

To obtain your configuration information:

- At the command line, run the following command:
 - Linux — `MATLAB_Install/polyspace/bin/polyspace-ver`
 - Windows — `MATLAB_Install\polyspace\bin\polyspace-ver`
- In the Polyspace user interface, select **Help > About**.

Header file location not specified

Message

include.h: No such file or folder

Possible Cause

- You did not specify include folders.
- You specified include folders, but a header file is missing from the specified folders.

Solution

Do one of the following:

- Add the missing header file to the specified include folder.
- Specify another include folder that contains the missing file.

For more information, see “Add Source Files and Include Folders” on page 3-6.

Polyspace software 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 **Options > Preferences**.
- 2 Select the **Server Configuration** tab. Provide your server information.

For more information, see “Set Up Remote Verification and Analysis”.

Errors due to disk defragmentation and antivirus software

Message

```
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 verification does not fail, the antivirus software can reduce the speed of your verification. This reduction occurs because the software checks the temporary verification files. Configure the antivirus software to exclude your temporary folder, for example, C:\Temp, from the checking process.

Software runs out of 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 insufficient memory.

Solution

To avoid this error, increase the Java® heap size. In the *MATLAB_Install\polyspace\bin\architecture\java.opts* file, modify the *-Mx* option.

The default heap size is 512 MB. You can increase it to:

- 1 GB for 32-bit machines.
- 2 GB for 64-bit machines.

Compilation Error Overview

You can use Polyspace software instead of your compiler to make syntactical, semantic, and other static checks. The Polyspace compiler follows the ANSI C90 standard.

Polyspace detects compilation errors during the standard compliance checking stage, which takes place before the verification stage. The compliance checking stage takes about the same amount of time to run as a compiler. Using Polyspace software early in development yields a number of benefits:

- Detection of link errors
- Detection of errors that only appear with two or more files
- Detection of compiler directives that you need to explicitly declare
- Objective, automatic, and early control of development work (possibly to check code into a configuration management system)

Running multiple Polyspace processes

Polyspace Code Prover can be opened simultaneously with Polyspace Bug Finder. However, only one code analysis can be run at a time.

If you try to run multiple Polyspace processes, you will get a `License Error 4,0`. To avoid this error, close any additional Polyspace windows before running an analysis.

Troubleshoot using preprocessed files

In this section...
“Preprocessed Files” on page 9-15
“Troubleshoot Using Preprocessed Files” on page 9-15
“Examples” on page 9-15

Preprocessed Files

Content: The Polyspace software, like other compilers, converts source code to preprocessed code. The preprocessed files have a `.ci` extension. The preprocessed file expands preprocessor directives, including:

- Header files in `#include` statements.
- Macros defined with `#define` statements.
- Conditional compilations defined with `#if`, `#ifdef` or `#ifndef` statements.

Stage	Location of <code>.ci</code> files
Before compilation	<i>Results_folder</i> /ALL/SRC/
After compilation	In a zipped file, <i>ci.zip</i> , in <i>Results_folder</i> /ALL/SRC/MACROS/

Troubleshoot Using Preprocessed Files

To quickly find errors, view the preprocessed code when:

- 1 Your source code includes several header files. Check the preprocessed `*.ci` file to see the header files expanded in one code.
- 2 Your source code contains conditional compilations using `#if`, `#ifdef` or `#ifndef` statements. Check the preprocessed files to find which branch of the conditional statements are active.

Examples

This example shows how to use preprocessed files for troubleshooting.

View Expanded Headers and Macros

The following example uses a source file `Extension.cpp` that:

- Includes a header file `Extension.h`.
- Uses an object-like macro `MAX_VALUE` and a function-like macro `ABS(x)`.
- Uses a conditional compilation statement with the flag `_DEBUG`.

The resulting preprocessed file `Extension.ci`:

- Expands the header file `Extension.h`.
- Replaces the macros `MAX_VALUE` and `ABS(x)` with their contents.
- Replaces the conditional compilation statement based on whether you used the compile flag `_DEBUG`.

Extension.cpp	Extension.h	Partial content of Extension.ci, using compile flag _DEBUG
<pre>#include "Extension.h" Extension::Extension(int val) { num = 0; ABS(val); if (val > MAX_VALUE) num = -1; } #ifdef _DEBUG void Extension::message(char*) {} #else void print(char*) {} #endif #endif</pre>	<pre>#define MAX_VALUE 10 #define ABS(x) ((x)>0?(x):- (x)) class Extension { public: int num; Extension(int val); #ifdef _DEBUG void message(char*); #else void print(char*); #endif };</pre>	<pre># 1 "H:\\Polyspace\\Sources \\PreProcessor\\Extension.cpp" 2 # 1 "H:\\Polyspace\\Sources \\PreProcessor\\Extension.h" 1 class Extension { public: int num; Extension(int val); void message(char*); }; # 2 "H:\\Polyspace\\Sources \\PreProcessor\\Extension.cpp" 2 Extension::Extension(int val) { num = 0; ((val)>0?(val): -(val));</pre>

Extension.cpp	Extension.h	Partial content of Extension.ci, using compile flag _DEBUG
		<pre> if (val > 10) num = -1; } void Extension::message(char*) {} </pre>

Investigate Linking Error

The following example uses two source files, `Child1.cpp` and `Child2.cpp`, that include a header file `Test.h`. Running verification on the two files together causes a linking error because:

- The header file defines a class `Test` that uses a conditional compilation with a `#ifdef` statement. The `#ifdef` statement uses a variable `DEBUG`.
- `DEBUG` is defined in `child1.cpp` but not in `child2.cpp`. This mismatch causes two conflicting definitions of the class `Test`.

Child1.cpp	Child2.cpp	Test.h
<pre> #define DEBUG #include "Test.h" class Child1 : public Test { public: Child1(); Child1(int val); void search(int val); }; </pre>	<pre> #undef DEBUG #include "Test.h" class Child2 : public Test { public: Child2(); Child2(int val); void qshort(int val); protected: int m_status; }; </pre>	<pre> class Test { public: Test(); Test(int val); int getVal(); void setVal(int val); #ifdef DEBUG void algorithm(int val, int max); #endif private: </pre>

Child1.cpp	Child2.cpp	Test.h
		int m_val; };

Error message: For the following error message, the source files are located in H:\Polyspace\Sources\PreProcessor\.

```
File H:\Polyspace\Sources\PreProcessor\Test.h line 1
Error: declaration of function "Test::Test(const Test &)" does not match ..
      function "Test::algorithm" during compilation of ...
      "H:\Polyspace\Sources\PreProcessor\Child2.cpp"
      (one may have been removed due to #define)
```

Preprocessed Files: To find the conflicting definitions of the class Test, compare the two .ci files. Class Test in Child1.ci contains a method algorithm;Child2.ci does not.

Child1.ci	Child2.ci
<pre>.... # 1 "../sources/Test.cpp" 2 # 1 "../sources/test.h" 1 class Test { public: Test(); Test(int val); int getVal(); void setVal(int val); void algorithm(int val, int max); private: int m_val; };</pre>	<pre>.... # 1 "../sources/Child2.cpp" 2 # 1 "../sources/Child2.h" 1 # 1 "../sources/test.h" 1 class Test { public: Test(); Test(int val); int getVal(); void setVal(int val); private: int m_val; };</pre>

Child1.ci	Child2.ci
<pre># 2 "../sources/Test.cpp" 2</pre>	<pre># 2 "../sources/Child2.h" 2</pre>

Check Compilation Before Verification

Before running a verification, you can enable the Compilation Assistant. If the Compilation Assistant detects compilation errors, the software stops the verification. In the Project Manager perspective, on the **Output Summary** tab, the software displays errors and suggests possible solutions.

For more information, see “Check for Compilation Problems” on page 8-4.

Syntax Error

Message

```
Verifying compilation.c  
compilation.c:3: syntax error; found `x' expecting `;'
```

Code Used

```
void main(void)  
{  
int far x;  
x = 0;  
x++;  
}
```

Solution

The `far` keyword is unknown in ANSI C. This causes confusion at compilation time. Should `far` be a variable or a qualifier? The `int far x;` construction is illegal.

Possible corrections include:

- Remove `far` from the source code.
- Define `far` as a qualifier, such as `const` or `volatile`.
- Remove `far` artificially by specifying a compilation flag such as `-D far=` (with a space after the equal sign).

Note To specify `-D` compilation flags that are generic to the project, for efficiency, use the `-include` option. Refer to “Gather Compilation Options Efficiently” on page 6-31.

Undeclared Identifier

Message

```
Verifying compilation.c  
compilation.c:3: undeclared identifier `x'
```

Code Used

```
void main(void)  
{  
  x = 0;  
  x++;  
}
```

Solution

The type is unknown, and therefore the compilation stops. Should `x` be a float, an int, or a char?

Some cross compilers define variables implicitly. Your code must declare variables verification. Polyspace software has no knowledge about implicit variables.

Similarly, some compilers interpret `__SP` as a reference to the stack pointer. Use the `-D` compilation flag.

Note To specify `-D` compilation flags that are generic to the project, for efficiency, use the `-include` option. Refer to “Gather Compilation Options Efficiently” on page 6-31.

Unknown Prototype

Message

Error: function 'myfunc' has unknown prototype

Code Used

```
var = myfunc(s32var1, ptr->s32var2, 24);
```

var, s32var are signed long data types.

Solution

- 1 In an include file, for example, `myinclude.h`, specify the complete prototype for the function:

```
#ifndef _INC_H  
#define _INC_H
```

```
extern signed long myfunc(signed long, signed long, signed long);
```

```
#endif
```

- 2 Rerun your verification with the option `-include myinclude.h`.

No Such File or Folder

Messages

Here are examples of messages that include No such file or folder and catastrophic error: could not open source file:

```
compilation.c:1: one_file.h: No such file or folder
```

```
compilation.c:1: catastrophic error: could not open source file  
"one_file.h" (where one_file.h is an include file)
```

Code Used

```
#include "one_file.h"
```

Solution

The `one_file.h` file is missing.

These files are essential for Polyspace software to complete the compilation, for

- Data coherency
- Automatic stubbing

The Polyspace software must be able to find the include folder that contains this file. Specify the include folder In the Project Manager perspective, or use the `-I` option at the command line, as described in the `-I` reference page.

#error directive

The Polyspace software can terminate during compilation with an unsupported platform `#error`. This error means that the software does not recognize the header data types due to missing compilation flags.

Message

```
#error directive: !Unsupported platform; stopping!
```

Code Used

```
#if defined(__BORLANDC__) || defined(__VISUALC32__)  
# define MYINT int // then use the int type  
#elif defined(__GNUC__) // GCC doesn't support myint  
# define MYINT long // but uses 'long' instead  
#else  
# error !Unsupported platform; stopping!  
#endif
```

Solution

In the Polyspace software, compilation directives must be explicit. In this example, the compilation stops because you did not specify the `__BORLANDC__`, or the `__VISUALC32__`, or the `__GNUC__` compilation flags. To fix this error, in the **Target/Compilation** section, under **Analysis options**, for the **Defined Preprocessor Macros** option, specify one of those three compilation flags and restart the verification.

Class, Array, Struct or Union is Too Large

A verification can terminate during compilation with a message saying that an object is too large. This error means that the software has detected an object that is too big for the pointer size of the selected target.

Messages

- error: array is too large
- error: struct or union is too large
- error: class is too large for pointer type of %d-bits

Code Used

```
struct S
{
    char tab[32728];
}s;
```

When using a 16-bit target (for example: `-target mcpu`)

Solution

Use a larger pointer size.

To select a larger pointer:

- If you are using `-target mcpu`, specify `-pointer-is-32bits`.
- If you are using a specific target, specify `-pointer-is-xxbits` if available, otherwise use a larger target.

Unsupported Non-ANSI Keywords (C)

Code that includes non-ANSI keywords (such as @interrupt) that Polyspace software does not support generate compilation errors. For example, keywords containing @ as a first character cause a compilation error. But in this case, you cannot address the problem by using a compilation flag, nor with a file associated with the -include option.

To address this problem, use the -post-preprocessing-command option.

When you use the -post-preprocessing-command option, write a script or command to replace the unsupported, non-ANSI keyword with a supported keyword. The command must process the standard output from preprocessing and produce its results in accordance with standard output.

The specified script file or command runs just after the preprocessing phase on each source file. The script executes on each preprocessed c file.

Note Preprocessed files have the extension .ci. Preprocessed files are contained in a single compressed file named ci.zip. This file is in the results folder in one of the following locations:

- <results>/ALL/SRC/MACROS/ci.zip
- <results>/C-ALL/ci.zip.

Caution Always preserve the number of lines in a preprocessed .ci file. Adding or removing a line, can result in unpredictable behavior, including changes to the location of checks and MACROS in the Run-Time checks perspective.

Here is an example of such a script file. Save the script in a file named myscript.pl.

```
#!/usr/bin/perl
```

```
binmode STDOUT;

# Process every line from STDIN until EOF
while ($line = <STDIN>)
{
# Replace keyword titi with toto
$line =~ s/titi/toto/g;
# Remove @interrupt (replace with nothing)
$line =~ s/@interrupt/ /g;

# DONT DELTE: Print the current processed line to STDOUT
print $line;
}
```

To run the script on each preprocessed c file, use this command:

```
-post-preprocessing-command MATLAB_Install\sys\perl\win32\bin\perl.exe
<absolute path to myscript.pl>\myscript.pl
```


Initialization of Global Variables (C++)

When a data member of a class is declared static in the class definition, it is a *static member* of the class. Static data members are initialized and destroyed outside the class, as they exist even when no instance of the class has been created.

```
class Test
{
public:

    static int m_number = 0;
};
```

Error message:

```
Verifying test_ko.cpp
/sources/test_ko.cpp, line 4: error: a member with an in-class
initializer must be const
| static int m_number = 0;
|           ^

1 error detected in the compilation of "test_ko.cpp".
```

Corrected code:

in file Test.h	in file Test.cpp
<pre>class Test { public: static int m_number; };</pre>	<pre>int Test::m_number = 0;</pre>

Note Some dialects, other than those supported by Polyspace Code Prover, accept the default initialization of static data member during the declaration.

Double Declarations of Standard Template Library Functions

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, use 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 the verification with the directive for the `list` container:

```
-D __PST_STL_LIST_CONST_ITERATOR_DIFFER_ITERATOR__
```

Large Static Initializer

If you see a large initializer warning during the compilation phase, for example,

```
lsi_eg.c, line 86: warning: initializer is too large, this may cause scaling troubles.  
Please refer to the "Troubleshooting" section of the User Guide  
| const unsigned char CJK_S_MKT_CTS_BE_HDB[] = {  
|                                         ^
```

the compilation might:

- Fail with an error if no memory space is available:
 - Windows — Error: A segmentation fault occurred in "edgcpfe.x86-mingw32.exe"
 - Linux — Error: The process "edgcpfe.x86-linux" received the signal 11.
- Take a long time to complete.

To avoid this issue, rerun your verification with the following option:

```
-cfe-extra-flags --truncate_huge_initializer
```

This option is valid:

- For a C verification only.
- Only if no variable or function address is referenced within the initializer

Compilation Messages Described in This Section

This section describes compiler messages that include the following phrases:

Phrase Found in Message	See
syntax error	“Syntax Error” on page 9-21
undeclared identifier	“Undeclared Identifier” on page 9-22
unknown prototype	“Unknown Prototype” on page 9-23
No such file or folder or Catastrophic error: could not open source file	“No Such File or Folder” on page 9-24
#error: directive	“#error directive” on page 9-25

This section also describes error messages triggered by unsupported keywords. See “Unsupported Non-ANSI Keywords (C)” on page 9-27.

This section includes sample code that triggers the example message.

C++ Dialect Issues

In this section...

“ISO versus Default Dialects” on page 9-33

“CFront2 and CFront3 Dialects” on page 9-35

“Visual Dialects” on page 9-36

“GNU Dialect” on page 9-38

ISO versus Default Dialects

The ISO dialect strictly follows the ISO/IEC 14882:1998 ANSI C++ standard. If you specify the `-dialect iso` option, the Polyspace compiler might produce permissiveness errors. The following code contains five common permissiveness errors that occur if you specify the `-dialect iso` option. These errors are explained in detail following the code.

If you do not specify the `-dialect` option, the Polyspace compiler uses a default dialect that many C++ compilers use; the default dialect is more permissive with regard to the C++ standard.

Original code (file `permissive.cpp`):

```
1
2  class B {} ;
3  class A
4  {
5  friend B ;
6  enum e ;
7  void f() { long float ff = 0.0 ;}
8  enum e { OK = 0, KO } ;
9  };
10 template <class T>
11 struct traits
12 {
13 typedef T * pointer ;
14 typedef T * pointer ;
15 } ;
16 template<class T>
```

```
17 struct C
18 {
19 typedef traits<T>::pointer pointer ;
20 } ;
21 int main()
22 {
23 C<int> c ;
23 }
```

- Using `-dialect iso`, line 5 should be: `friend class B`:

```
"/sources/permissive.cpp", line 5: error: omission of "class"
is nonstandard
    friend B ;
```

- Using `-dialect iso`, the line 6 must be removed:

```
"/sources /permissive.cpp", line 6: error: forward declaration
of enum type
is nonstandard
    enum e ;
    ^
```

- Using `-dialect iso`, line 7 should be: `double ff = 0.0`:

```
"/sources/permissive.cpp", line 7: error: invalid combination
of type
specifiers
    long float ff = 0.0 ;
    ^
```

- Using `-dialect iso`, line 14 needs to be removed:

```
"/sources/permissive.cpp", line 14: error: class member typedef
may not be
redeclared
    typedef T * pointer ; // duplicate !
    ^
```

- Using `-dialect iso`, line 21 needs to be changed by: `typedef typename traits<T>::pointer pointer`

```
"/sources/permissive.cpp", line 21: error: nontype
"traits<T>::pointer [with T=T]" is not a type name
typedef traits<T>::pointer pointer ;
```

These error messages disappear if you specify the `-dialect` default option.

CFront2 and CFront3 Dialects

The `cfront2` and `cfront3` dialects were being used before the publication of the ANSI C++ standard in 1998. Nowadays, these two dialects are used to compile legacy C++ code.

If the `cfront2` or `cfront3` options are not selected, you may get the common error messages below.

Variable Scope Issues

The ANSI C++ standard specifies that the scope of the declarations occurring inside loop definition is local to the loop. However some compilers may assume that the scope is local to the bloc (`{ }`) that contains the loop.

Original code:

```
for (int i = 0; i < maxval; i++) {...}
if (i == maxval) {
    ...
}
```

Error message:

Verifying Test.cpp

```
"/sources/Test.cpp", line 26: error: identifier "i" is undefined
    if (i == maxval) {
        ^
```

Note This kind of construction has been allowed by compilers until 1999, before the standard became more strict.

“bool” Issues

Standard type may need to be turned into boolean type.

Original code:

```
enum bool
{
    FALSE=0,
    TRUE
};
class CBool
{
public:
    CBool ();
    CBool (bool val);
    bool m_val;
};
```

Error message:

```
Verifying C++ sources ...
Verifying CBool.cpp
"../sources/CBool.h", line 4: error: expected either a definition
or a tag name
enum bool
  ^
```

Visual Dialects

The following messages appears if the compiler is based on a Visual® dialect (including visual8).

Import Folder

When a Visual application uses #import directives, the Visual C++ compiler generates a header file that contains some definitions. These header files have a .tlh extension, and Polyspace for C/C++ requires 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). Therefore, in order to provide those files, the application needs to be built first. Then, the option `-import-dir=<build folder>` must be set with the path folder.

pragma Pack

Using a different value with the compile flag (`#pragma pack`) can lead to a linking error message.

Original code:

test1.cpp	type.h	test2.cpp
<pre> #pragma pack(4) #include "type.h" </pre>	<pre> struct A { char c ; int i ; } ; </pre>	<pre> #pragma pack(2) #include "type.h" </pre>

Error message:

```

Pre-linking C++ sources ...
"../sources/type.h", line 2: error: declaration of class "A" had

```

```
a different meaning during compilation of "CPP-ALL/SRC/MACROS/test1.cpp"
(class types do not match)
struct A
  ^
  detected during compilation of secondary translation unit
"CPP-ALL/SRC/MACROS/test2.cpp"
```

The option `-ignore-pragma-pack` is mandatory to continue the verification.

GNU Dialect

For the GNU dialect, you can select the GCC 3.4 or GCC 4.6 version. The GNU dialect supports the keyword `__asm__ __volatile__`, which is used to support inline functions. For example, the `<sys/io.h>` header includes many inline functions. The GNU dialect supports these inline functions.

Polyspace software supports the following GNU elements:

- Variable length arrays
- Anonymous structures:

```
void f(int n) { char tmp[n] ; /* ... */ }
```

```
union A {
  struct {
    double x ;
    double y ;
    double z ;
  };
  double tab[3];
} a ;
```

```
void main(void) {

  assert(&(a.tab[0]) == &(a.x)) ;

}
```

- Other syntactic constructions allowed by GCC, except as noted below

Partial Support

Zero-length arrays have the same support as in Visual Mode. They are allowed when used through a pointer, but not in a local variable.

Syntactic Support Only

Polyspace software provides syntactic support for the following options, but not semantic support:

- `__attribute__(...)` is allowed, but generally not taken into account.
- No special stubs are computed for predeclared functions such as `__builtin_cos`, `__builtin_exit`, and `__builtin_fprintf`.

Not Supported

The following options are not supported:

- The keyword `__thread`
- Statement expressions:

```
int i = ({ int tmp ; tmp = f() ; if (tmp > 0 ) { tmp = 0 ; } tmp ; })
```
- Taking the address of a label:

```
{ L : void *a = &&L ; goto *a ; }
```
- General C99 features supported by default in GCC, such as complex built-in types (`__complex__`, `__real__`, etc.).
- Extended designators initialization:

```
struct X { double a; int b[10] } x = { .b = { 1, [5] =2 },  
.b[3] = 1, .a = 42.0 };
```
- Nested functions

Examples

Example 1: `__asm__ __volatile__` keyword

In the following example, for the `inb_p` function to manage the return of the local variable `_v`, the `__asm__ __volatile__` keyword is used as follows:

```
extern inline unsigned char
inb_p (unsigned short port)
{
    unsigned char _v;

    __asm__ __volatile__ ("inb %w1,%0\noutb %a1,$0x80":"=a"
                          (_v):"Nd" (port));
    return _v;
}
...
```

Example 2: Anonymous Structure

The following example shows an unnamed structure supported by GNU:

```
class x
{
public:

    struct {
        unsigned int a;
        unsigned int b;
        unsigned int c;
    };
    unsigned short pcia;
    enum{
        ea = 0x1,
        eb = 0x2,
        ec = 0x3
    };

    struct {
        unsigned int z1;
        unsigned int z2;
    };
};
```

```
    unsigned int z3;
    unsigned int z4;
};

int main(int argc, char *argv[])
{
    class x myx;

    myx.a = 10;
    myx.z1 = 11;
    return(0);
}
```

C Link Errors

In this section...

“Link Error Overview (C)” on page 9-42
“Function: Procedure Multiply Defined” on page 9-43
“Function: Wrong Argument Type” on page 9-43
“Function: Wrong Argument Number” on page 9-44
“Function: Wrong Return Type” on page 9-45
“Variable: Wrong Type” on page 9-45
“Variable: Signed/Unsigned” on page 9-46
“Variable: Different Qualifier” on page 9-46
“Variable: Array Against Variable” on page 9-47
“Variable: Wrong Array Size” on page 9-47
“Missing Required Prototype for varargs” on page 9-48

Link Error Overview (C)

This section describes how to address some common types of link errors for C code.

Link errors result from the checking that Polyspace performs for compliance with ANSI C standards. Link error messages can apply to functions, variables, and varargs.

The error message includes specific information that reflects the code that the Polyspace software is checking, such as the function name and type declaration.

Examining Preprocessed Code

Looking at the preprocessed code can help you to find link errors faster.

Preprocessed files have the extension `.ci`. Preprocessed files are contained in a single compressed file named `ci.zip`. This file is in the `results` folder in one of the following locations:

- `<results>/ALL/SRC/MACROS/ci.zip`
- `<results>/C-ALL/ci.zip`.

Function: Procedure Multiply Defined

Files Used

<code>header.h</code>	<code>file1.c</code>	<code>file2.c</code>
<pre>#include <stdio.h> void func() { }</pre>	<pre>#include "header.h"</pre>	<pre>#include "header.h"</pre>

Polyspace Output

```
Error:
procedure func multiply defined
```

Solution

For C code, to allow such multiple inclusion of the header containing the function body, use the option `-static-headers-object`.

Function: Wrong Argument Type

Polyspace Output

```
Error:
global declaration of 'f' function has incompatible type with its definition
Declared function type has 'arg 1' type incompatible with definition.
```

```
int f(float y)          int f(int *y);
```

```
{
  int r;
  r=12;
}

void main(void)
{
  int r;
  r = f(&r);
}
```

Solution

The first parameter for the `f` function is either a float or a pointer to an integer. The global declaration must match the definition.

Function: Wrong Argument Number

Polyspace Output

Error:

global declaration of 'f' function has incompatible type with its definition
Declared function type has incompatible number of arguments with definition.

```
int f(float y)
{
  int r;
  r=12;
}

int f(float y, float x);

void main(void)
{
  int a;
  float b, c;
  a = f(b, c);
}
```

Solution

These two functions have a different number of arguments. This mismatch in the number of arguments results in a nondeterministic execution.

Function: Wrong Return Type

Polyspace Output

Verifying cross-files ANSI C compliance ...

Error: global declaration of 'f' function has incompatible type with its definition
declared function type has incompatible return type with definition
declared 'float' (size 64) type incompatible with defined 'int' (size 32) type

```
float f(int y)          int f(int y);  
{  
float r;                void main(void)  
r=1.0;                  {  
return r;               int r;  
}                        r = f(r);  
                        }  
                        }
```

Solution

Use the same return type for the declaration and definition of function f.

Variable: Wrong Type

Polyspace Output

Verifying cross-files ANSI C compliance ...

Error: global declaration of 'x' variable has incompatible type with its definition
declared 'float' (32) type incompatible with defined 'int' (32) type

```
extern float x          int x;  
                        void main(void)  
                        {}  
                        }
```

Solution

Declare the x variable the same way in every file. If a variable x is as an integer equal to 1, which is 0x0001, what does this value mean when seen as a float? It could result in a NaN (Not a Number) during execution.

Variable: Signed/Unsigned

Polyspace Output

```
Verifying cross-files ANSI C compliance ...  
Error: global declaration of 'x' variable has incompatible type with its definition  
      declared 'unsigned' type incompatible with defined 'signed' type
```

```
extern unsigned char x;      char x;  
                             void main(void)  
                             {}
```

Solution

Consider the 8-bit binary value 10000010. Given that a char is 8 bits, it is not clear whether it is 130 (unsigned), or maybe -126 (signed).

Variable: Different Qualifier

Polyspace Output

```
Verifying cross-files ANSI C compliance ...  
Warning: global declaration of 'x' variable has incompatible type with its definition  
        declared 'non qualified' type incompatible with defined 'volatile' type  
        'volatile' qualifier used
```

```
extern int x;                volatile int x;  
  
                             void main(void)  
                             {}
```

Solution

Polyspace software flags the volatile qualifier, because that qualifier has major implications for the verification. Because it is not clear which statement is correct, the verification process generates a warning.

Variable: Array Against Variable

Polyspace Output

Verifying cross-files ANSI C compliance ...

Error: global declaration of 'x' variable has incompatible type with its definition
declared 'array' (384) type incompatible with defined 'int' (32) type

```
extern int x[12];          int x;

                           void main(void)
                           {
                               }

```

Solution

The real allocated size for the x variable is one integer. Any function attempting to manipulate x[] corrupts memory.

Variable: Wrong Array Size

Polyspace Output

Verifying cross-files ANSI C compliance ...

Warning: global declaration of 'x' variable has incompatible type with its definition
declared array type has 'upper bound' 5 inferior to definition 'upper bound' 12

```
extern int x[12];          int x[5];

                           void main(void)
                           {
                               }

```

Solution

The real allocated size for the x variable is five integers. Any function attempting to manipulate x[] between x[5] and x[11] corrupts memory.

Missing Required Prototype for varargs

Polyspace Output

Verifying cross-files ANSI C compliance ...
Error: missing required prototype for varargs. procedure 'g'.

```
void g(int, ...);          void main(void)
                           {
void f(void)              g(4);
{                          }
g(12, abcde ,40)
}
```

Solution

Declare the prototype for `g` when `main` executes.

To eliminate this error, you can add the following line to `main`:

```
void g(int, ...)
```

Or, you can avoid modifying `main` by adding that same line in a new file and then when you launch the verification, use the `-include` option:

```
include c:\Polyspace\new_file.h
```

where `new_file.h` is the new file that includes the line `void g(int, ...)`.

C++ Link Errors

In this section...
“STL Library C++ Stubbing Errors” on page 9-49
“Lib C Stubbing Errors” on page 9-50

STL Library C++ Stubbing Errors

Polyspace software provides an efficient implementation of all functions in the Standard Template Library (STL). The STL and platforms may have different declarations and definitions; otherwise, the following error messages appear:

Original code:

```
#include <map>

struct A
{
    int m_val;
};

struct B
{
    int m_val;
    B& operator=(B &) ;
};

typedef std::map<A, B> MAP ;

int main()
{
    MAP m ;
    A a ;
    B b ;

    m.insert(std::make_pair(a,b)) ;
}
```

Error message:

```
Verifying template.cpp
"<Product>/Verifier/cinclude/new_stl/map", line 205: error: no operator
"=" matches these operands
operand types are: pair<A, B> = const map<A, B, less<A>>::value_type
{ volatile int random_alias = 0 ; if (random_alias) *((pair<Key, T> * )
_pst_elements) = x ; } ; // read of x is done here

detected during instantiation of
"pair<__pst_generic_iterator<bidirectional_iterator_tag, pair<const Key,
T>>, bool> map<Key, T, Compare>::insert(const map<Key, T, Compare>::
value_type &) [with Key=A, T=B, Compare=less<A>]" at line 23 of "/cygdrive/
c/_BDS/Test-Polyspace/sources/template.cpp"
```

Using the `-no-stub-stl` option avoids this error message. Then, you need to add the folder containing definitions of own STL library as a folder to include using the option `-I`.

The preceding message can also appear with the folder names:

```
"<Product>/cinclude/new_stl/map", line 205:
error: no operator "="
matches these operands
```

```
"<Product>/cinclude/pst_stl/vector", line 64: error: more than one
operator "=" matches these operands:
```

Be careful that other compile or linking troubles can appear with your own template definitions.

Lib C Stubbing Errors

Extern C Functions

Some functions may be declared inside an `extern C { }` bloc 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 `siglongjmp`.

Standard Library Function Stubbing Errors

In this section...

“Conflicts Between Library Functions and Polyspace Stubs” on page 9-53

“_polyspace_stdstubs.c Compilation Errors” on page 9-53

“Troubleshooting Approaches for Standard Library Function Stubs” on page 9-55

“Restart with the -I option” on page 9-55

“Replace Automatic Stubbing with Include Files” on page 9-56

“Create _polyspace_stdstubs.c File with Required Includes” on page 9-57

“Provide .c file Containing Prototype Function” on page 9-58

“Ignore _polyspace_stdstubs.c” on page 9-59

Conflicts Between Library Functions and Polyspace Stubs

A code set compiles successfully for a target, but during the `_polyspace_stdstubs.c` compilation phase for the same code, Polyspace software generates an error message.

The error message highlights conflicts between:

- A standard library function that the application includes
- One of the standard stubs that Polyspace software uses in place of the function

For more information about errors generated during automatic stub creation, see “Automatic Stubbing Errors” on page 9-60.

`_polyspace_stdstubs.c` Compilation Errors

Here are examples of the errors relating to stubbing standard library functions. The code uses standard library functions such as `sprintf` and `strcpy`, illustrating possible problems with these functions.

Example 1

C-STUBS/___polyspace__stdstubs.c:1117: string.h: No such file or folder

Verifying C-STUBS/___polyspace__stdstubs.c

C-STUBS/___polyspace__stdstubs.c:1118: syntax error; found `strlen' expecting `;'

C-STUBS/___polyspace__stdstubs.c:1120: syntax error; found `i' expecting `;'

C-STUBS/___polyspace__stdstubs.c:1120: undeclared identifier `i'

Example 2

Verifying C-STUBS/___polyspace__stdstubs.c

Error: missing required prototype for varargs. procedure 'sprintf'.

Example 3

Verifying C-STUBS/___polyspace__stdstubs.c

C-STUBS/___polyspace__stdstubs.c:3027: missing parameter 4 type

C-STUBS/___polyspace__stdstubs.c:3027: syntax error; found `n' expecting `)'

C-STUBS/___polyspace__stdstubs.c:3027: skipping `n'

C-STUBS/___polyspace__stdstubs.c:3037: undeclared identifier `n'

Troubleshooting Approaches for Standard Library Function Stubs

You can use a range of techniques to address errors relating to stubbing standard library functions. These techniques reflect different balances for the verification between:

- Precision
- Amount of time preparing the code
- Execution time

Try the techniques in any order. Consider trying the simplest approaches first, and trying other techniques as required to achieve the balance of the trade-offs that you seek. Here are the techniques, listed in order of estimated simplicity, from simplest to most thorough:

- “Restart with the `-I` option” on page 9-55
- “Replace Automatic Stubbing with Include Files” on page 9-56
- “Create `_polyspace_stdstubs.c` File with Required Includes” on page 9-57
(Use when precision is important enough to justify extensive code preparation time)
- “Provide `.c` file Containing Prototype Function” on page 9-58
(Use when you do not want to invest much time for code preparation time)
- “Ignore `_polyspace_stdstubs.c`” on page 9-59

If the problem persists after trying these solutions, contact MathWorks support.

Restart with the `-I` option

Generally you can best address stubbing errors by restarting the verification. Include the header file containing the prototype and the required definitions, as used during compilation for the target.

The least invasive way of including the header file containing the prototype is to use the `-I` option.

Replace Automatic Stubbing with Include Files

The Polyspace software provides a selection of files that contain stubs for most standard library functions. You can use those stubs in place of automatic stubbing.

For replacement of stubbing to work effectively, provide the include file for the function. In the following example, the standard library function is `strlen`. This example assumes that you have included `string.h`. Because the `string.h` file can differ between targets, there are no default include folders for Polyspace stub files.

If the compiler has implicit include files, manually specify those include files, as shown in this example.

```
(_polyspace_stdstubs.c located in <<results_dir>>/C-ALL/C-STUBS)

_polyspace_stdstubs.c
#if defined(_polyspace_strlen) || ... || defined(_polyspace_strtok)
#include <string.h>
size_t strlen(const char *s)
{
    size_t i=0;
    while (s[i] != 0)
        i++;
    return i;
}
#endif /* _polyspace_strlen */
```

If problems persist, try one of these solutions:

- “Create `_polyspace_stdstubs.c` File with Required Includes” on page 9-57
- “Provide `.c` file Containing Prototype Function” on page 9-58
- “Ignore `_polyspace_stdstubs.c`” on page 9-59

Create `_polyspace_stdstubs.c` File with Required Includes

- 1 Copy `<<results_dir>>/C-ALL/C-STUBS/_polyspace_stdstubs.c` to the sources folder and rename it `polyspace_stdstubs.c`.

This file contains the whole list of stubbed functions, user functions, and standard library functions. For example:

```
#define _polyspace_strlen
#define a_user_function
```

- 2 Find the problem function in the file. For example:

```
#if defined(_polyspace_strlen) || ... || defined(_polyspace_strtok)
#include <string.h>
size_t strlen(const char *s)
{
    size_t i=0;
    while (s[i] != 0)
        i++;
    return i;
}
#endif /* __polyspace_strlen */
```

The verification requires you to include the `string.h` file that the application uses.

- 3 Provide the `string.h` file that contains the real prototype and type definitions for the stubbed function.

Alternatively, extract the relevant part of that file for inclusion in the verification.

For example, for `strlen`:

```
string.h
// put it in the /homemade_include folder
typedef int size_t;
size_t strlen(const char *s);
```

- 4 Specify the path for the include files and relaunch Polyspace, using one of these commands:

```
polyspace-code-prover-nodesktop -I /homemade_include
```

or

```
polyspace-code-prover-nodesktop -I /our_target_include_path
```

Provide .c file Containing Prototype Function

- 1 Identify the function causing the problem (for example, `printf`).
- 2 Add a `.c` file to your verification containing the prototype for this function.
- 3 Restart the verification either from the Project Manager perspective or from the command line.

You can find other `__polyspace_no_<function_name>` options in `__polyspace__stdstubs.c` files, such as:

```
__polyspace_no_vprintf  
__polyspace_no_vsprintf  
__polyspace_no_fprintf  
__polyspace_no_fscanf  
__polyspace_no_printf  
__polyspace_no_scanf  
__polyspace_no_sprintf  
__polyspace_no_sscanf  
__polyspace_no_fgetc  
__polyspace_no_fgets  
__polyspace_no_fputc  
__polyspace_no_fputs  
__polyspace_no_getc
```

Note If you are considering defining multiple project generic `-D` options, using the `-include` option can provide a more efficient solution to this type of error. Refer to “Gather Compilation Options Efficiently” on page 6-31.

Ignore `_polyspace_stdstubs.c`

When all other troubleshooting approaches have failed, you can try ignoring `_polyspace_stdstubs.c`. To ignore `_polyspace_stdstubs.c`, but still see which standard library functions are in use:

1 Do one of the following:

- Deactivate all standard stubs using `-D POLYSPACE_NO_STANDARD_STUBS` option. For example:

```
polyspace-code-prover-nodesktop -D  
POLYSPACE_NO_STANDARD_STUBS
```

- Deactivate all stubbed extensions to ANSI C standard by using `-D POLYSPACE_STRICT_ANSI_STANDARD_STUBS`. For example:

```
polyspace-code-prover-nodesktop -D  
POLYSPACE_STRICT_ANSI_STANDARD_STUBS
```

This approach presents a list of functions Polyspace software tries to stub. It also lists the standard functions in use (most probably without a prototype), and generates the following type of message:

```
* Function strcpy may write to its arguments and may  
return parts of them. Does not model pointer effects.  
Returns an initialized value.
```

```
Fatal error: function 'strcpy' has unknown prototype
```

2 Add an include file in the C file that uses your standard library function. If you restart Polyspace with the same options, the default behavior results for these stubs for this particular function.

Consider the example `size_t strcpy(char *s, const char *i)` stubbed to

- Write anything in `*s`
- Return any possible `size_t`

Automatic Stubbing Errors

In this section...
“Three Types of Error Messages” on page 9-60
“Unknown Prototype Error” on page 9-60
“Parameter -entry-points Error” on page 9-60

Three Types of Error Messages

The Polyspace software generates three different types of error messages during the automatic creation of stubs.

For more information about stubbing errors, see “Standard Library Function Stubbing Errors” on page 9-53.

Unknown Prototype Error

Message

```
Fatal error: function 'f' has unknown prototype
```

```
-----
```

```
Error message explanation:
```

- "function has wrong prototype" means that either the function has no prototype or its prototype is not ANSI compliant.
- "task is undefined" means that a function has been declared to be a task but has no known body

Solution

Provide an ANSI-compliant prototype.

Parameter -entry-points Error

Message

```
*** Verifier found an error in parameter -entry-points: task "w"
```



```
must be a userdef function
-----
---
--- Found some errors in launching command. ---
--- Please consult rte-kernel -h to correct them ---
--- and launch the verification again. ---
---
-----
```

Solution

A function or procedure declared to be an `-entry-points` cannot be an automatically stubbed function.

Reduce Verification Time

In this section...

“Factors Affecting Verification Time” on page 9-62

“Techniques to Improve Verification Performance” on page 9-62

“Tune Polyspace Parameters” on page 9-65

“Subdivide Code” on page 9-66

“Reduce Procedure Complexity” on page 9-76

“Reduce Task Complexity” on page 9-78

“Reduce Variable Complexity” on page 9-78

“Choose Lower Precision” on page 9-79

Factors Affecting Verification Time

These factors affect how long it takes to run a verification:

- The size of the code
- The number of global variables
- The nesting depth of the variables (the more nested they are, the longer it takes)
- The depth of the call tree of the application
- The intrinsic complexity of the code, particularly with regards to pointer manipulation

Because many factors affect verification time, there is no precise formula for calculating verification duration. Instead, Polyspace software provides graphical and textual output to indicate how the verification is progressing.

Techniques to Improve Verification Performance

This section suggests methods to reduce the duration of a particular verification, with minimal compromise for the launch parameters or the precision of the results.

You can increase the size of a code sample for effective analysis by tuning the tool for that sample. Beyond that point, subdividing the code or choosing a lower precision level offers better results (-01, -00).

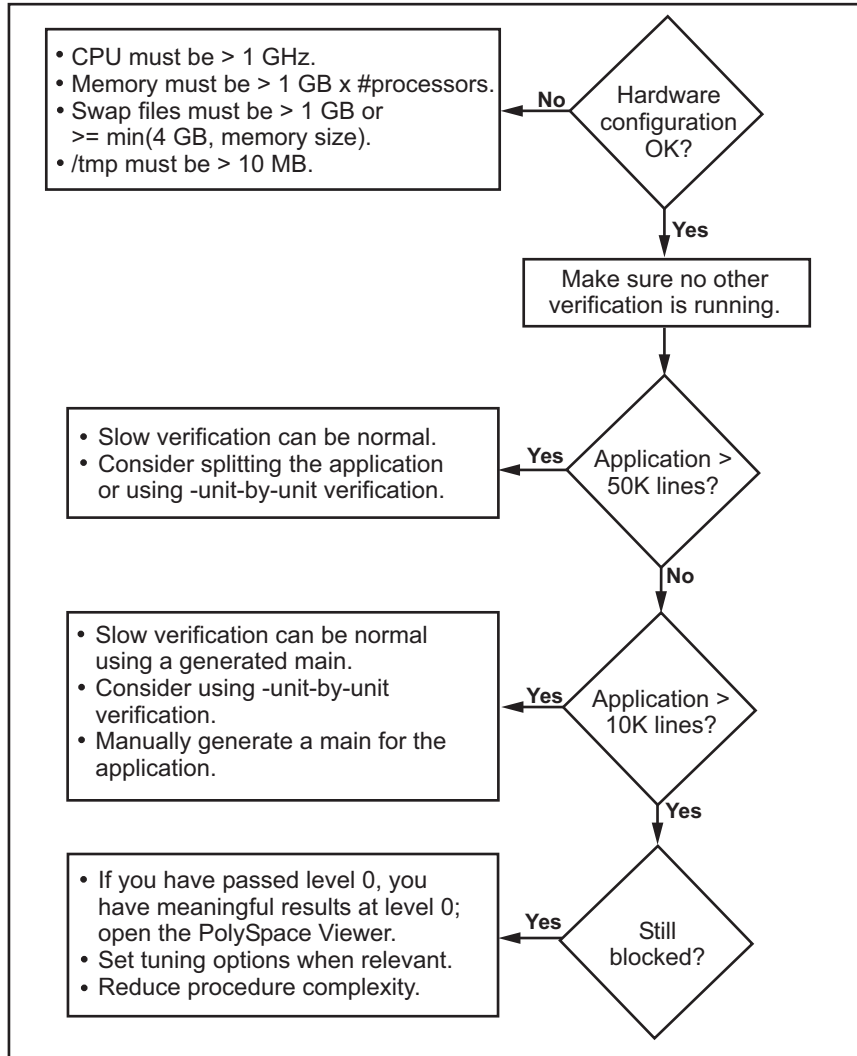
You can use several techniques to reduce the amount of time required for a verification, including

- “Errors due to disk defragmentation and antivirus software” on page 9-10
- “Tune Polyspace Parameters” on page 9-65
- “Subdivide Code” on page 9-66
- “Reduce Procedure Complexity” on page 9-76
- “Reduce Task Complexity” on page 9-78
- “Reduce Variable Complexity” on page 9-78
- “Choose Lower Precision” on page 9-79

You can combine these techniques. See the following performance-tuning flow charts:

- “Standard Scaling Options Flow Chart” on page 9-64
- “Reduce Code Complexity” on page 9-65

Standard Scaling Options Flow Chart



Reduce Code Complexity

To reduce code complexity, try the following techniques, in the order listed:

- “Reduce Procedure Complexity” on page 9-76
- “Reduce Task Complexity” on page 9-78
- “Reduce Variable Complexity” on page 9-78

After you use any of these techniques, restart the verification.

Tune Polyspace Parameters

Impact of Parameter Settings

Compromise to balance the time required to perform a verification and the time required to review the results. Launching Polyspace verification with the following options reduces the time taken for verification. However, these parameter settings compromise the precision of the results. The less precise the results of the verification, the more time you can spend reviewing the results.

Recommended Parameter Tuning

Use the parameters in the sequence listed. If the first suggestion does not increase the speed of verification sufficiently, then introduce the second, and so on.

- Switch from `-O2` to a lower precision.
- Set the `-respect-types-in-globals` and `-respect-types-in-fields` options.
- Set the `-k-limiting` option to 2, then 1, or 0.
- Manually stub missing functions which write into their arguments.
- If some code uses some large arrays, use the `-no-fold` option.

For example:

```
polyspace-code-prover-nodesktop -00 -respect-types-in-globals  
-k-limiting 0
```

Subdivide Code

- “An Ideal Application Size” on page 9-66
- “Benefits of Subdividing Code” on page 9-66
- “Possible Issues with Subdividing Code” on page 9-67
- “Approach” on page 9-68
- “Select a Subset of Code” on page 9-70

An Ideal Application Size

People have used Polyspace software to analyze numerous applications with greater than 100,000 lines of code.

There is a trade-off between the time and resources required to analyze an application, and the resulting selectivity. The larger the project size, the broader the approximations Polyspace software makes. Broader approximations produce more oranges. Large applications can require you to spend much more time analyzing the results and your application.

These approximations enable Polyspace software to extend the range of project sizes it can manage, to perform the verification further, and to solve traditionally incomputable problems. Balance the benefits derived from verifying a whole large application against the loss of precision that results.

Benefits of Subdividing Code

Subdividing a large application into smaller subsets of code provides several benefits. You:

- Quickly isolate a meaningful subset
- Keep all functional modules
- Can maintain a high precision level (for example, level O2)
- Reduce the number of orange items

- Do not have to remove threads that affect shared data
- Reduce the code complexity considerably

Possible Issues with Subdividing Code

Subdividing code can lead to these problems:

- Orange checks can result from a lack of information regarding the relationship between modules, tasks, or variables.
- Orange checks can result from using too wide a range of values for stubbed functions.
- Some loss of precision; the verification consider all possible values for a variable.

When the Application is Incomplete. When the code consists of a small subset of a larger project, Polyspace software automatically stubs many procedures. Polyspace bases the stubbing on the specification or prototype of the missing functions. Polyspace verification assumes that all possible values for the parameter type are returnable.

Consider two 32-bit integers a and b , which are initialized with their full range due to missing functions. Here, $a*b$ causes an overflow, because a and b can be equal to 2^{31} . Precise stubbing can reduce the number of incidences of these data set issue **orange checks**.

Now consider a procedure f that modifies its input parameters a and b . f passes both parameters by reference. Suppose a can be from 0 through 10, and b any value between -10 and 10. In an automatically stubbed function, the combination $a=10$ and $b=10$ is possible, even if it is not possible with the real function. This situation introduces orange checks in a code snippet such as $1/(a*b - 100)$, where the division would be **orange**.

- So, even with precise stubbing, verification of a small section of code can introduce extra orange checks. However, the net effect from reducing the complexity is to reduce the total number of orange checks.
- With default stubbing, the increase in the number of orange checks as the result of this phenomenon tends to be more pronounced.

Considering the Effects of Application Code Size. Polyspace can make approximations when computing the possible values of the variables, at any point in the program. Such an approximation use a superset of the actual possible values.

For instance, in a relatively small application, Polyspace software can retain detailed information about the data at a particular point in the code. For example, the variable VAR can take the values

$$\{-2 ; 1 ; 2 ; 10 ; 15 ; 16 ; 17 ; 25 \}$$

If the code uses VAR to divide, the division is green (because 0 is not a possible value).

If the program is large, Polyspace software simplifies the internal data representation by using a less precise approximation, such as:

$$[-2 ; 2] \cup \{10\} \cup [15 ; 17] \cup \{25\}$$

Here, the same division appears as an orange check.

If the complexity of the internal data becomes even greater later in the verification, Polyspace can further simplify the VAR range to (for example):

$$[-2 ; 20]$$

This phenomenon increases the number of orange warnings when the size of the program becomes large.

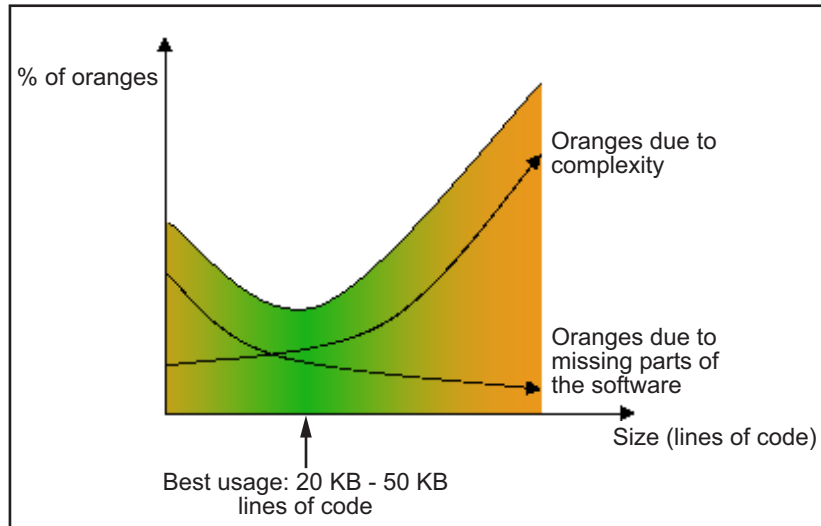
Approach

Begin with file-by-file verifications (when dealing with C language), package-by-package verifications (when dealing with Ada language), and class-by-class verifications (when dealing with C++ language).

The maximum application size is between 20,000 (for C++) and 50,000 lines of code (for C and Ada). For such applications of that size, approximations are not too significant. However, sometimes verification time is extensive.

Experience suggests that subdividing an application before verification normally has a beneficial impact on selectivity. The verification produces

more red, green and gray checks, and fewer unproven orange checks. This subdivision approach makes bug detection more efficient.



A compromise between selectivity and size

Polyspace verification is most effective when you use it as early as possible in the development process, before any other form of testing.

When you analyze a small module (for example, a file, piece of code, or package) using Polyspace software, focus on the red and gray checks. orange unproven checks at this stage are interesting, because most of them deal with robustness of the application. The orange checks change to red, gray, or green as the project progresses and you integrate more modules.

In the integration process, code can become so large (50,000 lines of code or more). This amount of code can cause the verification to take an unreasonable amount of time. You have two options:

- Stop using Polyspace verification at this stage (you have gained many benefits already).
- Analyze subsets of the code.

Select a Subset of Code

Subdividing a project for verification takes considerably less verification time for the sum of the parts than for the whole project considered in one pass. Consider data flow when you subdivide the code.

Consider two distinct concepts:

- **Function entry-points** — Function entry-points refer to the Polyspace execution model, because they start concurrently, without assumptions regarding sequence or priority. They represent the beginning of your call tree.
- **Data entry-points** — Regard lines in the code that acquire data as data entry points.

Example 1

```
int complete_treatment_based_on_x(int input)
{
    thousand of line of computation...
}
```

Example 2

```
void main(void)
{
    int x;
    x = read_sensor();
    y = complete_treatment_based_on_x(x);
}
```

Example 3

```
#define REGISTER_1 (*(int *)0x2002002)
void main(void)
{
    x = REGISTER_1;
    y = complete_treatment_based_on_x(x);
}
```

In each case, the x variable is a data entry point and y is the consequence of such an entry point. y can be formatted data, due to a complex manipulation of x .

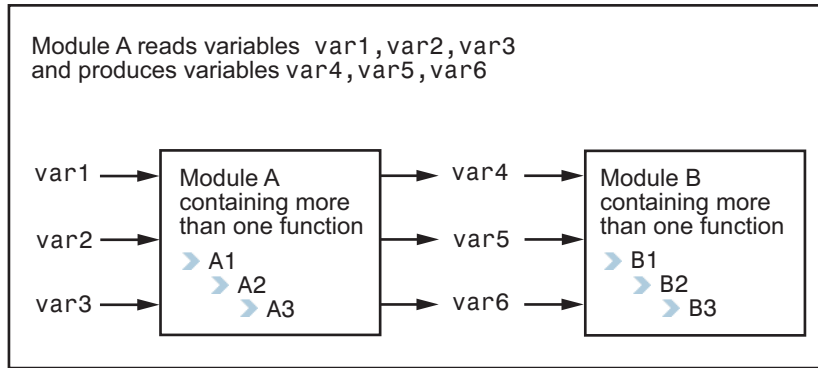
Because x is volatile, a probable consequence is that y contains all possible formatted data. You could remove the procedure `complete_treatment_based_on_x` completely, and let automatic stubbing work. The verification process considers y as potentially taking any value in the full range data.

```
//removed definition of complete_treatment_based_on_x
void main(void)
{
    x = ... // what ever
    y = complete_treatment_based_on_x(x); // now stubbed!
}
```

Typical Examples of Removable Components, According to the Logic of the Data. Here are some examples of removable components, based on the logic of the data:

- **Error management modules** often contain a large array of structures accessed through an API, but return only a Boolean value. Removing the API code and retaining the prototype causes the automatically generated stub to return a value in the range $[-2^{31}, 2^{31} - 1]$, which includes 1 and 0. Polyspace considers the procedure able to return all possible values.
- **Buffer management for mailboxes coming from missing code** – Suppose an application reads a huge buffer of 1024 char. The application then uses the buffer to populate three small arrays of data, using a complicated algorithm before passing it to the main module. If the verification excludes the buffer, and initializes the arrays with random values instead, then the verification of the remaining code is just the same.
- Display modules

Subdivision According to Data Flow. Consider the following example.



In this application, var1, var2, and var3 can vary between the following ranges:

var1	From 0 through 10
var2	From 1 through 100
var3	From -10 through 10

Module A consists of an algorithm that interpolates between var1 and var2. That algorithm uses var3 as an exponential factor, so when var1 is equal to 0, the result in var4 is also equal to 0.

As a result, var4, var5, and var6 have the following specifications:

Ranges	var4 var5 var6	Between -60 and 110 From 0 through 12 From 0 through 100
Properties	And a set of properties between variables	<ul style="list-style-type: none"> • If var2 is equal to 0, then var4 > var5 > 5. • If var3 is greater than 4, then var4 < var5 < 12 • ...

Subdivision in accordance with data flow allows you to analyze modules A and B separately:

- A uses var1, var2, and var3, initialized respectively to [0;10], [1;100], and [-10;10].
- B uses var4, var5, and var6, initialized respectively to [-60;110], [0;12], and [-10;10].

The consequences are:

- A slight loss of precision on the B module verification, because now Polyspace considers all combinations for var4, var5, and var6. It includes all possible combinations, even those combinations that the module A verification restricts.

For example, if the B module included the test

```
If var2 is equal to 0, then var4 > var5 > 5
```

then the dead code on any subsequent else clause is undetected.

- An in-depth investigation of the code is not required to isolate a meaningful subset. It means that a logical split is possible for an application, in accordance with the logic of the data.
- The results remain valid, because there is no requirement to remove, for example, a thread that changes shared data.
- The code is less complex.
- You can maintain the maximum precision level.

Typical examples of removable components:

- Error management modules. A function `has_an_error_already_occurred` can return TRUE or FALSE. Such a module can contain a large array of structures accessed through an API. Removing API code with the retention of the prototype results in the Polyspace verification producing a stub that returns $[-2^{31}, 2^{31}-1]$. That result clearly includes 1 and 0 (yes and no). The procedure `has_an_error_already_occurred` returns all possible values.

- Buffer management for mailboxes coming from missing code. Suppose the code reads a large buffer of 1024 char and then collates the data into three small arrays of data, using a complicated algorithm. It then gives this data to a main module for treatment. For the verification, Polyspace can remove the buffer and initialize the arrays with random values.
- Display modules.

Subdivide According to Real-Time Characteristics. Another way to split an application is to isolate files which contain only a subset of tasks, and to analyze each subset separately.

If a verification initiates using only a few tasks, Polyspace loses information regarding the interaction between variables.

Suppose an application involves tasks T1 and T2, and variable x.

If T1 modifies x and reads it at a particular moment, the values of x affect subsequent operations in T2.

For example, consider that T1 can write either 10 or 12 into x and that T2 can both write 15 into x and read the value of x. Two ways to achieve a sound standalone verification of T2 are:

- You could declare x as volatile to take into account all possible executions. Otherwise, x takes only its initial value or x variable remains constant, and verification of T2 is a subset of possible execution paths. You can get precise results, but it includes one scenario among all possible states for the variable x.
- You could initialize x to the whole possible range [10;15], and then call the T2 entry-point. Use this approach if x is calibration data.

Subdivide According to Files. This method is simple, but it can produce good results when you are trying to find defects in gray code.

Simply extract a subset of files and perform a verification using one of these approaches:

- Use entry points.
- Create a `main` that calls randomly functions that the subset of the code does not call.

Reduce Procedure Complexity

If the log file does not display any messages for several hours, you probably have a scaling issue. You can reduce the complexity of some of the procedures by cloning the calling context for specific procedures. One way to reduce complexity is to specify the `-inline` option on procedures whose names appear in the log file in one or both of two lists.

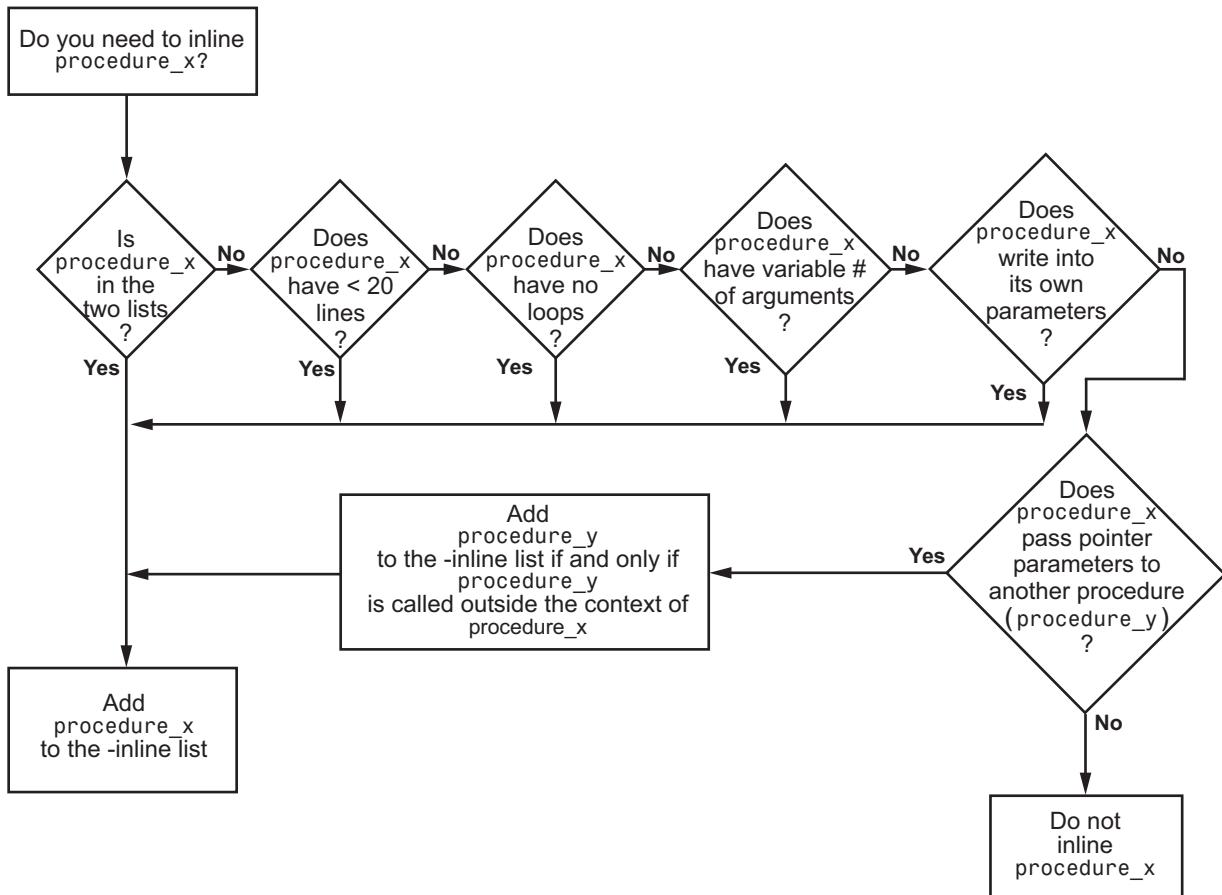
The `-inline` option creates clones of each specified procedure for each call to it. This option reduces the number of aliases in a procedure, and can improve precision in some situations.

Suppose that the log file contains two lists that look like the following:

```
%%% BEGIN PRE%%%  
* inlining procedure_1 could decrease the number of aliases of parameter #3 from 752  
                                                                    to 3  
* inlining procedure_2 could decrease the number of aliases of parameter #3 from 2687  
                                                                    to 3  
* inlining procedure_3 could decrease the number of aliases of parameter #4 from 1542  
                                                                    to 4  
  
%%%END PRE%%%  
  
%%% BEGIN PRE%%%  
  
procedures that write the biggest sets of aliases: procedure_4 (2442),  
                                                    procedure_2 (1120),  
procedure_5 (500)  
  
%%%END PRE%%%
```

Looking at this example log file, `procedure_1` through `procedure_5` are good candidates to be inlined.

Follow the steps on this flow chart to determine which `procedure_x` must be inlined, that is, for which `procedure_x` you need to specify the `-inline` option.



Here are three example situations:

- Using the preceding log file, inline `procedure_2` because it appears in both lists. In addition, if it has no loops, inline `procedure_5`.
- Inline procedures that have a variable number of arguments, such as `printf` and `sprintf`.
- In the following examples, consider whether each procedure, `procedure_x`, passes its pointer parameters to another procedure.

Does this procedure pass pointer parameters?		
Yes	No	No
<pre>void procedure_x(int *p) { procedure_y(p) }</pre>	<pre>void procedure_x(int q)</pre>	<pre>void procedure_x(int *r) { *r = 12 }</pre>

Exercise caution when you inline procedures. Inlining duplicates code and can drastically increase the number of lines of code, resulting in increased computation time.

For example, suppose `procedure_2` has 30 lines of codes and is called 30 times; `procedure_5` has 100 lines of code and is called 50 times. The number of lines of code becomes more than 5000 lines, so computation time increases.

Reduce Task Complexity

If the code contains two or more tasks, and particularly if there are more than 10,000 alias reads, set the option **Reduce task complexity** (`-lightweight-thread-model`). This option reduces:

- Task complexity
- Verification time

However, using this option causes more oranges and a loss of precision on reads of shared variables through pointers.

Reduce Variable Complexity

Variable Characteristic	Action
The types are complex.	Set the <code>-k-limiting [0-2]</code> option. Begin with 0. Go up to 1, or 2 in order to gain precision.
There are large arrays	Set the <code>-no-fold</code> option.

Choose Lower Precision

The amount of simplification applied to the data representations depends on the required precision level (O0, O2), Polyspace software adjusts the level of simplification. For example:

- -00 — shorter computation time
- -02 — less orange warnings
- -03 — less orange warnings and longer computation time. Use this option for projects containing less than 1,000 lines of code.

Storage of Temporary Files

If you specify the option `-tmp-dir-in-results-dir`, Polyspace does not use the standard `/tmp` or `C:\Temp` folder to store temporary files. Instead, Polyspace uses a subfolder of the results folder. This action may affect processing speed if the results folder is mounted on a network drive. Use this option only when the temporary folder partition is not large enough and troubleshooting is required.

You can specify `-tmp-dir-in-results-dir` through a line command or the **Configuration > Advanced Settings > Other** field.

Reviewing Verification Results

- “Open Remote Verification Results” on page 10-4
- “Download Remote Verification Results From Command Line” on page 10-5
- “Open Unit-by-Unit Verification Results” on page 10-6
- “Open Local Verification Results” on page 10-7
- “Search Results in Results Manager” on page 10-8
- “Set Character Encoding Preferences” on page 10-12
- “Open Results for Generated Code” on page 10-15
- “Review Results Progressively” on page 10-16
- “Assign Review Status to Result” on page 10-18
- “Review Methodologies” on page 10-24
- “Organize Results Using Predefined Methodologies” on page 10-26
- “Organize Results Using Custom Methodologies” on page 10-30
- “Organize Results Using Filters and Groups” on page 10-34
- “View Call Sequence for Checks” on page 10-42
- “View Call Tree for Functions” on page 10-44
- “View Access Graph for Global Variables” on page 10-49
- “Customize Review Status” on page 10-50
- “Use Range Information in Results Manager” on page 10-55
- “View Pointer Information in Results Manager” on page 10-60

- “View Probable Cause for Checks” on page 10-61
- “Check Colors” on page 10-64
- “Source Code Colors” on page 10-65
- “Results Manager Overview” on page 10-66
- “Results Summary” on page 10-67
- “Source” on page 10-71
- “Check Details” on page 10-86
- “Check Review” on page 10-87
- “Call Hierarchy” on page 10-91
- “Variable Access” on page 10-94
- “Red Checks” on page 10-102
- “Gray Checks” on page 10-103
- “Orange Checks” on page 10-105
- “Color Sequence of Checks” on page 10-109
- “Defects from Code Integration” on page 10-113
- “Defects in Unprotected Shared Data” on page 10-114
- “Defects Related to Pointers” on page 10-115
- “Global Variables” on page 10-118
- “Dataflow Verification” on page 10-120
- “Results Folder” on page 10-121
- “Reusing Review Comments” on page 10-124
- “Import Review Comments from Previous Verifications” on page 10-125
- “View Checks and Comments Report” on page 10-127
- “Generate Report After Verification Automatically” on page 10-129
- “Generate Report After Verification Manually” on page 10-130
- “Generate Report from Command Line” on page 10-132
- “Open Verification Report” on page 10-134

-
- “Customize Verification Report” on page 10-135

Open Remote Verification Results

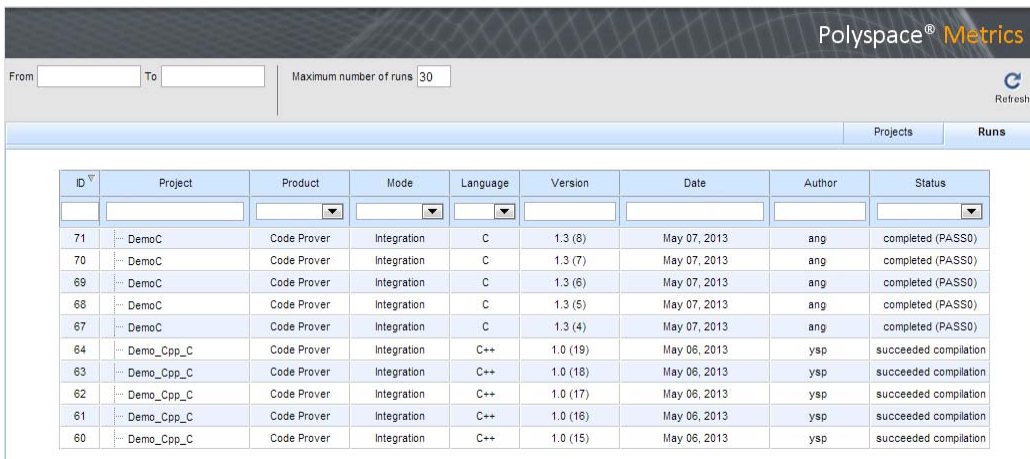
Use Polyspace Metrics to open results from a remote verification.

1 In the address bar of your Web browser, enter the following URL:

protocol://ServerName:PortNumber

- *protocol* is either http (default) or https.
- *ServerName* is the name or IP address of your Polyspace Metrics server.
- *PortNumber* is the Web server port number (default 8080).

For reference, save the Polyspace Metrics Web page as a bookmark.



The screenshot shows the Polyspace Metrics web interface. At the top, there is a header with the Polyspace Metrics logo. Below the header, there are input fields for 'From' and 'To' dates, and a 'Maximum number of runs' field set to 30. A 'Refresh' button is also visible. The main content area is a table with columns for ID, Project, Product, Mode, Language, Version, Date, Author, and Status. The table contains 10 rows of data, with the first 5 rows showing 'DemoC' projects and the last 5 rows showing 'Demo_Cpp_C' projects.

ID	Project	Product	Mode	Language	Version	Date	Author	Status
71	DemoC	Code Prover	Integration	C	1.3 (8)	May 07, 2013	ang	completed (PASS0)
70	DemoC	Code Prover	Integration	C	1.3 (7)	May 07, 2013	ang	completed (PASS0)
69	DemoC	Code Prover	Integration	C	1.3 (6)	May 07, 2013	ang	completed (PASS0)
68	DemoC	Code Prover	Integration	C	1.3 (5)	May 07, 2013	ang	completed (PASS0)
67	DemoC	Code Prover	Integration	C	1.3 (4)	May 07, 2013	ang	completed (PASS0)
64	Demo_Cpp_C	Code Prover	Integration	C++	1.0 (19)	May 06, 2013	ysp	succeeded compilation
63	Demo_Cpp_C	Code Prover	Integration	C++	1.0 (18)	May 06, 2013	ysp	succeeded compilation
62	Demo_Cpp_C	Code Prover	Integration	C++	1.0 (17)	May 06, 2013	ysp	succeeded compilation
61	Demo_Cpp_C	Code Prover	Integration	C++	1.0 (16)	May 06, 2013	ysp	succeeded compilation
60	Demo_Cpp_C	Code Prover	Integration	C++	1.0 (15)	May 06, 2013	ysp	succeeded compilation

2 Click the **Project** or **Version** cell of your verification.

The software downloads and opens the results in the Results Manager perspective of Polyspace Code Prover.

For more information, see:

- “Set Up Polyspace Metrics”
- “Results Manager Overview” on page 10-66

Download Remote Verification Results From Command Line

To download verification results from the command line, use the `polyspace-jobs-manager` command:

```
MATLAB_Install\polyspace\bin\polyspace-jobs-manager -download  
-job Verification_ID -results-folder FolderPath
```

For more information, see “Manage Remote Analyses at the Command Line” on page 8-18.

After downloading results, use the Results Manager to view the results. See “Open Local Verification Results” on page 10-7.

Open Unit-by-Unit Verification Results

If you run a unit-by-unit verification, the software submits each source file separately for verification. Polyspace Metrics displays these verifications using a tree structure.

ID	Project	Product	Mode	Language	Version	Date
5	Polyspace	Code Prover	Unit By Unit	C	1.0 (3)	May 21, 2013
5/1	example			C		
5/2	initialisations			C		
5/3	main			C		
5/4	single_file_analysis			C		
5/5	tasks1			C		
5/6	tasks2			C		

To download and open all results for the project:

- 1 In the parent row, click the **Project** or **Version** cell.
- 2 Select the **Download all results sets** check box. Then click **OK**.

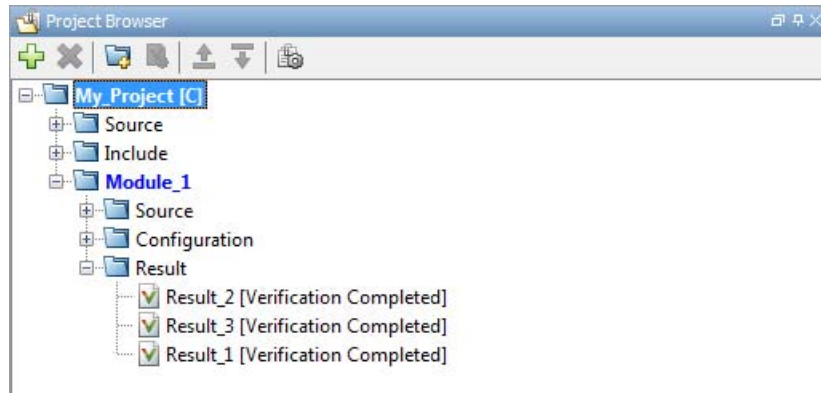
To download and open results for a specific, in the **Project** cell, expand the parent node. Then double-click the required file verification.

Alternatively:

- 1 In the parent row, click the **Project** or **Version** cell.
- 2 In the Select the results set to review dialog box, from the **Results Set** drop-down list, select the results that you want to review. Then click **OK**.

Open Local Verification Results

- 1 From the Project Manager perspective, in the Project Browser, navigate to the results that you want to review.



- 2 Double-click the results file, for example, `example_project.pscp`.

The software opens the verification results in the Results Manager perspective.

Alternatively:

- 1 On the Polyspace Code Prover toolbar, select **File > Open Result**.
- 2 In the Open Results dialog box, navigate to the results folder. For example:
`polyspace_project\Module_1\Result_example_project_1`
- 3 Select the results file, for example, `example_project.pscp`.
- 4 Click **Open**.

Search Results in Results Manager

This example shows how to search for all occurrences of a variable or function name in the Results Manager perspective. Search for the variable or function name in the following situations:

- A read/write operation on a variable causes a check. However, the check might be related to an instruction prior to this read/write operation.

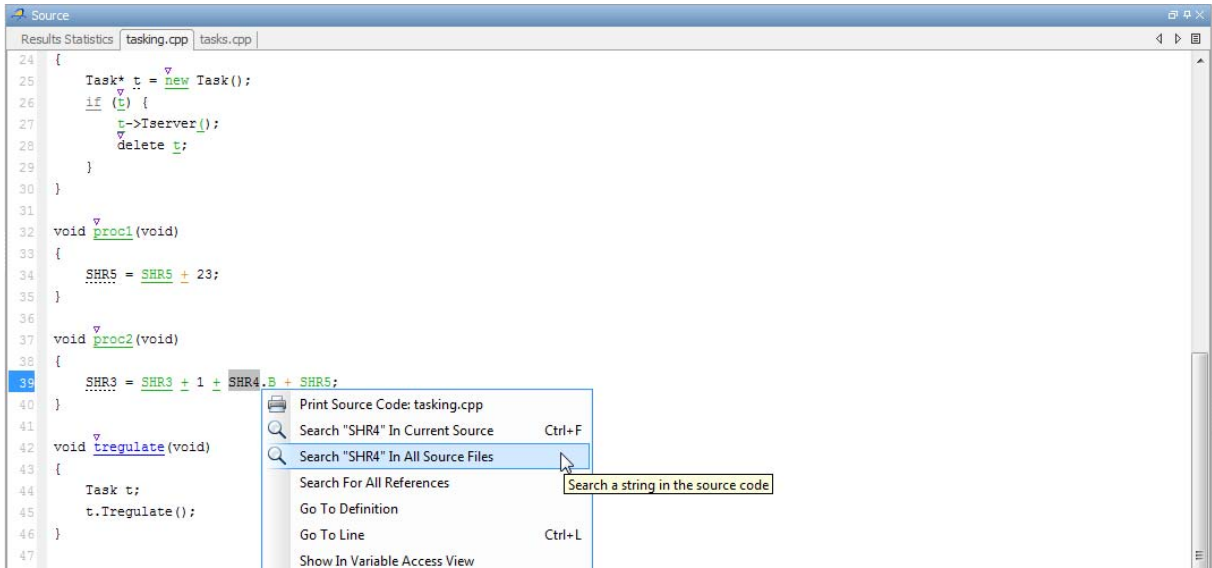
Selecting a check in the **Results Summary** pane displays the read/write operation only. On the **Source** pane, you can look in the source code for prior instructions containing the variable name. Instead, searching for the occurrences is an easier way to find and quickly navigate to them.

For instance, consider the check, **Out of bounds array index**. Though an access operation on the array causes the check, it is useful to quickly navigate to the array declaration.

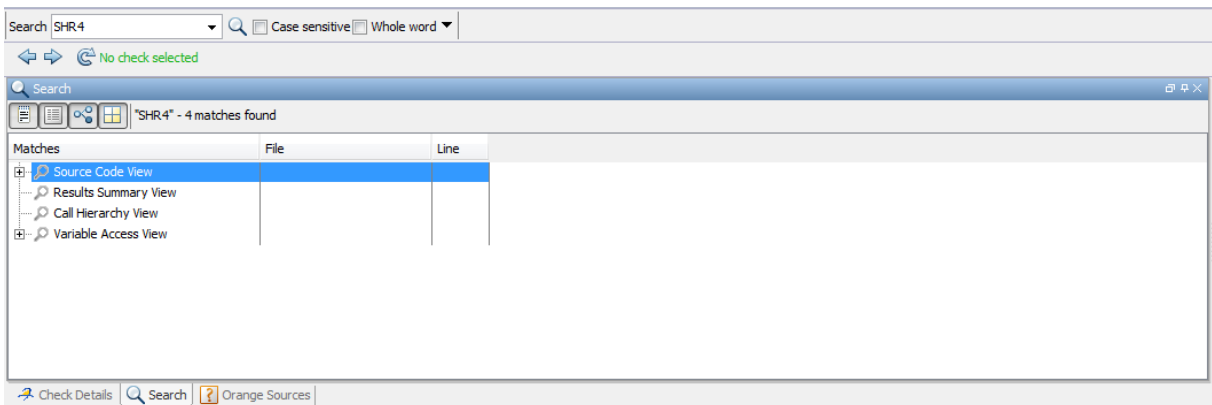
- A function call causes a check. However, the check might be related to an instruction in the function definition. Therefore, it is useful to quickly navigate to the function definition.

Search Variable Name

- 1 Enter the variable name in the Search box in the Results Manager perspective toolbar. Alternatively, on the **Source** pane, right-click the variable name and select **Search “*variable_name*” In All Source Files**.



The **Search** tab displays all occurrences of the variable name under four categories.



2 To see all occurrences of the variable name in the source code, expand the node **Source Code View**.

This node lists each occurrence of the variable name along with the file name and the line number. Use the file name and line number to identify all occurrences of the variable name before a check occurs.

- 3** To navigate to a particular occurrence of the variable name in the source code, use the up and down arrow keys.

The **Source** pane displays the corresponding line of code.

- 4** If the variable is a global variable, to navigate to its declaration quickly, expand the node **Variable Access View**. Select the only search result under this node.

Search Function Name

- 1** Enter the function name in the Search box in the Results Manager perspective toolbar. Alternatively, on the **Source** pane, right-click the function name and select **Search “*function_name*” In All Source Files**.

The **Search** tab displays all occurrences of the function name under four categories.

- 2** To see all occurrences of the function name in the source code, expand the node **Source Code View**.

This node lists each occurrence of the function name along with the file name and the line number.

- 3** To navigate to a particular occurrence of the function name in the source code, use the up and down arrow keys.

The **Source** pane displays the corresponding line of code.

- 4** To navigate to the function definition quickly:
 - a** On the **Results Summary** pane, select **Checks by File/Function** from the drop-down list.

The **Results Summary** pane displays file names in alphabetical order. Under each file name, the pane displays the function names in alphabetical order.

- b** Select the name of the function.

The **Source** pane displays the function definition.

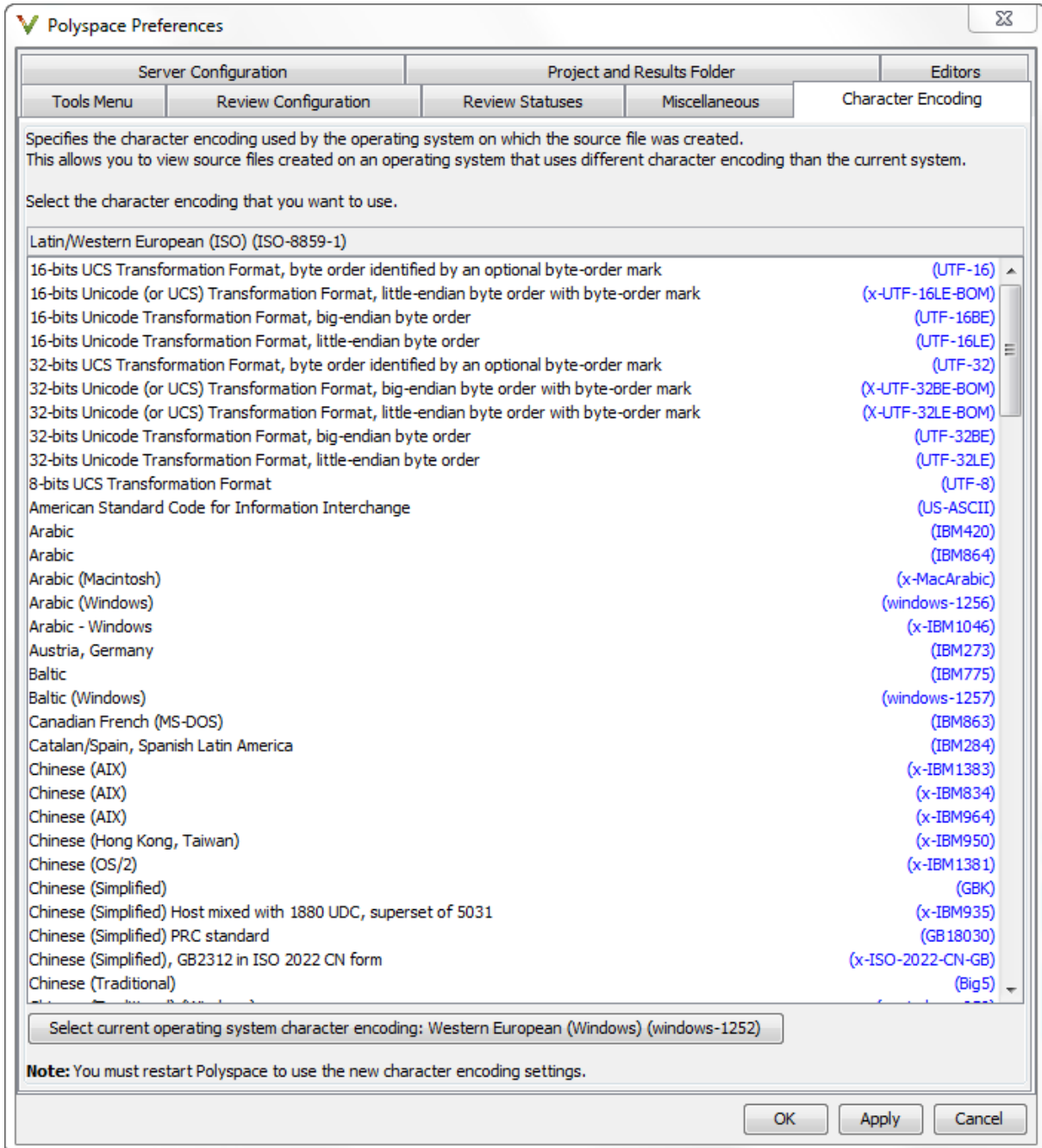
Set Character Encoding Preferences

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 **Options > Preferences**.
- 2** In the **Polyspace Preferences** dialog box, select the **Character encoding** tab.



- 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.

Open Results for Generated Code

When opening results for automatically generated code, the software must know which code generator created the code, so that it can interpret comments and create back-to-source links in the Results Manager perspective.

If you start the verification from Simulink, the software automatically creates a file in the results folder called `code_generator_used.txt` to provide this information. Otherwise, you must provide this information manually.

To manually specify the code generator that created the code:

- 1 Open your results in the Results Manager perspective.
- 2 Select **Review > Code Generator Support > *code_generator***

Manually Create the Code Generator Text File

To avoid specifying the code generator each time you open your results, you can manually create a file named `code_generator_used.txt` in your results folder. The software then automatically uses this file each time you open the results.

The format of this file is:

```
<Code generator>  
MATLABROOT=<Path to MATLAB>  
ModelVersion=<model name>:<model version>
```

<Code generator> can be either `RTWEmbeddedCoder` or `TargetLink`.

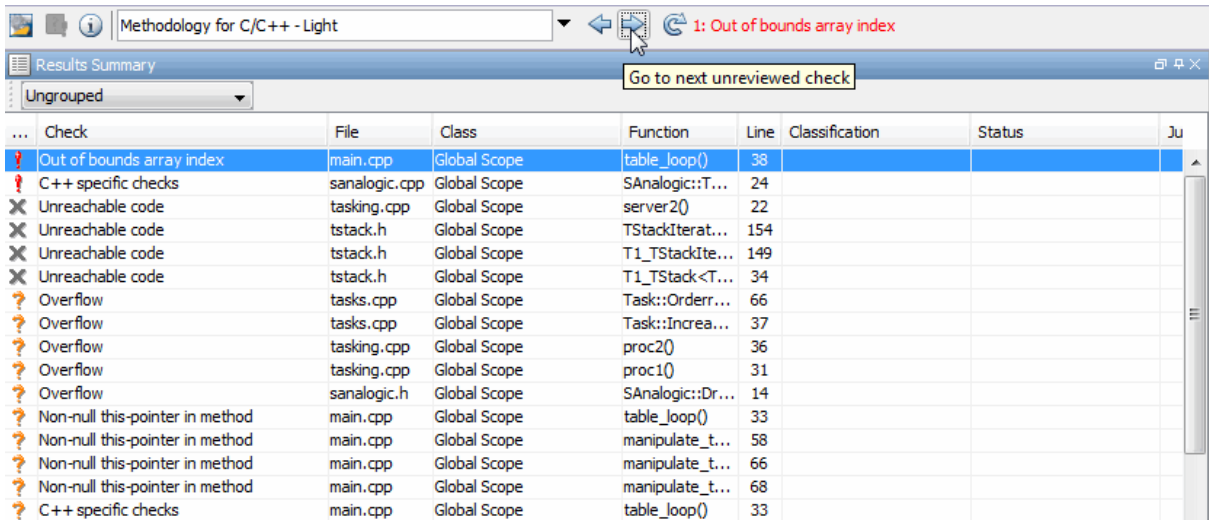
For example:

```
RTWEmbeddedCoder  
MATLABROOT=C:\MATLAB\R2010b  
ModelVersion=demo_m1:1.94
```

Review Results Progressively

This example shows how to review checks progressively using the Results Manager perspective.

1 Select the **Results Summary** view.



2 Click the forward arrow to go to the first check in the set:

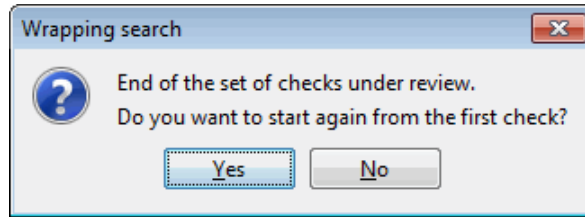
- The **Source** pane displays the source code for this check.
- The **Check Details** pane displays information about this check.

3 Review the current check.



After you review a check, you can classify the check and enter comments to describe the results of your review. You can also mark the check as Justified to help track your review progress.

4 Continue to click the forward arrow until you have gone through all of the checks.

After the last check, a dialog box opens asking if you want to start again from the first check.



5 Click **No**.

Note If you want to navigate through justified checks, use the justified check forward arrow  and back arrow .

Related Examples

- “Assign Review Status to Result” on page 10-18
- “View Call Sequence for Checks” on page 10-42

Assign Review Status to Result

This example shows how to review and comment checks using the Results Manager perspective. When reviewing checks, you can assign a status to checks, 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 check twice.

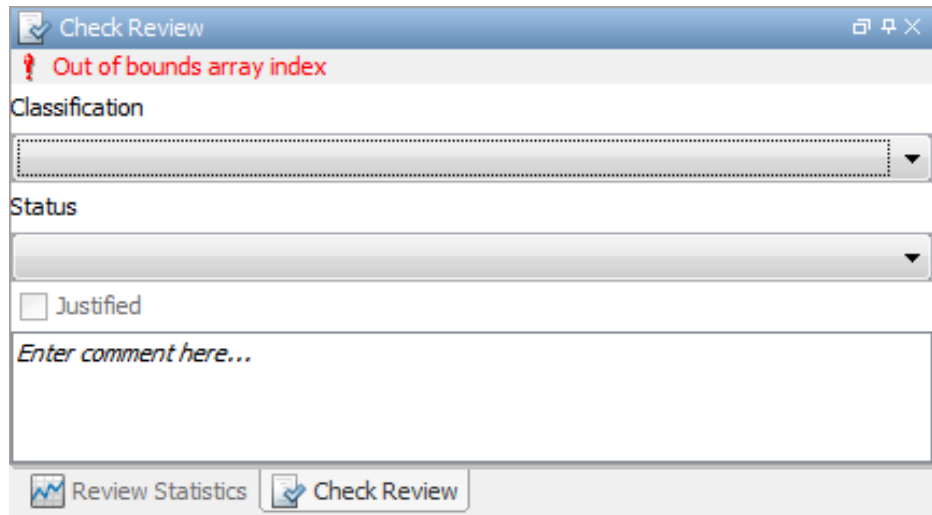
Review Individual Check

- 1 On the **Results Summary** pane, select the check that you want to review.

The **Check Details** pane displays information about the current check.



The **Check Review** tab displays fields where you can enter review information.



Check Review

Out of bounds array index

Classification

Status

Justified

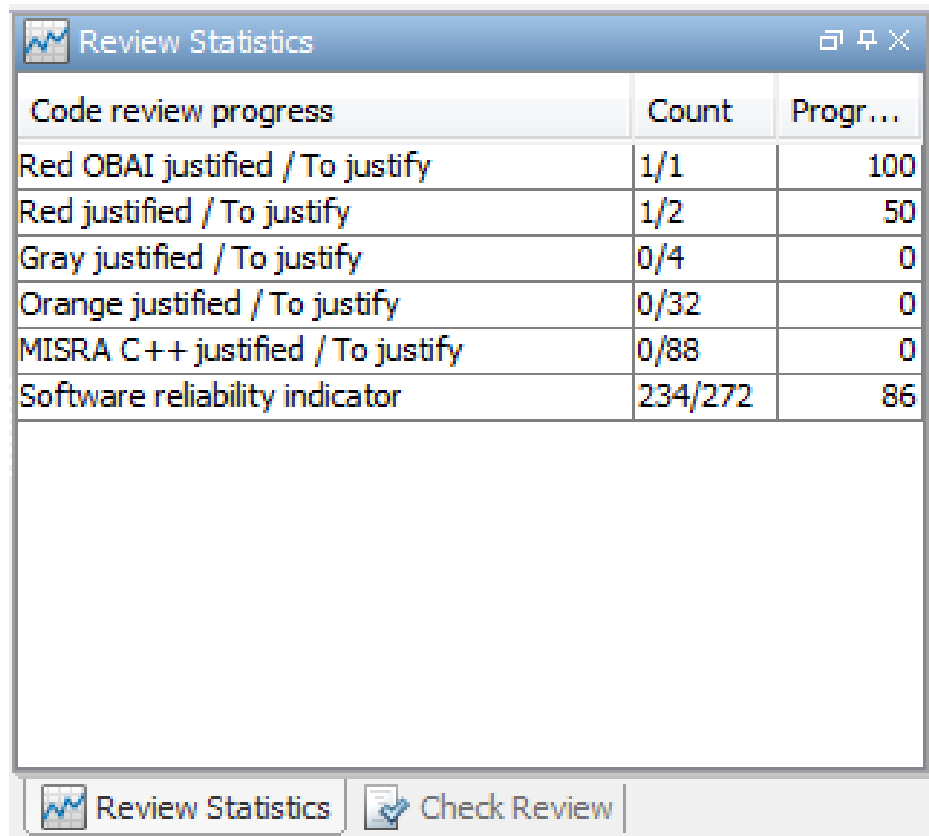
Enter comment here...

Review Statistics Check Review

- 2 Select a **Classification** to describe the severity of the issue:
 - Unset
 - High
 - Medium
 - Low
 - Not a defect
- 3 Select a **Status** to describe how you intend to address the issue:
 - Fix
 - Improve
 - Investigate
 - Justify with annotations
 - No action planned
 - Other
 - Restart with different options
 - Undecided

- 4 To justify the check, select one of the **Status** options, Justify with annotations or No action planned.

On the **Review Statistics** pane, the software updates the ratios of errors justified to total errors.



The screenshot shows a window titled "Review Statistics" with a table of error counts and progress. The table has three columns: "Code review progress", "Count", and "Progr...". The data rows are:

Code review progress	Count	Progr...
Red OBAI justified / To justify	1/1	100
Red justified / To justify	1/2	50
Gray justified / To justify	0/4	0
Orange justified / To justify	0/32	0
MISRA C++ justified / To justify	0/88	0
Software reliability indicator	234/272	86

At the bottom of the window, there are two buttons: "Review Statistics" and "Check Review".

- 5 In the **Comment** field, enter remarks, for example, defect or justification information.

Note You can also enter the review information through the **Classification**, **Status**, and **Comment** fields on the **Results Summary** pane.

Review Group of Checks

1 On the **Results Summary** pane, select a group of checks using one of the following methods:

- For contiguous checks, left-click the first check. Then **Shift**-left click the last check.

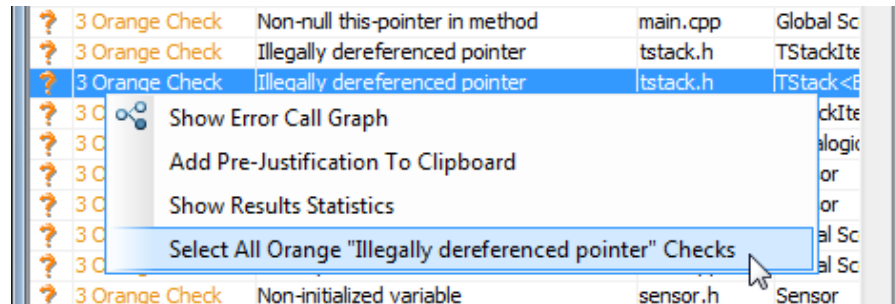
?	3 Orange Check	Non-null this-pointer in method	main.cpp	Global Sc
?	3 Orange Check	Non-null this-pointer in method	main.cpp	Global Sc
?	3 Orange Check	Illegally dereferenced pointer	tstack.h	TStackIt
?	3 Orange Check	Illegally dereferenced pointer	tstack.h	TStack<B
?	3 Orange Check	Illegally dereferenced pointer	tstack.h	TStackIt
?	3 Orange Check	Illegally dereferenced pointer	sanalogic.h	SAnalogic
?	3 Orange Check	Illegally dereferenced pointer	sensor.h	Sensor
?	3 Orange Check	Illegally dereferenced pointer	sensor.h	Sensor
?	3 Orange Check	C++ specific checks	main.cpp	Global Sc
?	3 Orange Check	C++ specific checks	main.cpp	Global Sc
?	3 Orange Check	Non-initialized variable	sensor.h	Sensor

To group together checks belonging to a certain category, click the **Check** column header on the **Results Summary** pane.

- For non-contiguous checks, **Ctrl**-left click each check.

?	3 Orange Check	Non-null this-pointer in method	main.cpp	Global Sc
?	3 Orange Check	Non-null this-pointer in method	main.cpp	Global Sc
?	3 Orange Check	Illegally dereferenced pointer	tstack.h	TStackIt
?	3 Orange Check	Illegally dereferenced pointer	tstack.h	TStack<B
?	3 Orange Check	Illegally dereferenced pointer	tstack.h	TStackIt
?	3 Orange Check	Illegally dereferenced pointer	sanalogic.h	SAnalogic
?	3 Orange Check	Illegally dereferenced pointer	sensor.h	Sensor
?	3 Orange Check	Illegally dereferenced pointer	sensor.h	Sensor
?	3 Orange Check	C++ specific checks	main.cpp	Global Sc
?	3 Orange Check	C++ specific checks	main.cpp	Global Sc
?	3 Orange Check	Non-initialized variable	sensor.h	Sensor

- For checks of a similar color and category, right-click one check. From the context menu, select **Select All *Color Type* Checks**, for instance, **Select All Orange "Illegally dereferenced pointer" Checks**.



- 2 On the **Check Review** tab, enter the required information. The software applies this information to the selected checks.

Save Review Comments

After you have reviewed your results, save your comments with the verification results. Saving your comments makes them available the next time that you open the results file, allowing you to avoid reviewing the same check twice.

To save your review comments, select **File > Save**. Your comments are saved with the verification results.

Track Review Progress

- 1 To see how many checks of a certain type you have reviewed, select a check of that type on the **Results Summary** pane.

The **Count** column on the **Review Statistics** pane displays the number of unreviewed checks as a ratio of total number of checks. The first row on the **Review Statistics** pane displays the ratio for checks of the same color and type as the selected check.

- 2 To see how many checks of a certain color you have reviewed, select a check on the **Results Summary** pane.

The **Count** column on the **Review Statistics** pane displays the ratio of unreviewed checks to total number of checks for all check colors.

The **Progress (%)** column displays the same ratio as a percentage.

For more information, see “Review Statistics” on page 10-87.

**Related
Examples**

- “Organize Results Using Filters and Groups” on page 10-34
- “Customize Review Status” on page 10-50

Review Methodologies

To facilitate your review of verification results, with Polyspace Code Prover, you can specify the number and type of checks displayed in the **Results Summary** pane of the Results Manager perspective. These specifications are known as review methodologies.

By choosing a methodology, you can review a subset of checks. Polyspace Code Prover provides four predefined review methodologies.

Select the predefined review methodologies in the following order to incrementally view the checks displayed. For more information, see “Organize Results Using Predefined Methodologies” on page 10-26. For information on the check colors, see “Check Colors” on page 10-64.

- 1 First checks to review** — The software displays all red and gray checks. It also displays a few orange checks that are most likely to be run-time errors. For more information, see “Orange Check Identified as Potential Errors” on page 10-105. When reviewing code for the first time, choose this methodology.
- 2 Methodology for C/C++ > Light** — The software displays all red, gray, and purple checks. The software also displays a subset of orange checks based on specifications in **Polyspace Preferences > Review Configuration**.
- 3 Methodology for C/C++ > Moderate** — The software displays all red, gray, and purple checks. The software also displays a larger subset of orange checks based on specifications in **Polyspace Preferences > Review Configuration**.
- 4 All** — The software displays all red, gray, purple, green, and orange checks. When you want to carry out an exhaustive review of your verification results, use this methodology.

You can also define review methodologies customized to your priorities. For more information, see “Organize Results Using Custom Methodologies” on page 10-30.

In the Results Manager perspective, the following toolbar provides controls related to review methodologies.



The controls include:

- A menu for selecting the review methodology.
- Arrows for navigating through checks.

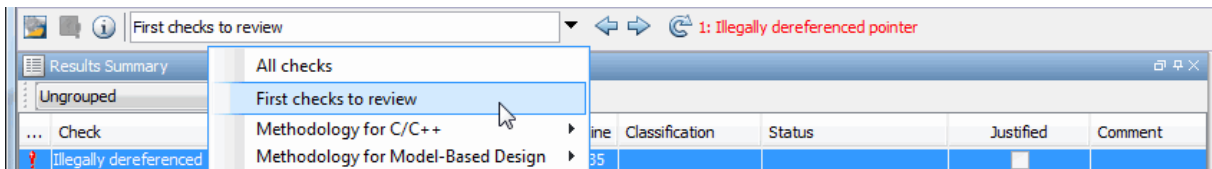
Organize Results Using Predefined Methodologies

This example shows how to incrementally view the checks using predefined review methodologies provided by Polyspace Code Prover.

Review methodologies specify the number and type of checks displayed in the **Results Summary** pane.

Review Checks Incrementally

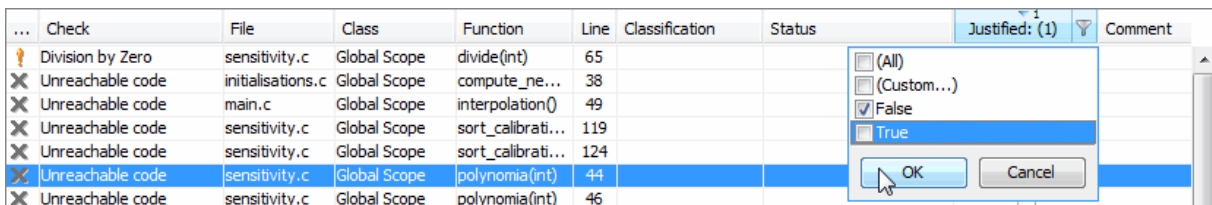
- 1 In the Results manager perspective, from the drop-down list above the **Results Summary** pane, select the methodology, **First checks to review**.



The **Results Summary** pane displays all red and gray checks, as well as orange checks most likely to be run-time errors. Investigate and fix the errors. Assign a **Status** and **Classification** to the checks. To mark a check as justified, select the status **Justify with annotation** or **No action planned**. You can also create custom statuses or add justification to existing statuses.

- 2 After justifying checks, select the methodology, **Methodology for C/C++ > Light**.

In the **Results Summary** pane, you can view all red, gray, and purple checks, as well as a subset of orange checks. To filter the unjustified checks, from the drop-down list beside the **Justified** column header, clear all boxes except **False** and select **OK**.



Investigate and fix the errors. Assign a **Status** and **Classification** to the checks.

- 3** After justifying checks, to see a larger subset of orange checks, select **Methodology for C/C++ > Moderate**. The number of orange checks in the **Results Summary** pane increases.

To refresh the list to show unjustified checks only, reopen the drop-down list beside the **Justified** column header and select **OK**.

Investigate and fix the errors. Assign a **Status** and **Classification** to the remaining checks.

- 4** To exhaustively review all of the checks, select **All checks**. In addition to all orange checks, this methodology also reveals all the green checks in the **Results Summary** pane.

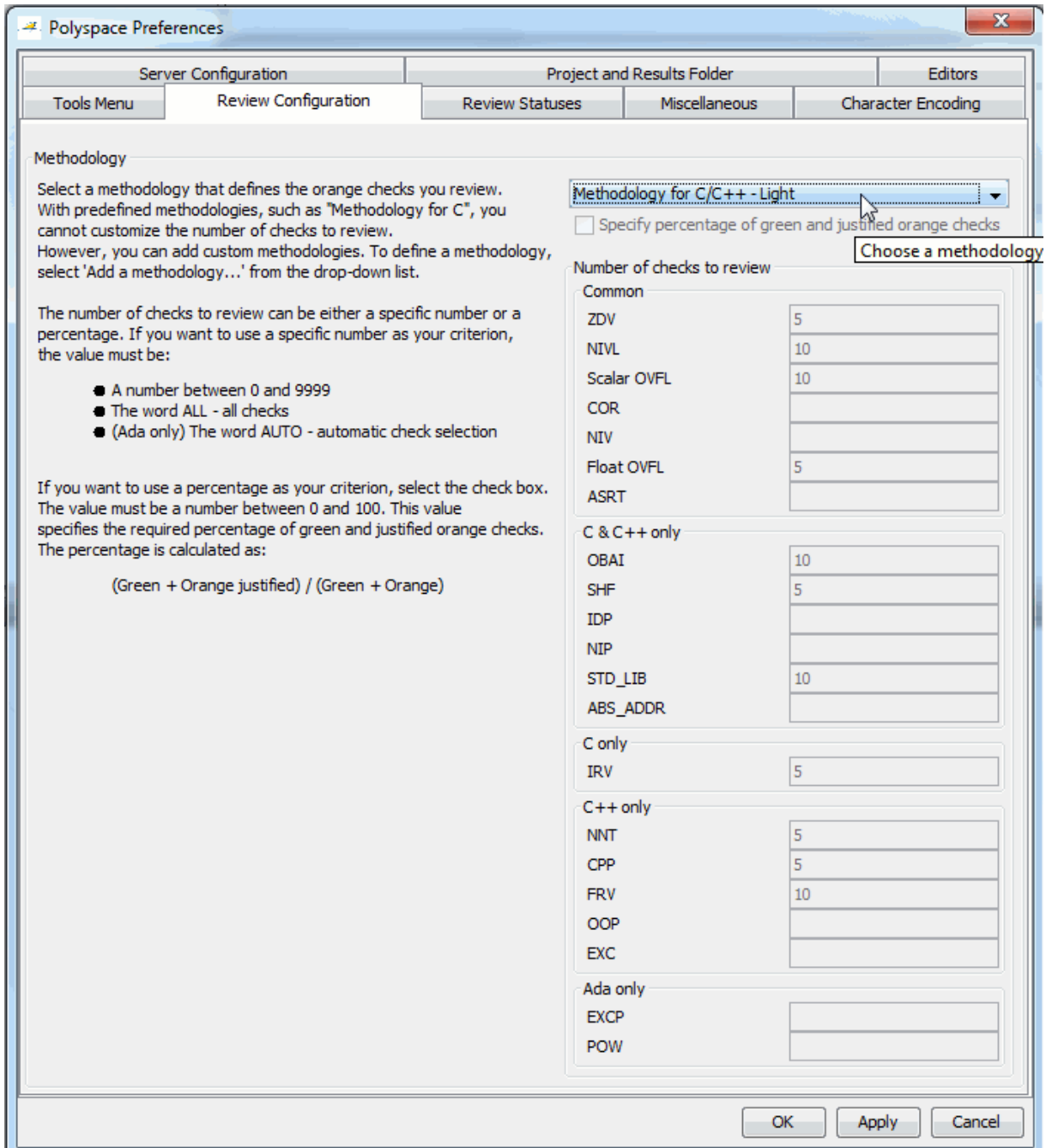
View Light and Moderate Methodology Requirements

You can view the requirements of the methodologies, **Methodology for C/C++ > Light** and **Methodology for C/C++ > Moderate** through the Polyspace Preferences dialog box.

Note You cannot change the parameters specified in any of the predefined methodologies. However, you can create your own custom methodologies.

- 1** In the Polyspace verification environment, select **Options > Preferences**.
- 2** In the Polyspace Preferences dialog box, select the **Review configuration** tab.
- 3** From the drop-down list on this tab, select **Methodology for C/C++-Light**.

The table shows the number of orange checks of each type you review when you select this methodology in the Results Manager perspective.



For example, the table specifies that you review five orange ZDV checks when you select the methodology, **Methodology for C/C++-Light**. The number of checks of each type increases as you move from **Methodology for C/C++-Light** to **Methodology for C/C++-Moderate**.

Set Default Methodology

To set a default methodology that will be applied every time you open your results:

- 1** Select **Options > Preferences**.
- 2** On the **Miscellaneous** tab, from the **Methodology mode** dropdown list, select the default methodology you want.

Related Examples

- “Assign Review Status to Result” on page 10-18
- “Organize Results Using Custom Methodologies” on page 10-30
- “Customize Review Status” on page 10-50

Concepts

- “Review Methodologies” on page 10-24

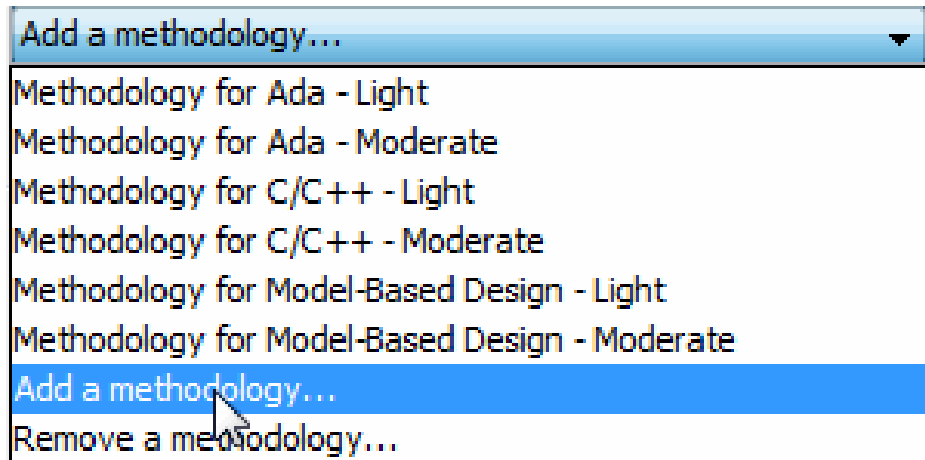
Organize Results Using Custom Methodologies

This example shows how to define and use a custom review methodology to specify the number and type of checks displayed in the **Results Summary** pane. Define a custom methodology to:

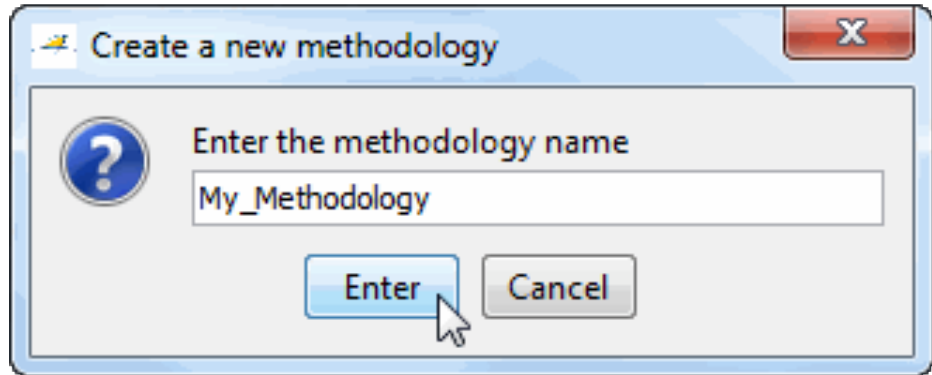
- Prioritize the checks that you review.
- Set standards that your team of developers must meet.

Define a Custom Methodology

- 1 In the Polyspace verification environment, select **Options > Preferences**.
- 2 In the Polyspace Preferences dialog box, select the **Review configuration** tab.
- 3 From the drop-down list on this tab, select **Add a methodology...**



- 4 Enter a name for your methodology in the Create a new methodology dialog box. For this example, enter the name, `My_Methodology`. Then, click **Enter**.



- 5 If you want to review orange checks by percentage, select the **Specify percentage of green and justified orange checks** check box.

With custom methodologies, you can specify either a specific number of orange checks to review or a minimum percentage of orange checks that must be reviewed. The percentage is calculated by:

$$(\text{green checks} + \text{justified orange checks}) \times 100 / (\text{green checks} + \text{total orange checks})$$

- 6 Enter the total number of checks (or percentage of checks) to review for each type of check for your methodology.

My Methodology

Specify percentage of green and justified orange checks

Number of checks to review

Common

ZDV	5
NIVL	5
Scalar OVFL	5
COR	5
NIV	5
Float OVFL	5
ASRT	5

C & C++ only

OBAI	
SHF	
IDP	
NIP	
STD_LIB	
ABS_ADDR	

C only

IRV	
-----	--

C++ only

NNT	
CPP	
FRV	
OOP	
EXC	

Ada only

EXCP	
POW	

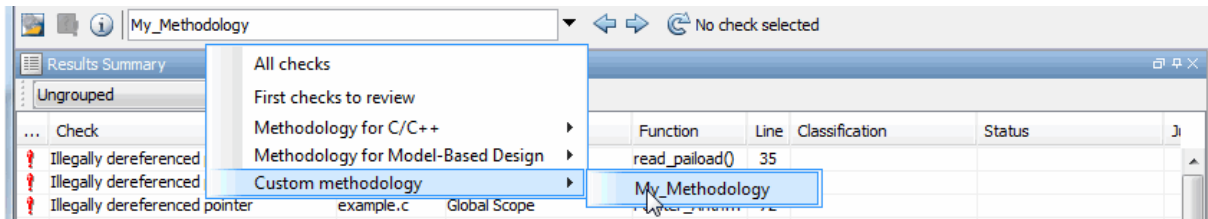
OK Apply Cancel

In this example, 5 was entered for ZDV indicating that 5 Division by Zero orange checks must be displayed when you choose **Custom methodology > My_Methodology** from the Results Manager perspective.

7 Click **OK** to save the methodology and close the dialog box.

Use Your Custom Methodology

In the Results manager perspective, from the drop-down list above the **Results Summary** pane, select the methodology, **Custom methodology > My_Methodology**.



The **Results Summary** pane displays orange checks according to the definition specified for **My_Methodology**. For instance, it displays 5 Division by Zero orange checks.

Related Examples

- “Organize Results Using Predefined Methodologies” on page 10-26

Concepts

- “Review Methodologies” on page 10-24

Organize Results Using Filters and Groups

This example shows how to filter and group checks on the **Results Summary** pane. To organize your review of checks, use filters and groups when you want to:

- Review certain categories of checks in preference to others. For instance, you first want to address checks resulting from `Out of bounds array index`.
- Not address the full set of coding rule violations detected by the coding rules checker.
- Not review checks you have already justified.

Typically, in your second or later rounds of review, you would have some checks already justified.

- Review only those checks that you have already assigned a certain status. For instance, you want to review only those checks to which you have assigned the status, `Investigate`.
- Review all checks in the body of a particular file or function. Because of continuity of code, reviewing these checks together can help you organize your review process.

You can also review checks 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 checks in automatically generated functions.
- C++ only: Review all checks dealing with a class definition.

Review Checks in a Given Category

To review checks resulting from `Out of bounds array index`:

- 1 Open the results file, with extension, `.pscp`.
- 2 On the **Results Summary** pane, from the drop-down list, select `Checks by Family`.

The checks are grouped by type of check.

Family	File	Class	Function	Line	%	Classification	Status	Justified	Comment
-1 Red Check					100				
⊕ C++		1			100				
⊕ Static memory		1			100				
-2 Gray Check		4			100				
⊕ Data flow		4			100				
-3 Orange Check		32			0				
⊕ C++		14			0				
⊕ Data flow		7			0				
⊕ Numerical		5			0				
⊕ Static memory		6			0				
-4 MISRA CPP Warning			88						
⊕ 0 Language independent issues			3						
⊕ 18 Language support library			8						
⊕ 2 Lexical conventions			2						
⊕ 3 Basic concepts			23						
⊕ 5 Expressions			16						
⊕ 6 Statements			12						
⊕ 7 Declarations			23						
⊕ 8 Declarators			1						
-5 Green Check		234			100				
⊕ C++		31			100				
⊕ Control flow		98			100				
⊕ Data flow		61			100				
⊕ Numerical		11			100				
⊕ Other		3			100				
⊕ Static memory		30			100				

3 Under the category **1 Red Check**, expand the subcategory **Static memory**.

You see the subcategory **Out of bounds array index**.

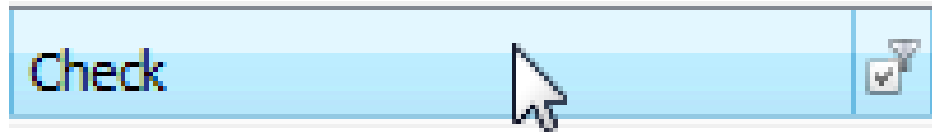
Family	File	Class	Function	Line	%	Classification	Status	Justified	Comment
-1 Red Check		2			100				
⊕ C++		1			100				
⊕ Static memory		1			100				
⊕ Out of bounds array index		1			100				
⊕ main.cpp		Global Scope	table_loop()	41				<input type="checkbox"/>	

Expand **Out of bounds array index** to view red checks of this kind.

To see further information about a check, select it. The information appears on the **Check Details** pane.

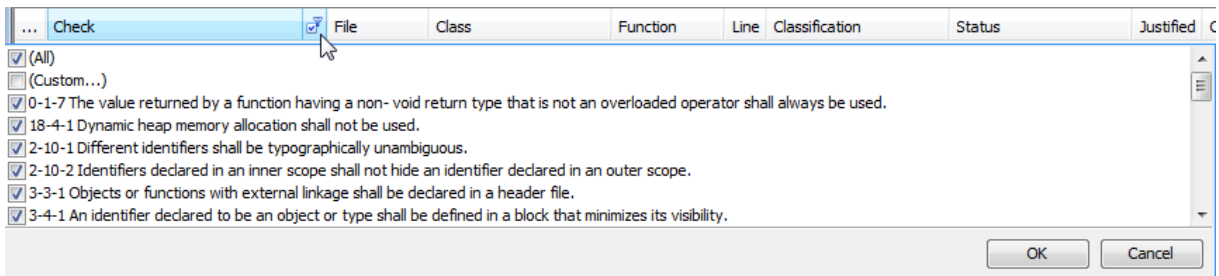
4 To view all orange checks resulting from this error, repeat step 3 for the subcategory **Static memory** under the category **3 Orange Check**.

- 5 To view only the checks resulting from the error, **Out of bounds array index**, on the **Results Summary** pane, from the drop-down list, select **List of Checks**.
- 6 Place your cursor on the **Check** column head.



- 7 Click the filter icon.

A context menu lists the filter options available.



- 8 Clear the **All** check box.
- 9 Scroll down to the **Out of bounds array index** check box and select it. Click **OK**.

The **Results Summary** pane displays only the checks resulting from the **Out of bounds array index** error.

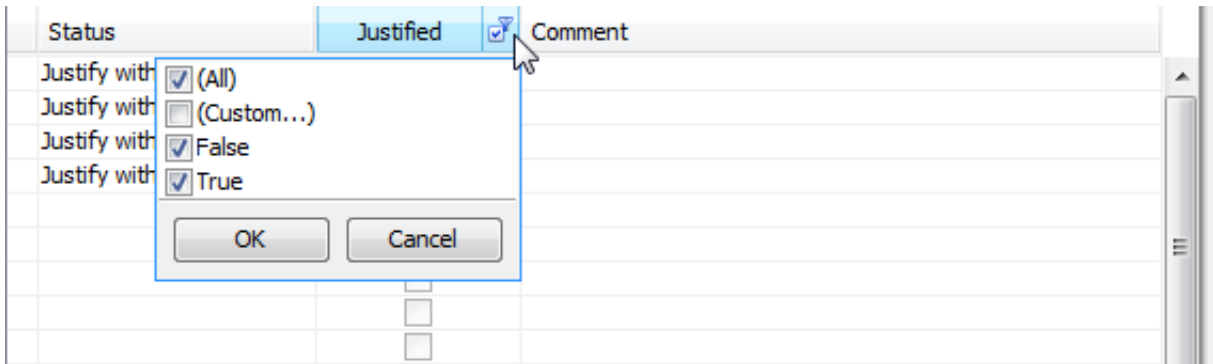
Review Checks Not Justified

To review only the checks that you have not justified:

- 1 Open the results file, with extension, **.pscp**.
- 2 On the **Results Summary** pane, place your cursor on the **Justified** column head.

- 3 Click the filter icon.

A context menu lists the filter options available.



- 4 Clear the **True** check box. Click **OK**.

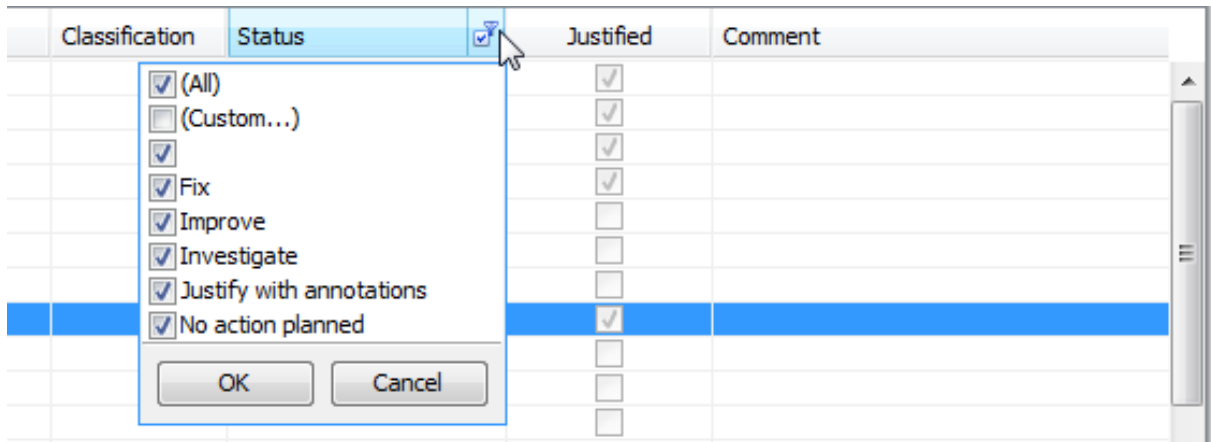
The **Results Summary** pane displays only the checks that you have not justified.

Review Checks with Given Status

To review only the checks with Investigate status:

- 1 Open the results file, with extension, .pscp.
- 2 On the **Results Summary** pane, place your cursor on the **Status** column head.
- 3 Click the filter icon.

A context menu lists the filter options available.



4 Clear the **All** check box.

5 Select the **Investigate** check box. Click **OK**.

The **Results Summary** pane displays only the checks with the Investigate status.

Review All Checks in a File

To review the checks in the file, `tasks.cpp`:

1 On the **Results Summary** pane, from the drop-down list, select **Checks by File/Function**.

The checks displayed are grouped by files. The file names are sorted alphabetically. Within each file name, the checks are grouped by functions, sorted alphabetically. Each file or function is colored by the most severe check that occurs. The severity decreases in this order:

- Red
- Gray
- Orange
- Purple
- Green

The screenshot shows the 'Results Summary' window with a tree view of checks. The tree is organized by file/function, with a dropdown menu open over the 'Checks by File/Function' header. A callout box points to the dropdown menu with the text 'Select option to organize checks'.

The tree view shows the following structure:

- Checks by File/Function
 - Family
 - main.cpp
 - Global Scope
 - main()
 - manipulate_template()
 - table_loop()
 - sanalogic.cpp
 - Global Scope
 - SAnalogic::TypeInfo()
 - sanalogic.h
 - SAnalogic::Draw()
 - SAnalogic::SAnalogic()
 - SAnalogic::~SAnalogic()
 - sensor.h
 - Sensor::Draw()
 - Sensor::getID()
 - Sensor::Sensor()
 - tasking.cpp
 - Begin_CS()
 - End_CS()
 - Global Scope
 - proc1()
 - proc2()
 - server1()
 - server2()
 - tasks.cpp
 - ._dynamic_init_globals()
 - Global Scope
 - Task::Command_Ordering(int)
 - Task::Computing_from_Sensors(int;int)
 - Task::Drive_Balance(int)
 - Task::Exec_One_Cycle(int)
 - Task::Increase_PowerLevel()
 - Task::Orderregulate()
 - Task::Scheduler(int)
 - Task::Task()
 - Task::Tserver()
 - tasks.h
 - Task::~~Task()
 - tstack.h
 - bool_TStack<T1>::empty()const_[with_T1=Base*]
 - int_TStackIterator<T1>::operator_++()[with_T1=Base*]
 - int_TStackIterator<T1>::operator_++(int)[with_T1=Base*]

- To view the checks in `tasks.cpp`, expand any function name under the category, **tasks.cpp**.

tasks.cpp			2	36	23	
dynamic_init_globals()				1		
Global Scope					20	
Task::Command_Ordering(int)				3		
Task::Computing_from_Sensors(int;int)					1	
Task::Drive_Balance(int)				3		
Task::Exec_One_Cycle(int)				5		
Task::Increase_PowerLevel()			1	2		
?	Overflow	Task				39
✓	Exception handling	Task				37
✓	Non-initialized variable	Task				39
Task::Orderregulate()			1	7	2	

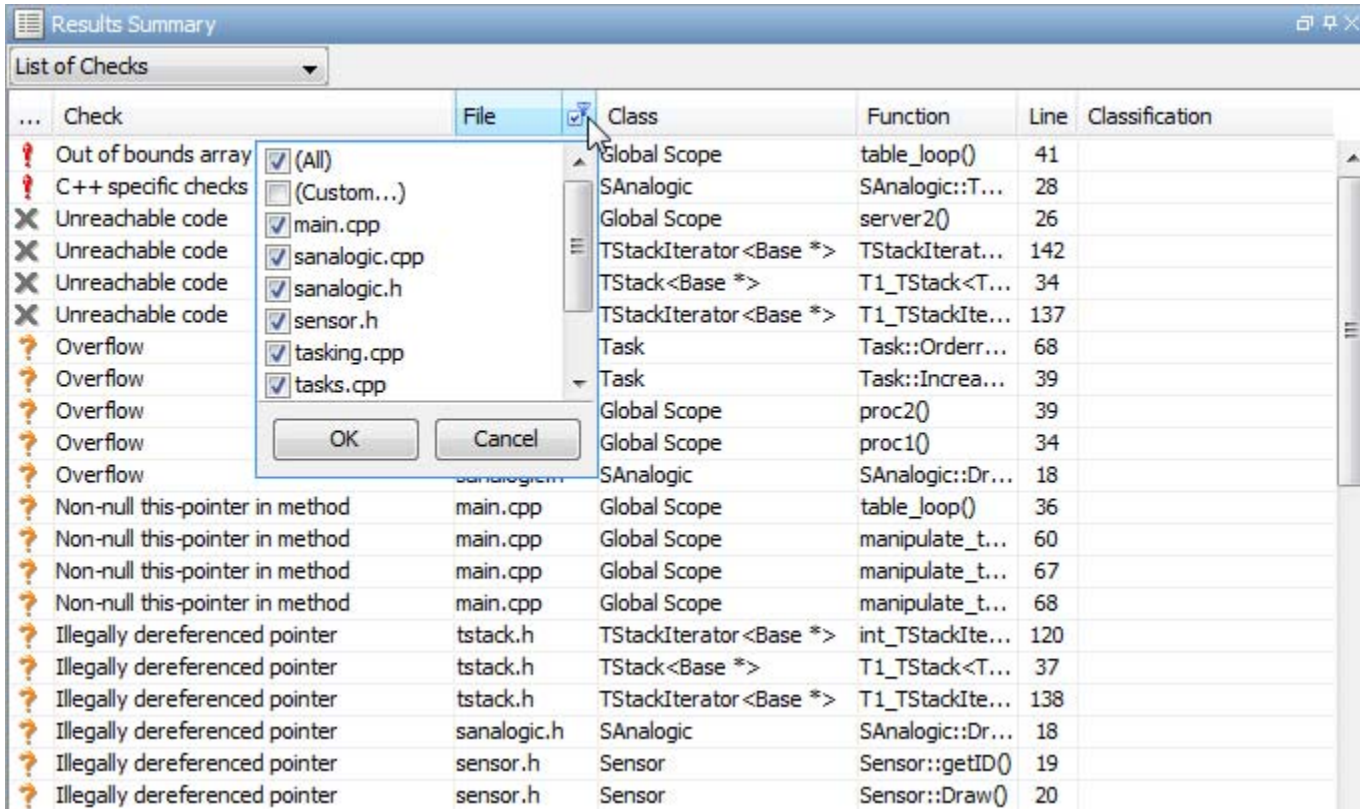
To view further information on a check, select the check. The information on the check appears on the **Check Details** pane.

- To view only the checks in `tasks.cpp`, on the Results Summary pane, from the drop-down list, select **List of Checks**.

The **Results Summary** pane displays all checks without any grouping.

- Place your cursor on the **File** column head.
- Click the filter icon.

A context menu lists the filter options available.



6 Clear the **All** check box.

7 Select the **tasks.cpp** check box. Click **OK**.

The **Results Summary** pane displays only the checks in `tasks.cpp`.

Tip If you apply a filter on a column on the **Results Summary** pane, the column header displays the number of check boxes selected in the filter menu. Use this information to keep track of filters that you have applied.

Related Examples

- “Apply Coding Rule Violation Filters” on page 13-17

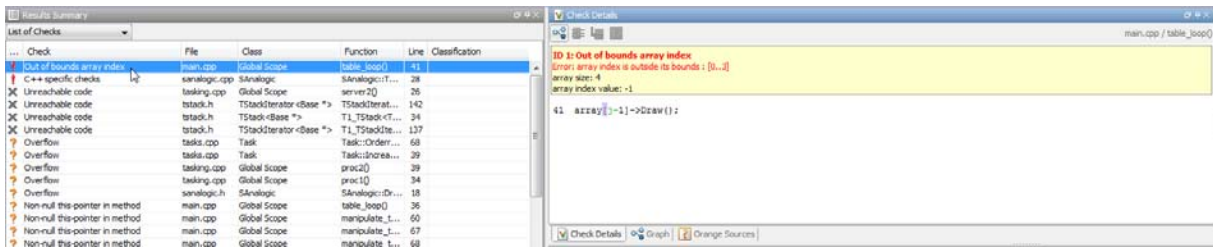
View Call Sequence for Checks

This example shows how to display the call sequence that leads to the code line associated with a check.

- 1 On the **Results Summary** pane, select the check that you want to review.

On the **Check Details** pane, the **Check Details** tab displays further information about the check.

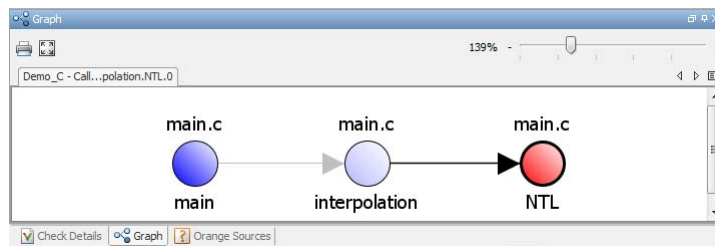
The **Source** pane displays the code line associated with the check.



- 2 On the **Check Details** tab toolbar, click the Show error call graph button,



On the **Check Details** pane, the **Graph** tab displays the call graph.



The call graph displays the call sequence leading to the code associated with the check. Each node on the graph, except for the terminal node, represents a function. The function name is below the node. The name of the file containing the function is above the node.

- 3** Select a node to navigate to the function definition in the source code.

The **Source** pane displays the function definition.

- 4** Select the terminal node to navigate back to the code line associated with the check.

View Call Tree for Functions

In this section...
“View Callers and Callees of a Function” on page 10-45
“Navigate Call Tree” on page 10-47

The call tree (or call graph) shows the calling relationship between functions (and tasks) in a program. From the call tree, for each function or task, `foo`, you can see its:

- Callers: functions and tasks calling `foo`.
- Callees: functions and tasks called by `foo`.

Sometimes, an error in a function might be related to an instruction in its callers or callees. Therefore, to review errors quickly, it is useful to:

- View all callers and callees of a function without navigating in the source code. The callers and callees are listed even for indirect calls through function pointers.
- Navigate quickly between a function, and its callers and callees.
- Verify dataflow for certification purposes. For more information, see “Dataflow Verification” on page 10-120.

You can perform these tasks from the **Call Hierarchy Pane** in the Results Manager perspective.

For a complete description of the **Call Hierarchy** pane, see “Call Hierarchy” on page 10-91.

Note If you do not see the **Call Hierarchy** pane in the Results Manager perspective, select **Window > Show/Hide View > Call Hierarchy**.

View Callers and Callees of a Function

You can view all callers and callees of a function from the **Call Hierarchy** pane.

- 1 View the function in the **Call Hierarchy** pane.

Begin from function name.

To view a function starting from its name, group the checks in the **Results Summary** pane by function names first. Select **Checks by File/Function** from the dropdown menu at the top of this pane.

Then, list the checks under a function by double-clicking the function name. Select a check. The function appears in the **Call Hierarchy** pane. This pane also lists the names of callers and callees.

Begin from check in function.

To view a function starting from a check inside the function, select the check in the **Results Summary** pane (under the Results Manager perspective).

The function containing the check appears in the **Call Hierarchy** pane. This pane also lists the names of callers and callees.

Begin from global variable access in function

To view the callers and callees starting from a global variable access inside the function, first list the functions performing read/write access on the variable. Double-click the variable name in the **Variable Access** pane.

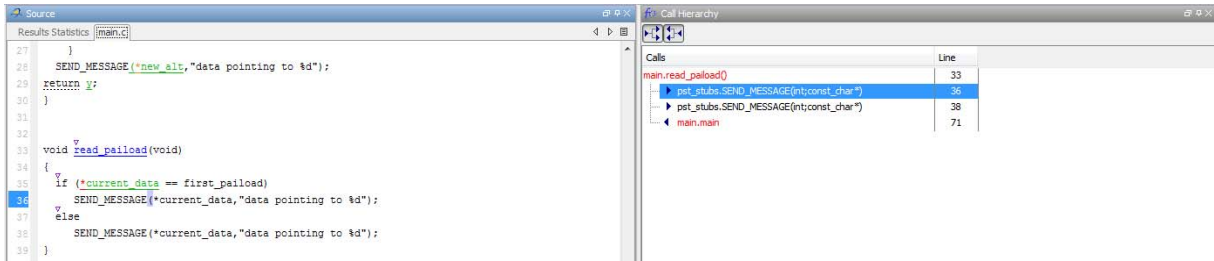
Select the name of a function. The function appears in the **Call Hierarchy** pane. This pane also lists the names of callers and callees.

- 2 Select the function name.

In the **Source** pane, the current line shows the beginning of the function definition.

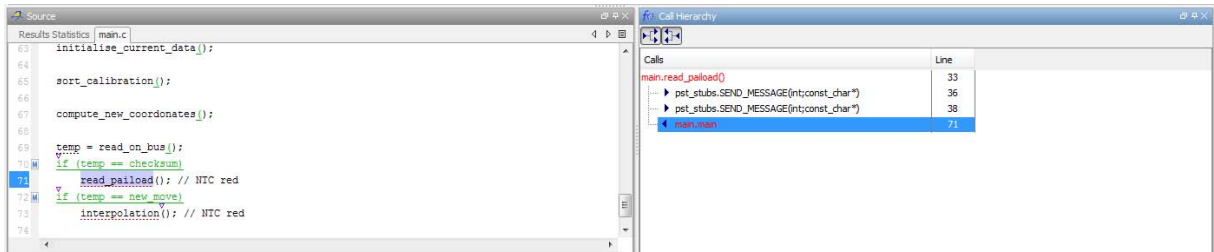
- 3 Select a callee name. These are listed below the function name and marked by ▶ (functions) or ||▶ (tasks).

In the **Source** pane, the current line shows where the callee is called.



- 4 Select a caller name. These are listed below the function name and marked by ◀ (functions) or ▶ (tasks).


In the **Source** pane, the current line shows where the caller calls the function.



- 5 View all branches of a callee by progressively clicking ⊕ next to the callee name.


This figure displays the callee, `Exec_One_Cycle`, defined in the file, `tasks2.c`, with all branches shown.

▣	▶ tasks2.Exec_One_Cycle(int)	89
▣	▶ tasks2.Pilot_Balance(int)	68
▣	▶ tasks2.Command_Ordering(int)	56
▣	▶ tasks1.orderregulate()	50
	▶ tasks2.Increase_PowerLevel()	44
▣	▶ tasks2.Sequencer(int)	69
▣	▶ tasks2.Command_Ordering(int)	62
▣	▶ tasks1.orderregulate()	50
	▶ tasks2.Increase_PowerLevel()	44

- 6 View all branches ending with the caller by progressively clicking  next to the caller name.



This figure displays the caller, Tserver, defined in the file, tasks1.c with all branches shown.

▣	◀ tasks1.Tserver()	86
▣	◀ tasks1.server2	97
	◀ main.main	56
▣	◀ tasks1.server1	103
	◀ main.main	56
	◀ tasks1.server2	0
	◀ tasks1.server1	0



Tip Instead of progressively viewing the branches by clicking , you can expand all caller/callee names at once. Right-click anywhere in the **Call Hierarchy** pane. From the context menu, select **Expand All Nodes**. You can collapse all caller/callee names by right-clicking anywhere in the **Call Hierarchy** pane and selecting **Collapse All Nodes**.

Navigate Call Tree

To navigate between a function and its callers and callees in the source code:

- 1 Select a check contained in the function from the **Results Summary** pane. The **Call Hierarchy** pane shows the function.
- 2 To navigate to a callee in the source code, double-click the callee name. These names are listed below the function name and marked by  (functions) or  (tasks). Alternatively, right-click the callee name and from the context menu, select **Go To Definition**.

The **Call Hierarchy** pane now shows the callee. In the **Source** pane, the current line shows the beginning of the callee function definition.

- 3 To navigate to a caller, double-click the caller name. These names are listed below the function name and marked by  (functions) or  (tasks). Alternatively, right-click the caller name and from the context menu, select **Go To Definition**.

The **Call Hierarchy** pane now shows the caller. In the **Source** pane, the current line shows the beginning of the caller function definition.

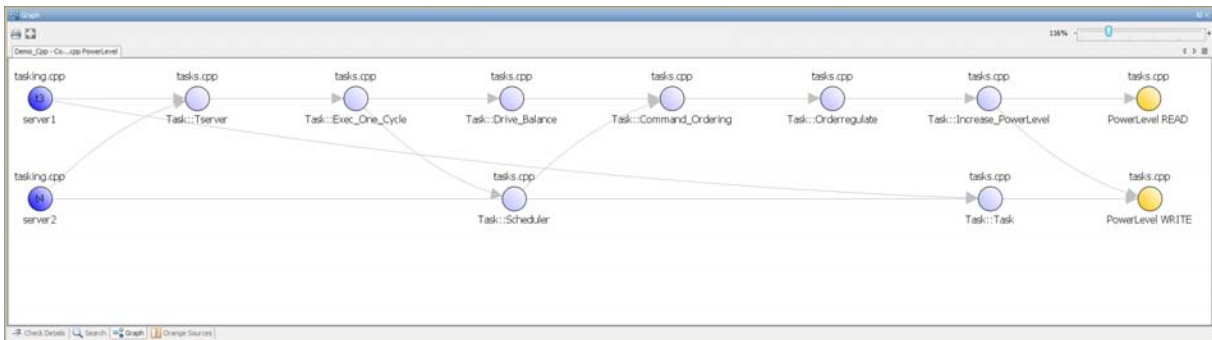
View Access Graph for Global Variables

This example shows how to display the access sequence for any global variable that is read or written in the code.

- 1 On the **Variable Access** pane, select the variable that you want to view.
- 2 On the **Variable Access** pane toolbar, click the Show Access Graph button



A window displays the access graph.



The access graph displays the read and write accesses for the variable. Each node represents a function.

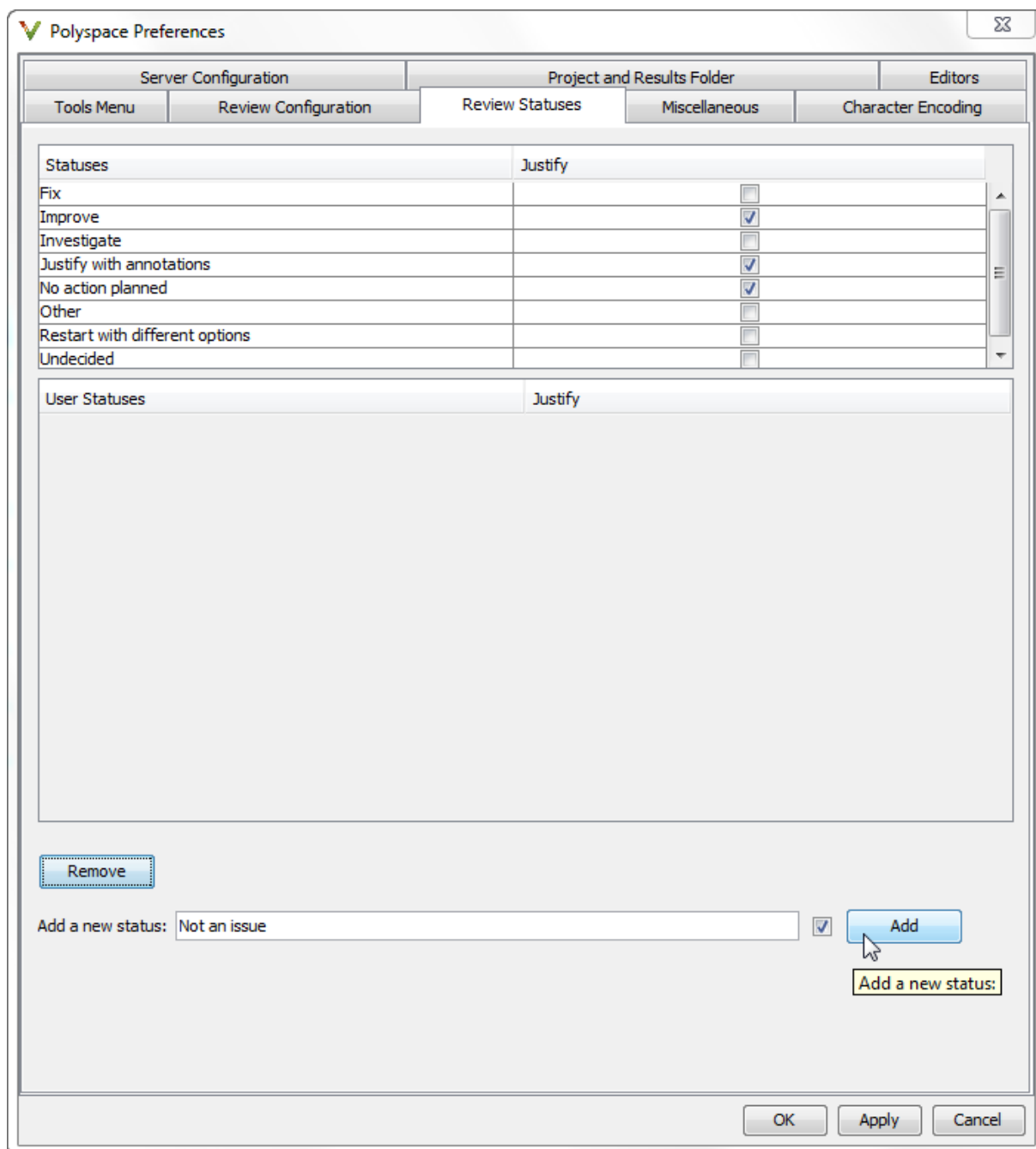
- 3 On the graph, click a node to navigate to the corresponding function on the **Source** pane. The **Call Hierarchy** pane displays the call tree of the function.

Customize Review Status

This example shows how to customize the statuses you assign on the **Check Review** pane.

Define Custom Status

- 1** Select **Options > Preferences**.
- 2** Select the **Review Statuses** tab.
- 3** Enter your new status at the bottom of the dialog box, then click **Add**.



The new status appears in the **Status** list.

4 Click **OK** to save your changes and close the dialog box.

When reviewing checks, you can select the new status from the **Check Review > Status** drop-down list.

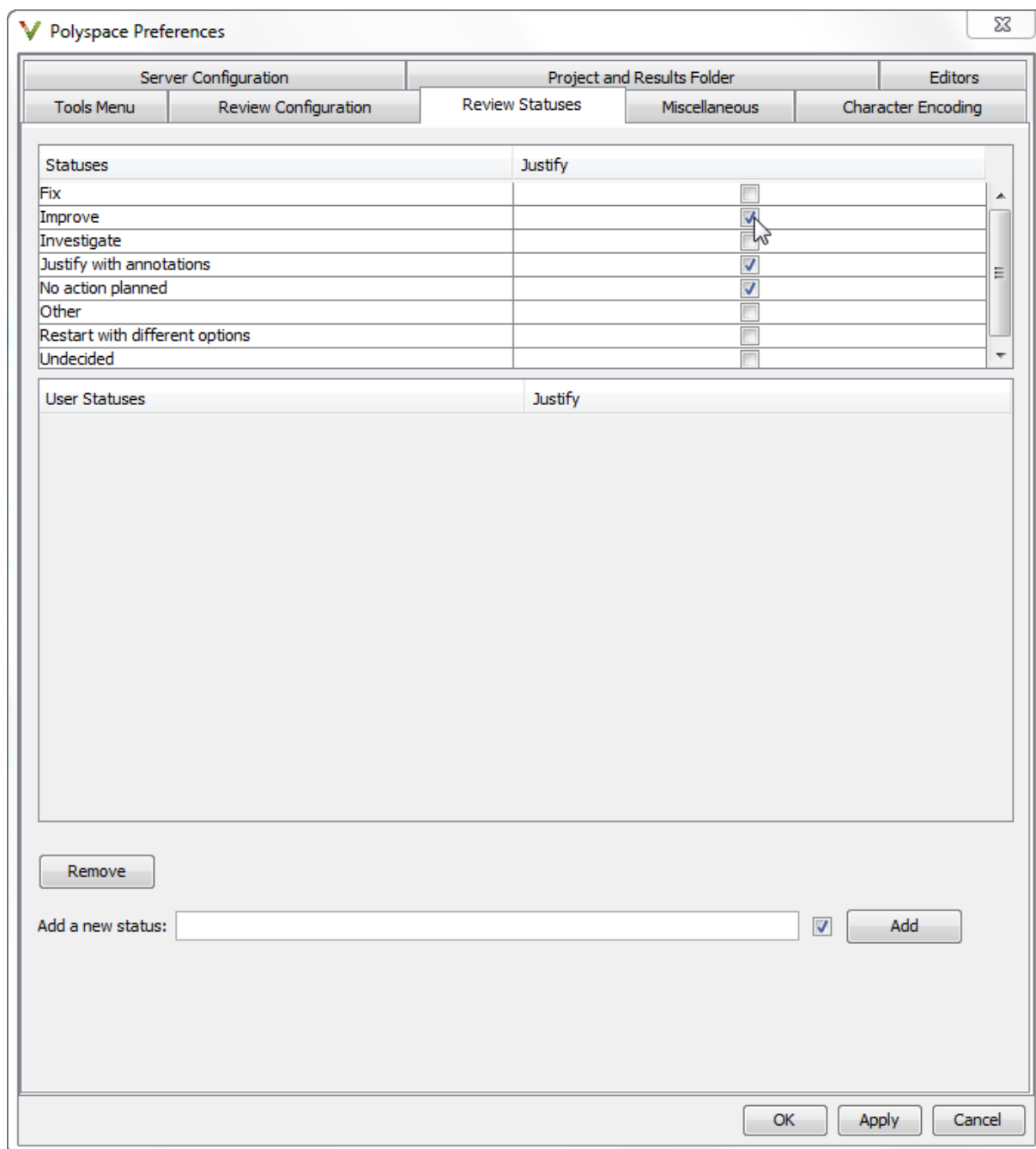
Add Justification to Existing Status

By default, a check is automatically justified if you assign the status, **Justify with annotations** or **No action planned**. However, you can change this default setting so that a check is justified when you assign one of the other existing statuses.

To add justification to existing status **Improve**:

1 Select **Options > Preferences**.

2 Select the **Review Statuses** tab. For the **Improve** status, select the check box in the **Justify** column. Click **OK**.



If you assign the Improve status to a check on the **Check Review** pane, the check gets automatically justified.

Use Range Information in Results Manager

This example shows how to use the variable range information available in the Results Manager.

View Range Information

- 1 On the **Source** pane, place your cursor over an operator or variable. A tooltip message displays the range information, if it is available.

The displayed range information represents a superset of dynamic values, which the software computes using static methods.

- 2 On the **Source** pane, select a check to display the error or warning message along with range information on the **Check Details** pane.

Interpret Range Information

The software uses the following syntax to display range information of variables:

name (data_type) : [min1 .. max1] or [min2 .. max2] or [min3 .. max3] or exact value

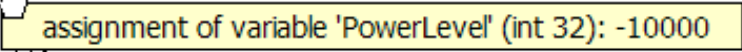
The following are examples of range information displayed in tooltips on the **Source** pane:

-

```

30  {
31    int temp;
32    PowerLevel = -10000;
33
34    RTE
35

```



The screenshot shows a code editor with a tooltip displayed over the variable 'PowerLevel' on line 32. The tooltip text is 'assignment of variable 'PowerLevel' (int 32): -10000'. The code lines are: 30 {, 31 int temp;, 32 PowerLevel = -10000;, 33, 34 RTE, 35.

The tooltip message indicates the variable `PowerLevel` is a 32-bit integer with the value `-10000`.

-

```

140
141 *depth = *depth + 1;
142 advance = 1.0f/(float)(*depth); /* potential division by zero */
143
144

```

assignment of variable 'advance' (float 32): [-1.0001 .. -4.6566E-10] or [1.9999E-2 .. 3.3334E-1]

The tooltip message indicates that the variable advance is a 32-bit float that lies between either -1.0001 and -4.6566E-10 or 1.9999E-2 and 3.3334E-1.



```

37
38 temp = read_on_bus();
39 switch(temp)
40 {

```

returned value of read_on_bus (int 32): full-range [-2³¹ .. 2³¹-1]

The tooltip message indicates that the returned value of the function read_on_bus is a 32-bit integer that occupies the full range of the data type, -2147483648 to 2147483647.



```

50
51 static s32 new_speed(s32 in, s8 ex_speed, u8 c_speed)
52 {
53     return (in / 9 + ((s32)ex_speed + (s32)c_speed) / 2 );
54 }
55
56 static char re
57 {

```

operator / on type int 32
left: [-1701 .. 3276]
right: 9
result: [-189 .. 364]

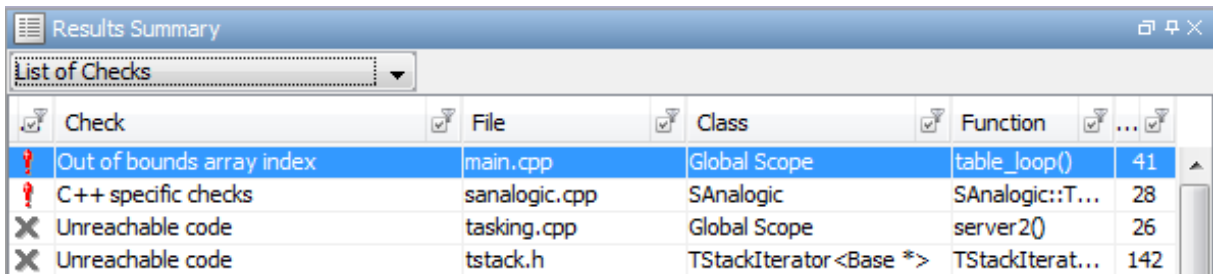
The tooltip message for the division operator / indicates that the:

- Operation is performed on 32-bit integers
- Dividend (left) is a value between -1701 and 3276

- Divisor (right) is an exact value, 9
- Quotient (result) lies between -189 and 364

Diagnose Errors with Range Information

Use range information to diagnose errors. Consider the check **Out of bounds array index** in the following example:



Check	File	Class	Function	
! Out of bounds array index	main.cpp	Global Scope	table_loop()	41
! C++ specific checks	sanalogic.cpp	SAnalogic	SAnalogic::T...	28
X Unreachable code	tasking.cpp	Global Scope	server2()	26
X Unreachable code	tstack.h	TStackIterator <Base *>	TStackIterat...	142

- 1 On the **Results Summary** pane, select the red check. Alternatively, select [on line 41 in the **Source** pane.

The **Check Details** pane displays the error message along with range information.



main.cpp / table_loop()

ID 1: Out of bounds array index
 Error: array index is outside its bounds : [0..3]
 array size: 4
 array index value: -1

```
41 array[j-1]->Draw();
```

Check Details Search Graph Orange Sources

The error message shows that the array size is 4, but the array index has a negative value of -1 .

- 2 Place the cursor over `j` in line 41 in the source code.

```

38
39 // Error: array index is outside its bounds
40 if (u.random_int())
41     array[j-1]->Draw();
42
43 return true;
44 }
45
46

```

array size: 4
array index value: -1
Press 'F2' for focus

Although `j` is green (as a local variable), it has a value 0. This results in a negative index range.

3 Look through the source code to identify the cause of the red check.

```

20
21 static bool table_loop(void)
22 {
23     int j = 4;
24
25     // Table of basic element
26     Base* array[] = { new SAnalogic, new Sensor, new Sensor, new SAnalogic };
27
28     for (int i = 4; i >= 0; i--, j--) {
29         array[i-1]->Draw();
30
31         // Error for the 2 last elements
32         // the TypeInfo function on
33         if (i > 2)
34             ((SAnalogic*)(array[i-1]))->TypeInfo();
35         else
36             (dynamic_cast<SAnalogic*>(array[i-1]))->TypeInfo();
37     }
38
39     // Error: array index is outside its bounds
40     if (u.random_int())
41         array[j-1]->Draw();
42
43     return true;
44 }
45
46

```

operator - on type int 32
left: [1..4]
right: 1
result: [0..3]
(result is truncated)
Press 'F2' for focus

In this case, `j` is initialized to 4 and decreased by 1 four times in the `for`-loop. Therefore the value of `j` after the `for`-loop is 0.

View Pointer Information in Results Manager

This example shows how to view information about pointers to variables or functions in the Results Manager.

View Pointer Information on Source Pane

Place your cursor over a check related to a pointer variable or dereference character ([, -, >, *). A tooltip message displays pointer information.

```

32  I pop() {
33      if (top == 0)
34          return 0;
35
36      StackNode* node = top;    // remember top node
37      top = top->next;
38
39      I data = node->data;    // remember node data
40      delete node;
41
42      return data;
43  };
44
45

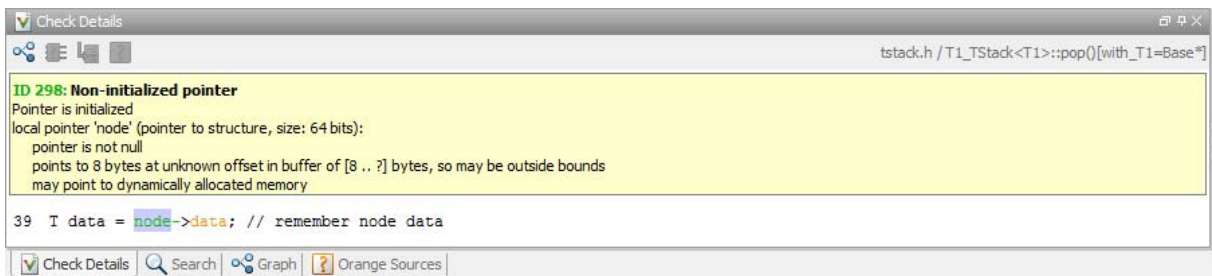
```

local variable 'node' (pointer to structure, size: 64 bits):
 pointer is not null
 points to 8 bytes at unknown offset in buffer of [8 .. ?] bytes, so may be outside bounds
 may point to dynamically allocated memory

Press 'F2' for focus

View Pointer Information on Check Details Pane

Click a check related to a pointer variable or dereference character. Further information about the check appears on the **Check Details** pane.

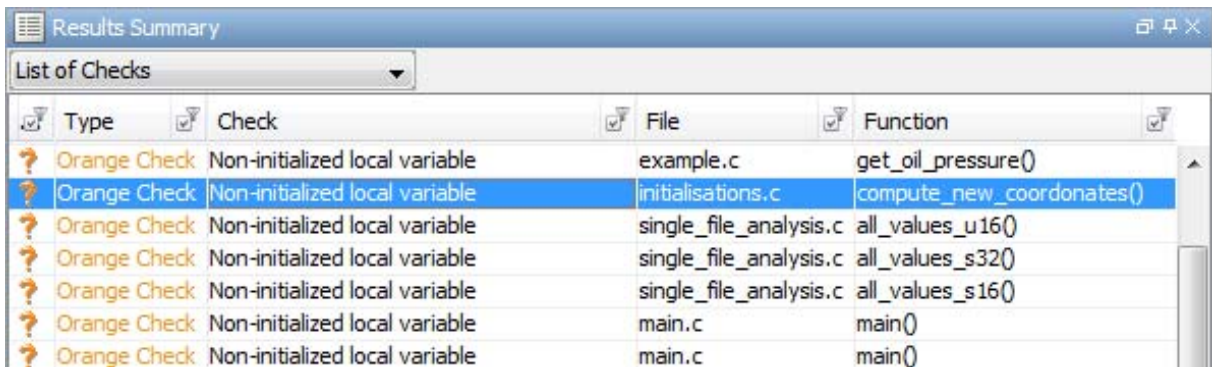


View Probable Cause for Checks

This example shows how to view the code sequence that is probably causing the check. In some cases, on the **Check Details** pane, the software outlines the subset of code causing the check.

View Code Sequence Causing Check

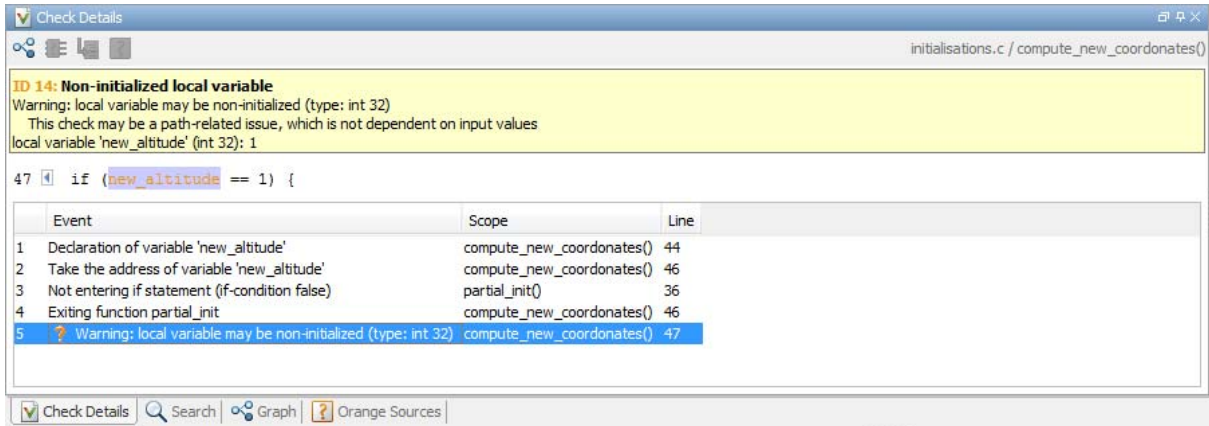
- 1 Open the results file with extension `.pscp`.
- 2 On the **Results Summary** pane, select a check.



Type	Check	File	Function
Orange Check	Non-initialized local variable	example.c	get_oil_pressure()
Orange Check	Non-initialized local variable	initialisations.c	compute_new_coordonates()
Orange Check	Non-initialized local variable	single_file_analysis.c	all_values_u16()
Orange Check	Non-initialized local variable	single_file_analysis.c	all_values_s32()
Orange Check	Non-initialized local variable	single_file_analysis.c	all_values_s16()
Orange Check	Non-initialized local variable	main.c	main()
Orange Check	Non-initialized local variable	main.c	main()

In this example, the check Non-initialized local variable on line 47 is selected.

- 3 On the **Check Details** pane, view the code sequence causing the check.



Each statement of the sequence contains a line number and a comment. You can use the information to understand and rectify your code.

- 4 To navigate to a statement in the code sequence, on the **Check Details** pane, click the statement. The **Source** pane displays the relevant code.
- 5 To navigate to the probable cause of the check, on the **Results Summary** pane, right-click the check. From the context menu, select **Go To Cause**.

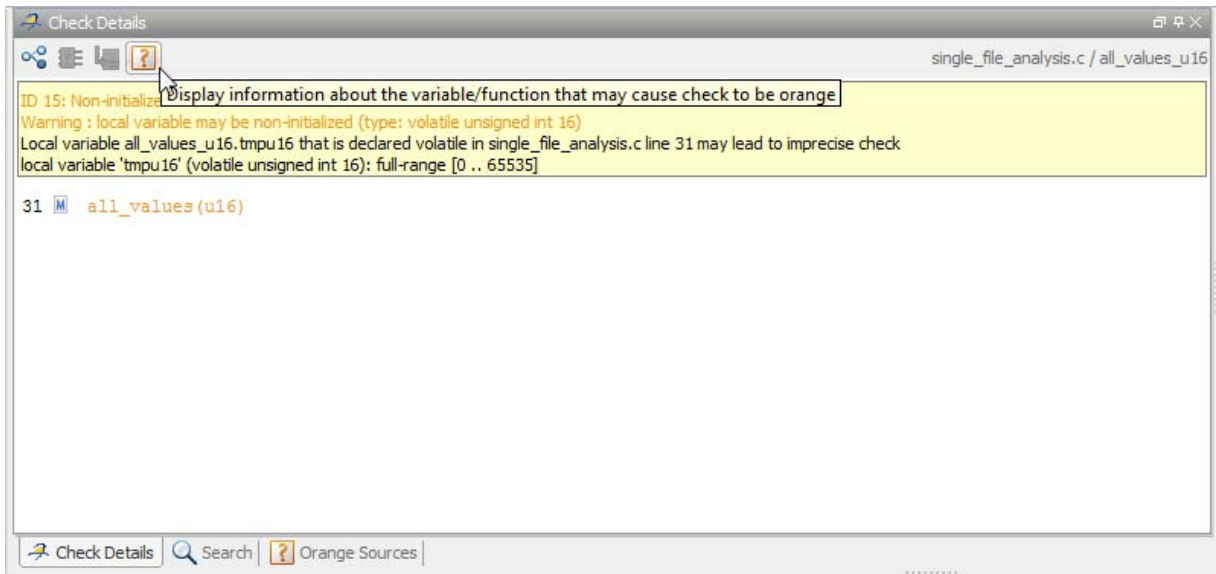
View Input Variables or Functions Causing Check

For orange checks caused by input, when the code sequence is not available, the software provides more information on input variables or functions causing the check. To view this information:

- 1 On the **Results Summary** pane, select the check.

Further information on the check appears on the **Check Details** pane.

- 2 On the **Check Details** pane, select the icon.



Information on source variables or functions causing the orange check appears on the **Orange Sources** tab in the **Check Details** pane.

Check Colors

Polyspace software presents verification results as colored entries in the source code. There are four main colors in the results:

- **Red** – Indicates code that always has an error (errors occur every time the code is executed).
- **Gray** – Indicates unreachable code (dead code).
- **Orange** – Indicates unproven code (code might have a run-time error).
- **Green** – Indicates code proved not to have a run-time error.
- **Dark Orange** – Indicates unproven code that most likely has a run-time error

When reviewing verification results, remember these rules:

- An instruction is verified only if no run-time error is detected in the previous instruction.
- The verification assumes that each run-time error causes a “core dump.” The corresponding instruction is considered to have stopped, even if the actual run-time execution of the code might not stop. With orange checks, only the green parts propagate through to subsequent checks.
- Focus on the verification message. Do not jump to false conclusions. You must understand the color of a check step by step, until you find the root cause of a problem.
- Determine the cause by examining the actual code. Do not focus on what the code is supposed to do.

Concepts

- “Source Code Colors” on page 10-65

Source Code Colors

Polyspace uses the following color scheme for displaying code on the **Source** pane.

- For every check on the **Results Summary** 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.

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.
- For every coding rule violation on the **Results Summary** 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++.
 - A ▼ symbol if the coding rule is a custom rule.
- 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.
- When a function is defined, Polyspace colors the function name in blue.

Concepts

- “Check Colors” on page 10-64
- “Source” on page 10-71

Results Manager Overview

The Results Manager perspective has the following panes below the toolbar:

Pane	Function
“Results Summary” on page 10-67	List of checks (diagnostics) for each file and function in the project
“Dashboard” on page 10-80	<ul style="list-style-type: none"> • Graphical view of code coverage and check distribution • Top five orange checks (likely errors in unproven code) and purple checks (coding rule violations)
“Source” on page 10-71	Source code for a selected check in the procedural entities view
“Check Details” on page 10-86	Details about the selected check
“Review Statistics” on page 10-87	Statistics about the review progress for checks with the same type and category as the selected check
“Check Review” on page 10-87	Review information about selected check
“Variable Access” on page 10-94	Information about global variables declared in the source code
“Call Hierarchy” on page 10-91	Tree structure of function calls

You can resize or hide these sections.

Results Summary

The **Results Summary** pane lists all checks along with their attributes. To organize your check review, from the drop-down list on this pane, select one of the following options:

- **List of checks:** Lists all checks without any grouping. The checks are sorted in the following order:
 - 1 **Red:** Indicates code that is proven to contain an error. The check indicates that the code will fail every time it is executed.
 - 2 **Gray** — Indicates unreachable code.
 - 3 **Orange** — Indicates unproven code that might contain an error.
 - 4 **Purple.** — Indicates coding rule violation.
 - 5 **Green** — Indicates code that is proven to not contain an error.
- **Checks by Family:** Lists all checks grouped by color. Within each color, the checks are grouped by category. For more information on the checks covered by a category, see the check reference pages.
- **Checks by Class:** Lists all checks grouped by class. Within each class, the checks are grouped by method. The first group, **Global Scope**, lists all checks not occurring in a class definition.

This option is available for C++ code only.
- **Checks by File/Function:** Lists all checks grouped by file. Within each file, the checks are grouped by function.

For each check, the **Results Summary** pane contains the check attributes, listed in columns:

Attribute	Description
Family	Group to which the check belongs. For instance, if you choose the grouping Checks by File/Function, this column contains the name of the file and function containing the check.
ID	Unique identification number of the check. In the default view on the Results Summary pane, the checks appear sorted by this number.
Type	Check color
Category	Category of the check. For more information on the checks covered by a category, see the check reference pages.
Check	Description of the error
Information	For run-time errors, this attribute indicates whether the check is related to path or bounded input values. For coding rule violations, this attribute indicates whether the rule is Required .
File	File containing the instruction where the check occurs
Class	Class containing the instruction where the check occurs. If the check is not inside a class definition, then this column contains the entry, Global Scope .
Function	Function containing the instruction where the check occurs. If the function is a method of a class, it appears in the format <i>class_name::function_name</i> .

Attribute	Description
Line	Line number of the instruction where the check occurs.
Col	Column number of the instruction where the check occurs. The column number is the number of characters from the beginning of the line.
%	Percentage of checks that are not orange. This column is most useful when you choose the grouping Checks by File/Function . The entry in this column against a file or function indicates the percentage of checks in the file or function that are not orange.
Classification	Level of severity you have assigned to the check. The possible levels are: <ul style="list-style-type: none">• Unset• High• Medium• Low• Not a defect
Status	Review status you have assigned to the check. The possible statuses are: <ul style="list-style-type: none">• Fix• Improve• Investigate• Justify with annotations• No action planned• Other• Restart with different options

Attribute	Description
Justified	Check boxes showing whether you have justified the checks
Comments	Comments you have entered about the check

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:

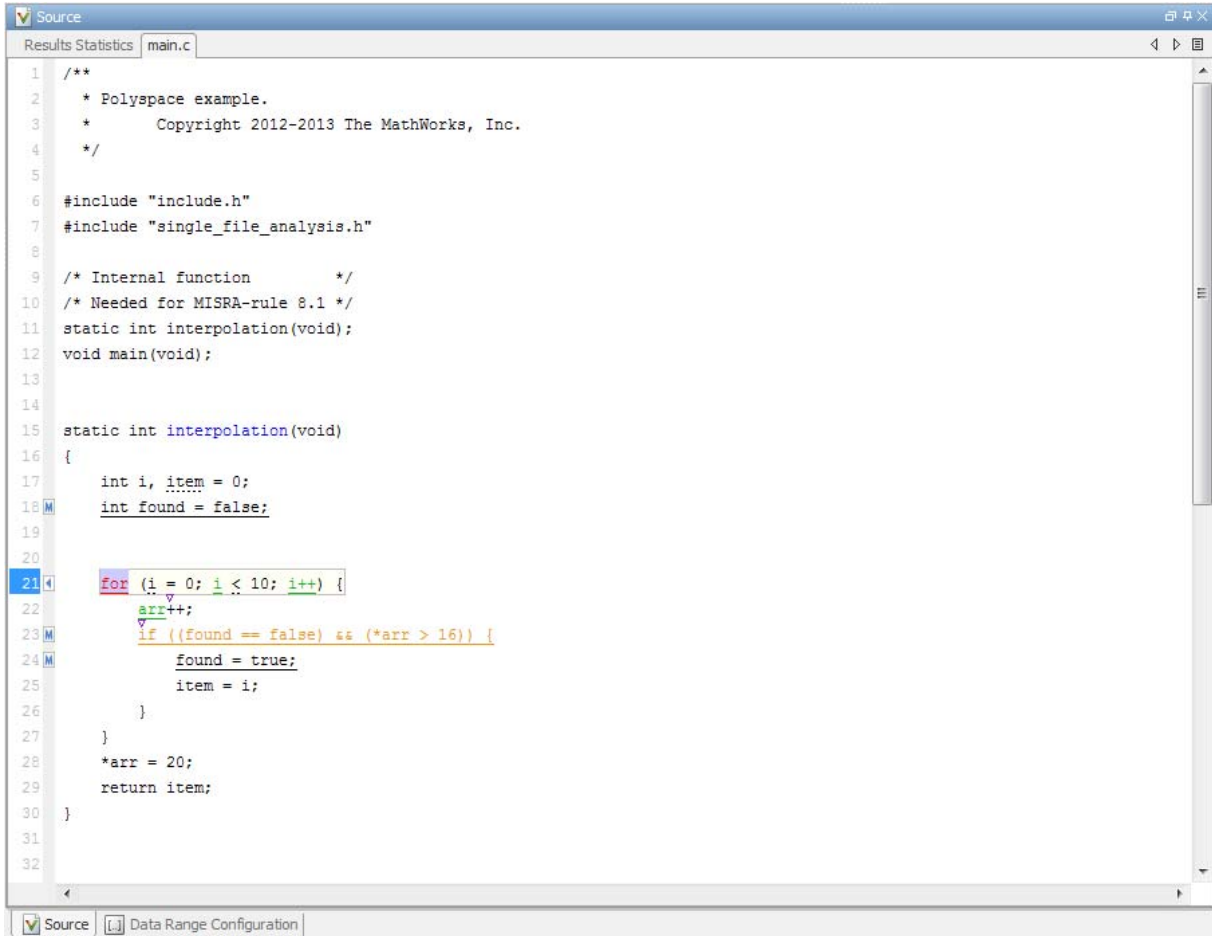
- Navigate through the checks. For more information, see “Assign Review Status to Result” on page 10-18.
- Organize your check review using filters on the columns. For more information, see “Organize Results Using Filters and Groups” on page 10-34.

Source

In this section...
“Source” on page 10-71
“Dashboard” on page 10-80

Source

The **Source** pane shows the source code with colored checks highlighted.



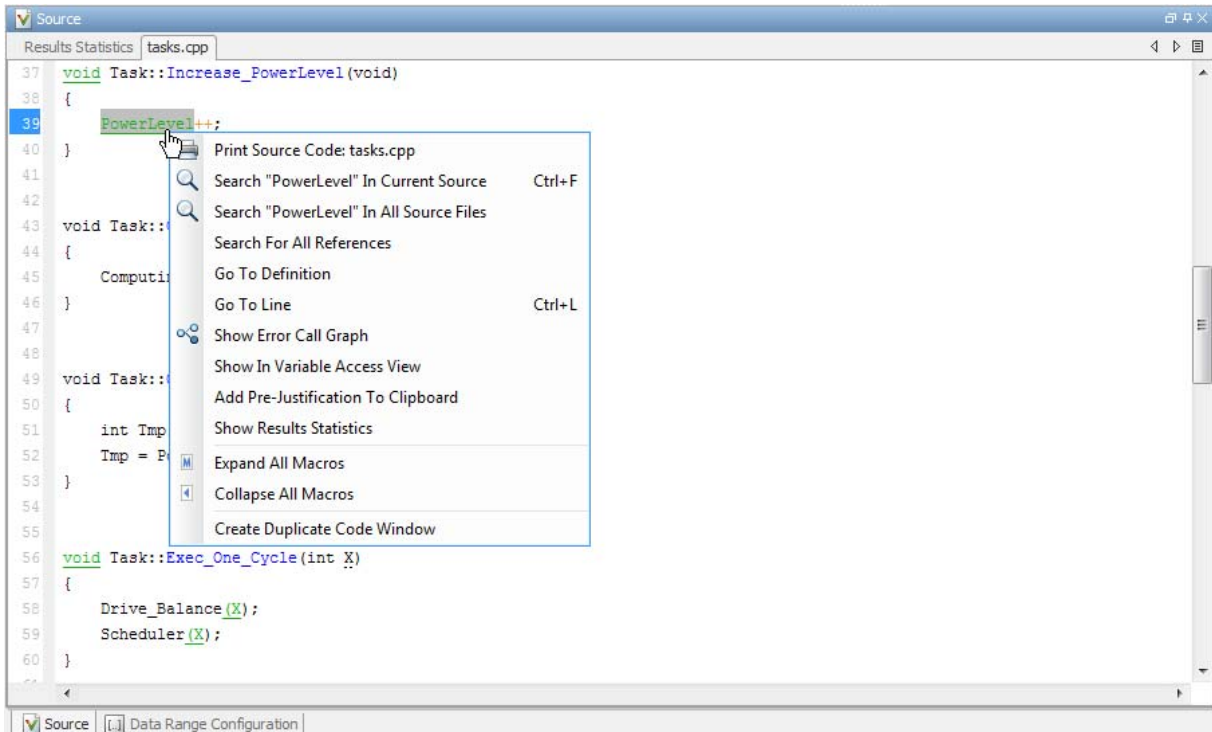
```
1  /**
2   * Polyspace example.
3   *   Copyright 2012-2013 The MathWorks, Inc.
4   */
5
6  #include "include.h"
7  #include "single_file_analysis.h"
8
9  /* Internal function      */
10 /* Needed for MISRA-rule 8.1 */
11 static int interpolation(void);
12 void main(void);
13
14
15 static int interpolation(void)
16 {
17     int i, item = 0;
18     int found = false;
19
20
21     For (i = 0; i < 10; i++) {
22         arr++;
23         if ((found == false) && (*arr > 16)) {
24             found = true;
25             item = i;
26         }
27     }
28     *arr = 20;
29     return item;
30 }
31
32
```

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** — 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.
- **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.

-

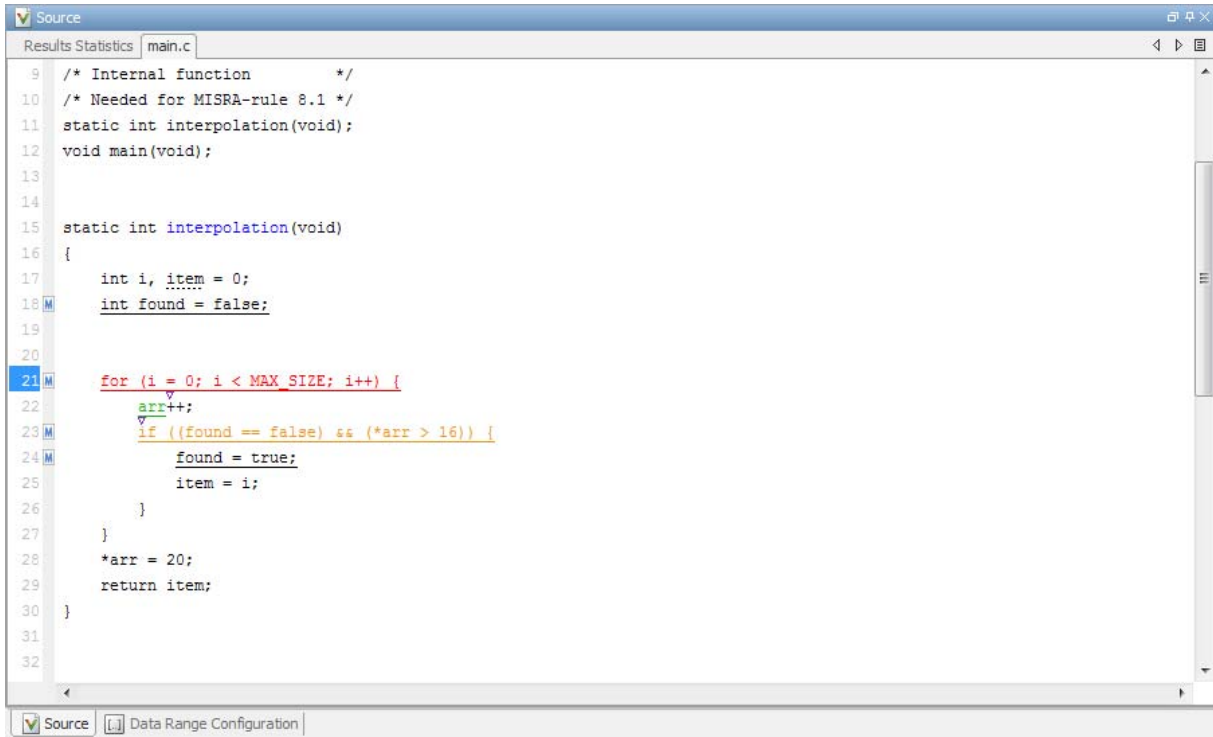
View Variable Range

Place your cursor over a check to view range information for variables, operands, function parameters, and return values. For more information, see “Use Range Information in Results Manager” on page 10-55

-

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.



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

```
9  /* Internal function      */
10 /* Needed for MISRA-rule 8.1 */
11 static int interpolation(void);
12 void main(void);
13
14
15 static int interpolation(void)
16 {
17     int i, item = 0;
18     int found = false;
19
20
21     for (i = 0; i < MAX_SIZE; i++) {
22         arr++;
23         if ((found == false) && (*arr > 16)) {
24             found = true;
25             item = i;
26         }
27     }
28     *arr = 20;
29     return item;
30 }
31
32
```

Line 21 is highlighted in blue. A small 'M' icon is visible to the left of line 21. A small 'v' icon is visible below line 22. A small 'M' icon is visible to the left of line 23. A small 'M' icon is visible to the left of line 24. The bottom of the window shows a status bar with "Source" and "Data Range Configuration" buttons.

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



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

```
9  /* Internal function      */
10 /* Needed for MISRA-rule 8.1 */
11 static int interpolation(void);
12 void main(void);
13
14
15 static int interpolation(void)
16 {
17     int i, item = 0;
18     int found = false;
19
20
21     For (i = 0; i < 10; i++) {
22         arr++;
23         if ((found == false) && (*arr > 16)) {
24             found = true;
25             item = i;
26         }
27     }
28     *arr = 20;
29     return item;
30 }
31
32
```

The region from line 21 to 26 is highlighted in blue. A small arrow icon is visible on the left side of line 21, indicating that the content is collapsed. The bottom of the window shows a toolbar with "Source" and "Data Range Configuration" buttons.

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:

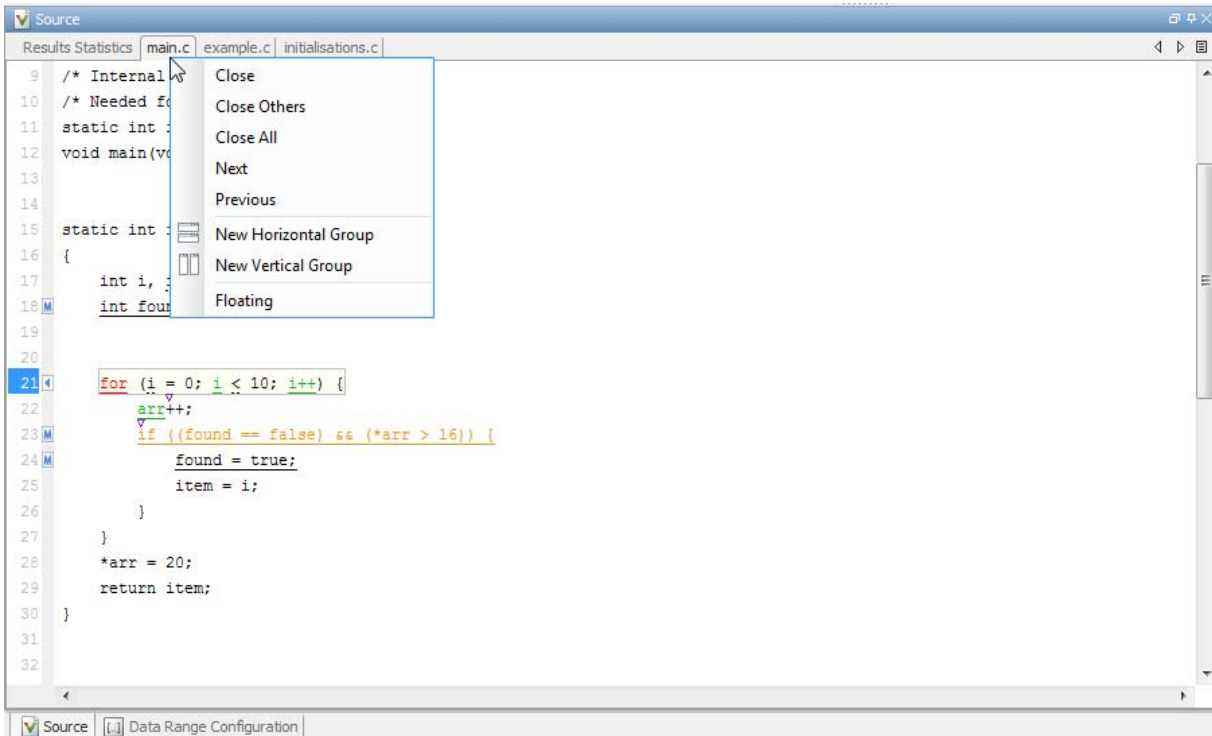
- 1 Right-click any point within the source code view.
- 2 From the context menu, select either **Expand All Macros** or **Collapse All Macros**.

Note The **Check Details** pane also allows you to view the contents of a macro if the check you select lies within a macro.

Manage Multiple Files

You can view multiple source files in the **Source** pane.

On the **Source** pane toolbar, right-click a view.



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

- **Close** – Close the currently selected source file.
- **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

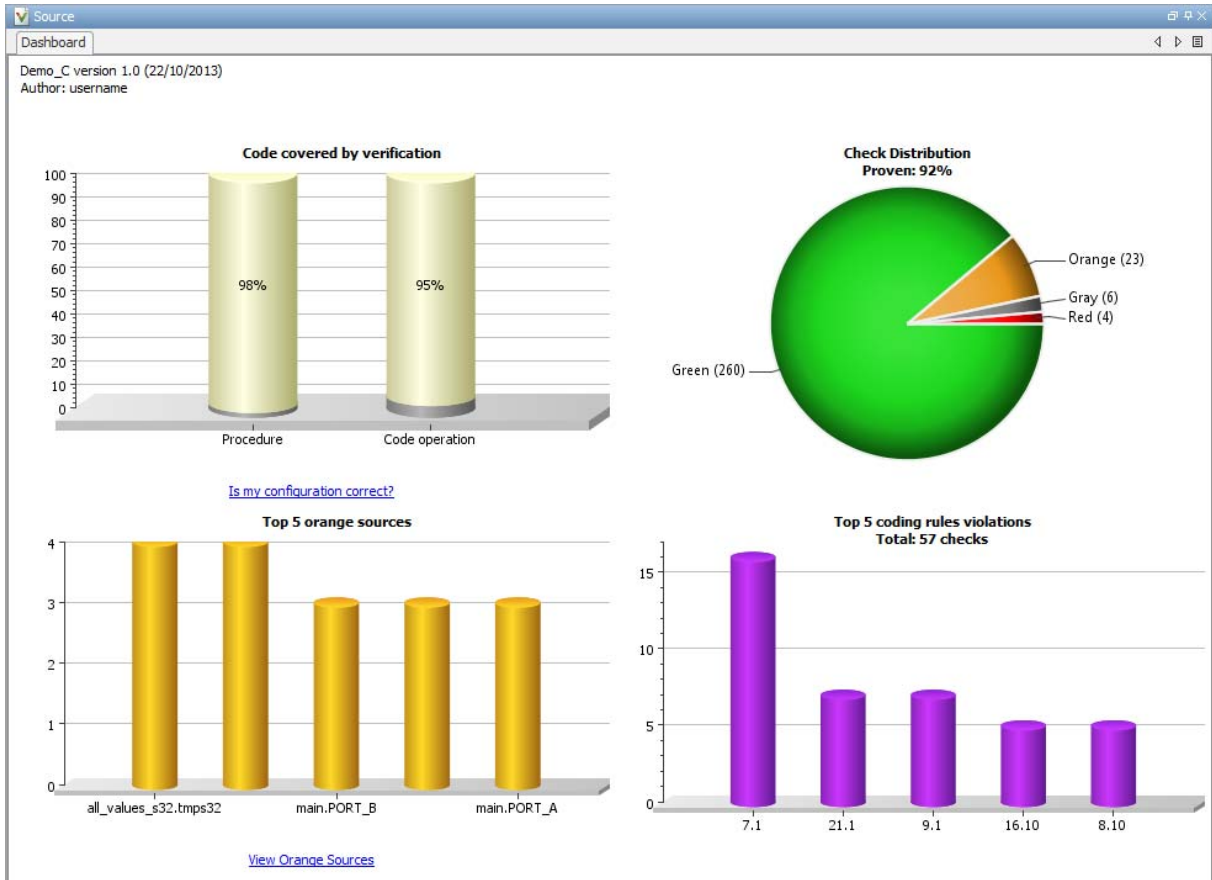
On the **Source** pane, to highlight a block of code, click either its opening or closing brace.

```
Source
Dashboard code_unreachable.c
20
21 int main (void)
22 {
23     int i, res_end;
24     enumState inter;
25
26     res_end = State(Init);
27     if (res_end == 0) {
28         res_end = State(End);
29         inter = (enumState)intermediate_state(0);
30         if (res_end || inter == Wait) { // UNR code on inter== Wait
31             inter = End;
32         }
33         // use of I not initialized
34         if (random_int()) {
35             inter = (enumState)intermediate_state(i); // NIV ERROR
36             if (inter == Intermediate) { // UNR code because of NIV ERROR
37                 inter = End;
38             }
39         }
40     } else {
41         i = 1; // UNR code
42         inter = (enumState)intermediate_state(i); // UNR code
43     }
44     if (res_end) { // UNR code always reached, but no else
45         inter = End;
46     }
47
48     return res_end;
49 }
50
```

Note This action does not highlight the code block if the brace itself is already highlighted. The opening brace can be highlighted, for instance, if there is an **Unreachable code** error on the code block.

Dashboard

The **Dashboard** tab on the **Source** pane provides statistics on the verification results in a graphical format.



In the Results Manager perspective, this tab is displayed by default when you open a results file with extension `.pscp`. On this tab, you can view four graphs and charts:

-

Code covered by verification

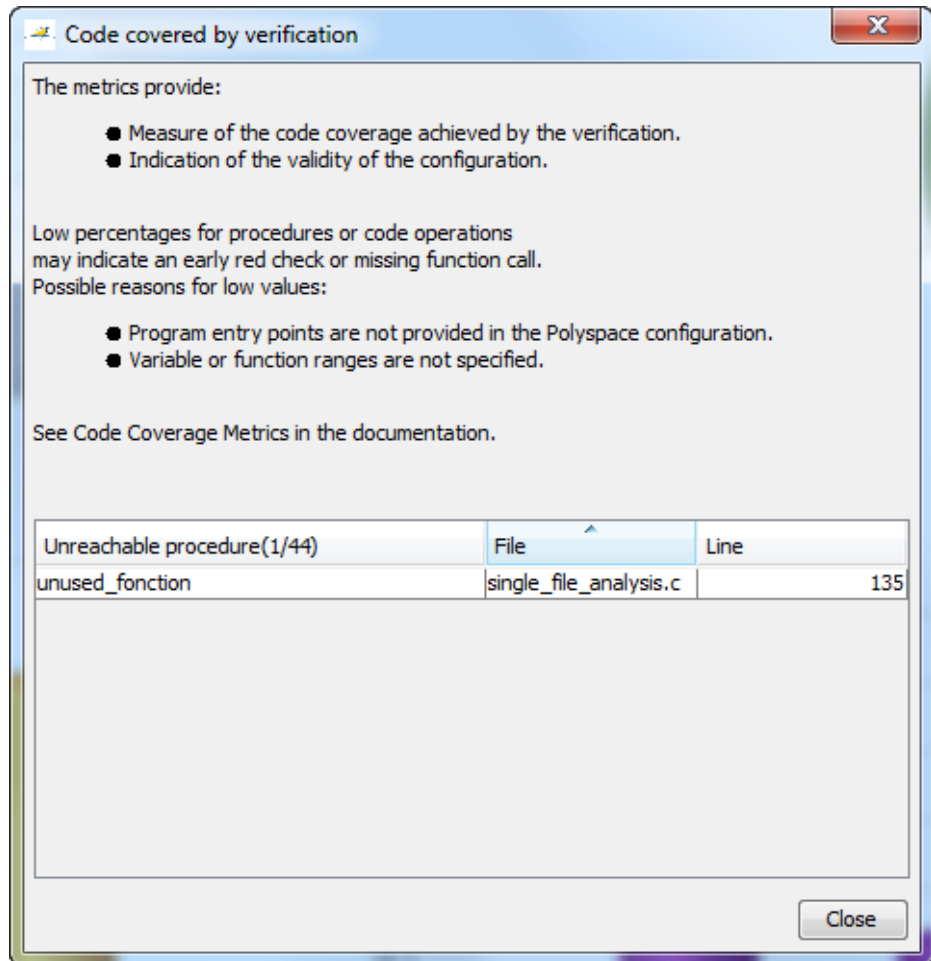
This column graph displays:

- The percentage of procedures covered by verification. You can see this percentage in the Procedure column.
- The percentage of elementary operations in executable procedures covered by verification. You can see this percentage in the Code operation column.

These percentages provide a measure of:

- Code coverage achieved by the Polyspace verification.
- Validity of your Polyspace configuration.

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 NIVL 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 ZDV check.
- On line 5, for the division operation, an OVFL check.
- On line 6, for the addition operation, an OVFL check.
- On line 6, for another read operation on `x`, an NIVL 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()`.

•

Check distribution

This pie chart displays the number of checks of each color. For a description of the check colors, see “Check Colors” on page 10-64.

Using this pie chart, you can obtain an estimate of:

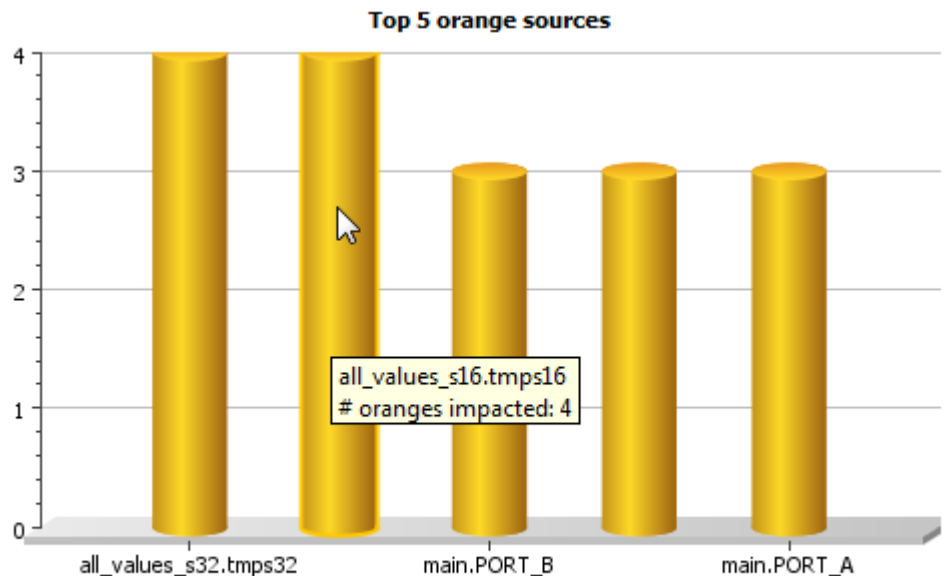
- The number of checks to review.
- The selectivity of your verification — the fraction of checks that are not orange.

•

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.



[View Orange Sources](#)

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. For information on the **Orange Sources** pane, see “View Sources of Orange Checks” on page 11-28.

- Prioritize your review of orange checks. For more information, see “Prioritize Orange Check Review” on page 11-30. If there are sources affecting a large number of orange checks, using this chart can quickly improve the selectivity of your verification.

-

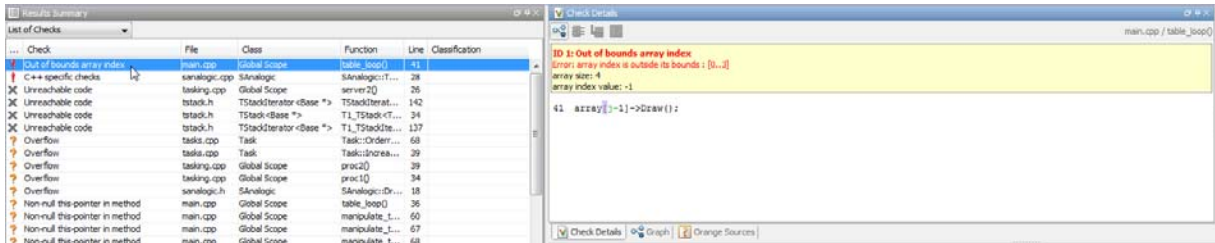
Top 5 coding rule violations

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 Rules” on page 12-18, “Supported MISRA C++ Coding Rules” on page 12-69 and “Supported JSF C++ Coding Rules” on page 12-96.

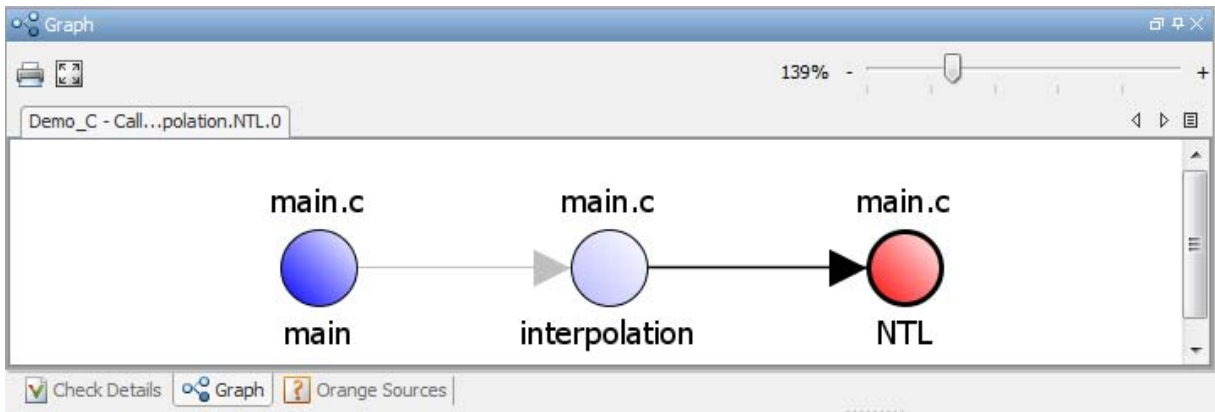
Check Details

On the **Results Summary** pane, if you click a check, you see additional information on the **Check Details** pane.



Error Call Graph

Click the **Show error call graph** icon,  in the **Check Details** pane toolbar to display the call sequence that leads to the code associated with a check.



For more information, see “View Call Sequence for Checks” on page 10-42.

Check Review

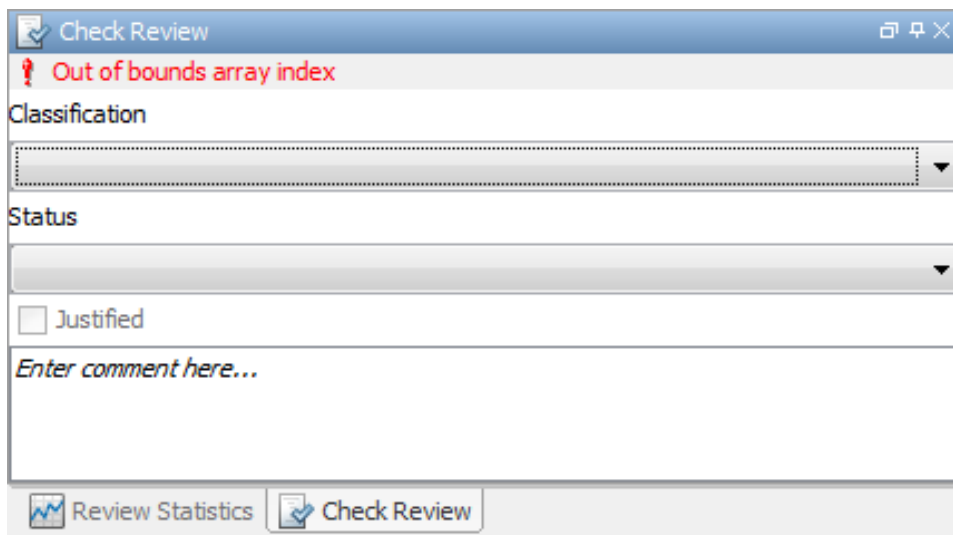
In this section...

“Check Review” on page 10-87

“Review Statistics” on page 10-87

Check Review

When reviewing checks, use the **Check Review** tab to assign a **Classification** and **Status** to each check. You can also enter comments to describe the results of your review. This action helps you track the progress of your review and avoid reviewing the same check twice.



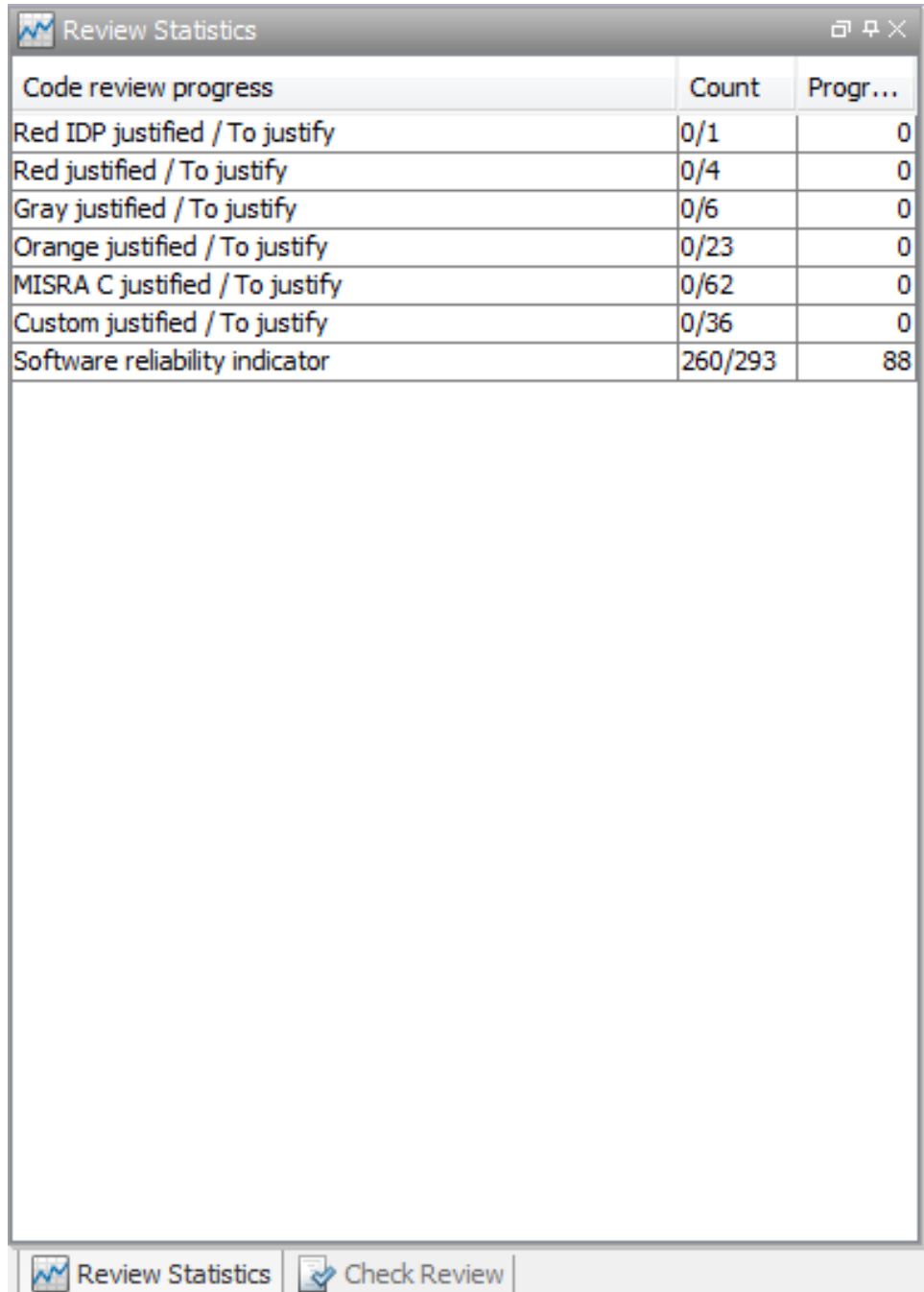
The screenshot shows a dialog box titled "Check Review" with a close button. The main content area displays a red error message: "Out of bounds array index". Below the message are two dropdown menus: "Classification" and "Status". Under the "Status" dropdown is a checkbox labeled "Justified". At the bottom of the main area is a text input field with the placeholder text "Enter comment here...". The dialog box has a footer with two buttons: "Review Statistics" (with a bar chart icon) and "Check Review" (with a checkmark icon).

For more information, see “Assign Review Status to Result” on page 10-18.

Review Statistics

The **Review Statistics** view displays statistics about how many run-time checks and coding rule violations you have reviewed. As you review checks, the software updates these statistics.

Consider the following **Review Statistics** view:



Code review progress	Count	Progr...
Red IDP justified / To justify	0/1	0
Red justified / To justify	0/4	0
Gray justified / To justify	0/6	0
Orange justified / To justify	0/23	0
MISRA C justified / To justify	0/62	0
Custom justified / To justify	0/36	0
Software reliability indicator	260/293	88

Review Statistics

Check Review

The **Count** column displays a ratio and the **Progress** column displays the equivalent percentage.

The first row displays the ratio of justified checks to total checks that have the same color and category of the current check. In this case, the first row displays the ratio of reviewed red IDP checks to total red IDP errors in the project.

The second, third, and fourth rows display the ratio of justified checks to total checks for red, gray, and orange checks respectively.

If you specified coding rules checking, the next row displays the ratio of justified coding rule violations to total coding rule violations.

The last row displays the ratio of the number of green checks to the total number of run-time checks, providing an indicator of the reliability of the software.

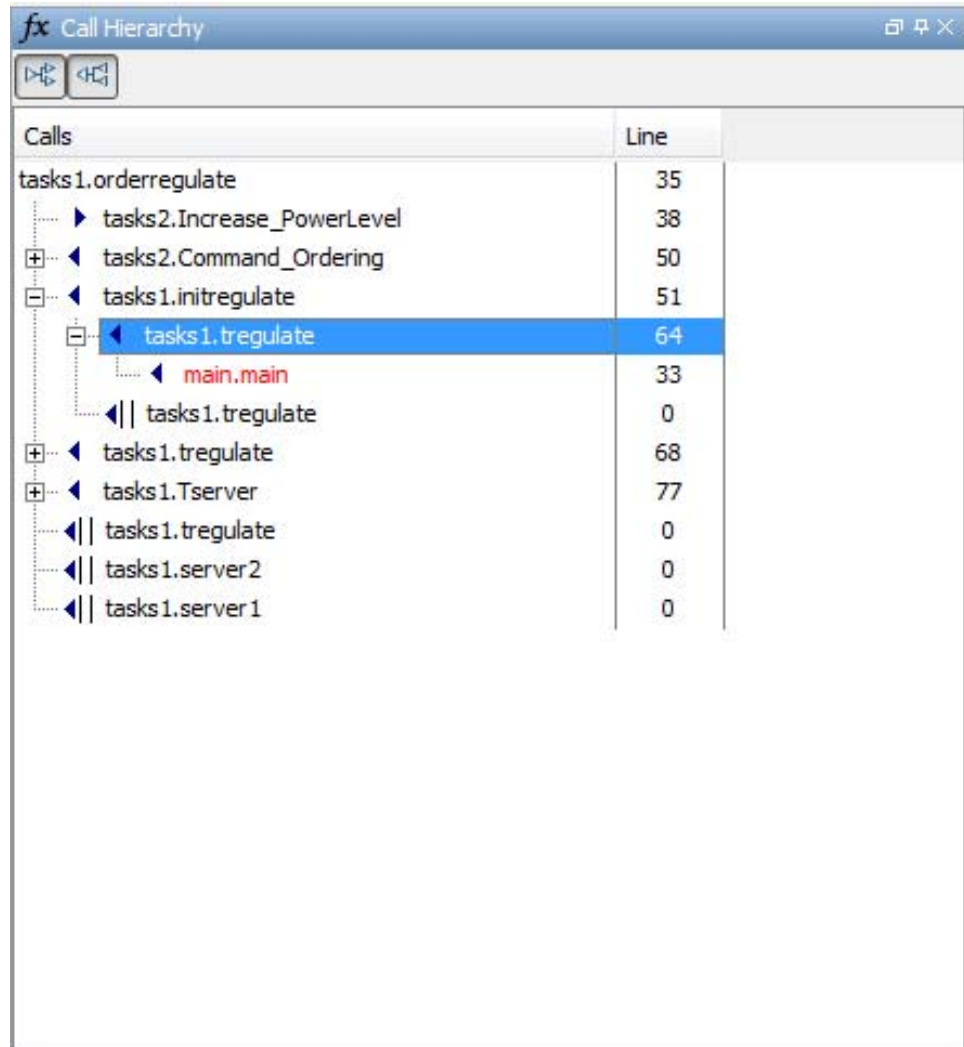
Call Hierarchy

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.

For more information, see “View Call Tree for Functions” on page 10-44.

In the following example, the **Call Hierarchy** pane displays the function, `orderregulate`, in the file, `tasks1.c`. It also displays the callers and the callees of `orderregulate`.



Calls	Line
tasks1.orderregulate	35
▶ tasks2.Increase_PowerLevel	38
⊕ ◀ tasks2.Command_Ordering	50
⊖ ◀ tasks1.initregulate	51
⊖ ◀ tasks1.tregulate	64
◀ main.main	33
◀◀ tasks1.tregulate	0
⊕ ◀ tasks1.tregulate	68
⊕ ◀ tasks1.Tserver	77
◀◀ tasks1.tregulate	0
◀◀ tasks1.server2	0
◀◀ tasks1.server1	0

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 `tasks1.orderregulate` begins on line 35.

- For a callee name, the number refers to the line where the callee is called. In the preceding example, callee, `tasks2.Increase_PowerLevel1`, is called by `tasks1.orderregulate` on line 38.
- For a caller name, the number refers to the line where the caller calls the function. In the preceding example, caller, `tasks2.Command_Ordering`, calls `tasks1.orderregulate` on line 50.

Tip Select a caller or callee name to navigate to the function call 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



-

Go to Caller/Callee Definition

Go directly to the definition of a caller or callee in the source code. Right-click the name of the caller or callee and select **Go to definition**. For more information, see “Navigate Call Tree” on page 10-47.

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.

Variables	Values	# Reads	# Writes	Written by task	Read by task	Protection	Usage	Scalar	Line	Col	File	Type	Detailed Type
Demo_Cpp_C													
example_beta	0.0 or [0.09999 .. 0.5]	1	2						136	6	example.c	float	
example_init_globals	0.0								136	6	example.c		
example.Square_Root_conv(float;float*)	[0.09999 .. 0.5]								140	8	example.c		
example.Square_Root()	[0.09999 .. 0.5]								149	28	example.c		
initialisations.arr		3	2						11	5	initialisations.c	int*	
initialisations.current_data		2	2						8	13	initialisations.c	int*	
initialisations.first_payload	100	1	3						13	4	initialisations.c	int	
initialisations.tab	0 or 12	1	3						10	4	initialisations.c	int[10]	
main.current_data		3	1						11	12	main.c	int*	
sensitivity.array	0	1	1						8	4	sensitivity.c	int[10]	
tasks1.Injection	0	1	1						24	4	tasks1.c	int	
tasks1.PowerLevel	[-2147483639 .. 2 ³¹ -1]	4	3	t3 t4 t5	t3 t4 t5		shared		20	4	tasks1.c	int	
tasks1.SHR	0 or 22	1	2	t3 t4	t5	Critical section	shared		25	11	tasks1.c	int	
tasks1.SHR2	0 or 22	1	3	t3 t4	t5		shared		26	11	tasks1.c	int	
tasks1.SHR3	[0 .. 51]	1	2						27	11	tasks1.c	int	
tasks1.SHR4		2	3	t2 t3 t4 t5	t2 t3 t4 t5		shared		22	4	tasks1.c	class rec	
tasks1.SHR5	5 or 28	2	2	t1	t1 t2		shared		23	4	tasks1.c	int	
tasks1.SHR6	0	2	1						28	11	tasks1.c	int	
__polyspace_stdstubs.errno	0	0	2						383	15	__polyspace_...	int	

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, <i>File_Name</i> . <i>Variable_NameFile_Name</i> : Name of file where variable is declared
Values	Value (or range of values) of variable

Attribute	Description
# Reads	Number of times the variable is read
# Writes	Number of times the variable is written
Written by task	<p>Name of tasks writing on variable using aliases, t1,t2,t3</p> <hr/> <p>Tip To see the full names for aliases, right-click anywhere on the Variable Access pane and select Show Legend.</p> <hr/>
Read by task	Name of tasks reading variable using aliases, t1,t2,t3
Protection	<p>Whether shared variable is protected from concurrent access</p> <p>(Filled only when Usage column has entry, Shared)</p> <p>The possible entries in this column are:</p> <ul style="list-style-type: none"> • Critical Section: If variable is accessed in critical section of code • Temporal Exclusion: If variable is accessed in mutually exclusive tasks <p>For more details on these entries, see “Shared Variables” on page 7-45.</p>
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

Attribute	Description
File	Source file containing variable declaration
Detailed 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”.

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, <i>File_Name</i> . <i>Function_NameFile_Name</i> : Name of file containing function (or task) definition
Values	Value or range of values of variable in the function or task performing read/write access
Written by task	<i>Only for tasks</i> : Name of task performing write access on variable
Read by task	<i>Only for tasks</i> : Name of task performing read access on variable
Line	Line number where function or task accesses variable
Col	Column number where function or task accesses variable
File	Source file containing access operation on variable

For example, consider the global variable, SHR2:

Variables	Values	# Reads	# Writes	Written by task	Read by task	Protection	Usage	Scalar	Line	Col	File	Type	Detailed Type
example_beta	0.0 or [0.09999 .. 0.9]	1	2						136	6	example.c	float	
initializers.arr		3	2						11	5	initializers.c	int*	
initializers.current_data		2	2						8	13	initializers.c	int*	
initializers.first_payload	100	1	3						13	4	initializers.c	int	
initializers.tab	0 or 12	1	3						10	4	initializers.c	int[10]	
main.current_data		3	1						11	12	main.c	int*	
sensitivity.array		1	1						8	4	sensitivity.c	int[10]	
tasks1.Injection	0	1	1						24	4	tasks1.c	int	
tasks1.PowertLevel	[-2147483639 .. 231.1]	4	3	13 14 15	13 14 15		shared		20	4	tasks1.c	int	
tasks1.SHR1	0 or 22	1	2	13 14	15	Critical section	shared		25	11	tasks1.c	int	
tasks1.SHR2	0 or 22	1	3	13 14	15		shared		26	11	tasks1.c	int	
tasks1.int_globals													
tasks1.Tserver()	22								11	5	tasks1.c		
tasks1.Tserver()	0								91	6	tasks1.c		
tasks1.intregulate()	0 or 22								59	10	tasks1.c		
tasks1.server1				13									
tasks1.server2				14									
tasks1.tregulate					15								
tasks1.SHR3	[0 .. 51]	1	2						27	11	tasks1.c	int	
tasks1.SHR4		2	3	12 13 14 15	12 13 14 15		shared		22	4	tasks1.c	class rec	
tasks1.SHR5	5 or 28	2	2	11	11 12		shared		23	4	tasks1.c	int	
tasks1.SHR6	0	2	1						25	11	tasks1.c	int	
__polyqasm___stdtsube.erno	0	0	2						383	15	__polyqasm___.c	int	

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 tasks1.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.

The information about global variables and read/write access operations obtained from the **Variable Access** pane is called the data dictionary. For more information on the data dictionary, see “Dataflow Verification” on page 10-120.

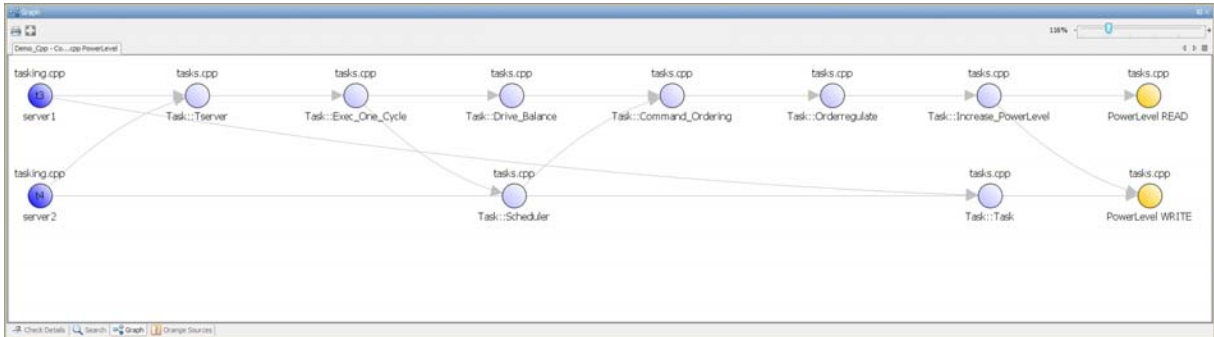
You can also perform the following actions from the **Variable Access** pane.

•

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 . For more information, see “View Access Graph for Global Variables” on page 10-49.

Here is an example of an access graph:



•

View Structured Variables



For structured variables, view the individual fields from the **Variable Access** pane. For example, for the structure, SHR4, the pane displays the fields, SHR4.A and SHR4.B, and the functions performing read/write access on them.

The screenshot shows the Variable Access pane for a program named 'tasks1.c'. The pane displays a list of variables and their access statistics. The variables listed are:

- `tasks1_init_globals`: 2 reads, 3 writes, accessed by tasks1_prec2 (lines 12, 13, 14, 15).
- `tasks1_prec2`: 1 read, 1 write, accessed by tasks1_prec2 (line 115).
- `tasks1_orderregulate`: 1 read, 1 write, accessed by tasks1_orderregulate() (lines 22, 45, 46).
- `tasks1_server1`: 12 reads, accessed by tasks1_server1 (lines 13, 14).
- `tasks1_server2`: 13 reads, accessed by tasks1_server2 (lines 13, 14).
- `tasks1_tregulate`: 15 reads, accessed by tasks1_tregulate (line 15).

View Access Through Pointers



View access operations on global variables performed indirectly through pointers.

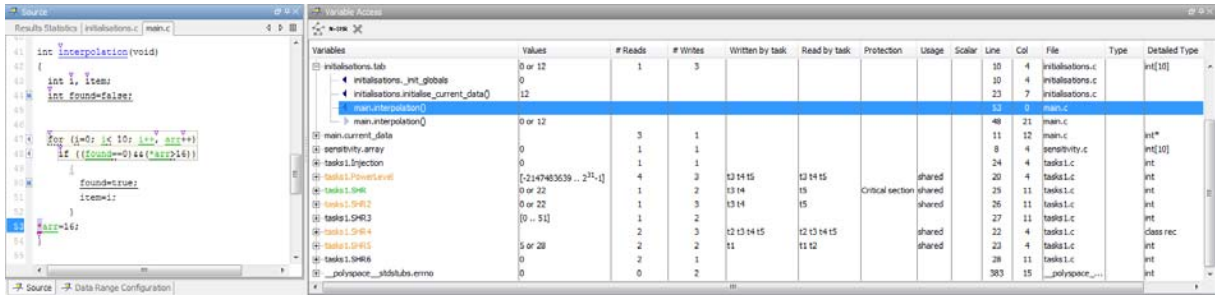
If a read/write access on a variable is performed through pointers, then the access is marked by  (read) or  (write).

For instance, in the file, `initialisations.c`, the variable, `arr`, is declared as a pointer to the array, `tab`.

The screenshot shows the Variable Access pane for a program named 'initialisations.c'. The pane displays a list of variables and their access statistics. The variables listed are:


- `initialisations_init_globals`: 1 read, 3 writes, accessed by initialisations_init_globals (line 10).
- `initialisations_initiale_current_data`: 10 reads, accessed by initialisations_initiale_current_data (line 23).
- `main_interpolation()`: 53 reads, accessed by main_interpolation() (lines 53, 54).
- `main_interpolation()`: 48 writes, accessed by main_interpolation() (line 21).

In the file `main.c`, `tab` is both read and written in the function, `interpolation()`, through the pointer variable, `arr`. This operation is shown in the **Variable Access** pane by the  and  icons respectively.




-

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 dead 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`.

Red Checks

Red checks indicate code that always causes a run-time error.

Run-time errors highlighted by Polyspace Code Prover verification are determined with reference to the language standard. Though some of the errors can be acceptable for a particular environment, they are unacceptable according to the language standard.

Consider an overflow on a type restricted from -128 to 127. The type cannot store the result of the computation $127+1=128$. However, depending on the environment a “wrap around” might be performed to give a result of -128. This result is mathematically incorrect, and could have serious consequences if, for example, the computation represents the altitude of a plane.

By default, Polyspace verification does not make assumptions about the way you use a variable. A deviation from the recommendations of the language standard is treated as a red error. Most of the errors you find are easy to fix once the software identifies them. Polyspace verification identifies errors regardless of their consequence, or how difficult they may be to fix.

Polyspace verification identifies two kinds of red checks:

- Red errors which are compiler-dependant in a specific way. A Polyspace option may be used to allow compiler specific behavior .

Examples in C include options to deal with constant overflows, shift operation on negative values, and so on.

- You must fix all other red errors. They are code defects.

Gray Checks

In this section...
“Gray Checks” on page 10-103
“Common Causes for Gray Checks” on page 10-103

Gray Checks

Gray checks denote unreachable sections of code. Unreachable code can arise in the following situations:

- Unreachable code resulting from bugs in the source code
- Unreachable code resulting from a particular configuration
- Defensive code that is never reached
- Libraries that are not used to their full extent in a particular context

Common Causes for Gray Checks

- A lack of parenthesis and operand priorities in the testing clause changes the meaning significantly.

Consider a line of code such as:

```
IF NOT a AND b OR c AND d
```

For this line of code, misplaced parentheses can severely influence how the line behaves. For instance, the following placement of parentheses can lead to significantly different test conditions:

```
IF NOT (a AND b OR c AND d)
```

```
IF (NOT (a) AND b) OR (c AND d))
```

```
IF NOT (a AND (b OR c) AND d)
```

- The test variable takes values that never satisfy the condition tested by an `if` statement.
- The wrong variable is tested in the `if` statement.
- The test variable should be local to a file but is instead local to a function.

- The data type of the test variable leads to a comparison that is always false.

Orange Checks

Orange checks indicates that the code cannot be proved to either have or not have a run-time error.

The number of orange checks you need to review is determined by several factors, including:

- The stage of the development process
- Your quality goals

You can also take steps to reduce the number of orange checks. For more information, see “Orange Check Management”.

Orange Check Identified as Potential Errors

The software identifies a subset of orange checks that are most likely run-time errors. If you choose the review methodology **First checks to review**, you can view this subset. These orange checks are related to path and bounded input values. For more information, see:

- “Path” on page 10-106
- “Bounded Input Values” on page 10-107
- “Unbounded Input Values” on page 10-107

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”.
- Global and volatile variables.
- Data returned by a stubbed function. The data can be the value returned by the function or a function parameter modified through a pointer.

Path

The following example shows a path-related orange check that might be identified as a potential run-time error.

Consider the following code.

```
void path(int x) {
    int result;
    result = 1 / (x - 10);
    // Orange Division by Zero
}

void main() {
    path(1);
    path(10);
}
```

The software identifies the orange ZDV check as a potential error. The **Check 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 **Check 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 **Check 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 **Check Details** pane provides information about the potential error:

```
Warning: array index may be outside bounds: [0..9]  
This check may be an issue related to unbounded input values  
If appropriate, applying DRS to extern variable val may remove this orange.  
array size: 10  
array index value: [-231 .. 231-1]
```


Color Sequence of Checks

The following examples show how the checks obtained in a verification can depend on each other.

- 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 $x = x + 1$;

- The following example shows how a green check can propagate out of an orange check.

```
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]`.
- The following example shows why a check should be reviewed in the context of the code.

Consider an orange Non-initialized local variable on `x` in the following statement:

```
if (x > 101);
```

You might conclude that the verification continues after this statement because the check is orange. However, consider the same statement in the context of the code:

```
extern int read_an_input(void);

void main(void)
{
    int x;
    if (read_an_input()) x = 100;
    if (x > 101)
        //Orange Non-initialised local variable
        {x++; }
        // Gray code
}
```

The correct interpretation of this verification result is that if `x` is initialized, the only possible value for it is 100. Therefore, `x` can never be both initialized and greater than 101, so the rest of the code is gray. This conclusion is different from what you expect considering the line in isolation.

- The following example shows how a red error can hide 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 $x=1/\text{old}_x$; 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 $x=1/\text{old}_x$; in `file1.c` because the software assumes old_x to be 0 as well.
- The following example shows how skipping intermediate code while tracing the cause of a check might lead to erroneous conclusions.

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; // [array index within bounds]
  y[x-1] = (1 / x) + x ;
  if (x == 0)
    y[x] = 1; // gray code on this line
}
```

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 check on `y[x]=0;` means that subsequent lines deal with `x` in `[0,99]`.
- The orange 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.

Defects from Code Integration

When you integrate sections of code, the number of checks can change from when the sections were verified in isolation. The following examples show this behavior:

- A function receives two unbounded integers. When verifying the function in isolation, the software assumes that inputs are well-behaved. The software can check for the presence of an overflow only during integration.
- A function takes a structure as an input parameter. When verifying the function in isolation, the software assumes that the structure is well initialized. Consequentially, the software displays a green `Non-initialized local variable` check at the first read access to a field. During integration, this check can turn orange if a context does not initialize these fields.

If you have already performed an exhaustive review for the individual sections, during integration, review only checks that have turned from green to another color .

Defects in Unprotected Shared Data

Based on the list of entry points in a multi-task application, Polyspace verification identifies a list of shared data and provides some information about each entry:

- The data type.
- A list of read and write access to the data through functions and entry points.
- The type of any implemented protection against concurrent access.

You can specify entry points through the **Multitasking** tab on the **Configuration** pane in the Project Manager perspective. For information on command-line specification, see “Entry points”.

A shared data item is a global data item that is read from or written to by two or more tasks. You can view information on shared data on the **Variable Access** pane. For more information, see “Variable Access” on page 10-94.

A shared variable is protected from concurrent access when one task cannot access it while another task is in the process of doing so. A defect can arise from unprotected concurrent access on variables. To prevent defects arising from concurrent access, protect the variables by placing them in a critical section or temporally exclusive tasks. For more information, see “Critical section details” and “Temporally exclusive tasks”.

Defects Related to Pointers

In this section...
“Messages on Dereferences” on page 10-115
“Variables in Structures (C)” on page 10-117

For a check related to a pointer variable, on separate lines in the tooltip message, the software displays:

- The pointer name, data type of the variable, and size of the data type in bits.
- A comment that indicates whether the pointer is `null`, `is not null`, or `may be null`. See also “Messages on Dereferences” on page 10-115.
- The number of bytes that the pointer accesses, the offset position of the pointer in the allocated buffer, and the size of this buffer in bytes.
- A comment that indicates whether the pointer *may* point to dynamically allocated memory.
- The names of the variables at which the pointer may point. See also “Variables in Structures (C)” on page 10-117.

For a check related to a function pointer, the software displays:

- The pointer name.
- A comment that indicates whether the pointer is `null`, `is not null`, or `may be null`.
- The names of the functions that the pointer may point to, and a comment indicating whether the functions are well or badly typed (whether the number or types of arguments in a function call are compatible with the function definition).

Messages on Dereferences

Tooltip messages on dereferences give information about the expression that is dereferenced.

Consider the following code:

```
int *p = (int*) malloc ( sizeof(int) * 20 );
p[10] = 0;
```

In the verification results, the tooltip on “[” displays information about the expression that is dereferenced.

```
23
24     int *p = (int*) malloc ( sizeof(int) * 20 );
25     p[10] = 0;
26     }
27
28
29
```

dereference of expression (pointer to int 32, size: 32 bits):
 pointer is not null
 points to 4 bytes at offset 40 in allocated buffer of 80 bytes
 points to dynamically allocated memory

p[10] refers to the contents of address p + 10 * sizeof(int), so the tooltip message displays the following:

- The dereferenced pointer is at offset 40.
Explanation: p has offset 0, so p+10 has offset 10 * sizeof(int)=40.
- The dereferenced pointer is not null.
Explanation: p is null, but p+10 is not null (0+40 ≠ 0).

The software reports an orange dereference check (IDP) on p[10] because malloc may have put NULL into p. In that case, p + 10 * sizeof(int) is not null, but it is not properly allocated.

Variables in Structures (C)

The information that the software displays for structure variables depends on whether you specify the option `Enable pointer arithmetic across fields`.

Consider the following code:

```
Struct { int x; int y; int z; } s ;  
int *p = &s.y ;
```

If you do not specify the option (this is the default), then placing the cursor over `p` produces the following information:

```
accessing 4 bytes at offset 0 in buffer of 4 bytes
```

This information conforms with ANSI C, which

- Requires that `&s.y` points only at the field `y`
- Does not allow pointer arithmetic for access to other fields, for example, `z`

If you specify the option `-allow-ptr-arith-on-struct`, you are allowed to carry out pointer arithmetic using the addresses of structure fields. In this case, placing the cursor over `p` produces the following information:

```
accessing 4 bytes at offset 4 in buffer of 12 bytes
```

Global Variables

Initializing Global Variables

If your application defines global variables, then the software uses the dummy function `_init_globals()` to initialize the global variables. The `_init_globals()` function is the first function called in the main function.

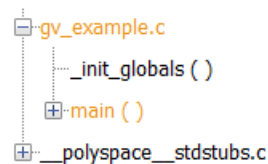
Consider the following code in the application `gv_example.c`.

```
extern int func(int); /* External function */

/* Global variables initialized in _init_globals() */
/* before the execution of main() procedure      */
int garray[3] = {1, 2, 3};
/* Initialized: written in __init_globals() */
int gvar = 12;
/* Initialized: written in __init_globals() */

int main(void) {
    int i, lvar = 0;
    for (i = 0; i < 3; i++)
        lvar += func(garray[i] + gvar);
    return lvar;
}
```

Verification produces the following procedural entities:



In the Variables view, `gv_example._init_globals` represents the first write access operation on a global variable, for example, `garray`. The corresponding value in the **Values** column represents the value of the global variable after initialization.

gv_example.garray	[1 .. 3]	1	1
└─ gv_example._init_globals	[1 .. 3]		
└─ gv_example.main	[1 .. 3]		
gv_example.gvar	12	1	1
└─ gv_example._init_globals	12		
└─ gv_example.main	12		
__polyspace__stdstubs.errno		0	0

Using Global Variables

For global variables, it is not always apparent which global variables are produced or used by a given file. Excessive use of global variables can lead to design problems, such as:

- File APIs (or functions accessible from outside the file) without procedure parameters.
- The requirement for a formal list of variables which are produced and used, and the theoretical ranges they can take as input and output values.

Dataflow Verification

You can verify dataflow in Polyspace for certification purposes. Dataflow verification is a typical requirement in the avionic, aerospace, or transport markets.

Verify data flow for functions and global variables through the following Polyspace results:

- Call tree (also known as call graph) for functions (and tasks). The call tree shows the calling relationship between functions (and tasks). For more information, see “View Call Tree for Functions” on page 10-44.

You can view the call tree in two ways:

- Through the **Call Hierarchy** pane. On this pane, you can view the branch of the call tree containing a given function. You can also navigate the entire call tree from this pane. For more information, see “Call Hierarchy” on page 10-91.
- In text format. Open the file, *projectname_Call_Tree.txt*, in the folder, *Polyspace-doc*, in your results folder.
- Data dictionary for global variables. The data dictionary lists global variables with their read/write access operations.

You can view the data dictionary in two ways:

- Through the **Variable Access** pane. On this pane, you can view global variables and their attributes. For more information, see “Variable Access” on page 10-94. You can also access a graphical representation of the call sequence for global variables using this pane. For more information, see “View Access Graph for Global Variables” on page 10-49.
- In text format. Open the file, *projectname_Variable_View.txt*, in the folder, *Polyspace-doc*, in your results folder.

Results Folder

The `Result_n` folder contains the following files:

- `Polyspace_release_project_name_date-time.log` — A log file associated with each verification, for example, `Polyspace_R2013b_example_project_05_17_2013-12h01.log`.
- `project_name.pscp` — An ASCII file containing the location of the most recent results and log. The software uses this file to open results in the Results Manager.
- `drs-template.xml` — A file containing the data range specifications that is generated by Polyspace Code Prover during the compile phase.
- `coding_rules_std_rules.txt` — A template of coding rules. For example, `misra_rules.txt` is a template of MISRA rules, generated when you specify the `-misra2 all-rules` option.
- `options` — The list of options used for the most recent verification.
- `pst_user_stubs.c` — The list of functions and procedures stubbed by Polyspace Code Prover during the compile phase.
- `source_list.txt` — A list of sources verified by the latest verification.

In addition, the `Result_n` folder contains the following subfolders:

In this section...
“ALL Subfolder” on page 10-121
“Polyspace-Doc Subfolder” on page 10-122
“Polyspace-Instrumented Subfolder” on page 10-123

ALL Subfolder

The ALL subfolder contains internal information that is used by Polyspace Code Prover to show sources and checks.

- `SRC\MACROS\ci.zip` — A zip file containing expanded source files with a `.ci` suffix.

- `_deadproc.txt` — A text file of unreachable procedures.
- `SRC*. [c or h]` — Source code file required for the verification. The file contains user source code and code generated by Polyspace Code Prover.

Polyspace-Doc Subfolder

The Polyspace-Doc subfolder contains the following files:

- `Code_Metrics.xml` — A list of metrics from the most recent verification.
- `CODING_RULES_STD-report.xml` and `CODING_RULES_STD-summary-report.xml` — Lists coding rules violated during the most recent verification. For example:
 - If you specify the option **Check MISRA C:2004 rules**, the software generates `MISRA-report.xml` and `MISRA-summary-report.xml`. These files list the violated MISRA C rules.
 - If you specify the option **Check custom rules**, the software generates `Custom-rules-report.xml` and `Custom-rules-summary-report.xml`. These files list the violated custom rules.
- `Project_name_Call_Tree.txt` — Call tree starting from entry point functions. Each level in the tree hierarchy is denoted by `|`. For example, a function call two levels away from an entry point function is denoted as:

```
| | - > file_name.function_name
```

Each row in this file contains a function name and the file, line, and column of the:

- Function call
 - Function definition
- `Project_name_Variable_View.txt` — Data dictionary of global variables. For each variable, the rows below the variable name contain:
 - Variable reads denoted by `>` and writes denoted by `<`.
 - Call tree level of reading or writing function denoted using `|` in the same way as in `Project_name_Call_Tree.txt`

- Additional information available on the **Variable Access** pane. See “Variable Access” on page 10-94.
- **Polyspace_Macros** — This .xls file contains a **Generate Spreadsheet** macro. The macro collects the information contained in *Project_name_Call_Tree.txt* and *Project_name_Variable_View.txt*. The macro then displays them in a spreadsheet. For the macro to function, both .txt files must be in the same folder as the .xls file.

Polyspace-Instrumented Subfolder

The Polyspace-Instrumented folder and files are generated only when you use the Automatic Orange Tester (AOT), that is, when you specify the `-automatic-orange-tester` option.

- `_testgen.tgf` — A configuration file for the AOT. The software uses this file to open the AOT user interface.
- `reachedchecks.txt` — Contains a list of checks. Some these checks may have been tested by the AOT.
- `stderr` and `stdout` — Contains the output of `stderr` and `stdout` if they are used in the code. The output is generated only when the AOT is used.

Reusing Review Comments

After you have reviewed verification results, you can reuse your review comments for subsequent verifications. By reusing your review comments, you can:

- Avoid reviewing the same check twice.
- Compare verification results over time.

You can directly import review comments from another set of results into the current results. However, it is possible that your review comments do not apply 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.

Related Examples


- “Import Review Comments from Previous Verifications” on page 10-125
- “View Checks and Comments Report” on page 10-127

Import Review Comments from Previous Verifications

In this section...

“Import Comments from Previous Verifications” on page 10-125
“Automatically Import Comments from Last Verification” on page 10-125
“Automatically Import Comments During Command-Line Verification” on page 10-126

After you have reviewed verification results, you can reuse your review comments for subsequent verifications.

After you import checks and comments, clicking the  icon skips justified checks. Therefore, you do not have to review checks twice.

Import Comments from Previous Verifications

- 1 Open your verification results in the Results Manager perspective.
- 2 Select **Review > Import > 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, and the Import checks and comments report opens. For more information, see “View Checks and Comments Report” on page 10-127.

Automatically Import Comments from Last Verification

- 1 Select **Options > Preferences**, which opens the Polyspace Preferences dialog box.
- 2 Select the **Project and result folder** tab.
- 3 Under **Import Comments**, select the **Automatically import comments from last verification** check box.

4 Click **OK**.

After you set this preference, for every run, the software imports review comments from the last run.

Automatically Import Comments During Command-Line Verification

To automatically import comments from a specific verification, use the option `-import-comments`. For example:

```
polyspace-codeprover-nodesktop -version 1.3 -import-comments C:\PolyspaceResults\1.2
```

See Also `-import-comments`

Related Examples

- “View Checks and Comments Report” on page 10-127

Concepts

- “Reusing Review Comments” on page 10-124

View Checks and Comments Report

After you have reviewed verification results, you can reuse your review comments for subsequent verifications. However, it is possible that your review comments do not apply 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 **Review > Import > Open Import Report**.

The Import Checks and Comments Report opens, highlighting differences in the two results.

File	Function	Line	C#I	Check	Import details	Justified	Classification	Status	Comment
example.c	example.c	11	201%	Check color has changed from Green to Orange	Check color has changed from Green to Orange	[X]	Just a defect	For action planned	This might overflow

2 Review the differences between the two results.

- If the check color changes, Polyspace populates the **Comment** field but not the fields **Classification**, **Status** or **Justified**.
- 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.

Related Examples

- “Import Review Comments from Previous Verifications” on page 10-125

Concepts

- “Reusing Review Comments” on page 10-124

Generate Report After Verification Automatically

You can specify that Polyspace software automatically generate reports for each verification using an option in the Project Manager perspective.

Note You cannot generate reports of software quality objectives automatically.

To automatically generate reports for each verification:

- 1** In the Project Manager perspective, open your project.
- 2** Select the **Configuration > Reporting** pane.
- 3** Select the **Generate report** check box.
- 4** From the **Report template** drop-down list, select a template.
- 5** From the **Output format** drop-down list, select a format for the report.
- 6** Save your project.

Generate Report After Verification Manually

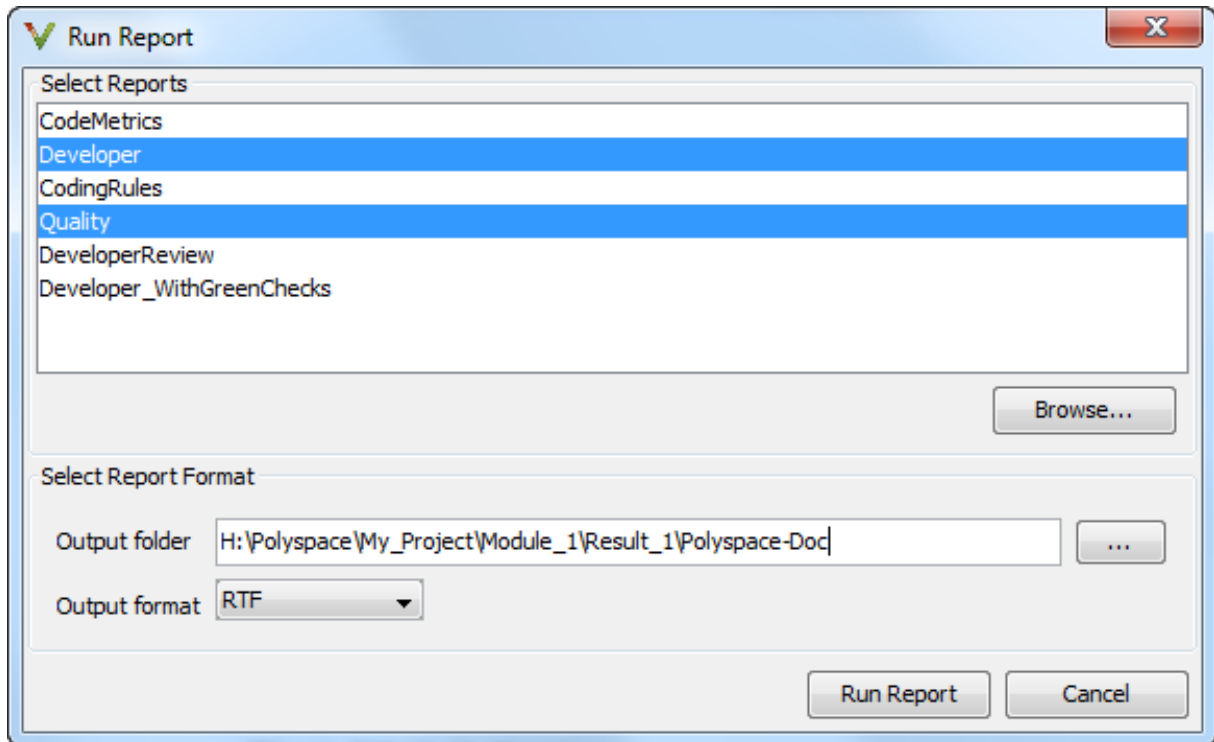
You can generate reports for verification results using the Polyspace Report Generator.

To generate a verification report:

- 1 In the Results Manager perspective, open your verification results.
- 2 Select **Run > Run Report > Run Report...**

The Run Report dialog box opens.

- 3 In the **Select Reports** section, select the types of reports that you want to generate. For example, you can select **Developer** and **Quality**.



- 4** Select the Output folder in which to save the reports.
- 5** Select the Output format for the reports.
- 6** Click **Run Report**.

The software creates the specified reports.

Generate Report from Command Line

You can also run the Report Generator, with options, from the command line, for example:

```
Matlab_Install\polyspace\bin\polyspace-report-generator -template  
path -format type -results-dir folder_paths
```

For information about the available options, see the following sections.

-template *path*

Specify the *path* to a valid Report Generator template file, for example:

```
Matlab_Install\polyspace\toolbox\psrptgen\templates\Developer.rpt
```

Other supplied templates are CodingRules.rpt, Developer_WithGreenChecks.rpt, DeveloperReview.rpt, and Quality.rpt.

-format *type*

Specify the format *type* of the report. Use HTML, PDF, RTF, WORD, or XML. The default is RTF.

-help or -h

Displays help information.

-output-name *filename*

Specify the *filename* for the report generated.

-results-dir *folder_paths*

Specify the paths to the folders that contain your verification results.

You can generate a single report for multiple verifications by specifying *folder_paths* as follows:

```
"folder1, folder2, folder3,..., folderN"
```


where *folder1*, *folder2*, ... are the file paths to the folders that contain the results of your verifications (normal or unit-by-unit). For example,

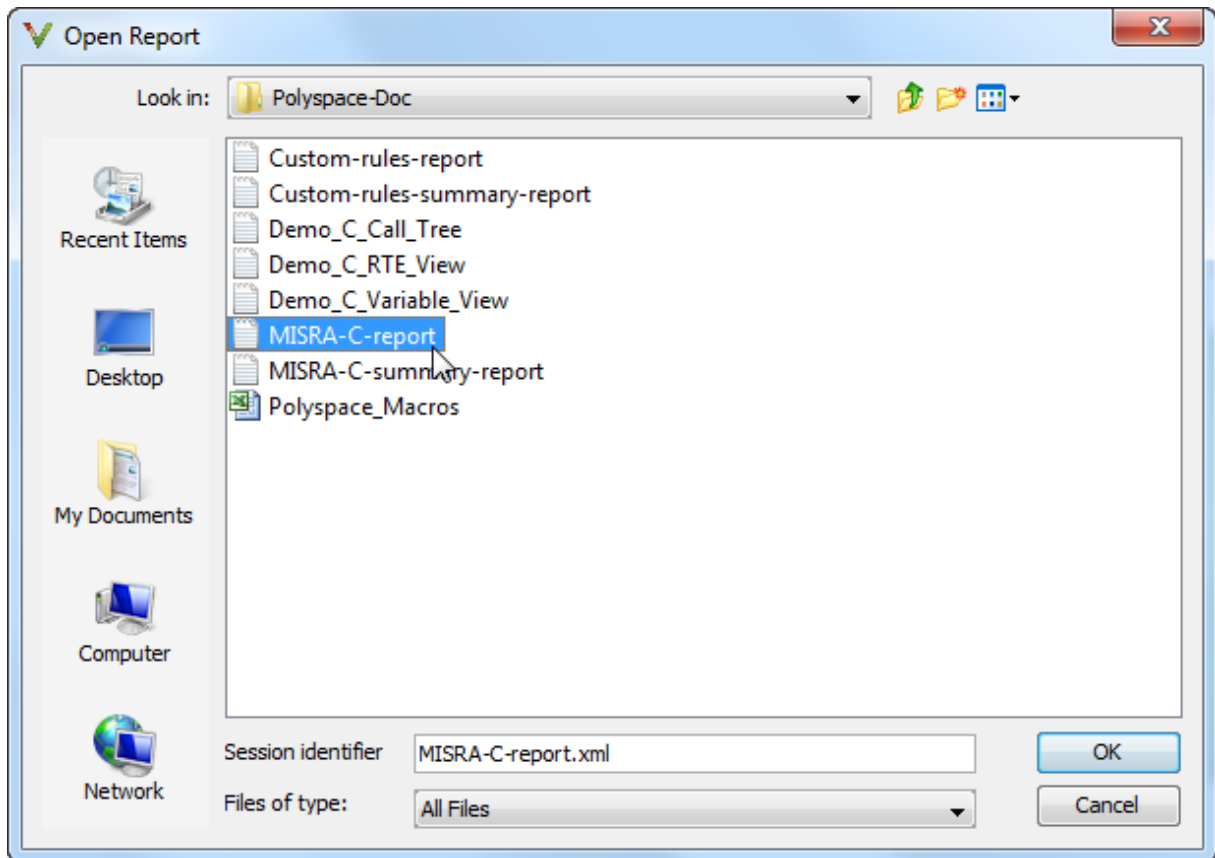
```
"C:\Results1,C:\Recent\results,C:\Old"
```

If you do not specify a folder path, the software uses verification results from the current folder.

Open Verification Report

To view a verification results report:

- 1 From the Results Manager toolbar, select **Run > Run Report > Open Report**, which opens the Open Report dialog box.
- 2 Navigate to the folder that contains your report. Then select the report.



- 3 Click **OK**.

Customize Verification Report

If you have MATLAB Report Generator™ software installed on your system, you can customize the Polyspace report templates or create your own reports. You can then generate these custom reports using the Polyspace Report Generator.

Before you can customize Polyspace reports, you must configure the MATLAB Report Generator software to access the following folders:

- **Custom components** –
Matlab_Install/polyspace/toolbox/psrptgen/psrptgen
- **Report templates** –
Matlab_Install/polyspace/toolbox/psrptgen/templates

To customize a Polyspace report:

1 Open MATLAB.

2 Add the Polyspace reports custom components folder to the MATLAB search path, using the following command:

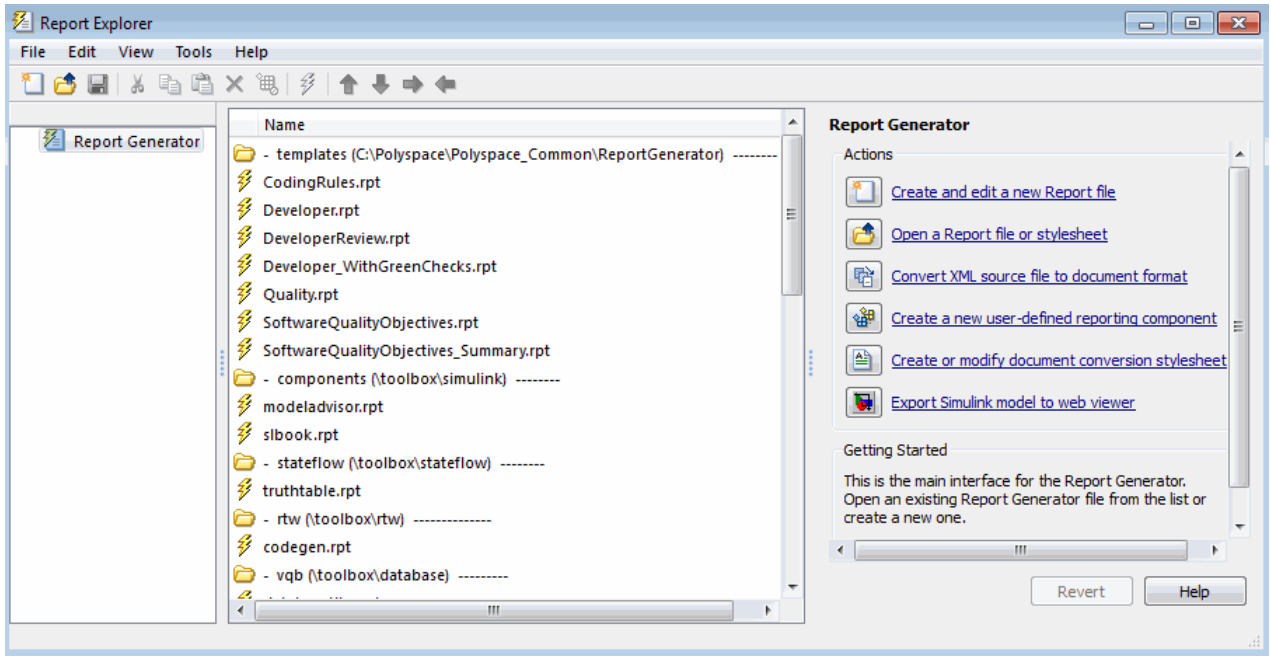
```
addpath('Matlab_Install/polyspace/toolbox/psrptgen/psrptgen')
```

3 Set the current folder in MATLAB to the Polyspace reports template folder, using the following command:

```
cd('Matlab_Install/polyspace/toolbox/psrptgen/templates')
```

4 Open the Report Explorer using the following command:

```
report
```



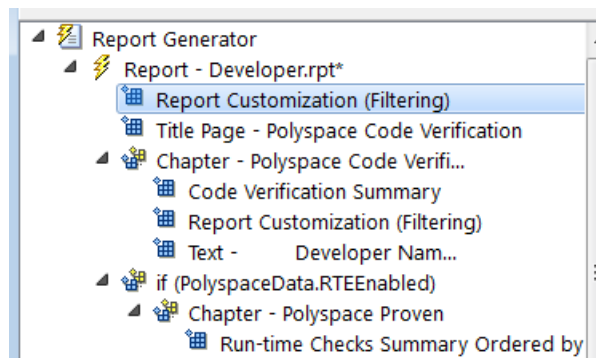
- 5 Open the template that you want to customize. For example, double-click `Developer.rpt`. You can add, delete, or alter components of this template.
- 6 To save the modified template, select **File > Save As**. The Save Report File as dialog box opens. If you are not in the `Matlab_Install/polyspace/toolbox/psrptgen/templates` folder, navigate to this folder.
- 7 In the **File name** field, specify the name of the modified template, for example `Developer_modified.rpt`. Then, click **Save**.

When you next generate a report, `Developer_modified.rpt` will be available as an option in the Run Report dialog box.

Note To produce custom reports with the Polyspace Report Generator, you must save the report template in: `Polyspace_Install/polyspace/toolbox/psrptgen/templates`.

You can use the **Report Customization (Filtering)** component to apply filters to your report. This component provides a number of filters, for example, for code metrics, coding rules, and run-time checks.

- 1 From the **Name** list, under **Polyspace**, select the **Report Customization (Filtering)** component. Then drag this component to the required point in the template. For example, if you want to apply global filters to the report, place the component above the **Title Page** component.



- 2 Select the **Report Customization (filtering)** component.
- 3 On the right of the dialog box, specify your filters. For example, you might want your report to contain only specific run-time checks. In addition, you might want to see these checks for only certain functions. Under **Advanced Filters**:
 - In the **Check types to include** field, enter the run-time checks, for example, ASRT and OBAI.
 - In the **Function names to include** field, enter the functions, for example, function1 and functionX.

Advanced Filters

Justification status: All

For each filter give a list of regular expressions matching the content to include in the report. Example to include only ASRT and OBAI checks in the report in the check types filter enter:
 ASRT
 OBAI

Files to include: .*	Check types to include: ASRT OBAI
Function names to include: function1 functionX	Classification types to include: .*
Status types to include: *	Comments to include: *

Revert Help

Note You can also exclude items. For example, to exclude all comments containing the string GEN, in the **Comments to include** field, enter the expression `^(?:(!GEN).)*$`. For more information about regular expressions, refer to *Regular Expressions* in the MATLAB documentation.

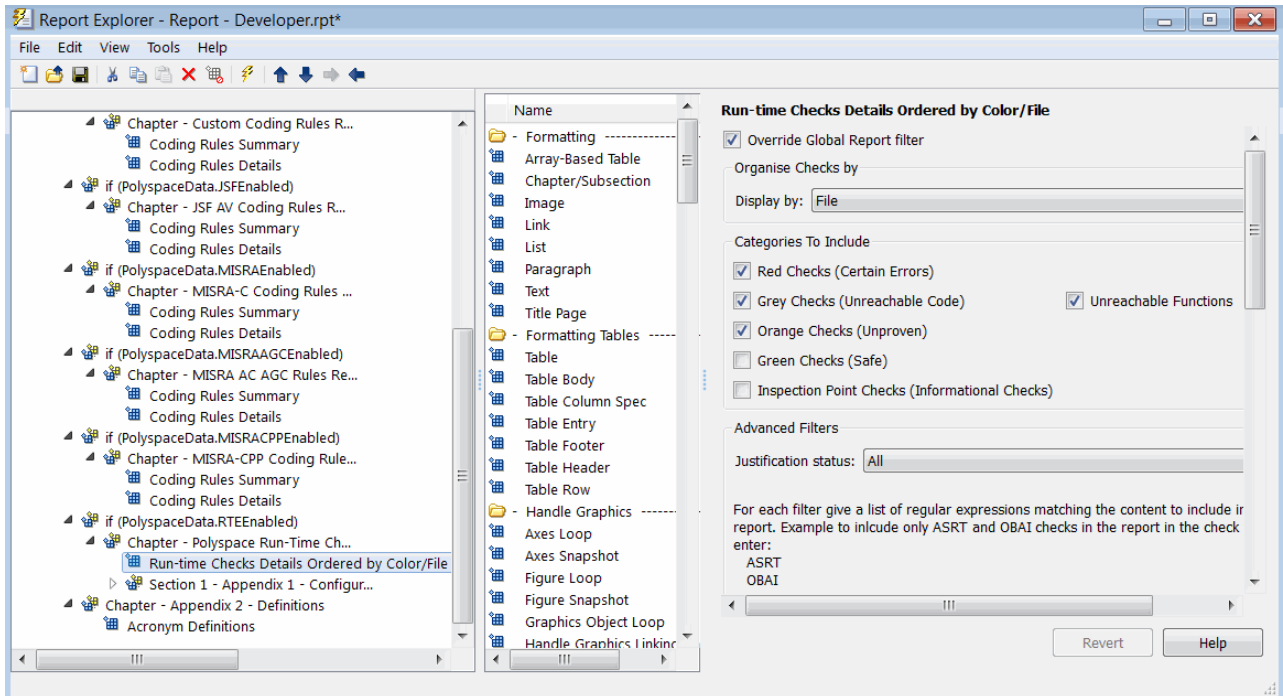
4 Save the modified template.

For the **Run-time Check Details Ordered by Color/File** component, you can override the global filters and specify different filters for this component.

For example, you might want to have a report chapter that contains NIV checks filtered by the global filter.

To override global filters:

- 1 Select the **Run-time Check Details Ordered by Color/File** component.



- 2 On the right of the dialog box, select the **Override Global Report filter** check box.

- 3 Specify your filters for this component. For example, in the **Check types to include** field, enter NIV.

- 4 Save the template.

For more information, refer to the MATLAB Report Generator documentation.

Managing Orange Checks

- “What is an Orange Check?” on page 11-3
- “Sources of Orange Checks” on page 11-7
- “Do I Have Too Many Orange Checks?” on page 11-12
- “Manage Orange Checks” on page 11-13
- “Overview: Reducing Orange Checks” on page 11-14
- “Apply Coding Rules to Reduce Orange Checks” on page 11-15
- “Use Generated Code” on page 11-16
- “Improve Verification Precision” on page 11-17
- “Specify Multitasking Behavior” on page 11-21
- “Effects of Application Code Size” on page 11-22
- “Overview: Reviewing Orange Checks” on page 11-23
- “Define Your Review Methodology” on page 11-24
- “Perform Selective Review” on page 11-25
- “View Sources of Orange Checks” on page 11-28
- “Prioritize Orange Check Review” on page 11-30
- “Refine Data Range Specifications” on page 11-33
- “Perform Exhaustive Review” on page 11-37
- “Automatic Orange Tester Overview” on page 11-40
- “How the Automatic Orange Tester Works” on page 11-42
- “Select the Automatic Orange Tester” on page 11-44
- “Start the Automatic Orange Tester Manually” on page 11-45

- “Review Test Results After Manual Run” on page 11-48
- “Refine Data Ranges with Automatic Orange Tester” on page 11-50
- “Save and Reuse Your Configuration” on page 11-53
- “Export Data Ranges for Polyspace Verification” on page 11-54
- “Polyspace-Instrumented Folder” on page 11-55
- “Technical Limitations” on page 11-56

What is an Orange Check?

Orange checks indicate *unproven code*, which means the software cannot prove whether the code leads to a run-time error or not.

Polyspace verification attempts to prove the absence or existence of run-time errors. Therefore, the software considers all code unproven before a verification. During a verification, the software attempts to prove that code is:

- Without run-time errors (green)
- Certain to fail (red)
- Unreachable (gray)

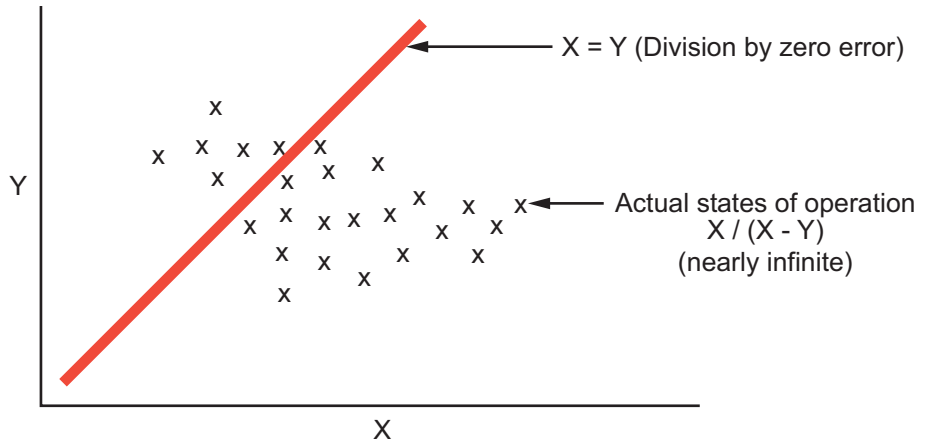
Code that is not assigned one of these categories (colors) stays unproven (orange).

Code often remains unproven in situations where some execution paths fail while others succeed. For example, in the instruction:

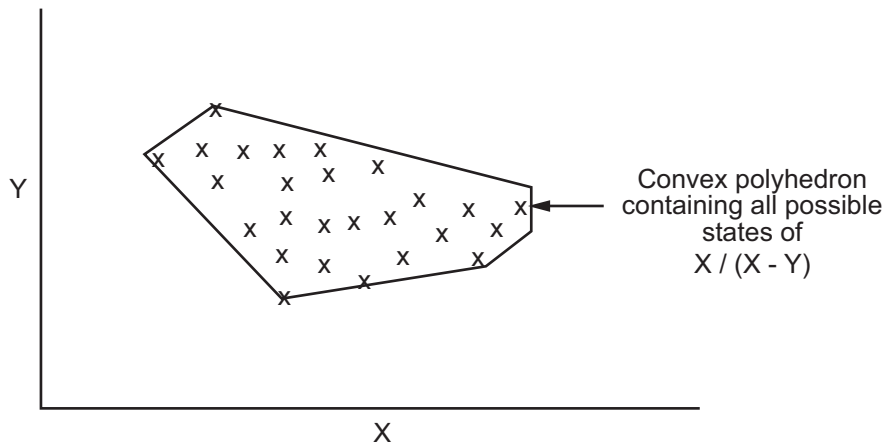
```
X = 1 / (X - Y);
```

the presence or absence of a Division by Zero error depends on the values of X and Y .

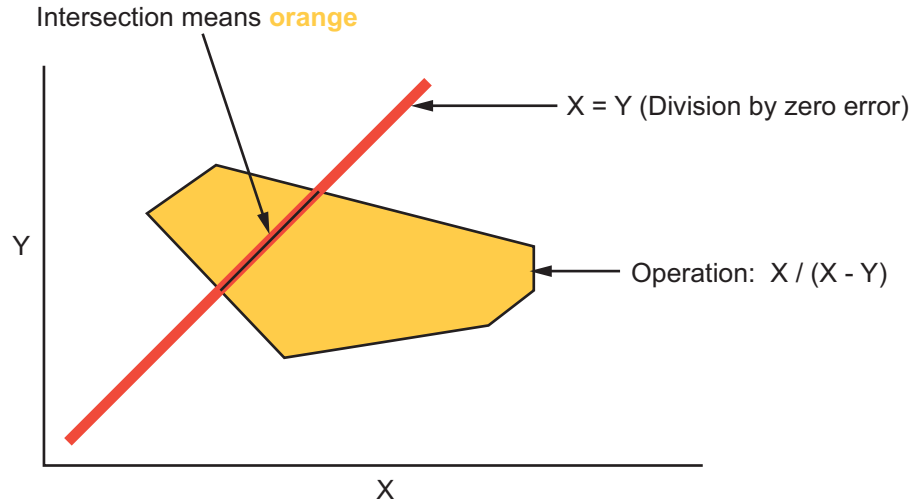
However, there are an almost infinite number of possible values. Creating test cases for all possible values is not practical.



Although it is not possible to test every value for each variable, the target computer and programming language provide limits on the possible values of the variables. Polyspace verification uses these limits to compute a *cloud of points* (upper-bounded convex polyhedron) that contains all possible states for the variables.

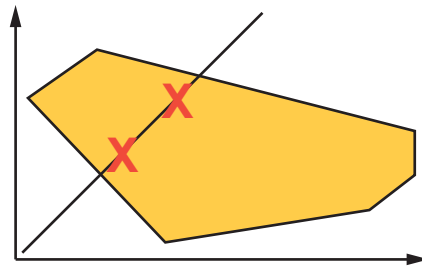


Polyspace verification then compares the data set represented by this polyhedron to possible values leading to an error. If the two data sets intersect, the check is orange.

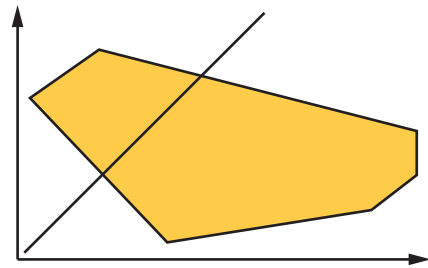


Graphical Representation of an Orange Check

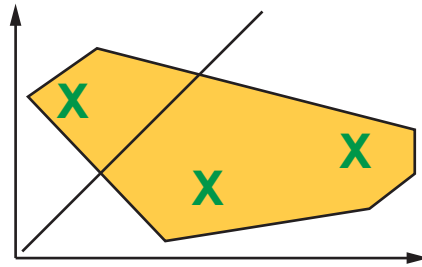
A true orange check represents a situation where some paths fail while others succeed. However, because the data set used for verification is a superset of actual run-time values, an orange check can represent a check of another color, as shown below.



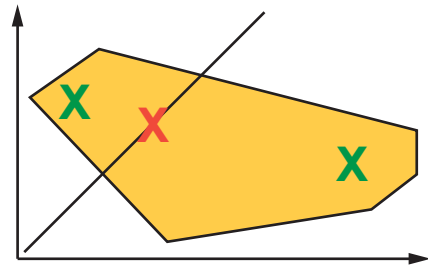
Red approximated by orange



Gray approximated by orange



Green approximated by orange



Any other situation (true orange)

Polyspace verification reports an orange check when the two data sets intersect. Therefore, it is possible that some of the orange checks indicate run-time errors while others do not.

Sources of Orange Checks

Orange checks can be separated into two categories:

In this section...
“Orange Checks from Code” on page 11-7
“Orange Checks from Verification Limitations” on page 11-9

Orange Checks from Code

Most orange checks are caused by the code. These checks can represent real bugs or execution paths that are not relevant for your application.

An orange check can occur when the verification finds an error for certain:

- “Data Values” on page 11-7
- “Function Sequences” on page 11-8

Data Values

An orange check can reveal code that will fail for certain data values.

For example, consider the function `Recursion()`:

```
int Recursion(int i)
{
    if (i>10)
        return(1);
    else
        return(i/Recursion(i+1));
}
```

If the initial value passed to `Recursion()` is negative, then the recursive loop will at some point attempt a division by zero. Therefore, the division operation causes an orange `Division by Zero` check.

When an orange check occurs for certain data values, you can usually identify the cause quickly. The range information provided in the Results Manager

perspective can help you identify whether the orange check represents a bug that should be fixed. See “Use Range Information in Results Manager” on page 10-55.

If the orange check represents a situation that cannot occur (say, if the initial value in the above example cannot be negative), you can do one of the following:

- Annotate the code to justify the check. For more information, see “Comment Code for Known Defects” on page 7-54.
- Modify the code to help Polyspace determine actual run-time values of variables. In the above example, rewrite the code as

```
int Recursion(int i)
{
    if(i>0)
        {if (i>10)
            return(1);
            else
                return(i/Recursion(i+1));
        }
    else
        return(1);
}
```

- Constrain the data ranges used in the verification using DRS. For more information, see “Data Range Specifications (DRS)” on page 6-55.

Function Sequences

An orange check can occur if the verification finds an error for certain function sequences in the automatically generated main.

For example, consider a variable X , and two functions, $F1$ and $F2$:

- $F1$ makes the assignment $X = 12$.
- $F2$ divides a local variable by X .
- The automatically generated main ($F0$) initializes X to 0.

- The generated main then randomly calls the functions in the following sequence:

```
If (random)
  Call F1
  Call F2
Else
  Call F2
  Call F1
```

A `Division by Zero` error occurs when F1 is called after F2. Therefore, the division operation in F2 causes an orange check. The verification cannot determine if an error will occur unless you define the call sequence.

Many inconclusive orange checks take some time to investigate, due to the complexity of the code. When an orange check is caused by function sequence, you have several options:

- Provide manual stubs for some functions. For more information, see “Constrain Data with Stubbing” on page 7-14.
- Use `-main-generator` options to describe the function call sequence, or to specify a function called before the main. For more information, see “Specify Call Sequence” on page 6-36.
- Write defensive code to prevent potential problems.
- Annotate the code to justify the check. For more information, see “Comment Code for Known Defects” on page 7-54.

Orange Checks from Verification Limitations

Some orange checks are caused by limitations of the verification process itself.

In these cases, the orange check is a false positive, because the code does not contain an actual bug. However, these types of orange checks can suggest design issues with the code.

Orange checks from verification limitations can be caused by:

- “Code Complexity” on page 11-10
- “Imprecise Approximation” on page 11-11

Code Complexity

An orange check can occur when the code structure is too complex to be verified by Polyspace software.

When a code is extremely complex, the verification cannot conclude whether a problem exists. The software then reports an orange check in the results.

For example, consider the following sequence of operations on a variable *Computed_Speed*:

- *Computed_Speed* is first copied into a signed integer (between -2^{31} and $2^{31}-1$).
- *Computed_Speed* is then copied into an unsigned integer (between 0 and $2^{31}-1$).
- *Computed_Speed* is next copied into a signed integer again.
- Finally, *Computed_Speed* is added to another variable.

Polyspace verification reports an orange `Overflow` on the addition.

Although this type of orange check does not indicate a real bug, it does suggest that the code might be poorly designed.

Orange checks caused by code complexity often take some time to investigate, but generally share certain characteristics:

- Code complexity problems usually result in multiple orange checks in the same module.
- The multiple checks in the same module are often related. Further analysis often reveals that the checks pertain to a single variable or function.

Depending on the criticality of the function and the required speed of execution, you can rewrite the code to remove risk of failure.

To limit the number of orange checks caused by code complexity, you can:

- Enforce coding rules during development. For more information, see “Coding Rules Compliance”.

- Perform unit-by-unit verification to verify smaller sections of code. For more information, see “Run unit by unit verification”.

Imprecise Approximation

An orange check can be caused by imprecise approximation of the data set used for verification.

Static verification uses approximations of software operations and data. For certain code constructions, these approximations can lead to a loss of precision. This loss of precision can cause orange checks in the verification results.

For example, consider a variable X :

- Before the function call, X is defined as having the following values: -5, -3, 8, or any value in range [10 . . .20]. This means that 0 has been excluded from the set of possible values for X .
- However, due to optimization (especially at low precision levels), the verification approximates X in the range [-5 . . .20], instead of the previous set of values.
- The instruction $y = 1/x$ causes an orange `Division by Zero` error.

Polyspace verification is unable to prove the absence of a run-time error in this case.

In cases of imprecise approximations, you can resolve orange checks by increasing the precision level. If this does not remove the orange check, review the code to determine the problem. To limit the number of orange checks caused by basic imprecision, avoid code constructions that cause imprecision. For more information, see “Approximations Made by Polyspace Verification”.

Do I Have Too Many Orange Checks?

If the goal of code verification is to prove the absence of run-time errors, you might be concerned by the number of orange checks in your results.

However, the presence of multiple orange checks need not be a cause for concern. There is no ideal minimum number of orange checks for all applications. The minimum number that you want depends on several factors:

- **Development Stage** – When verifying the first version of a software component, focus exclusively on resolving red checks. As development progresses, start considering the orange checks more and more.
- **Application Requirements** – Sometimes, to write provable code, you can compromise with properties such as code size, speed, and portability. Depending on the requirements of your application, you might optimize one or more of these properties at the expense of more orange checks.
- **Quality Goals** – Using Polyspace software, you can meet your quality goals. Therefore, before you verify code, you must define quality goals for your application. These goals should be based on the criticality of the application, as well as time and cost constraints. Based on your quality goals, you can choose to retain a specific minimum number of orange checks in your application.

Therefore, it is essential to understand how to manage the remaining orange checks. For more information, see “Manage Orange Checks” on page 11-13.

Manage Orange Checks

Polyspace verification by itself cannot produce quality code at the end of the development process. Verification helps you measure the quality of your code, identify issues, and achieve your quality goals. To do this, you must effectively integrate Polyspace verification into your development process.

To manage orange checks effectively, perform each of the following steps:

- 1** Define your quality goals. See “Analysis and Review Criteria” on page 2-4.
- 2** Set Polyspace analysis options to match your quality goals. See “Specify Options to Match Your Quality Goals” on page 4-11.
- 3** Define a process to reduce orange checks. See “Overview: Reducing Orange Checks” on page 11-14.
- 4** Apply the process to work with remaining orange checks.

Overview: Reducing Orange Checks

Actions that reduce orange checks and improve the quality of your code:

- “Apply Coding Rules to Reduce Orange Checks” on page 11-15.
- “Use Generated Code” on page 11-16.

Actions that reduce orange checks through increased verification precision:

- “Improve Verification Precision” on page 11-17.
- “Constrain Data with Stubbing” on page 7-14.
- “Specify Multitasking Behavior” on page 11-21.

Options that reduce orange checks but do not improve code quality or verification precision:

- Specify data range for global variables and stubbed functions. For more information, see “Data Range Specifications (DRS)” on page 6-55.

Each of these actions have trade-offs, either in development time, verification time, or the risk of errors. Therefore, before taking any of these actions, it is important to define your quality goals. For more information, see “Analysis and Review Criteria” on page 2-4.

Your quality goals determine how many orange checks are acceptable, what actions you should take to reduce orange checks, and what you should do with the remaining orange checks.

Apply Coding Rules to Reduce Orange Checks

The number of orange checks in your verification results depends strongly on the coding style used in the project. Applying coding rules is an efficient way of reducing the number of orange checks and improves the quality of your code.

Polyspace allows you to check your code with reference to coding rules:

- If your code complies with the coding rules subset that has a *direct* impact on selectivity, the total number of orange checks decreases substantially and the percentage of orange checks representing real bugs increases.
- Some forms of code construction are known to produce orange checks. If your design avoids these forms of construction, you see fewer orange checks in your verification results. You can avoid these forms of construction by checking that your code complies with the coding rules subset that has an *indirect* impact on selectivity.

You can check compliance with the following coding rule subsets:

- MISRA C “Rules in SQ0-Subset1” on page 12-11 and “Rules in SQ0-Subset2” on page 12-13
- MISRA AC AGC “Rules in SQ0-Subset1” on page 12-16 and “Rules in SQ0-Subset2” on page 12-16
- MISRA C++ “SQO Subset 1 – Direct Impact on Selectivity” on page 12-60 and “SQO Subset 2 – Indirect Impact on Selectivity” on page 12-63

For more information on checking coding rules, see “Activate Coding Rules Checker” on page 13-2.

Use Generated Code

Generated code causes fewer orange checks and improves the overall quality of your software.

Generated code obeys a well-defined set of coding rules, which eliminates certain types of coding errors. You observe a higher ratio of green to orange checks in your verification results.

For information about a generated code workflow, see “Simulink Verification”.

Improve Verification Precision

This example shows how to improve the precision of your verification. Improving the verification precision can reduce the number of orange checks in your results. The trade off for this improved precision is increased verification time. Increasing verification precision does not improve the quality of the code itself.

Set Precision Level

The precision level specifies the mathematical algorithm used to compute the cloud of points (polyhedron) containing all possible states for the variables. Changing the precision level does not affect the quality of your code. However, orange checks caused by low precision can become green when verified with higher precision. The default precision level is 2. To set the precision level:

- 1** In the Project Manager perspective, on the **Configuration** pane, select **Precision**.
- 2** From the **Precision level** drop-down list, select the required precision level.

Set Verification Level

The verification level specifies how many times the abstract interpretation algorithm passes through your code. Each pass results in a deeper level of propagation of calling and called context. The deeper the verification goes, the more precise it is. By default, verification proceeds to **Software Safety Analysis Level 4**. To set the verification level:

- 1** In the Project Manager perspective, on the **Configuration** pane, select **Precision**.
- 2** From the **Verification level** drop-down list, select the appropriated level.

Software Safety Analysis Level 0	Software Safety Analysis Level 1
<pre> #include <stdlib.h> void ratio (float x, float *y) { *y=(abs(x-*y))/(x+*y); } void level1 (float x, float y, float *t) { float v; v = y; ratio (x, &y); *t = 1.0/(v - 2.0 * x); } float level2(float v) { float t; t = v; level1(0.0, 1.0, &t); return t; } void main(void) { float r,d; d= level2(1.0); r = 1.0 / (2.0 - d); } </pre>	<pre> #include <stdlib.h> void ratio (float x, float *y) { *y=(abs(x-*y))/(x+*y); } void level1 (float x, float y, float *t) { float v; v = y; ratio (x, &y); *t = 1.0/(v - 2.0 * x); } float level2(float v) { float t; t = v; level1(0.0, 1.0, &t); return t; } void main(void) { float r,d; d= level2(1.0); r = 1.0 / (2.0 - d); } </pre>

In the table, verification produces an orange Division by Zero check during level 0 verification, but turns to green during level 1. The verification gains more knowledge of x as the value is propagated deeper.


Improve Precision of Interprocedural Analysis

This option causes the verification to propagate information within procedures earlier than usual. The precision within each verification level improves. However, using this option can increase verification time. In some cases, a level 1 verification to take longer than a level 4 verification. To use this option:

- 1 In the Project Manager perspective, on the **Configuration** pane, select **Precision**.
- 2 Enter the required value in the field, **Improve Precision of interprocedural analysis**.


Provide Sensitivity Context

This option splits each check within a procedure into sub-checks, depending on the context of a call. This improves precision for discrete calls to the procedure. For example, if a check is red for one call to the procedure and green for another, both colors will be revealed. To use this option:

- 1 In the Project Manager perspective, on the **Configuration** pane, select **Precision**.
- 2 From the **Sensitivity context** drop-down list, select none, auto or custom.
- 3 If you select custom, to add procedure names, use the  button on the **Procedure** box. The verification will split each check into sub-checks only for those procedures.

Inline Procedures

This option creates clones of the specified procedure for each call to it. Using this option reduces the number of aliases in a procedure and can improve precision. To use this option:

- 1 In the Project Manager perspective, on the **Configuration** pane, select **Scaling**.
- 2 To add procedure names, use the  button on the **Procedure** box. The verification will create clones only for those procedures.

Concepts

- “Precision level”
- “Verification level”
- “Improve precision of interprocedural analysis”
- “Sensitivity context”
- “Inline”

Specify Multitasking Behavior

The asynchronous characteristics of your application can have a direct impact on the number of orange checks. Specifying characteristics such as implicit task declarations, mutual exclusion, and critical sections can reduce the number of orange checks in your results.

For example, consider a variable X , and two concurrent tasks T1 and T2.

- X is initialized to 0.
- T1 assigns the value 12 to X .
- T2 divides a local variable by X .
- A division by zero error is possible because T1 can be started before or after T2, so the division causes an orange `Division by Zero` error.

The verification cannot determine if an error will occur without knowing the call sequence. Modelling the task differently could turn this orange check green or red.

For more information, see “Model Synchronous Tasks” on page 7-39.

Effects of Application Code Size

Polyspace verification can make approximations when computing the possible values of the variables, at any point in the program. Such an approximation will always use a superset of the actual possible values.

For example, in a relatively small application, Polyspace verification might retain very detailed information about the data at a particular point in the code, so that for example the variable VAR can take the values { -2; 1; 2; 10; 15; 16; 17; 25 }. If VAR is used to divide, the division is green (because 0 is not a possible value).

If the program being analyzed is large, Polyspace verification would simplify the internal data representation by using a less precise approximation, such as [-2; 2] U {10} U [15 ; 17] U {25} . Here, the same division appears as an orange check.

If the complexity of the internal data becomes even greater later in the verification, Polyspace verification might further simplify the VAR range to (say) [-2; 20].

This phenomenon leads to the increase or the number of orange warnings when the size of the program becomes large.

Note Polyspace verification adjusts the level of simplification based on the required precision level :

- -O0: shorter computation time. Focus only red and gray.
- -O2: less orange warnings.
- -O3: less orange warnings and bigger computation time.

For more information, see “Precision level”.

Overview: Reviewing Orange Checks

After you define a process that matches your quality goals, you end up with a certain number of orange checks for your quality model.

At this point, the goal is not to eliminate orange checks, but to work efficiently with them.

To work efficiently with orange checks:

- “Define Your Review Methodology” on page 11-24
- “Prioritize Orange Check Review” on page 11-30
- “Perform Selective Review” on page 11-25
- “Perform Exhaustive Review” on page 11-37

Define Your Review Methodology

Before reviewing verification results, configure a methodology for your project. The methodology specifies both the type and number of orange checks you need to review. For more information, see “Review Methodologies” on page 10-24.

As part of the process for defining quality goals for your project:

- Polyspace Code Prover provides four predefined review methodologies. Choose one of the predefined review methodologies to specify the number and type of orange checks to review. For more information, see “Organize Results Using Predefined Methodologies” on page 10-26.
- To control the number and type of orange checks to review, define a custom methodology. For more information, see “Organize Results Using Custom Methodologies” on page 10-30.

After you define a methodology, each developer can use the methodology to review verification results. This approach enables the same standard to be used in the review of orange checks at each stage of the development cycle.

Note For information on setting the quality levels for your project, see “Define Software Quality Levels” on page 2-7.

Perform Selective Review

This example shows how to perform a selective orange check review using a predefined methodology. The number and type of orange checks that you review is determined by your review methodology. As a project progresses, change your review methodology to progressively review greater number of orange checks. For example, you can choose the review methodologies, **First checks to review** and **Methodology for C > Light** in the early stages of development. Later, you can choose the methodology, **Methodology for C > Moderate**.

To perform a selective orange review using the methodology **Methodology for C > Light**:

- 1** In the Results Manager perspective, from the drop-down list above the **Results Summary** pane, select the methodology **Methodology for C > Light**.
- 2** Select the first check on the **Results Summary** pane.



The **Source** pane displays the source code for this check.

- 3** Perform a quick code review on each orange check. Your goal is to quickly identify whether the orange check is a:
 - Potential bug — code that will fail under some circumstances.
 - Inconclusive check — check that requires additional information to resolve, such as the call sequence.
 - Data set issue — check originating from a set of data that cannot actually occur.

Note If you cannot understand an orange check quickly, it can be caused by complex code structure or approximate data set used for verification. These checks can often take a substantial amount of time to understand.

- 4** If you cannot identify a cause for the check, proceed to the next check.

- 5** Once you understand the cause of an orange check, record your review on the **Check Review** pane.
 - a** Select a **Classification** to describe the severity of the issue:
 - High
 - Medium
 - Low
 - Not a defect
 - b** Select a **Status** to describe how you intend to address the issue:
 - Fix
 - Improve
 - Investigate
 - Justify with annotations
 - No Action Planned
 - Other
 - Restart with different options
 - Undecided

You can also define your own statuses, which then appear in the user-defined acronym menu.
 - c** In the text box, enter a comment for the reviewed check.
- 6** Click  to navigate to the next check, and repeat steps 3, 4, 5 and 6.
- 7** Continue to click  until you have reviewed all of the checks identified on the **Results Summary** pane.
- 8** Select **File > Save** to save your review comments.

Tip The goal of a selective orange review is to find the maximum number of bugs in a short period of time. Many orange checks take only a few seconds to understand. To maximize the number of defects you can identify, focus on those checks you can understand quickly. Leave the checks that take longer to understand for later analysis.

Related Examples

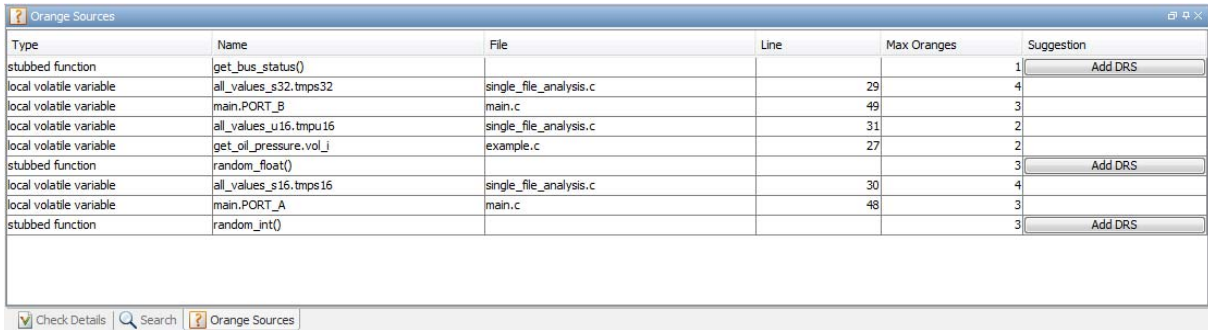
- “Customize Review Status” on page 10-50

Concepts

- “Sources of Orange Checks” on page 11-7

View Sources of Orange Checks

During a verification, the software identifies code that might be the source of orange checks. To view these possible sources of orange checks, in the Results Manager perspective, select **Window > Show/Hide View > Orange Sources**.



Type	Name	File	Line	Max Oranges	Suggestion
stubbed function	get_bus_status()				1 <input type="button" value="Add DRS"/>
local volatile variable	all_values_s32.tmps32	single_file_analysis.c		29	4
local volatile variable	main.PORT_B	main.c		49	3
local volatile variable	all_values_u16.tmpu16	single_file_analysis.c		31	2
local volatile variable	get_oil_pressure.vol_j	example.c		27	2
stubbed function	random_float()				3 <input type="button" value="Add DRS"/>
local volatile variable	all_values_s16.tmps16	single_file_analysis.c		30	4
local volatile variable	main.PORT_A	main.c		48	3
stubbed function	random_int()				3 <input type="button" value="Add DRS"/>

You can see the following information about code that is the source of orange checks:

- **Type** — Type of code element, for example, stubbed function, volatile variable
- **Name** — Name of code element
- **File** — Name of source file
- **Line** — Line number in source file
- **Max Oranges** — Maximum number of orange checks arising from code element
- **Suggestion** — How you can fix the orange check. For example, **Add DRS** suggests that adding a data range specification might resolve the orange check.

Note In rare cases, the **Max Oranges** value may be an approximate value of the maximum number of orange checks.

You can sort the information by category. For example, to sort the information by file name, click **File**.

If the orange source is a variable, to see the line of code where the check occurs, on the **Orange Sources** tab, select the source.

Prioritize Orange Check Review

This example shows how to prioritize your review of orange checks using the **Top 5 orange sources** graph in the **Dashboard** pane. If there are sources (variables or functions) that affect a large number of orange checks, this method can quickly reduce the number of orange checks.

- 1 Open a verification result file, with extension `.pscp`.
- 2 On the **Dashboard** pane, select a column in the **Top 5 orange sources** graph.

Further information about the orange source represented by the column is displayed in the **Orange Sources** pane.

Type	Name	File	Line	Max Oranges	Suggestion
stubbed function	get_bus_status()				1 <input type="button" value="Add DRS"/>
local volatile variable	all_values_s32.tmps32	single_file_analysis.c		29	4
local volatile variable	main.PORT_B	main.c	49		3
local volatile variable	all_values_u16.tmpu16	single_file_analysis.c		31	2
local volatile variable	get_oil_pressure.vol_j	example.c		27	2
stubbed function	random_float()				3 <input type="button" value="Add DRS"/>
local volatile variable	all_values_s16.tmps16	single_file_analysis.c		30	4
local volatile variable	main.PORT_A	main.c	48		3
stubbed function	random_int()				3 <input type="button" value="Add DRS"/>

Source
main.c

Demo_C version 1.0 (22/10/2013)
Author: username

Code covered by verification

Category	Coverage
Procedure	98%
Code operation	95%

Check Distribution

Proven: 92%

Color	Count
Green	260
Orange	23
Gray	6
Red	4

Top 5 orange sources

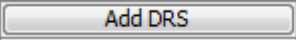
Source	Count
all_values_s32.tmps32	4
main.PORT_B	3
main.PORT_A	3
all_values_u16.tmpu16	2
get_oil_pressure.vol_j	2

[View Orange Sources](#)

Top 5 coding rules violations

Total: 57 checks

Rule ID	Count
7.1	16
21.1	7
9.1	7
16.10	5
8.10	5

- 3 Click the  button, to add data range specification to that source from the **Orange Sources** pane. You can add data range specification for:
- Global variables
 - Stubbed functions
 - User-defined functions
- 4 Repeat step 3 for all the columns in the **Top 5 orange sources** graph.

Note You cannot specify data ranges for volatile variables. Polyspace Code Prover assumes that such variables can take all allowed values.

- 5 Rerun the verification. Reopen the verification result file.

If the checks were caused by input values that Polyspace Code Prover assumed for the sources, you might see a reduction in the number of orange checks.

- 6 Repeat all of these steps in the newly generated **Dashboard** pane. If the same orange source appears as before, consider narrowing the data range before rerunning verification.

Related Examples

- “Refine Data Range Specifications” on page 11-33

Concepts

- “Dashboard” on page 10-80
- “Orange Check Identified as Potential Errors” on page 10-105

Refine Data Range Specifications

This example shows how to refine data range specifications through the **Add DRS** button on the **Orange Sources** tab.

- 1** Select the **Orange Sources** tab.
- 2** On the **Orange Sources** tab, click the **Add DRS** button when available. The **Data Range Configuration** tab opens.

In the example below, clicking the **Add DRS** button for `random_int()` opens the **Data Range Configuration** tab with the node for `random_int()` expanded.

11 Managing Orange Checks

The screenshot shows two windows from the Orange IDE. The top window, titled "Orange Sources", displays a table of source files and functions. The bottom window, titled "Data Range Configuration", shows a tree view of the project's components and their associated data ranges.

Orange Sources Table:

Type	Name	File	Line	Max Oranges	Suggestion
local volatile variable	main.PORT_B	main.c		49	3
local volatile variable	main.PORT_A	main.c		48	3
local volatile variable	get_oil_pressure.vol_j	example.c		27	2
local volatile variable	all_values_u16.tmpu16	single_file_analysis.c		31	2
local volatile variable	all_values_s32.tmps32	single_file_analysis.c		29	4
local volatile variable	all_values_s16.tmps16	single_file_analysis.c		30	4
stubbbed function	random_int()				3
stubbbed function	random_float()				3
stubbbed function	get_bus_status()				1

Data Range Configuration Table:

Name	File	Attributes	Type	Main Generator Called	Init Mode	Init Range	Initialize Pointer	Init Allocated	# Allocated Objects
Global Variables									
User Defined Functions									
Stubbbed Functions									
SEND_MESSAGE()	include.h	extern							
get_bus_status()	example.c	extern							
random_float()	include.h	extern							
random_int()	include.h	extern							
random_int.return	include.h		int32		PERMANENT	min..max			
read_bus_status()	include.h	extern							
read_on_bus()	include.h	extern							
Non Applicable									

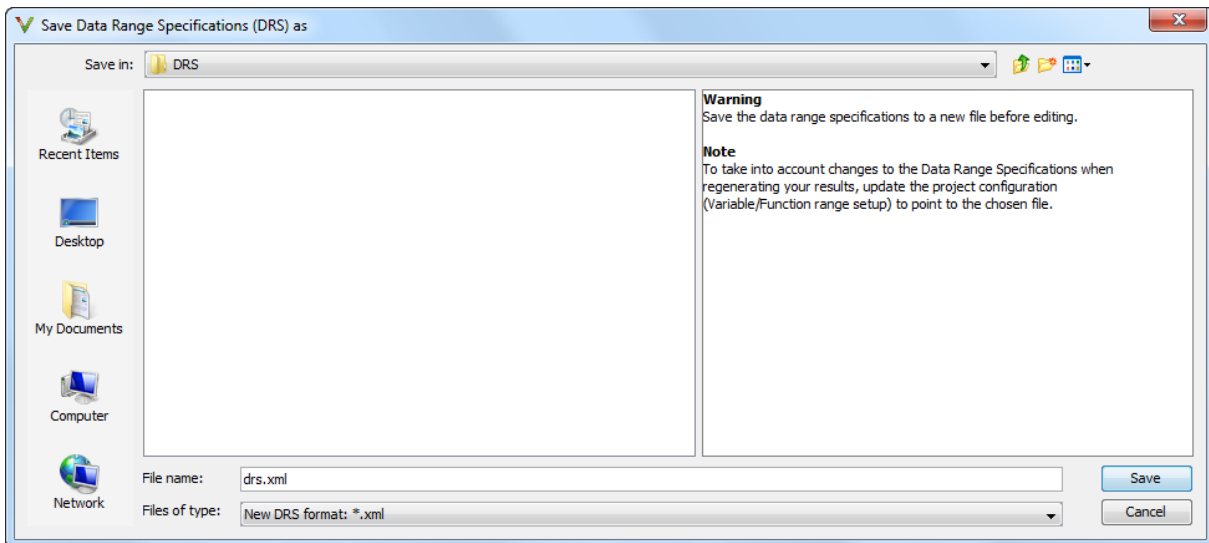
- 3** You can specify a range for the value returned by the function `random_int`. In the **Init Range** column, replace `min..max` by `-10..10`.

In the **Comment** column, you can also add remarks.

- 4** Save the changes to a new configuration file:

- a** Click .

The Save Data Range Specifications (DRS) as dialog box opens.



- b** Navigate to the required folder, and in the **File name** field, specify the name for the new configuration file. Then click **Save**.
- 5** In the Project Manager perspective, use the **Configuration > Code Prover Verification > Inputs & Stubbing > Variable/function range setup** field to specify the new DRS configuration file.
- 6** Rerun the verification. Depending on the data range you specified, the software can replace the orange checks for the source `random_int()` with a green check.

Related Examples

- “Specify Data Ranges Using Existing DRS Configuration” on page 6-58

Perform Exhaustive Review

Most orange checks can be resolved using multiple selective reviews. For more information, see “Perform Selective Review” on page 11-25. However, for extremely critical applications, you might want to resolve all orange checks.

To display all orange checks, in the Results Manager perspective, select **All checks** from the drop-down list above the **Results summary** pane.

An exhaustive orange review is conducted later in the development process, during the unit testing or integration testing phase. The purpose of an exhaustive orange review is to analyze orange checks that are not resolved by selective orange reviews, to identify potential defects in these orange checks.

Before performing an exhaustive orange review, you must balance the time and cost of performing an exhaustive orange review against the potential cost of leaving a defect in the code.

Exhaustive Orange Review Methodology

Performing an exhaustive orange review involves reviewing each orange check individually. As with selective orange review, your goal is to identify whether the orange check is a:

- Potential bug – code which will fail under some circumstances.
- Inconclusive check – a check that requires additional information to resolve, such as the call sequence.
- Data set issue – a theoretical set of data that cannot actually occur.
- Basic imprecision – checks caused by imprecise approximation of the data set used for verification.

Note For more information on each of these causes, see “Sources of Orange Checks” on page 11-7.

Although you must review each check individually, there are some general guidelines to follow.

- Start your review with the files that have the highest selectivity in your application.

If the verification finds only one or two orange checks in a file or function, these checks are probably not caused by either inconclusive verification or basic imprecision. Therefore, it is more likely that these orange checks contain actual defects. These types of orange checks can also be resolved more quickly.

- Next, examine files that contain a large percentage of orange checks compared to the rest of the application. These files can highlight design issues.

If the verification is unable to draw a conclusion, it often means the code is very complex, which can mean low robustness and quality. See “Inconclusive Verification and Code Complexity” on page 11-38.

- For files that you review, first identify checks that you can quickly categorize (such as potential bugs and data set issues).
- Depending on the results of your review, update the code or insert comments to identify the source of the orange check.

Inconclusive Verification and Code Complexity

Sometimes an inconclusive check occurs because the code is too complicated. In these cases, most orange checks in a file are related, and careful analysis identifies a single cause — perhaps a function or a variable modified many times. These situations often focus on functions or variables that have caused problems earlier in the development cycle.

For example, consider a variable `Computed_Speed`.

- `Computed_Speed` is first copied into a signed integer (between -2^{31} and $2^{31}-1$).
- `Computed_Speed` is then copied into an unsigned integer (between 0 and $2^{31}-1$).
- `Computed_Speed` is next copied into a signed integer again.
- Finally, `Computed_Speed` is added to another variable.

The verification can report orange `Overflow`s.

This report does not indicate a real defect , but informs the development team that the variable `Computed_Speed` caused trouble during development and testing phases. The report also suggests that the code is poorly designed.

Resolving Orange Checks Caused by Imprecise Approximation

On rare occasions, a module may contain many orange checks caused by imprecise approximation of the data set used for verification. These checks are usually local to functions, so their impact on the project as a whole is limited.

In cases of imprecise approximation, you can resolve orange checks by increasing the precision level. If this does not resolve the orange check, however, verification cannot help directly.

In these cases, Polyspace software can assist you only through the call tree and dictionary. For more information, see “Dataflow Verification” on page 10-120. The code must be reviewed using alternate means. These alternate means may include:

- Additional unit tests
- Code review with the developer
- Checking an interpolation algorithm in a function
- Checking calibration data

For more information on basic imprecision, see “Sources of Orange Checks” on page 11-7.

Automatic Orange Tester Overview

The Polyspace Automatic Orange Tester performs dynamic stress tests on unproven code (orange checks) to help you identify run-time errors.

Performing an exhaustive orange review manually can be time consuming. The Automatic Orange Tester saves time by automatically creating test cases for all input variables in orange code, and then dynamically testing the code to find actual run-time errors.

Before you run a verification, select the Automatic Orange Tester through the option `-automatic-orange-tester`. See “Select the Automatic Orange Tester” on page 11-44. When the software runs the Automatic Orange Tester at the end of static verification, the software might categorize some orange checks as potential run-time errors. For more information, see “Orange Check Identified as Potential Errors” on page 10-105.

The verification log indicates whether the Automatic Orange Tester has identified potential run-time errors. For example, the following log states that no orange checks have been selected for Review Level 0, which indicates that the Automatic Tester has not identified potential run-time errors.

```
...  
  
Automatic Orange Tester (AOT) statistics:  
- Execution status:  
  * Number of executions: 50  
  ** Successful: 3  
  ** Failed: 47  
- No orange checks selected for Level 0 review.  
- Execution times:  
  * Fastest run: 00:00:00.2  
  * Slowest run: 00:00:00.19  
  
...
```

The Automatic Orange Tester is only one of a few ways by which the software identifies potential run-time errors.

You can also run the Automatic Orange Tester manually. See “Start the Automatic Orange Tester Manually” on page 11-45.

How the Automatic Orange Tester Works

Polyspace verification mathematically analyzes the operations in the code to derive its dynamic properties without actually executing it (see “What is Static Verification” on page 1-6). Although this verification can identify almost all run-time errors, some operations cannot be proved either true or false because the input values are unknown. The software reports these operations as orange checks in the Results Manager perspective (see “What is an Orange Check?” on page 11-3).

If you select the Automatic Orange Tester, at the end of the verification Polyspace generates an *instrumented* version of the source code. For each orange check that could lead to a run-time error, the software generates instrumented code around the orange check. The software compiles the instrumented code and generates binary code. In addition, the software generates randomized test cases based on the input variables. For each test case, the Automatic Orange Tester executes the binary code and records whether the test is a failure. Consider the following example.

```
int x;  
  
x = f();  
x = 1 / x; // orange ZDV: division by zero
```

During static verification, Polyspace determines that the function `f()` can return values between -10 and 10. Therefore, for each test, the Automatic Orange Tester assigns `x` to be a random number between -10 and 10. If the number is 0, division by zero occurs, and the Automatic Orange Tester records the failure.

Limitations of Dynamic Testing

As the Automatic Orange Tester uses a finite number of test cases to analyze the code, there is no guarantee that it will identify a problem in any particular run. Consider an example where a specific variable value causes an error. If no test case uses this value, then the Automatic Orange Tester does not record a failure.

The Automatic Orange Tester creates new randomized test cases for each run. Therefore, there is no guarantee that the results from two separate runs will be the same.

Running more tests increases the chances of finding run-time errors. However, the tests take more time to complete.

Select the Automatic Orange Tester

Before verification, enable the Automatic Orange Tester if you want:

- The software to run the Automatic Orange Tester at the end of the verification
- To manually run the Automatic Orange Tester after the verification

To enable the Automatic Orange Tester:

- 1 In the Project Manager perspective, on the **Configuration** pane, select **Advanced Settings**.
- 2 Select the **Automatic Orange Tester** check box.
- 3 Specify values for the following options:
 - **Number of automatic tests** — Total number of test cases that you want to run. Running more tests increases the chances of finding a run-time error, but takes more time to complete. The default is 500. The maximum value that the software supports is 100,000.
 - **Maximum loop iterations** — Maximum number of iterations allowed before a loop is considered to be an infinite loop. A larger number of iterations decreases the chances of incorrectly identifying an infinite loop, but takes more time to complete. The default is 1000, which is also the maximum value that the software supports.
 - **Maximum test time** — Maximum time (in seconds) allowed for a test before Automatic Orange Tester moves on to the next test. Increasing test time reduces the number of tests that time out, but increases total verification time. The default is 5 seconds. The maximum value that the software supports is 60.

See Also

“Automatic Orange Tester” | “Number of automatic tests” | “Maximum loop iterations” | “Maximum test time”


Related Examples

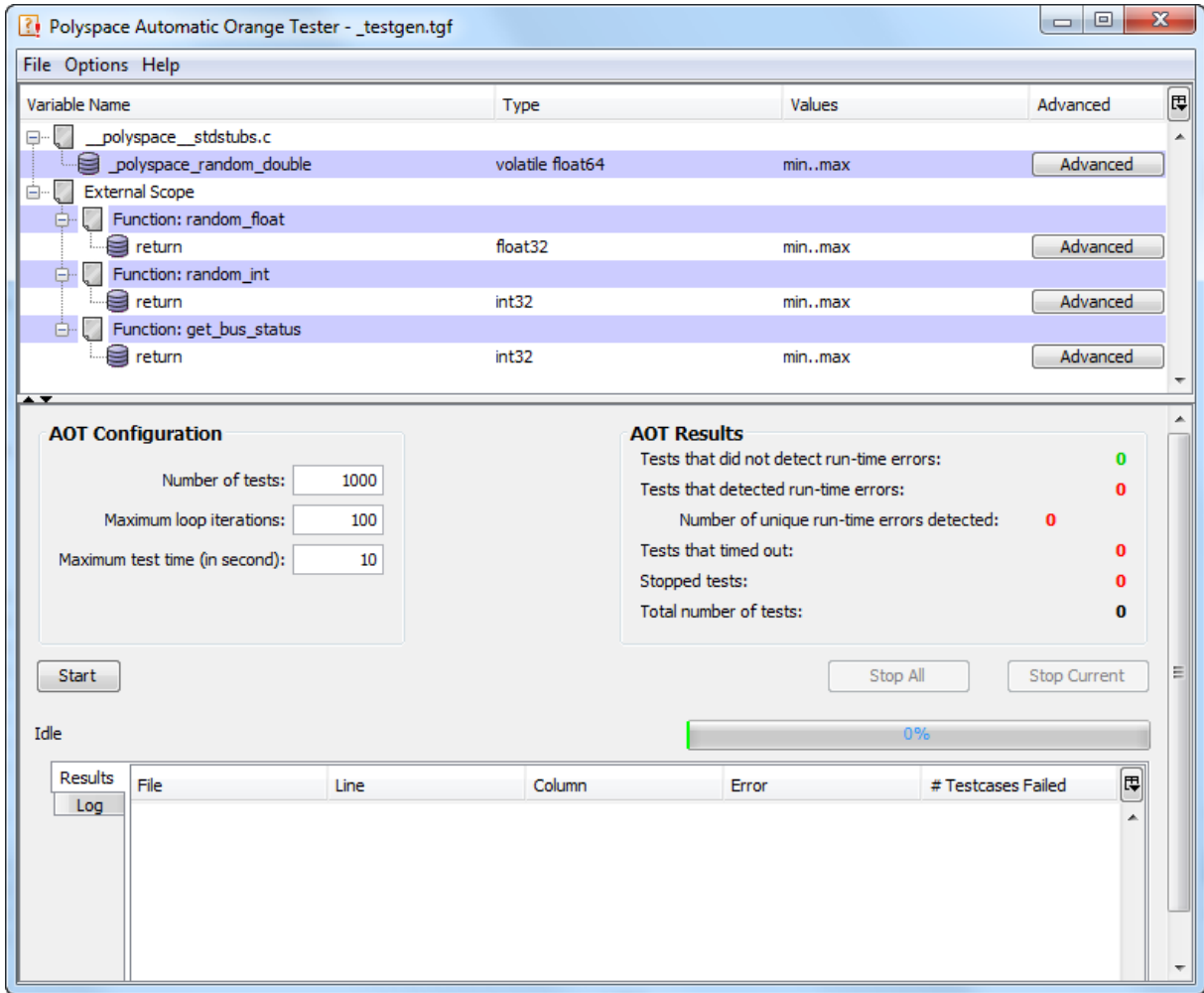
- “Start the Automatic Orange Tester Manually” on page 11-45
- “Review Test Results After Manual Run” on page 11-48

Start the Automatic Orange Tester Manually

If you ran a verification with the **Automatic Orange Tester** option selected, you can run the Automatic Orange Tester manually. See “Select the Automatic Orange Tester” on page 11-44.

To start the Automatic Orange Tester:

- 1 Open your results in the Results Manager perspective.
- 2 On the Results Manager toolbar, click  to open the Automatic Orange Tester.



3 Under AOT Configuration, specify the following parameters:

- **Number of tests** – Specifies the total number of test cases that you want to run. Running more tests increases the chances of finding a run-time error, but also takes more time to complete.
- **Maximum loop iterations** – Specifies the maximum number of loop iterations to perform before the Automatic Orange Tester identifies an

infinite loop. A larger number of iterations decreases the chances of incorrectly identifying an infinite loop, but also might take more time to complete.

- **Maximum test time (in second)** – Specifies the maximum time that an individual test can run (in seconds) before the Automatic Orange Tester moves on to the next test. Increasing the time limit reduces the number of tests that time out, but can also increase the total verification time.

4 Click **Start** to begin testing.

The Automatic Orange Tester generates test cases and runs the dynamic tests.

5 If you want to stop the testing before it is complete:

- Click **Stop Current** to stop the current test and move on to the next one.
 - Click **Stop All** to immediately stop all tests.
- “Review Test Results After Manual Run” on page 11-48

Related Examples

Review Test Results After Manual Run

After testing is complete, the Automatic Orange Tester displays an overview of the testing results, along with detailed information about each failed test.

AOT Results

The AOT Results window displays overview information about the results of your dynamic tests, including:

- **Tests that did not detect run-time errors** – The number of tests that did not produce a run-time error.
- **Tests that detected run-time errors** – The number of tests that produced a run-time error.
- **Number of unique run-time errors detected** – The number of Polyspace checks that produced at least one failed test, as well as the total number of tests that produced a run-time error.
- **Timeout** – The number of tests that exceeded the specified **Per test timeout** limit.
- **Tests that timed out** – The number of tests that were stopped manually.
- **Total number of tests** – The total number of tests completed.

Results

The Results table displays detailed information about each failed test to help you identify the cause of the run-time error. This information includes:

- The file name, line number, and column in which the error was found.
- The type of error that occurred.
- The number of test cases in which the error occurred.

You can view more details about a failed test by clicking the corresponding row in the Results table. The Test Case Detail dialog box opens.

The Test Case Detail dialog box displays the portion of the code in which the error occurred, and gives detailed information about why each test case

failed. Because the Automatic Orange Tester performs run-time tests, this information includes the actual values that caused the error.

You can use this information to quickly identify the cause of the error, and determine whether the code contains a bug.

Log

The Log window displays a complete list of the tests which failed, as well as summary information.

You can copy information from the log window to paste into other applications, such as Microsoft® Excel®.

The log file is also saved in the Polyspace-Instrumented folder with the following file name:

`TestGenerator_day_month_year-time.out`

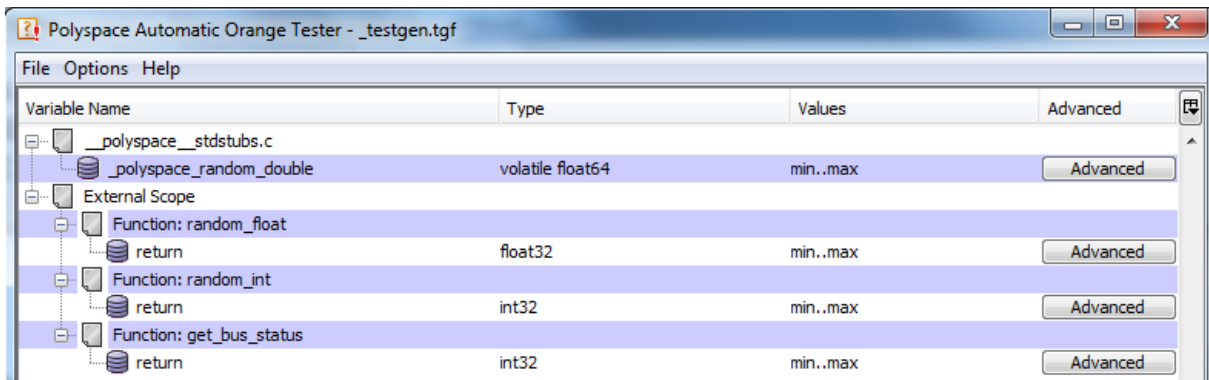
Refine Data Ranges with Automatic Orange Tester

The Automatic Orange Tester allows you to specify ranges for external variables. You can perform run-time tests using real-world values for your variables, rather than randomly selected values.

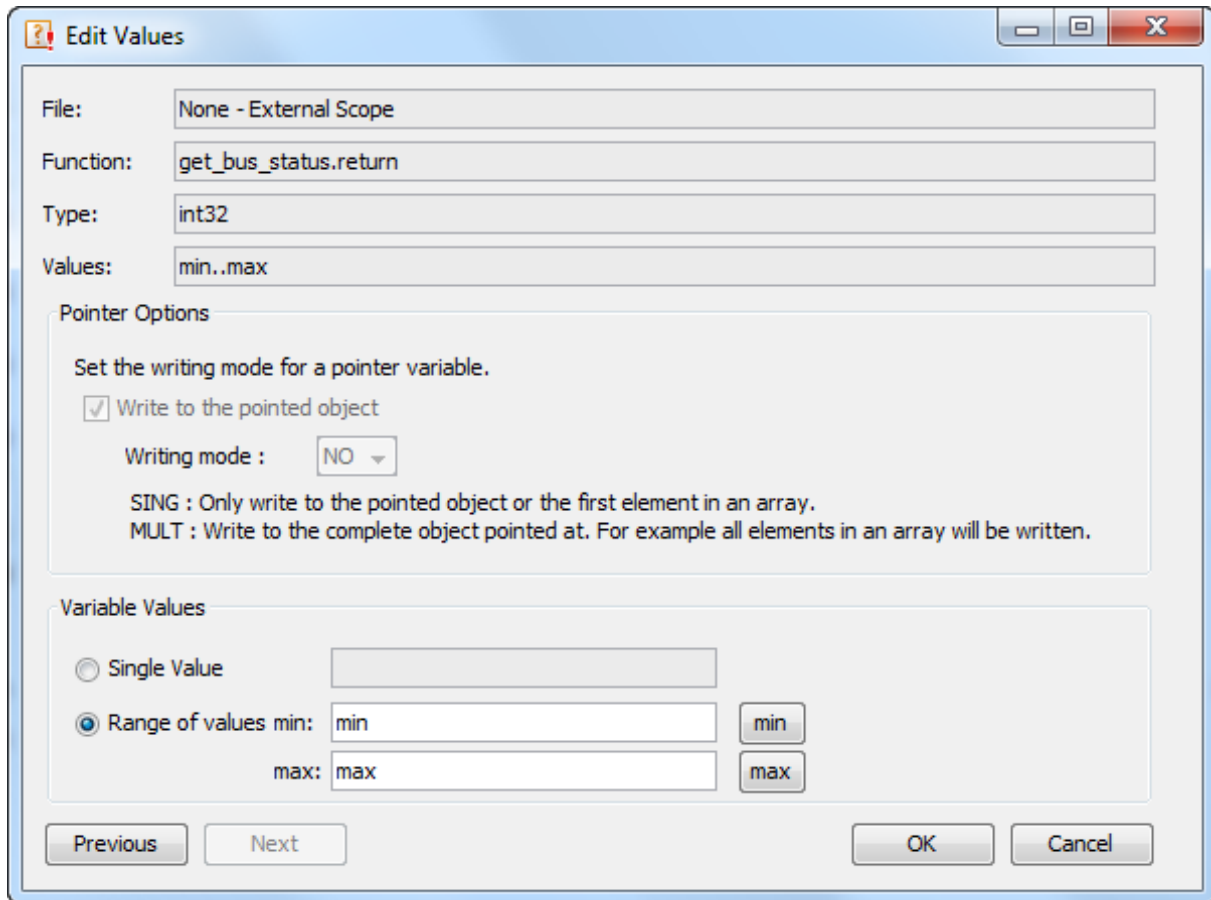
Setting ranges for your variables reduces the number of tests that fail due to unrealistic data values, allowing you to focus on actual problems, rather than purely theoretical problems. Once you set data ranges, you can export them to a DRS file for use in future verifications, reducing the number of orange checks in your results (see “Export Data Ranges for Polyspace Verification” on page 11-54).

To refine your data ranges:

- 1 In the Variables section at the top of the Automatic Orange Tester, identify the variable for which you want to set a data range.



- 2 Select **Advanced**.



3 In the Edit Values dialog box, set the values for the variable:

- **Single Value** — A constant value for the variable.
- **Range of values** — A minimum and maximum value for the variable.

Note For pointers, you can also specify the writing mode:

- **SING** — The tests write only the object or first element in the array.
 - **MULT** — The tests write the complete object, or all elements in the array.
-

- 4** Click **Next** to edit the values for the next variable.
- 5** When you have finished setting values, click **OK** to save your changes and close the Edit Values dialog box.
- 6** Click **Start** to retest the code.

The Automatic Orange Tester generates test cases, runs the tests, and displays the updated results.

The updated results show fewer failed tests, allowing you to focus on actual code problems.

Save and Reuse Your Configuration

You can save your Automatic Orange Tester preferences and variable ranges for use in future dynamic testing.

To save your configuration:

- 1 Select **File > Save**.
- 2 Enter a name and click **Save**.

Your configuration is saved in a `.tgf` file.

To open a configuration from a previous verification:

- 1 Select **File > Open**.
- 2 Select the required `.tgf` file. Then click **Open**.

When you open a previously saved configuration, the **Log** window displays differences in the configuration files. For example:

- If a variable does not exist in the new configuration, a warning is displayed.
- If the ranges for a variable are no longer valid (if the variable type changes, for example), a warning is displayed and the range is changed to the largest valid range for the new data type (if possible).

Export Data Ranges for Polyspace Verification

Once you have set the data ranges for your variables, you can export them to a Data Range Specifications (DRS) file for use in future Polyspace verifications. Using these data ranges allows you to reduce the number of orange checks identified in the Results Manager perspective.

To export your data ranges:

- 1 Set the values for each variable that you want to specify.
- 2 Select **File > Export DRS**.
- 3 Enter a name. Then click **Save**.

The DRS file is saved.

For information on using a DRS file for Polyspace verifications, see “Specify Data Ranges Using Text Files” on page 6-61.

Polyspace-Instrumented Folder

When the software runs the Automatic Orange Tester (AOT) at the end of a static verification, the software creates the Polyspace-Instrumented folder within the verification results folder, for example,

`\Demo_C_Single-File\Module_1\Result_2\Polyspace-Instrumented.`

The Polyspace-Instrumented folder contains files associated with the configuration and running of the Automatic Orange Tester. These files include the following:

- `_testgen.tgf` — A configuration file that contains your Automatic Orange Tester preferences and variable ranges.
- `reachedchecks.txt` — Statistics for orange checks covered by the last run of the Automatic Orange Tester.
- `reachedchecks_dd_mm_yyyy-hhmmss.txt` — Statistics for orange checks covered by the run of the Automatic Orange Tester at the given date and time.
- `TestGenerator_dd_mm_yyyy-hhmmss.out` — Log file created at the given date and time, which contains a list of failed tests as well as summary information.
- `stdout.txt` — Contains data from the standard output (stdout) stream generated by your code during the last run of the Automatic Orange tester.
- `stderr.txt` — Contains messages from the standard error (stderr) stream generated by your code during the last run of the Automatic Orange tester.

Technical Limitations

The Automatic Orange Tester has the following limitations:

In this section...
“Unsupported Polyspace Options” on page 11-56
“Options with Restrictions” on page 11-56
“Unsupported C Routines” on page 11-56

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`:

- `-target [c18 | tms320c3c | x86_64 | sharc21x61]`
- `-data-range-specification` (in global assert mode)

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`

Coding Rule Sets and Concepts

- “Rule Checking” on page 12-2
- “Custom Naming Convention Rules” on page 12-3
- “Polyspace MISRA C and MISRA AC AGC Checkers” on page 12-10
- “Software Quality Objective Subsets (C)” on page 12-11
- “Software Quality Objective Subsets (AC AGC)” on page 12-16
- “MISRA C:2004 Coding Rules” on page 12-18
- “Polyspace MISRA C++ Checker” on page 12-59
- “Software Quality Objective Subsets (C++)” on page 12-60
- “MISRA C++ Coding Rules” on page 12-69
- “Polyspace JSF C++ Checker” on page 12-95
- “JSF C++ Coding Rules” on page 12-96

Rule Checking

Polyspace software allows you to analyze code to demonstrate compliance with established C and C++ coding standards (MISRA C 2004, MISRA C++:2008 or JSF++:2005).

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 individual rules to enforce. Polyspace software then performs rule checking before starting analysis, and reports any errors or warnings in the Results Manager perspective.

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 The Compiler Assistant is selected by default. However, when you enable the Compiler Assistant *and* coding rules checking, the software does not report coding rule violations if there are compilation errors.

Custom Naming Convention Rules

The following table provides information about the custom rules that you can define.

Rule group	Number	Rule Applied	Message generated if rule is violated	Other details
Files (C/C++)	1.1	All source file names must follow the specified pattern.	The source file name “file_name” does not match the specified pattern.	Only the base name is checked. A source file is a file that is not included.
	1.2	All source folder names must follow the specified pattern.	The source dir name “dir_name” does not match the specified pattern.	Only the folder name is checked. A source file is a file that is not included.
	1.3	All include file names must follow the specified pattern.	The include file name “file_name” does not match the specified pattern.	Only the base name is checked. An include file is a file that is included.
	1.4	All include folder names must follow the specified pattern.	The include dir name “dir_name” does not match the specified pattern.	Only the folder name is checked. An include file is a file that is included.
Preprocessing (C/C++)	2.1	All macros must follow the specified pattern.	The macro “macro_name” does not match the specified pattern.	Macro names are checked before preprocessing.
	2.2	All macro parameters must follow the specified pattern.	The macro parameter “param_name” does not match the specified pattern.	Macro parameters are checked before preprocessing.

Rule group	Number	Rule Applied	Message generated if rule is violated	Other details
Type definitions (C/C++)	3.1	All integer types must follow the specified pattern.	The integer type “type_name” does not match the specified pattern.	Applies to integer types specified by typedef statements. Does not apply to enumeration types. For example: typedef signed int int32_t;
	3.2	All float types must follow the specified pattern.	The float type “type_name” does not match the specified pattern.	Applies to float types specified by typedef statements. For example: typedef float f32_t;
	3.3	All pointer types must follow the specified pattern.	The pointer type “type_name” does not match the specified pattern.	Applies to pointer types specified by typedef statements. For example: typedef int* p_int;
	3.4	All array types must follow the specified pattern.	The array type “type_name” does not match the specified pattern.	Applies to array types specified by typedef statements. For example: typedef int[3] a_int_3;
	3.5	All function pointer types must follow the specified pattern.	The function pointer type “type_name” does not match the specified pattern.	Applies to function pointer types specified by typedef statements. For example: typedef void (*pf_callback) (int);

Rule group	Number	Rule Applied	Message generated if rule is violated	Other details
Structures (C/C++)	4.1	All struct tags must follow the specified pattern.	The struct tag “tag_name” does not match the specified pattern.	
	4.2	All struct types must follow the specified pattern.	The struct type “type_name” does not match the specified pattern.	This is the typedef name.
	4.3	All struct fields must follow the specified pattern.	The struct field “field_name” does not match the specified pattern.	
	4.4	All struct bit fields must follow the specified pattern.	The struct bit field “field_name” does not match the specified pattern.	
Classes (C++)	5.1	All class names must follow the specified pattern.	The class tag “tag_name” does not match the specified pattern.	
	5.2	All class types must follow the specified pattern.	The class type “type_name” does not match the specified pattern.	This is the typedef name.
	5.3	All data members must follow the specified pattern.	The data member “member_name” does not match the specified pattern.	
	5.4	All function members must follow the specified pattern.	The function member “member_name” does not match the specified pattern.	

Rule group	Number	Rule Applied	Message generated if rule is violated	Other details
	5.5	All static data members must follow the specified pattern.	The static data member “member_name” does not match the specified pattern.	
	5.6	All static function members must follow the specified pattern.	The static function member “member_name” does not match the specified pattern.	
	5.7	All bitfield members must follow the specified pattern.	The bitfield “member_name” does not match the specified pattern.	
Enumerations (C/C++)	6.1	All enumeration tags must follow the specified pattern.	The enumeration tag “tag_name” does not match the specified pattern.	
	6.2	All enumeration types must follow the specified pattern.	The enumeration type “type_name” does not match the specified pattern.	This is the typedef name.
	6.3	All enumeration constants must follow the specified pattern.	The enumeration constant “constant_name” does not match the specified pattern.	

Rule group	Number	Rule Applied	Message generated if rule is violated	Other details
Functions (C/C++)	7.1	All global functions must follow the specified pattern.	The global function “function_name” does not match the specified pattern.	A global function is a function with external linkage.
	7.2	All static functions must follow the specified pattern.	The static function “function_name” does not match the specified pattern.	A static function is a function with internal linkage.
	7.3	All function parameters must follow the specified pattern.	The function parameter “param_name” does not match the specified pattern.	In C++, applies to non-member functions.
Constants (C/C++)	8.1	All global constants must follow the specified pattern.	The global constant “constant_name” does not match the specified pattern.	A global constant is a constant with external linkage.
	8.2	All static constants must follow the specified pattern.	The static constant “constant_name” does not match the specified pattern.	A static constant is a constant with internal linkage.
	8.3	All local constants must follow the specified pattern.	The local constant “constant_name” does not match the specified pattern.	A local constant is a constant with no linkage.
	8.4	All static local constants must follow the specified pattern.	The static local constant “constant_name” does not match the specified pattern.	A static local constant is a constant declared static in a function.

Rule group	Number	Rule Applied	Message generated if rule is violated	Other details
Variables (C/C++)	9.1	All global variables must follow the specified pattern.	The global variable “var_name” does not match the specified pattern.	A global variable is a variable with external linkage.
	9.2	All static variables must follow the specified pattern.	The static variable “var_name” does not match the specified pattern.	A static variable is a variable with internal linkage.
	9.3	All local variables must follow the specified pattern.	The local variable “var_name” does not match the specified pattern.	A local variable is a variable with no linkage.
	9.4	All static local variables must follow the specified pattern.	The static local variable “var_name” does not match the specified pattern.	A static local variable is a variable declared static in a function.
Name spaces (C++)	10.1	All namespaces must follow the specified pattern.	The namespace “namespace_name” does not match the specified pattern.	
Class templates (C++)	11.1	All class templates must follow the specified pattern.	The class template “template_name” does not match the specified pattern.	
	11.2	All class template parameters must follow the specified pattern.	The class template parameter “param_name” does not match the specified pattern.	

Rule group	Number	Rule Applied	Message generated if rule is violated	Other details
Function templates (C++)	12.1	All function templates must follow the specified pattern.	The function template “template_name” does not match the specified pattern.	Applies to non-member functions.
	12.2	All function template parameters must follow the specified pattern.	The function template parameter “param_name” does not match the specified pattern.	Applies to non-member functions.
	12.3	All function template members must follow the specified pattern.	The function template member “member_name” does not match the specified pattern.	

Polyspace MISRA C and MISRA AC AGC Checkers

The Polyspace MISRA C checker helps you comply with the MISRA C 2004 coding standard.¹⁰

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)” on page 12-11
- “Software Quality Objective Subsets (AC AGC)” on page 12-16

Note The Polyspace MISRA checker is based on MISRA C:2004, which also incorporates MISRA-C Technical Corrigendum (<http://www.misra-c.com>).

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

Software Quality Objective Subsets (C)

In this section...
“Rules in SQ0-Subset1” on page 12-11
“Rules in SQ0-Subset2” on page 12-13

Rules in SQ0-Subset1

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

Rule number	Description
MISRA 8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
MISRA 8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
MISRA 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.
MISRA 11.3	A cast should not be performed between a pointer type and an integral type.
MISRA 12.12	The underlying bit representations of floating-point values shall not be used.
MISRA 13.3	Floating-point expressions shall not be tested for equality or inequality.
MISRA 13.4	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type.

Rule number	Description
MISRA 13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control.
MISRA 14.4	The <i>goto</i> statement shall not be used.
MISRA 14.7	A function shall have a single point of exit at the end of the function.
MISRA 16.1	Functions shall not be defined with variable numbers of arguments.
MISRA 16.2	Functions shall not call themselves, either directly or indirectly.
MISRA 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.
MISRA 17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
MISRA 17.4	Array indexing shall be the only allowed form of pointer arithmetic.
MISRA 17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.
MISRA 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.
MISRA 18.3	An area of memory shall not be reused for unrelated purposes.
MISRA 18.4	Unions shall not be used.
MISRA 20.4	Dynamic heap memory allocation shall not be used.

Note Polyspace software does not check MISRA rule 18.3.

Rules in SQ0-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.

Note Specifying SQ0-subset2 in your **MISRA C rules configuration** checks both the rules listed in SQ0-subset1 and SQ0-subset2.

Rule number	Description
MISRA 6.3	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types
MISRA 8.7	Objects shall be defined at block scope if they are only accessed from within a single function
MISRA 9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures
MISRA 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
MISRA 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
MISRA 10.5	Bitwise operations shall not be performed on signed integer types
MISRA 11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type
MISRA 11.5	Type casting from any type to or from pointers shall not be used
MISRA 12.1	Limited dependence should be placed on C's operator precedence rules in expressions
MISRA 12.2	The value of an expression shall be the same under any order of evaluation that the standard permits

Rule number	Description
MISRA 12.5	The operands of a logical && or shall be primary-expressions
MISRA 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 !)
MISRA 12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned
MISRA 12.10	The comma operator shall not be used
MISRA 13.1	Assignment operators shall not be used in expressions that yield Boolean values
MISRA 13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean
MISRA 13.6	Numeric variables being used within a “for” loop for iteration counting should not be modified in the body of the loop
MISRA 14.8	The statement forming the body of a <i>switch</i> , <i>while</i> , <i>do while</i> or <i>for</i> statement shall be a compound statement
MISRA 14.10	All <i>if else if</i> constructs should contain a final <i>else</i> clause
MISRA 15.3	The final clause of a <i>switch</i> statement shall be the <i>default</i> clause
MISRA 16.3	Identifiers shall be given for all of the parameters in a function prototype declaration
MISRA 16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
MISRA 16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty
MISRA 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

Rule number	Description
MISRA 19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives
MISRA 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 ##
MISRA 19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator
MISRA 19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.
MISRA 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; }
```

Software Quality Objective Subsets (AC AGC)

In this section...
“Rules in SQ0-Subset1” on page 12-16
“Rules in SQ0-Subset2” on page 12-16

Rules in SQ0-Subset1

The following set of MISRA AC AGC coding rules typically reduces the number of unproven results.

- 5.2
- 8.11 and 8.12
- 11.2 and 11.3
- 12.12
- 14.7
- 16.1 and 16.2
- 17.3 and 17.6
- 18.4

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 SQ0-Subset2

Good design practices generally lead to less code complexity, which can reduce the number of unproven results. The following set of coding rules enforce good design practices.

- 5.2
- 6.3
- 8.7, 8.11, and 8.12
- 9.3

- 11.1, 11.2, 11.3, and 11.5
- 12.2, 12.9, 12.10, and 12.12
- 14.7
- 16.1, 16.2, 16.3, 16.8, and 16.9
- 17.3, and 17.6
- 18.4
- 19.9, 19.10, 19.11, and 19.12
- 20.3

Note When you specify `SQO-subset2` for your MISRA AC AGC rules configuration, the software checks the rules listed in `SQO-subset1` and `SQO-subset2`.

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 Coding Rules

In this section...
“Supported MISRA C:2004 Rules” on page 12-18
“MISRA C:2004 Rules Not Checked” on page 12-56

Supported MISRA C:2004 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 “Detailed 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

N.	MISRA Definition	Messages in report file	Detailed 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.	<p>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:</p> <ul style="list-style-type: none"> • ANSI C does not allow <code>#include_next</code> • ANSI C does not allow macros with variable arguments list • ANSI C does not allow <code>#assert</code> • ANSI C does not allow <code>#unassert</code> • ANSI C does not allow testing assertions • ANSI C does not allow <code>#ident</code> • ANSI C does not allow <code>#scs</code> • text following <code>#else</code> violates ANSI standard. • text following <code>#endif</code> 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	Detailed Polyspace Specification
		<ul style="list-style-type: none"> • text following '#else' or '#endif' violates ANSI standard. • 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. 	

Language Extensions

N.	MISRA Definition	Messages in report file	Detailed 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 asm functions or in asm pragma (only warning is given on

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
			asm statements even if it is encapsulated by a MACRO).
2.2	Source code shall only use <code>/* */</code> style comments	C++ comments shall not be used.	C++ comments are handled as comments but lead to a violation of this MISRA rule Note: This rule cannot be annotated in the source code.
2.3	The character sequence <code>/*</code> shall not be used within a comment	The character sequence <code>/*</code> shall not appear within a comment.	This rule violation is also raised when the character sequence <code>/*</code> inside a C++ comment. Note: This rule cannot be annotated in the source code.

Documentation

Rule	MISRA Definition	Messages in report file	Detailed Polyspace Specification
3.4	All uses of the <code>#pragma</code> directive shall be documented and explained.	All uses of the <code>#pragma</code> directive shall be documented and explained.	To check this rule, the option <code>-allowed-pragmas</code> must be set to the list of pragmas that are allowed in source files. Warning if a pragma that does not belong to the list is found.

Character Sets

N.	MISRA Definition	Messages in report file	Detailed 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	Detailed 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.
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	<ul style="list-style-type: none"> Local declaration of XX is hiding another identifier. Declaration of parameter XX is hiding another identifier. 	Assumes that rule 8.1 is not violated.
5.3	A typedef name shall be a unique identifier	{ typedef name }'%s' should not be reused. (already used as { typedef name } at %s:%d)	Warning when a typedef name is reused as another identifier name.
5.4	A tag name shall be a unique identifier	{tag name }'%s' should not be reused. (already used as {tag name } at %s:%d)	Warning when a tag name is reused as another identifier name

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
5.5	No object or function identifier with a static storage duration should be reused.	{ static identifier/parameter name }'%s' should not be reused. (already used as {static identifier/parameter name } with static storage duration at %s:%d)	Warning when a static name is reused as another identifier name
5.6	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.	{member name }'%s' should not be reused. (already used as { member name } at %s:%d)	Warning when a idf in a namespace is reused in another namespace
5.7	No identifier name should be reused.	{identifier}'%s' should not be reused. (already used as { identifier} at %s:%d)	No violation reported when: <ul style="list-style-type: none"> • Different functions have parameters with the same name • Different functions have local variables with the same name • A function has a local variable that has the same name as a parameter of another function

Types

N.	MISRA Definition	Messages in report file	Detailed 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.	<ul style="list-style-type: none"> • Value of type plain char is implicitly converted to signed char. • Value of type plain char is implicitly converted to unsigned char. • Value of type signed char is implicitly converted to plain char. • Value of type unsigned char is implicitly converted to plain char. 	Warning if value of type plain char is implicitly converted to value of type signed char or unsigned char.
6.3	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types	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 <i>unsigned int</i> or <i>signed int</i> .	Bit fields shall only be defined to be of type unsigned int or signed int.	
6.5	Bit fields of type <i>signed int</i> shall be at least 2 bits long.	Bit fields of type signed int shall be at least 2 bits long.	No warning on anonymous signed int bitfields of width 0 - Extended to all signed bitfields of size <= 1 (if Rule 6.4 is violated).

Constants

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.	<ul style="list-style-type: none"> • Octal constants other than zero and octal escape sequences shall not be used. • Octal constants (other than zero) should not be used. • Octal escape sequences should not be used. 	

Declarations and Definitions

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	<ul style="list-style-type: none"> • Function XX has no complete prototype visible at call. • Function XX has no prototype visible at definition. 	Prototype visible at call must be complete.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated	Whenever an object or function is declared or defined, its type shall be explicitly stated.	
8.3	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.	Definition of function 'XX' incompatible with its declaration.	Assumes that rule 8.1 is not violated. The rule is restricted to compatible types. Can be turned to Off

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
8.4	If objects or functions are declared more than once their types shall be compatible.	<ul style="list-style-type: none"> • If objects or functions are declared more than once their types shall be compatible. • Global declaration of 'XX' function has incompatible type with its definition. • Global declaration of 'XX' variable has incompatible type with its definition. 	Violations of this rule might be generated during the link phase.
8.5	There shall be no definitions of objects or functions in a header file	<ul style="list-style-type: none"> • Object 'XX' should not be defined in a header file. • Function 'XX' should not be defined in a header file. • Fragment of function should not be defined in a header file. 	Tentative of definitions are considered as definitions.
8.6	Functions shall always be declared at file scope.	Function 'XX' should be declared at file scope.	
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 of definitions are ignored).

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
8.9	Definition: An identifier with external linkage shall have exactly one external definition.	<ul style="list-style-type: none"> • Procedure/Global variable XX multiply defined. • Forbidden multiple tentative of definition for object XX • Global variable has multiples tentative of definitions • Undefined global variable XX 	Tentative of definitions are considered as definitions, no warning on predefined symbols.
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.
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage	static storage class specifier should be used on internal linkage symbol XX.	
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization	Array XX has unknown size.	

Initialization

N.	MISRA Definition	Messages in report file	Detailed 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.
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	Detailed Polyspace Specification
10.1	<p>The value of an expression of integer type shall not be implicitly converted to a different underlying type if:</p> <ul style="list-style-type: none"> • it is not a conversion to a wider integer type of the same signedness, or • the expression is complex, or 	<ul style="list-style-type: none"> • Implicit conversion of the expression of underlying type XX to the type XX that is not a wider integer type of the same signedness. • Implicit conversion of one of the binary operands whose underlying types are XX and XX 	<p>1 ANSI C base types order (signed char, short, int, long) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1. The same interpretation is applied on the unsigned version of base types.</p>

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
	<ul style="list-style-type: none"> • the expression is not constant and is a function argument, or • the expression is not constant and is a return expression 	<ul style="list-style-type: none"> • Implicit conversion of the binary right hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary left hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary right hand operand of underlying type XX to XX that is not a wider integer type of the same signedness or Implicit conversion of the binary ? left hand operand of underlying type XX to XX, but it is a complex expression. • Implicit conversion of complex integer expression of underlying type XX to XX. • Implicit conversion of non-constant integer expression of underlying type XX in function return whose expected type is XX. • Implicit conversion of non-constant integer expression of underlying 	<p>2 An expression of bool or enum types has int as underlying type.</p> <p>3 Plain char may have signed or unsigned underlying type (depending on Polyspace target configuration or option setting).</p> <p>4 The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width is not taken into account and it assumes that only signed unsigned int are used for bitfield (Rule 6.4).</p> <p>5 No violation reported when:</p> <ul style="list-style-type: none"> • The implicit conversion is a type widening, without change of signedness if integer • The expression is an argument expression or a return expression <p>6 No violation reported when the following are all true:</p>

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
		<p>type XX as argument of function whose corresponding parameter type is XX.</p>	<ul style="list-style-type: none"> • Implicit conversion applies to a constant expression and is a type widening, with a possible change of signedness if 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
10.2	<p>The value of an expression of floating type shall not be implicitly converted to a different type if</p> <ul style="list-style-type: none"> • it is not a conversion to a wider floating type, or • the expression is complex, or • the expression is a function argument, or • the expression is a return expression 	<ul style="list-style-type: none"> • Implicit conversion of the expression from XX to XX that is not a wider floating type. • Implicit conversion of the binary ? right hand operand from XX to XX, but it is a complex expression. • Implicit conversion of the binary ? right hand operand from XX to XX that is not a wider floating type or Implicit conversion of the binary ? left hand operand from 	<p>ANSI C base types order (float, double) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1.</p> <p>No violation reported when:</p> <ul style="list-style-type: none"> • The implicit conversion is a type widening • The expression is an argument expression or a return expression.

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
		<p>XX to XX, but it is a complex expression.</p> <ul style="list-style-type: none"> • 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. 	
10.3	<p>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</p>	<p>Complex expression of underlying type XX may only be cast to narrower integer type of same signedness, however the destination type is XX.</p>	<ul style="list-style-type: none"> • 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

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
			<p>configuration or option setting).</p> <ul style="list-style-type: none"> 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.
10.5	If the bitwise operator ~ and << are applied to an operand of underlying type <i>unsigned char</i> or <i>unsigned short</i> , the result shall be immediately cast to the underlying type of the operand	Bitwise [<< ~] is applied to the operand of underlying type [unsigned char unsigned short], the result shall be immediately cast to the underlying type.	
10.6	The “U” suffix shall be applied to all constants of <i>unsigned</i> types	No explicit 'U suffix on constants of an unsigned type.	<p>Warning when the type determined from the value and the base (octal, decimal or hexadecimal) is unsigned and there is no suffix u or U.</p> <p>For example, when the size of the int and long int data types is 32 bits, the coding rule checker will</p>

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
			<p>report a violation of rule 10.6 for the following line:</p> <pre>int a = 2147483648;</pre> <p>There is a difference between decimal and hexadecimal constants when <code>int</code> and <code>long int</code> are not the same size.</p>

Pointer Type Conversion

N.	MISRA Definition	Messages in report file	Detailed 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.	<p>Casts and implicit conversions involving a function pointer.</p> <p>Casts or implicit conversions from <code>NULL</code> or <code>(void*)0</code> do not give any warning.</p>
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.	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

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.	A cast should not be performed between a pointer to object type and a different pointer to object type.	
11.5	A cast shall not be performed that removes any <i>const</i> or <i>volatile</i> qualification from the type addressed by a pointer	A cast shall not be performed that removes any <i>const</i> or <i>volatile</i> qualification from the type addressed by a pointer	Extended to all conversions

Expressions

N.	MISRA Definition	Messages in report file	Detailed 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.	<ul style="list-style-type: none"> 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. 	The expression is a simple expression of symbols (Unlike <code>i = i++</code> ; no detection on <code>tab[2] = tab[2]++</code>);. Rule 12.2 check assumes that no assignment in expressions that yield a Boolean values (rule 13.1) and the comma operator is not used (rule 12.10).
12.2	The <code>sizeof</code> operator should not be used on expressions that contain side effects.	The <code>sizeof</code> operator should not be used on expressions that contain side effects.	No warning on volatile accesses

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
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.	<ul style="list-style-type: none"> • 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. 	<p>During preprocessing, violations of this rule are detected on the expressions in #if directives.</p> <p>Allowed exception on associatively (a && b && c), (a b c).</p>
12.6	Operands of logical operators (&&, and !) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (&&, or !).	<ul style="list-style-type: none"> • Operand of '!' logical operator should be effectively Boolean. • Left operand of '%s' logical operator should be effectively Boolean. • Right operand of '%s' logical operator should be effectively Boolean. • %s operand of '%s' is effectively Boolean. Boolean should not be used as operands to operators other than '&&', ' ', '!', '=', '==', '!=', and '?:'. 	<p>The operand of a logical operator should be a Boolean data type. Although the C standard does not explicitly define the Boolean data type, the standard implicitly assumes the use of the Boolean data type.</p> <p>Some operators may return Boolean-like expressions, for example, (var == 0).</p> <p>Consider the following code:</p> <pre>unsigned char flag; if (!flag)</pre> <p>The rule checker reports a violation of rule 12.6:</p> <p>Operand of '!' logical operator should be effectively Boolean.</p>

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
			<p>The operand flag is not a Boolean but an unsigned char.</p> <p>To be compliant with rule 12.6, the code must be rewritten either as</p> <pre>if (!(flag != 0))</pre> <p>or</p> <pre>if (flag == 0)</pre> <p>The use of the option <code>-boolean-types</code> may increase or decrease the number of warnings generated.</p>
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed	<ul style="list-style-type: none"> • [~/Left Shift/Right shift/&] operator applied on an expression whose underlying type is signed. • Bitwise ~ on operand of signed underlying type XX. • Bitwise [<< >>] on left hand operand of signed underlying type XX. • Bitwise [& ^] on two operands of s 	<p>The underlying type for an integer is signed when:</p> <ul style="list-style-type: none"> • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
12.8	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	<ul style="list-style-type: none"> • shift amount is negative • shift amount is bigger than 64 • Bitwise [<< >>] count out of range [0 ..X] (width of the underlying type XX of the left hand operand - 1).. 	<p>The numbers that are manipulated in preprocessing directives are 64 bits wide so that valid shift range is between 0 and 63</p> <p>Check is also extended onto bitfields with the field width or the width of the base type when it is within a complex expression</p>
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	<ul style="list-style-type: none"> • Unary - on operand of unsigned underlying type XX. • Minus operator applied to an expression whose underlying type is unsigned 	<p>The underlying type for an integer is signed when:</p> <ul style="list-style-type: none"> • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number
12.10	The comma operator shall not be used.	The comma operator shall not be used.	
12.11	Evaluation of constant unsigned expression should not lead to wraparound.	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
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: <ul style="list-style-type: none"> • A float pointer is cast as a pointer to another data type. Casting a float pointer as a pointer to void does not generate a warning. • A float is packed with another data type. For example: <pre style="margin-left: 20px;">union { float f; int i; }</pre>
12.13	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	Warning when ++ or -- operators are not used alone.

Control Statement Expressions

N.	MISRA Definition	Messages in report file	Detailed 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.	
13.2	Tests of a value against zero should be made explicit,	Tests of a value against zero should be made explicit,	No warning is given on integer constants. Example: if (2)

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
	unless the operand is effectively Boolean	unless the operand is effectively Boolean	The use of the option <code>-boolean-types</code> may increase or decrease the number of warnings generated.
13.3	Floating-point expressions shall not be tested for equality or inequality.	Floating-point expressions shall not be tested for equality or inequality.	Warning on directs tests only.
13.4	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type	If <i>for</i> index is a variable symbol, checked that it is not a float.
13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control	<ul style="list-style-type: none"> • 1st expression should be an assignment. • Bad type for loop counter (XX). • 2nd expression should be a comparison. • 2nd expression should be a comparison with loop counter (XX). • 3rd expression should be an assignment of loop counter (XX). • 3rd expression: assigned variable should be the loop counter (XX). • The following kinds of <i>for</i> loops are allowed: 	Checked if the <i>for</i> loop index (V) is a variable symbol; checked if V is the last assigned variable in the first expression (if present). Checked if, in first expression, if present, is assignment of V; checked if in 2nd expression, if present, must be a comparison of V; Checked if in 3rd expression, if present, must be an assignment of V.

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
		<p>(a) all three expressions shall be present;</p> <p>(b) the 2nd and 3rd expressions shall be present with prior initialization of the loop counter;</p> <p>(c) all three expressions shall be empty for a deliberate infinite loop.</p>	
13.6	Numeric variables being used within a <i>for</i> loop for iteration counting should not be modified in the body of the loop.	Numeric variables being used within a <i>for</i> loop for iteration counting should not be modified in the body of the loop.	Detect only direct assignments if the <i>for</i> loop index is known and if it is a variable symbol.
13.7	Boolean operations whose results are invariant shall not be permitted	<ul style="list-style-type: none"> • Boolean operations whose results are invariant shall not be permitted. Expression is always true. • Boolean operations whose results are invariant shall not be permitted. Expression is always false. • Boolean operations whose results are invariant shall not be permitted. 	During compilation, check comparisons with at least one constant operand.

Control Flow

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
14.1	There shall be no unreachable code.	There shall be no unreachable code.	
14.2	All non-null statements shall either have at least one side effect however executed, or cause control flow to change	<ul style="list-style-type: none"> • All non-null statements shall either: • have at least one side effect however executed, or • cause control flow to change 	
14.3	<p>All non-null statements shall either</p> <ul style="list-style-type: none"> • have at least one side effect however executed, or • cause control flow to change 	A null statement shall appear on a line by itself	<p>We assume that a ';' is a null statement when it is the first character on a line (excluding comments). The rule is violated when:</p> <ul style="list-style-type: none"> • there are some comments before it on the same line. • there is a comment immediately after it • there is something else than a comment after the ';' on the same line.
14.4	The <i>goto</i> statement shall not be used.	The <i>goto</i> statement shall not be used.	
14.5	The <i>continue</i> statement shall not be used.	The <i>continue</i> statement shall not be used.	
14.6	For any iteration statement there shall be at most one <i>break</i> statement used for loop termination	For any iteration statement there shall be at most one <i>break</i> statement used for loop termination	

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
14.7	A function shall have a single point of exit at the end of the function	A function shall have a single point of exit at the end of the function	
14.8	The statement forming the body of a <i>switch</i> , <i>while</i> , <i>do while</i> or <i>for</i> statement shall be a compound statement	<ul style="list-style-type: none"> • The body of a do while statement shall be a compound statement. • The body of a for statement shall be a compound statement. • The body of a switch statement shall be a compound statement 	
14.9	An <i>if (expression)</i> construct shall be followed by a compound statement. The <i>else</i> keyword shall be followed by either a compound statement, or another <i>if</i> statement	<ul style="list-style-type: none"> • An if (expression) construct shall be followed by a compound statement. • The else keyword shall be followed by either a compound statement, or another if statement 	
14.10	All <i>if else if</i> constructs should contain a final <i>else</i> clause.	All if else if constructs should contain a final else clause.	

Switch Statements

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
15.0	<p>Unreachable code is detected between switch statement and first case.</p> <hr/> <p>Note This is not a MISRA C2004 rule.</p> <hr/>	switch statements syntax normative restrictions.	<p>Warning on declarations or any statements before the first switch case.</p> <p>Warning on label or jump statements in the body of switch cases.</p> <p>On the following example, the rule is displayed in the log file at line 3:</p> <pre> 1 ... 2 switch(index) { 3 var = var + 1; // RULE 15.0 // violated 4case 1: ... </pre> <p>The code between switch statement and first case is checked as dead code by Polyspace. It follows ANSI standard behavior.</p>
15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a <i>switch</i> statement	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	
15.2	An unconditional <i>break</i> statement shall terminate every non-empty switch clause	An unconditional break statement shall terminate every non-empty switch clause	Warning for each non-compliant case clause.

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
15.3	The final clause of a <i>switch</i> statement shall be the <i>default</i> clause	The final clause of a switch statement shall be the default clause	
15.4	A <i>switch</i> expression should not represent a value that is effectively Boolean	A switch expression should not represent a value that is effectively Boolean	The use of the option <code>-boolean-types</code> may increase the number of warnings generated.
15.5	Every <i>switch</i> statement shall have at least one <i>case</i> clause	Every switch statement shall have at least one case clause	

Functions

N.	MISRA Definition	Messages in report file	Detailed 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 (Call graph in the Results Manager perspective gives the information). Polyspace also checks that partially during 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.
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.

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
16.5	Functions with no parameters shall be declared with parameter type <i>void</i> .	Functions with no parameters shall be declared with parameter type <i>void</i> .	Definitions are also checked.
16.6	The number of arguments passed to a function shall match the number of parameters.	<ul style="list-style-type: none"> • Too many arguments to XX. • Insufficient number of arguments to XX. 	Assumes that rule 8.1 is not violated.
16.7	A pointer parameter in a function prototype should be declared as pointer to <i>const</i> if the pointer is not used to modify the addressed object.	Pointer parameter in a function prototype should be declared as pointer to <i>const</i> if the pointer is not used to modify the addressed object.	Warning if a non- <i>const</i> pointer parameter is either not used to modify the addressed object or is passed to a call of a function that is declared with a <i>const</i> pointer parameter.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Missing return value for non-void function XX.	Warning when a non-void function is not terminated with an unconditional return with an expression.
16.9	A function identifier shall only be used with either a preceding <i>&</i> , or with a parenthesized parameter list, which may be empty.	Function identifier XX should be preceded by a <i>&</i> or followed by a parameter list.	
16.10	If a function returns error information, then that error information shall be tested.	If a function returns error information, then that error information shall be tested.	<p>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 <i>void</i>.</p> <p>No check performed for calls of <i>memcpy</i>, <i>memmove</i>,</p>

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
			memset, strcpy, strncpy, strcat, or strncpy.

Pointers and Arrays

N.	MISRA Definition	Messages in report file	Detailed 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).
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	Detailed 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.	<ul style="list-style-type: none"> • An object shall not be assigned to an overlapping object. • Destination and source of XX overlap, the behavior is undefined. 	
18.4	Unions shall not be used	Unions shall not be used.	

Preprocessing Directives

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
19.1	#include statements in a file shall only be preceded by other preprocessors directives or comments	A message is displayed when a #include directive is preceded by other things than preprocessor directives, comments, spaces or “new lines”.	
19.2	Nonstandard characters should not occur in header file names in #include directives	<ul style="list-style-type: none"> • A message is displayed on characters ', \, " or /* between < and > in #include <filename> • A message is displayed on characters ', \ or /* between " and " in #include "filename" 	

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
19.3	The <i>#include</i> directive shall be followed by either a <filename> or "filename" sequence.	<ul style="list-style-type: none"> • '#include' expects "FILENAME" or <FILENAME> • '#include_next' expects "FILENAME" or <FILENAME> 	
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.	Macro '<name>' does not expand to a compliant construct.	<p>We assume that a macro definition does not violate this rule when it expands to:</p> <ul style="list-style-type: none"> • a braced construct (not necessarily an initializer) • a parenthesized construct (not necessarily an expression) • a number • a character constant • a string constant (can be the result of the concatenation of string field arguments and literal strings) • the following keywords: typedef, extern, static, auto, register, const, volatile, __asm__ and __inline__ • a do-while-zero construct

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
19.5	Macros shall not be #defined and #undefd within a block.	<ul style="list-style-type: none"> • Macros shall not be #defined 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.	Message on all function-like macros expansions	
19.8	A function-like macro shall not be invoked without all of its arguments	<ul style="list-style-type: none"> • arguments given to macro '<name>' • macro '<name>' used without args. • macro '<name>' used with just one arg. • macro '<name>' used with too many (<number>) args. 	
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	Macro argument shall not look like a preprocessing directive.	This rule is detected as violated when the '#' character appears in a macro argument (outside a string or character constant)

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.	Parameter instance shall be enclosed in parentheses.	<p>If x is a macro parameter, the following instances of x as an operand of the # and ## operators do not generate a warning: #x, ##x, and x##. Otherwise, parentheses are required around x.</p> <p>The software does not generate a warning if a parameter is reused as an argument of a function or function-like macro. For example, consider a parameter x. The software does not generate a warning if x appears as (x) or (x, or ,x) or ,x,.</p>
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	'<name>' is not defined.	
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.	More than one occurrence of the # or ## preprocessor operators.	
19.13	The # and ## preprocessor operators should not be used	Message on definitions of macros using # or ## operators	

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
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.	<p>When a header file is formatted as:</p> <pre>#ifndef <control macro> #define <control macro> <contents> #endif</pre> <p>or:</p> <pre>#ifdef <control macro> #error ... #else #define <control macro> <contents> #endif</pre> <p>it is assumed that precautions have been taken to prevent multiple inclusions. Otherwise, a violation of this MISRA rule is detected.</p>

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	directive is not syntactically meaningful.	
19.17	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> or <code>#ifdef</code> directive to which they are related.	<ul style="list-style-type: none"> • <code>#elif</code> not within a conditional. • <code>#else</code> not within a conditional. • <code>#elif</code> not within a conditional. • <code>#endif</code> not within a conditional. • unbalanced <code>#endif</code>. • unterminated <code>#if</code> conditional. • unterminated <code>#ifdef</code> conditional. • unterminated <code>#ifndef</code> conditional. 	

Standard Libraries

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
20.1	Reserved identifiers, macros and functions in the standard library, shall	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be redefined. 	

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
	not be defined, redefined or undefined.	<ul style="list-style-type: none"> The macro '<code><name></code>' shall not be undefined. 	
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 of definitions are considered as definitions.
20.3	The validity of values passed to library functions shall be checked.	Validity of values passed to library functions shall be checked	<p>Warning for argument in library function call if the following are all true:</p> <ul style="list-style-type: none"> Argument is a local variable Local variable is not tested between last assignment and call to the library function Library function is a common mathematical function Corresponding parameter of the library function has a restricted input domain. <p>The library function can be one of the following : <code>sqrt</code>, <code>tan</code>, <code>pow</code>, <code>log</code>, <code>log10</code>, <code>fmod</code>, <code>acos</code>, <code>asin</code>, <code>acosh</code>, <code>atanh</code>, or <code>atan2</code>.</p>

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
20.4	Dynamic heap memory allocation shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier <code>XX</code> should not be used. 	In case the dynamic heap memory allocation functions are actually macros and the macro is expanded in the code, this rule is detected as violated. Assumes rule 20.2 is not violated.
20.5	The error indicator <code>errno</code> shall not be used	The error indicator <code>errno</code> shall not be used	Assumes that rule 20.2 is not violated
20.6	The macro <code>offsetof</code> , in library <code><stddef.h></code> , shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier <code>XX</code> should not be used. 	Assumes that rule 20.2 is not violated
20.7	The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier <code>XX</code> should not be used. 	In case the <code>longjmp</code> function is actually a macro and the macro is expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.8	The signal handling facilities of <code><signal.h></code> shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier <code>XX</code> should not be used. 	In case some of the signal functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.9	The input/output library <code><stdio.h></code> shall not be used in production code.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier <code>XX</code> 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

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
20.10	The library functions atof, atoi and toll from library <stdlib.h> shall not be used.	<ul style="list-style-type: none"> • The macro '<name>' shall not be used. • Identifier XX should not be used. 	In case the atof, atoi and atoll functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.11	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	<ul style="list-style-type: none"> • The macro '<name>' shall not be used. • Identifier XX should not be used. 	In case the abort, exit, getenv and system functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.12	The time handling functions of library <time.h> shall not be used.	<ul style="list-style-type: none"> • The macro '<name>' shall not be used. • Identifier XX should not be used. 	In case the time handling functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated

Runtime Failures

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
21.1	Minimization of runtime failures shall be ensured by the use of at least one of: <ul style="list-style-type: none"> • static verification tools/techniques; • dynamic verification tools/techniques; • explicit coding of checks to handle runtime faults. 		Done by Polyspace

MISRA C:2004 Rules Not Checked

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 “**Comments**” column describes the reason each rule is not checked.

Environment

Rule	Description	Comments
1.2 (Required)	No reliance shall be placed on undefined or unspecified behavior	Not statically checkable unless the data dynamic properties is taken into account
1.3 (Required)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the language/compilers/assemblers conform.	It is a process rule method.
1.4 (Required)	The compiler/linker/Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	The documentation of compiler must be checked.
1.5 (Advisory)	Floating point implementations should comply with a defined floating point standard.	The documentation of compiler must be checked as this implementation is done by the compiler

Language Extensions

Rule	Description	Comments
2.4 (Advisory)	Sections of code should not be “commented out”	It might be some pseudo code or code that does not compile inside a comment.

Documentation

Rule	Description	Comments
3.1 (Required)	All usage of implementation-defined behavior shall be documented.	The documentation of compiler must be checked. Error detection is based on undefined behavior, according to choices made for implementation-defined constructions. Documentation can not be checked.
3.2 (Required)	The character set and the corresponding encoding shall be documented.	The documentation of compiler must be checked.
3.3 (Advisory)	The implementation of integer division in the chosen compiler should be determined, documented and taken into account.	The documentation of compiler must be checked.

Rule	Description	Comments
3.5 (Required)	The implementation-defined behavior and packing of bitfields shall be documented if being relied upon.	The documentation of compiler must be checked.
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.	The documentation of compiler must be checked.

Structures and Unions

Rule	Description	Comments
18.3 (Required)	An area of memory shall not be reused for unrelated purposes.	"purpose" is functional design issue.

Polyspace MISRA C++ Checker

The Polyspace MISRA C++ checker helps you comply with the MISRA C++:2008 coding standard.¹¹

When MISRA C++ rules are violated, the Polyspace 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 185 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 12-60.

Note The Polyspace MISRA C++ checker is based on MISRA C++:2008 – “Guidelines for the use of the C++ language in critical systems.” For more information on these coding standards, see <http://www.misra-cpp.com>.

11. MISRA is a registered trademark of MISRA 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 12-60
“SQO Subset 2 – Indirect Impact on Selectivity” on page 12-63

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.

MISRA C++ Rule	Description
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
7-5-4	Functions should not call themselves, either directly or indirectly.
8-4-1	Functions shall not be defined using the ellipsis notation.
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.

MISRA C++ Rule	Description
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-3	An empty throw (throw;) shall only be used in the compound-statement of a catch handler.
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
18-4-1	Dynamic heap memory allocation shall not be used.

SQO Subset 2 – Indirect Impact on Selectivity

Good design practices generally lead to less code complexity, which can improve the selectivity of your results. The following set of coding rules may help to address design issues that impact selectivity.

Note When you specify SQO-subset2 for your MISRA C++ rules configuration, the software checks the rules listed in SQO Subset 1 *and* SQO Subset 2.

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.
3-9-2	typedefs that indicate size and signedness should be used in place of the basic numerical types.
3-9-3	The underlying bit representations of floating-point values shall not be used.
4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, , !, the equality operators == and !=, the unary & operator, and the conditional operator.
5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.
5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions.
5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression.

MISRA C++ Rule	Description
5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.
5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.
5-0-10	If the bitwise operators <code>~</code> and <code><<</code> are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.
5-0-13	
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	<code>></code> , <code>>=</code> , <code><</code> , <code><=</code> shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-2-1	Each operand of a logical <code>&&</code> or <code> </code> shall be a postfix - expression.
5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of <code>dynamic_cast</code> .
5-2-5	A cast shall not remove any <code>const</code> or <code>volatile</code> qualification from the type of a pointer or reference.
5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.
5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
5-2-11	The comma operator, <code>&&</code> operator and the <code> </code> operator shall not be overloaded.

MISRA C++ Rule	Description
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
5-3-3	The unary & operator shall not be overloaded.
5-18-1	The comma operator shall not be used.
6-2-1	Assignment operators shall not be used in sub-expressions.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.
6-4-2	All if ... else if constructs shall be terminated with an else clause.
6-4-6	The final clause of a switch statement shall be the default-clause.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.

MISRA C++ Rule	Description
6-6-5	A function shall have a single point of exit at the end of the function.
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
7-5-4	Functions should not call themselves, either directly or indirectly.
8-4-1	Functions shall not be defined using the ellipsis notation.
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.

MISRA C++ Rule	Description
12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.
12-8-2	The copy assignment operator shall be declared protected or private in an abstract class.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

MISRA C++ Rule	Description
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 12-69
“MISRA C++ Rules Not Checked” on page 12-89

Supported MISRA C++ Coding Rules

- “Language Independent Issues” on page 12-70
- “General” on page 12-70
- “Lexical Conventions” on page 12-70
- “Basic Concepts” on page 12-72
- “Standard Conversions” on page 12-73
- “Expressions” on page 12-74
- “Statements” on page 12-77
- “Declarations” on page 12-80
- “Declarators” on page 12-81
- “Classes” on page 12-82
- “Derived Classes” on page 12-82
- “Member Access Control” on page 12-83
- “Special Member Functions” on page 12-83
- “Templates” on page 12-84
- “Exception Handling” on page 12-85
- “Preprocessing Directives” on page 12-86
- “Library Introduction” on page 12-88
- “Language Support Library” on page 12-88
- “Diagnostic Library” on page 12-89
- “Input/output Library” on page 12-89

Language Independent Issues

N.	MISRA Definition	Comments
0-1-1	A project shall not contain unreachable code.	
0-1-2	A project shall not contain infeasible paths.	
0-1-7	The value returned by a function having a non- void return type that is not an overloaded operator shall always be used.	
0-1-10	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.

General

N.	MISRA Definition	Comments
1-0-1	All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".	

Lexical Conventions

N.	MISRA Definition	Comments
2-3-1	Trigraphs shall not be used.	
2-5-1	Digraphs should not be used.	
2-7-1	The character sequence /* shall not be used within a C-style comment.	This rule cannot be annotated in the source code.
2-10-1	Different identifiers shall be typographically unambiguous.	

N.	MISRA Definition	Comments
2-10-2	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.
2-10-3	A typedef name (including qualification, if any) shall be a unique identifier.	No detection across namespaces.
2-10-4	A class, union or enum name (including qualification, if any) shall be a unique identifier.	No detection across namespaces.
2-10-5	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.
2-10-6	If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.	If the identifier is a function and the function is both declared and defined then the violation is reported only once.
2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.	
2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.	
2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.	
2-13-4	Literal suffixes shall be upper case.	
2-13-5	Narrow and wide string literals shall not be concatenated.	

Basic Concepts

N.	MISRA Definition	Comments
3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.	
3-1-2	Functions shall not be declared at block scope.	
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.	
3-2-1	All declarations of an object or function shall have compatible types.	
3-2-2	The One Definition Rule shall not be violated.	Report type, template, and inline function defined in source file
3-2-3	A type, object or function that is used in multiple translation units shall be declared in one and only one file.	
3-2-4	An identifier with external linkage shall have exactly one definition.	
3-3-1	Objects or functions with external linkage shall be declared in a header file.	
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.	
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.	
3-9-1	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.	Comparison is done between current declaration and last seen declaration.

N.	MISRA Definition	Comments
3-9-2	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	The underlying bit representations of floating-point values shall not be used.	

Standard Conversions

N.	MISRA Definition	Comments
4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, , !, the equality operators == and !=, the unary & operator, and the conditional operator.	
4-5-2	Expressions with type enum shall not be used as operands to built-in operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.	
4-5-3	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator. N	

Expressions

N.	MISRA Definition	Comments
5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.	
5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions.	
5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type.	Assumes that <code>ptrdiff_t</code> is signed integer
5-0-4	An implicit integral conversion shall not change the signedness of the underlying type.	Assumes that <code>ptrdiff_t</code> 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	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	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	There shall be no explicit floating-integral conversions of a cvalue expression.	
5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.	
5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.	

N.	MISRA Definition	Comments
5-0-10	If the bitwise operators <code>~</code> and <code><<</code> are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	
5-0-11	The plain char type shall only be used for the storage and use of character values.	For numeric data, use a type which has explicit signedness.
5-0-12	Signed char and unsigned char type shall only be used for the storage and use of numeric values.	
5-0-14	The first operand of a conditional-operator shall have type bool.	
5-0-15	Array indexing shall be the only form of pointer arithmetic.	Warning on operations on pointers. (<code>p+I</code> , <code>I+p</code> and <code>p-I</code> , where <code>p</code> is a pointer and <code>I</code> an integer, <code>p[i]</code> accepted).
5-0-18	<code>></code> , <code>>=</code> , <code><</code> , <code><=</code> shall not be applied to objects of pointer type, except where they point to the same array.	Report when relational operator are used on pointers types (casts ignored).
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.	
5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type.	
5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type.	
5-2-1	Each operand of a logical <code>&&</code> or <code> </code> shall be a postfix - expression.	During preprocessing, violations of this rule are detected on the expressions in <code>#if</code> directives. Allowed exception on associativity (<code>a && b && c</code>), (<code>a b c</code>).
5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of <code>dynamic_cast</code> .	

N.	MISRA Definition	Comments
5-2-3	Casts from a base class to a derived class should not be performed on polymorphic types.	
5-2-4	C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.	
5-2-5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.	
5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.	No violation if pointer types of operand and target are identical.
5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.	"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."
5-2-8	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	A cast should not convert a pointer type to an integral type.	
5-2-10	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	
5-2-11	The comma operator, && operator and the operator shall not be overloaded.	
5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer.	

N.	MISRA Definition	Comments
5-3-1	Each operand of the ! operator, the logical && or the logical operators shall have type bool.	
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	
5-3-3	The unary & operator shall not be overloaded.	
5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects.	No warning on volatile accesses and function calls
5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	
5-14-1	The right hand operand of a logical && or operator shall not contain side effects.	No warning on volatile accesses and function calls.
5-18-1	The comma operator shall not be used.	
5-19-1	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	

Statements

N.	MISRA Definition	Comments
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.	

N.	MISRA Definition	Comments
6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character.	
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.	
6-4-1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.	
6-4-2	All if ... else if constructs shall be terminated with an else clause.	Detects also 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	A switch statement shall be a well-formed switch statement.	Return statements are considered as jump statements.
6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	
6-4-5	An unconditional throw or break statement shall terminate every non - empty switch-clause.	
6-4-6	The final clause of a switch statement shall be the default-clause.	
6-4-7	The condition of a switch statement shall not have bool type.	
6-4-8	Every switch statement shall have at least one case-clause.	

N.	MISRA Definition	Comments
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.	Detect only direct assignments if for_index is known (see 6-5-1).
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.	
6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression.	
6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.	
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.	
6-6-2	The goto statement shall jump to a label declared later in the same function body.	
6-6-3	The continue statement shall only be used within a well-formed for loop.	Assumes 6.5.1 to 6.5.6: so it is implemented only for supported 6_5_x rules.
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.	At most one return not necessarily as last statement for void functions.

Declarations

N.	MISRA Definition	Comments
7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations.	
7-3-2	The identifier main shall not be used for a function other than the global function main.	
7-3-3	There shall be no unnamed namespaces in header files.	
7-3-4	using-directives shall not be used.	
7-3-5	Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.	
7-3-6	using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.	
7-4-2	Assembler instructions shall only be introduced using the asm declaration.	
7-4-3	Assembly language shall be encapsulated and isolated.	
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.	

N.	MISRA Definition	Comments
7-5-3	A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.	
7-5-4	Functions should not call themselves, either directly or indirectly.	

Declarators

N.	MISRA Definition	Comments
8-0-1	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or	
8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.	
8-4-1	Functions shall not be defined using the ellipsis notation.	
8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.	
8-4-3	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	
8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.	
8-5-1	All variables shall have a defined value before they are used.	Non-initialized variable in results and error messages for obvious cases

N.	MISRA Definition	Comments
8-5-2	Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.	
8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	

Classes

N.	MISRA Definition	Comments
9-3-1	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	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	Unions shall not be used.	
9-6-2	Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.	
9-6-3	Bit-fields shall not have enum type.	
9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit.	

Derived Classes

N.	MISRA Definition	Comments
10-1-1	Classes should not be derived from virtual bases.	
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.	Assumes 10.1.1 not required

N.	MISRA Definition	Comments
10-1-3	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.	
10-2-1	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	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	Each overriding virtual function shall be declared with the virtual keyword.	
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.	

Member Access Control

N.	MISRA Definition	Comments
11-0-1	Member data in non- POD class types shall be private.	

Special Member Functions

N.	MISRA Definition	Comments
12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.	
12-1-2	All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.	

N.	MISRA Definition	Comments
12-1-3	All constructors that are callable with a single argument of fundamental type shall be declared explicit.	
12-8-1	A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member.	
12-8-2	The copy assignment operator shall be declared protected or private in an abstract class.	

Templates

N.	MISRA Definition	Comments
14-5-2	A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.	
14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.	
14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->	
14-6-2	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.	
14-7-3	All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.	

N.	MISRA Definition	Comments
14-8-1	Overloaded function templates shall not be explicitly specialized.	All specializations of overloaded templates are rejected even if overloading occurs after the call.
14-8-2	The viable function set for a function call should either contain no function specializations, or only contain function specializations.	

Exception Handling

N.	MISRA Definition	Comments
15-0-2	An exception object should not have pointer type.	NULL not detected (see 15-1-2).
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.	
15-1-2	NULL shall not be thrown explicitly.	
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.	
15-3-2	There should be at least one exception handler to catch all otherwise unhandled exceptions.	Detect that there is no try/catch in the main and that the catch does not handle all exceptions. Not detected if no "main".
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.	

N.	MISRA Definition	Comments
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.	Limit detection to throw and catch that are internals to the destructor; rethrows are partially processed; no detections in nested handlers.
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).	Limit detection to throw that are internals to the function; rethrows are partially processed; no detections in nested handlers.

Preprocessing Directives

N.	MISRA Definition	Comments
16-0-1	#include directives in a file shall only be preceded by other preprocessor directives or comments.	
16-0-2	Macros shall only be #define 'd or #undef 'd in the global namespace.	
16-0-3	#undef shall not be used.	
16-0-4	Function-like macros shall not be defined.	

N.	MISRA Definition	Comments
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	
16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.	
16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.	
16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.	
16-1-1	The defined preprocessor operator shall only be used in one of the two standard forms.	
16-1-2	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	
16-2-1	The 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.
16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.	
16-2-3	Include guards shall be provided.	
16-2-4	The ', ", /* or // characters shall not occur in a header file name.	
16-2-5	The \ character should not occur in a header file name.	
16-2-6	The #include directive shall be followed by either a <filename> or "filename" sequence.	

N.	MISRA Definition	Comments
16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.	
16-3-2	The # and ## operators should not be used.	

Library Introduction

N.	MISRA Definition	Comments
17-0-1	Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.	
17-0-2	The names of standard library macros and objects shall not be reused.	
17-0-5	The setjmp macro and the longjmp function shall not be used.	

Language Support Library

N.	MISRA Definition	Comments
18-0-1	The C library shall not be used.	
18-0-2	The library functions atof, atoi and atol from library <stdlib> shall not be used.	
18-0-3	The library functions abort, exit, getenv and system from library <stdlib> shall not be used.	The option -dialect iso must be used to detect violations (e.g.:exit).
18-0-4	The time handling functions of library <ctime> shall not be used.	
18-0-5	The unbounded functions of library <cstring> shall not be used.	
18-2-1	The macro offsetof shall not be used.	

N.	MISRA Definition	Comments
18-4-1	Dynamic heap memory allocation shall not be used.	
18-7-1	The signal handling facilities of <csignal> shall not be used.	

Diagnostic Library

N.	MISRA Definition	Comments
19-3-1	The error indicator errno shall not be used.	

Input/output Library

N.	MISRA Definition	Comments
27-0-1	The stream input/output library <cstdio> shall not be used.	

MISRA C++ Rules Not Checked

- “Language Independent Issues” on page 12-90
- “General” on page 12-91
- “Lexical Conventions” on page 12-91
- “Standard Conversions” on page 12-92
- “Expressions” on page 12-92
- “Declarations” on page 12-92
- “Classes” on page 12-93
- “Templates” on page 12-93
- “Exception Handling” on page 12-94
- “Preprocessing Directives” on page 12-94

- “Library Introduction” on page 12-94

Language Independent Issues

N.	MISRA Definition	Comments
0-1-3	A project shall not contain unused variables.	
0-1-4	A project shall not contain non-volatile POD variables having only one use.	
0-1-5	A project shall not contain unused type declarations.	
0-1-6	A project shall not contain instances of non-volatile variables being given values that are never subsequently used.	
0-1-8	All functions with void return type shall have external side effects.	
0-1-9	There shall be no dead code.	Not checked by the coding rules checker. Can be enforced through detection of dead code during analysis.
0-1-11	There shall be no unused parameters (named or unnamed) in nonvirtual functions.	
0-1-12	There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.	
0-2-1	An object shall not be assigned to an overlapping object.	
0-3-1	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.	MISRA Definition	Comments
0-3-2	If a function generates error information, then that error information shall be tested.	
0-4-1	Use of scaled-integer or fixed-point arithmetic shall be documented.	
0-4-2	Use of floating-point arithmetic shall be documented.	
0-4-3	Floating-point implementations shall comply with a defined floating-point standard.	

General

N.	MISRA Definition	Comments
1-0-2	Multiple compilers shall only be used if they have a common, defined interface.	
1-0-3	The implementation of integer division in the chosen compiler shall be determined and documented.	

Lexical Conventions

N.	MISRA Definition	Comments
2-2-1	The character set and the corresponding encoding shall be documented.	
2-7-2	Sections of code shall not be "commented out" using C-style comments.	
2-7-3	Sections of code should not be "commented out" using C++ comments.	

Standard Conversions

N.	MISRA Definition	Comments
4-10-1	ULL shall not be used as an integer value.	
4-10-2	Literal zero (0) shall not be used as the null-pointer-constant.	

Expressions

N.	MISRA Definition	Comments
5-0-13	The condition of an if-statement and the condition of an iteration- statement shall have type bool.	
5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	
5-0-17	Subtraction between pointers shall only be applied to pointers that address elements of the same array.	
5-17-1	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.	

Declarations

N.	MISRA Definition	Comments
7-1-1	A variable which is not modified shall be const qualified.	
7-1-2	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.	

N.	MISRA Definition	Comments
7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	
7-4-1	All usage of assembler shall be documented.	

Classes

N.	MISRA Definition	Comments
9-3-3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.	
9-6-1	When the absolute positioning of bits representing a bit-field is required, then the behavior and packing of bit-fields shall be documented.	

Templates

N.	MISRA Definition	Comments
14-5-1	A non-member generic function shall only be declared in a namespace that is not an associated namespace.	
14-7-1	All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.	
14-7-2	For any given template specialization, an explicit instantiation of the template with the template-arguments used in the specialization shall not render the program ill-formed.	

Exception Handling

N.	MISRA Definition	Comments
15-0-1	Exceptions shall only be used for error handling.	
15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	
15-3-1	Exceptions shall be raised only after start-up and before termination of the program.	
15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to	
15-5-3	The terminate() function shall not be called implicitly.	

Preprocessing Directives

N.	MISRA Definition	Comments
16-6-1	All uses of the #pragma directive shall be documented.	

Library Introduction

N.	MISRA Definition	Comments
17-0-3	The names of standard library functions shall not be overridden.	
17-0-4	All library code shall conform to MISRA C++.	

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 JSF program. They are designed to improve the robustness of C++ code, and improve maintainability.

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. For more information on these coding standards, see http://www.jsf.mil/downloads/documents/JSF_AV_C++_Coding_Standards_Rev_C.doc.

JSF C++ Coding Rules

In this section...
“Supported JSF C++ Coding Rules” on page 12-96
“JSF++ Rules Not Checked” on page 12-121

Supported JSF C++ Coding Rules

- “Code Size and Complexity” on page 12-97
- “Environment” on page 12-97
- “Libraries” on page 12-98
- “Pre-Processing Directives” on page 12-99
- “Header Files” on page 12-100
- “Style” on page 12-100
- “Classes” on page 12-104
- “Namespaces” on page 12-108
- “Templates” on page 12-108
- “Functions” on page 12-108
- “Comments” on page 12-110
- “Declarations and Definitions” on page 12-110
- “Initialization” on page 12-111
- “Types” on page 12-112
- “Constants” on page 12-112
- “Variables” on page 12-112
- “Unions and Bit Fields” on page 12-113
- “Operators” on page 12-113
- “Pointers and References” on page 12-115
- “Type Conversions” on page 12-116

- “Flow Control Standards” on page 12-117
- “Expressions” on page 12-119
- “Memory Allocation” on page 12-120
- “Fault Handling” on page 12-120
- “Portable Code” on page 12-120

Code Size and Complexity

N.	JSF++ Definition	Comments
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	Comments
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 -dialect iso

N.	JSF++ Definition	Comments
13	Multi-byte characters and wide string literals will not be used.	Report L'c' and 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	Comments
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.

N.	JSF++ Definition	Comments
24	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> from library <code><stdlib.h></code> shall not be used.	<code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> should not be used as a macro or a global with external "C" linkage.
25	The time handling functions of library <code><time.h></code> shall not be used.	<code>clock</code> , <code>difftime</code> , <code>mktime</code> , <code>asctime</code> , <code>ctime</code> , <code>gmtime</code> , <code>localtime</code> and <code>strftime</code> should not be used as a macro or a global with external "C" linkage.

Pre-Processing Directives

N.	JSF++ Definition	Comments
26	Only the following preprocessor directives shall be used: <code>#ifndef</code> , <code>#define</code> , <code>#endif</code> , <code>#include</code> .	
27	<code>#ifndef</code> , <code>#define</code> and <code>#endif</code> will be used to prevent multiple inclusions of the same header file. Other techniques to prevent the multiple inclusions of header files will not be used.	Detects the patterns <code>#if !defined</code> , <code>#pragma once</code> , <code>#ifdef</code> , and missing <code>#define</code> .
28	The <code>#ifndef</code> and <code>#endif</code> preprocessor directives will only be used as defined in AV Rule 27 to prevent multiple inclusions of the same header file.	Detects any use that does not comply with AV Rule 27. Assuming 35/27 is not violated, reports only <code>#ifndef</code> .
29	The <code>#define</code> preprocessor directive shall not be used to create inline macros. Inline functions shall be used instead.	Rule is split into two parts: the definition of a macro function (29.def) and the call of a macrofunction (29.use). Messages in report file: <ul style="list-style-type: none"> • 29.1 : The <code>#define</code> preprocessor directive shall not be used to create inline macros. • 29.2 : Inline functions shall be used instead of inline macros

N.	JSF++ Definition	Comments
30	The <code>#define</code> preprocessor directive shall not be used to define constant values. Instead, the <code>const</code> qualifier shall be applied to variable declarations to specify constant values.	Reports <code>#define</code> of simple constants.
31	The <code>#define</code> preprocessor directive will only be used as part of the technique to prevent multiple inclusions of the same header file.	Detects use of <code>#define</code> that are not used to guard for multiple inclusion, assuming that rules 35 and 27 are not violated.
32	The <code>#include</code> preprocessor directive will only be used to include header (<code>*.h</code>) files.	

Header Files

N.	JSF++ Definition	Comments
33	The <code>#include</code> directive shall use the <code><filename.h></code> notation to include header files.	
35	A header file will contain a mechanism that prevents multiple inclusions of itself.	
39	Header files (<code>*.h</code>) will not contain non-const variable definitions or function definitions.	Reports definitions of global variables / function in header.

Style

N.	JSF++ Definition	Comments
40	Every implementation file shall include the header files that uniquely define the inline functions, types, and templates used.	Reports when type, template, or inline function is defined in source file.
41	Source lines will be kept to a length of 120 characters or less.	

N.	JSF++ Definition	Comments
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.	
47	Identifiers will not begin with the underscore character '_'.	
48	<p>Identifiers will not differ by:</p> <ul style="list-style-type: none"> • Only a mixture of case • The presence/absence of the underscore character • The interchange of the letter 'O'; with the number '0' or the letter 'D' • The interchange of the letter 'I'; with the number '1' or the letter 'l' • The interchange of the letter 'S' with the number '5' • The interchange of the letter 'Z' with the number 2 • The interchange of the letter 'n' with the letter 'h' 	<p>Checked regardless of scope. Not checked between macros and other identifiers.</p> <p>Messages in report file:</p> <ul style="list-style-type: none"> • Identifier "Idf1" (file1.cpp line 11 column c1) and "Idf2" (file2.h line 12 column c2) only differ by the presence/absence of the underscore character. • Identifier "Idf1" (file1.cpp line 11 column c1) and "Idf2" (file2.h line 12 column c2) only differ by a mixture of case. • Identifier "Idf1" (file1.cpp line 11 column c1) and "Idf2" (file2.h line 12 column c2) only differ by letter 'O', with the number '0'.

N.	JSF++ Definition	Comments
50	The first word of the name of a class, structure, namespace, enumeration, or type created with <code>typedef</code> will begin with an uppercase letter. All others letters will be lowercase.	Messages in report file: <ul style="list-style-type: none"> • The first word of the name of a class will begin with an uppercase letter. • The first word of the namespace of a class will begin with an uppercase letter.
51	All letters contained in function and variables names will be composed entirely of lowercase letters.	Messages in report file: <ul style="list-style-type: none"> • All letters contained in variable names will be composed entirely of lowercase letters. • All letters contained in function names will be composed entirely of lowercase letters.
52	Identifiers for constant and enumerator values shall be lowercase.	Messages in report file: <ul style="list-style-type: none"> • Identifier for enumerator value shall be lowercase. • Identifier for template constant parameter shall be lowercase.
53	Header files will always have file name extension of ".h".	.H is allowed if you set the option -dos.
53.1	The following character sequences shall not appear in header file names: ', \, /*, //, or ".	
54	Implementation files will always have a file name extension of ".cpp".	Not case sensitive if you set the option -dos.
57	The public, protected, and private sections of a class will be declared in that order.	

N.	JSF++ Definition	Comments
58	When declaring and defining functions with more than two parameters, the leading parenthesis and the first argument will be written on the same line as the function name. Each additional argument will be written on a separate line (with the closing parenthesis directly after the last argument).	Detects that two parameters are not on the same line, The first parameter should be on the same line as function name. Does not check for the closing parenthesis.
59	The statements forming the body of an if, else if, else, while, do ... while or for statement shall always be enclosed in braces, even if the braces form an empty block.	<p>Messages in report file:</p> <ul style="list-style-type: none"> • The statements forming the body of an if statement shall always be enclosed in braces. • The statements forming the body of an else statement shall always be enclosed in braces. • The statements forming the body of a while statement shall always be enclosed in braces. • The statements forming the body of a do ... while statement shall always be enclosed in braces. • The statements forming the body of a for statement shall always be enclosed in braces.
60	Braces ("{}") which enclose a block will be placed in the same column, on separate lines directly before and after the block.	Detects that statement-block braces should be in the same columns.
61	Braces ("{}") which enclose a block will have nothing else on the line except comments.	

N.	JSF++ Definition	Comments
62	The dereference operator '*' and the address-of operator '&' will be directly connected with the type-specifier.	Reports when there is a space between type and "*" "&" for variables, parameters and fields declaration.
63	Spaces will not be used around '.' or '->', nor between unary operators and operands.	<p>Reports when the following characters are not directly connected to a white space:</p> <ul style="list-style-type: none"> • . • -> • ! • ~ • - • ++ • — <hr/> <p>Note A violation will be reported for "." used in float/double definition.</p> <hr/>

Classes

N.	JSF++ Definition	Polyspace Comments
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.

N.	JSF++ Definition	Polyspace Comments
74	Initialization of nonstatic class members will be performed through the member initialization list rather than through assignment in the body of a constructor.	<p>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:</p> <p>Initialization of nonstatic class members "<field>" will be performed through the member initialization list.</p>
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.	<p>Messages in report file:</p> <ul style="list-style-type: none"> • no copy constructor and no copy assign • no copy constructor • no copy assign
77.1	The definition of a member function shall not contain default arguments that produce a signature identical to that of the implicitly-declared copy constructor for the corresponding class/structure.	Does not report when an explicit copy constructor exists.
78	All base classes with a virtual function shall define a virtual destructor.	
79	All resources acquired by a class shall be released by the class's destructor.	<p>Reports when the number of "new" called in a constructor is greater than the number of "delete" called in its destructor.</p> <hr/> <p>Note A violation is raised even if "new" is done in a "if/else".</p> <hr/>

N.	JSF++ Definition	Polyspace Comments
81	The assignment operator shall handle self-assignment correctly.	<p>Reports when copy assignment body does not begin with “if (this != arg)” A violation is not raised if an empty else statement follows the if, or the body contains only a return statement.</p> <p>A violation is raised when the if statement is followed by a statement other than the return statement.</p>
82	An assignment operator shall return a reference to <code>*this</code> .	<p>The following operators should return <code>*this</code> on method, and <code>*first_arg</code> on plain function.</p> <pre>operator= operator+= operator-= operator*= operator >>= operator <<= operator /= operator %= operator = operator &= operator ^= prefix operator++ prefix operator--</pre> <p>Does not report when no return exists.</p> <p>No special message if type does not match.</p> <p>Messages in report file:</p> <ul style="list-style-type: none"> • An assignment operator shall return a reference to <code>*this</code>. • An assignment operator shall return a reference to its first arg.

N.	JSF++ Definition	Polyspace Comments
83	An assignment operator shall assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).	Reports when a copy assignment does not assign all data members. In a derived class, it also reports when a copy assignment does not call inherited copy assignments.
88	Multiple inheritance shall only be allowed in the following restricted form: n interfaces plus m private implementations, plus at most one protected implementation.	<p>Messages in report file:</p> <ul style="list-style-type: none"> • Multiple inheritance on public implementation shall not be allowed: <code><public_base_class></code> is not an interface. • Multiple inheritance on protected implementation shall not be allowed : <code><protected_base_class_1></code> • <code><protected_base_class_2></code> are not interfaces.
88.1	A stateful virtual base shall be explicitly declared in each derived class that accesses it.	
89	A base class shall not be both virtual and nonvirtual in the same hierarchy.	
94	An inherited nonvirtual function shall not be redefined in a derived class.	<p>Does not report for destructor. Message in report file:</p> <p>Inherited nonvirtual function %s shall not be redefined in a derived class.</p>
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.

N.	JSF++ Definition	Polyspace Comments
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 Comments
98	Every nonlocal name, except <code>main()</code> , should be placed in some namespace.	
99	Namespaces will not be nested more than two levels deep.	

Templates

N.	JSF++ Definition	Polyspace Comments
104	A template specialization shall be declared before its use.	Reports the actual compilation error message.

Functions

N.	JSF++ Definition	Polyspace Comments
107	Functions shall always be declared at file scope.	
108	Functions with variable numbers of arguments shall not be used.	

N.	JSF++ Definition	Polyspace Comments
109	A function definition should not be placed in a class specification unless the function is intended to be inlined.	Reports when "inline" is not in the definition of a member function inside the class definition.
110	Functions with more than 7 arguments will not be used.	
111	A function shall not return a pointer or reference to a non-static local object.	Simple cases without alias effect detected.
113	Functions will have a single exit point.	Reports first return, or once per function.
114	All exit points of value-returning functions shall be through return statements.	
116	Small, concrete-type arguments (two or three words in size) should be passed by value if changes made to formal parameters should not be reflected in the calling function.	Report constant parameters references with <code>sizeof <= 2 * sizeof(int)</code> . Does not report for copy-constructor.
119	Functions shall not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	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 Comments
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 Comments
135	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	
136	Declarations should be at the smallest feasible scope.	<p>Reports when:</p> <ul style="list-style-type: none"> • A global variable is used in only one function. • A local variable is not used in a statement (<code>expr</code>, <code>return</code>, <code>init ...</code>) of the same level of its declaration (in the same block) or is not used in two sub-statements of its declaration. <hr/> <p>Note</p> <ul style="list-style-type: none"> • Non-used variables are reported. • Initializations at definition are ignored (not considered an access) <hr/>

N.	JSF++ Definition	Polyspace Comments
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
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 Comments
142	All variables shall be initialized before use.	Done with Non-initialized variable checks in the software.
144	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	This covers partial initialization.
145	In an enumerator list, the '=' construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	Generates one report for an enumerator list.

Types

N.	JSF++ Definition	Polyspace Comments
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 Comments
149	Octal constants (other than zero) shall not be used.	
150	Hexadecimal constants will be represented using all uppercase letters.	
151	Numeric values in code will not be used; symbolic values will be used instead.	Reports direct numeric constants (except integer/float value 1, 0) in expressions, non -const initializations, and switch cases. char constants are allowed. Does not report on templates non-type parameter.
151.1	A string literal shall not be modified.	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 Comments
152	Multiple variable declarations shall not be allowed on the same line.	

Unions and Bit Fields

N.	JSF++ Definition	Polyspace Comments
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 Comments
157	The right hand operand of a && or operator shall not contain side effects.	<p>Assumes rule 159 is not violated. Messages in report file:</p> <ul style="list-style-type: none"> • The right hand operand of a && operator shall not contain side effects. • The right hand operand of a operator shall not contain side effects.
158	The operands of a logical && or shall be parenthesized if the operands contain binary operators.	<p>Messages in report file:</p> <ul style="list-style-type: none"> • The operands of a logical && shall be parenthesized if the operands contain binary operators. • The operands of a logical shall be parenthesized if the operands contain binary operators.

N.	JSF++ Definition	Polyspace Comments
		Exception for: X Y Z , Z&&Y &&Z
159	Operators , &&, and unary & shall not be overloaded.	Messages in report file: <ul style="list-style-type: none"> • Unary operator & shall not be overloaded. • Operator shall not be overloaded. • Operator && shall not be overloaded.
160	An assignment expression shall be used only as the expression in an expression statement.	Only simple assignment, not +=, ++, etc.
162	Signed and unsigned values shall not be mixed in arithmetic or comparison operations.	
163	Unsigned arithmetic shall not be used.	
164	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the left-hand operand (inclusive).	
164.1	The left-hand operand of a right-shift operator shall not have a negative value.	Detects constant case +. Found by the software for dynamic cases.
165	The unary minus operator shall not be applied to an unsigned expression.	
166	The sizeof operator will not be used on expressions that contain side effects.	
168	The comma operator shall not be used.	

Pointers and References

N.	JSF++ Definition	
169	Pointers to pointers should be avoided when possible.	Reports second-level pointers, except for arguments of main.
170	More than 2 levels of pointer indirection shall not be used.	Only reports on variables/parameters.
171	Relational operators shall not be applied to pointer types except where both operands are of the same type and point to: <ul style="list-style-type: none"> • the same object, • the same function, • members of the same object, or • elements of the same array (including one past the end of the same array). 	Reports when relational operator are used on pointer types (casts ignored).
173	The address of an object with automatic storage shall not be assigned to an object which persists after the object has ceased to exist.	
174	The null pointer shall not be de-referenced.	Done with checks in software.
175	A pointer shall not be compared to NULL or be assigned NULL; use plain 0 instead.	Reports usage of NULL macro in pointer contexts.
176	A typedef will be used to simplify program syntax when declaring function pointers.	Reports non-typedef function pointers, or pointers to member functions for types of variables, fields, parameters. Returns type of function, cast, and exception specification.

Type Conversions

N.	JSF++ Definition	Comments
177	User-defined conversion functions should be avoided.	<p>Reports user defined conversion function, non-explicit constructor with one parameter or default value for others (even undefined ones). Does not report copy-constructor.</p> <p>Additional message for constructor case: This constructor should be flagged as "explicit".</p>
178	<p>Down casting (casting from base to derived class) shall only be allowed through one of the following mechanism:</p> <ul style="list-style-type: none"> • Virtual functions that act like dynamic casts (most likely useful in relatively simple cases). • Use of the visitor (or similar) pattern (most likely useful in complicated cases). 	Reports explicit down casting, <code>dynamic_cast</code> included. (Visitor patter does not have a special case.)
179	A pointer to a virtual base class shall not be converted to a pointer to a derived class.	Reports this specific down cast. Allows <code>dynamic_cast</code> .
180	Implicit conversions that may result in a loss of information shall not be used.	<p>Reports the following implicit casts :</p> <pre>integer => smaller integer unsigned => smaller or eq signed signed => smaller or eq un-signed integer => float float => integer</pre> <p>Does not report for cast to bool reports for implicit cast on constant done with the options <code>-scalar-overflows-checks signed-and-unsigned</code> or <code>-ignore-constant-overflows</code></p>

N.	JSF++ Definition	Comments
		.
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	Comments
186	There shall be no unreachable code.	Done with gray checks in the software.
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).	

N.	JSF++ Definition	Comments
192	All <code>if</code> , <code>else if</code> constructs will contain either a final <code>else</code> clause or a comment indicating why a final <code>else</code> clause is not necessary.	<code>else if</code> should contain an <code>else</code> clause.
193	Every non-empty <code>case</code> clause in a <code>switch</code> statement shall be terminated with a <code>break</code> statement.	
194	All <code>switch</code> statements that do not intend to test for every enumeration value shall contain a final <code>default</code> clause.	Reports only for missing <code>default</code> .
195	A <code>switch</code> expression will not represent a Boolean value.	
196	Every <code>switch</code> statement will have at least two cases and a potential <code>default</code> .	
197	Floating point variables shall not be used as loop counters.	Assumes 1 loop parameter.
198	The initialization expression in a <code>for</code> loop will perform no actions other than to initialize the value of a single <code>for</code> loop parameter.	Reports if loop parameter cannot be determined. Assumes Rule 200 is not violated. The loop variable parameter is assumed to be a variable.
199	The increment expression in a <code>for</code> loop will perform no action other than to change a single loop parameter to the next value for the loop.	Assumes 1 loop parameter (Rule 198), with non class type. Rule 200 must not be violated for this rule to be reported.
200	Null initialize or increment expressions in <code>for</code> loops will not be used; a <code>while</code> loop will be used instead.	
201	Numeric variables being used within a <code>for</code> loop for iteration counting shall not be modified in the body of the loop.	Assumes 1 loop parameter (AV rule 198), and no alias writes.

Expressions

N.	JSF++ Definition	Polyspace Comments
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	<p>A single operation with side-effects shall only be used in the following contexts:</p> <ul style="list-style-type: none"> • by itself • the right-hand side of an assignment • a condition • the only argument expression with a side-effect in a function call • condition of a loop • switch condition • single part of a chained operation 	<p>Reports when:</p> <ul style="list-style-type: none"> • A side effect is found in a return statement • A side effect exists on a single value, and only one operand of the function call has a side effect.
204.1	The value of an expression shall be the same under any order of evaluation that the standard permits.	<p>Reports when:</p> <ul style="list-style-type: none"> • Variable is written more than once in an expression • Variable is read and write in sub-expressions • Volatile variable is accessed more than once <hr/> <p>Note Read-write operations such as ++, are only considered as a write.</p> <hr/>
205	The volatile keyword shall not be used unless directly interfacing with hardware.	Reports if volatile keyword is used.

Memory Allocation

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

Portable Code

N.	JSF++ Definition	Comments
209	The basic types of int, short, long, float and double shall not be used, but specific-length equivalents should be typedef'd accordingly for each compiler, and these type names used in the code.	Only allows use of basic types through direct typedefs.
213	No dependence shall be placed on C++'s operator precedence rules, below arithmetic operators, in expressions.	Reports when a binary operation has one operand that is not parenthesized and is an operation with inferior precedence level. Reports bitwise and shifts operators that are used without parenthesis and binary operation arguments.
215	Pointer arithmetic will not be used.	Reports: p + I p - I p++ p-- p+= p-= Allows p[i].

JSF++ Rules Not Checked

- “Code Size and Complexity” on page 12-122
- “Rules” on page 12-122
- “Environment” on page 12-122
- “Libraries” on page 12-122
- “Header Files” on page 12-123
- “Style” on page 12-123
- “Classes” on page 12-123
- “Namespaces” on page 12-125
- “Templates” on page 12-125
- “Functions” on page 12-126
- “Comments” on page 12-127
- “Initialization” on page 12-127
- “Types” on page 12-127
- “Unions and Bit Fields” on page 12-128
- “Operators” on page 12-128
- “Type Conversions” on page 12-128
- “Expressions” on page 12-128
- “Memory Allocation” on page 12-129
- “Portable Code” on page 12-129
- “Efficiency Considerations” on page 12-129
- “Miscellaneous” on page 12-129
- “Testing” on page 12-130

Code Size and Complexity

N.	JSF++ Definition
2	There shall not be any self-modifying code.

Rules

N.	JSF++ Definition
4	To break a “should” rule, the following approval must be received by the developer: <ul style="list-style-type: none"> • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)
5	To break a “will” or a “shall” rule, the following approvals must be received by the developer: <ul style="list-style-type: none"> • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool) • approval from the software product manager (obtained by the unit approval in the developmental CM tool)
6	Each deviation from a “shall” rule shall be documented in the file that contains the deviation. Deviations from this rule shall not be allowed, AV Rule 5 notwithstanding.
7	Approval will not be required for a deviation from a “shall” or “will” rule that complies with an exception specified by that rule.

Environment

N.	JSF++ Definition
10	Values of character types will be restricted to a defined and documented subset of ISO 10646 1.

Libraries

N.	JSF++ Definition
16	Only DO-178B level A [15] certifiable or SEAL 1 C/C++ libraries shall be used with safety-critical (i.e. SEAL 1) code.

Header Files

N.	JSF++ Definition
34	Header files should contain logically related declarations only.
36	Compilation dependencies should be minimized when possible.
37	Header (include) files should include only those header files that are required for them to successfully compile. Files that are only used by the associated .cpp file should be placed in the .cpp file — not the .h file.
38	Declarations of classes that are only accessed via pointers (*) or references (&) should be supplied by forward headers that contain only forward declarations.

Style

N.	JSF++ Definition
45	All words in an identifier will be separated by the ‘_’ character.
49	All acronyms in an identifier will be composed of uppercase letters.
55	The name of a header file should reflect the logical entity for which it provides declarations.
56	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	<p>A member function that does not affect the state of an object (its instance variables) will be declared const.</p> <p>Member functions should be const by default. Only when there is a clear, explicit reason should the const modifier on member functions be omitted.</p>
70	<p>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.</p>
70.1	<p>An object shall not be improperly used before its lifetime begins or after its lifetime ends.</p>
71	<p>Calls to an externally visible operation of an object, other than its constructors, shall not be allowed until the object has been fully initialized.</p>
72	<p>The invariant for a class should be:</p> <ul style="list-style-type: none"> • A part of the postcondition of every class constructor, • A part of the precondition of the class destructor (if any), • A part of the precondition and postcondition of every other publicly accessible operation.
73	<p>Unnecessary default constructors shall not be defined.</p>
77	<p>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).</p>
80	<p>The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.</p>
84	<p>Operator overloading will be used sparingly and in a conventional manner.</p>
85	<p>When two operators are opposites (such as == and !=), both will be defined and one will be defined in terms of the other.</p>
86	<p>Concrete types should be used to represent simple independent concepts.</p>
87	<p>Hierarchies should be based on abstract classes.</p>
90	<p>Heavily used interfaces should be minimal, general and abstract.</p>
91	<p>Public inheritance will be used to implement “is-a” relationships.</p>

N.	JSF++ Definition
92	<p>A subtype (publicly derived classes) will conform to the following guidelines with respect to all classes involved in the polymorphic assignment of different subclass instances to the same variable or parameter during the execution of the system:</p> <ul style="list-style-type: none"> • Preconditions of derived methods must be at least as weak as the preconditions of the methods they override. • Postconditions of derived methods must be at least as strong as the postconditions of the methods they override. <p>In other words, subclass methods must expect less and deliver more than the base class methods they override. This rule implies that subtypes will conform to the Liskov Substitution Principle.</p>
93	<p>“has-a” or “is-implemented-in-terms-of” relationships will be modeled through membership or non-public inheritance.</p>

Namespaces

N.	JSF++ Definition
100	<p>Elements from a namespace should be selected as follows:</p> <ul style="list-style-type: none"> • using declaration or explicit qualification for few (approximately five) names, • using directive for many names.

Templates

N.	JSF++ Definition
101	<p>Templates shall be reviewed as follows:</p> <ol style="list-style-type: none"> 1 with respect to the template in isolation considering assumptions or requirements placed on its arguments. 2 with respect to all functions instantiated by actual arguments.
102	<p>Template tests shall be created to cover all actual template instantiations.</p>

N.	JSF++ Definition
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: <ul style="list-style-type: none"> • 117.1 – An object should be passed as <code>const T&</code> if the function should not change the value of the object. • 117.2 – An object should be passed as <code>T&</code> if the function may change the value of the object.
118	Arguments should be passed via pointers if NULL values are possible: <ul style="list-style-type: none"> • 118.1 – An object should be passed as <code>const T*</code> if its value should not be modified. • 118.2 – An object should be passed as <code>T*</code> if its value may be modified.
120	Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal
122	Trivial accessor and mutator functions should be inlined.
123	The number of accessor and mutator functions should be minimized.
124	Trivial forwarding functions should be inlined.
125	Unnecessary temporary objects should be avoided.

Comments

N.	JSF++ Definition
127	Code that is not used (commented out) shall be deleted. Note: This rule cannot be annotated in the source code.
128	Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented will not be allowed.
129	Comments in header files should describe the externally visible behavior of the functions or classes being documented.
130	The purpose of every line of executable code should be explained by a comment, although one comment may describe more than one line of code.
131	One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).
132	Each variable declaration, typedef, enumeration value, and structure member will be commented.
134	Assumptions (limitations) made by functions should be documented in the function's preamble.

Initialization

N.	JSF++ Definition
143	Variables will not be introduced until they can be initialized with meaningful values. (See also AV Rule 136, AV Rule 142, and AV Rule 73 concerning declaration scope, initialization before use, and default constructors respectively.)

Types

N.	JSF++ Definition
146	Floating point implementations shall comply with a defined floating point standard. The standard that will be used is the ANSI/IEEE Std 754 [1].

Unions and Bit Fields

N.	JSF++ Definition
155	Bit-fields will not be used to pack data into a word for the sole purpose of saving space.

Operators

N.	JSF++ Definition
167	The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.

Type Conversions

N.	JSF++ Definition
183	Every possible measure should be taken to avoid type casting.

Expressions

N.	JSF++ Definition
204	<p>A single operation with side-effects shall only be used in the following contexts:</p> <ol style="list-style-type: none">1 by itself2 the right-hand side of an assignment3 a condition4 the only argument expression with a side-effect in a function call5 condition of a loop6 switch condition7 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.

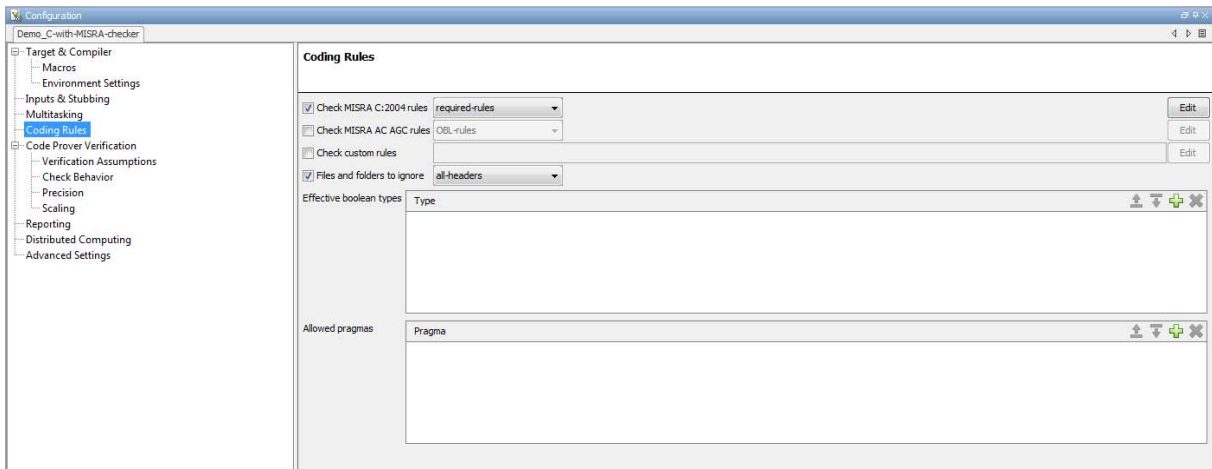
Checking Coding Rules

- “Activate Coding Rules Checker” on page 13-2
- “Select Specific MISRA or JSF Coding Rules” on page 13-6
- “Create Custom Coding Rules” on page 13-8
- “Format of Custom Coding Rules File” on page 13-10
- “Exclude Files from Rules Checking” on page 13-12
- “Allow Custom Pragma Directives” on page 13-13
- “Specify Boolean Types” on page 13-14
- “Review Coding Rule Violations” on page 13-15
- “Apply Coding Rule Violation Filters” on page 13-17
- “Generate Coding Rules Report” on page 13-18

Activate Coding Rules Checker

This example shows how to activate the coding rules checker before you start a verification. This activation enables Polyspace Code Prover to search for coding rule violations. You can view the coding rule violations in your verification results.

- 1 Open project configuration.
- 2 In the **Configuration** tree view, select **Coding Rules**.



- 3 Select the check box for the type of coding rules that you want to check.

For C code, you can check compliance with:

- MISRA C:2004
- MISRA AC AGC
- Custom coding rules

For C++ code, you can check compliance with:

- MISRA C++
- JSF C++
- Custom coding rules

- 4 For each rule type that you select, from the drop-down list, select the subset of rules to check.

MISRA C:2004

Option	Description
required-rules	All required MISRA C coding rules.
all-rules	All required and advisory MISRA C coding rules.
SQ0-subset1	A small subset of MISRA C rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
SQ0-subset2	A second subset of rules that include the rules in SQ0-subset1 and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
custom	A set of MISRA C coding rules that you specify.

MISRA AC AGC

Option	Description
OBL-rules	All required MISRA AC AGC coding rules.
OBL-REC-rules	All required and recommended MISRA AC AGC coding rules.
all-rules	All required, recommended, and readability coding rules.
SQ0-subset1	A small subset of MISRA AC AGC rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.

Option	Description
SQ0-subset2	A second subset of MISRA AC AGC rules that include the rules in SQ0-subset1 and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
custom	A set of MISRA AC AGC coding rules that you specify.

MISRA C++

Option	Description
required-rules	All required MISRA C++ coding rules.
all-rules	All required and advisory MISRA C++ coding rules.
SQ0-subset1	A small subset of MISRA C++ rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
SQ0-subset2	A second subset of rules with indirect impact on the selectivity in addition to SQ0-subset1. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
custom	A specified set of MISRA C++ coding rules.

JSF C++

Option	Description
shall-rules	Shall rules are mandatory requirements. These rules require verification.
shall-will-rules	All Shall and Will rules. Will rules are intended to be mandatory requirements. However, these rules do not require verification.

Option	Description
all-rules	All Shall , Will , and Should rules. Should rules are advisory rules.
custom	A set of JSF C++ coding rules that you specify.

- 5** If you select **Check custom rules**, specify the path to your custom rules file or click **Edit** to create one.

When rules checking is complete, the software displays the coding rule violations in purple on the **Results Summary** pane.

Related Examples

- “Select Specific MISRA or JSF Coding Rules” on page 13-6
- “Create Custom Coding Rules” on page 13-8
- “Exclude Files from Rules Checking” on page 13-12

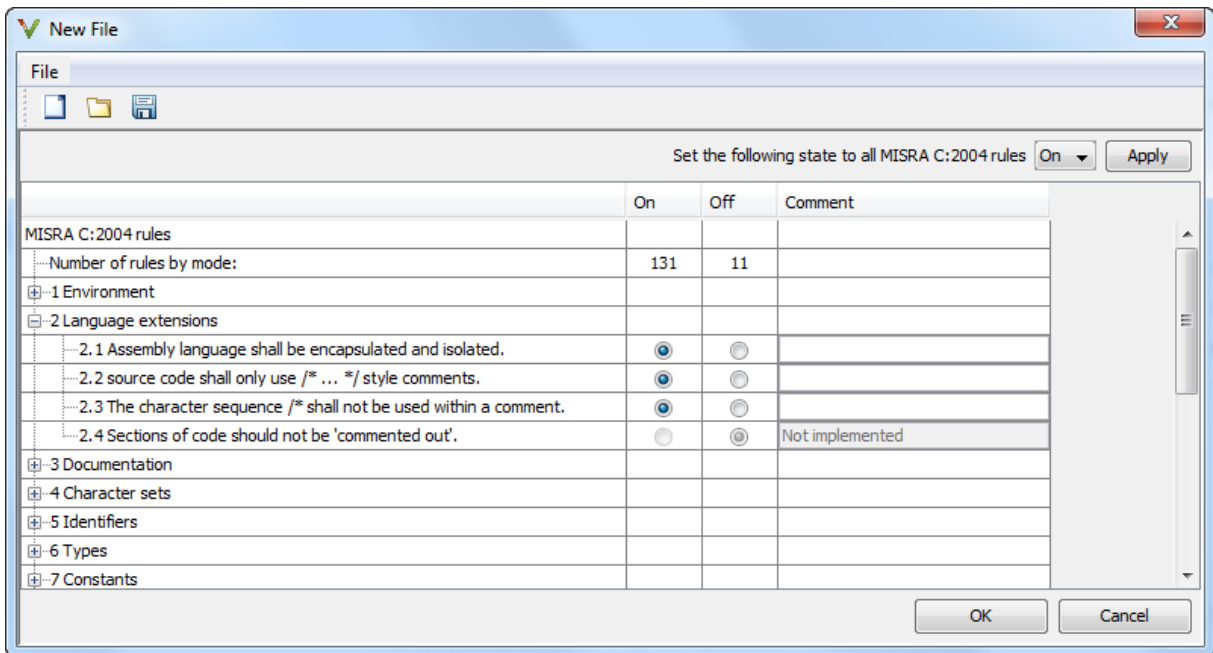
Concepts

- “Rule Checking” on page 12-2
- “Software Quality Objective Subsets (C)” on page 12-11
- “Software Quality Objective Subsets (AC AGC)” on page 12-16
- “Software Quality Objective Subsets (C++)” on page 12-60

Select Specific MISRA or JSF Coding Rules

This example shows how to specify a subset of MISRA or JSF rules for the coding rules checker. If you select **custom** from the MISRA or JSF drop-down list, you must provide a file that specifies the rules to check.

- 1 Open project configuration.
- 2 In the **Configuration** tree view, select **Coding Rules**.
- 3 Select the check box for the type of coding rules you wish to check
- 4 From the corresponding drop-down list, select **custom**. The software displays a new field for your custom file.
- 5 To the right of this field, click **Edit**. A New File window opens, displaying a table of rules.




Select **On** for the rules you want to check.

- 6 Click **OK** to save the rules and close the window.

The **Save as** dialog box opens.

- 7 In the **File** field, enter a name for the rules file.

- 8 Click **OK** to save the file and close the dialog box.

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.

Related Examples

- “Activate Coding Rules Checker” on page 13-2
- “Create Custom Coding Rules” on page 13-8

Concepts

- “Rule Checking” on page 12-2

Create Custom Coding Rules

This example shows how to create a custom coding rules file. You can use this file to check names or text patterns in your source code 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.

Save Example Code

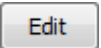
Save the following code in a file `printInitialValue.c`:

```
#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);
}
```

Create Coding Rules File

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

The New File window opens, displaying a table of rule groups.

- 4 From the drop-down list **Set the following state to all Custom C rules**, select **Off**. Click **Apply**.

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

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

Review Coding Rule Violations

- 1** Save the file and run the verification. On the **Results Summary** 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 **Check Details** pane, you see the error message you had entered, All struct fields must begin with s_ and have capital letters.
- 2** Right-click on the **Source** pane and select **Open Source File**. The file `printInitialValue.c` opens in a text editor.
- 3** In the file, replace all instances of `a` with `s_A` and `b` with `s_B`. Save the file and rerun the verification.

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

Related Examples

- “Activate Coding Rules Checker” on page 13-2
- “Select Specific MISRA or JSF Coding Rules” on page 13-6
- “Exclude Files from Rules Checking” on page 13-12

Concepts

- “Rule Checking” on page 12-2
- “Format of Custom Coding Rules File” on page 13-10

Format of Custom Coding Rules File

In a custom coding rules file, each rule appears in the following format:

```
N.n off|warning
convention=violation_message
pattern=regular_expression
```

- *N.n* — Custom rule number, for example, 1.2.
- *off* — Rule is not considered.
- *warning* — The software checks for violation of the rule. After verification, it displays the coding rule violation on the **Results Summary** 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 Naming Convention Rules” on page 12-3.

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 warning     # 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 warning     # Non-adherence to custom rule 9.1 produces a warning
convention=Global variables should begin by g_.
pattern=g_.*
```

Related Examples

- “Create Custom Coding Rules” on page 13-8

Exclude Files from Rules Checking

This example shows how to exclude certain files from coding rules checking.

- 1** Open project configuration.
- 2** In the **Configuration** tree view, select **Coding Rules**.
- 3** Select the **Files and folders to ignore** check box.
- 4** From the corresponding drop-down list, select one of the following:
 - **all-headers** (default) — Rule checker excludes folders that contain only header files, that is, folders without source files.
 - **all** — Rule checker excludes all include folders. For example, if you are checking a large code base with standard or Visual headers, excluding include folders can significantly improve the speed of code analysis.
 - **custom** — Rule checker excludes files or folders specified in the **File/Folder** view. To add files to the custom **File/Folder** list, select  to choose the files and folders to exclude. To remove a file or folder from the list of excluded files and folders, select the row. Then click .

Related Examples


- “Activate Coding Rules Checker” on page 13-2

Concepts


- “Rule Checking” on page 12-2

Allow Custom Pragma Directives

This example shows how to exclude custom pragma directives from coding rules checking. MISRA C 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.

- 1 Open project configuration.
- 2 In the **Configuration** tree view, select **Coding Rules**.
- 3 To the right of **Allowed pragmas**, click .

In the **Pragma** view, the software displays an active text field.

- 4 In the text field, enter a pragma directive.
- 5 To remove a directive from the **Pragma** list, select the directive. Then click .

Related Examples


- “Activate Coding Rules Checker” on page 13-2

Concepts


- “Rule Checking” on page 12-2

Specify 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 may affect the checking of MISRA C rules 12.6, 13.2, and 15.4.

- 1 Open project configuration.
- 2 In the **Configuration** tree view, select **Coding Rules**.
- 3 To the right of **Effective boolean types**, click .

In the **Type** view, the software displays an active text field.

- 4 In the text field, specify the data type that you want Polyspace to treat as Boolean.
- 5 To remove a data type from the **Type** list, select the data type. Then click .

Related Examples

- “Activate Coding Rules Checker” on page 13-2

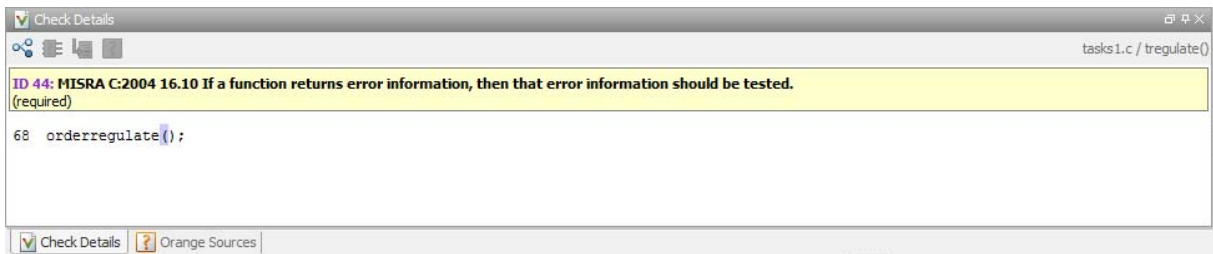
Concepts

- “Rule Checking” on page 12-2

Review Coding Rule Violations

This example shows how to review coding rule violations in the Results Manager perspective once code analysis is complete. After analysis, the **Results Summary** pane displays the rule violations with a

- ▼ symbol for predefined coding rules such as MISRA C:2004.
 - ▼ symbol for custom coding rules.
- 1 Select a coding-rule violation on the **Results Summary** pane.
 - The predefined rules such as MISRA C or C++ or JSF C++ are indicated by ▼.
 - The custom rules are indicated by ▼.
 - 2 On the **Check Details** pane, view the location and description of the violated rule. In the source code, the line containing the violation appears highlighted.



- 3 Review the violation. On the **Check Review** tab, select a **Classification** to describe the severity of the issue:
 - High
 - Medium
 - Low
 - Not a defect
- 4 Select a **Status** to describe how you intend to address the issue:
 - Fix

- Improve
- Investigate
- Justify with annotations
- No Action Planned
- Other
- Restart with different options
- Undecided

You can also define your own statuses.

- 5** In the comment box, enter additional information about the violation.
- 6** To open the source file that contains the coding rule violation, on the **Source** pane, right-click the code with the purple check. From the context menu, select **Open Source File**. The file opens in your text editor.
- 7** Fix the coding rule violation.
- 8** When you have corrected the coding rule violations, run the analysis again.

Related Examples

- “Activate Coding Rules Checker” on page 13-2
- “Apply Coding Rule Violation Filters” on page 13-17

Apply Coding Rule Violation Filters

This example shows how to use filters in the **Results Summary** pane to focus on specific kinds of coding rule violations. By default, the software displays all coding rule violations and run-time checks.

To filter violations by rule number:

- 1** On the **Results Summary** pane, place your cursor on the **Check** column header. Click the filter icon that appears.
- 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**.

Related Examples

- “Activate Coding Rules Checker” on page 13-2
- “Review Coding Rule Violations” on page 13-15

Generate Coding Rules Report

This example shows how to generate and view a coding rules report after verification.

Generate Report

- 1 In the Results Manager perspective, select **Run > Run Report > 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 In the Results Manager perspective, select **Run > Run Report > Open Report**.
- 2 In the Open Report dialog box, navigate to the folder that contains the coding rules report.

The default location is in *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 Summary** 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.
- “Activate Coding Rules Checker” on page 13-2

Related Examples

Software Quality with Polyspace Metrics

- “Software Quality with Polyspace Metrics” on page 14-3
- “Set Up Verification to Generate Metrics” on page 14-5
- “Open Polyspace Metrics” on page 14-12
- “Organize Polyspace Metrics Projects” on page 14-14
- “Protect Access to Project Metrics” on page 14-16
- “Web Browser Support” on page 14-18
- “What You Can Do with Polyspace Metrics” on page 14-19
- “Review Overall Progress” on page 14-20
- “Display Metrics for Single Project Version” on page 14-24
- “Create File Module and Specify Quality Level” on page 14-25
- “Compare Project Versions” on page 14-27
- “Review New Findings” on page 14-28
- “Review Results” on page 14-29
- “Save Review Comments” on page 14-31
- “Fix Defects” on page 14-32
- “Review Code Complexity” on page 14-33
- “Customize Software Quality Objectives” on page 14-34
- “SQO Levels” on page 14-36
- “Coding Rules Sets” on page 14-44

- “Run-Time Checks Sets” on page 14-48
- “Status Acronyms” on page 14-52
- “Code Metrics” on page 14-53
- “Run-Time Checks” on page 14-61
- “Number of Lines of Code Calculation” on page 14-63
- “Administer Results Repository” on page 14-64

Software Quality with Polyspace Metrics

Polyspace Metrics is a Web-based tool for software development managers, quality assurance engineers, and software developers, to do the following in software projects:

- Evaluate software quality metrics
- Monitor the variation of code metrics, coding rule violations, and run-time checks through the lifecycle of a project
- View defect numbers, run-time reliability of the software, review progress, and the status of the code with respect to software quality objectives.

If you are a development manager or a quality assurance engineer, through a Web browser, you can:

- View software quality information about your project. See “Open Polyspace Metrics” on page 14-12.
- Observe trends over time, by project or module. See “Review Overall Progress” on page 14-20.
- Compare metrics of one project version with those of another. See “Compare Project Versions” on page 14-27.

If you have the Polyspace product installed on your computer, you can drill down to coding rule violations and run-time checks in the Polyspace verification environment. This feature allows you to:

- Review coding rule violations
- Review run-time checks and, if required, classify these checks as defects

In addition, if you think that coding rule violations and run-time checks can be justified, you can mark them as justified and enter comments. See “Review Results” on page 14-29.

If you are a software developer, Polyspace Metrics allows you to focus on the latest version of the project that you are working on. You can use the view filters and drill-down functionality to go to code defects, which you can then fix. See “Fix Defects” on page 14-32.

Polyspace calculates metrics that are used to determine whether your code fulfills predefined software quality objectives. You can redefine these software quality objectives. See “Customize Software Quality Objectives” on page 14-34.

Set Up Verification to Generate Metrics

You can run, either manually or automatically, verifications that generate metrics. In each case, Polyspace uses a metrics computation engine to evaluate metrics for your code, and stores these metrics in a results repository.

Before you run a verification manually, in the Project Manager perspective:

- 1 Select the **Configuration > Distributed Computing** pane.
- 2 Select the **Batch** check box.
- 3 Select the **Add to results repository** check box.

To set up scheduled, automatic verification runs, see “Specify Automatic Verification” on page 14-5.

The software saves generated metrics in the following XML file:

Results_Folder/Polyspace-Doc/Code_Metrics.xml

See “Results Folder” on page 10-121.

Specify Automatic Verification

You can configure verifications to start automatically and periodically, for example, at a specific time every night. At the end of each verification, the software stores results in the repository and updates the project metrics. You can also configure the software to send you an email at the end of the verification. This email will contain:

- Links to results
- An attached log file if the verification produces compilation errors
- A summary of new findings, for example, new coding rule violations, and new potential and actual run-time errors

To configure automatic verification, you must create an XML file `Projects.pspproj` that has the following elements:

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

```
<!-- Polyspace Metrics Automatic Verification Project File -->
<Configuration>
  <Project>
    <Options>
    </Options>
    <LaunchingPeriod>
    </LaunchingPeriod>
    <Commands>
    </Commands>
    <Users>
      <User>
      </User>
    </Users>
  </Project>
  <SmtpConfiguration>
  </SmtpConfiguration>
</Configuration>
```

Configure the verification by providing data for the elements (and their attributes) within `Configuration`. See “Element and Attribute Data for `Projects.pspj`” on page 14-6.

After creating `Projects.pspj`, on the Polyspace Metrics server, place the file in the results repository. For example:

```
/var/Polyspace/results-repository
```

Element and Attribute Data for `Projects.pspj`

The following topics describe the data required to configure automatic verification.

Project. Specify the following attributes:

- `name` — Your project name.
- `language` — C or CPP.
- `verificationKind` — Mode, which is either `INTEGRATION` or `UNIT-BY-UNIT`.
- `product` — Product name, which is either `BUG-FINDER` or `CODE-PROVER`.

For example,

```
<Project name="Demo_C" language="C" verificationKind="INTEGRATION"
product="CODE-PROVER">
```

The Project element also contains the following elements:

- “Options” on page 14-7
- “LaunchingPeriod” on page 14-7
- “Commands” on page 14-8
- “Users” on page 14-9

Options. Specify a list of the Polyspace options required for your verification, with the exception of `-unit-by-unit`, `-results-dir`, `-prog` and `-verif-version`. If these four options are present, they are ignored.

The following is an example.

```
<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
  -includes-to-ignore sources/math.h
  -D NEW_DEFECT
</Options>
```

LaunchingPeriod. For the starting time of the verification, specify five attributes:

- `hour`. Any integer in the range 0–23.
- `minute`. Any integer in the range 0–59.
- `month`. Any integer in the range 1–12.

- **day.** Any integer in the range 1–31.
- **weekDay.** Any integer in the range 1–7, where 1 specifies Monday.

Use `*` to specify all values in range, for example, `month="*"` specifies a verification every month.

Use `-` to specify a range, for example, `weekDay="1-5"` specifies Monday to Friday.

You can also specify a list for each attribute. For example, `day="1,15"` specifies the first and the fifteenth day of the month.

Default: If you do not specify attribute data for `LaunchingPeriod`, then a verification is started each week day at midnight.

The following is an example.

```
<LaunchingPeriod hour="12" minute="20" month="*" weekDay="1-5">
```

Commands. You can provide a list of optional commands. There must be only one command per line, and these commands must be executable on the computer that starts the verification.

- **GetSource.** A command to retrieve source files from the configuration management system, or the file system of the user. 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 spooled to the Polyspace server.

For example:

```
<GetSource>
  cvs co r 1.4.6.4 myProject
  mkdir sources
  cp fvr myProject/*.c sources
</GetSource>
```

You can also use:

```
<GetSource>
  find / /myProject name *.cpp | tee sources_list.txt
```



```
</GetSource>
```

and add `-sources-list-file sources_list.txt` to the options list.

- **GetVersion.** A command to retrieve the version identifier of your project. Polyspace uses the version identifier as a parameter for `-verif-version`.

For example:

```
<GetVersion>
  cd / ../myProject ; cvs status Makefile 2>/dev/null | grep 'Sticky Tag:'
  | sed 's/Sticky Tag:/' | awk '{print $1"-"$3}' | sed 's/).*$/'
</GetVersion>
```

Users. A list of users, where each user is defined using the element “User” on page 14-9.

User. Define a user using three elements:

- **FirstName.** First name of user.
- **LastName.** Last name of user.
- **Mail.** Use the attributes `resultsMail` and `compilationFailureMail` to specify conditions for sending an email at the end of verification. Specify the email address in the element.
 - **resultsMail.** You can use any of the following values:
 - **ALWAYS.** Default. Email sent at the end of each automatic verification (even if the verification does not produce new run-time checks or coding rule violations).
 - **NEW-CERTAIN-FINDINGS.** Email sent only if verification produces new red, gray, NTC, or NTL checks.
 - **NEW-POTENTIAL-FINDINGS.** Email sent only if verification produces new orange check.
 - **NEW-CODING-RULES-FINDINGS.** Email sent only if verification produces new coding rule violation or warning.
 - **ALL-NEW-FINDINGS.** Email sent if verification produces a new run-time check or coding rule violation.

- `compilationFailureMail`. Either Yes (default) or No. If Yes, email sent when automatic verification fails because of a compilation failure.

The following is an example of Mail.

```
<Mail resultsMail="NEW-POTENTIAL-FINDINGS|NEW-CODING-RULES-FINDINGS"
compilationFailureMail="yes">
  user_id@yourcompany.com
</Mail>
```

SmtpConfiguration. This element is mandatory for sending email, and you must specify the following attributes:

- `server`. Your Simple Mail Transport Protocol (SMTP) server.
- `port`. SMTP server port. Optional, default is 25.

For example:

```
<SmtpConfiguration server="smtp.yourcompany.com" port="25">
```

Example of Projects.psjproj

The following is an example of `Projects.psjproj`:

```
<?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
    -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
    -includes-to-ignore sources/math.h
    -D NEW_DEFECT
  </Options>
```

```
<LaunchingPeriod hour="12" minute="20" month="*" weekDay="1-5">
</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>
```

Open Polyspace Metrics

1 In the address bar of your Web browser, enter the following URL:


```
protocol:// ServerName: PortNumber
```

- *protocol* is either http (default) or https.
- *ServerName* is the name or IP address of your Polyspace Metrics server.
- *PortNumber* is the Web server port number (default 8080)

To use HTTPS, you must set up the configuration file and the **Metrics configuration** preferences. For more information, see “Configure Web Server for HTTPS”.


2 Select the **Projects** tab.

You can save the project index page as a bookmark for future use. You can also save as bookmarks any Polyspace Metrics pages that you subsequently navigate to.

To refresh the page at any point, click .

At the top of each column, use the filters to shorten the list of displayed projects. For example:

- In the **Project** filter, if you enter `demo_`, the browser displays a list of projects with names that begin with `demo_`.
- From the drop-down list for the **Language** filter, if you select **C**, the browser displays only C projects, if you select **C++**, the browser displays only C++ projects.

If a new verification has been carried out for a project since your last visit to the project index page, then the icon  appears before the name of the project.

If you place your cursor anywhere on a project row, in a box on the left of the window, you see the following project information:

- **Language** — For example, Ada, C, C++.
- **Mode** — Either Integration or Unit by Unit.
- **Last Run Name** — Identifier for last verification performed.
- **Number of Runs** — Number of verifications performed in project.

In a project row, click the **Project** name to go to the **Summary** view for that project.

Organize Polyspace Metrics Projects

The Polyspace Metrics project index allows you to display projects as categories, a useful feature when you have a large number of projects to manage. You can:

- Create multiple-level project categories.
- Move projects between categories by dragging and dropping projects.
- Rename and remove categories. When you remove a category, the software does not delete the projects within the category but moves the projects back to the parent or root level.

To create a root-level project category:

- 1** On the Polyspace Metrics project index, right-click a project.
- 2** From the context menu, select **Create Project Category**. The Add To Category dialog box opens.
- 3** In **Enter the name of the project category** field, enter the required name, for example, MyNewCategory. Then click **OK**.
- 4** To add projects to this new category, drag and drop the required projects into this category.

To create a subroot-level category:

- 1** Right-click a project category.
- 2** From the context menu, select **Create Project Category**. The Add To Category dialog box opens.
- 3** In **Enter the name of the project category** field, enter the required name, for example, SubCategory1. If you decide that you do not want a subroot category, but want a new root category instead, select the **Create a root project category** check box. Then click **OK**.
- 4** To add projects to this new category, drag and drop the required projects into this category.

To rename a project category:

- 1** Right-click the project category.
- 2** From the context menu, select **Rename Project Category**. The category name becomes editable.
- 3** Enter the new name for your category. Press **Return**.
- 4** A message dialog box opens requesting confirmation. Click **OK**. The software updates the category name.

To remove a project category:

- 1** Right-click the project category.
- 2** From the context menu, select **Delete Project Category**. If the project category is a:
 - Root-level project category, the software moves all projects to the root level and removes the project category and all associated subroot categories.
 - Subroot-level category, the software moves all projects within the subroot category to the parent level and removes the subroot category.

Note The software does not delete projects when removing project categories.

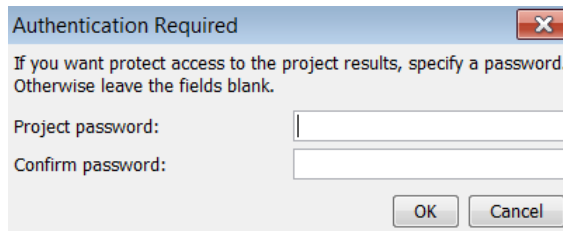
You can move projects back to the root level from project categories without removing the project categories:

- 1** From within project categories, select the projects that you want to move to the root level.
- 2** Right-click the selected projects. From the context menu, select **Move to Root**. The software moves the projects back to the root level.

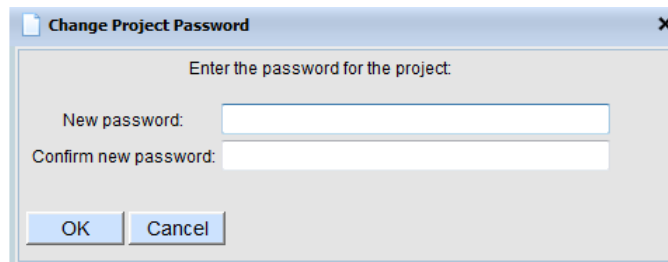
Protect Access to Project Metrics


You can restrict access to the metrics for a project by specifying a password:

- When you run a verification with Polyspace Metrics enabled or upload results to Polyspace Metrics:
 - 1 The Authentication Required dialog box opens.



- 2 In the **Project password** and **Confirm password** fields, enter your password.
 - 3 Click **OK**.
- After the creation of a project:
 - 1 From the Polyspace Metrics project index, right-click the project.
 - 2 From the context menu, select **Change/Set Password**. The Change Project Password dialog box opens.




- 3 In the **New password** and **Confirm new password** fields, enter your password.
- 4 Click **OK**. The software displays the password-restricted icon  next to the project.

From the command line, you can use the `-password` option. For example:

```
polyspace-results-repository.exe -prog psdemo_model_link_sl -password my_passwd
```

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”.

After you enter your password, the project pages are accessible for a session that lasts 30 minutes. Access is available for this period of time, even if you close your Web browser. If you return to the Polyspace Metrics project index,

the session ends. If you click  during a session, the project pages are accessible for another 30 minutes.

Web Browser Support

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

To use Polyspace Metrics, you must install on your computer Java, version 1.4 or later.

For the Firefox Web browser, you must 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/libnjp2.so
```

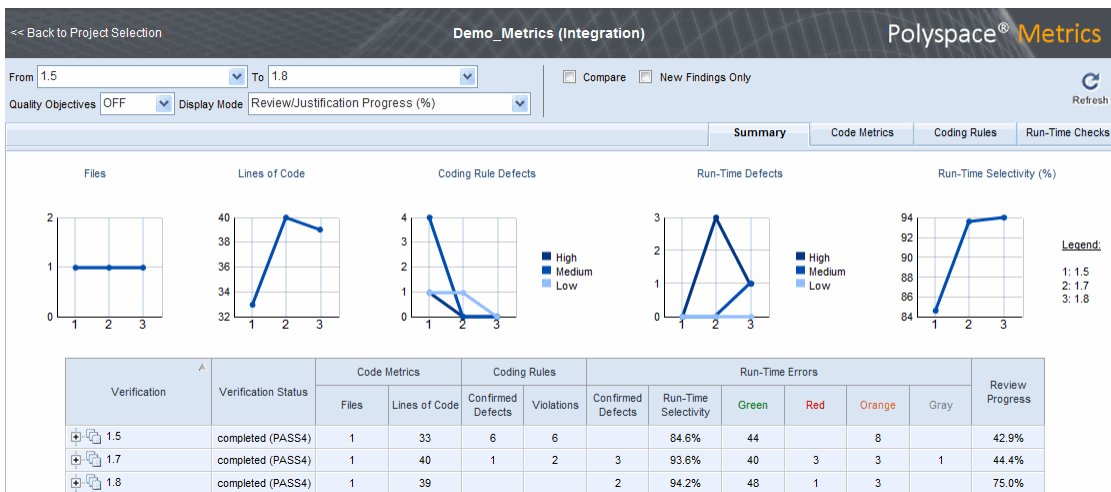
What You Can Do with Polyspace Metrics

Review Overall Progress

For a development manager or quality assurance engineer, the Polyspace Metrics **Summary** view provides useful high-level information, including quality trends, over the course of a project.

To obtain the **Summary** view for a project:

- 1 Open the Polyspace Metrics project index. See “Open Polyspace Metrics” on page 14-12.
- 2 Click anywhere in the row that contains your project. You see the **Summary** view.



At the top of the **Summary** view, use the **From** and **To** filters to specify the project versions that you want to examine. By default, the **From** and **To** fields specify the earliest and latest project versions respectively.

In addition, by default, the **Quality Objectives** filter is OFF, and the **Display Mode** is Review/Justification Progress (%).

Below the filters, you see:

- Plots that reveal how the number of verified files, lines of code, defects, and run-time selectivity vary over the different versions of your project
- A table containing summary information about your project versions.

If you have projects with two or more file modules in the Polyspace verification environment, by default, Polyspace Metrics displays verification results using the same module structure. However, Polyspace Metrics also allows you to create or delete file modules. See “Create File Module and Specify Quality Level” on page 14-25.

With the default filter settings, you can monitor progress in terms of coding rule violations and run-time checks that quality assurance engineers or developers have reviewed.

You can also monitor progress in terms of software quality objectives. You may, for example, want to find out whether the latest version fulfills quality objectives.

To display software quality information, from the **Quality Objectives** drop-down list, select **ON**.

Verification	Verification status	Code Metrics		Coding Rules		Run-Time Errors		Software Quality Objectives					
		Files	Lines of Code	Confirmed Defects	Violations	Confirmed Defects	Run-Time Reliability	Overall Status	Level	Review Progress	Code Metrics Over Threshold	Justified Coding Rules	Justified Run-Time Errors
V4	completed (PASS4)	6	463		4	3	99.4%	FAIL	SQO-4	85.7%	8	—	95.8%
V3	completed (PASS4)	6	463		4		89.9%	FAIL	Exhaustive	0.0%	8	25.0%	5.6%
V2	completed (PASS4)				4	3	88.2%	FAIL	SQO-2	23.1%	—	0.0%	55.6%
V1	completed (PASS4)				10		87.9%						

Under **Software Quality Objectives**, you look at **Review Progress** for the latest version (V4), which indicates that the review of verification results is incomplete (only 85.7% reviewed). You also see that the Overall Status is FAIL. This status indicates that, although the review is incomplete, the project code fails to meet software quality objectives for the quality level SQO-4. With this information, you may conclude that you cannot release version V4 to your customers.

When Polyspace Metrics adds the results for a new project version to the repository, Polyspace Metrics also imports comments from the previous version. For this reason, you rarely see the review progress metric at 0% after verification of the first project version.

Note You may want to find out whether your code fulfills software quality objectives at another quality level, for example, SQ0-3. **Under Software Quality Objectives**, in the **Level** cell, select SQ0-3 from the drop-down list.

There are seven quality levels, which are based on predefined software quality objectives. You can customize these software quality objectives and modify the way quality is evaluated. See “Customize Software Quality Objectives” on page 14-34.

To investigate further, under **Run-Time Errors**, in the **Confirmed Defects** cell, you click the link 3. This action takes you to the **Run-Time Checks** view, where you see an expanded view of check information for each file in the project.

Verification	Confirmed Defects	Run-Time Reliability	Green Code			Systematic Runtime Errors (Red Checks)		Unreachable Branches (Gray Checks)		Other Runtime Errors (Orange Checks)		Non-terminating Constructs		Software Quality Objectives		
			Checks	Justified	Checks	Justified	Checks	Justified	Checks	Justified	Checks	Justified	Quality Status	Level	Review Progress	
V4	3	99.4%	272	66.7%	3	100.0%	6	100.0%	27	100.0%	6	FAIL	SQ0-4	100.0%		
└─ _polyspace_stdstubs.c		100.0%	14	100.0%	1	100.0%		100.0%	2	100.0%		PASS	SQ0-4	100.0%		
└─ example.c	2	99.0%	83	0.0%	1	100.0%	2	100.0%	8	100.0%	3	FAIL	SQ0-4	100.0%		
└─ initialisations.c	1	97.7%	41	100.0%		100.0%	1	100.0%	1	100.0%		PASS	SQ0-4	100.0%		
└─ main.c		100.0%	9	100.0%		100.0%	1	100.0%	3	100.0%	2	PASS	SQ0-4	100.0%		
└─ single_file_analysis.c		100.0%	82	100.0%	1	100.0%	2	100.0%	8	100.0%	1	PASS	SQ0-4	100.0%		
└─ tasks1.c		100.0%	26	100.0%		100.0%		100.0%	3	100.0%		PASS	SQ0-4	100.0%		
└─ tasks2.c		100.0%	17	100.0%		100.0%		100.0%	2	100.0%		PASS	SQ0-4	100.0%		

To view a check in the Polyspace verification environment, in the relevant cell, click the numeric value for the check. The Polyspace product opens with the Results Manager perspective displaying verification information for this check.

Note If you update check information through the Results Manager (see “Review Results” on page 14-29), when you return to Polyspace Metrics, click **Refresh** to incorporate this updated information.

If you want to view check information with reference to check type, from the **Group by** drop-down list, select **Run-Time Categories**.

Verification	Confirmed Defects	Run-Time Reliability	Green Code	Systematic Runtime Errors (Red Checks)		Unreachable Branches (Gray Checks)		Other Runtime Errors (Orange Checks)		Non-Terminating Constructs		Software Quality Objectives		
			Checks	Justified	Checks	Justified	Checks	Justified	Checks	Justified	Checks	Justified	Quality Status	Level
V4	3	99.4%	272	66.7%	3	100.0%	6	100.0%	27	100.0%	6	FAIL	SQO-4	100.0%
ASRT - failure of user asse		100.0%		100.0%	1	100.0%		100.0%	6			PASS	SQO-4	100.0%
COR (Scalar) - failure of co		100.0%	13	100.0%		100.0%						PASS	SQO-4	100.0%
IDP - pointer within bounds	1	91.7%	9	0.0%	1	100.0%		100.0%	2			FAIL	SQO-4	100.0%
IRV - function returns an in		100.0%	34	100.0%		100.0%		100.0%				PASS	SQO-4	100.0%
NIP - non-initialized global p		100.0%	16	100.0%		100.0%		100.0%				PASS	SQO-4	100.0%
NIV - non-initialized global v		100.0%	32	100.0%		100.0%		100.0%				PASS	SQO-4	100.0%
NIVL - non-initialized local v	1	99.2%	115	100.0%		100.0%		100.0%	8			PASS	SQO-4	100.0%
NTC - non termination of ca		100.0%								100.0%	5	PASS	SQO-4	100.0%
NTL - non termination of loc		100.0%								100.0%	1	PASS	SQO-4	100.0%
OBAL - array index within b		100.0%	1	100.0%	1	100.0%		100.0%	1			PASS	SQO-4	100.0%
OVFL (Float) - overflow	1	100.0%	6	100.0%		100.0%		100.0%	3			PASS	SQO-4	100.0%
OVFL (Scalar) - overflow		100.0%	31	100.0%		100.0%		100.0%	6			PASS	SQO-4	100.0%
UNFL (Float) - underflow				100.0%		100.0%		100.0%				PASS	SQO-4	100.0%
UNFL (Scalar) - underflow				100.0%		100.0%		100.0%				PASS	SQO-4	100.0%
UNR - unreachable code		100.0%				100.0%	6					PASS	SQO-4	100.0%
ZDV (Float) - denominator r		100.0%	2	100.0%		100.0%		100.0%	1			PASS	SQO-4	100.0%
ZDV (Scalar) - denominator		100.0%	13	100.0%		100.0%		100.0%				PASS	SQO-4	100.0%

Returning to the **Summary** view, under **Coding Rules** and in the **Violations** cell, you also see that there are coding rule violations. You may want to review these violations. See “Review Results” on page 14-29.

Display Metrics for Single Project Version

To display metrics for a single project version:

- 1** In the **From** field, from the drop-down list, select the required project version.
- 2** In the **To** field, from the drop-down list, select the same project version.
- 3** In **# items** field, enter the maximum number of files for which you want information displayed.

The software displays:

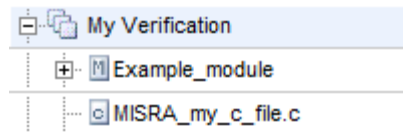
- Bar charts with file defect information, ordering the files according to the number of defects in each file
- A table with information about the selected project version

Create File Module and Specify Quality Level

You can group files into a module and specify a quality level for the module, which applies to all files within the module. By grouping your files in different modules, you can specify different quality levels for your files.

To create a module of files:

- 1 Select a tab, for example, **Summary**.
- 2 In the **Verification** column, expand the node corresponding to the verification that you are interested. You see the verified files.
- 3 Select the files that you want to place in a module.
- 4 Right-click the selected files, and, from the context menu, select **Add To Module**. The Add to Module dialog box opens.
- 5 In the text field, enter the name for your new module, for example, `Example_module`. Click **OK**. You see a new node.



To specify a quality level for the module:

- 1 Select the row containing the module.
- 2 Under **Software Quality Objectives**, click the **Level** cell.
- 3 From the drop-down list, select the quality level for your module.

To remove files from a module:

- 1 Expand the node corresponding to the module.
- 2 Select the files that you want to remove from the module.

- 3 Right-click your selection, and from the context menu, select **Remove From Module**. The software removes the files from the module. If you remove all files from the module, the software also removes the module from the tree.

Note You can drag and drop files into and out of folders. For example, you can select `MISRA_my_c_file.c` and drag the file to `Example_module`.

Compare Project Versions

You can compare metrics of two versions of a project.

- 1 In the **From** drop-down list, select one version of your project.
- 2 In the **To** drop-down list, select a newer version of your project.
- 3 Select the **Compare** check box.

In each view, for example, **Summary**, you see metric differences and tooltip messages that indicate whether the newer version is an improvement over the older version.

1.0 vs 2.0	Verification Status	Code Metrics			Coding Rules			Run-Time Errors					Overall Trend		
		Files	Lines of Code	All Metrics Trend	Confirmed Defects	Violations	All Metrics Trend	Confirmed Defects	Run-Time Selectivity	Green	Red	Orange		Gray	All Metrics Trend
2.0 (delta)	completed (PASS)	6	853 (+1)	▼		286 (-4) ▲	▲		96.8% (+0.1%) ▲	682 (+22) ▲	13 (-4) ▲	23 ▲	3 (+3) ▼	▲	▲
__polyspace_	completed (PASS)								99.5%	213		1			
dataflow.c	completed (PASS)		164			39			92.7%	76		6			
dynamicmemc	completed (PASS)		156 (+1)	▼		77 (+1) ▲	▲		97.3% (-0.6%) ▼	141 (+3) ▲	1 (-1) ▲	4 (+1) ▼		▲	▲
numeric.c	completed (PASS)		217			54 (-2) ▲	▲		99.1% (+0.2%) ▲	97 (+18) ▲	6 (-2) ▲	1	3 (+3) ▼	▲	▲
other.c	completed (PASS)		87			23			72.4%	20	1	8			
programming.	completed (PASS)		117	▼		60 (-3) ▲	▲		96.9% (+1.5%) ▲	62 (+1) ▲	0 (-1) ▲	2 (-1) ▲		▲	▲
staticmemory.	completed (PASS)		112			33			98.7%	73	5	1			

Review New Findings

You can specify a project version and focus on the differences between the verification results of your specified version and the previous verification. For example, consider a project with versions 1.0, 1.1, 1.2, 2.0, and 2.1.

- 1** In the **To** field, specify a version of your project, for example, 2.0.
- 2** Select the **New Findings Only** check box. In the **From** field, you see 1.2 in dimmed lettering, which is the previous verification. The charts and tables now show the changes in results with respect to the previous verification.

To manage the content of the bar charts, in the **# items** field, enter the maximum number of files for which you want information displayed. The software displays file defect information, ordering the files according to the number of defects in each file.

Review Results

This example shows how to review results beginning from the Polyspace Metrics interface. To review results, you must have Polyspace installed on your local computer.

1 In the Polyspace Metrics interface, click the **Summary** 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	completed (PASS4)	6	428		43		92.2%	260	4	23	6	0.0%
example.c	completed (PASS4)		125		7		90.5%	82	2	9	2	0.0%
initialisations.c	completed (PASS4)		59		13		97.7%	41		1	1	0.0%
main.c	completed (PASS4)		42		4		78.6%	9	1	3	1	0.0%
single_file_analysis.c	completed (PASS4)		72		8		91.4%	82	1	8	2	0.0%
tasks1.c	completed (PASS4)		80		3		96.6%	28		1		0.0%
tasks2.c	completed (PASS4)		50		8		94.7%	18		1		0.0%

2 To see details about run-time errors, on the **Run-Time Errors** column, click a cell value.

Verification	Confirmed Defects	Run-Time Selectivity	Green Code		Systematic Run-Time Errors (Red Checks)		Unreachable Branches (Gray Checks)		Other Run-Time Errors (Orange Checks)				Non-terminating Constructs		Review Progress
			Checks	Reviewed	Checks	Reviewed	Checks	Reviewed	Checks	Reviewed	Checks	Reviewed	Checks		
1.0		92.2%	260	0.0%	3	0.0%	6	0.0%	23	1		6	0.0%	1	0.0%
example.c		90.5%	82	0.0%	2	0.0%	2	0.0%	9			4			0.0%
initialisations.c		97.7%	41			0.0%	1	0.0%	1	1					0.0%
main.c		78.6%	9			0.0%	1	0.0%	3				0.0%	1	0.0%
single_file_analysis.c		91.4%	82	0.0%	1	0.0%	2	0.0%	8			2			0.0%
tasks1.c		96.6%	28				0.0%	1							0.0%
tasks2.c		94.7%	18				0.0%	1							0.0%

3 To see a breakdown of the errors in a file by checks, expand a filename node.


Verification	Confirmed Defects	Run-Time Selectivity	Green Code		Systematic Run-Time Errors (Red Checks)		Unreachable Branches (Gray Checks)		Other Run-Time Errors (Orange Checks)				Non-terminating Constructs	
			Checks	Reviewed	Checks	Reviewed	Checks	Reviewed	Checks	Reviewed	Checks	Reviewed	Checks	
1.0		92.2%	260	0.0%	3	0.0%	6	0.0%	23	1		6	0.0%	1
example.c		90.5%	82	0.0%	2	0.0%	2	0.0%	9			4		
-ASRT - User assertion		0.0%							1					
-IDP - Illegally dereferenced pointer		88.9%	7	0.0%	1				1			1		
-IRV - Initialized return value		100.0%	14											
-NIP - Non-initialized pointer		100.0%	12											
-NIV - Non-initialized variable		100.0%	3											

Tip To expand all subnodes under a node, right-click the node and select **Expand All Nodes**.

- 4 If a check produces a red error, the check has a value under the **Systematic Runtime Errors (Red Checks)** column. Click this value to view the check in the Polyspace verification environment.


For instance, if you click the **Systematic Runtime Errors (Red Checks)** value on the **IDP** row in `example.c`, the **Illegally dereferenced pointer** check in that file appears in the Polyspace verification environment.

- 5 In the Polyspace verification environment, on the **Results Summary** pane, enter review information such as:
 - **Classification:** If you choose the classifications High, Medium or Low, when you save the classification, the software updates the **Confirmed Defects** column in Polyspace Metrics.
 - **Status:** If you choose a review status, when you save the status, the software updates the **Review Progress** column in Polyspace Metrics.
 - **Comment**
- 6 Save the review information. The software saves this information to a local folder. To change this local folder, select **Options > Preferences** and enter the location under the **Server Configuration** tab.

If you want to save the information to the local folder *and* the Polyspace Metrics repository, on the Results Manager toolbar, click the  button.

Save Review Comments

By default, when you save your project (**Ctrl+S**), the software saves your comments and justifications to a local folder. To specify the folder location, select **Options > Preferences** and enter the location under the **Server Configuration** tab.

If you want to save your comments and justifications to a local folder *and* the Polyspace Metrics repository, on the Results Manager toolbar, click the button .

This default behavior allows you to upload your review comments and justifications only when you are satisfied that your review is, for example, correct and complete.

If you want the software to save your comments and justifications to the local folder *and* the Polyspace Metrics repository whenever you save your project (**Ctrl+S**):

- 1** Select **Options > Preferences > Server configuration**.
- 2** Select the check box **Save justifications in the Polyspace Metrics repository**.

Note In Polyspace Metrics, click  to view updated information.

Fix Defects

If you are a software developer, you can begin to fix defects in code when, for example:

- In the **Summary** view, **Review Progress** shows 100%
- Your quality assurance engineer informs you

You can use Polyspace Metrics to access defects that you must fix.

Within the **Summary** view, under **Run-Time Errors**, click any cell value. This action takes you to the **Run-Time Checks** view.

You want to fix defects that are classified as defects.

Verification	Confirmed Defects	Run-Time Selectivity	Green Code	Systematic Runtime Errors (Red Checks)	
			Checks	Reviewed	Checks
V4	4	93.2%	272	100.0%	3
└─ polyspace_stc	1	97.7%	14	100.0%	1
└─ example.c	2	92.2%	83	100.0%	1
└─ initialisations.c	1	97.8%	41	100.0%	
└─ main.c		85.0%	9	100.0%	
└─ single_file_analys		91.7%	82	100.0%	1
└─ tasks1.c		89.7%	26	100.0%	
└─ tasks2.c		89.5%	17	100.0%	

In the **Confirmed Defects** column, click a non-zero cell value. For example, if you click 2, Polyspace Code Prover opens with the checks visible in the **Results Summary** tab.

Double-click the row containing a check. In the **Check Details** pane, you see information about this check. You can now go to the source code and fix the defect.

Review Code Complexity

Polyspace Metrics supports the generation of code complexity metrics. The majority of these metrics are predefined and based on the Hersteller Initiative Software (HIS) standard.

To review the complexity of the code in your project, in the **Summary** view, click any value in a **Code Metrics** cell. The **Code Metrics** view opens.

Verification	Project Metrics						File Metrics				Function Metrics											Software Quality Objectives			
	Files	Header Files	Recursion	Direct Recursion	Protected Shared Variables	Non-Protected Shared Variables	Lines	Lines of Code	Comment Density	Estimated Function Coupling	Lines Within Body	Executable Lines	Cyclomatic Complexity	Language Scope	Paths	Calling Functions	Called Functions	Call Occurrence	Instructions	Call Levels	Function Parameters	Goto Statements	Return Points	Quality Status	Level
1.0 (18)	6	10	1	1	0	0	755	463	FAIL		359	170	PASS	FAIL	112	PASS	PASS	76	186	PASS	PASS	0	FAIL	FAIL	Exhaustive
1.0 (22)	6	10	1	1			755	463	FAIL		319	147	PASS	FAIL	106	PASS	PASS	68	164	PASS	PASS	0	FAIL	FAIL	Exhaustive
1.0 (21)	6	10	1	1	0	0	755	463	FAIL		319	147	PASS	FAIL	106	PASS	PASS	68	164	PASS	PASS	0	FAIL	FAIL	Exhaustive
1.0 (20)	6	10	1	1	0	0	755	463	FAIL		359	170	PASS	FAIL	112	PASS	PASS	76	186	PASS	PASS	0	FAIL	FAIL	Exhaustive
1.0 (19)	6	10	1	1	0	0	755	463	FAIL		359	170	PASS	FAIL	112	PASS	PASS	76	186	PASS	PASS	0	FAIL	FAIL	Exhaustive

The software generates numeric values or pass/fail results for various metrics. For information about:

- The Hersteller Initiative Software (HIS) standard, see HIS Source Code Metrics.
- Other, non-HIS, code metrics, see “Code Metrics” on page 14-53 and “SQO Level 1” on page 14-36.
- How Polyspace evaluates these metrics and how you can customize code complexity metrics, see “Customize Software Quality Objectives” on page 14-34 and “SQO Level 1” on page 14-36.

Customize Software Quality Objectives

When you run your first verification to produce metrics, Polyspace software uses *predefined* software quality objectives (SQO) to evaluate quality. In addition, when you use Polyspace Metrics for the first time, Polyspace creates the following XML file that contains definitions of these software quality objectives:

```
RemoteDataFolder/Custom-SQO-Definitions.xml
```

RemoteDataFolder is the folder where Polyspace stores data generated by remote verifications.

If you want to customize SQOs and modify the way quality is evaluated, you must change *Custom-SQO-Definitions.xml*. This XML file has the following form:

```
<?xml version="1.0" encoding="utf-8"?>
<MetricsDefinitions>
  SQO Level 1
  SQO Level 2
  SQO Level 3
  SQO Level 4
  SQO Level 5
  SQO Level 6
  SQO Exhaustive
  Coding Rules Set 1
  Coding Rules Set 2
  Run-Time Checks Set 1
  Run-Time Checks Set 2
  Run-Time Checks Set 3
  Status Acronym 1
  Status Acronym 2
</MetricsDefinitions>
```

You can redefine the pass/fail thresholds for the various SQO levels of Polyspace Metrics by editing the content of elements that make up *MetricsDefinitions*, for example, *SQO Level 2*, and *Run-Time Checks Set 1*. In addition, you can create elements that contain SQO levels, and coding

rule and run-time check sets that you define. You can use these new elements to replace or augment the default elements.

Note Although Polyspace provides support for MISRA C++ and JSF++ coding rules, Polyspace Metrics does not generate a default coding rules set for these standards in `Custom-SQO-Definitions.xml`. However, you can define your own set for these standards. See “Coding Rules Set 1” on page 14-44.

Concepts

- “SQO Levels” on page 14-36
- “Coding Rules Sets” on page 14-44
- “Run-Time Checks Sets” on page 14-48
- “Status Acronyms” on page 14-52

SQO Levels

In this section...

“SQO Level 1” on page 14-36
 “SQO Level 2” on page 14-41
 “SQO Level 3” on page 14-41
 “SQO Level 4” on page 14-41
 “SQO Level 5” on page 14-42
 “SQO Level 6” on page 14-42
 “SQO Exhaustive” on page 14-42

SQO Level 1

The default *SQO Level 1* element is:

```

<SQO ID="SQO-1">
  <!-- HIS metrics -->
  <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>MISRA_Rules_Set_1</Num_Unjustified_Violations>
</SQO>
  
```

The SQO Level 1 element is composed of sub-elements with data that specify thresholds. The sub-elements represent metrics that are calculated in a

verification. If the metrics do not exceed the thresholds, the code meets the quality level specified by SQA Level 1.

By default, Polyspace Metrics does not evaluate C++ coding rule violations for SQA Level 1. However, you can incorporate coding rule violations by adding the sub-element `Num_Unjustified_Violations` to the list of sub-elements in *SQA Level 1*.

For example, to apply a MISRA C++ rules set, add the following sub-element:

```
<Num_Unjustified_Violations>MISRA_Cpp_Rules_Set</Num_Unjustified_Violations>
```

`MISRA_Cpp_Rules_Set` is the ID attribute of the MISRA C++ coding rules set that you create. For information about creating MISRA C++ and JSF++ rule sets, see “Coding Rules Set 1” on page 14-44.

The following table describes the Hersteller Initiative Software (HIS) standard metrics specified by the sub-elements and provides default thresholds.

Element	Default threshold	Description of metric
<code>comf</code>	20	Comment density of a file
<code>path</code>	80	Number of paths through a function
<code>goto</code>	0	Number of <code>goto</code> statements
<code>vg</code>	10	Cyclomatic complexity
<code>calling</code>	5	Number of calling functions
<code>calls</code>	7	Number of calls
<code>param</code>	5	Number of parameters per function
<code>stmt</code>	50	Number of instructions per function
<code>level</code>	4	Number of call levels in a function
<code>return</code>	1	Number of return statements in a function

Element	Default threshold	Description of metric
vocf	4	<p>Language scope, an indicator of the cost of maintaining or changing functions. Calculated as follows:</p> $(N1+N2) / (n1+n2)$ <ul style="list-style-type: none"> • <i>n1</i> — Number of different operators • <i>N1</i> — Total number of operators • <i>n2</i> — Number of different operands • <i>N2</i> — Total number of operands <p>The computation is based on the preprocessed source code. Consider the following code.</p> <pre>int f(int i) { if (i == 1) return i; else return i * g(i-1); }</pre> <p>The code contains the following:</p> <ul style="list-style-type: none"> • Distinct operators — int, (,), {, if, ==, return, else, *, -, ;, and } • Number of operators —12 • Number of operator occurrences —19 • Distinct operands — f, i, 1, and g • Number of operands — 4 • Number of operand occurrences — 9

Element	Default threshold	Description of metric
		Therefore, the language scope for the code is $VOCF = (19 + 9) / (12 + 4)$, that is, 1.8.
ap_cg_cycle	0	Number of recursions
ap_cg_direct_cycle	0	Number of direct recursions
Num_Unjustified_Violations	See “Coding Rules Set 1” on page 14-44	Number of unjustified violations of MISRA C rules specified by “Coding Rules Set 1” on page 14-44

For more information about these metrics, see HIS Source Code Metrics.

The following points also apply:

- Polyspace does not evaluate metrics for template functions or member functions of a template class.
- A `catch` statement is treated as a control flow statement that generates two paths and increments cyclomatic complexity by one.
- Explicit and implicit calls to constructors are taken into account in the computation of the number of distinct calls (`calls`).
- The computation of the number of call graph cycles does not take into account template functions or member functions of a template class.

Polyspace Metrics also supports the evaluation of non-HIS code metrics, which the following table describes.

Element	Description of metric
fco	<p>Estimated function coupling, which is calculated as follows:</p> $SOC - DFF + 1$ <ul style="list-style-type: none"> • SOC — Sum (over all file functions) of calls within body of each function • DFF — Number of defined file functions <p>Does not take into account member functions of a template class or template functions. Computed metric reflects coupling of non-template functions only.</p>
flin	Number of lines within function body
fxln	Number of execution lines within function body A variable declaration with initialization is treated as a statement, but not as an execution line of function body.
ncalls	Number of calls within function body Includes explicit and implicit calls to constructors.
pshv	Number of protected shared variables
unpshv	Number of unprotected shared variables

To generate these metrics, insert the sub-elements into the SQO Level 1 element and specify thresholds:

```
<SQO ID="SQO-1">
  <!-- HIS metrics -->
  ...
  ...
  <!-- Other non-HIS metrics -->
  <fco>user_defined_threshold</fco>
  <flin>user_defined_threshold</flin>
  <fxln>user_defined_threshold</fxln>
  <ncalls>user_defined_threshold</ncalls>
```



```

    <pshv>user_defined_threshold</pshv>
    <unpshv>user_defined_threshold</unpshv>
</SQA>

```

SQA Level 2

The default SQA Level 2 element is:

```

<SQA ID="SQA-2" ParentID="SQA-1">
  <Num_Unjustified_Red>0</Num_Unjustified_Red>
  <Num_Unjustified_NT_Constructs>0</Num_Unjustified_NT_Constructs>
</SQA>

```

To fulfill requirements of SQA Level 2, the code must meet the requirements of SQA Level 1 **and** the following:

- Number of unjustified red checks `Num_Unjustified_Red` must not be greater than the threshold (default is zero)
- Number of unjustified NTC and NTL checks `Num_Unjustified_NT_Constructs` must not be greater than the threshold (default is zero)

SQA Level 3

The default SQA Level 3 element is:

```

<SQA ID="SQA-3" ParentID="SQA-2">
  <Num_Unjustified_UNR>0</Num_Unjustified_UNR>
</SQA>

```

To fulfill requirements of SQA Level 3, the code must meet the requirements of SQA Level 2 **and** the number of unjustified UNR checks must not exceed the threshold (default is zero).

SQA Level 4

The default SQA Level 4 element is:

```

<SQA ID="SQA-4" ParentID="SQA-3">
  <Percentage_Proven_Or_Justified>Runtime_Checks_Set_1</Percentage_Proven_Or_Justified>
</SQA>

```

To fulfill requirements of SQO Level 4, the code must meet the requirements of SQO Level 3 **and** the following ratio as a percentage

$(\text{green checks} + \text{justified orange checks}) / (\text{green checks} + \text{all orange checks})$

must not be less than the thresholds specified by “Run-Time Checks Set 1” on page 14-48.

SQO Level 5

The default SQO Level 5 element is:

```
<SQO ID="SQO-5" ParentID="SQO-4">
  <Num_Unjustified_Violations>MISRA_Rules_Set_2</Num_Unjustified_Violations>
  <Percentage_Proven_Or_Justified>Runtime_Checks_Set_2</Percentage_Proven_Or_Justified>
</SQO>
```

To fulfill requirements of SQO Level 5, the code must meet the requirements of SQO Level 4 **and** the following:

- Number of unjustified violations of MISRA C rules must not exceed thresholds specified by “Coding Rules Set 2” on page 14-46.
- Percentage of green and justified checks must not be less than the thresholds specified by “Run-Time Checks Set 2” on page 14-49

SQO Level 6

The default SQO Level 6 element is:

```
<SQO ID="SQO-6" ParentID="SQO-5">
  <Percentage_Proven_Or_Justified>Runtime_Checks_Set_3</Percentage_Proven_Or_Justified>
</SQO>
```

To fulfill requirements of SQO Level 6, the code must meet the requirements of SQO Level 5 **and** the percentage of green and justified checks must not be less than the thresholds specified by “Run-Time Checks Set 3” on page 14-50.

SQO Exhaustive

The default Exhaustive element is:

```
<SQA ID="Exhaustive" ParentID="SQA-1">
  <Num_Unjustified_Violations>0</Num_Unjustified_Violations>
  <Num_Unjustified_Red>0</Num_Unjustified_Red>
  <Num_Unjustified_NT_Constructs>0</Num_Unjustified_NT_Constructs>
  <Num_Unjustified_UNR>0</Num_Unjustified_UNR>
  <Percentage_Proven_Or_Justified>100</Percentage_Proven_Or_Justified>
</SQA>
```

To fulfill the requirements for this level, the code must meet the requirements of SQA Level 1 **and** the following:

- Number of unjustified violations of MISRA C rules must not exceed the threshold (default is zero)
- Number of unjustified red checks must not exceed the threshold (default is zero)
- Number of unjustified NTC and NTL checks must not exceed the threshold (default is zero)
- Number of unjustified UNR checks must not exceed the threshold (default is zero)
- Percentage of green and justified checks must not be less than the threshold (default is 100%)

Coding Rules Sets

In this section...
“Coding Rules Set 1” on page 14-44
“Coding Rules Set 2” on page 14-46

Coding Rules Set 1

For C code, this element defines a set of MISRA C rules that can be applied to the code during the compilation phase, with corresponding violation thresholds.

For C++ code, you can specify the `CodingRulesSet` element to contain a set of MISRA C++ or JSF++ rules. The software applies these rules to the code during the compilation phase together with the violation thresholds that you specify.

MISRA C Rules

The default structure of Coding Rules Set 1 is:

```
<CodingRulesSet ID="MISRA_Rules_Set_1">
  <Rule Name="MISRA_C_8_11">0</Rule>
  <Rule Name="MISRA_C_8_12">0</Rule>
  <Rule Name="MISRA_C_11_2">0</Rule>
  <Rule Name="MISRA_C_11_3">0</Rule>
  <Rule Name="MISRA_C_12_12">0</Rule>
  <Rule Name="MISRA_C_13_3">0</Rule>
  <Rule Name="MISRA_C_13_4">0</Rule>
  <Rule Name="MISRA_C_13_5">0</Rule>
  <Rule Name="MISRA_C_14_4">0</Rule>
  <Rule Name="MISRA_C_14_7">0</Rule>
  <Rule Name="MISRA_C_16_1">0</Rule>
  <Rule Name="MISRA_C_16_2">0</Rule>
  <Rule Name="MISRA_C_16_7">0</Rule>
  <Rule Name="MISRA_C_17_3">0</Rule>
  <Rule Name="MISRA_C_17_4">0</Rule>
  <Rule Name="MISRA_C_17_5">0</Rule>
  <Rule Name="MISRA_C_17_6">0</Rule>
```

```

<Rule Name="MISRA_C_18_3">0</Rule>
<Rule Name="MISRA_C_18_4">0</Rule>
<Rule Name="MISRA_C_20_4">0</Rule>
</CodingRulesSet>

```

To modify the default set, you can:

- Add rules by inserting a `Rule` element with the relevant `Name` attribute. For example, to add MISRA C rule 13.1 with a zero threshold, specify the following element in `CodingRulesSet`>

```
<Rule Name="MISRA_C_13_1">0</Rule>
```

- Remove rules.

MISRA C++ Rules

To create a MISRA C++ rule set, specify the `CodingRulesSet` element using the following `Rule Name` element:

```
<Rule Name= MISRA_CPP_Rule_Number >Threshold</Rule>
```

For example:

```

<CodingRulesSet ID="MISRA_Cpp_Rules_Set">
  <Rule Name="MISRA_CPP_0_1_2">0</Rule>
  <Rule Name="MISRA_CPP_5_0_6">0</Rule>
  ....
</CodingRulesSet>

```

JSF++ Rules

To create a JSF++ rule set, specify the `CodingRulesSet` element using the following `Rule Name` element:

```
<Rule Name= JSF_CPP_Rule_Number >Threshold</Rule>
```

For example:

```

<CodingRulesSet ID="JSF_Cpp_Rules_Set">
  <Rule Name="JSF_CPP_180">0</Rule>
  <Rule Name="JSF_CPP_190">0</Rule>

```

```
....  
</CodingRulesSet>
```

Coding Rules Set 2

This element defines a set of MISRA C rules that can be applied to the code during the compilation phase, with corresponding violation thresholds. The default structure of Coding Rules Set 2 is:

```
<CodingRulesSet ID="MISRA_Rules_Set_2" ParentID="MISRA_Rules_Set_1">  
  <Rule Name="MISRA_C_6_3">0</Rule>  
  <Rule Name="MISRA_C_8_7">0</Rule>  
  <Rule Name="MISRA_C_9_2">0</Rule>  
  <Rule Name="MISRA_C_9_3">0</Rule>  
  <Rule Name="MISRA_C_10_3">0</Rule>  
  <Rule Name="MISRA_C_10_5">0</Rule>  
  <Rule Name="MISRA_C_11_1">0</Rule>  
  <Rule Name="MISRA_C_11_5">0</Rule>  
  <Rule Name="MISRA_C_12_1">0</Rule>  
  <Rule Name="MISRA_C_12_2">0</Rule>  
  <Rule Name="MISRA_C_12_5">0</Rule>  
  <Rule Name="MISRA_C_12_6">0</Rule>  
  <Rule Name="MISRA_C_12_9">0</Rule>  
  <Rule Name="MISRA_C_12_10">0</Rule>  
  <Rule Name="MISRA_C_13_1">0</Rule>  
  <Rule Name="MISRA_C_13_2">0</Rule>  
  <Rule Name="MISRA_C_13_6">0</Rule>  
  <Rule Name="MISRA_C_14_8">0</Rule>  
  <Rule Name="MISRA_C_14_10">0</Rule>  
  <Rule Name="MISRA_C_15_3">0</Rule>  
  <Rule Name="MISRA_C_16_3">0</Rule>  
  <Rule Name="MISRA_C_16_8">0</Rule>  
  <Rule Name="MISRA_C_16_9">0</Rule>  
  <Rule Name="MISRA_C_19_4">0</Rule>  
  <Rule Name="MISRA_C_19_9">0</Rule>  
  <Rule Name="MISRA_C_19_10">0</Rule>  
  <Rule Name="MISRA_C_19_11">0</Rule>  
  <Rule Name="MISRA_C_19_12">0</Rule>  
  <Rule Name="MISRA_C_20_3">0</Rule>  
</CodingRulesSet>
```

To modify the default set, you can:

- Add rules by inserting a `Rule` element with the relevant `Name` attribute. For example, to add MISRA C rule 6.1 with a zero threshold, specify the following element in `CodingRulesSet`:

```
<Rule Name="MISRA_C_6_1">0</Rule>
```

- Remove rules.

Run-Time Checks Sets

In this section...

“Run-Time Checks Set 1” on page 14-48

“Run-Time Checks Set 2” on page 14-49

“Run-Time Checks Set 3” on page 14-50

Run-Time Checks Set 1

The Run-Time Checks Set 1 is composed of Check elements with data that specify thresholds. The Name and Type attribute in each Check element defines a run-time check, while the element data specifies a threshold in percentage. The default structure of Run-Time Checks Set 1 is:

```
<RuntimeChecksSet ID="Runtime_Checks_Set_1">
  <Check Name="OBAI">80</Check>
  <Check Name="ZDV" Type="Scalar">80</Check>
  <Check Name="ZDV" Type="Float">80</Check>
  <Check Name="NIVL">80</Check>
  <Check Name="NIV">60</Check>
  <Check Name="IRV">80</Check>
  <Check Name="FRIV">80</Check>
  <Check Name="FRV">80</Check>
  <Check Name="UOVFL" Type="Scalar">60</Check>
  <Check Name="UOVFL" Type="Float">60</Check>
  <Check Name="OVFL" Type="Scalar">60</Check>
  <Check Name="OVFL" Type="Float">60</Check>
  <Check Name="UNFL" Type="Scalar">60</Check>
  <Check Name="UNFL" Type="Float">60</Check>
  <Check Name="IDP">60</Check>
  <Check Name="NIP">60</Check>
  <Check Name="POW">80</Check>
  <Check Name="SHF">80</Check>
  <Check Name="COR">60</Check>
  <Check Name="NNR">50</Check>
  <Check Name="EXCP">50</Check>
  <Check Name="EXC">50</Check>
  <Check Name="NNT">50</Check>
</RuntimeChecksSet>
```



```

    <Check Name="CPP">50</Check>
    <Check Name="OOP">50</Check>
    <Check Name="ASRT">60</Check>
  </RuntimeChecksSet>

```

When you use Run-Time Checks Set 1 in evaluating code quality, the software calculates the following ratio as a percentage for each run-time check in the set:

$$(\text{green checks} + \textit{justified} \text{ orange checks}) / (\text{green checks} + \textit{all} \text{ orange checks})$$

If the percentage values do not exceed the thresholds in the set, the code meets the quality level.

To modify the default set, you can change the check threshold values.

Run-Time Checks Set 2

This set is similar to “Run-Time Checks Set 1” on page 14-48, but has more stringent threshold values.

```

<RuntimeChecksSet ID="Runtime_Checks_Set_2">
  <Check Name="OBAI">90</Check>
  <Check Name="ZDV" Type="Scalar">90</Check>
  <Check Name="ZDV" Type="Float">90</Check>
  <Check Name="NIVL">90</Check>
  <Check Name="NIV">70</Check>
  <Check Name="IRV">90</Check>
  <Check Name="FRIV">90</Check>
  <Check Name="FRV">90</Check>
  <Check Name="UOVFL" Type="Scalar">80</Check>
  <Check Name="UOVFL" Type="Float">80</Check>
  <Check Name="OVFL" Type="Scalar">80</Check>
  <Check Name="OVFL" Type="Float">80</Check>
  <Check Name="UNFL" Type="Scalar">80</Check>
  <Check Name="UNFL" Type="Float">80</Check>
  <Check Name="IDP">70</Check>
  <Check Name="NIP">70</Check>
  <Check Name="POW">90</Check>
  <Check Name="SHF">90</Check>
  <Check Name="COR">80</Check>

```

```

<Check Name="NNR">70</Check>
<Check Name="EXCP">70</Check>
<Check Name="EXC">70</Check>
<Check Name="NNT">70</Check>
<Check Name="CPP">70</Check>
<Check Name="OOP">70</Check>
<Check Name="ASRT">80</Check>
</RuntimeChecksSet>

```

Run-Time Checks Set 3

This set is similar to “Run-Time Checks Set 1” on page 14-48, but has more stringent threshold values.

```

<RuntimeChecksSet ID="Runtime_Checks_Set_3">
  <Check Name="OBAI">100</Check>
  <Check Name="ZDV" Type="Scalar">100</Check>
  <Check Name="ZDV" Type="Float">100</Check>
  <Check Name="NIVL">100</Check>
  <Check Name="NIV">80</Check>
  <Check Name="IRV">100</Check>
  <Check Name="FRIV">100</Check>
  <Check Name="FRV">100</Check>
  <Check Name="UOVFL" Type="Scalar">100</Check>
  <Check Name="UOVFL" Type="Float">100</Check>
  <Check Name="OVFL" Type="Scalar">100</Check>
  <Check Name="OVFL" Type="Float">100</Check>
  <Check Name="UNFL" Type="Scalar">100</Check>
  <Check Name="UNFL" Type="Float">100</Check>
  <Check Name="IDP">80</Check>
  <Check Name="NIP">80</Check>
  <Check Name="POW">100</Check>
  <Check Name="SHF">100</Check>
  <Check Name="COR">100</Check>
  <Check Name="NNR">90</Check>
  <Check Name="EXCP">90</Check>
  <Check Name="EXC">90</Check>
  <Check Name="NNT">90</Check>
  <Check Name="CPP">90</Check>
  <Check Name="OOP">90</Check>
  <Check Name="ASRT">100</Check>

```

</RuntimeChecksSet>

Status Acronyms

When you click a link, `StatusAcronym` elements are passed to the Polyspace verification environment. This feature allows you to define, through your Polyspace server, additional items for the drop-down list of the **Status** field in **Check Review**.

Polyspace Metrics provides the following default elements:

```
<StatusAcronym Justified="yes" Name="Justify with code/model annotations"/>
<StatusAcronym Justified="yes" Name="No action planned"/>
```

The **Name** attribute specifies the name that appears on the **Status** field drop-down list. If you specify the **Justify** attribute to be **yes**, then when you select the item, for example, `No action planned`, the software automatically selects the **Justified** check box. If you do not specify the **Justify** attribute, then the **Justified** check box is not selected automatically.

You can remove the default elements and create new `StatusAcronym` elements, which are available to users of your Polyspace server.

Code Metrics

The following table provides descriptions of the metrics that you see in the **Code Metrics** view.

Level	Metric name	Description	HIS metric?
Project	Files	Number of source files.	No
	Header Files	Directly and indirectly included header files, including Polyspace internal header files and the header files included by these internal files. The number of included headers shows how many header files are verified for the current project.	No
	Recursions	Call graph recursions. Number of call cycles over one or more functions. If one function is at the same time directly recursive (it calls itself) and indirectly recursive, the call cycle is counted only once. Call cycle through pointer is not considered.	Yes
	Direct Recursions	Number of direct recursions.	Yes
	Protected Shared Variables	Number of protected shared variables. This measure is provided only from the verification PASS0.	No
	Non-Protected Shared Variables	Number of unprotected shared variables. This measure is provided only from the verification PASS0.	No

Level	Metric name	Description	HIS metric?
File	Lines	Number of lines. Physical lines including comment and blank lines	No
	Lines of Code	Number of lines without comment, that is, lines excluding blank or comment lines. A line that contains code and comment is counted. See “Number of Lines of Code Calculation” on page 14-63.	No
	Comment Density	Relationship of the number of comments (outside and within functions) to the number of statements. An internal comment is a comment that begins and/or ends with the source code line; otherwise a comment is considered external. In the comment density calculation, the comments in the header file (before the first preprocessing directive or the first token in the source file) are ignored. Two comments that are not separated by a token are considered as one occurrence. The number of statements within a file is the number of semicolons in the preprocessed source code except within for loops, structure or union field definitions, comments, literal strings, preprocessing directives, or parameters lists in the scope of K & R style function declarations. The comment density is: number of external comment occurrences / number of statements	Yes
	Estimated Function Coupling	Inter-file dependency. Metric is equal to:	No

Level	Metric name	Description	HIS metric?
		<p>sum of call occurrences – number of functions defined in the file + 1.</p> <p>The function coupling is calculated in a preprocessed file.</p>	
Function	Lines Within Body	<p>Total number of lines in a function body, including blank and comment lines: number of lines between the first { and the last } of a function body.</p> <p>The number of lines within a function body is calculated in the preprocessed file. If a function body contains an #include directive, the included file source code is taken into account in the calculation of the lines of this function.</p> <p>The preprocessing directives lines are taken into account in the calculation of the lines.</p>	No
	Executable Lines	<p>Total number of lines with source code tokens between a function body '{' and '}' that are not declarations (w/o static initializer), comments, braces, or preprocessing directives.</p> <p>The number of execution lines within a function body is calculated in a preprocessed file.</p> <p>If the function body contains an #include directive, the included file source code is taken into account in the calculation of the execution lines of this function.</p>	No
	Cyclomatic Complexity	<p>Number of decisions + 1. The ?: operator is considered a decision, but the combination of && is considered to be only one decision.</p>	Yes
	Language Scope	<p>The language scope is an indicator of the cost of maintaining or changing functions.</p>	Yes

Level	Metric name	Description	HIS metric?
		<p>Metric value = $(N1+N2) / (n1+n2)$ where: n1 = number of different operators N1 = sum of all operators n2 = number of different operands N2 = sum of all operands</p> <p>The computation is based on the preprocessed source code. Consider the following code.</p> <pre data-bbox="560 755 793 968"> int f(int i) { if (i == 1) return i; else return i * g(i-1); } </pre> <p>In this code, the:</p> <ul data-bbox="565 1078 1099 1380" style="list-style-type: none"> • Distinct operators are int, (,), {, if, ==, return, else, *, -, ;, } • Number of operators is 12 • Number of operator occurrences is 17 • Distinct operands are f, i, 1, g • Number of operands is 4 • Number of operand occurrences is 9 <p>For this example, the metric value is:</p> <p>1.8 $((17 + 9) / (12 + 4))$</p>	

Level	Metric name	Description	HIS metric?
	Paths	<p>Estimated static path count.</p> <p>The following code contains one path.</p> <pre>switch (n) { case 1: case 2: case 3: case 4: default: break; }</pre> <p>The following code contains two paths.</p> <pre>switch (n) { case 1: case 2: break; case 3: case 4: default: break; }</pre> <p>Implicit <code>else</code> is considered as one path.</p> <p>This value is not computed when a <code>goto</code> exists within the function body.</p>	Yes
	Calling Functions	Number of distinct callers of a function. Call through pointer is not considered.	Yes
	Called Functions	Number of distinct functions called by a function. Call through pointer is not considered. See description for Call Occurences	Yes


Level	Metric name	Description	HIS metric?
	<p>Call Occurences</p>	<p>Number of call occurrences within function body.</p> <p>Similar to Called Functions except that each call of a function is counted.</p> <p>Consider the following code.</p> <pre>int callee_1() {return 0;} int callee_2() {return 0;} int get() { return callee_1() + callee_1() + callee_2() + callee_2(); }</pre> <p>For this code, the Called Functions value is 2 but the Call Occurences value is 4.</p>	<p>No</p>
	<p>Instructions</p>	<p>Number of instructions per function, which is a measure of function complexity.</p> <p>Let $STMT(function_code_element)$ represent the metric value for $function_code_element$. The following applies:</p> <p>$STMT(simple_statement) = 1$</p> <p>$STMT(empty_statement) = 0$</p> <p>$STMT(label) = 0$</p> <p>$STMT(block) = STMT(block_body)$</p> <p>$STMT(declaration_without_initializer) = 0;$</p> <p>$STMT(declaration_with_initializer) = 1;$</p> <p>$STMT(other_statements) = 1$ where $other_statements$ are break, continue,</p>	<p>Yes</p>

Level	Metric name	Description	HIS metric?
		do-while, for, goto, if, return, switch, while.	
	Call Levels	<p>Maximum depth of nesting of control flow structures such as if, switch, for or while inside a function body.</p> <p>In the following code, the function foo has a call level of 3.</p> <pre> int foo(int &x, int &y) { int ret = 0; if (x == 0) /* call level 1 */ { ret = 0; } else if (x >= y) /* call level 2 */ { ret = 0; } else { while(x<y) /* call level 3 */ { x+=2; ret++; } } return ret; } </pre> <p>If there are no control flow structures, the call level is 1.</p> <p>To improve code readability, reduce this metric. For instance, in the above code,</p>	Yes

Level	Metric name	Description	HIS metric?
		you can convert the content of the <code>else</code> branch into a separate function and call that function from the <code>else</code> branch. This action reduces the call level to 2.	
	Function Parameters	Number of parameters per function. A measure of the complexity of the function interface. Ellipsis (...) parameter is ignored.	Yes
	Goto Statements	Number of goto statements within a function. <code>break</code> and <code>continue</code> are not counted as <code>goto</code> statements. If this value is > 0, the number of Paths cannot be computed.	Yes
	Return Points	Number of return points within a function. Number of explicit return statements within a function body. The following function has zero return points: <code>void f(void) {}</code> , The following function has one return point: <code>void f(void) {return;}</code>	Yes

Run-Time Checks

Some of the columns on the **Run-Time Checks** tab are described below. You can group the information in the columns by **Files** or **Run-Time Categories**.

Name	Description
Run-Time Selectivity	Percentage of checks that returned either red or green.
Checks	Number of checks of a certain color
Reviewed	<p>Red, gray or orange checks for which you have performed the following actions in the Polyspace user interface:</p> <ul style="list-style-type: none"> • You have entered review information such as Classification and Status. • You have saved the review information in the Polyspace Metrics repository using the  button. <p>Depending on your Display Mode, this metric is:</p> <ul style="list-style-type: none"> • Expressed as a number or percentage. • Replaced by the To Review metric.
Path-Related Issues	Number of checks in a function body that are orange because a fraction of calls to the function cause a run-time error. For more information, see “Path” on page 10-106.

Name	Description
Bounded Input Issues	Number of checks in a function body that are orange because a fraction of the inputs to the function cause a run-time error. The checks come under the category Bounded Input Issues if you restrict the inputs using Data Range Specifications . For more information, see “Bounded Input Values” on page 10-107.
Unbounded Input Issues	Number of checks in a function body that are orange because a fraction of the inputs to the function cause a run-time error. The checks come under the category Unbounded Input Issues if you do not restrict the input values. For more information, see “Unbounded Input Values” on page 10-107.
Review Progress	Checks that you have reviewed. This column aggregates the information in the three Reviewed columns. Depending on your Display Mode , this metric is: <ul style="list-style-type: none"> • Expressed as a number or percentage. • Replaced by the Remaining Review Work column.

Related Examples

- “Review Results” on page 14-29

Concepts

- “Code Metrics” on page 14-53
- “Data Range Specifications (DRS)” on page 6-55

Number of Lines of Code Calculation

For the following code, the line count in a text editor is 15 lines.

```
1  #include <stddef.h>
2
3  unsigned char v1,v2,v3;
4
5  unsigned char myfunc(void)
6  {
7      if(v1>v2)
8      {
9          v3=v2
10         + v1;
11     }
12
13     return v3;
14 }
15
```

Polyspace Metrics calculates the following:

- Number of lines — 14
- Number of lines of code — 11
- Number of lines within body — 7
- Executables lines — 4

The verification log file displays the following:

- Lines of code — 14
- Lines of code without comments — 11

Administer Results Repository

In this section...
“Administer Repository Through Web Browser” on page 14-64
“Administer Repository From Command Line” on page 14-65
“Backup Results Repository” on page 14-66

Administer Repository Through Web Browser

To rename a project:

- 1 In your Polyspace Metrics project index, right-click the row with the project that you want to rename.
- 2 From the context menu, select **Rename Project**.
- 3 In the **Project** field, enter the new name.

To delete a project:

- 1 In your Polyspace Metrics project index, right-click the row with the project that you want to delete.
- 2 From the context menu, select **Delete Project from Repository**.

To rename a verification:

- 1 Select the **Summary** view for your project.
- 2 In the **Verification** column, right-click the verification that you want to rename.
- 3 From the context menu, select **Rename Run**.
- 4 In the **Project** field, edit the text to rename the verification.

To delete a verification:

- 1 Select the **Summary** view for your project.

- 2** In the **Verification** column, right-click the verification that you want to delete.
- 3** From the context menu, select **Delete Run from Repository**.

Administer Repository From Command Line

You can run the following batch command with various options.

```
MATLAB_Install/polyspace/bin/polyspace-results-repository[.exe]
```

- To rename a project or version, use the following options:

```
[-f] [-server hostname] -rename [-prog  
old_prog -new-prog new_prog]  
[-verif-version old_version -new-verif-version new_version]
```

- *hostname* — Polyspace server. localhost if you run the command directly on the server.
 - *old_prog* — Current project name
 - *new_prog* — New project name
 - *old_version* — Old version name
 - *new_version* — New version name
 - -f — Specifies that confirmation is not requested
- To delete a project or version, use the following options:

```
[-f] [ server hostname] -delete -prog  
prog [-verif-version version]  
[-unit-by-unit|-integration]
```

- *hostname* — Polyspace server. localhost if you run the command directly on the server.
- *prog* — Project name
- *version* — Version name. If omitted, all versions are deleted
- unit-by-unit|-integration — Delete only unit-by-unit or integration verifications
- -f — Specifies that confirmation is not requested

- To get information about other commands, for example, retrieve a list of projects or versions, and download and upload results, use the `-h` option.

Renaming and Deleting Verifications From the Command Line

To change the name of the project `psdemo_model_link_sl` to `Track_Quality`:

```
polyspace-results-repository.exe -prog psdemo_model_link_sl  
-new-prog Track_Quality -rename
```

To delete the fifth verification run with version 1.0 of the project `Track_Quality`:

```
polyspace-results-repository.exe -prog Track_Quality -verif-version 1.0  
-run-number 5 -delete
```

To rename verification 1.2 as 1.0:

```
polyspace-results-repository.exe -prog Track_Quality -verif-version 1.2  
-new-verif-version 1.0 -rename
```

To rename the fourth verification run with version 1.0 as version 0.4:

```
polyspace-results-repository.exe -prog Track_Quality -verif-version 1.0  
-run-number 4 -new-verif-version 0.4 -rename
```

Backup Results Repository

To preserve your Polyspace Metrics data, create a backup copy of the results repository `PolyspaceRLDatas/results-repository` — `PolyspaceRLDatas` is the path to the folder where Polyspace stores data generated by remote verifications. See “Set Up Polyspace Metrics”.

For example, on a Linux system, do the following:

- 1 `$cd PolyspaceRLDatas`
- 2 `$zip -r Path_to_backup_folder/results-repository.zip
results-repository`

If you want to restore data from the backup copy:

1 `$cd PolyspaceRLDats`

2 `$unzip Path_to_backup_folder/results-repository.zip`

Configure Model for Code Analysis

- “Model Configuration for Code Generation and Analysis” on page 15-2
- “Configure Simulink Model” on page 15-3
- “Recommended Model Settings for Code Analysis” on page 15-5
- “Check Simulink Model Settings” on page 15-7
- “Check Simulink Model Settings Before Code Generation” on page 15-8
- “Check Simulink Model Settings Before Analysis” on page 15-10
- “Annotate Blocks for Known Errors or Coding-Rule Violations” on page 15-12

Model Configuration for Code Generation and Analysis

To facilitate Polyspace code analysis and the review of results:

- There are certain settings that you should apply to your model before generating code. See “Recommended Model Settings for Code Analysis” on page 15-5.
- The Polyspace plug-in for Simulink software allows you to check your model configuration before starting the Polyspace software. See “Check Simulink Model Settings” on page 15-7
- You can constrain signals in your model to lie within specified ranges. See “Specify Signal Ranges” on page 16-3.
- You can highlight blocks that you know contain checks or coding rule violations. See “Annotate Blocks for Known Errors or Coding-Rule Violations” on page 15-12.

Configure Simulink Model

To configure a Simulink model for code generation and analysis:

- 1 Open Model Explorer.
- 2 From the Model Hierarchy tree, expand the model node.
- 3 Select **Configuration > Code Generation**, which displays Code Generation configuration parameters.
- 4 Select the **General** tab, and then set the **System target file** to `ert.tlc` (Embedded Coder).
- 5 In the **Report** tab, select:
 - **Create code-generation report**
 - **Code-to-model** navigation.
- 6 In the **Templates** tab, clear **Generate an example main program**.
- 7 In the **Interface** tab, select **Suppress error status in real-time model data structure**.
- 8 Click **Apply**.
- 9 Select **Configuration > Solver**, which displays Solver configuration parameters.
- 10 In the **Solver options** section, set:
 - **Type** to Fixed-step.
 - **Solver** to discrete (no continuous states).
- 11 Click **Apply**.
- 12 Select **Configuration > Optimization**, which displays Optimization configuration parameters. Then:
 - On the **General** tab, in the **Data initialization** section, select the **Remove root level I/O zero initialization** check box.

- On the **General** tab, clear the **Use memset to initialize floats and doubles to 0.0** check box
- On the **Signals and Parameters** tab, in the **Simulation and code generation** section, select the **Inline parameters** check box.

13 Save your model.

Recommended Model Settings for Code Analysis

For Polyspace analyses, you should configure your model with the following settings before generating code.

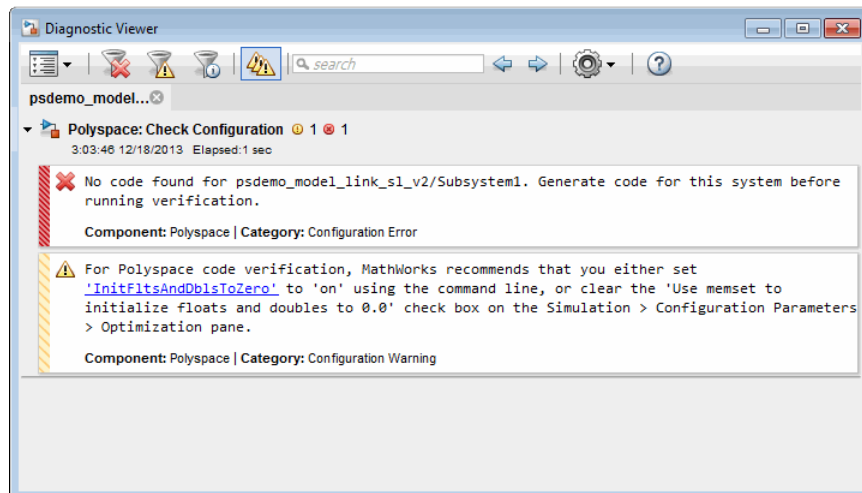
Parameter	Recommended value	How you specify value in Configuration Parameters dialog box	If you do not use recommended value...
InitFltsAndDblsToZero	'on'	Select check box Optimization > Use memset to initialize floats and doubles to 0.0	Warning
InlineParams	'on'	Select check box Optimization > Signals and Parameters > Inline parameters	Warning
MatFileLogging	'off'	Clear check box Code Generation > Interface > MAT-file logging	Warning
Solver	'FixedStepDiscrete'	Select discrete (no continuous states) from Solver > Solver drop-down list	Warning
SystemTargetFile	'ert.tlc'	Specify <code>ert.tlc</code> (for Embedded Coder) in Code Generation > System target file	Error

Parameter	Recommended value	How you specify value in Configuration Parameters dialog box	If you do not use recommended value...
GenerateComments	'on'	Select check box Code Generation > Comments > Include Comments	Error
ZeroExternalMemoryAtStartup	'off' when Configuration Parameters > Polyspace > Data Range Management > Output is Global assert	Clear check box Optimization > Remove root level I/O zero initialization	Warning

Check Simulink Model Settings

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.



You can also set the configuration check to run before you run 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, the software produces warnings when you run the analysis.

Related Examples

- “Check Simulink Model Settings Before Analysis” on page 15-10

Concepts

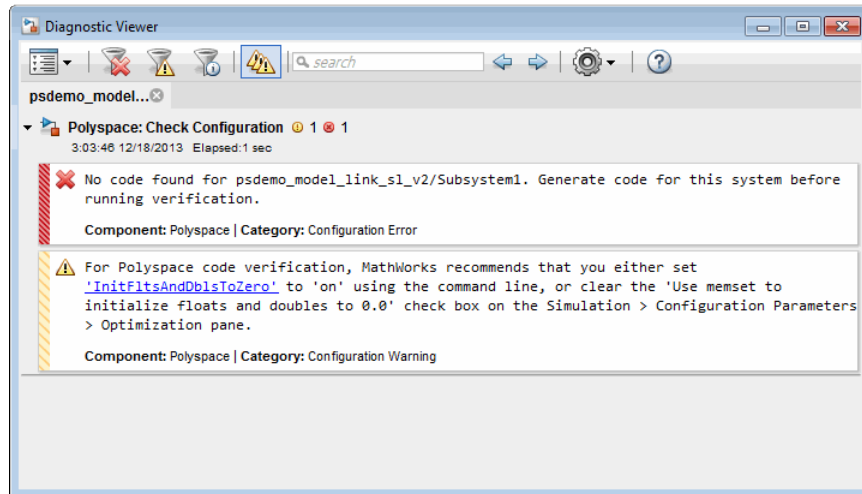
- “Recommended Model Settings for Code Analysis” on page 15-5

Check Simulink Model Settings Before Code Generation

Before generating code, you can check your model settings against the “Recommended Model Settings for Code Analysis” on page 15-5.

- 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** From the **Check model before generating code** drop-down list, select either:
 - On (stop for warnings)
 - On (proceed with warnings)
- 5** Select **Build** or **Generate Code**.

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 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.

Related Examples

- “Check Simulink Model Settings Before Analysis” on page 15-10
- “Check Simulink Model Settings” on page 15-7

Concepts

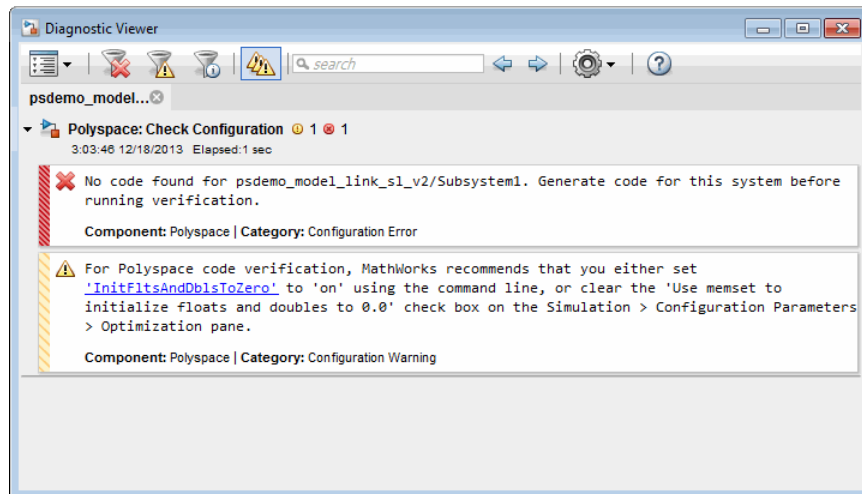
- “Recommended Model Settings for Code Analysis” on page 15-5

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 From the **Check configuration before verification** menu, select either:
 - On (stop for warnings) — will
 - On (proceed with warnings)
- 3 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.

Related Examples

- “Check Simulink Model Settings” on page 15-7

Concepts

- “Recommended Model Settings for Code Analysis” on page 15-5

Annotate Blocks for Known Errors or Coding-Rule Violations

You can annotate individual blocks in your Simulink model to inform Polyspace software of known defects, run-time checks, or coding-rule violations. This allows you to highlight and categorize previously identified results, so you can focus on reviewing new results.

The Polyspace Results Manager perspective displays the information that you provide with block annotations.

- 1** In the Simulink model window, right-click the block you want to annotate.
- 2** From the context menu, select **Polyspace > Annotate Selected Block > Edit**. The Polyspace Annotation dialog box opens.

Description

You can annotate blocks in your Simulink model to inform Polyspace software of known run-time checks or coding-rule violations. This allows you to highlight previously identified checks in your verification results, so you can focus on new checks.

Annotation

Annotation type:

Only 1 check

Select RTE check kind:

Status:

Classification:

Comment:

- 3** From the **Annotation type** drop-down list, select one of the following:
 - **Check** — To indicate a Code Prover run-time error
 - **Defect** — To indicate a Bug Finder defect
 - **MISRA-C** — To indicate a MISRA C coding rule violation
 - **MISRA-C++** — To indicate a MISRA C++ coding rule violation
 - **JSF** — To indicate a JSF C++ coding rule violation
- 4** If you want to highlight only one kind of result, select **Only 1 check** and the relevant error or coding rule from the **Select RTE check kind** (or **Select defect kind**, **Select MISRA rule**, **Select MISRA C++ rule**, or **Select JSF rule**) drop-down list.

If you want to highlight a list of checks, clear **Only 1 check**. In the **Enter a list of checks** (or **Enter a list of defects**, or **Enter a list of rule numbers**) field, specify the errors or rules that you want to highlight.

5 Select a **Status** to describe how you intend to address the issue:

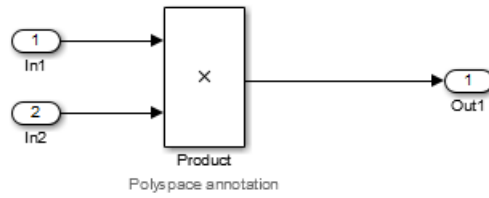
- Fix
- Improve
- Investigate
- Justify with annotations
(This status also marks the result as justified.)
- No Action Planned
(This status also marks the result as justified.)
- Other
- Restart with different options
- Undecided

6 Select a **Classification** to describe the severity of the issue:

- High
- Medium
- Low
- Not a defect

7 In the **Comment** field, enter additional information about the check.

8 Click **OK**. The software adds the Polyspace annotation is to the block.



Model Link for Polyspace Code Prover

- “Install Polyspace Plug-In for Simulink” on page 16-2
- “Specify Signal Ranges” on page 16-3
- “Annotate Code to Justify Polyspace Checks” on page 16-8
- “Configure Data Range Settings” on page 16-10
- “Main Generation for Model Verification” on page 16-13
- “Embedded Coder Considerations” on page 16-15
- “TargetLink Considerations” on page 16-22
- “Generate and Verify Code with Configured Model” on page 16-25
- “View Results in Polyspace® Code Prover™” on page 16-27
- “Identify Errors in Simulink Models” on page 16-29

Install Polyspace Plug-In for Simulink

The Simulink plug-in supports the four previous releases of MATLAB. For example, the R2014a version of the Polyspace plug-in supports MATLAB R2012a through R2014a.

By default, when you install Polyspace R2013b or later, the Simulink plug-in is installed and connected to MATLAB.

You can also install the Polyspace plug-in for Simulink on a previous version of MATLAB. You can then use the latest verification software with an older version of Embedded Coder or TargetLink®. However, if you use a cross-version installation of Polyspace and MATLAB, local batch analyses can only be submitted from the Polyspace environment or `polyspaceCodeProver`.

Note To install a newer version of Polyspace on MATLAB R2013b or later, you must have MATLAB installed without the corresponding version of Polyspace.

To install the Polyspace plug-in on a previous version of MATLAB:

- 1 Using an account with read/write privileges, open the older version of MATLAB.
- 2 Change your **Current Folder** to `matlabroot\polyspace\toolbox\pslink\pslink`. `matlabroot` is the Simulink plug-in that you want to install, for example, `C:\Program Files\MATLAB\R2014a`.
- 3 If you have a previous version of Polyspace installed, execute the `pslinksetup('uninstall')` command to uninstall it. This command does not work with MATLAB R2013b or later (see the preceding Note).
- 4 Execute the `pslinksetup('install')` command to install the new version of Polyspace.

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. This can reduce the number of orange checks in your verification 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 16-3.
- Constraining signals through the base workspace. See “Specify Signal Range through Base Workspace” on page 16-5.

Note You can also manually define data ranges using the DRS feature in the Polyspace verification environment. If you manually define a DRS file, the software automatically appends any signal range information from your model to the DRS file. However, manually defined DRS 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 16-5.

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.

The image shows a dialog box titled "Inport" with a "Signal Attributes" tab selected. The dialog contains the following fields and options:

- Output function call:** (unchecked)
- Minimum:** -15
- Maximum:** 15
- Data type:** Inherit: auto (dropdown menu)
- Lock output data type setting against changes by the fixed-point tools:** (unchecked)
- Port dimensions (-1 for inherited):** -1
- Variable-size signal:** Inherit (dropdown menu)
- Sample time (-1 for inherited):** -1
- Signal type:** auto (dropdown menu)
- Sampling mode:** auto (dropdown menu)

Buttons at the bottom include a help icon, OK, Cancel, and Help.

- 5** Click **OK**.

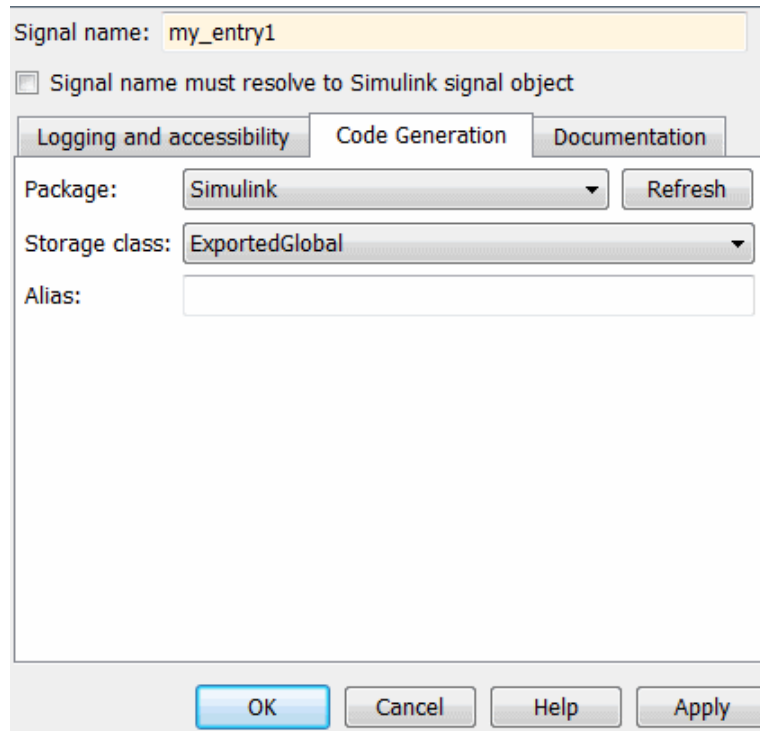
Specify Signal Range through Base Workspace

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 16-3.

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** From the **Package** drop-down menu, select `Simulink`.
 - e** In the **Storage class** drop-down menu, select `ExportedGlobal`.



- f** Click **OK**, which applies your changes and closes the dialog box.

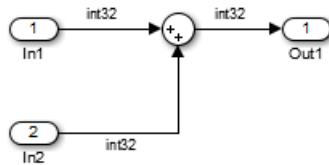
Note For information about supported storage classes, see “Data Range Specification” on page 16-16.

- 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** Click the **Add Simulink Signal** button to create a signal. 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**.

Annotate Code to Justify Polyspace Checks

A verification of Embedded Coder generated code might highlight overflows for certain operations that are legitimate because of the way Embedded Coder 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 }

```

Embedded Coder software 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, a 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**.

When you generate code, the Embedded Coder software annotates the 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:0vOk*/ sat_add_U.In2;
```

When you run a verification using Polyspace Code Prover, the software uses the annotations to justify the operator-related orange checks and assigns the Not a defect classification to the checks.

Configure Data Range Settings

There are two approaches to code verification, which can produce results that are slightly different:

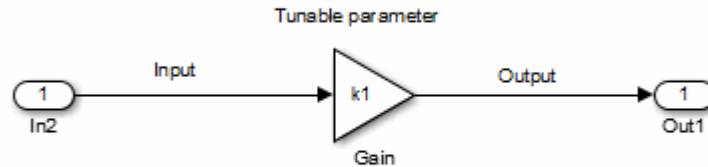
- **Contextual Verification** — Prove code does not generate run-time errors under predefined working conditions. This limits the scope of the verification to specific variable ranges, and verifies the code within these ranges.
- **Robustness Verification** — Prove code generate run-time errors for all verification conditions, including “abnormal” conditions for which the code was not designed. This can be thought of as “worst case” verification.

For more information, see:

- “Choose Robustness or Contextual Verification” on page 2-4.
- Data Range Specification — Model Link SL
- Data Range Specification — Model Link TL

Note The software supports data range management only with Simulink Version 7.4 (R2009b) or later.

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 Model Link** 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 16-3.
 - 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 16-3. 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**.

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.

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
```

Embedded Coder Considerations

In this section...

“Subsystems” on page 16-15

“Default Options” on page 16-15

“Data Range Specification” on page 16-16

“Recommended Polyspace options for Verifying Generated Code” on page 16-16

“Hardware Mapping Between Simulink and Polyspace” on page 16-21

Subsystems

A dialog will be presented after clicking on the Polyspace for Embedded Coder block if multiple subsystems are present in a diagram. Simply select the subsystem to analyze from the list. The subsystem list is generated from the directory structure from the code that has been generated.

Default Options

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

```
-sources path_to_source_code
-desktop
-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
-OS-target no-predefined-OS
-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. See “Configure Data Range Settings” on page 16-10.

The software automatically creates a Polyspace “Data Range Specifications (DRS)” on page 6-55 file using information from the MATLAB workspace and block parameters.

You can also manually define a DRS file using the Project Manager perspective of the Polyspace verification environment. 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 Model Link** pane. These values override the corresponding option values in the **Configuration** pane of the Project Manager.

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 Project Manager 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
Target & Compiler		
-D	See Comments	Defines macro compiler flags used during compilation. Use one -D for each line of the Embedded Coder generated <code>defines.txt</code> file. Polyspace Model Link™ SL does not do this by default.

Option	Recommended Value	Comments
-OS-target	Visual	<p>Specifies the operating system target for Polyspace stubs.</p> <p>This information allows the verification to use system definitions during preprocessing to analyze the included files.</p>
-target	i386	<p>Specifies the target processor type. This allows the verification to consider the size of fundamental data types and the endianness of the target machine.</p> <p>You can configure and specify generic targets. For more information, see “Target Processor Configuration”.</p>
-dos	Selected	<p>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:</p> <ul style="list-style-type: none"> • Header files – All include folders specified (-I option) • Source files – All source files selected for the verification (-sources option)

Option	Recommended Value	Comments
Verification Assumptions		
-allow-negative-operand-shift	Selected	<p>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, $-2 \ll 2$. If you select this option, Polyspace considers the operation to be valid. For the given example, $-2 \ll 2 = -8$.</p>
-ignore-float-rounding	Selected	<p>Specifies how the verification rounds floats.</p> <p>If this option is not selected, the verification rounds floats according to the IEEE® 754 standard – simple precision on 32-bits targets and double precision on targets that define double as 64-bits.</p> <p>When you select this option, the verification performs exact computation.</p> <p>Selecting this option can lead to results that differ from "real life," depending on the actual compiler and target. Some paths may be reachable (or not reachable) for the verification while they are not reachable (or are reachable) for the actual compiler and target.</p> <p>However, this option reduces the number of unproven checks caused by float approximation.</p>

Option	Recommended Value	Comments
Precision		
-0	2	<p>Specifies the precision level for the verification.</p> <p>Higher precision levels provide higher selectivity at the expense of longer verification time.</p> <p>Begin with the lowest precision level. You can then address red errors and gray code before rerunning the Polyspace verification using higher precision levels.</p> <p>Benefits:</p> <p>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.</p> <p>The precision level specifies the algorithms used to model the program state space during verification:</p> <ul style="list-style-type: none"> • -00 corresponds to static interval verification. • -01 corresponds to complex polyhedron model of domain values. • -02 corresponds to more complex algorithms to closely model domain values (a mixed approach with integer lattices and complex polyhedrons). • -03 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	Recommended Value	Comments
-to	<p>C source compliance checking – For C code, when checking coding rule compliance only.</p> <p>C++ source compliance checking – For C++ code, when checking coding rule compliance only.</p> <p>pass0 – When verifying code for the first time.</p> <p>pass4 – When performing subsequent verifications of code.</p>	<p>Specifies the phase after which the verification stops. Each verification phase improves the selectivity of your results, but increases the overall verification time.</p> <p>Improved selectivity can make results review more efficient, and hence make bugs in the code easier to isolate.</p> <p>Begin by running <code>-to pass0</code> (Software Safety Analysis level 0) You can then address red errors and gray code before relaunching verification using higher integration levels.</p>

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.

Note The software creates a generic target for the verification.

TargetLink Considerations

In this section...

“TargetLink Support” on page 16-22

“Subsystems” on page 16-22

“Default Options” on page 16-22

“Data Range Specification” on page 16-23

“Lookup Tables” on page 16-24

“Code Generation Options” on page 16-24

TargetLink Support

For Windows, Polyspace Code Prover is tested with releases 3.1, 3.2, and 3.3 of the dSPACE® Data Dictionary version and TargetLink Code Generator.

As Polyspace Code Prover extracts information from the dSPACE Data Dictionary, you must regenerate the code before performing a verification.

Subsystems

A dialog will be presented after clicking on the Polyspace for TargetLink block if multiple subsystems are present in a diagram. Simply select the subsystem to analyze from the list.

Default Options

The following default options are set by the tool:

```
-I path to source code
-desktop
-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
```

Note *dspaceroot* and *matlabroot* are the dSPACE and MATLAB tool installation directories respectively.

Data Range Specification

You can constrain inputs, parameters, and outputs to lie within specified data ranges. See “Configure Data Range Settings” on page 16-10.

The software automatically creates a Polyspace “Data Range Specifications (DRS)” on page 6-55 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 allows Polyspace software to model every value that is legal for the system during verification. Carefully defining the min-max information in the model allows the verification 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 using the Project Manager perspective of the Polyspace Verification Environment. 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 you may need to either rename one of or variables or disable this option in Polyspace configuration.

Lookup Tables

The tool by default provides stubs for the lookup table functions. This behavior can be disabled from the Polyspace menu. The dSPACE data dictionary is used to define the range of their return values. Note that a lookup table that uses extrapolation will return full range for the type of variable that it returns.

Code Generation Options

From the TargetLink Main Dialog, it is recommended to set the option Clean code and deselect the option Enable sections/pragmas/inline/ISR/user attributes.

When installing the Polyspace Model Link TL product, the `tlcgOptions` variable has been updated with 'PolyspaceSupport', 'on' (see variable in 'C:\dSPACE\Matlab\Tl\config\codegen\tl_pre_codegen_hook.m' file).

Generate and Verify Code with Configured Model

You can generate Embedded Coder code from the configured model `psdemo_model_link_sl`. You can then run a Polyspace verification on the generated code.

To open `psdemo_model_link_sl` in the Simulink model window:

- 1 In the MATLAB Command Window, enter `psdemo_model_link_sl`.

This command opens the `psdemo_model_link_sl` model that is compatible with your version of MATLAB (either `psdemo_model_link_sl`, `psdemo_model_link_sl_v1`, or `psdemo_model_link_sl_v2`).

To generate code and start the Polyspace verification:

- 1 Double-click the Reinstall the demo block to generate the legacy code related to the S-function.
- 2 If you want to apply data ranges to the input parameters, double-click the green block Use input constraints. To remove the data range constraints, double-click the orange block Worst case inputs.
- 3 Right-click the subsystem controller.
- 4 From the context-menu, select **C/C++ Code > Build This Subsystem**.
- 5 In the Build code for Subsystem dialog box, click **Build** to generate code. When the code generation is complete, the code generation report opens.
- 6 Right-click the subsystem controller. From the context menu, select **Polyspace > Verify Code Generated for > Selected Subsystem**. The verification starts.

To monitor the progress of the verification:

- If you specified server verification, select **Code > Polyspace > Open Spooler**. Use the Polyspace Queue Manager (Spooler) to monitor progress.
- If you specified client verification, you can monitor progress from the Command Window.

Once the verification is complete, to display the results:

- 1** Select **Code > Polyspace > Open Results > For Generated Code**.
- 2** In the Polyspace environment, select **File > Open Result**.
- 3** Use the Open Results dialog box to navigate to the specified results folder, for example, `C:\Polyspace_Results\controller`.
- 4** Select the results file, for example, `RTE_px_controller_LAST_RESULTS.pscp`. Then click **Open**. The software displays the results in the Results Manager perspective.

View Results in Polyspace Code Prover

When a verification completes, you can view the results using the Results Manager perspective of the Polyspace Code Prover.

To view your results:

- 1 From the Simulink model window, select **Code > Polyspace > Open Results**.

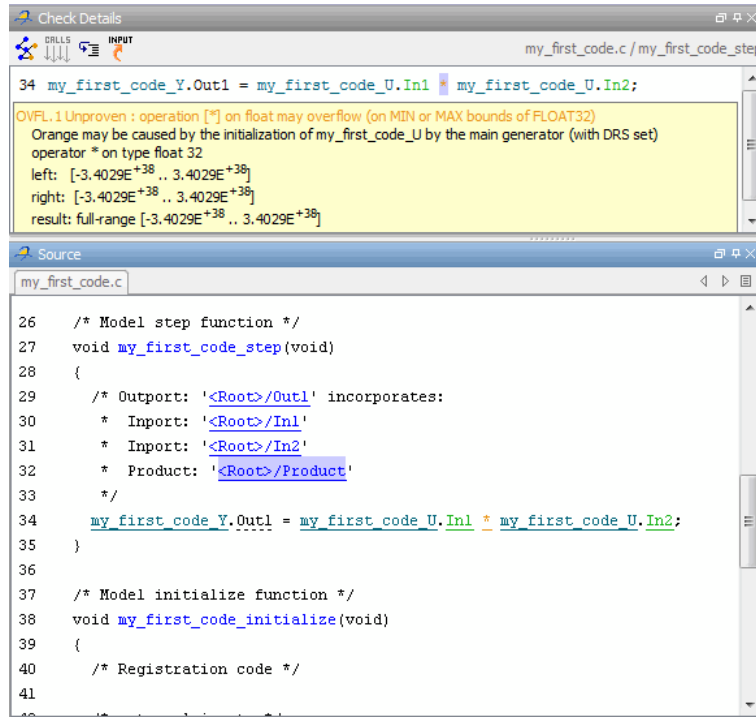
Note If you set **Model reference verification depth** to All and selected **Model by model verification**, the Select the Result Folder to Open in Polyspace dialog box opens. The dialog box displays a hierarchy of referenced models from which the software generates code. To view the verification results 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 Results Manager perspective of the Polyspace Code Prover opens.

- 2 On the **Results Summary** tab, click any check to review additional information.

In this example, the **Check Details** pane shows information about the orange check, and the **Source** pane shows the source code containing the orange check.



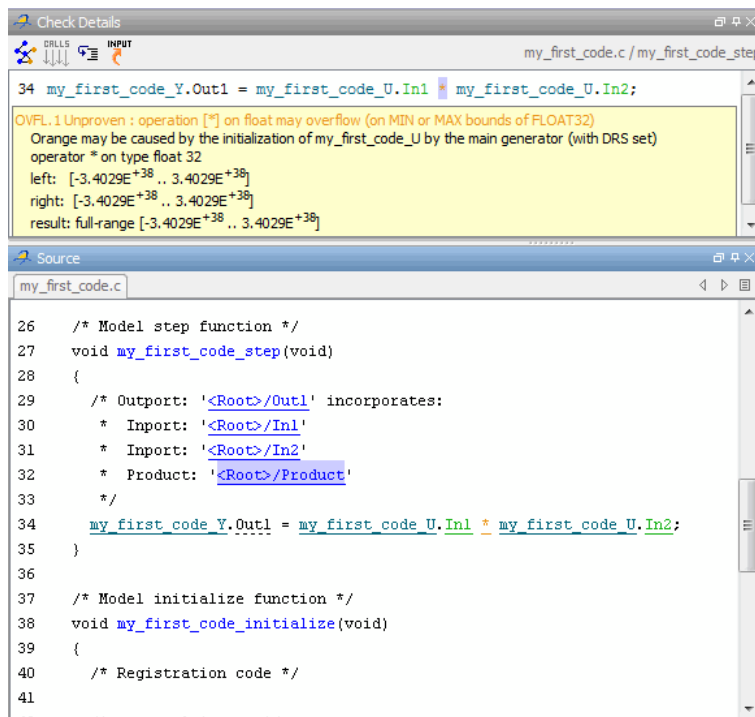
For more information on reviewing run-time checks, see “Run-Time Error Review”.

For information on specific checks, see “Run-Time Check Reference”.

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 **Check 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 16-3.

Applying either solution should address the issue and cause the orange check to turn green.

If your operating system is Windows Vista™ or Windows 7, you may encounter problems with the trace-back functionality if one of the following conditions apply:

- User Account Control (UAC) is enabled.
- You do not have administrator privileges.

If you have a MATLAB session running and your model is open, a possible workaround is:

- 1 Open a DOS window in administrator mode.
- 2 Go to your MATLAB installation folder.
- 3 From the bin folder, enter `matlab -regserver`.
- 4 Click the link again.

If your model extensively uses block coloring, the coloring from this feature may interfere with the colors already in your model. To change the color of blocks when they are linked to Polyspace results, use the following commands:

```
HILITEDATA = struct('HiliteType', 'find', 'ForegroundColor', 'black', ...  
                  'BackgroundColor', color);  
set_param(0, 'HiliteAncestorsData', HILITEDATA);
```

Where *color* is one of the following:

- 'cyan'
- 'magenta'
- 'orange'
- 'lightBlue'
- 'red'
- 'green'
- 'blue'
- 'darkGreen'

Configure Code Analysis Options

- “Polyspace Configuration for Generated Code” on page 17-2
- “Include Handwritten Code” on page 17-3
- “Specify Remote Analysis” on page 17-5
- “Configure Analysis Depth for Referenced Models” on page 17-6
- “Specify Location of Results” on page 17-7
- “Check Coding Rules Compliance” on page 17-8
- “Configure Polyspace Options from Simulink” on page 17-10
- “Configure Polyspace Project Properties” on page 17-11
- “Create a Polyspace Configuration File Template” on page 17-12
- “Specify Header Files for Target Compiler” on page 17-15
- “Open Polyspace Results Automatically” on page 17-16
- “Remove Polyspace Options From Simulink Model” on page 17-17

Polyspace Configuration for Generated Code

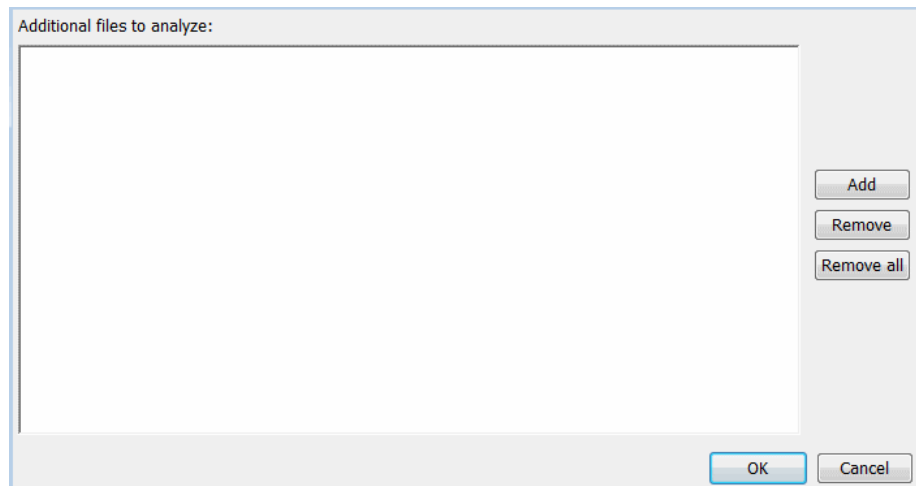
You do not have to manually create a Polyspace project or specify Polyspace options before running an analysis for your generated code. By default, Polyspace automatically creates a project and extracts the required information from your model. However, you can modify or specify additional options for your analysis:

- You may incorporate separately created code within the code generated from your Simulink model. See “Include Handwritten Code” on page 17-3.
- 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 16-10.
- You may customize the options for your analysis. For example, to specify the target environment or adjust precision settings. See “Configure Polyspace Options from Simulink” on page 17-10 and “Recommended Polyspace options for Verifying Generated Code” on page 16-16.
- You may create specific configurations for batch runs. See “Create a Polyspace Configuration File Template” on page 17-12.
- If you want to analyze code generated for a 16-bit target processor, you must specify header files for your 16-bit compiler. See “Specify Header Files for Target Compiler” on page 17-15.

Include Handwritten Code

Files such as S-function wrappers are, by default, not part of the Polyspace analysis. However, you can add these files manually.

- 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**.

Specify 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 **Distributed Computing** pane.
- 4** Select the **Batch** checkbox.
- 5** Close the configuration window and save your changes.
- 6** Select **Apply**.

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. It does not matter whether you open the **Polyspace** pane from the top model window (**Code > Polyspace > Options**) or through the right-click context menu of a particular Model block within the top model. However, you can run analyses for code generated from specific Model blocks. See “Run Analysis for Embedded Coder” on page 18-5.

Specify Location of Results

- 1** From the Simulink model window, select **Code > Polyspace > Options**. The Configuration Parameters dialog box opens with the Polyspace pane displayed.
- 2** In the **Output folder** field, specify the full path for your results folder. By default, the software stores results in `C:\Polyspace_Results\results_model_name`.
- 3** If you want to avoid overwriting results from previous analyses, select the **Make output folder name unique by adding a suffix** check box. 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.

Check Coding Rules Compliance

You can check compliance with MISRA C and MISRA AC AGC coding rules directly from your Simulink model.

In addition, you can choose to run coding rules checking either with or without full code analysis.

To configure coding rules checking:

- 1 From the Simulink model window, select **Code > Polyspace > Options**. The **Polyspace** pane opens.
- 2 In the **Settings from** drop-down menu, select the type of analysis you want to perform.

Depending on the type of code generated, different settings are available. The following tables describe the different settings.

C Code Settings

Setting	Description
Project configuration	Run Polyspace using the options specified in the Project configuration .
Project configuration and MISRA AC AGC rule checking	Run Polyspace using the options specified in the Project configuration and check compliance with the MISRA AC-AGC rule set.
Project configuration and MISRA rule checking	Run Polyspace using the options specified in the Project configuration and check compliance with MISRA C coding rules.

C Code Settings (Continued)

Setting	Description
MISRA AC AGC rule checking	Check compliance with the MISRA AC-AGC rule set. Polyspace stops after rules checking.
MISRA rule checking	Check compliance with MISRA C coding rules. Polyspace stops after rules checking.

C++ Code Settings

Setting	Description
Project configuration	Run Polyspace using the options specified in the Project configuration .
Project configuration and MISRA C++ rule checking	Run Polyspace using the options specified in the Project configuration and check compliance with the MISRA C++ coding rules.
Project configuration and JSF C++ rule checking	Run Polyspace using the options specified in the Project configuration and check compliance with JSF C++ coding rules.
MISRA C++ rule checking	Check compliance with the MISRA C++ coding rules. Polyspace stops after rules checking.
JSF C++ rule checking	Check compliance with JSF C++ coding rules. Polyspace stops after rules checking.

3 Click **Apply** to save your settings.

Configure Polyspace Options from Simulink

From Simulink, you can use a simplified version of the Polyspace Project Manager to customize Polyspace options. For example, you can specify the target processor type, target operating system, and compilation flags.

To open the **Configuration** pane of the Project Manager:

- 1** From the Simulink model window, select **Code > Polyspace > Options**. The **Polyspace** pane opens.
- 2** Click **Configure**. The Polyspace Configuration pane opens.

The first time you open the configuration, the software sets the following options:

- **Target operating system** (-OS-target) – Set to no-predefined-OS
- **Use result folder** (-results-dir) – Set to results_*modelName*

The software also configures other options automatically, but the settings depend on the code generator used.


- 3** Set other options required by your application.

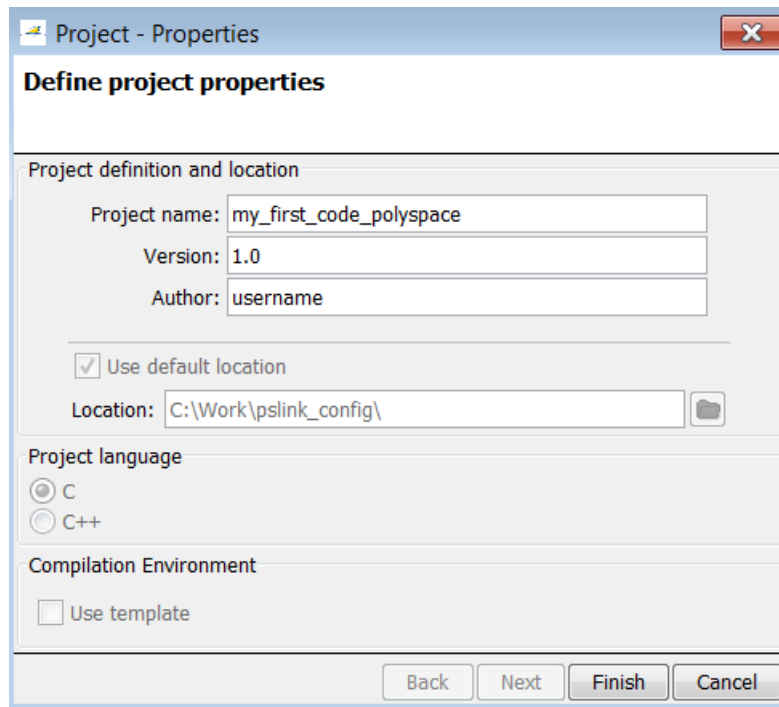
For recommended options for verifying generated code, see “Recommended Polyspace options for Verifying Generated Code” on page 16-16.

For descriptions of advanced options, see “Analysis Options for C Code” or “Analysis Options for C Code”.

Configure Polyspace Project Properties

You can specify project properties, for example, your project name, through the Polyspace Project - Properties dialog box. To open this dialog box:

- 1 From the Simulink model window, select **Code > Polyspace > Options**. The **Polyspace** pane opens.
- 2 Click **Configure**. The Polyspace configuration window opens.
- 3 On the Project Manager toolbar, click the **Project properties** icon .



Project - Properties

Define project properties

Project definition and location

Project name: my_first_code_polyspace

Version: 1.0

Author: username

Use default location

Location: C:\Work\pslink_config\

Project language

C

C++

Compilation Environment

Use template

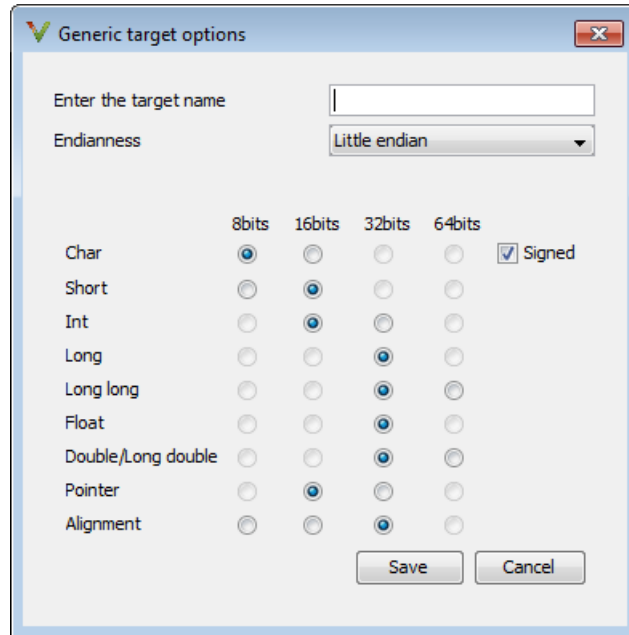
Back Next Finish Cancel

Create a Polyspace Configuration File Template

During a batch run, you may want use different configurations. At the MATLAB command-line, use `pslinkfun('settemplate',...)` to apply a configuration defined by a configuration file template.

To create a configuration file template:

- 1** In the Simulink model window, select **Code > Polyspace > Options**. The **Polyspace** pane opens.
- 2** Click **Configure**. The Project Manager opens, displaying the **Configuration** pane. 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**.

- 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, click the **+** button. In the new line, enter the required text.

To remove a macro, select the macro and click the **-** button.

Note If you use the LCC cross-compiler, then you must specify the `MATLAB_MEX_FILE` macro.

- Save your changes and close the Project Manager.

- 7** Make a copy of the updated project configuration file, for example, `my_first_code_polyspace.psprj`.
- 8** Rename the copy, for example, `my_cross_compiler.psprj`. This is your new configuration file template.

To use a configuration template, run the `pslinkfun` command in the MATLAB Command Window. For example:

```
pslinkfun('settemplate','C:\Work\my_cross_compiler.psprj')
```

Specify Header Files for Target Compiler

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 software produces 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.

To specify header file folders (or header files) for your compiler:

- 1** Open the Polyspace **Configuration** pane. From the Simulink model window, select **Code > Polyspace > Options**. The **Polyspace** pane opens.
- 2** Click **Configure**. The Project Manager opens, displaying the **Configuration** pane.
- 3** From the **Configuration** tree, expand the **Target & Compiler** node.
- 4** Select **Target & Compiler > Environment Settings**.
- 5** In the **Include folders** (or **Include**) section, specify a folder (or header file) path by doing one of the following:
 - Click the **+** button. Then, in the text field, enter the folder (or file) path.
 - Click the folder button and use the Open file 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 **-**.

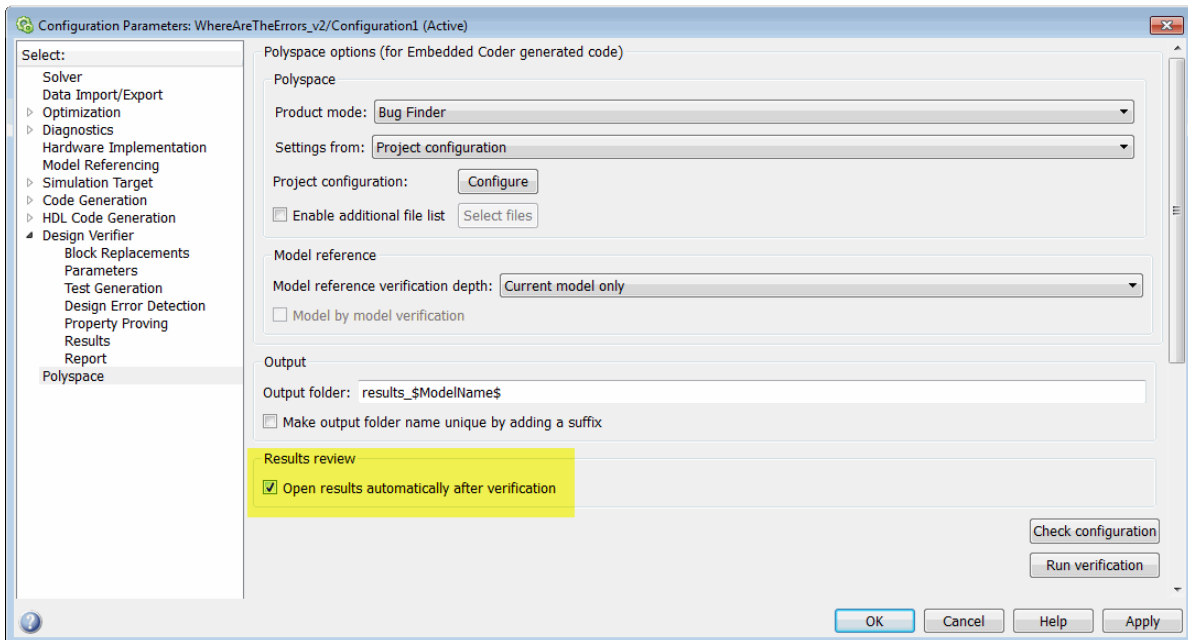
Open Polyspace Results Automatically

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 webpage 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 Results Manager perspective.

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.

Remove Polyspace Options From Simulink Model

You can remove Polyspace configuration information from your Simulink model.

For a top model:

- 1** Select **Code > Polyspace > Remove Options from Current Configuration**.
- 2** Save the model.

For a Model block or subsystem:

- 1** Right-click the Model block or subsystem.
- 2** From the context menu, select **Polyspace > Remove Options from Current Configuration**.
- 3** Save the model.

Run Polyspace on Generated Code

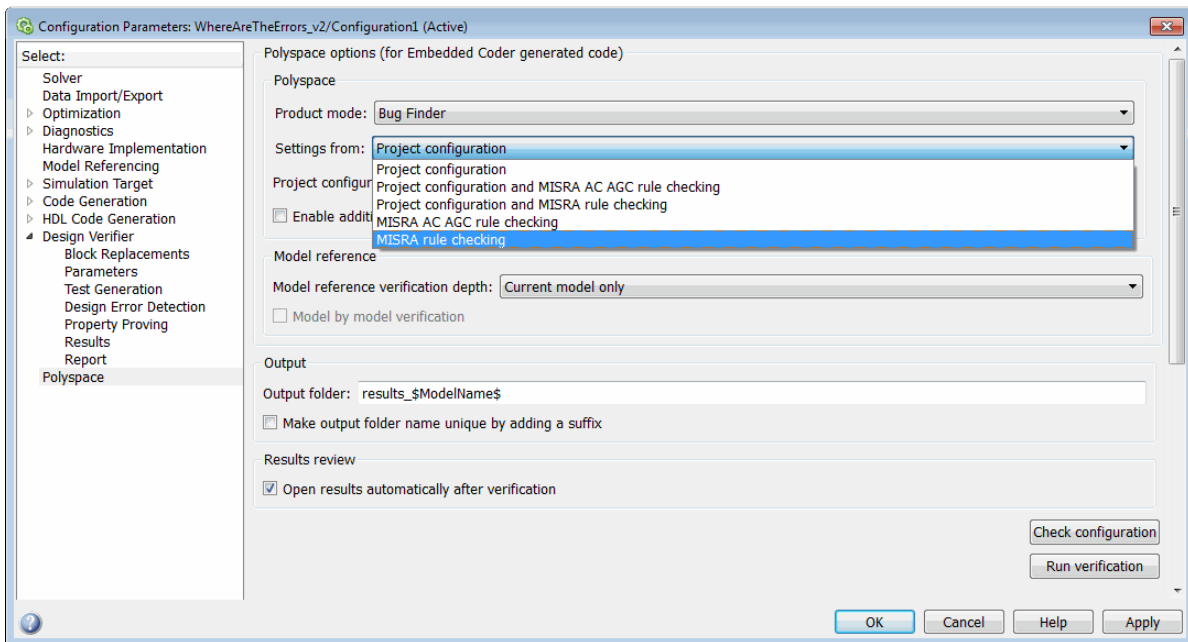
- “Specify Type of Analysis to Perform” on page 18-2
- “Run Analysis for Embedded Coder” on page 18-5
- “Run Analysis for TargetLink” on page 18-7
- “Monitor Progress” on page 18-8

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.

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.

Depending on the type of code generated, different settings are available. The following tables describe the different settings.

C Code Settings

Setting	Description
Project configuration	Run Polyspace using the options specified in the Project configuration .
Project configuration and MISRA AC AGC rule checking	Run Polyspace using the options specified in the Project configuration and check compliance with the MISRA AC-AGC rule set.
Project configuration and MISRA rule checking	Run Polyspace using the options specified in the Project configuration and check compliance with MISRA C coding rules.
MISRA AC AGC rule checking	Check compliance with the MISRA AC-AGC rule set. Polyspace stops after rules checking.
MISRA rule checking	Check compliance with MISRA C coding rules. Polyspace stops after rules checking.

C++ Code Settings

Setting	Description
Project configuration	Run Polyspace using the options specified in the Project configuration .
Project configuration and MISRA C++ rule checking	Run Polyspace using the options specified in the Project configuration and check compliance with the MISRA C++ coding rules.

C++ Code Settings (Continued)

Setting	Description
Project configuration and JSF C++ rule checking	Run Polyspace using the options specified in the Project configuration and check compliance with JSF C++ coding rules.
MISRA C++ rule checking	Check compliance with the MISRA C++ coding rules. Polyspace stops after rules checking.
JSF C++ rule checking	Check compliance with JSF C++ coding rules. Polyspace stops after rules checking.

3 Click **Apply** to save your settings.

Run Analysis for Embedded Coder

To start Polyspace with:

- Code generated from the top model, from the Simulink model window, select **Code > Polyspace > Verify Code Generated for > Model**.
- All code generated as model referenced code, from the model window, 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. From the context menu, 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 Queue Manager.

The software writes status messages to a log file in the results folder, for example `Polyspace_R2013b_my_first_code_05_16_2013-18h40.log`

Run Analysis for TargetLink

To start the Polyspace software:

- 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, for example `Polyspace_R2013b_my_first_code_05_16_2013-18h40.log`

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 through the Polyspace Queue Manager

Note Verification of a 3,000 block model will take approximately one hour to verify, or about 15 minutes for each 2,000 lines of generated code.

Monitor Progress

In this section...
“Local Analyses” on page 18-8
“Remote Batch Analyses” on page 18-8

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. For example:

```
Polyspace_R2013b_my_first_code_05_16_2013-18h40.log
```

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 Queue Manager.

From Simulink, select **Code > Polyspace > Open Spooler**

For more information, see “Verification Management”.

Using Polyspace Software in the Eclipse IDE

- “Install Polyspace Plug-In for Eclipse” on page 19-2
- “Verify Code in the Eclipse IDE” on page 19-5

Install Polyspace Plug-In for Eclipse

In this section...
“Install Polyspace Plug-In for Eclipse IDE” on page 19-2
“Uninstall Polyspace Plug-In for Eclipse IDE” on page 19-4

Install Polyspace Plug-In for Eclipse IDE

You can install the Polyspace plug-in only after you:

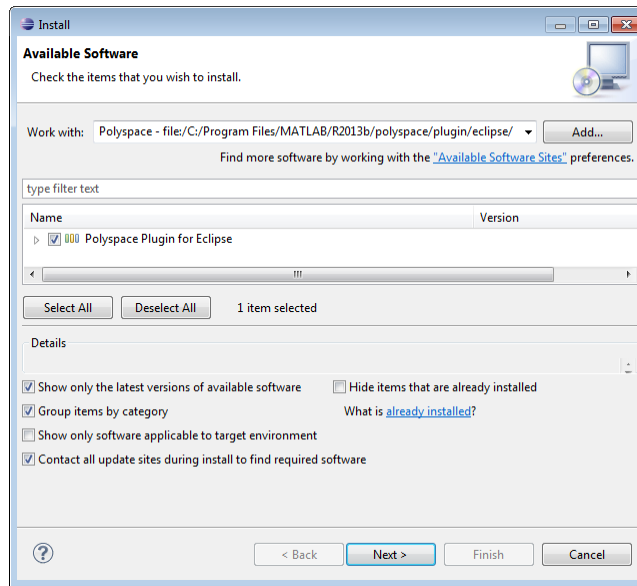
- Install and set up Eclipse™ Integrated Development Environment (IDE). For more information, see the Eclipse documentation at www.eclipse.org.
- Install Java 7. See Java documentation at www.java.com.
- Uninstall any previous Polyspace plug-ins. For more information, see “Uninstall Polyspace Plug-In for Eclipse IDE” on page 19-4.

To install the Polyspace plug-in:

- 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\matlab\polyspace\plugin\eclipse` folder. Then click **OK**.

MATLAB_Install is the installation folder for the Polyspace product, for example:

`C:\Program Files\MATLAB\R2013b`
- 6** Click **OK** to close the Add Repository dialog box.
- 7** On the Available Software page, select **Polyspace Plugin for Eclipse**.



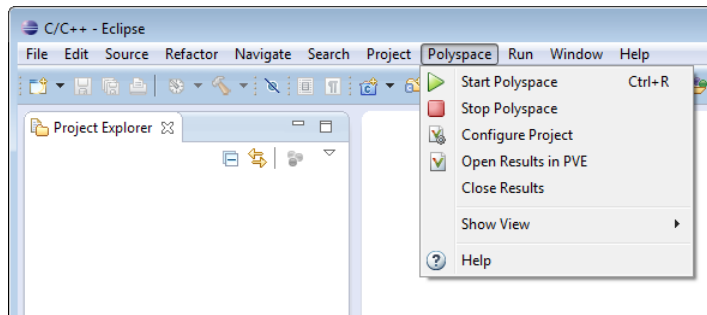
8 Click **Next**.

9 On the Install Details page, click **Next**.

10 On the Review Licenses page, review and accept the licence agreement. Then click **Finish**.

Once you install the plug-in, in the Eclipse editor, you'll see:

- A **Polyspace** menu
- A **Polyspace Run** view



Uninstall Polyspace Plug-In for Eclipse IDE

Before installing a new Polyspace plug-in, you must uninstall any previous Polyspace plug-ins:

- 1 In Eclipse, select **Help > About Eclipse**.
- 2 Select **Installation Details**.
- 3 Select the Polyspace plug-in and select **Uninstall**.

Follow the uninstall wizard to remove the Polyspace plug-in. You must restart Eclipse for changes to take effect.

Verify Code in the Eclipse IDE

In this section...

“Code Verification in the Eclipse IDE” on page 19-5

“Create an Eclipse Project” on page 19-5

“Set Up Polyspace Verification with Eclipse Editor” on page 19-6

“Start Verification from Eclipse Editor” on page 19-7

“Review Verification Results from Eclipse Editor” on page 19-7

Code Verification in the Eclipse IDE

You can use Polyspace software to verify code that you develop within the Eclipse Integrated Development Environment (IDE).

A typical workflow is:

- 1 Create an Eclipse project and develop code within your project.
- 2 Set up the Polyspace verification.
- 3 Start the verification.
- 4 Review the verification results. Fix run-time errors and restart the verification.

Install the Polyspace plug-in for Eclipse IDE before you verify code in Eclipse IDE. For more information, see “Install Polyspace Plug-In for Eclipse” on page 19-2.

Create an Eclipse Project

If your source files do not belong to an Eclipse project, then create a project using the Eclipse editor:

- 1 Select **File > New > C Project**.
- 2 Clear the **Use default location** check box.

- 3** Click **Browse** to navigate to the folder containing your source files, for example, C:\Test\Source_C.
- 4** In the **Project name** field, enter a name, for example, Demo_C.
- 5** In the **Project Type** tree, under **Executable**, select **Empty Project** .
- 6** Under **Toolchains**, select your installed toolchain, for example, MinGW GCC.
- 7** Click **Finish**. An Eclipse project is created.

For information on developing code within Eclipse IDE, refer to www.eclipse.org.

Set Up Polyspace Verification with Eclipse Editor

To configure your verification:

- 1** In **Project Explorer**, select the project or files that you want to verify.
- 2** Select **Polyspace > Configure Project** to open the **Configuration** pane in the Polyspace verification environment.
- 3** Select your options for the verification process.
- 4** Select **File > Save** to save your options.

For more information, see “Analysis Options for C Code”.

Note Your Eclipse compiler options for include paths (-I) and symbol definitions (-D) are automatically added to the list of Polyspace analysis options.

To view the -I and -D options in the Eclipse editor :

- 1 Select **Project > Properties** to open the Properties for Project dialog box.
 - 2 In the tree, under **C/C++ General** , select **Paths and Symbols** .
 - 3 Select **Includes** to view the -I options or **Symbols** to view the -D options.
-

Start Verification from Eclipse Editor

To start a Polyspace verification from the Eclipse editor:

- 1 Select the file, files, or class that you want to verify.
- 2 Either right-click and select **Start Polyspace Code Prover**, or select **Polyspace > Start Polyspace**.

You can see the progress of the verification in the **Polyspace Run** 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 on the **Results Summary** pane.

- 3 To stop a verification, select **Polyspace > Stop Polyspace**. Alternatively you can use the  button in the **Polyspace Run** view.

For more information, see “Monitor Progress of Verification” on page 8-16.

Review Verification Results from Eclipse Editor

You can examine results of the verification either in Eclipse or the Polyspace verification environment. After verification is over:

- View the results in Eclipse on the **Results Summary** pane. Select a check to see detailed information on the **Check Details** pane. For more information, see:
 - “Results Summary” on page 10-67
 - “Check Details” on page 10-86
- To open results in the Polyspace verification environment, select **Polyspace > Open Results in PVE**.

Tip Open results in the Polyspace verification environment if you want to save them for later reference. The results in Eclipse are overwritten every time a new verification is performed. However, any **Status**, **Classification** or **Comment** you enter is imported automatically into the new verification.

For information on reviewing and interpreting Polyspace results, see “Run-Time Error Review” .

Using Polyspace Software in Visual Studio

- “Install Polyspace Add-In for Visual Studio” on page 20-2
- “Verify Code in Visual Studio” on page 20-4

Install Polyspace Add-In for Visual Studio

Install Polyspace Add-In for Visual Studio

You can install the Polyspace add-in only after you:

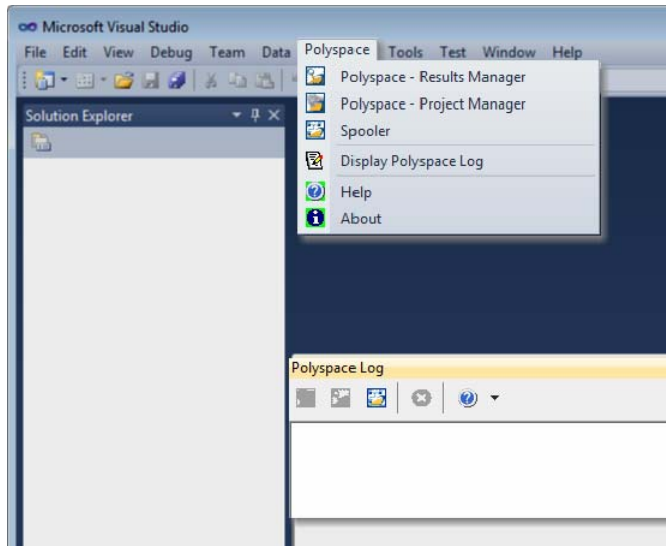
- Install Visual Studio.
- Uninstall any previous Polyspace add-ins. For more information see “Uninstall Polyspace Add-In for Visual Studio” on page 20-3.

To install the Polyspace add-in:

- 1** In the Visual Studio editor, select **Tools > Options** to open the Options dialog box.
- 2** Select the **Environment > Add-in/Macros Security** pane to display the list of Visual Studio add-in folders.
- 3** Select the following check boxes:
 - **Allow macros to run**
 - **Allow Add-in components to load**
- 4** Click **Add** to open the Browse For Folder dialog box.
- 5** Navigate to
`MATLAB_Install\matlab\polyspace\plugin\msvc\VS_version`
 - `MATLAB_Install` is the installation folder for the Polyspace product, for example:
`C:\Program Files\MATLAB\R2013b`
 - `VS_version` corresponds to the version of Visual Studio that you have installed, for example, 2010.
- 6** Click **OK** to close the Browse for Folder dialog box.
- 7** To close the Options dialog box, click **OK**.

You must restart Visual Studio for the changes to take effect. After you install the add-in, the Visual Studio editor has:

- A **Polyspace** menu
- A **Polyspace Log** view



Uninstall Polyspace Add-In for Visual Studio

Before installing a new Polyspace add-in, you must uninstall any previous Polyspace add-ins.

- 1 In the Visual Studio editor, select **Tools > Options** to open the Options dialog box.
- 2 Select the **Environment > Add-in/Macros Security** pane to display the list of Visual Studio add-in folders.
- 3 Select the Polyspace add-in and select **Remove**.
- 4 To close the Options dialog box, click **OK**.

You must restart Visual Studio for the changes to take effect.

Verify Code in Visual Studio

In this section...

“Code Verification in Visual Studio” on page 20-4

“Create Visual Studio Project” on page 20-4

“Verify Code in Visual Studio” on page 20-6

“Monitor Verification in Visual Studio” on page 20-14

“Review Verification Results in Visual Studio” on page 20-16

Code Verification in Visual Studio

You can apply the powerful code verification functionality of Polyspace software to code that you develop within the Visual Studio Integrated Development Environment (IDE).

A typical workflow is:

- 1** Use the Visual Studio editor to create a project and develop code within this project.
- 2** Set up the Polyspace verification by configuring analysis options and settings, and then start the verification.
- 3** Monitor the verification.
- 4** Review the verification results.

Before you can verify code in Visual Studio, you must install the Polyspace add-in for Visual.NET. For more information, see “Install Polyspace Add-In for Visual Studio” on page 20-2.

Create Visual Studio Project

If your source files do not belong to a Visual Studio project, you can create a project using the Visual Studio editor:

- 1** Select **File > New > Project > New > Project Console Win32** to create a project space
- 2** Enter a project name, for example, CppExample.
- 3** Save this project in a specific location, for example, C:\Polyspace\Visual. The software creates some files and a Project Console Win32.

To add files to your project:

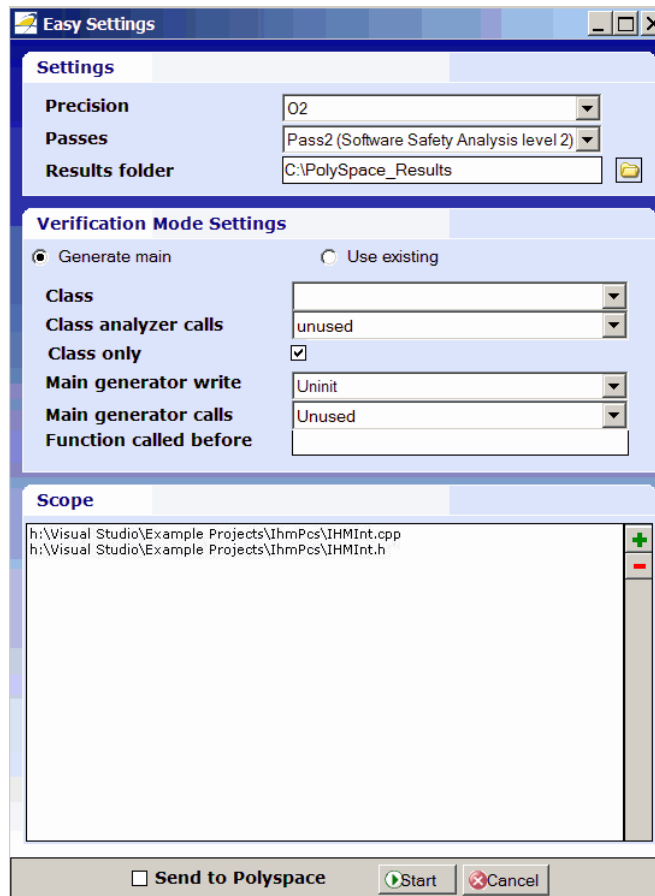
- 1** Select the **Browse the solution** tab.
- 2** Right-click the project name. From the pop-up menu, select **Add > Add existing element** .
- 3** Add the files you want to the project (for example, CppExample).

Verify Code in Visual Studio

To set up and start a verification:

- 1 In the Visual Studio **Solution Explorer** view, select one or more files that you want to verify.
- 2 Right-click the selection, and select **Polyspace Verification**.

The Easy Settings dialog box opens.



- 3** In the Easy Settings dialog box, you can specify the following options for your verification:
- Under **Settings**, configure the following:
 - **Precision** — Precision of verification (-0)
 - **Passes** — Level of verification (-to)
 - **Results folder** – Location where software stores verification results (-results-dir)
 - Under **Verification Mode Settings**, configure the following:
 - **Generate main** or **Use existing** — Whether Polyspace generates a main subprogram (-main-generator) or uses an existing subprogram (-main)
 - **Class** — Name of class to verify (-class-analyzer)
 - **Class analyzer calls** — Functions called by generated main subprogram (-class-analyzer-calls)
 - **Class only** — Verification of class contents only (-class-only)
 - **Main generator write** — Type of initialization for global variables (-main-generator-writes-variables)
 - **Main generator calls** — Functions (not in a class) called by generated main subprogram (-main-generator-calls)
 - **Function called before** — Function called before all functions (-function-call-before-main)
 - Under **Scope**, you can modify the list of files and classes to verify.

For information on *how* to choose your options, see “Analysis Options for C++ Code”.

Note In the Project Manager perspective of the Polyspace verification environment, you configure options that you cannot set in the Easy Settings dialog box. See “Set Standard Polyspace Options” on page 20-11.

- 4** Click **Start** to start the verification.

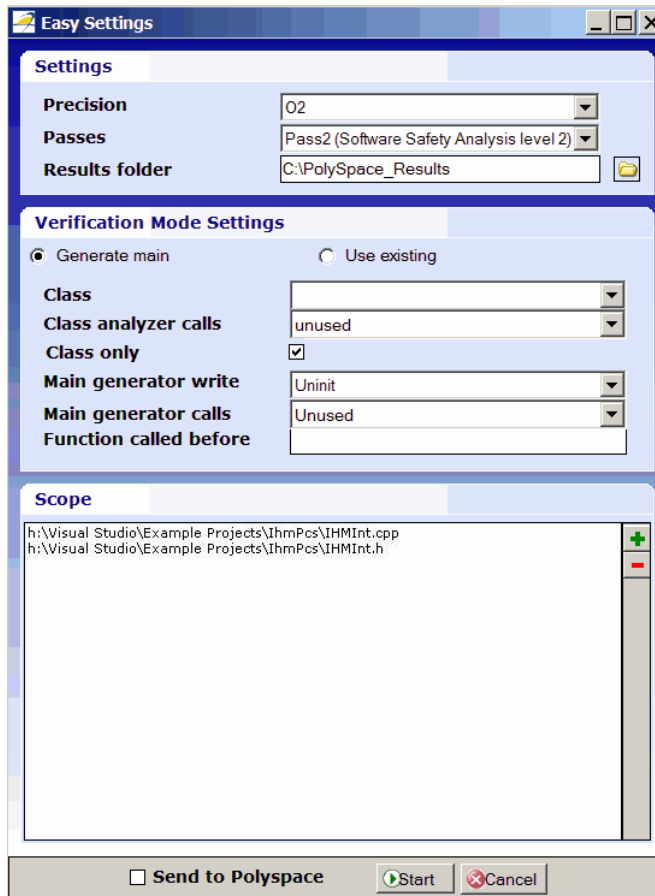
Verify Classes


In the Easy Settings dialog box, you can verify a C++ class by modifying the scope option.

To verify a class:

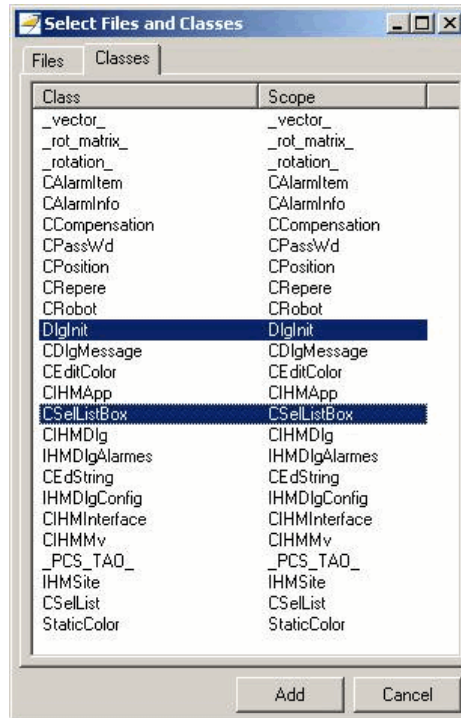
- 1 In the Visual Studio Solution Explorer, right-click a file and select **Polyspace Verification**.

The Easy Settings dialog box opens.



- 2 In the Scope window, click .

The Select Files and Classes dialog box opens.



- 3 Select the classes that you want to verify, then click **Add**.
- 4 In the Easy Settings dialog box, click **Start** to start the verification.

Verify an Entire Project

You can verify an entire project only through the Project Manager perspective of the Polyspace verification environment (select **Polyspace > Configure Project**).

For information on using the Project Manager perspective, see “Project Manager Verification”.

Import Visual Studio Project Information into Polyspace Project

You can extract information from a Visual Studio project file (vcproj) to configure your Polyspace project.

This Visual Studio import feature can retrieve the following information from a Visual Studio project:

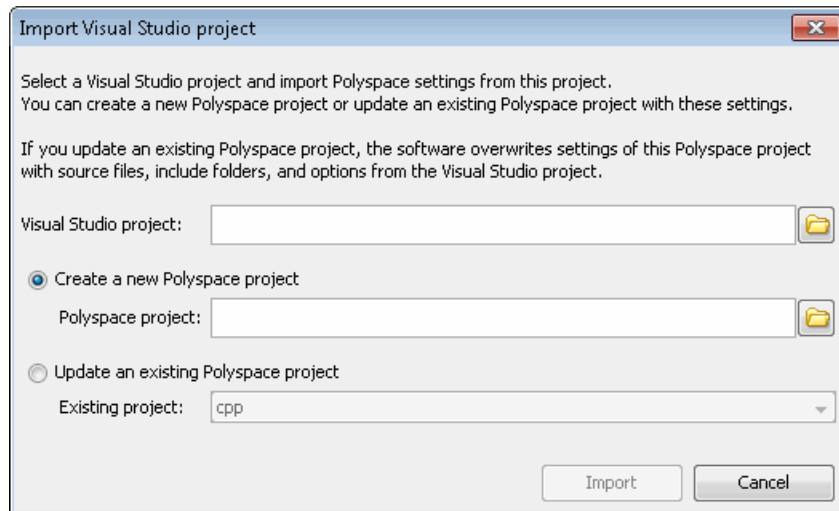
- Source files
- Include folders
- Preprocessing directives (-D, -U)
- Polyspace specific options about dialect used

Note This feature supports Visual Studio versions 2003, 2005, and 2008.

To import Visual Studio information into your Polyspace project:

- 1** In the Polyspace Project Manager, select **File > Import Visual Studio Project**.

The Import Visual Studio project dialog box opens.



- 2** Select the Visual Studio project you want to use.
- 3** Select the Polyspace project you want to use.
- 4** Click **Import**.

The Polyspace project is updated with the Visual Studio settings.

When you import a Visual Studio project, if all the source files are C files (with file extension `.c`), then the project will be a C project. Otherwise, the project will be a C++ project.

Set Standard Polyspace Options

In the Project Manager perspective of the Polyspace verification environment, you specify Polyspace verification options that you cannot set in the Easy Settings dialog box.

To open the Project Manager perspective, select **Polyspace > Configure Project**. The software opens the Project Manager perspective using the **last** configuration (`.psprj`) file updated in Visual Studio. The software does not check the consistency of this configuration file with the current project, so it always displays a warning message. This message indicates that the `.psprj`

file used by the Project Manager does not correspond to the .psprj file of the current project.

For information on *how* to choose your options, see “Analysis Options for C++ Code”.

Configuration File and Default Options

Some options are set by default while others are extracted from the Visual Studio project and stored in the associated Polyspace configuration file.

- The following table shows Visual Studio options that are extracted automatically, and their corresponding Polyspace options:

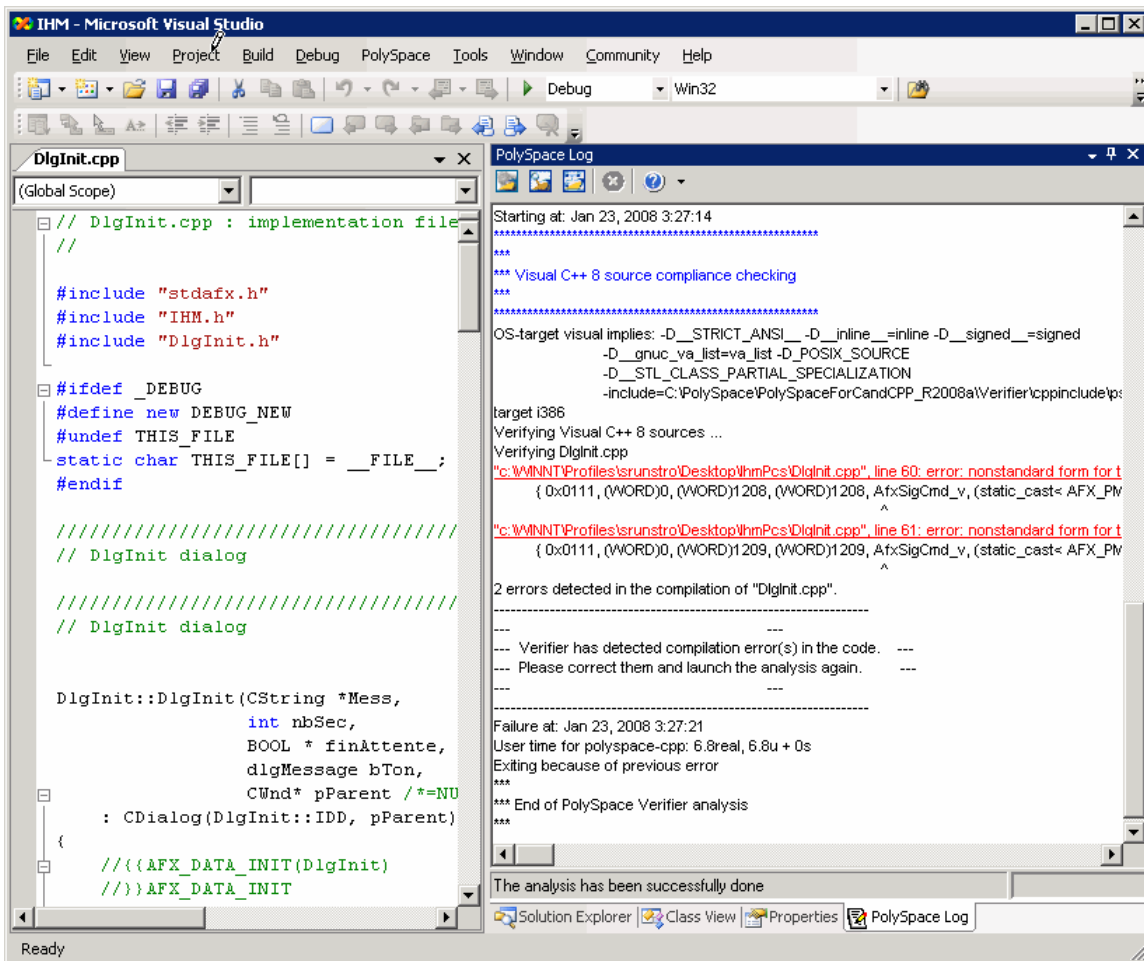
Visual Studio Option	Polyspace Option
/D <name>	-D <name>
/U <name>	-U <name>
/MT	-D_MT
/MTd	-D_MT -D_DEBUG
/MD	-D_MT -D_DLL
/MDd	-D_MT -D_DLL -D_DEBUG
/MLd	-D_DEBUG
/Zc:wchar_t	-wchar-t-is keyword
/Zc:forScope	-for-loop-index-scope in
/FX	-support-FX-option-results
/Zp[1,2,4,8,16]	-pack-alignment-value [1,2,4,8,16]

- Source and include folders (-I) are also extracted automatically from the Visual Studio project.
- Default options passed to the kernel depend on the Visual Studio release:
 - dialect Visual7.1 (or -dialect visual8) -OS-target Visual
 - target i386 -desktop

Monitor Verification in Visual Studio

Once you start a verification, you can follow its progress in the **Polyspace Log** view.

Compilation errors are highlighted as links. Click a link to display the file and line that produced the error.



If the verification is being carried out on a server, use the Polyspace Spooler to follow the verification progress. Select **Polyspace > Spooler**, which opens the Polyspace Queue Manager Interface dialog box.

To stop a verification, on the **Polyspace Log** toolbar, click **X**. For a server verification, this option works only during the compilation phase, before the verification is sent to the server. After the compilation phase, you can select **Polyspace > Spooler** and in the Polyspace Queue Manager Interface dialog box, stop the verification.

For more information on the Polyspace Spooler, see “Manage Previous Verifications With Polyspace Metrics” on page 8-12.

Review Verification Results in Visual Studio

Select **Polyspace > Open Verification Results** to open the Results Manager perspective of the Polyspace verification environment with the last available results. If verification has been carried out on a server, download the results before opening the Results Manager perspective.

For information on reviewing and understanding Polyspace verification results, see “Run-Time Error Review”.

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.

Selectivity

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.