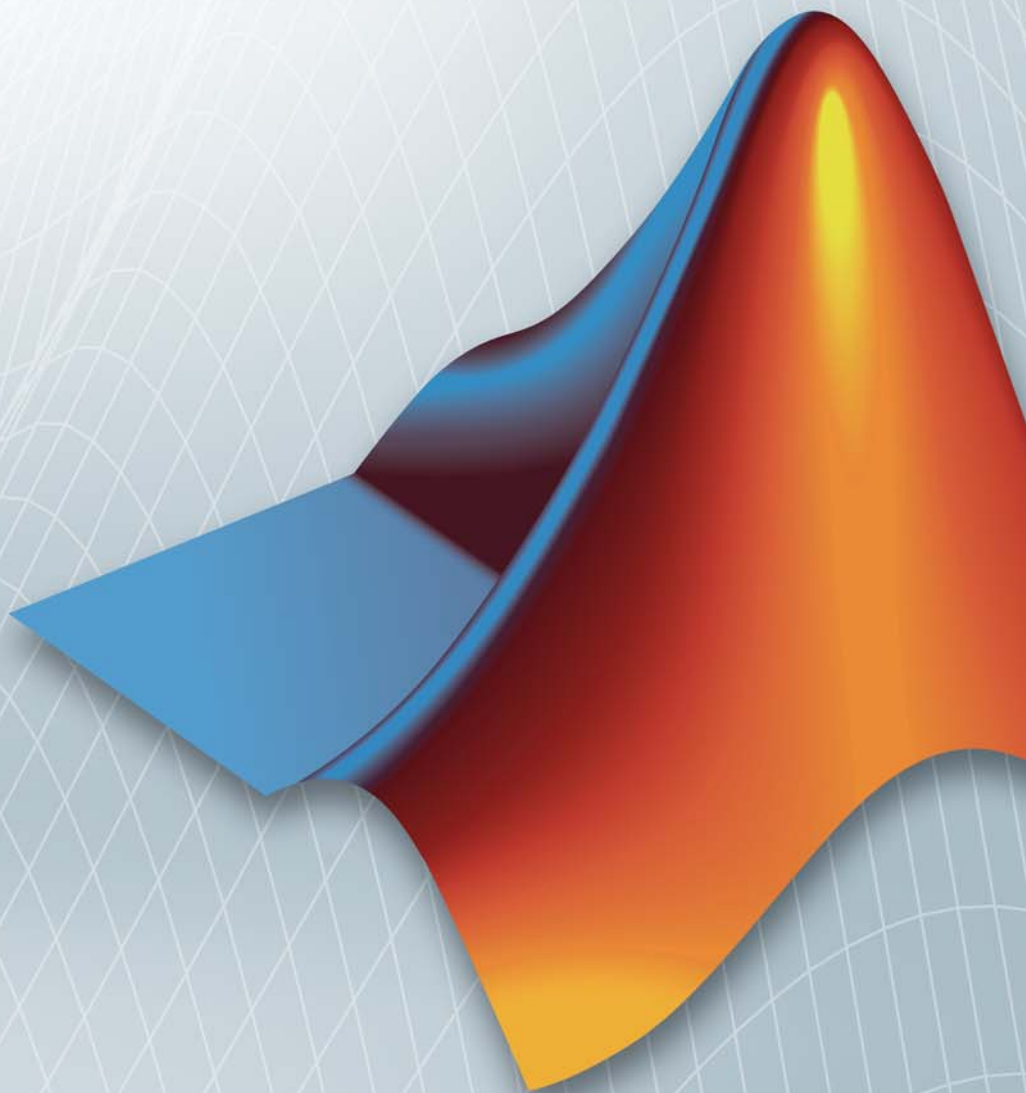


Polyspace® Products for C/C++ User's Guide

R2011b



How to Contact MathWorks



www.mathworks.com Web
comp.soft-sys.matlab Newsgroup
www.mathworks.com/contact_TS.html Technical Support



suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com 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® Products for C/C++ User's Guide

© COPYRIGHT 1999–2011 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

March 2008	Online Only	Revised for Version 5.1 (Release 2008a)
October 2008	Online Only	Revised for Version 6.0 (Release 2008b)
March 2009	Online Only	Revised for Version 7.0 (Release 2009a)
September 2009	Online Only	Revised for Version 7.1 (Release 2009b)
March 2010	Online Only	Revised for Version 7.2 (Release 2010a)
September 2010	Online Only	Revised for Version 8.0 (Release 2010b)
April 2011	Online Only	Revised for Version 8.1 (Release 2011a)
September 2011	Online Only	Revised for Version 8.2 (Release 2011b)

Introduction to Polyspace Products

1

Product Overview	1-2
Polyspace Products for C/C++	1-2
Overview of Polyspace Verification	1-2
The Value of Polyspace Verification	1-3
How Polyspace Verification Works	1-6
What is Static Verification	1-6
Exhaustiveness	1-7
Product Components	1-8
Polyspace Verification Environment	1-8
Other Polyspace Components	1-12
Installing Polyspace Products	1-15
Related Products	1-16
Polyspace Products for Verifying Ada Code	1-16
Polyspace Products for Linking to Models	1-16
Additional Information and Support	1-17
Related Documentation	1-17
MathWorks Online	1-17

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

Implementing a Process for Polyspace Verification . . .	2-4
Overview of the Polyspace Process	2-4
Defining Quality Objectives	2-5
Defining a Verification Process to Meet Your Objectives . .	2-11
Applying Your Verification Process to Assess Code Quality	2-12
Improving 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 a Verification Project

3

Creating a Project	3-2
What Is a Project?	3-2
Project Folders	3-3
Opening Polyspace Verification Environment	3-3
Creating New Projects	3-5
Opening Existing Projects	3-8
Closing Existing Projects	3-9
Specifying Source Files	3-10
Specifying Include Folders	3-12
Managing Include File Sequence	3-14
Creating Multiple Modules	3-15
Creating Multiple Analysis Option Configurations	3-16
Specifying Functions Not Called by Generated Main	3-18
Changing Project Location	3-20
Specifying Target Environment	3-21
Specifying Analysis Options	3-22
Configuring Text and XML Editors	3-23

Saving the Project	3-25
Specifying Options to Match Your Quality	
Objectives	3-26
Quality Objectives Overview	3-26
Choosing Contextual Verification Options for C Code	3-26
Choosing Contextual Verification Options for C++ Code ..	3-29
Choosing Strict or Permissive Verification Options	3-31
Choosing Coding Rules	3-33
Setting Up Project to Check Coding Rules	3-35
Polyspace Coding Rules Checker Overview	3-35
Checking Compliance with MISRA C Coding Rules	3-35
Checking Compliance with C++ Coding Rules	3-37
Setting up Project to Automatically Test Orange Code (C Only)	3-38
Polyspace Automatic Orange Tester	3-38
Enabling the Automatic Orange Tester	3-38
Setting Up Project to Generate Metrics	3-40
About Polyspace Metrics	3-40
Enabling Polyspace Metrics	3-40
Specifying Automatic Verification	3-41
Configuring Polyspace Project Using Visual Studio	
Project Information	3-42

Emulating Your Runtime Environment

4

Setting Up a Target	4-2
Target/Compiler Overview	4-2
Specifying Target Environment	4-3
Predefined Target Processor Specifications	4-4
Modifying Predefined Target Processor Attributes	4-7
Defining Generic Target Processors	4-9
Common Generic Targets	4-10

Viewing Existing Generic Targets	4-11
Deleting a Generic Target	4-12
Compiling Operating System Dependent Code (OS-target issues)	4-13
Address Alignment	4-19
Ignoring or Replacing Keywords Before Compilation	4-20
Verifying Code That Uses Keil or IAR Dialects	4-23
How to Gather Compilation Options Efficiently	4-30
Verifying a C Application Without a “Main”	4-33
Main Generator Overview	4-33
Automatically Generating a Main	4-34
Manually Generating a Main	4-35
Specifying Call Sequence	4-36
Specifying Functions Not Called by Generated Main	4-37
Main Generator Assumptions	4-39
Polyspace C++ Class Analyzer	4-40
Why Provide a Class Analyzer	4-40
How the Class Analyzer Works	4-41
Sources Verified	4-41
Architecture of the Generated main	4-41
Class Verification Log File	4-42
Characteristics of a Class and Messages in the Log File ..	4-43
Behavior of Global variables and members	4-44
Methods and Class Specificities	4-46
Simple Class	4-48
Simple Inheritance	4-50
Multiple Inheritance	4-52
Abstract Classes	4-53
Virtual Inheritance	4-54
Other Types of Classes	4-55
Specifying Data Ranges for Variables and Functions	
(Contextual Verification)	4-56
Overview of Data Range Specifications (DRS)	4-56
Specifying Data Ranges Using DRS Template	4-57
DRS Configuration Settings	4-60
Specifying Data Ranges Using Existing DRS	
Configuration	4-64
Editing Existing DRS Configuration	4-65
XML Format of DRS File	4-66
Specifying Data Ranges Using Text Files	4-73

Variable Scope	4-76
Performing Efficient Module Testing with DRS	4-80
Reducing Oranges with DRS	4-81

Preparing Source Code for Verification

5

Stubbing	5-2
Stubbing Overview	5-2
Manual vs. Automatic Stubbing	5-2
Stubbing Examples	5-6
Automatic Stubbing Behavior for C++ Pointer/Reference	5-9
Specifying Functions to Stub Automatically	5-10
Constraining Data with Stubbing	5-13
Default and Alternative Behavior for Stubbing (PURE and WORST)	5-18
Function Pointer Cases	5-21
Stubbing Functions with a Variable Argument Number ..	5-21
Stubbing Standard Library Functions	5-23
Preparing Code for Variables	5-24
Checking Variable Ranges with Assert	5-24
Checking Properties on Global Variables: Global Assert ..	5-25
Modeling Variable Values External to Your Application ..	5-25
Initializing Variables	5-26
Data and Coding Rules	5-27
Verifying Code with Undefined or Undeclared Variables and Functions	5-28
Preparing Code for Built-In Functions	5-30
Overview	5-30
Stubs of <code>stl</code> Functions	5-30
Stubs of <code>libc</code> Functions	5-30
Preparing Multitasking Code	5-32
Polyspace Software Assumptions	5-32
Modelling Synchronous Tasks	5-33

Modelling Interruptions and Asynchronous Events, Tasks, and Threads	5-35
Are Interruptions Maskable or Preemptive by Default? ...	5-37
Shared Variables	5-39
Mailboxes	5-42
Atomicity (Can an Instruction Be Interrupted by Another?)	5-44
Priorities	5-46
Highlighting Known Coding Rule Violations and Run-Time Errors	5-47
Annotating Code to Indicate Known Coding Rule Violations	5-47
Annotating Code to Indicate Known Run-Time Errors ...	5-51
Types Promotion	5-55
Unsigned Integers Promoted to Signed Integers	5-55
Promotions Rules in Operators	5-56
Example	5-56
Verifying “Unsupported” Code	5-58
Ignoring Assembly Code	5-58
Dealing with Backward “goto” Statements	5-66

Running a Verification

6

Before Running Verification	6-2
Types of Verification	6-2
Specifying Source Files to Verify	6-2
Specifying Results Folder	6-3
Specifying Analysis Options Configuration	6-4
Checking for Compilation Problems	6-5
Running Verifications on Polyspace Server	6-9
Starting Server Verification	6-9
What Happens When You Run Verification	6-10
Running Verification Unit-by-Unit	6-11
Verify All Modules in Project	6-13

Managing Verification Jobs Using the Polyspace Queue Manager	6-14
Monitoring Progress of Server Verification	6-16
Viewing Verification Log File on Server	6-21
Stopping Server Verification Before It Completes	6-22
Removing Verification Jobs from Server Before They Run	6-23
Changing Order of Verification Jobs in Server Queue	6-24
Purging Server Queue	6-25
Changing Queue Manager Password	6-27
Sharing Server Verifications Between Users	6-28
Running Verifications on Polyspace Client	6-31
Specifying Source Files to Verify	6-31
Starting Verification on Client	6-32
What Happens When You Run Verification	6-33
Monitoring the Progress of the Verification	6-34
Stopping the Verification Before It is Complete	6-35
Running Verifications from Command Line	6-37
Launching Verifications in Batch	6-37
Managing Verifications in Batch	6-37

Troubleshooting Verification Problems

7

Verification Process Failed Errors	7-2
Reasons Verification Can Fail	7-2
Viewing Error Information When Verification Stops	7-2
Hardware Does Not Meet Requirements	7-3
You Did Not Specify the Location of Included Files	7-4
Polyspace Software Cannot Find the Server	7-4
Limit on Assignments and Function Calls	7-7
Compilation Errors	7-9
Compilation Error Overview	7-9
Checking Compilation Before Running Verification	7-10
Examining Compile Log	7-10
Compilation Messages Described in This Section	7-12

Syntax Error	7-13
Undeclared Identifier	7-14
No Such File or Folder	7-15
#error directive	7-16
Class, Array, Struct or Union is Too Large	7-16
Unsupported Non-ANSI Keywords (C)	7-17
Initialization of Global Variables (C++)	7-19
C++ Dialect Issues	7-21
ISO versus Default Dialects	7-21
CFront2 and CFront3 Dialects	7-23
Visual Dialects	7-24
GNU Dialect	7-26
C Link Errors	7-30
Link Error Overview (C)	7-30
Function: Wrong Argument Type	7-31
Function: Wrong Argument Number	7-31
Variable: Wrong Type	7-32
Variable: Signed/Unsigned	7-32
Variable: Different Qualifier	7-33
Variable: Array Against Variable	7-33
Variable: Wrong Array Size	7-34
Missing Required Prototype for varargs	7-34
C++ Link Errors	7-36
STL Library C++ Stubbing Errors	7-36
Lib C Stubbing Errors	7-37
Standard Library Function Stubbing Errors	7-40
Conflicts Between Standard Library Functions and Polyspace Stubs	7-40
_polyspace_stdstubs.c Compilation Errors	7-40
Troubleshooting Approaches for Standard Library Function Stubs	7-42
Restart with the -I option	7-42
Include Files with Stubs to Replace Automatic Stubbing ..	7-43
Create a _polyspace_stdstubs.c File with Necessary Includes	7-44
Provide a .c file Containing a Prototype Function	7-45
Ignore _polyspace_stdstubs.c	7-46

Automatic Stubbing Errors	7-47
Three Types of Error Messages	7-47
Function Pointer Error	7-47
Unknown Prototype Error	7-49
Parameter -entry-points Error	7-49
Troubleshooting Using Preprocessed Files	7-50
Overview of Preprocessed (.ci) Files	7-50
Example .ci File	7-50
Troubleshooting Methodology	7-52
Reducing Verification Time	7-55
Factors Impacting Verification Time	7-55
Displaying Verification Status Information	7-56
Techniques for Improving Verification Performance	7-57
Turning Antivirus Software Off	7-59
Tuning Polyspace Parameters	7-59
Subdividing Code	7-60
Reducing Procedure Complexity	7-70
Reducing Task Complexity	7-72
Reducing Variable Complexity	7-72
Choosing Lower Precision	7-73
Obtaining Configuration Information	7-74
Removing Preliminary Results Files	7-77

Reviewing Verification Results

8

Before You Review Polyspace Results	8-2
Overview: Understanding Polyspace Results	8-2
Why Gray Follows Red and Green Follows Orange	8-3
The Message and What It Means	8-4
The Code Explanation	8-5
Opening Verification Results	8-8
Downloading Results from Server to Client	8-8

Downloading Server Results Using Command Line	8-10
Downloading Results from Unit-by-Unit Verifications	8-11
Opening Verification Results from Project Manager Perspective	8-12
Opening Verification Results from Run-Time Checks Perspective	8-13
Exploring the Run-Time Checks Perspective	8-14
Selecting Review Level	8-28
Searching Results in Run-Time Checks Perspective	8-29
Setting Character Encoding Preferences	8-30
Opening Results for Generated Code	8-32
Reviewing Results Systematically	8-34
What are Review Levels?	8-34
Reviewing Checks at Level 0	8-35
Reviewing Checks at Levels 1, 2, and 3	8-36
Reviewing Checks Progressively	8-41
Saving Review Comments	8-43
Reviewing All Checks	8-44
Selecting a Check to Review	8-44
Displaying the Call Sequence for a Check	8-48
Displaying the Access Graph for Variables	8-49
Filtering Checks	8-50
Saving Review Comments	8-53
Tracking Review Progress	8-55
Checking Coding Review Progress	8-55
Reviewing and Commenting Checks	8-56
Defining Custom Status	8-58
Tracking Justified Checks in Procedural Entities View	8-60
Commenting Code to Justify Known Checks	8-61
Importing and Exporting Review Comments	8-64
Reusing Review Comments	8-64
Importing Review Comments from Previous Verifications	8-65
Exporting Review Comments to Spreadsheet	8-66
Viewing Checks and Comments Report	8-66
Generating Reports of Verification Results	8-68
Polyspace Report Generator Overview	8-68

Generating Verification Reports	8-70
Running the Report Generator from the Command Line ..	8-72
Automatically Generating Verification Reports	8-73
Customizing Verification Reports	8-73
Generating Excel Reports	8-74
Using Polyspace Results	8-80
Review Runtime Errors: Fix Red Errors	8-80
Red Checks Where Gray Checks were Expected	8-81
Using Range Information in Run-Time Checks	
Perspective	8-83
Using Pointer Information in Run-Time Checks	
Perspective	8-88
Why Review Dead Code Checks	8-92
Reviewing Orange Checks	8-94
Integration Bug Tracking	8-94
How to Find Bugs in Unprotected Shared Data	8-95
Dataflow Verification	8-96
Data and Coding Rules	8-96
Potential Side Effect of a Red Error	8-97
Relationships Between Variables	8-98

Managing Orange Checks

9

Understanding Orange Checks	9-2
What is an Orange Check?	9-2
Sources of Orange Checks	9-6
Too Many Orange Checks?	9-12
Do I Have Too Many Orange Checks?	9-12
How to Manage Orange Checks	9-13
Reducing Orange Checks in Your Results	9-14
Overview: Reducing Orange Checks	9-14
Applying Coding Rules to Reduce Orange Checks	9-15
Considering Generated Code	9-20
Improving Verification Precision	9-21
Stubbing Parts of the Code Manually	9-26

Describing Multitasking Behavior Properly	9-28
Considering Contextual Verification	9-29
Considering the Effects of Application Code Size	9-30
Reviewing Orange Checks	9-31
Overview: Reviewing Orange Checks	9-31
Defining Your Review Methodology	9-31
Performing Selective Orange Review	9-32
Importing Review Comments from Previous Verifications	9-37
Commenting Code to Provide Information During Review	9-38
Viewing Sources of Orange Checks	9-39
Working with Orange Checks Caused by Input Data	9-40
Refining Data Range Specifications	9-44
Performing an Exhaustive Orange Review	9-47
Automatically Testing Orange Code	9-52
Automatic Orange Tester Overview	9-52
How the Automatic Orange Tester Works	9-53
Selecting the Automatic Orange Tester	9-54
Starting the Automatic Orange Tester Manually	9-56
Reviewing Test Results After Manual Run	9-60
Refining Data Ranges with Automatic Orange Tester	9-64
Saving and Reusing Your Configuration	9-68
Exporting Data Ranges for Polyspace Verification	9-68
Configuring Compiler Options	9-69
Technical Limitations	9-70

Day to Day Use

10

Polyspace In One Click Overview	10-2
Using Polyspace In One Click	10-3
Polyspace In One Click Workflow	10-3
Setting the Active Project	10-3
Launching Verification	10-5
Using the Taskbar Icon	10-7

Overview of Polyspace Code Analysis	11-2
Code Analysis Overview	11-2
Polyspace MISRA C Checker Overview	11-2
Polyspace MISRA C++ Checker Overview	11-3
Polyspace JSF C++ Checker Overview	11-4
Setting Up Coding Rules Checking	11-5
Activating the MISRA C Checker	11-5
Activating the MISRA C++ Checker	11-7
Activating the JSF C++ Checker	11-7
Creating a MISRA C Rules File	11-8
Creating a C++ Coding Rules File	11-10
Excluding Files from Rules Checking	11-12
Excluding All Include Folders from Coding Rules Checking	11-13
Redefine Data Types as Boolean	11-14
Configuring Text and XML Editors	11-14
Commenting Code to Indicate Known Rule Violations	11-16
Viewing Coding Rules Checker Results	11-18
Running a Verification with Coding Rules Checking	11-18
Examining Rule Violations	11-19
Commenting and Justifying Coding Rule Violations	11-22
Opening Source Files from Coding Rules Perspective	11-24
Opening Coding Rules Report	11-25
Generating Coding Rules Report	11-26
Copying and Pasting Justifications	11-27
Coding Rules Assistant	11-28
Polyspace Metrics and Coding Rules Assistant	11-28
Reviewing Assistant Coding Rules	11-28
Software Quality Objective Subsets of Coding Rules	
(C)	11-33
SQO Subset 1 – Coding Rules with a Direct Impact on Selectivity	11-33
SQO Subset 2 – Coding Rules with an Indirect Impact on Selectivity	11-35

Supported Coding Rules	11-38
MISRA C Rules Supported	11-38
MISRA C Rules Not Checked	11-75
Supported MISRA C++ Coding Rules	11-79
MISRA C++ Rules Not Checked	11-99
Supported JSF C++ Coding Rules	11-105
JSF++ Rules Not Checked	11-130

Software Quality with Polyspace Metrics

12

About Polyspace Metrics	12-2
Setting Up Verification to Generate Metrics	12-4
Specifying Automatic Verification	12-4
Accessing Polyspace Metrics	12-12
Monitoring Verification Progress	12-13
Web Browser Support	12-14
What You Can Do with Polyspace Metrics	12-15
Review Overall Progress	12-15
Displaying Metrics for Single Project Version	12-19
Creating a File Module and Specifying Quality Level	12-20
Compare Project Versions	12-21
Review New Findings	12-21
Review Coding Rule Violations and Run-Time Checks ...	12-22
Fix Defects	12-27
Review Code Complexity	12-29
Customizing Software Quality Objectives	12-30
About Customizing Software Quality Objectives	12-30
SQO Level 1	12-32
SQO Level 2	12-35
SQO Level 3	12-35
SQO Level 4	12-36
SQO Level 5	12-36
SQO Level 6	12-36
SQO Exhaustive	12-37

Coding Rules Set 1	12-37
Coding Rules Set 2	12-39
Run-Time Checks Set 1	12-41
Run-Time Checks Set 2	12-42
Run-Time Checks Set 3	12-43
Status Acronyms	12-43

Tips for Administering Results Repository	12-45
Through the Polyspace Metrics Web Interface	12-45
Through Command Line	12-46
Backup of Results Repository	12-48

Using Polyspace Software in the Eclipse IDE

13

Verifying Code in the Eclipse IDE	13-2
Creating an Eclipse Project	13-3
Setting Up Polyspace Verification with Eclipse Editor	13-4
Launching Verification from Eclipse Editor	13-6
Reviewing Verification Results from Eclipse Editor	13-6
Using the Polyspace Spooler	13-7

Using Polyspace Software in Visual Studio

14

Verifying Code in Visual Studio	14-2
Creating a Visual Studio Project	14-4
Setting Up and Starting a Polyspace Verification in Visual Studio	14-5
Monitoring a Verification	14-13
Reviewing Verification Results in Visual Studio	14-15
Using the Polyspace Spooler	14-15

Glossary

Index

Introduction to Polyspace Products

- “Product Overview” on page 1-2
- “How Polyspace Verification Works” on page 1-6
- “Product Components” on page 1-8
- “Installing Polyspace Products” on page 1-15
- “Related Products” on page 1-16
- “Additional Information and Support” on page 1-17

Product Overview

In this section...
“Polyspace Products for C/C++” on page 1-2
“Overview of Polyspace Verification” on page 1-2
“The Value of Polyspace Verification” on page 1-3

Polyspace Products for C/C++

Polyspace Client for C/C++

Polyspace® Client™ for C/C++ provides code verification that proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in source code using static code analysis that does not require program execution, code instrumentation, or test cases. Polyspace Client for C/C++ uses formal methods-based abstract interpretation techniques to verify code. You can use it on handwritten code, generated code, or a combination of the two, before compilation and test.

Polyspace Server for C/C++

Polyspace® Server™ for C/C++ provides code verification that proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in source code. For faster performance, Polyspace Server for C/C++ lets you schedule verification tasks to run on a computer cluster. Jobs are submitted to the server using Polyspace Client for C/C++. You can integrate jobs into automated build processes and set up e-mail notifications. You can view defects, regressions, and code metrics via a Web browser. You then use the client to download and visualize verification results.

Overview of 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. Results are color-coded:

- **Green** – Indicates code that never has an error.
- **Red** – Indicates code that always has an error.
- **Gray** – Indicates unreachable code.
- **Orange** – Indicates unproven code (code that might have an error).

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.

The Value of Polyspace Verification

Polyspace verification can help you to:

- “Ensure Software Reliability” on page 1-3
- “Decrease Development Time” on page 1-4
- “Improve the Development Process” on page 1-4

Ensure Software Reliability

Polyspace software ensures 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 possible 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 is free of 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++ standards.¹

Decrease Development Time

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

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

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

Using Polyspace verification software helps you to use your time effectively. Because you know which parts of your code are error-free, you can focus on the code that has definite errors or might have errors.

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.

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

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 runtime errors. Static verification derives the dynamic properties of a program without actually executing it. This differs significantly from other techniques, such as runtime 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 which derives dynamic properties of a program without actually executing it. However, most Static Verification tools only verify the complexity of the software, in a search for constructs which may be potentially dangerous. Polyspace verification provides deep-level verification identifying almost all runtime errors and possible access conflicts on global shared data.

Polyspace verification works by approximating the software under verification, using safe and 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 needed.

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

Any approximation leads by definition 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 variation domain 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 but it is generally not practical, as it would in general require the enumeration of all possible test cases. As a result, approximation is required if a usable tool is to result.

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 any program variable is always a superset of its actual variation domain. The direct consequence is that no runtime error (RTE) item to be checked can be missed by Polyspace verification.

Product Components

In this section...
“Polyspace Verification Environment” on page 1-8
“Other Polyspace Components” on page 1-12

Polyspace Verification Environment

The Polyspace verification environment (PVE) is the graphical user interface of the Polyspace Client for C/C++ software. You use the Polyspace verification environment to create Polyspace projects, launch verifications, and review verification results.

The Polyspace verification environment consists of three perspectives:

- “Project Manager Perspective” on page 1-8
- “Coding Rules Perspective” on page 1-10
- “Run-Time Checks Perspective” on page 1-11

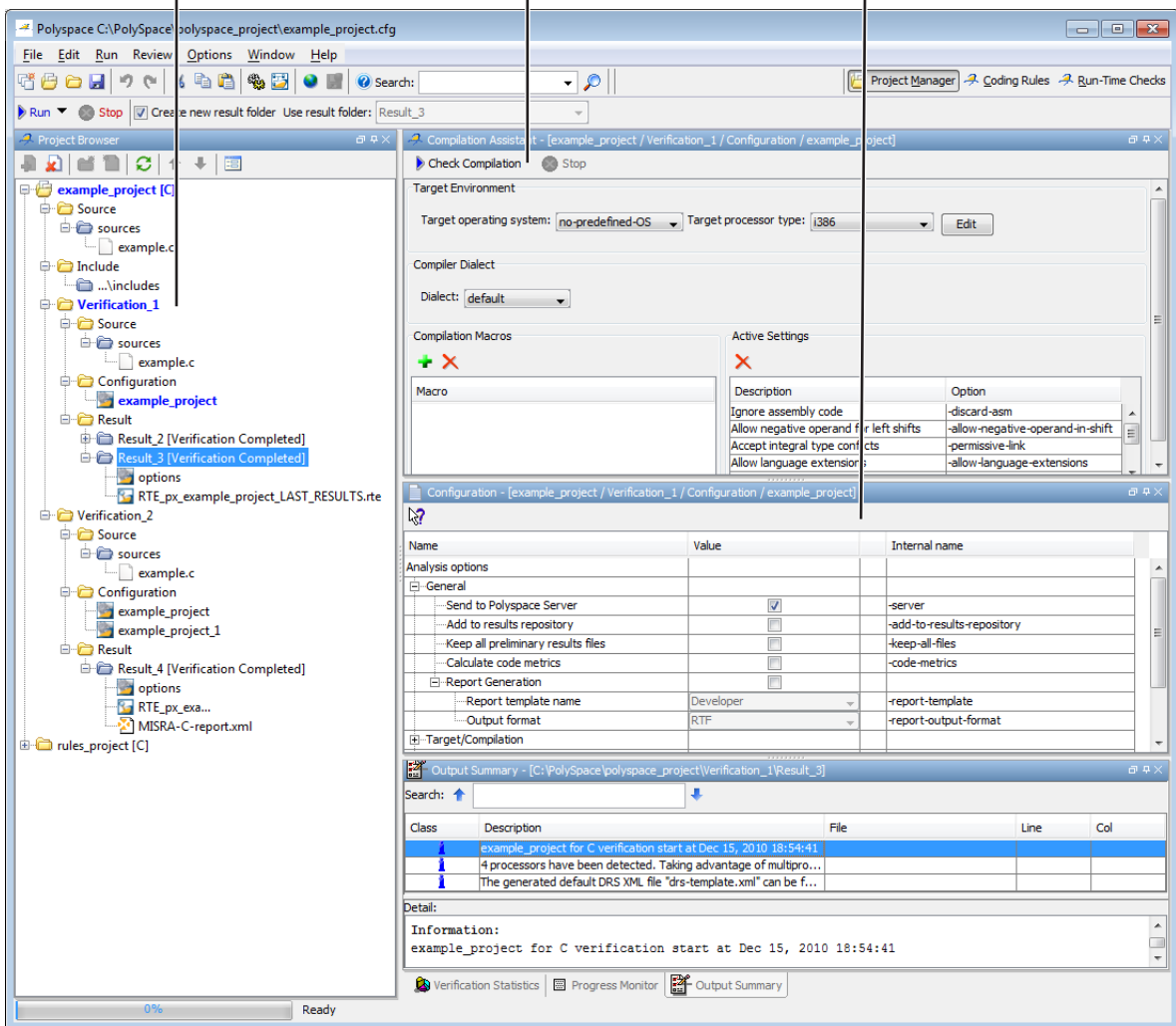
Project Manager Perspective

The Project Manager perspective allows you to create projects, set verification parameters, and launch verifications.

Specify source files and include folders

Set target environment and check compilation

Specify analysis options

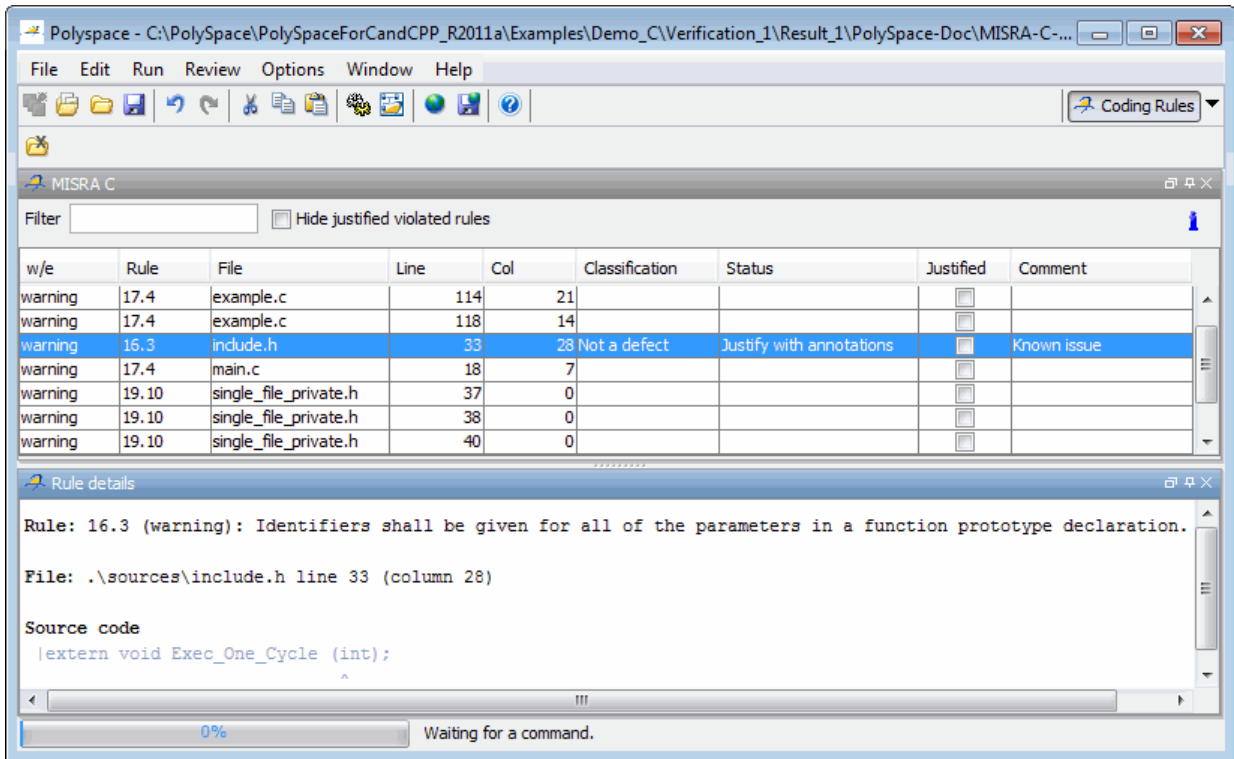


Monitor progress and view logs

For information on using the Project Manager perspective, see Chapter 3, “Setting Up a Verification Project”.

Coding Rules Perspective

The Coding Rules perspective allows you to review results from the Polyspace coding rules checker, to ensure compliance with established coding standards.



For information on using the Coding Rules perspective, see Chapter 11, “Checking Coding Rules”.

Run-Time Checks Perspective

The Run-Time Checks perspective allows you to review verification results, comment individual checks, and track review progress.

Review Details

Review Statistics

The screenshot displays the Polyspace Run-Time Checks perspective. The main window is titled "Polyspace - C:\PolySpace\polyspace_project\Verification\1\Result_2\RTE_px_example_project_LAST_RESULTS.rte". The interface includes a menu bar (File, Edit, Run, Review, Options, Window, Help), a toolbar, and a search bar. The "Run-Time Checks" panel on the left shows a tree view of procedural entities, with "Recursion_caller" selected. The "Review Details" panel in the center shows a table with columns for Classification, Status, Justified, and Comment. The "Review Statistics" panel on the right shows a table with columns for Coding review progress, Count, and Progress. The "Source" panel in the center shows the code for "example.c" with line numbers 150 to 170. The "Call Hierarchy" panel on the right shows a tree view of calls, with "example.Recursion_caller" selected. The "Variable Access" panel on the right shows a table with columns for Variables, # Read, # Write, V.T., R.T., Line, and File.

Coding review progress	Count	Progress
Red NTC justified / to justify	2/3	66
Red justified / to justify	2/5	40
Gray justified / to justify	0/9	0
Orange justified / to justify	0/7	0
Software reliability indicator	98/187	52

Variables	# Read	# Write	V.T.	R.T.	Line	File
example_project						
__polyspace__stdstubs.en	0	2			175	__polys...
__polyspace__stc					175	__polys...
__polyspace__stc					895	__polys...

Run-Time Checks

Source code

Variable Access

Call Hierarchy

For information on using the Run-Time Checks perspective, see Chapter 8, “Reviewing Verification Results”.

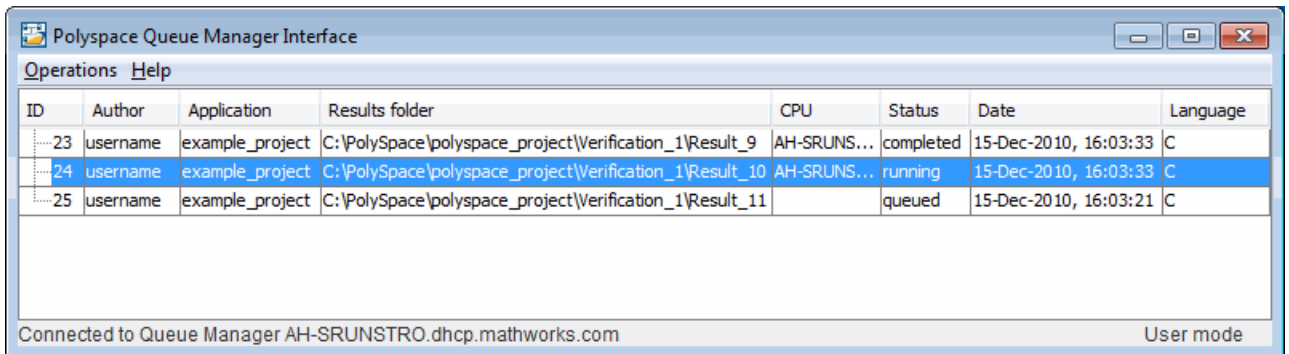
Other Polyspace Components

In addition to the Polyspace verification environment, Polyspace products provide several other components to manage verifications, improve productivity, and track software quality. These components include:

- Polyspace Queue Manager Interface (Spooler)
- Polyspace in One Click
- Polyspace Metrics Web Interface

Polyspace Queue Manager Interface (Polyspace Spooler)

The Polyspace Queue Manager (also called the Polyspace Spooler) is the graphical user interface of the Polyspace Server for C/C++ software. You use the Polyspace Queue Manager Interface to move jobs within the queue, remove jobs, monitor the progress of individual verifications, and download results.



For information on using the Polyspace Queue Manager Interface, see Chapter 6, “Running a Verification”.

Polyspace in One Click

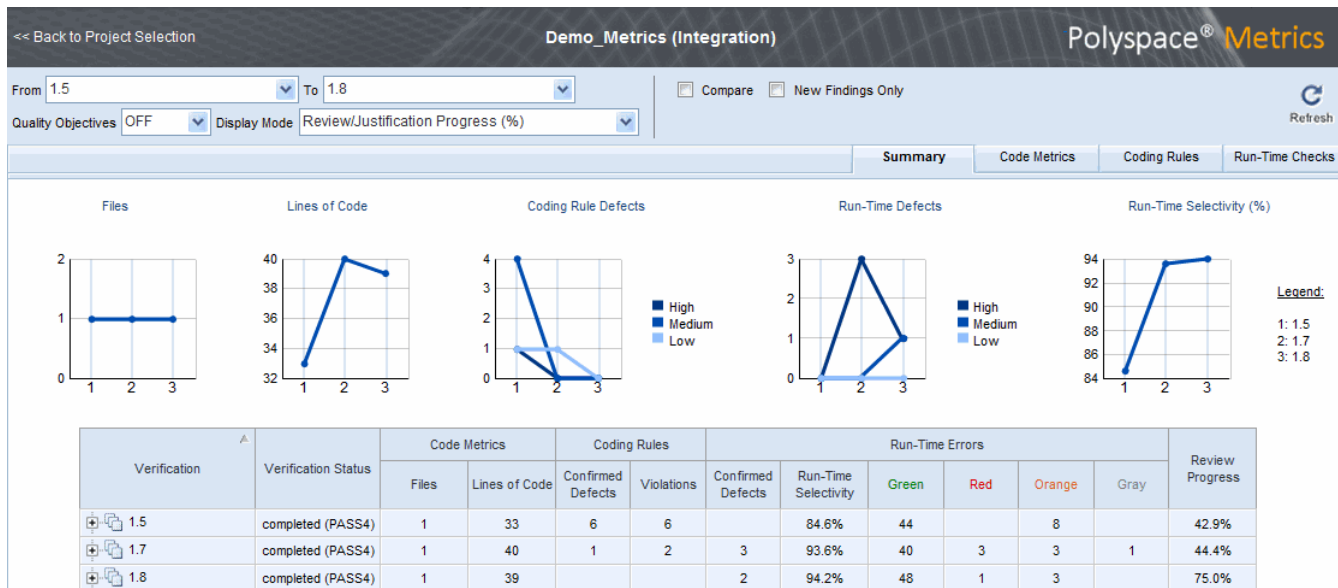
Polyspace in One Click is a convenient way to verify multiple files using the same set of options.

After creating a project with the options that you want, you can use Polyspace in One Click to designate that project as the *active project*, and then send source files to Polyspace software for verification with a single mouse click.

For information on using Polyspace in One Click, see Chapter 10, “Day to Day Use”.

Polyspace Metrics Web Interface

Polyspace Metrics is a web-based tool for software development managers, quality assurance engineers, and software developers. Polyspace Metrics allows you to evaluate software quality metrics, and monitor changes in code metrics, coding rule violations, and run-time checks through the lifecycle of a project.



For information on using Polyspace Metrics, see Chapter 12, “Software Quality with Polyspace Metrics”.

Installing Polyspace Products

For information on installing and licensing Polyspace products, refer to the *Polyspace Installation Guide*.

Related Products

In this section...
“Polyspace Products for Verifying Ada Code” on page 1-16
“Polyspace Products for Linking to Models” on page 1-16

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 Products for Linking to Models

For information about Polyspace products that link to models, see the following:

<http://www.mathworks.com/products/polyspacemodelsl/>

<http://www.mathworks.com/products/polyspaceumlrh/>

Additional Information and Support

In this section...
“Related Documentation” on page 1-17
“MathWorks Online” on page 1-17

Related Documentation

In addition to this guide, the following related documents are shipped with the software:

- ***Polyspace Products for C/C++ Getting Started Guide*** – Provides a basic workflow and step-by-step procedures for verifying C code using Polyspace software, to help you quickly learn how to use the software.
- ***Polyspace Products for C/C++ Reference*** – Provides detailed descriptions of all Polyspace options, as well as all checks reported in the Polyspace results.
- ***Polyspace Installation Guide*** – Describes how to install and license Polyspace products.
- ***Polyspace Release Notes*** – Describes new features, bug fixes, and upgrade issues.

You can access these guides from the **Help** menu, or by clicking the Help icon in the Polyspace window.

To access the online documentation for Polyspace products, go to:

www.mathworks.com/help/toolbox/polyspace/polyspace_product_page.html

MathWorks Online

For additional information and support, go to:

www.mathworks.com/products/polyspace

How to Use Polyspace Software

- “Polyspace Verification and the Software Development Cycle” on page 2-2
- “Implementing a Process for Polyspace Verification” on page 2-4
- “Sample Workflows for Polyspace Verification” on page 2-13

Polyspace Verification and the Software Development Cycle

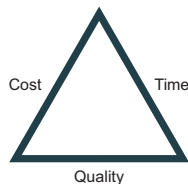
In this section...

“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 always three related variables: cost, quality, and time.



Changing the requirements for one of these variables always impacts 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 meets the required quality level. Unfortunately, this process often ends before quality objectives are met, because the available time or budget has been exhausted.

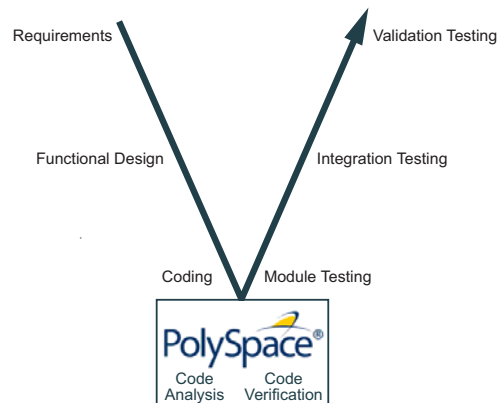
Polyspace verification allows a different process. Polyspace verification can support both productivity improvement and quality improvement at the same time, although there is always a balance between these goals.

To achieve maximum quality and productivity, however, you cannot simply perform code verification at the end of the development process. You must integrate verification into your development process, in a way that respects time and cost restrictions.

This chapter 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 any 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 code cannot be proven safe. In addition, defects are cheaper to fix at this stage, since they can be addressed before the code is integrated into a larger system.

Implementing a Process for Polyspace Verification

In this section...
“Overview of the Polyspace Process” on page 2-4
“Defining Quality Objectives” on page 2-5
“Defining a Verification Process to Meet Your Objectives” on page 2-11
“Applying Your Verification Process to Assess Code Quality” on page 2-12
“Improving 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. Verification is a tool that helps you measure the quality of your code, identify issues, and ultimately achieve your own quality goals. To do this, however, you must integrate Polyspace verification into your development process.

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

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

Defining Quality Objectives

Before you can verify whether your code meets your quality goals, you must define those goals. Therefore, the first step in implementing a verification process is to define your quality objectives.

This process involves:

- “Choosing Robustness or Contextual Verification” on page 2-5
- “Choosing Coding Rules” on page 2-6
- “Choosing Strict or Permissive Verification Objectives” on page 2-7
- “Defining Software Quality Levels” on page 2-8

Choosing 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 software works under all conditions.
- **Contextual Verification** – Prove software works 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 works under all 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

While this approach ensures that the software works under 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 objectives.

Contextual Verification. Contextual verification proves that the software works 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 “Specifying Data Ranges for Variables and Functions (Contextual Verification)” on page 4-56.
- Create a detailed main program to model the call sequence, instead of using the default main generator. For more information, see “Verifying a C Application Without a “Main”” on page 4-33.
- Provide manual stubs that emulate the behavior of missing functions, instead of using the default automatic stubs. For more information, see “Stubbing” on page 5-2.

Choosing 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 Chapter 11, “Checking Coding Rules”.

Choosing Strict or Permissive Verification Objectives

While defining the quality objectives 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 all global variables as non-initialized, therefore causing a red error if they are read before they are written to.
- **Give all warnings (-wall)** – Specifies that all C compliance warnings are written to the log file during compilation.
- **Strict (-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:

- **Enable pointer arithmetic out of bounds of fields (-allow-ptr-arith-on-struct)** – Enables navigation within a structure or union from one field to another.
- **Allow negative operand for left shifts (-allow-negative-operand-in-shift)** – Verification allows a shift operation on a negative number.
- **Ignore overflowing computations on constants (-ignore-constant-overflows)** – Verification is permissive with overflowing computations on constants.

- **Allow non int types for bitfields (-allow-non-int-bitfield)** – Allows you to define types of bitfields other than signed or unsigned int.
- **Allow undefined global variables (-allow-undef-variables)** – Verification does not stop due to errors caused by undefined global variables.
- **Allow anonymous union/structure fields (-allow-unnamed-fields)** – Verification does not stop due to errors caused by unnamed fields in structures.
- **Dialect support (-dialect)** – Verification allows syntax associated with the IAR and Keil dialects.

For more information on these options, see “Option Descriptions for C Code” in the *Polyspace Products for C/C++ Reference*.

Defining 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 appropriate 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

Software Quality Levels (Continued)

Criteria	Software Quality Levels			
	QL1	QL2	QL3	QL4
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

You define the quality criteria appropriate for your application. In the example above, the quality criteria include:

- **Static Information** – Includes information about the application architecture, the structure of each module, and all files. This information must be documented to ensure that your application is fully verified.
- **Coding rules** – Polyspace software can check that your code complies with specified coding rules. The section “Applying Coding Rules to Reduce Orange Checks” on page 9-15 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. Polyspace software allows you to define three criteria levels for reviewing orange checks in the Run-Time Checks perspective. For more information, see “Reviewing Results Systematically” on page 8-34.
- **Dataflow analysis** – Identifies errors such as non-initialized variables and variables that are written but never read. This can include inspection of:
 - Application call tree
 - Read/write accesses to global variables

- Shared variables and their associated concurrent access protection

Defining a Verification Process to Meet Your Objectives

Once you have defined your quality objectives, you must define a process that allows you to meet those objectives. 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 to match your quality objectives. For more information, see “Creating a Project” on page 3-2.
- Setting review criteria in the Run-Time Checks perspective to ensure results are reviewed consistently. For more information, see “Defining a Custom Methodology for Levels 1, 2, and 3” on page 8-39.

Applying Your Verification Process to Assess Code Quality

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

This process includes:

- 1 Launching Polyspace verification on each software component as it is written. See “Using Polyspace In One Click” on page 10-3.
- 2 Reviewing verification results consistently. See “Reviewing Results Systematically” on page 8-34.
- 3 Saving review comments for each component, so they are available for future review. See “Importing Review Comments from Previous Verifications” on page 9-37.
- 4 Performing additional verifications on each component, as defined by your quality objectives.

Improving 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 objectives.
- 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 Chapter 9, “Managing Orange Checks”.

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 objectives 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. The primary difference being how you exploit verification results.

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. Before implementing Polyspace verification, these users fit the following criteria:

- In Ada, no unit test tools or coverage tools are used – functional tests are performed just after coding.
- In C, either no coding rules are used, or rules are not followed consistently.

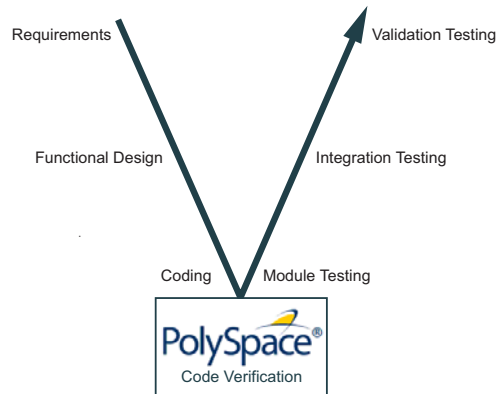
Quality Objectives

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 ensure 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 all 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 all **red** errors and examines **gray** code identified by the verification.
- 4 The developer repeats steps 2 and 3 as needed, while completing the code.
- 5 Once a developer considers a file complete, they perform a final verification.
- 6 The developer fixes any **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.

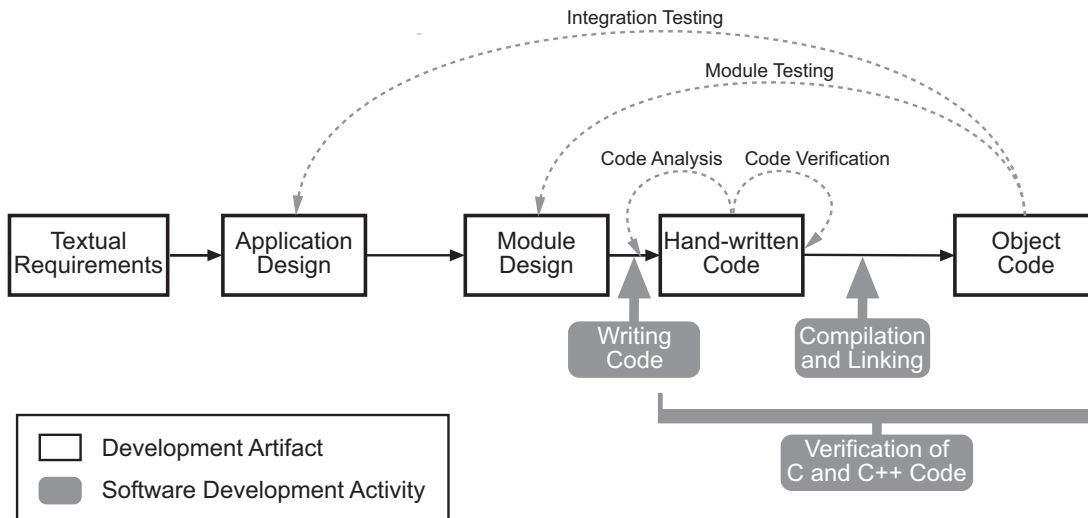
Quality Objectives

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

Verification can prove the absence of runtime errors, while helping developers and testers find and fix any bugs more quickly than other processes.

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 appropriate 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 any coding rule violations, fixes all **red** errors, examines **gray** code, and performs a selective orange review.
- 5 The developer repeats steps 2 and 3 as needed, while completing the code.
- 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 all orange checks that were not reviewed as part of previous reviews. This is possible when using coding rules because 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. This represents a step towards an exhaustive orange check review. However, spending more time is unlikely to be efficient, and will not guarantee that no bugs remain.
- 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 Objectives

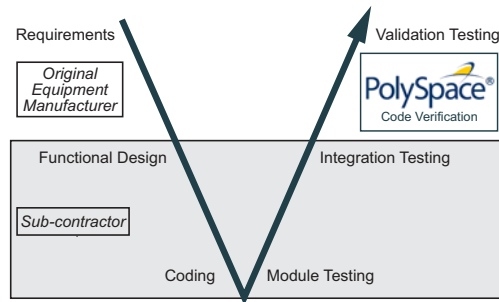
The main goal of Polyspace verification is to control and evaluate the safety of an application.

The criteria used to evaluate code can vary widely depending on the criticality of the application, from no red errors to exhaustive oranges 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 “Defining Software Quality Levels” on page 2-8.

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 objectives.



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 objectives 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 “Defining Quality Objectives” on page 2-5.
- 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 objectives, as described in “Defining a Verification Process to Meet Your Objectives” on page 2-11.
- 5** Quality engineers perform verification on the code.
- 6** Quality engineers review all **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 “Defining Software Quality Levels” on page 2-8).

- 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 all unresolved function calls.
- Using DRS to provide accurate data ranges for all input variables.

Taking these steps will minimize the number of orange checks reported by the verification, and make it likely that any remaining orange checks represent true 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-178B 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-178B 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 hand-written code with those created using generated code.

Quality Objectives

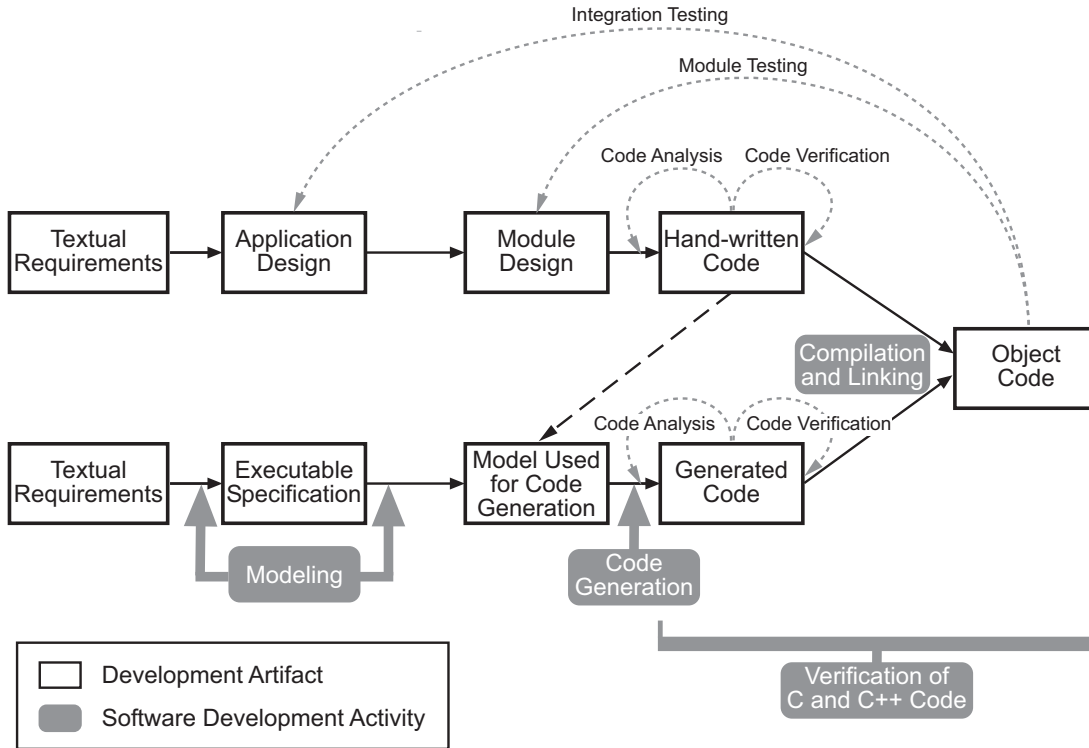
The goal of Polyspace verification is to improve the quality of the software by identifying implementation issues in the code, and ensuring the code is both semantically and logically correct.

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 hand-written and generated code

Verification Workflow

The workflow is different for hand-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 hand-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 objectives.
- 2** Developers write hand-coded sections of the application.
- 3** Developers or testers perform **Polyspace verification** on any hand-coded sections within the generated code, and review verification results according to the established quality objectives.
- 4** Developers create Simulink model based on requirements.
- 5** Developers validate model to ensure 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 hand-written and generated code.
- 8** Developers or testers review verification results according to the established quality objectives.

Note The Polyspace Model Link™ SL product allows you to quickly track any issues identified by the verification back to the appropriate 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 Objectives

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 objectives, including individual quality levels for each stage of the development cycle.
- 2** Project leader configures a Polyspace project to meet quality objectives.
- 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 all multitasking aspects accounted for as appropriate.
 - **Pre-peer review check-in** — On the whole project, with all multitasking aspects accounted for as appropriate.
- 4** Developers or testers review verification results for each check-in activity to ensure the code meets the appropriate quality level. For example, the transition criterion could be: “No bug found within 20 minutes of selective orange review”

Setting Up a Verification Project

- “Creating a Project” on page 3-2
- “Specifying Options to Match Your Quality Objectives” on page 3-26
- “Setting Up Project to Check Coding Rules” on page 3-35
- “Setting up Project to Automatically Test Orange Code (C Only)” on page 3-38
- “Setting Up Project to Generate Metrics” on page 3-40
- “Configuring Polyspace Project Using Visual Studio Project Information” on page 3-42

Creating a Project

In this section...

“What Is a Project?” on page 3-2

“Project Folders” on page 3-3

“Opening Polyspace Verification Environment” on page 3-3

“Creating New Projects” on page 3-5

“Opening Existing Projects” on page 3-8

“Closing Existing Projects” on page 3-9

“Specifying Source Files” on page 3-10

“Specifying Include Folders” on page 3-12

“Managing Include File Sequence” on page 3-14

“Creating Multiple Modules” on page 3-15

“Creating Multiple Analysis Option Configurations” on page 3-16

“Specifying Functions Not Called by Generated Main” on page 3-18

“Changing Project Location” on page 3-20

“Specifying Target Environment” on page 3-21

“Specifying Analysis Options” on page 3-22

“Configuring Text and XML Editors” on page 3-23

“Saving the Project” on page 3-25

What Is a Project?

In Polyspace software, a project is a named set of parameters for verification of 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 versions of source files used in the verification)
- Configuration (specific set of analysis options used for the verification)
- Verification results

You create and modify a project using the Project Manager perspective.

Project Folders

Before you begin verifying your code with Polyspace software, you must know the location of your source files and include files. You must also know where you want to store the verification results.

To simplify the location of your files, you may want to create a project folder, and then in that folder, create separate folders for the source files, include files, and results. For example:

```
polyspace_project/
```

- sources
- includes
- results

Opening Polyspace Verification Environment

You use the Polyspace verification environment to create projects, start verifications, and review verification results.

To open the Polyspace verification environment:

- 1 Double-click the **Polyspace** icon (Windows® systems).



On a Linux® or UNIX® system, use the following command:

```
$POLYSPACE/PVE/bin/polyspace
```

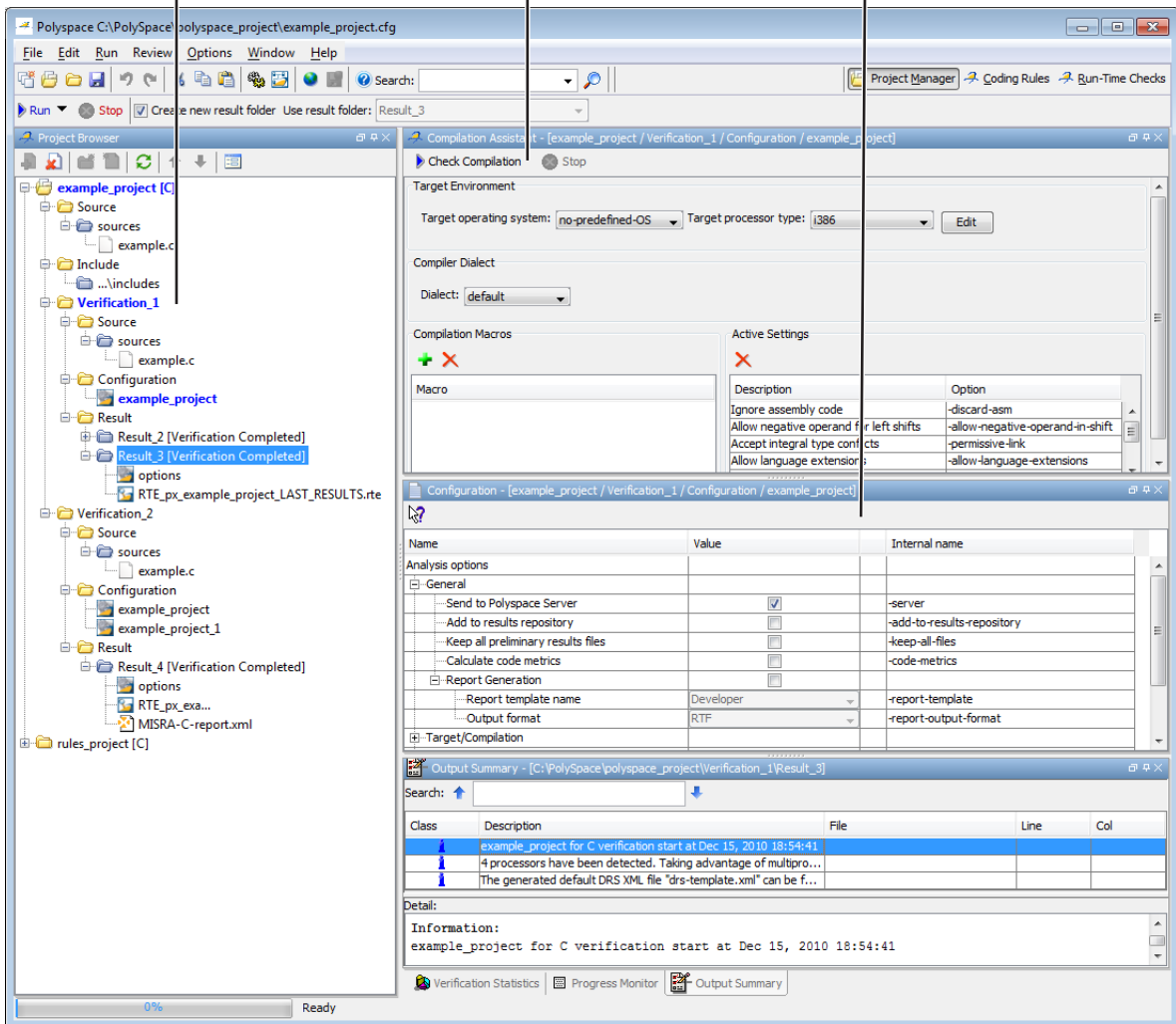
3 Setting Up a Verification Project

The Polyspace Verification Environment opens.

Specify source files
and include folders

Set target environment
and check compilation

Specify
analysis options



Monitor progress and view logs

By default, the Polyspace Verification Environment displays the Project Manager perspective. The Project Manager perspective has three main panes.

Use this section...	For...
Project Browser (upper-left)	Specifying: <ul style="list-style-type: none"> • Source files • Include folders • Results folder
Configuration (upper-right)	Specifying analysis options
Output (lower-right)	Monitoring the progress of a verification, and viewing status, log messages, and general verification statistics.

You can resize or hide any of these panes.

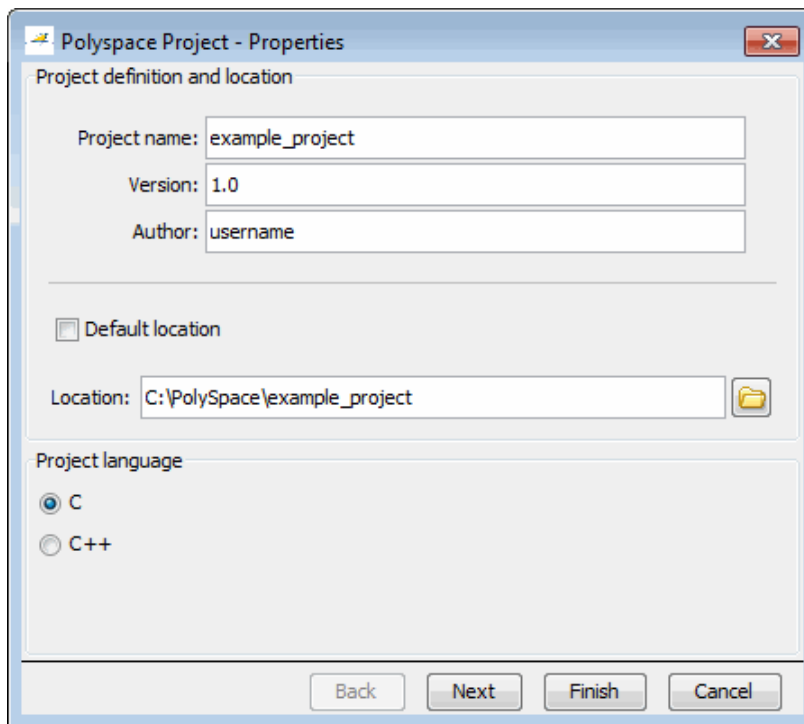
Creating New Projects

The Polyspace verification environment can manage multiple projects simultaneously. When you create a new project or open an existing project, the project is added to the Project Browser tree.

To create a new project:

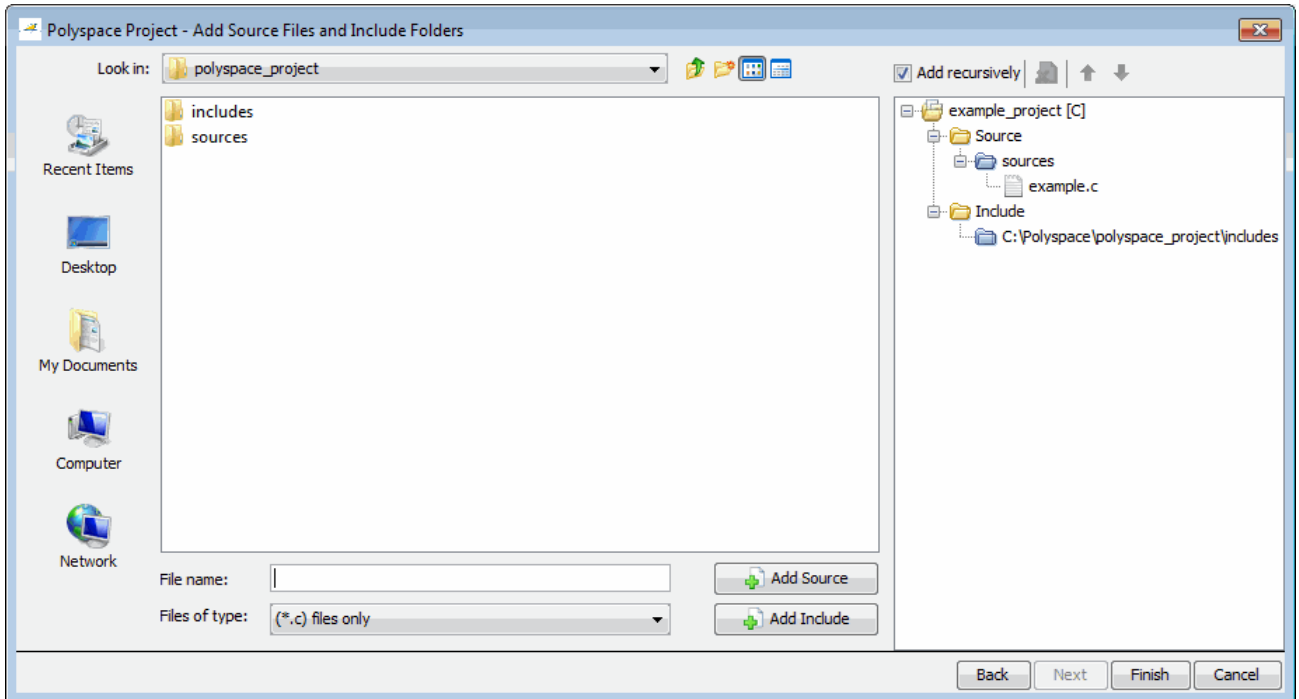
- 1 Select **File > New Project**.

The Polyspace Project – Properties dialog box opens:



- 2** In the **Project name** field, enter a name for your project.
- 3** If you want to specify a location for your project, clear the **Default location** check box, and enter a **Location** for your project.
- 4** In the Project language section, select **C**.
- 5** Click **Next**.

The Polyspace Project – Add Source Files and Include Folders dialog box opens.



6 The project folder Location you specified in step 3 should appear in **Look in**. If it does not, navigate to that folder.

7 Select the source files you want to include in the project, then click **Add Source**.

The source files appear in the Source tree for your project.

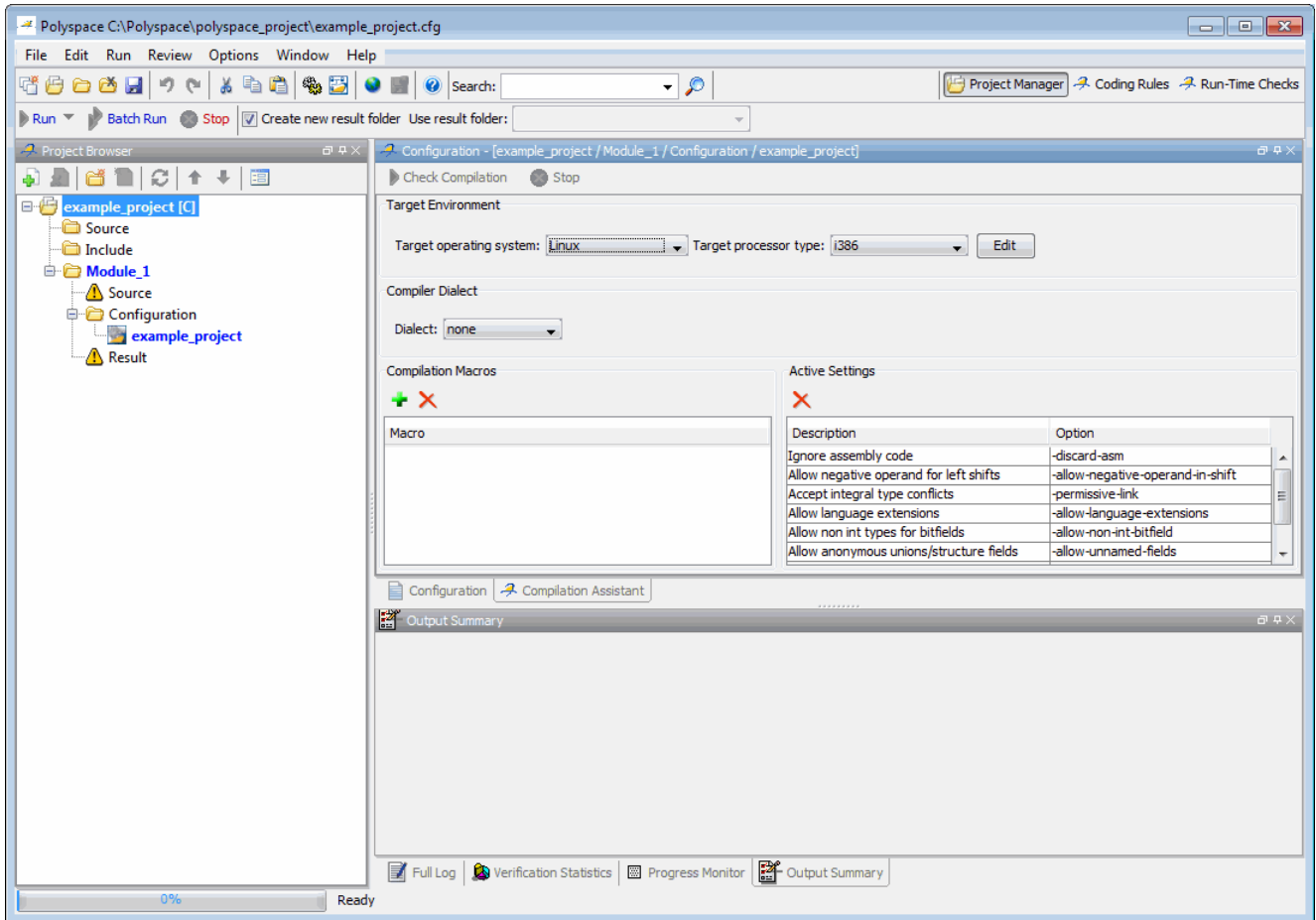
8 Select the Include folders you want to include in the project, then click **Add Include**.

The Include folders appear in the Include tree for your project.

9 Click **Finish**.

The new project opens in the Polyspace verification environment.

3 Setting Up a Verification Project



Opening Existing Projects

The Polyspace verification environment can manage multiple projects simultaneously. When you create a new project or open an existing project, the project is added to the Project Browser tree.

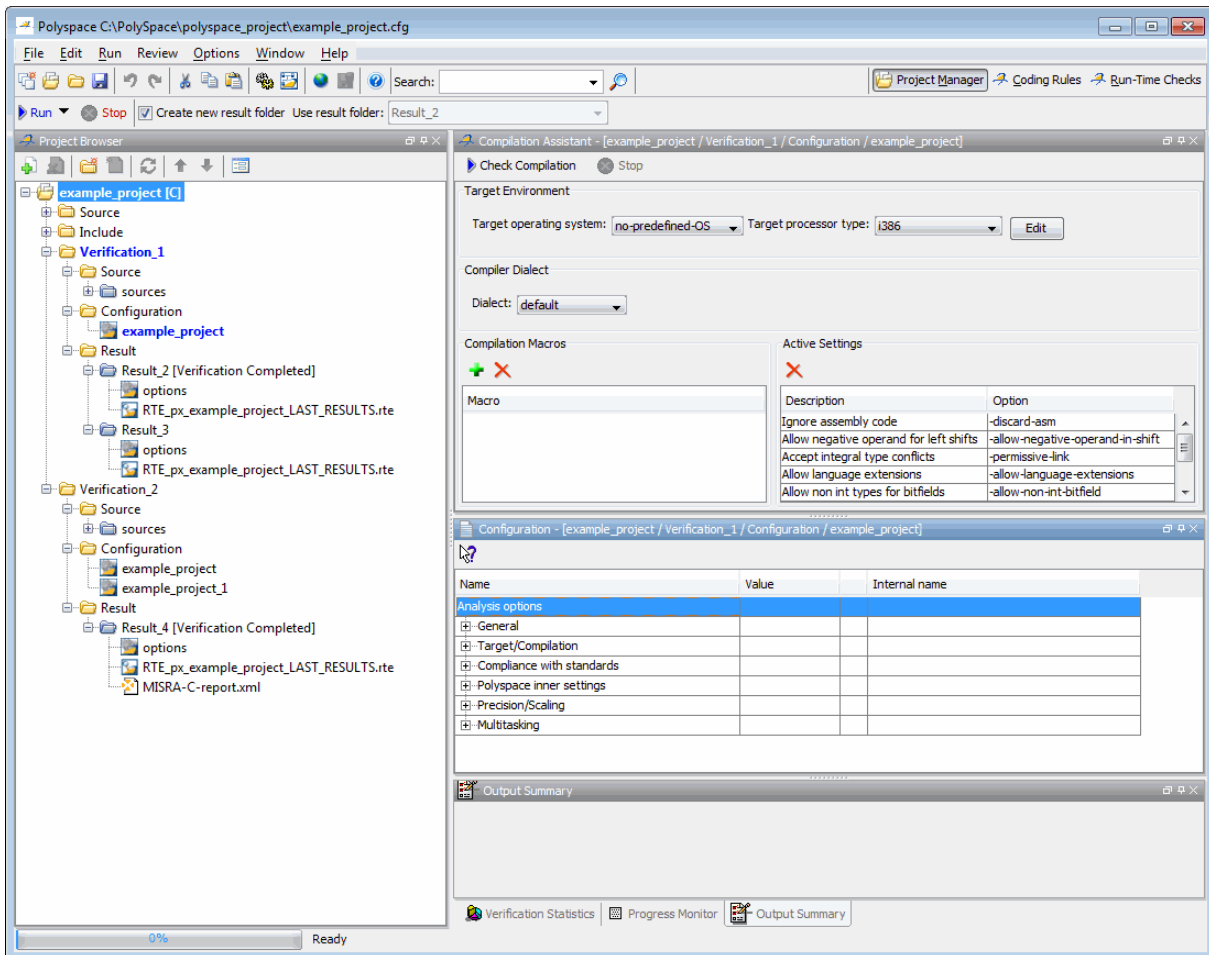
To open an existing project:

- 1 Select **File > Open Project**.

The **Please select a file** dialog box appears.

- 2 Select the project you want to open, then click **OK**.

The selected project opens in the Project Manager perspective.



Closing Existing Projects

The Polyspace verification environment can manage multiple projects simultaneously. When you create a new project or open an existing project,

the project is added to the Project Browser tree. To remove a project from the Project Browser tree, you must close the project.


To close a project:

- 1 In the Project Browser, select the project you want to close.
- 2 Right-click the project, then select **Close Active Project**.

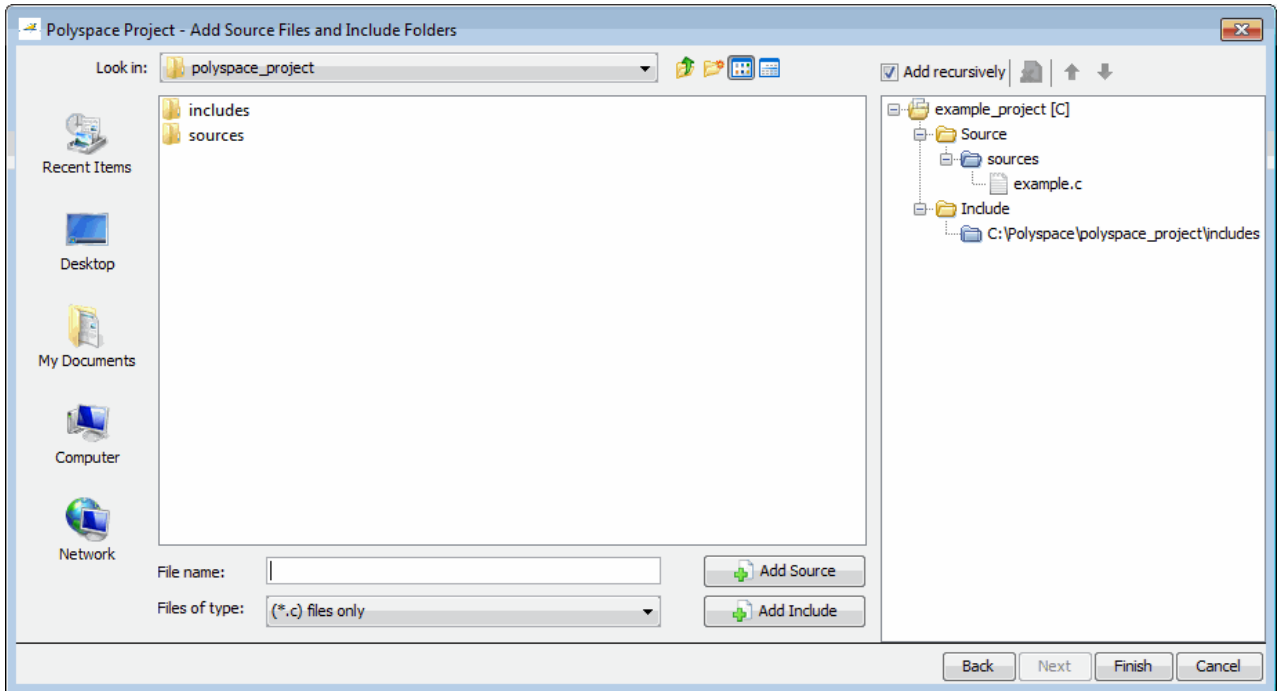
The project is closed and removed from the Project Browser tree.

Specifying Source Files

To specify the source files for your project:

- 1 In the Project Browser, select the **Source** folder.
- 2 Click the **Add source** icon  in the upper left of the Project Browser.

The Polyspace Project – Add Source Files and Include Folders dialog box opens.



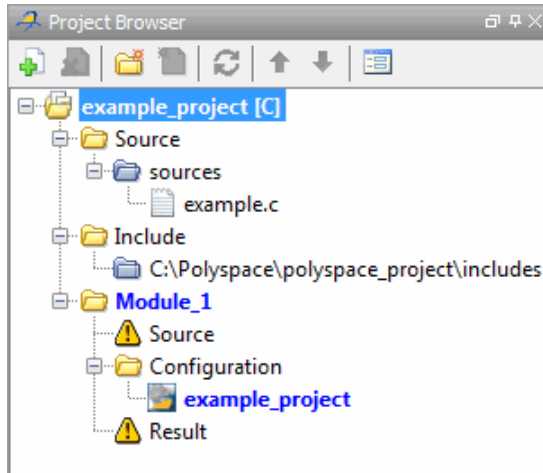
3 In the **Look in** field, navigate to the folder containing your source files.

4 Select the source files you want to include in the project, then click **Add Source**.

The source files appear in the Source tree for your project.

5 Click **Finish** to apply the changes and close the dialog box.

The source files you selected appear in the Project Browser.




Specifying Include Folders

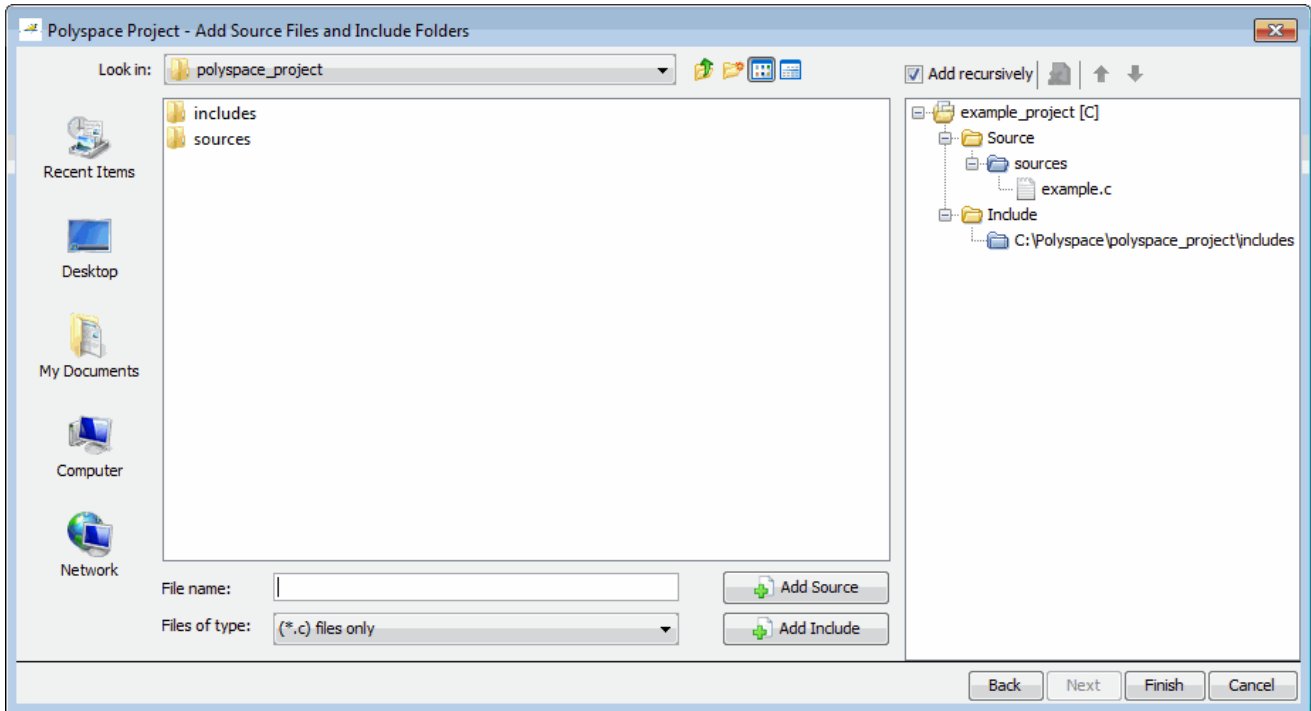
Polyspace software automatically adds the standard include folders to your project.

If your project uses additional include files, you can specify additional folders to include with your verification.

To specify the include folders for the project:

- 1 In the Project Browser, select the **Include** folder.
- 2 Click the **Add source** icon  in the upper left the Project Browser.

The Polyspace Project – Add Source Files and Include Folders dialog box opens.



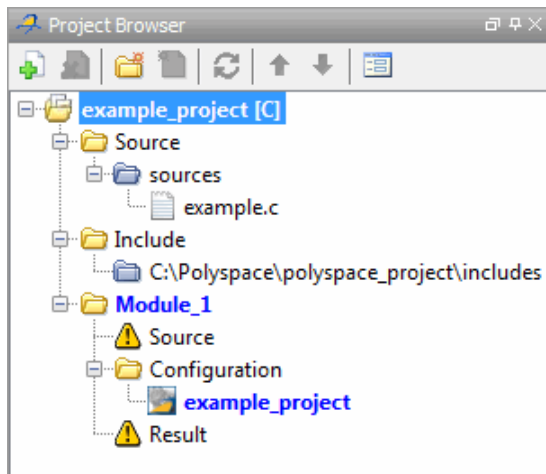
3 In the **Look in** field, navigate to the folder containing your Include files.

4 Select the Include folders you want to include in the project, then click **Add Include**.

The Include folders appear in the Include tree for your project.

5 Click **Finish** to apply the changes and close the dialog box.



The Include folders you selected appear in the Project Browser.



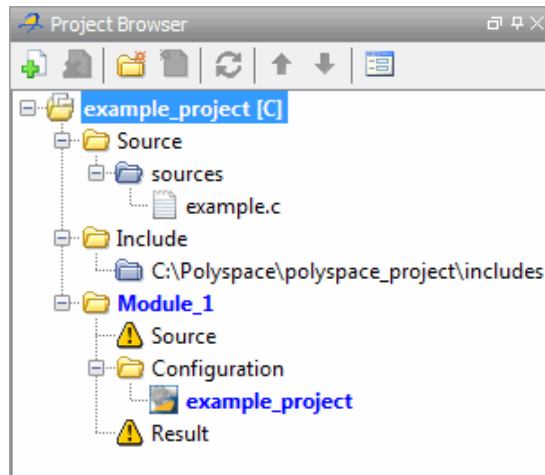
Managing Include File Sequence

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

To re-order the sequence of include folders for the project:

- 1 In the Project Browser, expand the **Include** folder.
- 2 Select the include folder you want to move.
- 3 Click the **Move up**  or **Move down**  icons in the Project Browser toolbar to move the include.

The Include folders are reordered in the Project Browser.




Creating Multiple Modules

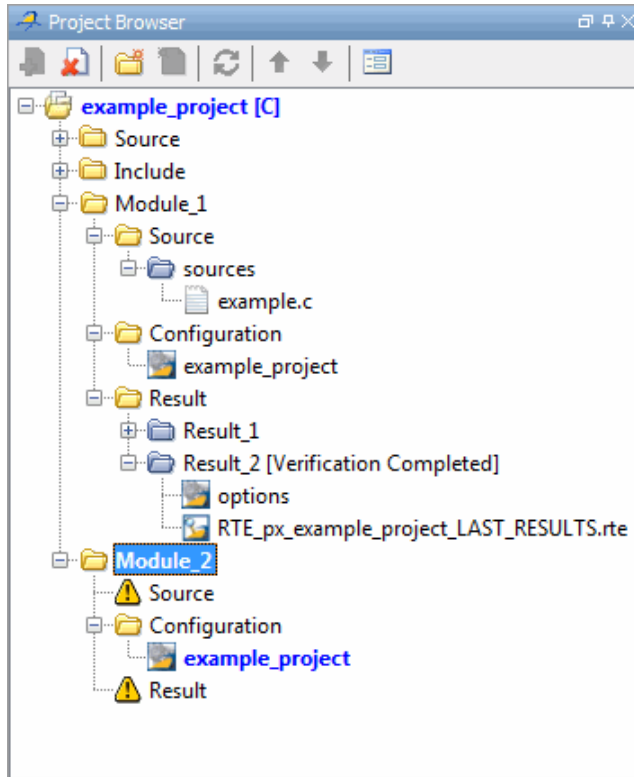
Each Polyspace project can contain multiple modules. Each of these modules can verify a specific set of source files using a specific set of analysis options.

By default, each module you create uses the same analysis options, allowing you to verify different subsets of source files using the same options. However, you can also create multiple configurations in each module, allowing you to change analysis options each time you run a verification.

To create a new module in your project:

- 1 In the Project Browser, select any project.
- 2 Click the Create a new module icon  in the upper left of the Project Browser.

A second module, `Module_(2)`, appears in the Project Browser tree.



- 3 In the Project Browser Source tree, right-click the files you want to add to the module, and select **Copy Source File to > Module_(2)**.

The source files appear in the Source tree of `Module_(2)`.

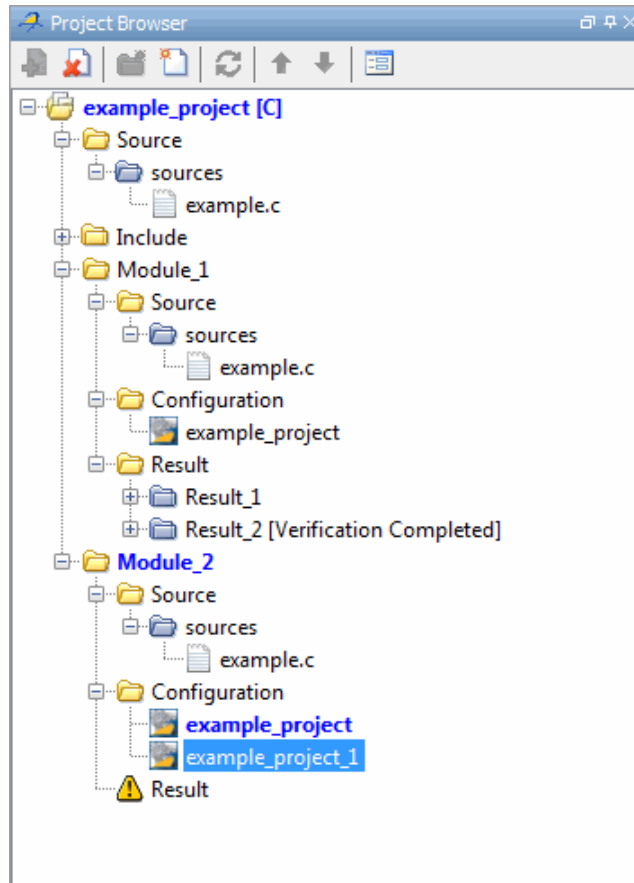
Creating Multiple Analysis Option Configurations

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

To create a new configuration in your project:

- 1 In the Project Browser, select any module.

- 2 Right-click the Configuration folder in the module, and select **Create New Configuration**. The new configuration appears in the Project Browser.



- 3 In the Configuration pane, specify the appropriate analysis options for the configuration.
- 4 Select **File > Save** to save your project with the new settings.

For detailed information about specific analysis options, see “Option Descriptions for C Code” in the *Polyspace Products or C Reference*.

Specifying Functions Not Called by Generated Main

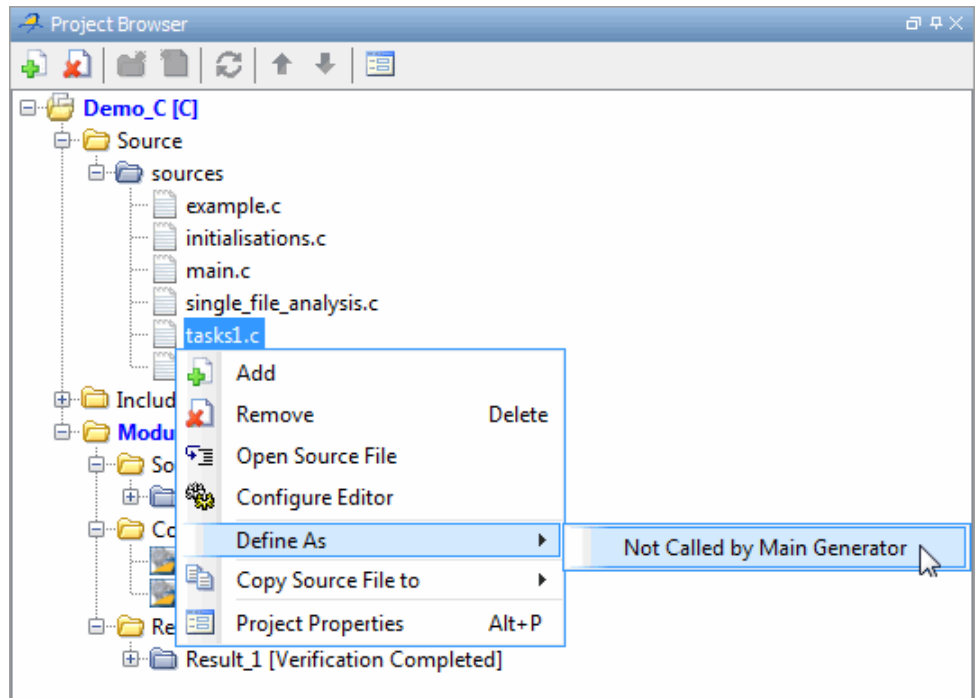
You can specify source files in your project that the main generator will ignore. Functions defined in these source files are not called by the automatically generated main.

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.

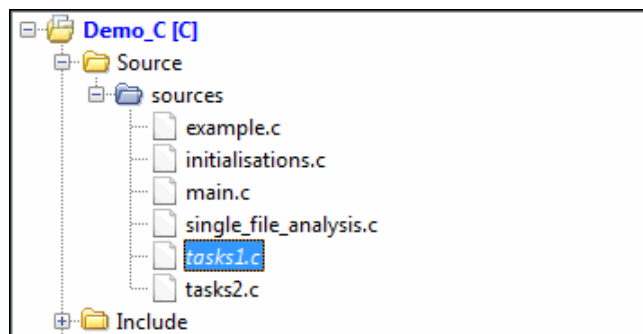
Note This option applies only to automatically generated mains. Therefore, you must also select the option **Generate a main** (-main-generator) for this option to take effect.

To specify a source file as not called by the main generator:

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



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



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

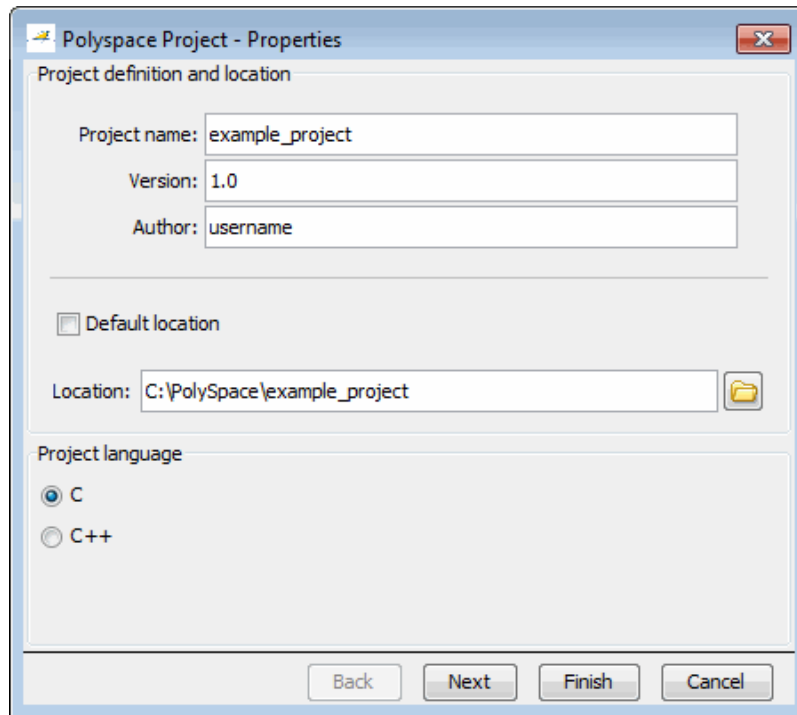
Changing Project Location

Polyspace software saves verification results in `Module_#` subfolders within the project folder. To change the location of your results, you must change the project location.

To change the location of an existing project:

- 1 In the Project Browser, right-click on the project name and select Project Properties.

The Polyspace Project – Properties dialog box opens:



- 2** Clear the **Default location** check box.
- 3** Enter the new **Location** for your project.
- 4** Click **Finish**.

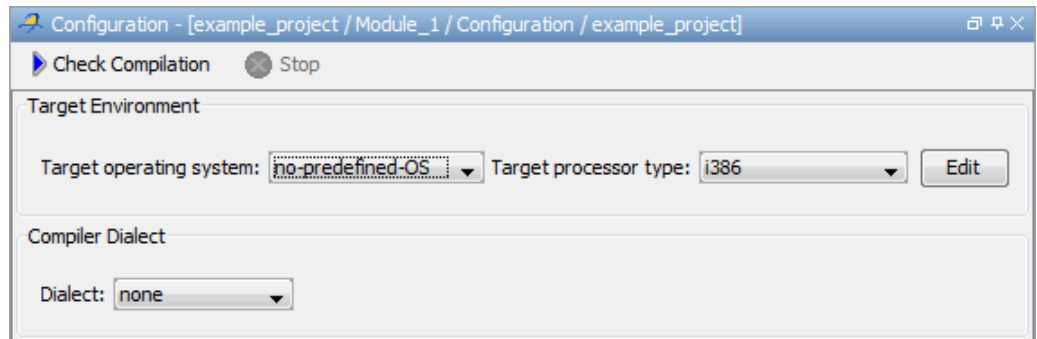
Specifying Target Environment

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

The Compilation Assistant window in the top-right section of the Project Manager perspective allows you to specify the target operating system and processor type for your application.

To specify the target environment for your application:

- 1 In the **Target operating system** drop-down menu, select the operating system on which your application is designed to run.



- 2 In the **Target processor type** drop down menu, select the processor on which your application is designed to run.

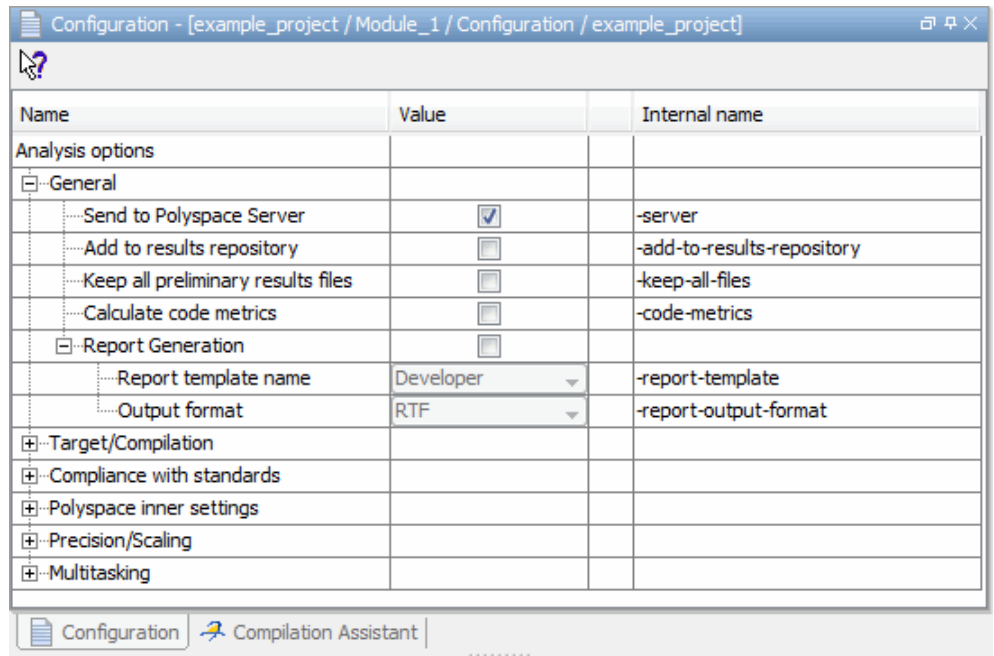
For more information about emulating your target environment, see “Setting Up a Target” on page 4-2.

Specifying Analysis Options

The Configuration window in the middle-right section of the Project Manager perspective allows you to set Analysis options that Polyspace software uses during the verification process. For more information about analysis options, see “Options Description” in the *Polyspace Products for C/C++ Reference*.

To specify analysis options for your project:

- 1 In the Configuration pane of the Project Manager perspective, expand **General**.
- 2 The General options appear.



3 Specify the appropriate analysis options for your project.

4 Select **File > Save** to save your project with the new settings.

For detailed information about specific analysis options, see “Option Descriptions for C Code” in the *Polyspace Products or C Reference*.

Configuring Text and XML Editors

Before you running a verification you should configure your text and XML editors in the Preferences. Configuring text and XML editors allows you to view source files and MISRA reports directly from the Polyspace Verification Environment.

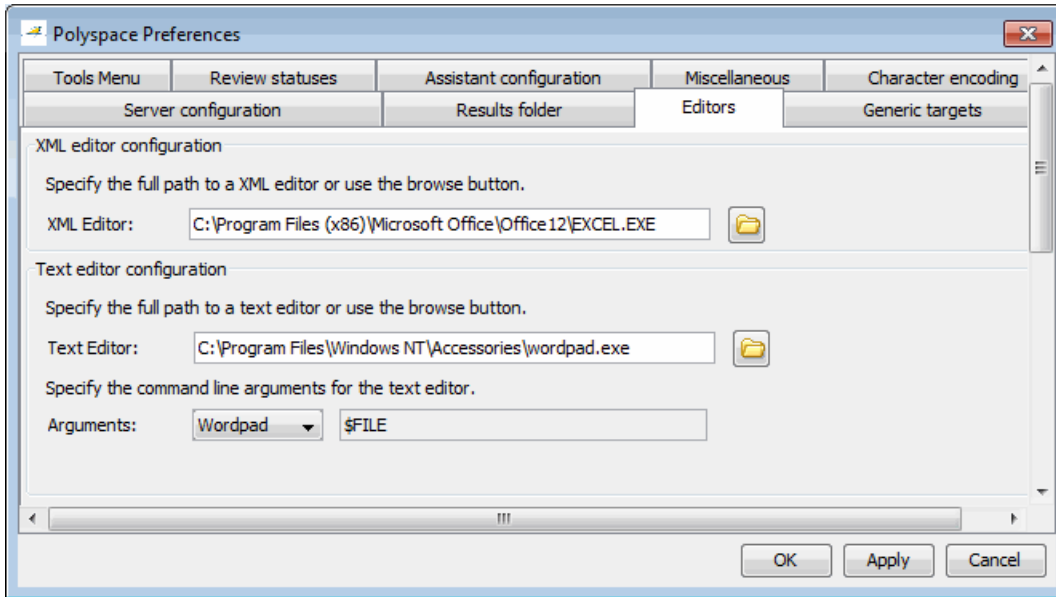
To configure your text and .XML editors:

1 Select **Options > Preferences**.

The Polyspace Preferences dialog box opens.

2 Select the **Editors** tab.

The Editors tab opens.



3 Specify an XML editor to use to view MISRA-C reports. For example:

`C:\Program Files\MSOffice\Office12\EXCEL.EXE`

4 Specify a Text editor to use to view source files from the Project Manager logs. For example:

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

5 Select your text editor in the Arguments drop-down menu to automatically specify the command line arguments for that editor.

- Emacs
- Notepad++
- UltraEdit
- VisualStudio

- Wordpad

If you are using another text editor, select **Custom** from the drop-down menu, and specify the command line arguments for the text editor.

6 Click **OK**.

Saving the Project

To save the project, select **File > Save**.

Polyspace software saves your project using the Project name and Location you specified when creating the project.

Specifying Options to Match Your Quality Objectives

While creating your project, you must configure analysis options to match your quality objectives.

This includes:

In this section...
“Quality Objectives Overview” on page 3-26
“Choosing Contextual Verification Options for C Code” on page 3-26
“Choosing Contextual Verification Options for C++ Code” on page 3-29
“Choosing Strict or Permissive Verification Options” on page 3-31
“Choosing Coding Rules” on page 3-33

Quality Objectives Overview

While creating your project, you must configure analysis options to match your quality objectives.

This includes choosing contextual verification options, coding rules, and options to set the strictness of the verification.

Note For information on defining the quality objectives for your project, see “Defining Quality Objectives” on page 2-5.

Choosing 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 “Choosing Robustness or Contextual Verification” on page 2-5.

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 Run-Time Checks perspective. For more information, see “Annotating Code to Indicate Known Run-Time Errors” on page 5-51.

To specify contextual verification for your project:

- 1** In the Configuration pane of the Project Manager perspective, expand **Polyspace Inner Settings**.
- 2** Expand the **Generate a main** and **Stubbing** options.

Name	Value	Internal name
Analysis options		
+ General		
+ Target/Compilation		
+ Compliance with standards		
- Polyspace inner settings		
+ Run a verification unit by unit	<input type="checkbox"/>	-unit-by-unit
- Generate a main	<input checked="" type="checkbox"/>	-main-generator
Variables written before loop	public ▾ ...	-variables-written-before-loop
Variables written in loop	none ▾ ...	-variables-written-in-loop
Functions called before loop	... ▾ ...	-functions-called-before-loop
Functions called in loop	unused ▾ ...	-functions-called-in-loop
Functions called after loop	... ▾ ...	-functions-called-after-loop
- Stubbing		
Variable/function range setup	... ▾ ...	-data-range-specifications
Stub all functions	<input type="checkbox"/>	-permissive-stubber
No automatic stubbing	<input type="checkbox"/>	-no-automatic-stubbing
+ Assumptions		
Continue with compile error	<input type="checkbox"/>	-continue-with-compile-error
Verification time limit		-timeout
Automatic Orange Tester	<input type="checkbox"/>	-prepare-automatic-tests
Run verification in 32 or 64-bit mode	auto ▾	-machine-architecture
Number of processes for multiple CPU	4	-max-processes
Other options		
+ Precision/Scaling		

3 To set ranges on variables, use the following options:

- **Variable/function range setup (-data-range-specifications)** – Activates the DRS option, allowing you to set specific data ranges for a list of global variables.
- **Variables written before loop (-variables-written-before-loop)** – Specifies how the generated main initializes global variables.

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

- **Functions called before loop (-functions-called-before-loop)** – Specifies an initialization function called after initialization of global variables but before the main loop.
- **Functions called in loop (-functions-called-in-loop)** – Specifies how the generated main calls functions.

5 To control stubbing behavior, use the following options:

- **No automatic stubbing (-no-automatic-stubbing)** – Specifies that the software will not automatically stub functions. The software lists the functions to be stubbed and stops the verification.
- **Stub all functions (-permissive-stubber)** – Specifies that the software stubs all functions, including those with function pointers as return type, or those with complex function pointers as parameters.

For more information on these options, see “Option Descriptions for C Code” in the *Polyspace Products for C/C++ Reference*.

Choosing 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 “Choosing Robustness or Contextual Verification” on page 2-5.

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 Run-Time Checks perspective. For more information, see “Annotating Code to Indicate Known Run-Time Errors” on page 5-51.

To specify contextual verification for your project:

- 1 In the Analysis options section of the Configuration pane, expand **Polyspace Inner Settings**.

2 Expand the **Generate a main** , **Main generation general options**, and **Stubbing** options.

Name	Value		Internal name
Analysis options			
⊕ General			
⊕ Target/Compilation			
⊕ Compliance with standards			
⊖ PolySpace inner settings			
⊕ Run a verification unit by unit	<input type="checkbox"/>		-unit-by-unit
⊕ Specify a Visual Studio compliant main	<input type="checkbox"/>		
⊖ Generate a main	<input checked="" type="checkbox"/>		-main-generator
⊕ Class name	custom	▼ ...	-class-analyzer
Function calls	unused	▼ ...	-main-generator-calls
First functions to call		▼ ...	-function-called-before-main
Write accesses to global variables	unit	▼ ...	-main-generator-writes-variables
⊖ Stubbing			
Variable/function range setup		▼ ...	-data-range-specifications
No automatic stubbing	<input type="checkbox"/>		-no-automatic-stubbing
⊕ Assumptions			

3 To set ranges on variables, use the following options:

- **Variable/function range setup (-data-range-specifications)** – Activates the DRS option, allowing you to set specific data ranges for a list of global variables.
- **Write accesses to global variables (-main-generator-writes-variables)** – Specifies how the generated main initializes global variables.

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

- **Function calls (-main-generator-calls)** – Specifies how the generated main calls functions.
- **First function to call (-function-called-before-main)** – Specifies an initialization function called after initialization of global variables but before the main loop.

5 To control stubbing behavior, use the following option:

- **No automatic stubbing (-no-automatic-stubbing)** – Specifies that the software will not automatically stub functions. The software list the functions to be stubbed and stops the verification.

For more information on these options, see “Options Description” in the *Polyspace Products for C++ Reference*.

Choosing 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 objectives 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 Run-Time Checks perspective. For more information, see “Annotating Code to Indicate Known Run-Time Errors” on page 5-51.

To specify the strictness of your verification:

- 1** In the Configuration pane of the Project Manager perspective, expand **Compliance with standards**.
- 2** Expand the **Strict** and **Permissive** options.

Name	Value	Internal name
Analysis options		
+ General		
+ Target/Compilation		
- Compliance with standards		
Code from DOS or Windows filesystem	<input checked="" type="checkbox"/>	-dos
+ Embedded assembler		
- Strict	<input type="checkbox"/>	-strict
Give all warnings	<input type="checkbox"/>	-Wall
- Permissive	<input type="checkbox"/>	-permissive
Allow non int types for bitfields	<input checked="" type="checkbox"/>	-allow-non-int-bitfield
Accept integral type conflicts	<input checked="" type="checkbox"/>	-permissive-link
Allow undefined global variables	<input checked="" type="checkbox"/>	-allow-undef-variables
Ignore overflowing computations on constants	<input type="checkbox"/>	-ignore-constant-overflows
Allow anonymous unions/structure fields	<input checked="" type="checkbox"/>	-allow-unnamed-fields
Allow negative operand for left shifts	<input checked="" type="checkbox"/>	-allow-negative-operand-in-shift
Allow language extensions	<input checked="" type="checkbox"/>	-allow-language-extensions
Ignore missing header files	<input type="checkbox"/>	-ignore-missing-headers
+ Check MISRA C rules	<input type="checkbox"/>	
+ Keil/IAR support	default ▾	-dialect
+ Polyspace inner settings		

3 In addition, expand **Polyspace Inner Settings > Assumptions**.

4 Use the following options to make verification more strict:

- **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 all global variables as non-initialized, therefore causing a red error if they are read before they are written to.
- **Give all warnings (-wall)** – Specifies that all C compliance warnings are written to the log file during compilation.

- **Strict (-strict)** – Specifies strict verification mode, which is equivalent to using the `-wall` and `-no-automatic-stubbing` options simultaneously.
- 5 Use the following options to make verification more permissive:
- **Enable pointer arithmetic out of bounds of fields (-allow-ptr-arith-on-struct)** – Enables navigation within a structure or union from one field to another.
 - **Allow negative operand for left shifts (-allow-negative-operand-in-shift)** – Verification allows a shift operation on a negative number.
 - **Ignore overflowing computations on constants (-ignore-constant-overflows)** – Verification is permissive with overflowing computations on constants.
 - **Allow non int types for bitfields (-allow-non-int-bitfield)** – Allows you to define types of bitfields other than signed or unsigned int.
 - **Allow undefined global variables (-allow-undef-variables)** – Verification does not stop due to errors caused by undefined global variables.
 - **Allow anonymous union/structure fields (-allow-unnamed-fields)** – Verification does not stop due to errors caused by unnamed fields in structures.
 - **Ignore missing header files (-ignore-missing-headers)** – Verification continues when the software detects that include files are missing.
 - **Dialect support (-dialect)** – Verification allows syntax associated with the IAR and Keil dialects.

For more information on these options, see “Option Descriptions for C Code” in the *Polyspace Products for C/C++ Reference*.

Choosing Coding Rules

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 “Setting Up Project to Check Coding Rules” on page 3-35.

Note If you are aware of coding rule violations, but still want to run a verification, you can annotate your code so that these known violations are highlighted in the Coding Rules perspective. For more information, see “Annotating Code to Indicate Known Coding Rule Violations” on page 5-47

Setting Up Project to Check Coding Rules

In this section...

“Polyspace Coding Rules Checker Overview” on page 3-35

“Checking Compliance with MISRA C Coding Rules” on page 3-35

“Checking Compliance with C++ Coding Rules” on page 3-37

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++ 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.

Checking 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 Configuration pane of the Project Manager perspective, expand **Compliance with standards**.

The Compliance with standards options appear.

- 2 Select the **Check MISRA C rules** check box.
- 3 Expand the **Check MISRA C rules** option.

Three options, **MISRA C rules configuration**, **Files and folders to ignore** and **Effective boolean types**, appear.

Name	Value	Internal name
Keep all preliminary results files	<input type="checkbox"/>	-keep-all-files
Calculate code metrics	<input type="checkbox"/>	-code-metrics
[-] Report Generation	<input type="checkbox"/>	
Report template name	Developer	-report-template
Output format	RTF	-report-output-format
[+] Target/Compilation		
[-] Compliance with standards		
Code from DOS or Windows filesystem	<input checked="" type="checkbox"/>	-dos
[+] Embedded assembler		
[+] Strict	<input type="checkbox"/>	-strict
[+] Permissive	<input type="checkbox"/>	-permissive
[-] Check MISRA C rules	<input checked="" type="checkbox"/>	
MISRA C rules configuration	all-rules	-misra2
Files and folders to ignore		-includes-to-ignore
Effective boolean types		-boolean-types
[+] Dialect support	none	-dialect

- 4 Specify:

- MISRA C rules to check
- Files, if any, 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 Chapter 11, “Checking Coding Rules”.

Checking 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 the coding rules checking option:

- 1 In the Analysis options, select **Compliance with standards > Coding rules checker**.

The software displays the coding rules options: `jsf-coding-rules`, `misra-cpp`, and `includes-to-ignore`.

<input type="checkbox"/> Coding rules checker			
<input checked="" type="checkbox"/> Check JSF C++ rules	<input checked="" type="checkbox"/>		
JSF C++ rules configuration		...	<code>-jsf-coding-rules</code>
<input type="checkbox"/> Check MISRA C++ rules	<input type="checkbox"/>		
MISRA C++ rules configuration		...	<code>-misra-cpp</code>
Files and folders to ignore		...	<code>-includes-to-ignore</code>

These options allow you to specify which rules to check and any files to exclude from the checker.

- 2 Select either the **Check JSF C++ rules** or **Check MISRA C++ rules** check box.
- 3 Specify which rules to check and which, if any, files to exclude from the checking.

Note For more information on using the coding rules checker, see Chapter 11, “Checking Coding Rules”.

Setting up Project to Automatically Test Orange Code (C Only)

In this section...
“Polyspace Automatic Orange Tester” on page 3-38
“Enabling the Automatic Orange Tester” on page 3-38

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 “Automatically Testing Orange Code” on page 9-52.

Enabling the Automatic Orange Tester

To enable the Automatic Orange Tester (-automatic-orange-tester):

- 1** Under **Analysis Options**, expand the **Polyspace inner settings** node.
- 2** Select the **Automatic Orange Tester** check box.

Name	Value	Internal name
Analysis options		
+ General		
+ Target/Compilation		
+ Compliance with standards		
- Polyspace inner settings		
+ Run a verification unit by unit	<input type="checkbox"/>	-unit-by-unit
+ Generate a main	<input type="checkbox"/>	-main-generator
+ Stubbing		
+ Assumptions		
Continue with compile error	<input type="checkbox"/>	-continue-with-compile-error
Verification time limit		-timeout
- Automatic Orange Tester	<input checked="" type="checkbox"/>	-automatic-orange-tester
Number of automatic tests	500	-automatic-orange-tester-tests-number
Maximum loop iterations	1000	-automatic-orange-tester-loop-max-iteration
Maximum test time	5	-automatic-orange-tester-timeout
Run verification in 32 or 64-bit mode	auto	-machine-architecture
Number of processes for multiple CPU core systems	4	-max-processes
Other options		
+ Precision/Scaling		
+ Multitasking		

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 “Automatically Testing Orange Code” on page 9-52.

Setting Up Project to Generate Metrics

In this section...
“About Polyspace Metrics” on page 3-40
“Enabling Polyspace Metrics” on page 3-40
“Specifying Automatic Verification” on page 3-41

About Polyspace Metrics

Polyspace Metrics is a Web-based tool for software development managers, quality assurance engineers, and software developers, which allows you 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.

For information on using Polyspace Metrics, see Chapter 12, “Software Quality with Polyspace Metrics”.

Enabling Polyspace Metrics

Before you can use Polyspace Metrics, you must run a Polyspace verification with the `-code-metrics` option enabled. This option enables a metrics computation engine that evaluates metrics for your code, and stores these metrics in a results repository.

To enable code metrics:

- 1 In the Analysis Options window, expand the **General** menu.
- 2 Select the **Add to results repository** check box.
- 3 Select the **Calculate code metrics** check box.

Name	Value	Internal name
Analysis options		
[-] General		
Send to Polyspace Server	<input checked="" type="checkbox"/>	-server
Add to results repository	<input checked="" type="checkbox"/>	-add-to-results-repository
Keep all preliminary results files	<input type="checkbox"/>	-keep-all-files
Calculate code metrics	<input checked="" type="checkbox"/>	-code-metrics
[-] Report Generation		
Report template name	Developer ▾	-report-template
Output format	RTF ▾	-report-output-format
[+] Target/Compilation		
[+] Compliance with standards		
[+] Polyspace inner settings		
[+] Precision/Scaling		
[+] Multitasking		

Polyspace Metrics are generated for the next verification.

Specifying 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.

For more information, see “Specifying Automatic Verification” on page 12-4.

Configuring Polyspace Project Using Visual Studio Project Information

You can extract informations from a Visual Studio® project file (vcproj) to help configure your Polyspace project.

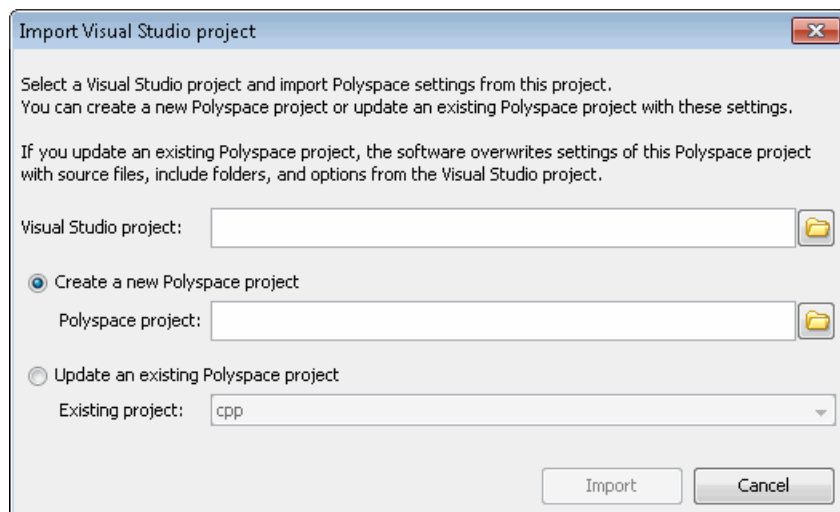
The Visual Studio import can retrieve the following information from a Visual Studio project:

- Source files
- Include folders
- Preprocessing directives (-D, -U)
- Polyspace specific options about dialect used

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.

For more information on using the Visual Studio integration, see Chapter 14, “Using Polyspace Software in Visual Studio”.

Emulating Your Runtime Environment

- “Setting Up a Target” on page 4-2
- “Verifying a C Application Without a “Main”” on page 4-33
- “Polyspace C++ Class Analyzer” on page 4-40
- “Specifying Data Ranges for Variables and Functions (Contextual Verification)” on page 4-56

Setting Up a Target

In this section...
“Target/Compiler Overview” on page 4-2
“Specifying Target Environment” on page 4-3
“Predefined Target Processor Specifications” on page 4-4
“Modifying Predefined Target Processor Attributes” on page 4-7
“Defining Generic Target Processors” on page 4-9
“Common Generic Targets” on page 4-10
“Viewing Existing Generic Targets” on page 4-11
“Deleting a Generic Target ” on page 4-12
“Compiling Operating System Dependent Code (OS-target issues)” on page 4-13
“Address Alignment” on page 4-19
“Ignoring or Replacing Keywords Before Compilation” on page 4-20
“Verifying Code That Uses Keil or IAR Dialects” on page 4-23
“How to Gather Compilation Options Efficiently” on page 4-30

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 affect whether errors (such as overflows) will occur.

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.

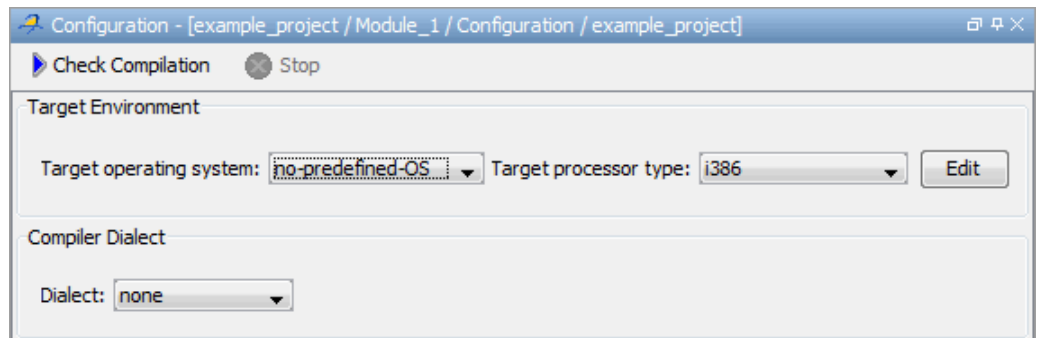
For detailed information on each Target/Compilation option, see “Target/Compilation Options” in the *Polyspace Products for C/C++ Reference*.

Specifying Target Environment

The Compilation Assistant window in the top-right section of the Project Manager perspective allows you to specify the target operating system and processor type for your application.

To specify the target environment for your application:

- 1 In the **Target operating system** drop-down menu, select the operating system on which your application is designed to run.



- 2 In the **Target processor type** drop down menu, select the processor on which your application is designed to run.

For detailed specifications for each predefined target processor, see “Predefined Target Processor Specifications” on page 4-4.

Specifying Target/Compilation Parameters

You can also set Target/Compilation options in the Configuration pane of the Project Manager.

To specify target parameters for your configuration:

- 1 In the Configuration pane of the Project Manager perspective, expand **Target/Compilation**.
- 2 The Target/Compilation options appear.

Name	Value		Internal name
Analysis options			
[-] General			
[-] Target/Compilation			
Target processor type	i386 ▼	...	-target
Target operating system	Linux ▼		-OS-target
Defined Preprocessor Macros		...	-D
Undefined Preprocessor Macros		...	-U
Include		...	-include
Command/script to apply to preprocessed files		...	-post-preprocessing-command
Command/script to apply after the end of the code verification		...	-post-analysis-command
[-] Compliance with standards			
[-] Polyspace inner settings			
[-] Precision/Scaling			
[-] Multitasking			

3 Select the **Target processor type** for your application.

4 Specify the **Operating system target** for your application.

For detailed specifications for each predefined target processor, see “Predefined Target Processor Specifications” on page 4-4.

For information on each **Target/Compilation** option, see “Target/Compilation Options” in the *Polyspace Products for C/C++ Reference*.

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. All kinds of pointers (near or far pointer) have 2 bytes (hc08) or 4 bytes (hc12) of width physically.
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 any processor described above, contact MathWorks technical support for advice.

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

Modifying 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 4-5 table.

To modify target processor attributes:

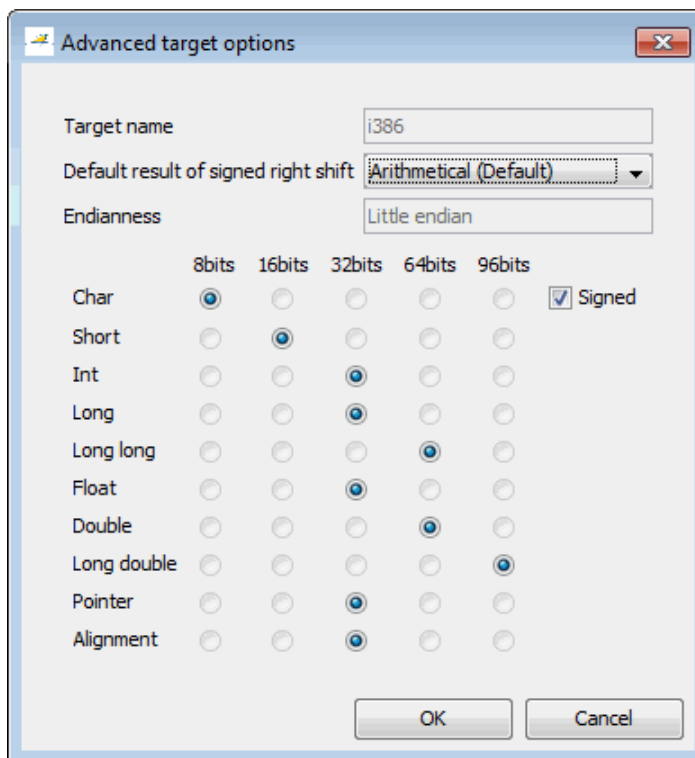
- 1 In the Configuration pane of the Project Manager perspective, expand **Target/Compilation**.

The Target/Compilation options appear.

Name	Value		Internal name
Analysis options			
[-] General			
[-] Target/Compilation			
Target processor type	i386	...	-target
Target operating system	Linux		-OS-target
Defined Preprocessor Macros		...	-D
Undefined Preprocessor Macros		...	-U
Include		...	-include
Command/script to apply to preprocessed files		...	-post-preprocessing-command
Command/script to apply after the end of the code verification		...	-post-analysis-command
[+] Compliance with standards			
[+] Polyspace inner settings			
[+] Precision/Scaling			
[+] Multitasking			

- 2 Select the **Target processor type** you want to use.
- 3 Select the browse button  to the right of the **Target processor type** drop-down menu.

The Advanced target options dialog box opens.



4 Modify the attributes as needed.

For information on each target option, see “Generic Target Options” in the *Polyspace Products for C/C++ Reference*.

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

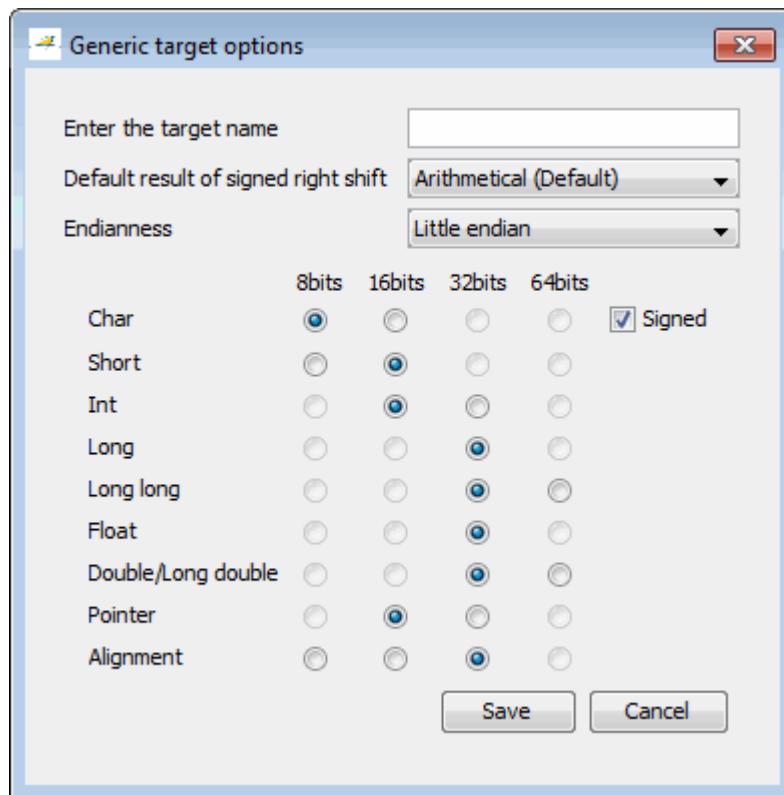
Defining Generic Target Processors

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 **Analysis options**, expand **Target/Compilation**.
- 2 In the **Target processor type** drop-down menu, select **mcpu...** (Advanced).

The **Generic target options** dialog box opens.



3 In **Enter the target name**, enter a name for your target.

4 Specify the appropriate 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 always 8, it implies that the storage assigned to arrays and structures is strictly 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” in the *Polyspace Products for C/C++ Reference*.

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

Viewing Existing Generic Targets

Generic targets that you create are listed in the Preferences dialog box.

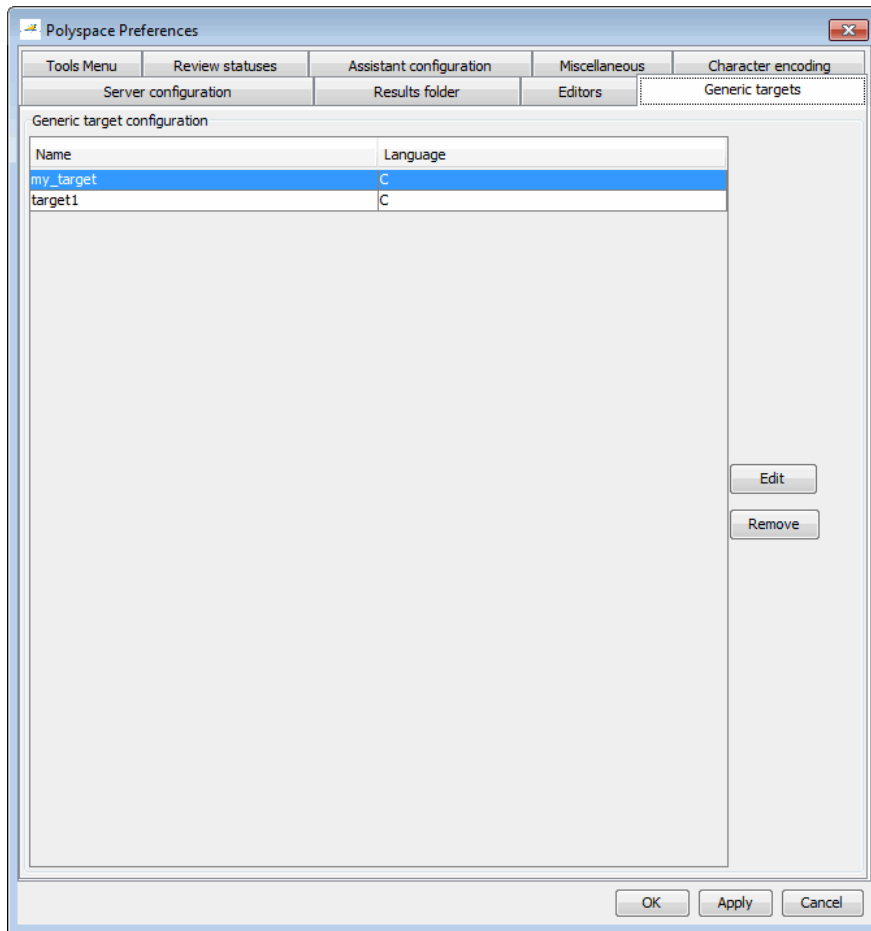
To view existing generic targets:

- 1 Select **Options > Preferences**.

The **Preferences** dialog box appears.

- 2 Select the **Generic targets** tab.

Previously defined generic targets appear in the generic targets list.



3 Click **Cancel** to close the dialog box.

Deleting a Generic Target

Generic targets that you create are stored as a Polyspace software preference. Generic targets remain in your preferences until you delete them.

Note You cannot delete a generic target if it is the currently selected target processor type for the project.

To delete a generic target:

1 Select **Options > Preferences**.

The **Preferences** dialog box appears.

2 Select the **Generic targets** tab.

3 Select the target you want to remove.

4 Click **Remove**.

5 Click **OK** to apply the change and close the dialog box.

Compiling Operating System Dependent Code (OS-target issues)

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 4-13
- “Predefined Compilation Flags for C++ Code” on page 4-15
- “My Target Application Runs on Linux” on page 4-18
- “My Target Application Runs on Solaris” on page 4-19
- “My Target Application Runs on Vxworks” on page 4-19
- “My Target Application Does Not Run on Linux, vxworks nor Solaris” on page 4-19

Predefined Compilation Flags for C Code

These flags concern the predefined **OS-target** options: `no-predefined-OS`, `linux`, `vxworks`, `Solaris` and `visual` (`-OS-target` option).

OS-target	Compilation flags	-include file and content
no-predefined-OS	-D __STDC__	
visual	-D __STDC__	-include <product_dir>/cininclude/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>/cininclude/pst-vxworks.h
linux	-D __STDC__ -D __GNUC__=2 -D __GNUC_MINOR__=6 -D __GNUC__=2 -D __GNUC_MINOR__=6 -D __ELF__ -Dunix -D __unix -D __unix__ -Dlinux -D __linux -D __linux__ -Di386 -D __i386 -D __i386__ -Di686 -D __i686 -D __i686__	<product_dir>/cininclude/pst-linux.h

OS-target	Compilation flags	-include file and content
	-Dpentiumpro -D__pentiumpro -D__pentiumpro__	
Solaris	-D__STDC__ -D__GNUC__=2 -D__GNUC_MINOR__=8 -D__GNUC__=2 -D__GNUC_MINOR__=8 -D__GCC_NEW_VARARGS__ -Dunix -D__unix -D__unix__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__ -D__svr4__ -D__SVR4	No -include file mentioned

Note The use of the OS-target option is entirely equivalent to the following alternative approaches.

- Setting the same -D flags manually, or
 - 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`):

-OS-target	Compilation flags	-include file	Minimum set of options
Linux	<pre>-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 -D__pentiumpro -D__pentiumpro__</pre>	<pre><product_dir>/ cininclude/ pst-linux.h</pre>	<pre>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></pre>
vxWorks	<pre>-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</pre>	<pre><product_dir>/ cininclude/ pstvxworks. h</pre>	<pre>polyspace-[desktop-]cpp \ -OS-target vxworks \ -I /your_path_to/ Vxworks_include_folders</pre>

-OS-target	Compilation flags	-include file	Minimum set of options
	-D__sun__ -D__svr4 -D__SVR4		
visual /visual6	-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>/ cinclude/ pstvisual. h	
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__		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>

-OS-target	Compilation flags	-include file	Minimum set of options
	-D__svr4 -D__SVR4		
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		polyspace-[desktop-]cpp \ -OS-target no-predefined-OS \ -I /your_path_to/ MyTarget_include_folders

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-c \  
-OS-target Linux \  
-I Polyspace_Install/Verifier/include/include-linux \  
-I Polyspace_Install/Verifier/include/include-linux/next \  
...
```

where the Polyspace product has been installed in the folder *Polyspace_Install*.

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

```
polyspace-c \  
-OS-target Linux \  
-I Polyspace_Install\Verifier\include\include-linux \  
-I Polyspace_Install\Verifier\include\include-linux\next \  
...
```

where the Polyspace product has been installed in the folder *Polyspace_Install*.

My Target Application Runs on Solaris

If Polyspace software runs on a Linux machine:

```
polyspace-c \  
-OS-target Solaris \  
-I /your_path_to_solaris_include
```

If Polyspace software runs on a Solaris™ machine:

```
polyspace-c \  
-OS-target Solaris \  
-I /usr/include
```

My Target Application Runs on Vxworks

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

```
polyspace-c \  
-OS-target vxworks \  
-I /your_path_to/Vxworks_include_folders
```

My Target Application Does Not Run on Linux, vxworks nor Solaris

If Polyspace software does not run on either a Solaris or a Linux machine:

```
polyspace-c \  
-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 guarantee that:

- that global `sizeof` and `offsetof` fields are optimum (i.e. as short as possible);

- the alignment of all 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.

Ignoring or Replacing Keywords Before Compilation

You can ignore noncompliant keywords such as “far” or 0x followed by an absolute address. The template provided in this section allows you to ignore these keywords.

To ignore keywords:

- 1 Save the following template in `c:\Polyspace\myTp1.pl`.
- 2 In the Target/Compilation options, select **Command/script to apply to preprocessed files**.
- 3 Select `myTp1.pl` using the browse button.

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

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

Content of the `myTpl.pl` file

```
#!/usr/bin/perl

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

```

Verifying Code That Uses 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 support** option (`-dialect`). 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 <code>-sfr-types</code> 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) 	<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</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
	will be considered as volatile.	-sfr-types sfr=8, \ sfr16=16	
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, epdata, 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. • Option -allow-unnamed-fields need to be used to allow anonymous struct fields. • On a conflict between a field of an anonymous struct with other identifiers: 	<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
	<ul style="list-style-type: none"> ▪ with a variable name, field name is hidden ▪ with a field of another anonymous struct at different scope, closer scope is chosen ▪ 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.

How to Gather Compilation Options Efficiently

The code is often tuned for the target (as discussed to “Verifying Code That Uses Keil or IAR Dialects” on page 4-23). Rather than applying minor changes to the code, create a single `polyspace.h` file which will contain 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 all 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);
```


Verifying a C Application Without a “Main”

In this section...

“Main Generator Overview” on page 4-33

“Automatically Generating a Main” on page 4-34

“Manually Generating a Main” on page 4-35

“Specifying Call Sequence” on page 4-36

“Specifying Functions Not Called by Generated Main” on page 4-37

“Main Generator Assumptions” on page 4-39

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 have Polyspace software generate one for you automatically.

When you run a verification on Polyspace Client for C/C++ software, the main is always generated. When you run a verification on Polyspace Server for C/C++ software, you can choose to automatically generate a main by selecting the **Generate a main** (-main-generator) option.

Polyspace Client for C/C++ Main Generator

The Polyspace Client for C/C++ product automatically checks your code for a main.

- If a main exists in the set of files, the verification uses that main.
- If a main does not exist, the tool generates a main using the options you specify.

Polyspace Server for C/C++ Main Generator

If you do not select the -main-generator option, a Polyspace Server for C/C++ verification stops if it does not detect a main. This behavior can help isolate files missing from the verification.

When you select the `-main-generator` option, the Polyspace Server for C/C++ product checks your code for a main.

- If a main exists in the set of files, the verification uses that main.
- If a main does not exist, the tool generates a main using the options you specify.

Automatically Generating a Main

When you run a client verification, or a server verification using the **Generate a main** (`-main-generator`) option, the software automatically generates a main.

The generated main has the following behavior.

- 1** It initializes any variables identified by the option `-variables-written-before-loop`.
- 2** It calls any functions specified by the option `-functions-called-before-loop`. This could be considered an initialization function.
- 3** It initializes any variables identified by the option `-variables-written-in-loop`.
- 4** It calls any functions specified by the option `-functions-called-in-loop`.
- 5** It calls any functions specified by the option `-functions-called-after-loop`. This could be a terminate function for a cyclic program.

For more information on the main generator, see “Main Generator Behavior for Polyspace Software”.

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

```

Manually Generating 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.

There are three steps involved in manually defining the main.

- 1** Identify the API functions and extract their declaration.
- 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 */
}
}
```

Specifying Call Sequence

Polyspace software verifies every function in any order. This means that in some particular situations, a function “f” might be called before a function “g”. In the default usage, Polyspace verification assumes that “f” and “g” can be called in any order. If some actions set by “f” must be executed before “g” is called, writing a main which will call “f” and “g” in the exact order will bring a higher selectivity.

Colored Source Code Example

With the default launching mode of Polyspace verification, no problem will be highlighted on the following example. With a bit of setup, more bugs can be found.

```
static char x;
static int y;

void f(void)
{
    y = 300;
}

void g(void)
{
    x = y; // red or green OVFL?
}
```

With knowledge of the relative call sequence between g and f: if g is called first, the assignment is green, otherwise its red. Thanks to the exact call order, an attempt to place 300 in a char fails, displaying a red.

Example of Call Sequence

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

Simply create a main that calls in the desired order the list of functions from the module.

Specifying Functions Not Called by Generated Main

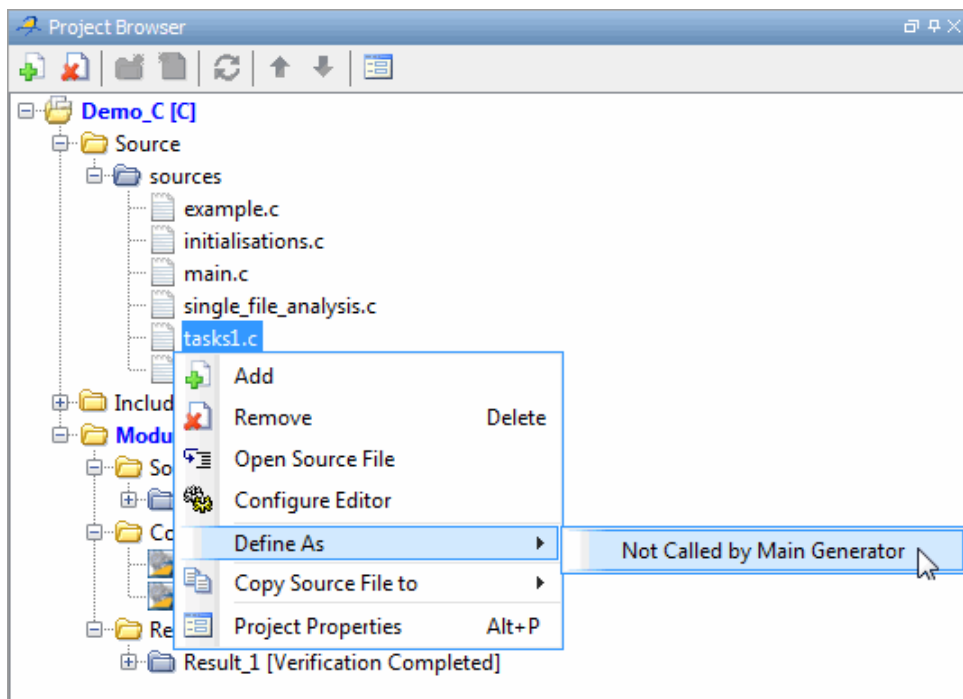
You can specify source files in your project that the main generator will ignore. Functions defined in these source files are not called by the automatically generated main.

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.

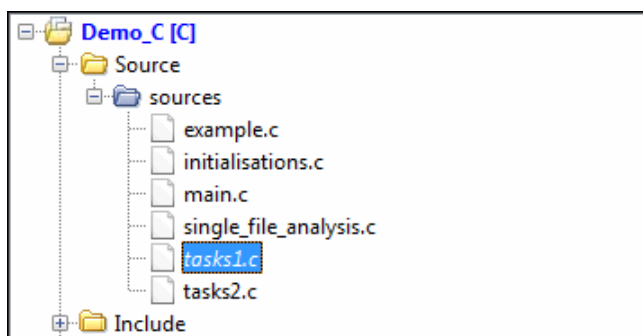
Note This option applies only to automatically generated mains. Therefore, you must also select the option **Generate a main** (-main-generator) for this option to take effect.

To specify a source file as not called by the main generator:

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



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



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

Main Generator Assumptions

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

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 due to the function itself, not the main.

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

Polyspace C++ Class Analyzer

In this section...

- “Why Provide a Class Analyzer” on page 4-40
- “How the Class Analyzer Works” on page 4-41
- “Sources Verified” on page 4-41
- “Architecture of the Generated main” on page 4-41
- “Class Verification Log File” on page 4-42
- “Characteristics of a Class and Messages in the Log File” on page 4-43
- “Behavior of Global variables and members” on page 4-44
- “Methods and Class Specificities” on page 4-46
- “Simple Class” on page 4-48
- “Simple Inheritance” on page 4-50
- “Multiple Inheritance” on page 4-52
- “Abstract Classes” on page 4-53
- “Virtual Inheritance” on page 4-54
- “Other Types of Classes” on page 4-55

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 bugs for all possible uses of the class. It can be considered free of bugs if run-time errors have been removed and functional tests are successful. 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 the possible uses of a class by:

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

- 2 Calling all methods (public, static, and protected) on previous objects in every order.
- 3 Calling all methods of the class between time zero and infinity.
- 4 Calling every destructor on 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 any 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 needed to verify a given class. All missing code will be 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 the 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 Run-Time Checks perspective, the main is not real. You cannot reuse or compile it.

Consider the example class `MathUtils` in `training.cpp` which is located in `Polyspace_Install\Examples\Demo_Cpp_Long\sources\training.cpp`. 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 An ASCII file representing the “pseudo” main can be seen in `C:\Polyspace_Results\ALL\SRC_polyspace_main.cpp`

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 will be analyzed in the log. Here is an example concerning the `MathUtils` class and associated log file which can be found at the root of the `C:\Polyspace_Results` folder:

```
*****
***
```

```

*** Beginning C++ source normalization
***
*****
Number of files : 1
Number of lines : 202
Number of lines with libraries : 7009
**** C++ source normalization 1 (Loading)
**** C++ source normalization 1 (Loading) took 20.8real, 7.9u + 11.4s
(1gc)
**** C++ source normalization 2 (P_INIT)
* 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()

```

It may be that a main is already defined in the files you are analyzing. In that case, you will receive this warning:

```

*** Beginning C++ source normalization

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

```

Characteristics of a Class and Messages in the Log File

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 Specificities

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 isvisible(void) {return visible;};
};
```

Although the two methods `Location::getx` and `Location::gety` are visible for derived classes, the generated main does not include these methods when analyzing the class `Point`.

Inherited members are considered to be volatile if they are not explicitly initialized in the father's constructors. In the example above, the two members `Location::x` and `Location::y` will be considered volatile. If we analyze the above example in its current state, the method `Location::Location(constructor)` will be stubbed.

Simple 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.

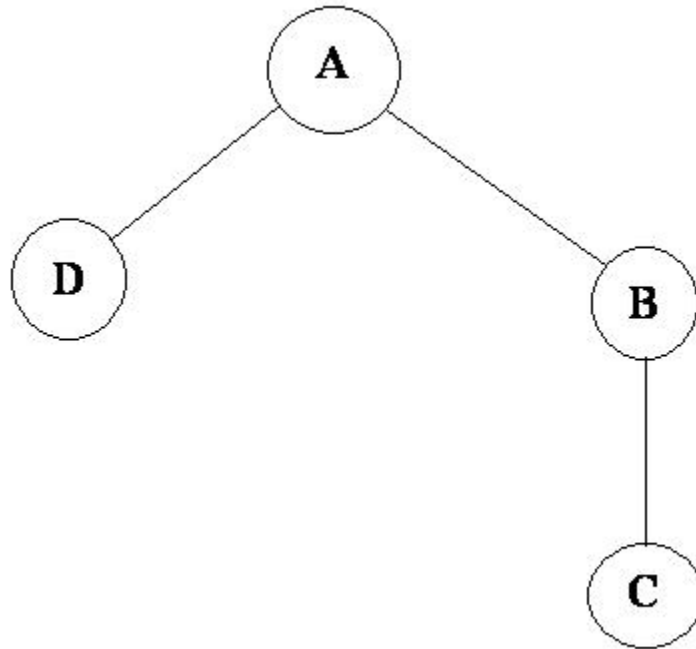
The verification of this class highlights two problems:

- 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

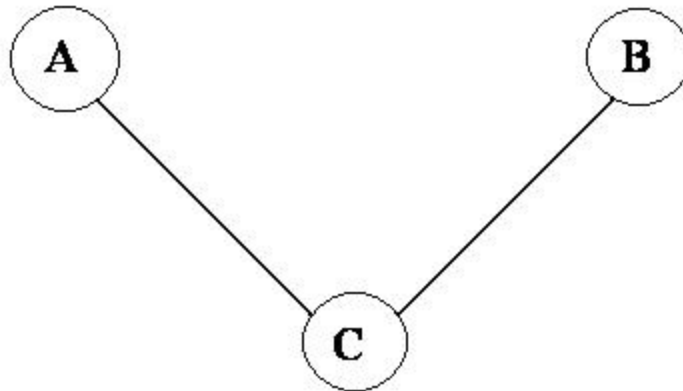
analyzer will not call A methods. In this case, the objective is to find bugs only in the class B code.

- 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 codes 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 bugs only 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:



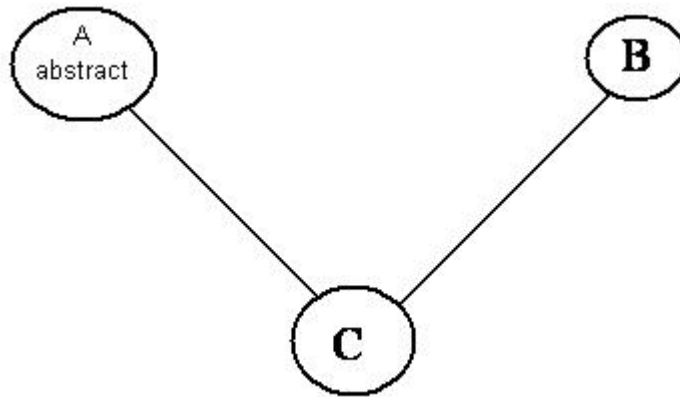
A and B are base classes of C.

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.

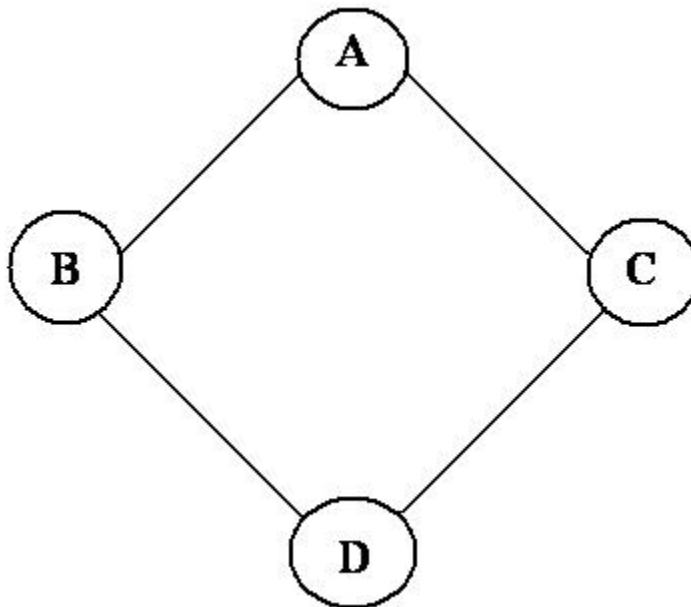
A and B are base classes of C.

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.

Specifying Data Ranges for Variables and Functions (Contextual Verification)

In this section...

“Overview of Data Range Specifications (DRS)” on page 4-56

“Specifying Data Ranges Using DRS Template” on page 4-57

“DRS Configuration Settings” on page 4-60

“Specifying Data Ranges Using Existing DRS Configuration” on page 4-64

“Editing Existing DRS Configuration” on page 4-65

“XML Format of DRS File ” on page 4-66

“Specifying Data Ranges Using Text Files” on page 4-73

“Variable Scope” on page 4-76

“Performing Efficient Module Testing with DRS” on page 4-80

“Reducing Oranges with DRS” on page 4-81

Overview of Data Range Specifications (DRS)

By default, Polyspace software performs *robustness verification*, proving that the software works under all 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 Data ranges are applied during verification level 2 (pass2).

Specifying Data Ranges Using DRS Template

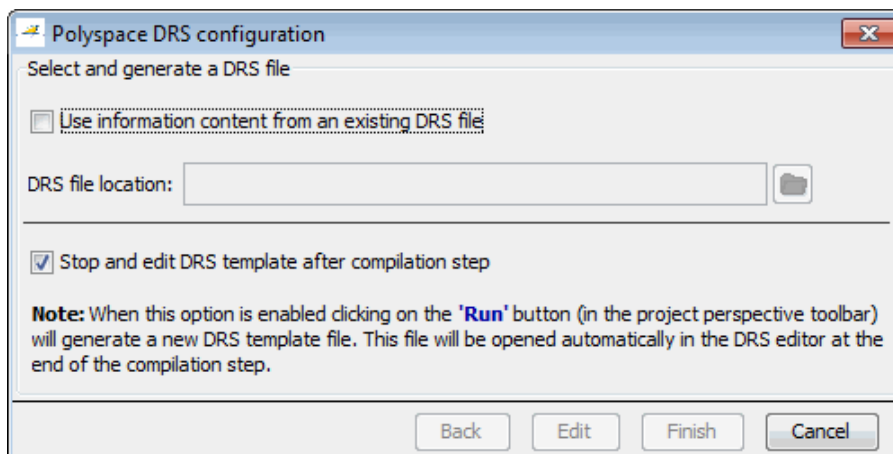
To use the DRS feature, you must provide a list of variables (or functions) and their associated data ranges.

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** Ensure that the Project contains all the source files and Include folders you want to verify, and specifies the Analysis options you want to use for the verification. The Compile phase of verification must complete successfully for the software to generate a DRS template.
- 3** In the Configuration pane of the Project Manager perspective, select **Polyspace inner settings > Stubbing**.
- 4** In the **Variable/function range setup** row, select the browse button .

The Polyspace DRS configuration dialog box opens.



5 Select **Stop and edit DRS template after compilation step**, then click **Finish**.

6 Click the **Run** button .

The software compiles the project and generates a DRS template. At the end of the Compile phase, verification stops and the Polyspace DRS configuration dialog box opens.

PolySpace DRS configuration


C:\PolySpace\polyspace_project\results\drs-template.xml

Search:

Name	File	Attributes	Type	Main Generator Called	Init Mode	Init Range	Initialize Pointer	Init Allocated	# Allocated Objects	Global Assert	Global Assert Range
Globals											
-v0	single_...	static	uint16		MAIN GEN...	min..max				NO	
-v1	single_...	static	int16		MAIN GEN...	min..max				NO	
-v2	single_...	static	int16		IGNORE	min..max				YES	0..1
-v3	single_...	static	uint8		IGNORE	min..max				YES	0..max
-v4	single_...	static	int16		INIT	25				NO	
-v5	single_...	static	int16		INIT	-100..100				NO	
-output_v6	single_...	static	int32		PERMANENT	0..max				NO	
-output_v7	single_...	static	int32		PERMANENT	min..0				NO	
-output_v1	single_...	static	int8		MAIN GEN...	min..max				NO	
-saved_values	single_...	static	int16 [127]								
User defined functions											
-generic_validation()	single_...			MAIN GENERATOR							
-all_values_s32()	single_...	static		MAIN GENERATOR							
-all_values_s16()	single_...	static		NO							
-all_values_u16()	single_...	static		NO							
-functional_ranges()	single_...	static		YES							
-new_speed()	single_...	static		YES							
-new_speed.arg1	single_...	static	int32		INIT	min..10					
-new_speed.arg2	single_...	static	int8		INIT	10..30					
-new_speed.arg3	single_...	static	uint8		INIT	30..max					
-new_speed.return	single_...	static	int32								
-reset_temperature()	single_...	static		MAIN GENERATOR							
-unused_fonction()	single_...	static		MAIN GENERATOR							
Stubbed functions											
-SEND_MESSAGE()	include.h	extern									
-SEND_MESSAGE.arg1	include.h		int32								
-SEND_MESSAGE.arg2	include.h		const int8 *					SINGLE			
-SEND_MESSAGE.*	include.h	const	int8		PERMANENT	min..max					
Non applicable											

Back Edit Finish Cancel

Note The DRS template file is generated in your results folder, named `drs-template.xml`.

- 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 4-60.
- Click  (Save DRS as), and save your DRS configuration file to a location other than the results folder.

Caution Do not save your DRS configuration file in the results folder. The results folder is overwritten each time you launch a verification, so your data ranges may be lost.

9 Click **Finish**. The Polyspace DRS configuration dialog box closes.

DRS Configuration Settings

The Polyspace DRS Configuration dialog box allows you specify data ranges for all the global variables, user defined functions, and stub functions in your project. The following table describes the parameters in the DRS Configuration interface.

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 all global variables in the Project. • User defined functions – Displays a list of all user-defined functions in the Project. Expand any function name to see a list of the input arguments for which you can specify a data range. • Stubbed functions – Displays a list of all stub functions in the Project. Expand any 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.

Column	Settings
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</code> (C) or <code>-main-generator-calls</code> (C++) parameter. • NO – Main generator will not call this function. • YES – Main generator will call this function.
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>

Column	Settings
	<ul style="list-style-type: none"> • MAIN GENERATOR – Pointer follows the options of the main generator. • IGNORE – Pointer is not initialized • INIT – Specify if the pointer is NULL, and how the pointed object is allocated (Initialize Pointer and Init Allocated options).
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> <hr/>

Column	Settings
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 4-64.</p> <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 4-64.</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>

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
}
```

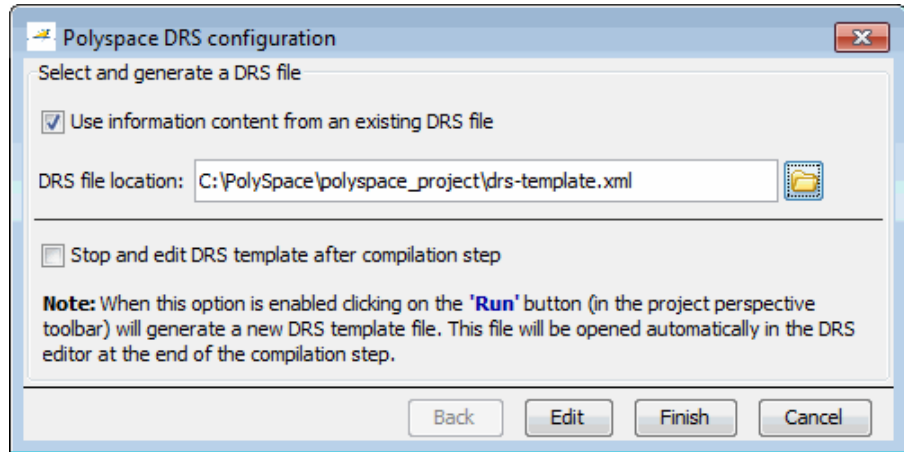
Specifying Data Ranges Using Existing DRS Configuration

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.
- 2 In the Configuration pane of the Project Manager perspective, select **Polyspace inner settings > Stubbing**.
- 3 In the **Variable/function range setup** row, select the browse button .

The Polyspace DRS configuration dialog box opens.



4 Select **Use information content from an existing DRS file**.

5 Specify the **DRS file location**, or click the Browse button  to select the DRS configuration file you want to use.

6 Click **Finish**.

The Polyspace DRS configuration dialog box closes.

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 launch a verification.

Editing Existing DRS Configuration

Once you have created a DRS configuration file for your Project, you can edit the configuration using the Polyspace DRS configuration interface.

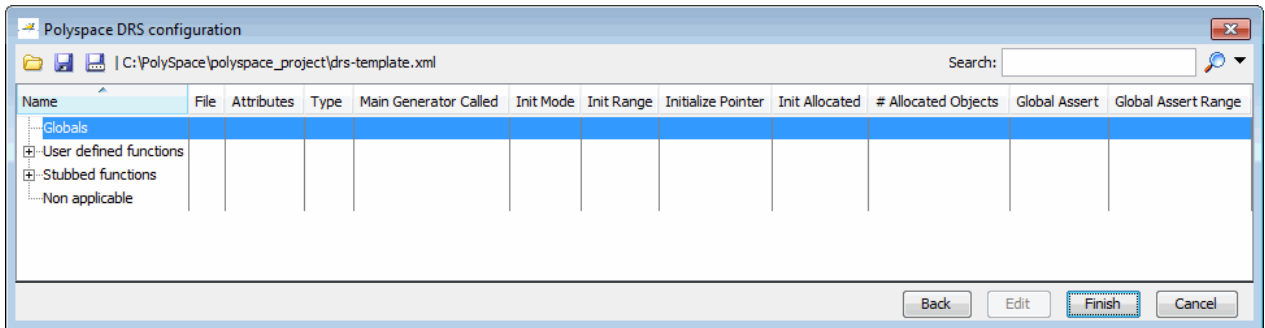
To edit an existing DRS configuration:

1 Open the Project.


2 In the Configuration pane of the Project Manager perspective, select **Polyspace inner settings > Stubbing**.

3 In the **Variable/function range setup** row, select the browse 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 Click  (Save DRS), to save your DRS configuration file.

6 Click **Finish**.

The Polyspace DRS configuration dialog box closes.

XML Format of DRS File

Syntax Description – XML Elements

The DRS file contains the following XML elements:

- “<file> mark” on page 4-67
- “<scalar> mark” on page 4-67
- “<pointer> mark” on page 4-68
- “<array> and <struct> marks” on page 4-69
- “<function> mark” on page 4-70

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 name is mandatory for scope marks `<file>` and `<function>`. For variable marks, 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.
- **(****)** – 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 4-70.

<file> mark. The file mark has only one element “name”. This element must contain the complete path to the file or only the file name.

<scalar> mark.

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

<pointer> mark.

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_allocated	MAIN_GENERATOR NONE SINGLE MULTI disabled

<array> and <struct> marks.

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

Field	Syntax
complete_type (*)	<i>type</i>
Attributes (***)	volatile extern static const

<function> mark.

Field	Syntax
Name (**)	<i>name</i>
line (*)	<i>line</i>
main_generator_called	MAIN_GENERATOR YES NO
Attributes (***)	volatile extern static const

Valid Modes and Default Values

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
Global variables	Base type	Declared scalar	MAIN_ - GENERATOR IGNORE INIT PERMANENT	YES NO			Main generator dependant
		Defined scalar	MAIN_ - GENERATOR IGNORE INIT PERMANENT	YES NO			Main generator dependant

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
		Static 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				

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
Global variables	Pointer	Extern pointer	IGNORE INIT		May-be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May-be NULL max MULTI
		Pointer	MAIN_ GENERATOR IGNORE INIT		May-be NULL Not NULL NULL	NONE SINGLE MULTI	Main generator dependant
		Pointed volatile scalar	not supported	not supported			
		Pointed other scalars	INIT	not supported			IINIT min..max
		Pointed pointer	INIT	not supported			INIT May-be NULL max MULTI
		Pointed function	not supported	not supported			
Function parameters	Userdef function	Scalar parameters	INIT	not supported			INIT min..max
		Pointer parameters	INIT	not supported	May-be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May-be NULL max MULTI
		Other parameters	Refer to parameter type				

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
	Stubbed function	Scalar parameter	disabled	not supported			
		Pointer parameters	disabled		disabled	NONE SINGLE MULTI	MULTI


Specifying Data Ranges Using Text Files

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

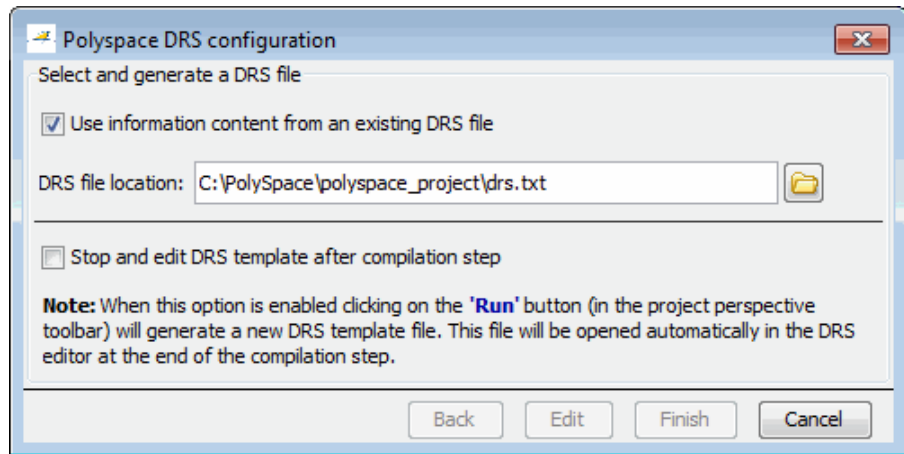
You can specify data ranges using the Polyspace DRS configuration interface (see “Specifying Data Ranges Using DRS Template” on page 4-57), or you can provide a text file that contains a list of variables and data ranges.

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 4-75.
- 2** Open your Project.
- 3** In the Configuration pane of the Project Manager perspective, select **Polyspace inner settings > Stubbing**.
- 4** In the **Variable/function range setup** row, select the browse button .

The Polyspace DRS configuration dialog box opens.



5 Select **Use information content from an existing DRS file**.

6 Specify the **DRS file location**, or click the Browse button  to select the DRS text file you want to use.

7 Click **Finish**.

The Polyspace DRS configuration dialog box closes.

8 Select **File > Save Project** to save your Project settings, including the DRS text file location.

When you launch 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.

You can continue to use the DRS text file for future verifications, or change the **DRS file location** to specify the generated file `drs-template.xml` (See “Specifying Data Ranges Using Existing DRS Configuration” on page 4-64).

If you specify the `.xml` template, you can then edit your data ranges using the DRS configuration interface (see “Editing Existing DRS Configuration” on page 4-65).

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 initializsation
# 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
```

Variable Scope

DRS supports variables with external linkages, const variables, and defined variables. In addition, extern variables are supported with the option `-allow-undef-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
Structure field	Ok	Ok	Ok	Only for int and real fields, including arrays or structures of int or real fields (see below)

	init	permanent	globalassert	comments
Structure field in array	Ok	No effect	No effect	Only when leaves are <code>int</code> or <code>real</code> . Moreover the syntax is the following: <code><array_name>. <field_name></code>
Array	Ok	Ok	Ok	Only for <code>int</code> and <code>real</code> 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 4-79.
Complete structure	No effect	No effect	No effect	
Array cell	No effect	No effect	No effect	Example: <code>array[0], array[10] ...</code>

	init	permanent	globalassert	comments
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 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.

Performing 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.	↑	↓

Reducing Oranges with DRS

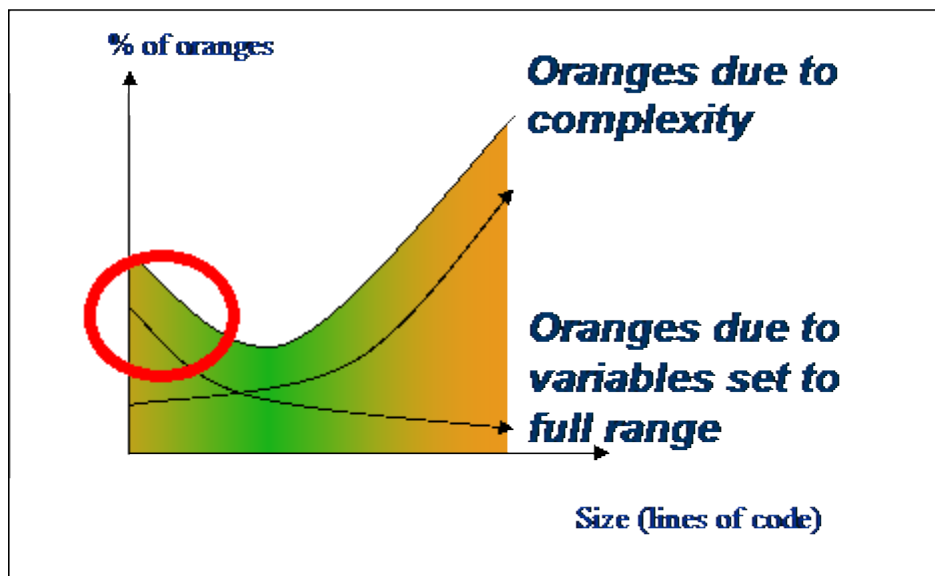
When performing robustness (worst case) verification, data inputs are always set to their full range. Therefore, every operation on these inputs, even a simple “one_input + 10” can produce an overflow, as the range of one_input varies between the min and the max of the type.

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 on the variable. For example, if you specify that “one-input” can vary between 0 and 10, Polyspace software will definitely know that:

- `one_input + 100` will never overflow
- the results of this operation will always be between 100 and 110

This not only eliminates the local overflow orange, 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?

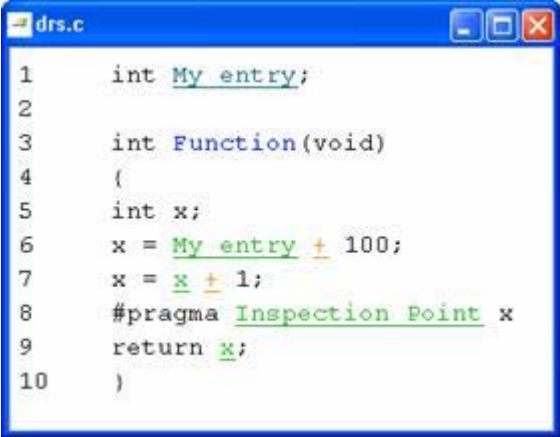
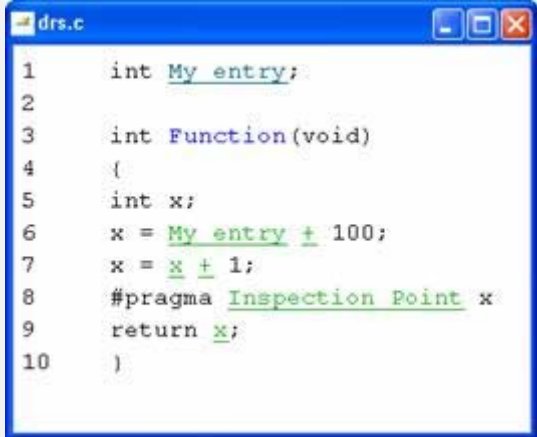
Removing oranges caused by full-range (worst-case) data can drastically reduce the total number of orange checks, especially when used on verifications of small files or modules. However, the number of orange checks caused by code complexity is not effected by DRS. For more information on oranges caused by code complexity, see “Subdividing Code” on page 7-60.

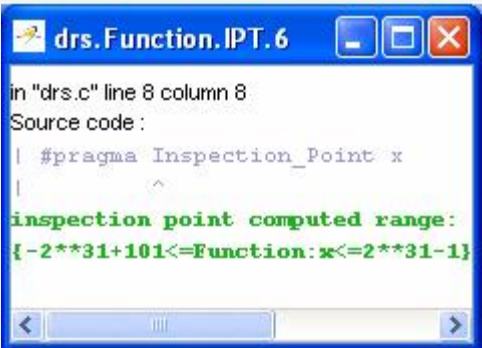
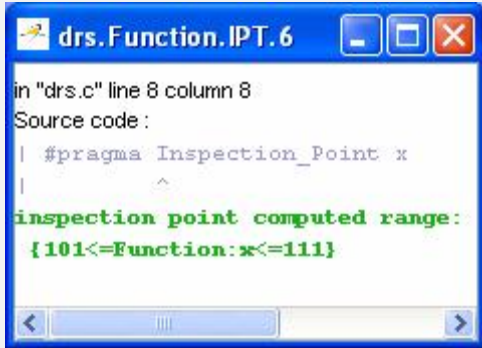
This section describes how DRS reduces oranges on files or modules only.

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 “My_entry“ 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 “My_entry” 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]
	

Preparing Source Code for Verification

- “Stubbing” on page 5-2
- “Preparing Code for Variables” on page 5-24
- “Preparing Code for Built-In Functions” on page 5-30
- “Preparing Multitasking Code” on page 5-32
- “Highlighting Known Coding Rule Violations and Run-Time Errors” on page 5-47
- “Types Promotion” on page 5-55
- “Verifying “Unsupported” Code” on page 5-58

Stubbing

In this section...
“Stubbing Overview” on page 5-2
“Manual vs. Automatic Stubbing” on page 5-2
“Stubbing Examples” on page 5-6
“Automatic Stubbing Behavior for C++ Pointer/Reference ” on page 5-9
“Specifying Functions to Stub Automatically” on page 5-10
“Constraining Data with Stubbing” on page 5-13
“Default and Alternative Behavior for Stubbing (PURE and WORST)” on page 5-18
“Function Pointer Cases” on page 5-21
“Stubbing Functions with a Variable Argument Number” on page 5-21
“Stubbing Standard Library Functions” on page 5-23

Stubbing Overview

A function stub is a small piece of code that emulates the behavior of a missing function.

Stubs do not need to model the details of functions or procedures. They represent only the effect that the code might have on the remainder of the system.

Stubbing allows you to verify code before all functions are developed.

Manual vs. Automatic Stubbing

The approach you take to stubbing can have a significant impact on the speed and precision of your verification.

In Polyspace verification, there are two types of stubs:

- **Automatic stubs** – When you attempt to verify code that calls an unknown function, the software automatically creates a stub function based on the

function prototype (the function declaration). Automatic stubs do not provide insight into the behavior of the function.

- **Manual stubs** – You create these stub functions to emulate the behavior of the missing functions, and manually include the stub functions in the verification with the rest of the source code.

By default, Polyspace software automatically stubs functions. However, in some cases you may want to manually stub functions. For example, when:

- Automatic stubbing does not provide an adequate representation of the code that it represents— both in regard to missing functions and assembly instructions.
- You verify the entire code. When the verification stops, it means the code is not complete.
- You want to improve the selectivity and speed of the verification.
- You want to gain precision by restricting return values generated by automatic stubs.
- You need to work with a function that writes to global variables.

For Example:

```
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.

Note Automatically generated stubs do not deinitialize variables that are given as parameters.

Deciding Which Stub Functions to Provide

In the following section, *procedure_to_stub* can represent either procedure or a sequence of assembly instructions which would be automatically stubbed in the absence of a manual stub. (For more information, refer to “Ignoring Assembly Code” on page 5-58).

Stubs do not need to model the details of functions or procedures. They represent only the effect that the code might have on the remainder of the system.

Consider *procedure_to_stub*. If it 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), then you can stub *procedure_to_stub* with an empty action (`void procedure(void)`). Polyspace does not need a concept of timing because the software takes into account all possible scheduling and interleaving of concurrent execution. Therefore, there is no need to 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 procedure(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 *procedure_to_stub* and why. Do not stub the concerned *procedure_to_stub* if:
 - The variable is volatile.

- The variable is a task list. Such lists are accounted for by default because all tasks declared with the `-task` option are automatically modelled as though they have been started. Write a `procedure_to_stub` 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.

Follow the Data Flow:

- Polyspace software uses only the C code which is provided.
- Polyspace does not need to be informed of timing constraints because all possible sequencing is taken into account.

Example

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.

Summary

Stub manually to gain precision by restricting return values generated by automatic stubs, for example, when you work with a function that writes to global variables.

Stub automatically to minimize preparation time. No run-time error is introduced by automatic stubbing.

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. The default stub would therefore have no 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	

File: <code>connection_lib</code>		Function: <code>Lib_connection_open</code>
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 stubbing it manually to accurately reflect 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 which 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, automatically stubbed C++ functions behave differently than 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
```


Specifying Functions to Stub Automatically

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

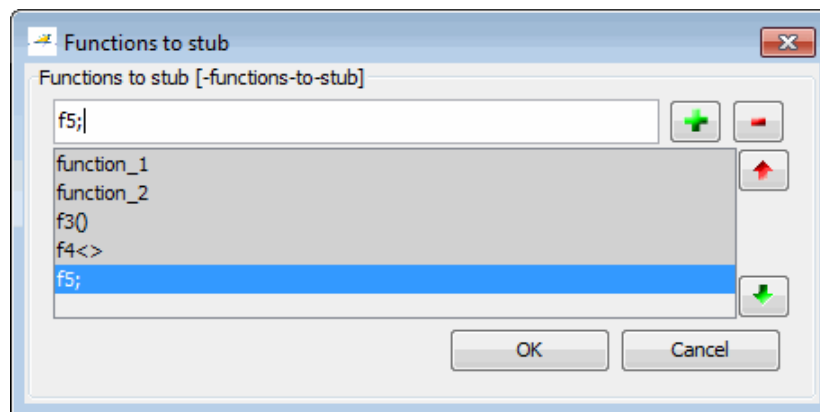
To specify functions to stub:


- 1 In the Configuration pane of the Project Manager perspective, expand **Polyspace inner settings > Stubbing**.
- 2 The Stubbing options appear.

Name	Value	Internal name
Analysis options		
+ General		
+ Target/Compilation		
+ Compliance with standards		
- Polyspace inner settings		
+ Run a verification unit by unit	<input type="checkbox"/>	-unit-by-unit
+ Generate a main	<input type="checkbox"/>	-main-generator
- Stubbing		
... Variable/function range setup		... -data-range-specifications
... Stub complex functions	<input type="checkbox"/>	-permissive-stubber
... Functions to stub	function_1,function_2	... -functions-to-stub
... No automatic stubbing	<input checked="" type="checkbox"/>	-no-automatic-stubbing

- 3 Select **Functions to stub**, then click the browse button .

The Functions to stub dialog box opens.



- 4 Enter the name of the function you want to stub, then click .

The function is added to the list of functions to stub.

5 Click **OK**.

Note You can also enter a comma-separated list of functions directly in the Configuration pane.

For example, `function_1,function_2`.

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	<code>test</code>
Class method	<code>A::test</code>
Template method	<code>A<T>::test</code>

- 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 All overloaded versions of the function will be discarded.

Constraining Data with Stubbing

- “Adding Precision Constraints Using Stubs” on page 5-13
- “Default Behavior of Global Data” on page 5-14
- “Constraining the Data” on page 5-15
- “Applying the Technique” on page 5-15
- “Integer Example” on page 5-16
- “Recoding Specific Functions” on page 5-16

Adding 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.

This default launching mode is often adequate, but it is sometimes useful to specify that the range of values which may be assigned to some variables is to be limited to what is appropriate on a functional level. These ranges will be propagated to the whole call tree, and hence will limit the number of “impossible values” which 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. However, it is not always fruitful and it is therefore recommended only where its application is not too labor intensive - that is, where its implementation can be automated.

The technique therefore 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.

Applying the Technique

To 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 all relevant variables into a single procedure
- 3** Call this procedure at the beginning of the main. This should replace any 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();
        // ...
    }
}
```

Recoding 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 (PURE and WORST)

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>
pragma POLYSPACE_WORST	<pre>int f(char *x) { strcpy(x, "the quick brown fox, etc."); return &(x[2]); }</pre>
pragma POLYSPACE_PURE	<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 <code>(int *) u</code>	Returns $[-2^{31}, 2^{31}-1]$
<code>int f3 (int *u);</code>			Assumes the ability to write into <code>*u</code> to any depth and returns $[-2^{31}, 2^{31}-1]$
<code>int* f4 (int u);</code>	Returns an absolute address (AA)	Returns AA or <code>(int *) u</code> and assumes the ability to write into <code>(int *) u</code>	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 <code>*u</code> , to any depth	Assumes the ability to write into <code>*u</code> , 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 <code>(int *)</code> associated with the function pointer is too complicated for the software to stub it, and verification stops. You must stub this function manually.	
		<p>Note If <code>(*ptr)</code> 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.</p>	

Function Pointer Cases

Function Prototype	Comments
<pre>int f(void (*ptr_ok)(int, char, float), other_type1 other_param1);</pre>	The -permissive-stubber option is not required.
<pre>int f(void (*ptr_ok)(int *, char, float), other_type1 other_param1);</pre>	The -permissive-stubber option is required because of the “int *” parameter of the function pointer passed as an argument.
<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>

Stubbing Functions with a 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-c -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.

Stubbing 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 Argument in Standard Library Function Call: `STD_LIB`”.

For more information about how you can use `__polyspace__stdstubs.c`, see “Standard Library Function Stubbing Errors” on page 7-40.

Preparing Code for Variables

In this section...

“Checking Variable Ranges with Assert” on page 5-24

“Checking Properties on Global Variables: Global Assert” on page 5-25

“Modeling Variable Values External to Your Application” on page 5-25

“Initializing Variables” on page 5-26

“Data and Coding Rules” on page 5-27

“Verifying Code with Undefined or Undeclared Variables and Functions” on page 5-28

Checking Variable Ranges with Assert

Assert is a UNIX, Linux, and Windows macro that aborts the program if the test performed inside the assertion proves to be false.

Assert failures are real RTEs because they lead to a processor halt. Because of this, the verification produces checks for the assert failures. The behavior matches the behavior exhibited during execution, because **all execution paths for unsatisfied conditions are truncated** (red and then gray). You can assume that any verification performed downstream of the assert uses value ranges which satisfy the assert conditions.

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;
}
```

Checking Properties on Global Variables: Global Assert

The global assert mechanism works by inserting a check on each write access to a global variable to ensure that it is the range specified.

You enable this feature using DRS `globalassert` mode.

For more information, see “Specifying Data Ranges for Variables and Functions (Contextual Verification)” on page 4-56.

Modeling Variable Values External to Your Application

There are three main considerations:

- Use of volatile variable.
- Express that the variable content can change at every new read access.
- Express that some variables are external to the application.

A volatile variable is a variable which does not respect the following axiom:

"If I write a value V in the variable X, and if I read X's value before any other writing to X occurs, I will get V."

The value of a volatile variable is "unknown". It can be any value that can be represented by a variable of its type, and that value can change at any time; even between two successive memory accesses.

A volatile variable is viewed as a "permanent random" by Polyspace verification because the value may have changed between one read access and the next.

Note Although the volatile characteristic of a variable is also commonly used by programmers to avoid compiler optimization, this characteristic has no consequence for Polyspace verification.

```
int return_random(void)
{
    volatile int random; // random ~ [-2^31, 2^31-1], although
                        // random is not initialized
    int y;
    y = 1 / random;    // division and init orange because
                      // random ~ [-2^31, 2^31-1]
    random = 100;
    y = 1 / random;    // division and init orange because
                      // random ~ [-2^31, 2^31-1]
    return random;    // random ~ [-2^31, 2^31-1]
}
```

Initializing Variables

Consider external, volatile, and absolute address variables in the following examples.

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 “Before You Review Polyspace Results” on page 8-2.

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 Addressing

The content of an absolute address is always considered to be potentially uninitialized (NIV orange):

```
int y;

void f1(void) {

#define X (* ((int *)0x20000))
  X = 100;
  y = 1 / X;    // NIV on X is orange
}

void f2(void) {
  int *p = (int *)0x20000;
  *p = 100;
  y = 1 / *p;  // NIV on *p is orange
}
```

Data and Coding 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 resulting problems in a design, such as

- File APIs (or function accessible from outside the file) with no 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.

Verifying Code with Undefined or Undeclared Variables and Functions

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, you must specify the option **Continue even with undefined global variables** (`-allow-undef-variable`) before you start a verification. When you specify this option, Polyspace software considers the variable to be initialized, and to potentially have any value in its full range (see “Initializing Variables” on page 5-26).

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.

Preparing Code for Built-In Functions

In this section...
“Overview ” on page 5-30
“Stubs of stl Functions” on page 5-30
“Stubs of libc Functions” on page 5-30

Overview

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

Stubs of stl Functions

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

Note All 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

All the functions are declared in the standard list of headers, and can be redefined using their own definitions by invalidating the associated set of functions:

- Using `D POLYSPACE_NO_STANDARD_STUBS` for all 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>/cinclude/__polyspace__stdstubs.c`), `signal.h` ('`signal`' and '`raise`' functions are partially implemented. For more

information, see `<polyspace>/cinclude/__polyspace__stdstubs.c`), `stdio.h`, `stdarg.h`, `stdlib.h`, `string.h`, and `time.h`.

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

These functions can be redefined and analyzed by invalidating the associated set of functions or only the specific function using `D __polyspace_no_<function name>`. For example, If you want to redefine the `fabs()` function, add the `D __polyspace_no_fabs` directive and add the code of your own `fabs()` function in a Polyspace verification.

There are five exceptions to the preceding rules. The following functions which deal with memory allocation can not be redefined: `malloc()`, `calloc()`, `realloc()`, `valloc()`, `alloca()`, `__built_in_malloc()` and `__built_in_alloca()`.

Preparing Multitasking Code

In this section...
“Polyspace Software Assumptions” on page 5-32
“Modelling Synchronous Tasks” on page 5-33
“Modelling Interruptions and Asynchronous Events, Tasks, and Threads” on page 5-35
“Are Interruptions Maskable or Preemptive by Default?” on page 5-37
“Shared Variables” on page 5-39
“Mailboxes” on page 5-42
“Atomicity (Can an Instruction Be Interrupted by Another?)” on page 5-44
“Priorities” on page 5-46

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.
- All tasks or entry-points start after the end of the main procedure without any 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.

MathWorks recommends that you read this entire section before applying the rules described. Some rules are mandatory while other rules allow you to gain selectivity.

Modelling 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 never interrupt each other. They include no infinite loops, and always 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 — that is, that they are functions to be called in a deterministic order. You can do this by writing a function to call each of the tasks in the correct 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-c -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-c -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.

Modelling Interruptions and Asynchronous Events, Tasks, and Threads

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 Run-Time Checks perspective. The gray code indicates that this code is not executed and is not taken into account, so all 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 all potential starting sequences being modelled automatically. You can adopt several different approaches to implement the required adaptations.

Solution 1: Where Interrupts (ISRs) Cannot Ppreempt 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-c -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 a 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-c -entry-points polys_it_1,polys_it_2,polys_one_task
```

Are Interruptions Maskable or Preemptive by Default?

For user interruptions, no *implicit* critical section is defined: you must write all of 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). Because Polyspace verification makes an **upper approximation of all scheduling and all**

interleaving, in this case, that **includes the possibility that the ISR might be interrupted by any other task**. 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-c \
-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 any options, all tasks are examined as though concurrent and with no 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 5-39
- “Mutual Exclusion” on page 5-41
- “Semaphores” on page 5-42

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, all 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-c \
-entry-point proc1,proc2 \
-critical-section-begin"begin_cs:label_1" \
```

```
-critical-section-end"end_cs:label1_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-c -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.

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 contains the full range of values appropriate to that 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. That means that even if the sender is only submitting data within a small range, the full data range appropriate for the type or types are for the receiver data.

Provide a **real mailbox** mechanism

- Costly (time consuming) to implement.
- Can introduce errors in the stubs.

Provide an **upper approximation of the mailbox**

- Provides little additional benefit when compared to the upper approximation solution.

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 an Instruction Be Interrupted by Another?)

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 none are.

Instructional decomposition

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

Polyspace verification assumes that instructions are never 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 always 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 5-39).

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. All other functions can still continue to run, even if somewhere else in another task a critical section has been started.

Polyspace verification of Runtime Errors (RTEs) supposes that there was no conflict when writing the shared variables. If a shared variable is not protected, the RTE verification is complete and correct.

More information is available in “Critical Sections” on page 5-39.

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.

While there is no capability to specify differing task priorities, all 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 in any order, no matter the sequence of instructions. Therefore, all possible interruptions are accounted for, in addition to some interruptions which do not actually occur.

If you have two tasks, `t1` and `t2`, in which `t1` has higher priority than `t2`, use `polyspace-c -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 all interleaving**. There are more paths modelled than could happen during execution, but this has no adverse effect on the results.

Highlighting Known Coding Rule Violations and Run-Time Errors

In this section...

“Annotating Code to Indicate Known Coding Rule Violations” on page 5-47

“Annotating Code to Indicate Known Run-Time Errors” on page 5-51

Annotating Code to Indicate Known Coding Rule Violations

You can place comments in your code that inform Polyspace software of known or acceptable coding rule violations. The software uses the comments to highlight, in the Coding Rules perspective, errors or warnings related to the coding rule violations. Using this functionality, you can:

- Hide known coding rule violations while analyzing new coding rule violations.
- Inform other users of known coding rule violations.

Note Source code annotations do not apply to code comments. Therefore, the following coding rules cannot be annotated:

- MISRA-C Rules 2.2 and 2.3
- MISRA-C++ Rule 2-7-1
- JSF++ Rules 127 and 133

To annotate your code before running a verification:

- 1** Open your source file using a text editor.
- 2** Locate the code that violates a coding rule.
- 3** Insert the required comment.

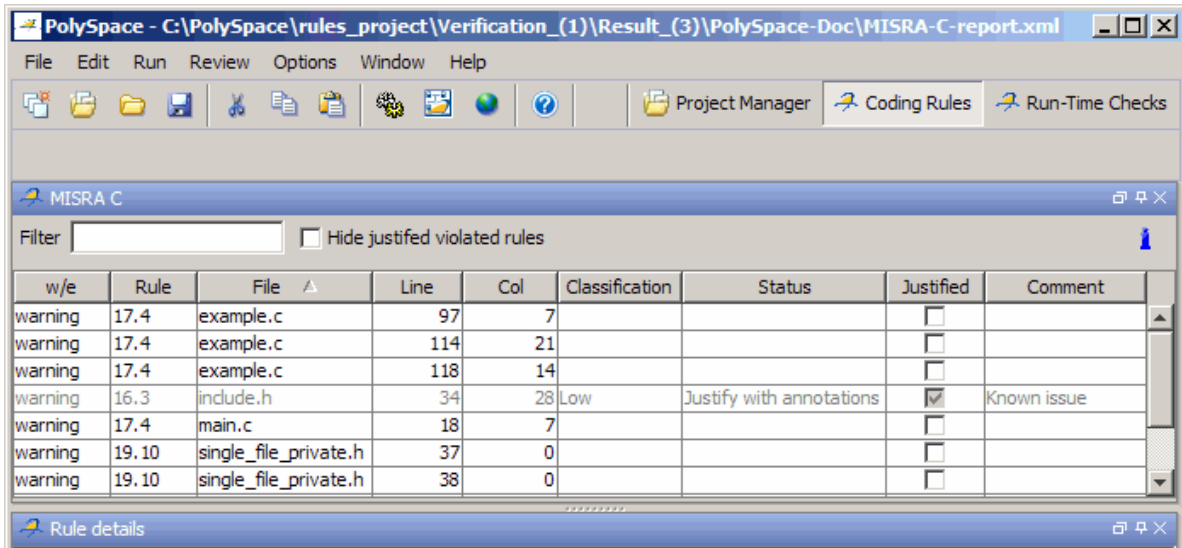
```
int    random_int ^ (void);
float  random_float(void);
extern void partial_init(int *new_alt);
extern void RTE(void);
/* polyspace<MISRA-C:16.3: Low : Justify with annotations > Known issue */
extern void Exec_One_Cycle (int);
extern int orderregulate (void);
extern void Begin_CS (void);
```

See also “Syntax for Coding Rule Violations” on page 5-49 .

Note Instead of typing the full syntax of the annotation, you can copy an annotation template from the Coding Rules perspective, paste it into your source code, and modify the template to comment the violation. To copy the annotation template, right-click any violation in the Coding Rules perspective and select **Add Pre-Justification to Clipboard**.

- 4 Save your file.
- 5 Start the verification. The software produces a warning if your comments do not conform to the prescribed syntax, and they do not appear in the Coding Rules perspective.

When the verification is complete, or stops because of a compilation error, you can view all coding rule violations in the Coding Rules perspective.



In the **Classification**, **Status** and **Comment** columns, the information that you provide within your code comments is now visible. In addition, in the **Justified** column, the check box is selected.

To hide coding rule violations that you annotate, select the **Hide justified violated rules** check box.

Syntax for Coding Rule Violations

To apply comments to a single line of code, use the following syntax:

```
/* polyspace<Rule_Set:Rule1[,Rule2] : [Classification] :
[Status] >
[Additional text] */
```

where

- Square brackets `[]` indicate optional information.
- `Rule_Set` is, depending on your rule checker, either MISRA-C, MISRA-CPP or JSF.

- *Rule1, Rule2*, are rules (for example, 10.3, 11.5), that are defined in your rules file (for example, `misra-rules.msr`). You can also specify `ALL`, which covers every coding rule.
- *Classification*, for example, `High` and `Low`, allows you to categorize the seriousness of the coding rule violation with a predefined classification. To see the complete list of predefined classifications, in the Preferences dialog box, click the **Review Statuses** tab.
- *Status* allows you to categorize the coding rule violation with either a predefined status, or a status that you define in the Preferences dialog box, through the **Review Statuses** tab.
- *Additional text* appears in the **Comment** column of the Coding Rules perspective. Use this text to provide information about the coding rule violations.

The software applies the comments, which are case-insensitive, to the first non-commented line of C code that follows the annotation.

Note The software does not process code annotations that occupy several lines through the use of the C++ line continuation character `\`. For example,

```
// polyspace<JSF: 11 > This comment starts on \  
one line but finishes on another.
```

Syntax Examples:

MISRA C rule violation:

```
/* polyspace<MISRA-C:6.3 : Low : Justify with annotations> Known issue */
```

JSF++ rule violation:

```
/* polyspace<JSF:9 : Low : Justify with annotations> Known issue */
```

Note Instead of typing the full syntax of the annotation, you can copy an annotation template from the Run-Time Checks perspective, paste it into your source code, and modify the template to comment the check. To copy the annotation template, right-click any check in the Source pane and select **Add Pre-Justification to Clipboard**.

Syntax for Sections of Code

To apply comments to a section of code, use the following syntax:

```
/* polyspace:begin<MISRA-C:Rule1[,Rule2] :  
[Classification] : [Status] >  
[Additional text] */  
  
... Code section ...  
  
/* polyspace:end<MISRA-C:Rule1[,Rule2] : [Classification] : [Status] >  
[Additional text] */
```

Annotating Code to Indicate Known Run-Time Errors

You can place comments in your code that inform Polyspace software of known run-time errors. Through the use of these comments, you can:

- Highlight run-time errors:
 - Identified in previous verifications.
 - That are not significant.
- Categorize previously reviewed run-time errors.

Therefore, during your analysis of verification results, you can disregard these known errors and focus on new errors.

Annotate your code before running a verification:

- 1 Open your source file using a text editor.
- 2 Locate the code that produces a run-time error.

3 Insert the required comment.

```

if (random_int() > 0)
{
    /* polyspace<RTE: NTC : Low : No Action Planned > This run-time error was discovered previously */
    Square_Root();
}

Unreachable_Code();

```

See also “Syntax for Run-Time Errors” on page 5-53.

Note Instead of typing the full syntax of the annotation, you can copy an annotation template from the Run-Time Checks perspective, paste it into your source code, and modify the template to comment the check. To copy the annotation template, right-click any check in the Source pane and select **Add Pre-Justification to Clipboard**.

4 Save your file.

5 Start the verification. The software produces a warning if your comments do not conform to the prescribed syntax, and they do not appear in the Run-Time Checks perspective.

When the verification is complete, open the Run-Time Checks perspective. You see run-time errors in the procedural entities view.

In the **Classification**, **Status**, and **Comment** columns, the information that you provide within your code comments is now visible. In addition, in the **Justified** column, the check box is selected.

Procedural entities	!	X	?	✓	Line	%	Justified	Comment	Classification	Status
example_project	5	04	8	99	95		<input type="checkbox"/>			
example.c	4	8	8	83	1	92	<input type="checkbox"/>			
Close_To_Zero ()			3	10	37	77	<input type="checkbox"/>			
Non_Infinite_Loop ()				11	68	100	<input type="checkbox"/>			
Pointer_Arithmetic ()	1	3	1	19	89	96	<input type="checkbox"/>			
RTE ()				3	222	100	<input type="checkbox"/>			
IRV.0				✓	1	229	<input type="checkbox"/>			
IRV.1				✓	1	231	<input type="checkbox"/>			
IRV.2				✓	1	238	<input type="checkbox"/>			
NTC.3				!	1	241	<input checked="" type="checkbox"/>	This run-time error was discovered previously	Low	No Action Planned
Recursion ()				1	14	137	<input type="checkbox"/>			

Syntax for Run-Time Errors

To apply comments to a single line of code, use the following syntax:

```
/* polyspace<RTE:RTE1[,RTE2] : [Classification] : [Status] >
   [Additional text] */
```

where,

- Square brackets *[]* indicate optional information.
- *RTE1, RTE2, ...* are formal Polyspace checks, for example, OBAI, IDP, and ZDV. You can also specify ALL, which covers every check.
- *Classification*, for example, High and Low, allows you to categorize the seriousness of the issue with a predefined classification. To see the complete list of predefined classifications, in the Preferences dialog box, click the **Review Statuses** tab.
- *Status* allows you to categorize the run-time error with either a predefined status, or a status that you define in the Preferences dialog box, through the **Review Statuses** tab.
- *Additional text* appears in the **Comment** column of the procedural entities view of the Run-Time Checks perspective. Use this text to provide information about the run-time errors.

The software applies the comments, which are case-insensitive, to the first non-commented line of C code that follows the annotation.

Note The software does not process code annotations that occupy several lines through the use of the C++ line continuation character `\`. For example,

```
// polyspace<RTE: OBAI > This comment starts on \
   one line but finishes on another.
```

Syntax Example:

```
/* polyspace<RTE: NTC : Low : No Action Planned > Known issue */
```

Note Instead of typing the full syntax of the annotation, you can copy an annotation template from the Run-Time Checks perspective, paste it into your source code, and modify the template to comment the check. To copy the annotation template, right-click any check in the Source pane and select **Add Pre-Justification to Clipboard**.

Syntax for Sections of Code

To apply comments to a section of code, use the following syntax:

```
/* polyspace:begin<RTE:RTE1[,RTE2] :  
[Classification] : [Status] >  
[Additional text] */  
  
... Code section ...  
  
/* polyspace:end<RTE:RTE1[,RTE2] : [Classification] : [Status] >  
[Additional text] */
```

Types Promotion

In this section...

“Unsigned Integers Promoted to Signed Integers” on page 5-55

“Promotions Rules in Operators” on page 5-56

“Example” on page 5-56

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 any 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 there is no possibility of any overflow (OVFL) on the +operator.

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 the 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.

Verifying “Unsupported” Code

In this section...
“Ignoring Assembly Code” on page 5-58
“Dealing with Backward “goto” Statements” on page 5-66

Ignoring Assembly Code

You can ignore assembly code during verification using the **Discard assembly code** option (`-discard-asm`). Using this option allows you to work with many instances of assembly code within a C application, but it is not always a valid route to take.

Ignored assembly instructions 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, refer to “Stubbing” on page 5-2, which applies to functions as well as to stubbed instructions.

Polyspace verification is designed for C code only. In most cases, the option `-discard-asm` combined with `-asm-begin` and `-asm-end` can be used to instruct the verification to discard a number of assembly code constructs.

- “Example: Ignore All Statements; the Rest of the Function Remains Unchanged” on page 5-59
- “Example: Automatic Stubbing” on page 5-61
- “Examples: Empty Body” on page 5-62
- “Example: #asm and #endasm Support” on page 5-63
- “Example: What to Do If `-discard-asm` Fails to Parse an asm Code Section” on page 5-64

Example: Ignore All Statements; the Rest of the Function Remains Unchanged

Discarding assembly code by using the `-discard-asm` is an acceptable approach where ignoring the assembly instructions will have no impact on the remainder of the function.

For more information, see the “Manual versus automatic stubbing”.

```
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
}

```

Example: Automatic Stubbing

When a function is preceded by `asm`, it is stubbed automatically, even if a body is defined.

```
asm int m(int tt);
```

You must use the `-discard-asm` option.

Examples: Empty Body

Using the option, `#pragma inline_asm(List of functions)`, has the same effect.

You must use the `-discard-asm` option.

```
pragma inline_asm(ex1, ex2) // the 2 functions ex1 and ex2 will be
                             // stubbed, even if their body is defined

int ex1(void)
{
    % reg val;
    mtmsr val;
    return 3;    // is ignored
};

int ex2(void)
{
    % reg val;
    mtmsr val;
    assert (1); // is ignored
    return 3;
};

#pragma inline_asm(ex3) // the definition of ex3 is ignored

int ex3(void)
{
    % reg val;
    mtmsr val;    // is ignored
    return 3;
};

asm int h(int tt) // using the qualifier asm is equivalent
                  // to #pragma inline_asm
{
    % reg val;    // is ignored
    mtmsr val;    // is ignored
}
```

```

    return 3;    // is ignored
};

void f(void) {
    int x;

    x = ex1();    // ex1 is stubbed : x is full-range
    x = ex2();    // x is full-range
    x = ex3();    // x is full-range
    x = h(3);     // x is full-range
}

```

For more information, see “Stubbing” on page 5-2.

Example: #asm and #endasm Support

Using #asm and #endasm allows fragments of assembly code to be disregarded, regardless of whether or not you use the -discard-asm.

Consider the following example.

```

void test(void)
{
#asm
    mov _as:pe, reg
    jre _nop
#endasm
    int r;
    r=0;
    r++;
}

```

Explanation

By default, using #asm and #endasm requires using the -asm-begin and -asm-end options. The options to enable this feature are accessible through the Project Manager perspective or in batch mode.

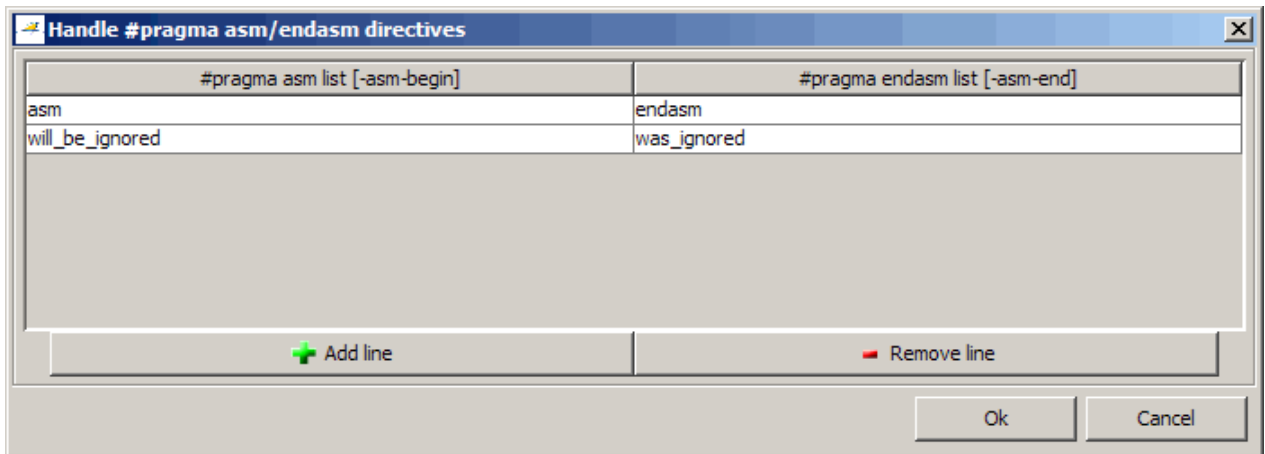
When launching Polyspace verification in batch mode, use this syntax:

```
polyspace-c -asm-begin asm -asm-end endasm
```

To enable this option using the Project Manager perspective:

- 1 In the Configuration pane of the Project Manager perspective, select **Compliance with standards > Embedded assembler**.
- 2 Select **Handle #pragma asm.endasm directives**.

The Handle #pragma asm/endasm directives dialog box opens.



- 3 Select **Add line**.
- 4 In the **#pragma asm list [-asm-begin]** column, enter `asm`.
- 5 In the **#pragma enasm list [-asm-end]** column, enter `endasm`.
- 6 Click **Ok**.

Example: What to Do If `-discard-asm` Fails to Parse an `asm` Code Section

Occasionally, the `-discard-asm` option does not deal with a particular assembly code construction, particularly when the code fragment is compiler-specific.

Note Consider using the `-asm-begin` and `-asm-end` options instead of the following approach.

```
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 }
```

As shown in this example, any text or code placed between the two `#pragma` statements is ignored by the verification. This allows any unsupported construction to be ignored without changing the meaning of the original code.

The options to enable this feature are accessible through the Project Manager perspective or in batch mode.

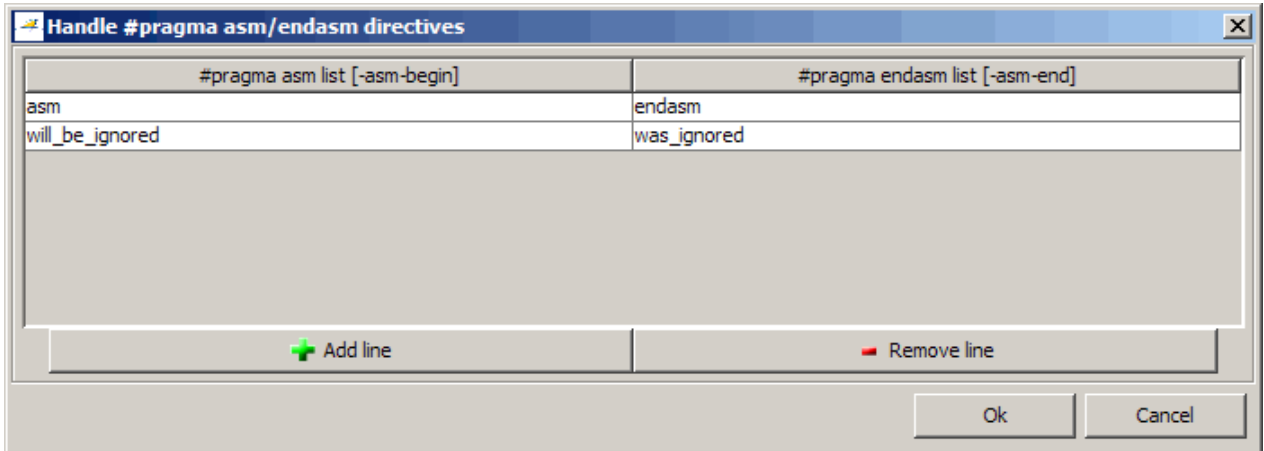
To enable this option in batch mode, enter the following command:

```
polyspace-c -asm-begin will_be_ignored -asm-end was_ignored
```

To enable this option using the Project Manager perspective:

- 1 In the Configuration pane of the Project Manager perspective, select **Compliance with standards > Embedded assembler**.
- 2 Select **Handle #pragma asm.endasm directives**.

The Handle #pragma asm/endasm directives dialog box opens.



- 3 Select **Add line**.
- 4 In the **#pragma asm list [-asm-begin]** column, enter `will_be_ignored`.
- 5 In the **#pragma enasm list [-asm-end]** column, enter `was_ignored`.
- 6 Click **Ok**.

Dealing with Backward “goto” Statements

Polyspace verification is not designed to support backward “goto” statements. However, macros provide a solution. Verifications that includes backward “goto” statements stop at an early stage, and a message appears saying that backward “goto” statements are not supported.

Macros provided with the Polyspace software can work around this limitation **as long as the “goto” labels and jump instructions are in the same code block (and in the same scope)**.

To insert these macros into the code:

- 1** Edit the C file containing the "goto" statements.
- 2** Add `#include pstgoto.h` at the beginning of the file (located in `Polyspace_Install/cinclude`).
- 3** Go to the beginning of the block containing the "goto" statements.
- 4** Insert the `USE_1_GOTO(<tag>)` macro call after the variable declarations (local to the block).
- 5** Insert the `EXIT_1_GOTO(<tag>)` macro call before the end of this same block (take care with the closing bracket `}`).
- 6** Replace `"goto <tag>"` with `"GOTO(<tag>)"`.

For example, the following code would cause a verification to terminate:

```
{
/* local variable declarations */
int x; ...
/* Instructions */
...
label1:
...
goto label1
...
}
```

You could address this problem as follows:

```
/* the pstgoto.h file is provided by Polyspace and its path */
{
/* local variable declarations */
int x; ...
USE_1_GOTO(label1);
/* Instructions */
...
label1:
```

```

...
GOTO(label1);
...
EXIT_1_GOTO(label1);
}

```

The code block may contain many instances of backward “goto” statements. Using matching USE_n_GOTO() and EXIT_n_GOTO() statements addresses this issue,(for example, USE_2_GOTO(), USE_3_GOTO(), etc.)

Note You must copy `pstgoto.h` from `Polyspace_Install/cinclude`, and add it to the list of include folders (-I).

The code block may also use several different tags. You can use multiple “tag” parameters to address these situations. For example, use:

```

USE_n_GOTO (<tag 1>, <tag 2>, ..., <tag n>);
EXIT_n_GOTO(<tag 1>, <tag 2>, ..., <tag n>);

```

Consider the following example.

Original Code	Modified Code for Verification
<pre> { . Reset: . { { if (X) goto Reset; } { if (Y) goto Reset; } </pre>	<pre> { USE_1_GOTO(Reset); Reset: { { if (X) GOTO(Reset); } { if (Y) GOTO(Reset); } </pre>

Original Code	Modified Code for Verification
<pre data-bbox="391 326 436 387"> } }</pre>	<pre data-bbox="869 326 1151 421"> } } EXIT_1_GOTO(Reset);</pre>

Running a Verification

- “Before Running Verification” on page 6-2
- “Running Verifications on Polyspace Server” on page 6-9
- “Running Verifications on Polyspace Client” on page 6-31
- “Running Verifications from Command Line” on page 6-37

Before Running Verification

In this section...
“Types of Verification” on page 6-2
“Specifying Source Files to Verify” on page 6-2
“Specifying Results Folder” on page 6-3
“Specifying Analysis Options Configuration” on page 6-4
“Checking for Compilation Problems” on page 6-5

Types of Verification

You can run a verification on a server or a client.

Use...	For...
Server	<ul style="list-style-type: none"> • Best performance • Large files (more than 800 lines of code including comments)
Client	<ul style="list-style-type: none"> • When the server is busy • Small files <hr/> <p>Note Verification on a client takes more time. You might not be able to use your client computer when a verification is running on it.</p> <hr/>

Specifying Source Files to Verify

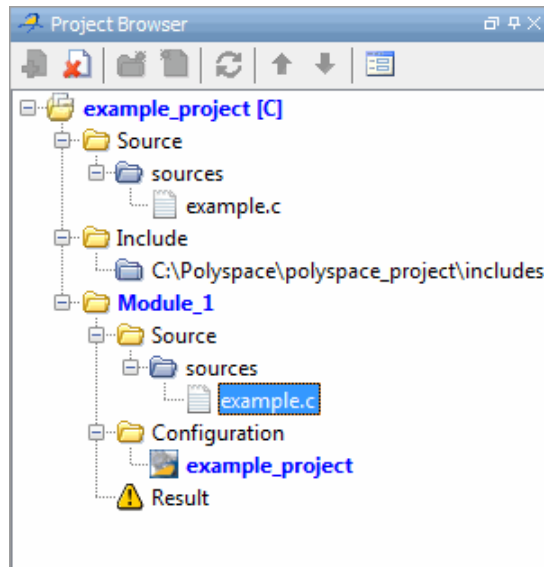
Each Polyspace project can contain multiple modules. Each of these modules verifies a specific set of source files using a specific set of analysis options. Therefore, before you can launch a verification, you must decide which files in your project to verify, and add them to a module.

To copy source files to a module:

- 1 Open the project containing the files you want to verify.

- 2 In the Project Browser Source tree, select the source files you want to verify.
- 3 Right click any selected file, and select **Copy Source File to > Module_(#)**.

The selected source files appear in the Source tree of the module.



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

Specifying Results Folder

Each Module in the Project Browser can contain multiple result folders. This allows you to save results from multiple verifications of the same source files, either to compare results using different analysis options, or to track verification results over time as your source files are revised.

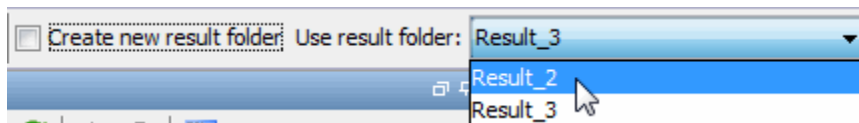
By default, Polyspace software creates a new result folder for each verification. However, if you want to reuse an existing result folder, you can select that folder before launching verification. For example, you may want to reuse

a result folder if you stopped a verification before it completed, and are restarting the same verification.

Caution If you specify an existing result folder, all results in that folder are deleted when you start a new verification.

To specify the result folder for a verification:

- 1 In the Project Browser select the module you want to verify.
- 2 In the Project Manager toolbar, clear the **Create a new result folder** check box.
- 3 In the **Use result folder** drop-down menu, select the folder you want to use.



When you launch verification, the software saves verification results to the selected result folder.

Specifying Analysis Options Configuration

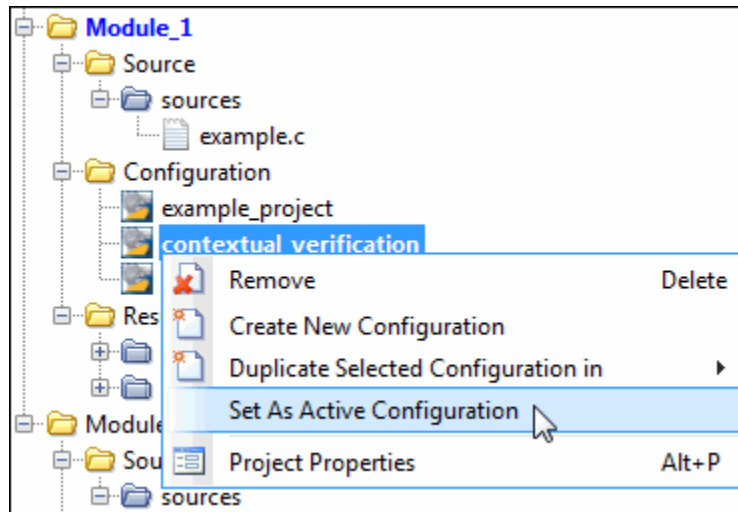
Each Module in the Project Browser can contain multiple configurations, each containing a specific set of analysis options. This allows you to verify the same source files multiple times using different analysis options for each verification.

If you have created multiple configurations, you must choose which configuration to use before launching a verification.

To specify the configuration to use 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 you want to use, and select **Set As Active Configuration**.



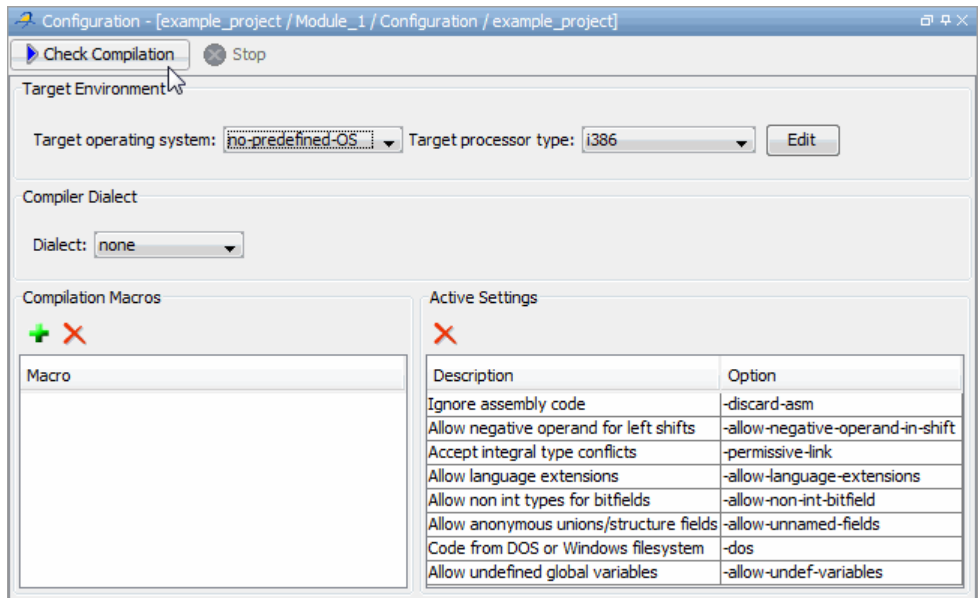
When you launch verification, the software uses the specified analysis options configuration.

Checking for Compilation Problems

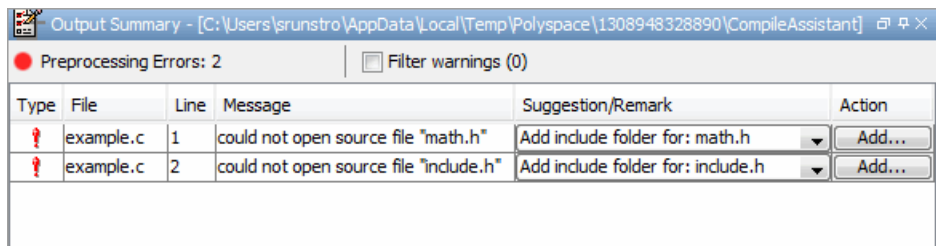
The Compilation Assistant allows you to check your project for compilation problems before launching a verification. When the Compilation Assistant detects an error, it reports the problem and suggests possible solutions.

To check your project for compilation problems:

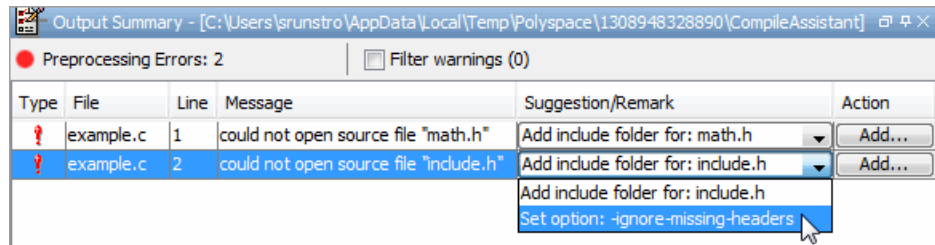
- 1 In the Compilation Assistant pane, click **Check Compilation**.



The software compiles your code and checks for errors, and reports the results in the Output Summary.



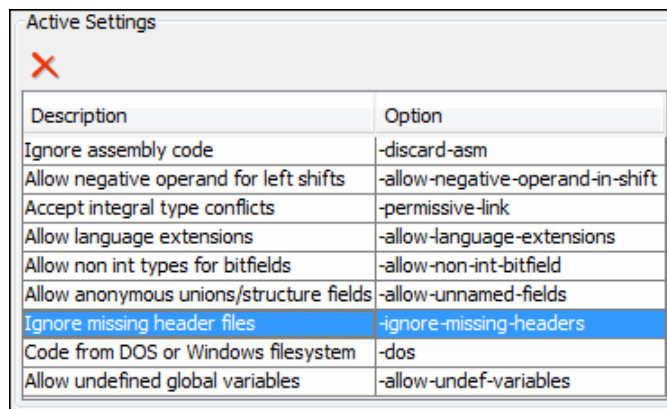
- 2 Select the Suggestion/Remark column to see a list of possible solutions for the problem.



In this example, you can either add the missing include file, or set an option that will attempt to compile the code without the missing include file.

3 Select **Apply** to set the selected option for your project.

The software automatically sets the option, and displays it in the Compilation Assistant Active Settings table.

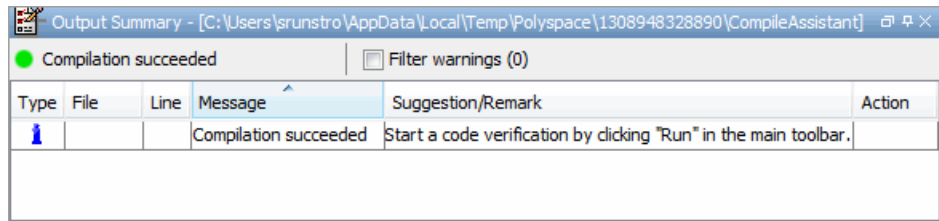


4 Select **Add** to add suggested include folders to your project.

The Add Source Files and Include Folders dialog box opens, allowing you to add additional include folders.

When you have addressed all compilation problems, the Compilation Assistant displays the message `Compilation succeeded` in the Output Summary pane.

6 Running a Verification



Running Verifications on Polyspace Server

In this section...

“Starting Server Verification” on page 6-9

“What Happens When You Run Verification” on page 6-10

“Running Verification Unit-by-Unit” on page 6-11

“Verify All Modules in Project” on page 6-13

“Managing Verification Jobs Using the Polyspace Queue Manager” on page 6-14

“Monitoring Progress of Server Verification” on page 6-16

“Viewing Verification Log File on Server” on page 6-21

“Stopping Server Verification Before It Completes” on page 6-22

“Removing Verification Jobs from Server Before They Run” on page 6-23

“Changing Order of Verification Jobs in Server Queue” on page 6-24

“Purging Server Queue” on page 6-25

“Changing Queue Manager Password” on page 6-27

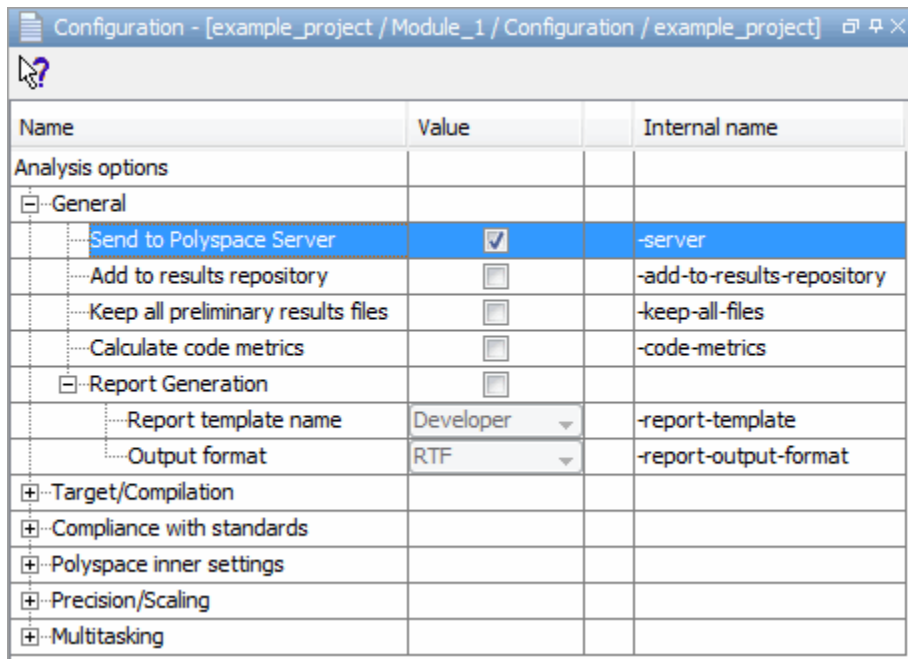
“Sharing Server Verifications Between Users” on page 6-28


Starting Server Verification

Most verification jobs run on the Polyspace server. Running verifications on a server provides optimal performance.

To start a verification that runs on a server:

- 1 In the Project Browser, select the Module you want to verify.
- 2 Select the **Send to Polyspace Server** check box in the General Analysis options.



- 3 Click the **Run** button  on the Project Manager toolbar.

The verification starts. For information on the verification process, see “What Happens When You Run Verification” on page 6-10.

Note If you see the message *Verification process failed*, click **OK** and go to “Verification Process Failed Errors” on page 7-2.

What Happens When You Run Verification

The verification has three main phases:

- 1 Checking syntax and semantics (the compile phase). Because Polyspace software is independent of any particular compiler, it ensures that your code is portable, maintainable, and complies with ANSI® standards.

- 2 Generating a main if the Polyspace software does not find a main and you have selected the **Generate a Main** option. For more information about generating a main, see “Main Generator Behavior for Polyspace Software” (C) or “Generate a main (-main-generator)”(C++) in the *Polyspace Products for C/C++ Reference*.
- 3 Analyzing the code for run-time errors and generating color-coded results.

The compile phase of the verification runs on the client. When the compile phase is complete:

- You see the message `queued on server` at the bottom of the Project Manager perspective. This message indicates that the part of the verification that takes place on the client is complete. The rest of the verification runs on the server.
- A message in the Output Summary window gives you the identification number (Analysis ID) for the verification. For this verification, the identification number is 1.

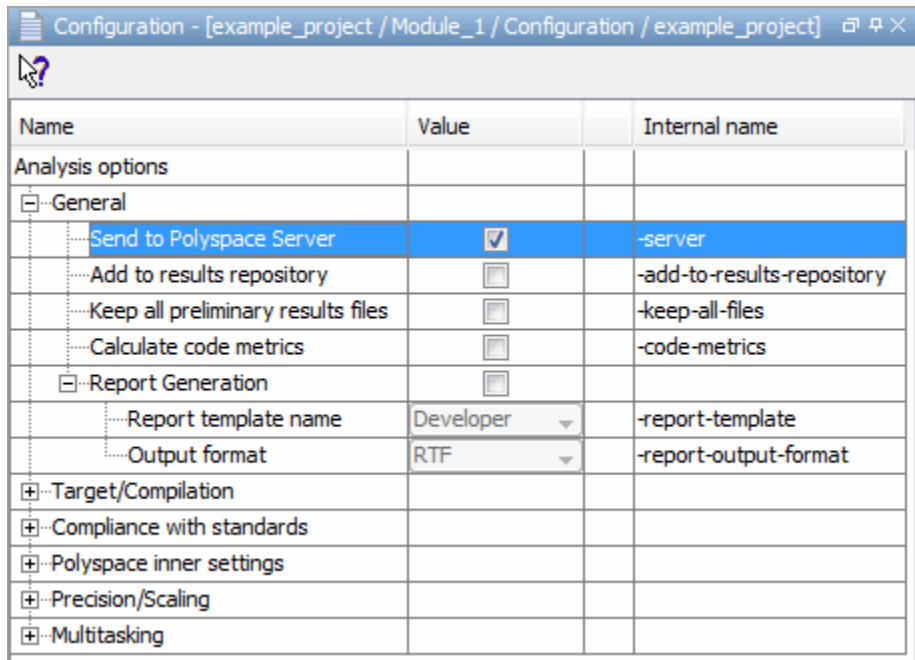
Class	Description	File	Line	Col
	example_project for C verification start at Dec 15, 2010 15:57:15			
	The generated default DRS XML file "drs-template.xml" can be found in <result_dir...			
	Analysis ID : 1			

Running Verification Unit-by-Unit

When launching a server verification, you can create a separate verification jobs for each source file in the project. Each file is compiled, sent to the Polyspace Server, and verified individually. Verification results can then be viewed for the entire project, or for individual units.

To run a unit-by-unit verification:

- 1 In the Project Manager General Analysis options, select the **Send to Polyspace Server** check box




2 In the Analysis options, expand **Polyspace inner settings**.

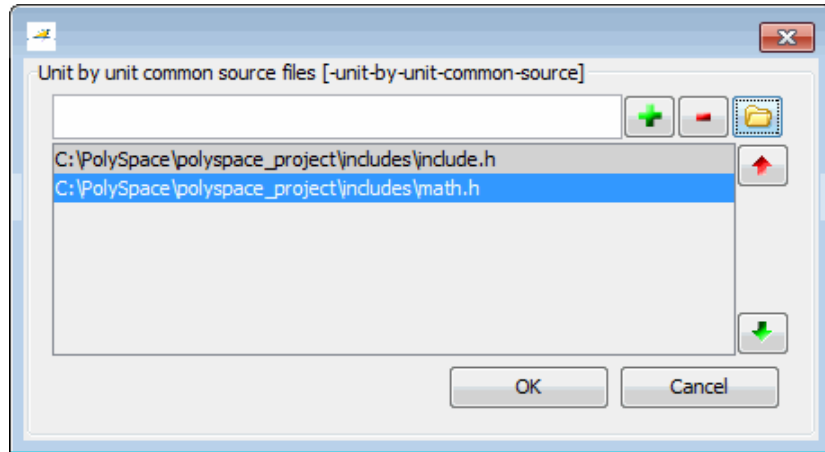
3 Select the **Run a verification unit by unit** check box.


[-] PolySpace inner settings		
[-] Run a verification unit by unit	<input checked="" type="checkbox"/>	-unit-by-unit
Unit by unit common source	C:\PolySpace\poly ...	-unit-by-unit-common-source

4 Expand the **Run a verification unit by unit** item.

5 Click the button  to the right of the **Unit by unit common source** option.

The Unit by unit common source dialog box opens.



- 6** Click the folder icon .

The **Select a file to include** dialog box appears.

- 7** Select the common files to include with each unit verification.

These files are compiled once, and then linked to each unit before verification. Functions not included in this list are stubbed.

- 8** Click **Ok**.

- 9** Click **Run**.

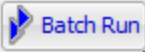
Each file in the project is compiled, sent to the Polyspace Server, and verified individually as part of a verification group for the project.

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. You can verify all modules in your project using the batch run option.

To verify all modules in a project:

- 1** In the Project Browser, select the project for which you want to run verifications.

- 2 Click the **Batch Run** button  on the Project Manager toolbar.

Each module is verified as an individual job. For information on the verification process, see “What Happens When You Run Verification” on page 6-10.

Note If you see the message `Verification process failed`, click **OK** and go to “Verification Process Failed Errors” on page 7-2.

Managing Verification Jobs Using the Polyspace Queue Manager

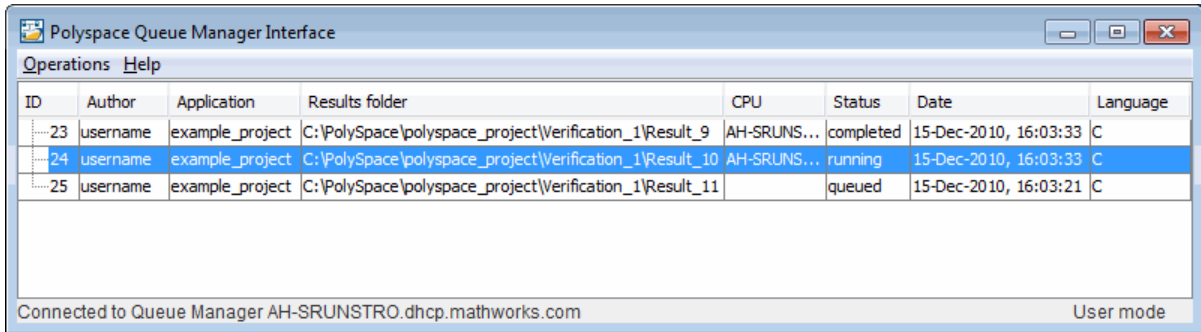
You manage all server verifications using the Polyspace Queue Manager (also called the Polyspace Spooler). The Polyspace Queue Manager allows you to move jobs within the queue, remove jobs, monitor the progress of individual verifications, and download results.

To manage verification jobs on the Polyspace Server:

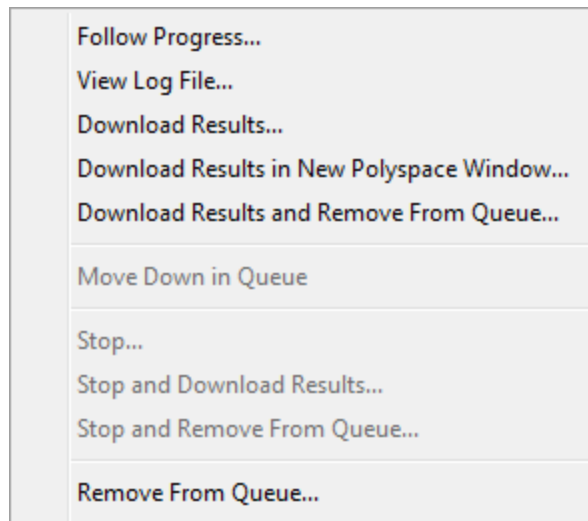
- 1 Double-click the **Polyspace Spooler** icon:



The **Polyspace Queue Manager Interface** opens.



- 2 Right-click any job in the queue to open the context menu for that verification.



- 3 Select the appropriate option from the context menu.

Tip You can also open the Polyspace Queue Manager Interface by clicking the Polyspace Queue Manager icon  in the Polyspace Verification Environment toolbar.

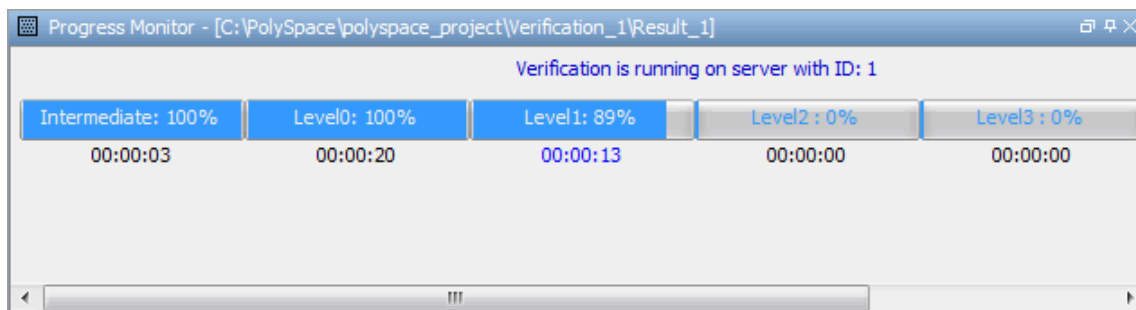
Monitoring Progress of Server Verification

There are two ways to monitor the progress of a verification:

- **Using the Project Manager** – allows you to follow the progress of the verifications you submitted to the server, as well as client verifications.
- **Using the Queue Manager (Spooler)** – allows you to follow the progress of any verification job in the server queue.

Monitoring Progress Using Project Manager

You can monitor the progress of your verification by viewing the progress monitor and logs at the bottom of the Project Manager perspective.



The progress monitor highlights the current phase in blue and displays the amount of time and completion percentage for that phase.

The logs report additional information about the progress of the verification. To view a log, click the button for that log. The information appears in the log display area at the bottom of the Project Manager window. Follow the next steps to view the logs:

- 1 Click the **Output Summary** tab to display compile phase messages and errors. You can search the log by entering search terms in the **Search in the log** box and clicking the left arrows to search backward or the right arrows to search forward.
- 2 Click the **Verification Statistics** tab to display statistics, such as analysis options, stubbed functions, and the verification checks performed.

- 3 Click the **Refresh** button  to update the display as the verification progresses.
- 4 Click the **Full Log** tab to display messages, errors, and statistics for all phases of the verification.

Note You can search the logs. In the **Search in the log** box, enter a search term and click the left arrows to search backward or the right arrows to search forward.

Monitoring Progress Using Queue Manager

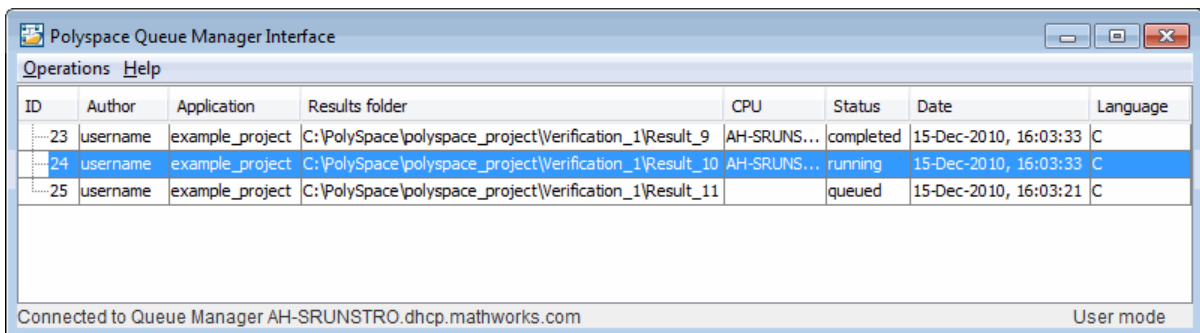
You monitor the progress of the verification using the Polyspace Queue Manager (also called the Spooler).

To monitor the verification of Example_Project:


- 1 Double-click the **Polyspace Spooler** icon on the desktop.



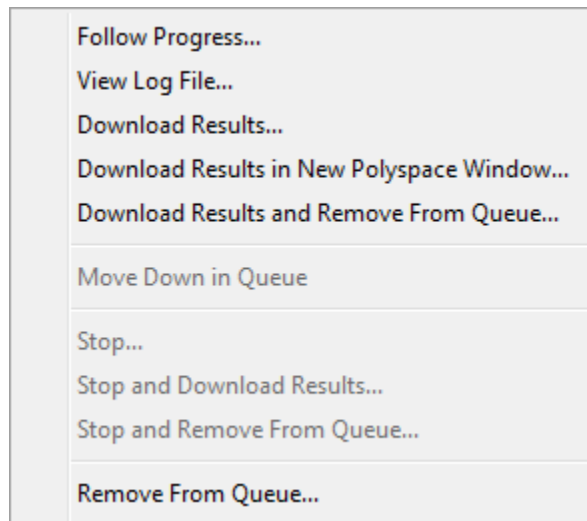
The Polyspace Queue Manager Interface opens.


 The screenshot shows a window titled "Polyspace Queue Manager Interface". The window has a menu bar with "Operations" and "Help". Below the menu bar is a table with the following columns: ID, Author, Application, Results folder, CPU, Status, Date, and Language. The table contains three rows of data. Row 24 is highlighted in blue. At the bottom of the window, there is a status bar that reads "Connected to Queue Manager AH-SRUNSTRO.dhcp.mathworks.com" and "User mode".

ID	Author	Application	Results folder	CPU	Status	Date	Language
23	username	example_project	C:\PolySpace\polyspace_project\Verification_1\Result_9	AH-SRUNS...	completed	15-Dec-2010, 16:03:33	C
24	username	example_project	C:\PolySpace\polyspace_project\Verification_1\Result_10	AH-SRUNS...	running	15-Dec-2010, 16:03:33	C
25	username	example_project	C:\PolySpace\polyspace_project\Verification_1\Result_11		queued	15-Dec-2010, 16:03:21	C

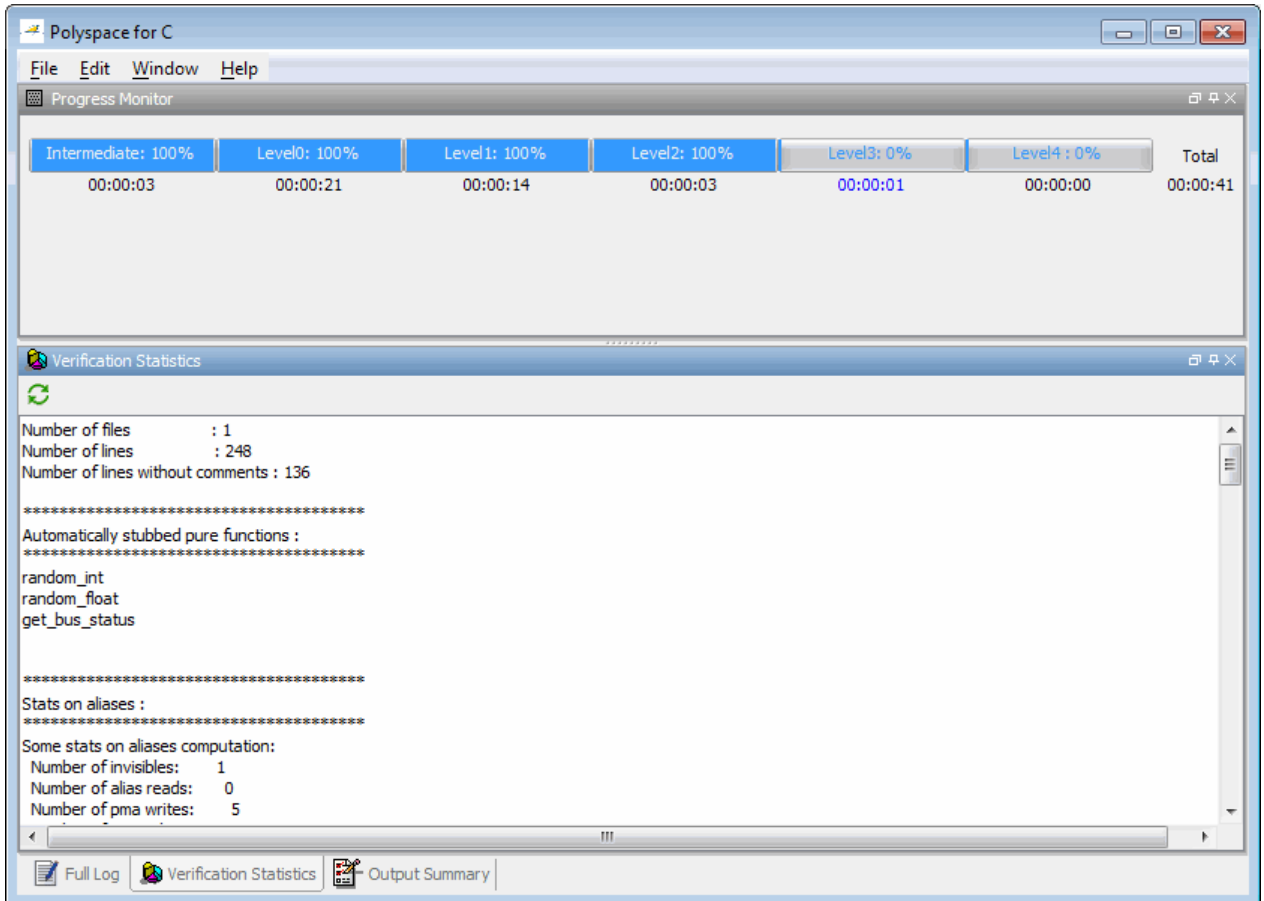
Tip You can also open the Polyspace Queue Manager Interface by clicking the Polyspace Queue Manager icon  on the Run-Time Checks perspective toolbar.

- 2 Point anywhere in the row for ID 1.
- 3 Right-click to open the context menu for this verification.




- 4 Select **Follow Progress** from the context menu.

The Progress Monitor opens.



You can monitor the progress of the verification by watching the progress bar and viewing the logs at the bottom of the window. The progress monitor highlights the current phase in blue and displays the amount of time and completion percentage for that phase.

The logs report additional information about the progress of the verification. To view a log, click the button for that log. The information appears in the log display area at the bottom of the Project Manager window. Follow the next steps to view the logs:

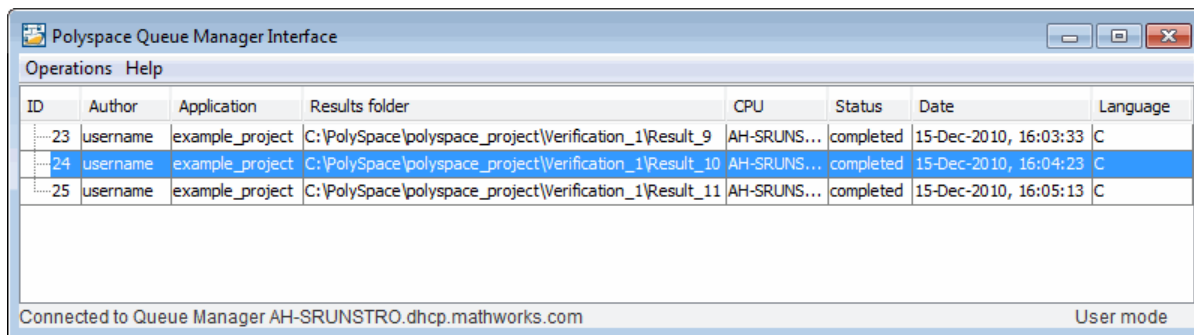
- Click the **Output Summary** tab to display compile phase messages and errors. You can search the log by entering search terms in the **Search in the log** box and clicking the left arrows to search backward or the right arrows to search forward.
- Click the **Verification Statistics** tab to display statistics, such as analysis options, stubbed functions, and the verification checks performed.
- Click the **Refresh** button  to update the display as the verification progresses.
- Click the **Full Log** tab to display messages, errors, and statistics for all phases of the verification.

Note You can search the logs. In the **Search in the log** box, enter a search term and click the left arrows to search backward or the right arrows to search forward.

5 Select **File > Quit** to close the progress window.

6 Wait for the verification to finish.

When the verification is complete, the status in the Polyspace Queue Manager Interface changes from running to completed.



The screenshot shows the Polyspace Queue Manager Interface window. It contains a table with the following data:

ID	Author	Application	Results folder	CPU	Status	Date	Language
23	username	example_project	C:\PolySpace\polyspace_project\Verification_1\Result_9	AH-SRUNS...	completed	15-Dec-2010, 16:03:33	C
24	username	example_project	C:\PolySpace\polyspace_project\Verification_1\Result_10	AH-SRUNS...	completed	15-Dec-2010, 16:04:23	C
25	username	example_project	C:\PolySpace\polyspace_project\Verification_1\Result_11	AH-SRUNS...	completed	15-Dec-2010, 16:05:13	C

At the bottom of the window, it says "Connected to Queue Manager AH-SRUNSTRO.dhcp.mathworks.com" and "User mode".

Viewing Verification Log File on Server

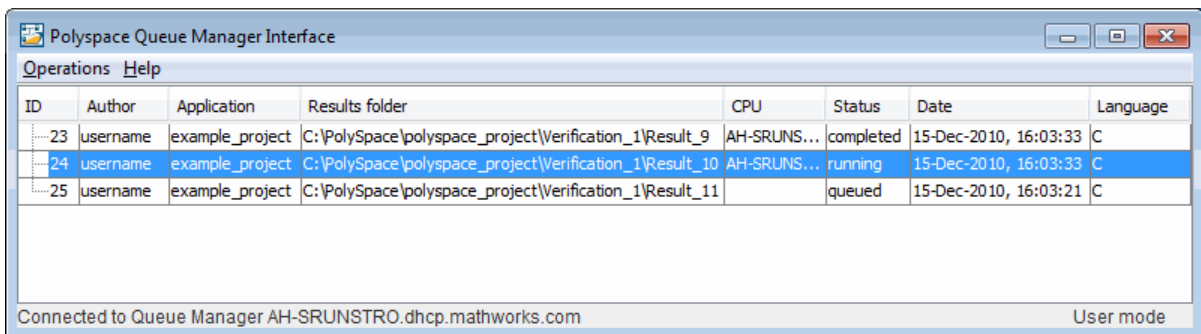
You can view the log file of a server verification using the Polyspace Queue Manager.

To view a log file on the server:

- 1 Double-click the **Polyspace Spooler** icon:

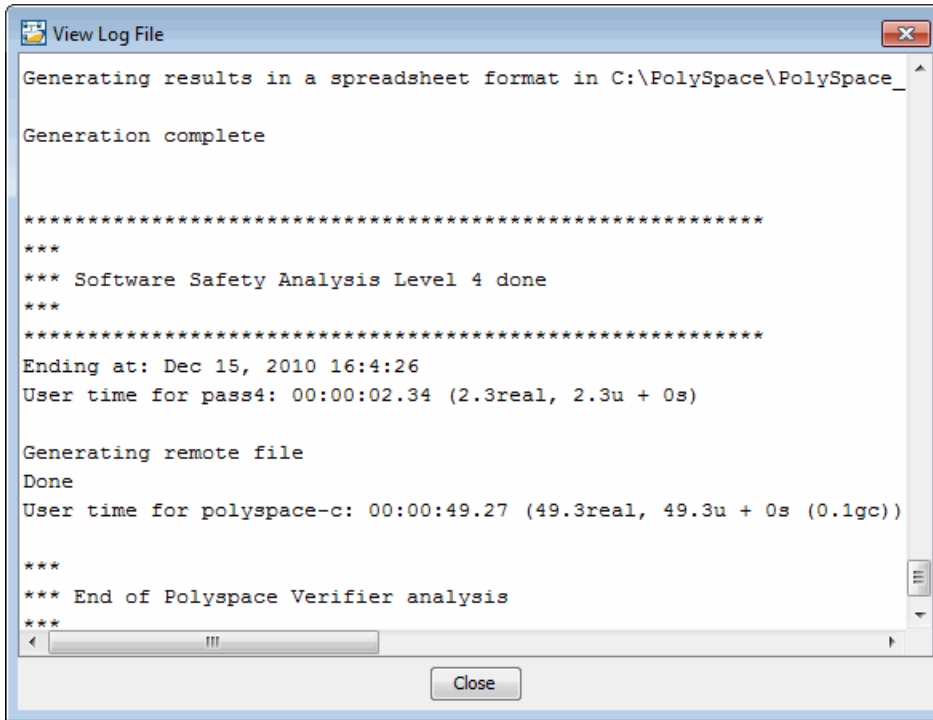


The **Polyspace Queue Manager Interface** opens.



- 2 Right-click the job you want to monitor, and select **View log file**.

A window opens displaying the last one-hundred lines of the verification.



3 Click **Close** to close the window.

Stopping Server Verification Before It Completes

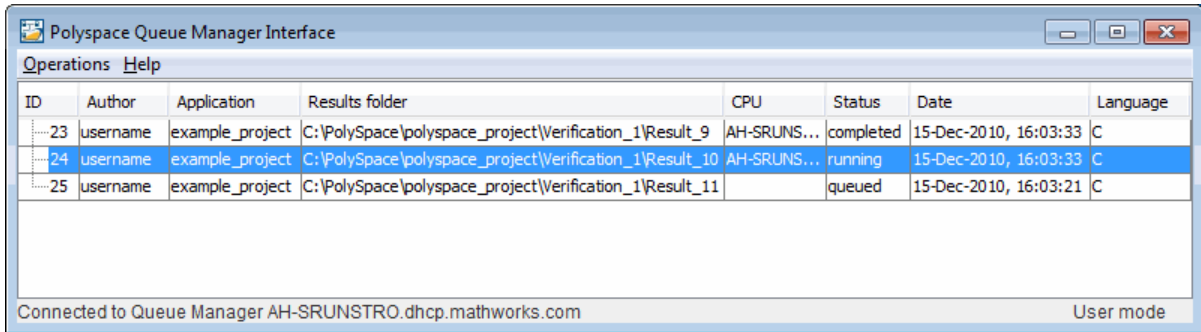
You can stop a verification running on the server before it completes using the Polyspace Queue Manager. If you stop the verification, results will be incomplete, and if you start another verification, the verification starts over from the beginning.

To stop a server verification:

1 Double-click the **Polyspace Spooler** icon:



The **Polyspace Queue Manager Interface** opens.



- 2 Right-click the job you want to monitor, and select one of the following options:

Right-click the job you want to monitor, and select one of the following options:

- **Stop** — Stops a unit-by-unit verification job without removing it. The status of the job changes from “running” to “aborted”. All jobs in the unit-by-unit verification group remain in the queue, and other jobs in the group will continue to run.
- **Stop and download results** — Stops the verification job immediately and downloads any preliminary results. The status of the verification changes from “running” to “aborted”. The verification remains in the queue.
- **Stop and remove from queue** — Stops the verification immediately and removes it from the queue. If the job is part of a unit-by-unit verification group, the entire verification is removed, not just the individual job.

Removing Verification Jobs from Server Before They Run

If your job is in the server queue, but has not yet started running, you can remove it from the queue using the Polyspace Queue Manager.

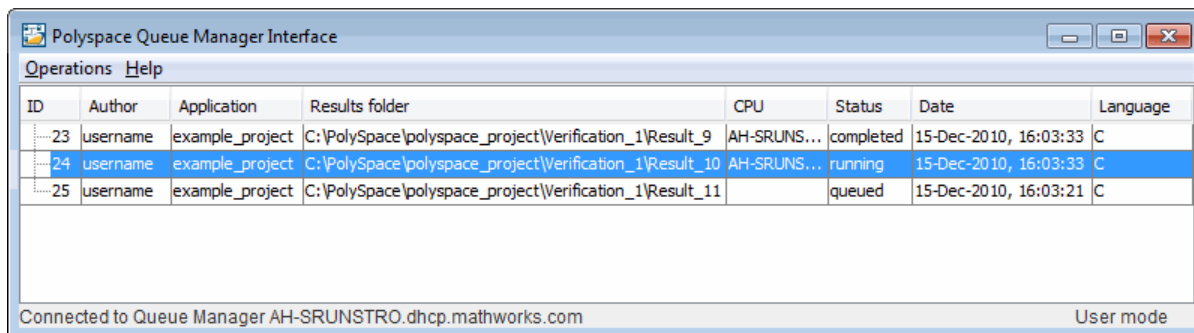
Note If the job has started running, you must stop the verification before you can remove the job (see “Stopping Server Verification Before It Completes” on page 6-22). Once you have aborted a verification, you can remove it from the queue.

To remove a job from the server queue:

- 1 Double-click the **Polyspace Spooler** icon:



The **Polyspace Queue Manager Interface** opens.



- 2 Right-click the job you want to remove, and select **Remove from queue**.

The job is removed from the queue.

Changing Order of Verification Jobs in Server Queue

You can change the priority of verification jobs in the server queue to determine the order in which the jobs run.

To move a job within the server queue:

- 1 Double-click the **Polyspace Spooler** icon:



The **Polyspace Queue Manager Interface** opens.

The screenshot shows a window titled "Polyspace Queue Manager Interface" with a menu bar containing "Operations" and "Help". The main area contains a table with the following data:

ID	Author	Application	Results folder	CPU	Status	Date	Language
23	username	example_project	C:\PolySpace\polyspace_project\Verification_1\Result_9	AH-SRUNS...	completed	15-Dec-2010, 16:03:33	C
24	username	example_project	C:\PolySpace\polyspace_project\Verification_1\Result_10	AH-SRUNS...	running	15-Dec-2010, 16:03:33	C
25	username	example_project	C:\PolySpace\polyspace_project\Verification_1\Result_11		queued	15-Dec-2010, 16:03:21	C

At the bottom of the window, it says "Connected to Queue Manager AH-SRUNSTRO.dhcp.mathworks.com" and "User mode".

- 2 Right-click the job you want to remove, and select **Move down in queue**.

The job is moved down in the queue.

- 3 Repeat this process to reorder the jobs as necessary.

Note You can move unit-by-unit verification groups in the queue, as well as individual jobs within a single unit-by-unit verification group. However, you can not move individual unit-by-unit verification jobs outside of the group.

Purging Server Queue

You can purge the server queue of all jobs, or completed and aborted jobs using the using the Polyspace Queue Manager.

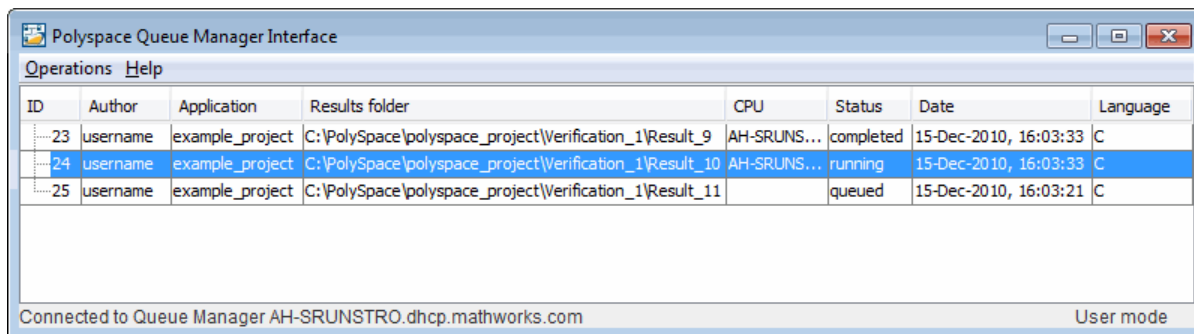
Note You must have the queue manager password to purge the server queue.

To purge the server queue:

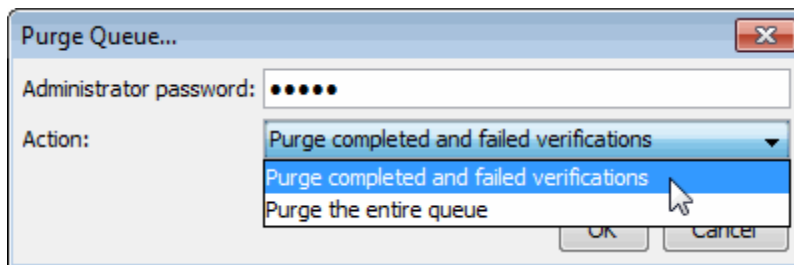
- 1 Double-click the **Polyspace Spooler** icon:



The **Polyspace Queue Manager Interface** opens.



- 2 Select **Operations > Purge queue**. The Purge queue dialog box opens.



- 3 Select one of the following options:
 - **Purge completed and aborted analysis** — Removes all completed and aborted jobs from the server queue.
 - **Purge the entire queue** — Removes all jobs from the server queue.

Note For unit-by-unit verification jobs, no jobs are removed until the entire group has been verified.

4 Enter the Queue Manager **Password**.

5 Click **OK**.

The server queue is purged.

Changing Queue Manager Password

The Queue Manager has an administrator password to control access to advanced operations such as purging the server queue. You can set this password through the Queue Manager.

Note The default password is `admin`.

To set the Queue Manager password:

1 Double-click the **Polyspace Spooler** icon:

The Polyspace Queue Manager Interface opens.

2 Select **Operations > Change Administrator Password**.

The Change Administrator Password dialog box opens.

3 Enter your old password and new passwords, then click **OK**.

The password is changed.

Note Passwords are limited to 8 characters.

Sharing Server Verifications Between Users

Security of Jobs in Server Queue

For security reasons, all verification jobs in the server queue are owned by the user who sent the verification from a specific account. Each verification has a unique encryption key, that is stored in a text file on the client system.

When you manage jobs in the server queue (download, kill, remove, etc.), the Queue Manager checks the public keys stored in this file to authenticate that the job belongs to you.

If the key does not exist, an error message appears: “key for verification <ID> not found”.

analysis-keys.txt File

The public part of the security key is stored in a file named `analysis-keys.txt` associated to a user account. This file is located in `%APPDATA%\Polyspace`:

- **UNIX** — `/home/<username>/ .Polyspace`
- **Windows** — `C:\Users\<username>\AppData\Roaming\Polyspace`

The format of this ASCII file is as follows (tab-separated):

```
<id of launching> <server name of IP address> <public key>
```

where *<public key>* is a value in the range [0..F]

The fields in the file are tab-separated.

The file cannot contain blank lines.

Example:

```
1 m120 27CB36A9D656F0C3F84F959304ACF81BF229827C58BE1A15C8123786
2 m120 2860F820320CDD8317C51E4455E3D1A48DCE576F5C66BEEF391A9962
8 m120 2D51FF34D7B319121D221272585C7E79501FBCC8973CF287F6C12FCA
```

Sharing Verifications Between Accounts

To share a server verification with another user, you must provide the public key.

To share a verification with another user:

- 1 Find the line in your `analysis-keys.txt` file containing the `<ID>` for the job you want to share.
- 2 Add this line to the `analysis-keys.txt` file of the person who wants to share the file.

The second user can then download or manage the verification.

Magic Key to Share Verifications

A magic key allows you to share verifications without copying individual keys. This allows you to use the same key for all verifications launched from a single user account.

The format for a magic key is as follows:

```
0 <Server id> <your hexadecimal value>
```

When you add this key to your `analysis-keys.txt` file, all verification jobs you submit to the server queue use this key instead of a random one. All users who have this key in their `analysis-keys.txt` file can then download or manage your verification jobs.

Note This only works for verification jobs launched after you place the magic key in the file. If the verification was launched before the key was added, the normal key associated to the ID is used.

If `analysis-keys.txt` File is Lost or Corrupted

If your `analysis-keys.txt` file is corrupted or lost (removed by mistake) you cannot download your verification results. To access your verification results you must use administrator mode.

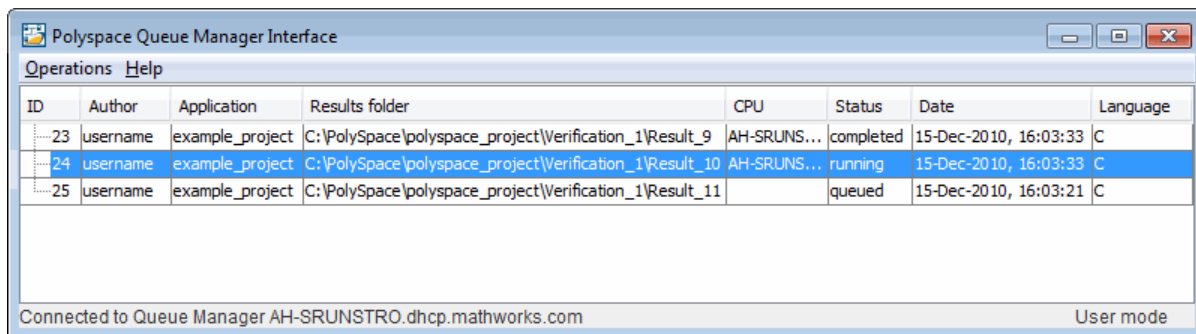
Note You must have the queue manager password to use Administrator Mode.

To use administrator mode:

- 1 Double-click the **Polyspace Spooler** icon:



The **Polyspace Queue Manager Interface** opens.



- 2 Select **Operations > Enter Administrator Mode**.

- 3 Enter the Queue Manager **Password**.

- 4 Click **OK**.

You can now manage all verification jobs in the server queue, including downloading results.

Running Verifications on Polyspace Client

In this section...

“Specifying Source Files to Verify” on page 6-31

“Starting Verification on Client” on page 6-32

“What Happens When You Run Verification” on page 6-33

“Monitoring the Progress of the Verification” on page 6-34

“Stopping the Verification Before It is Complete” on page 6-35

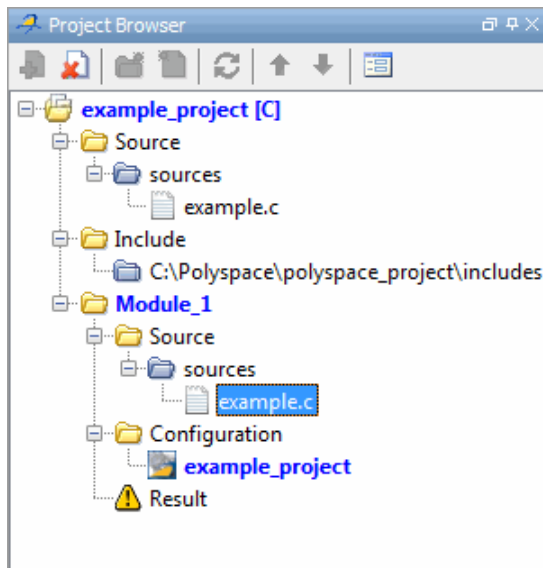
Specifying Source Files to Verify

Each Polyspace project can contain multiple modules. Each of these modules verifies a specific set of source files using a specific set of analysis options. Therefore, before you can launch a verification, you must decide which files in your project to verify, and add them to a module.

To copy source files to a module:

- 1 Open the project containing the files you want to verify.
- 2 In the Project Browser Source tree, select the source files you want to verify.
- 3 Right click any selected file, and select **Copy Source File to > Module_(#)**.

The selected source files appear in the Source tree of the module.



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

Starting Verification on Client

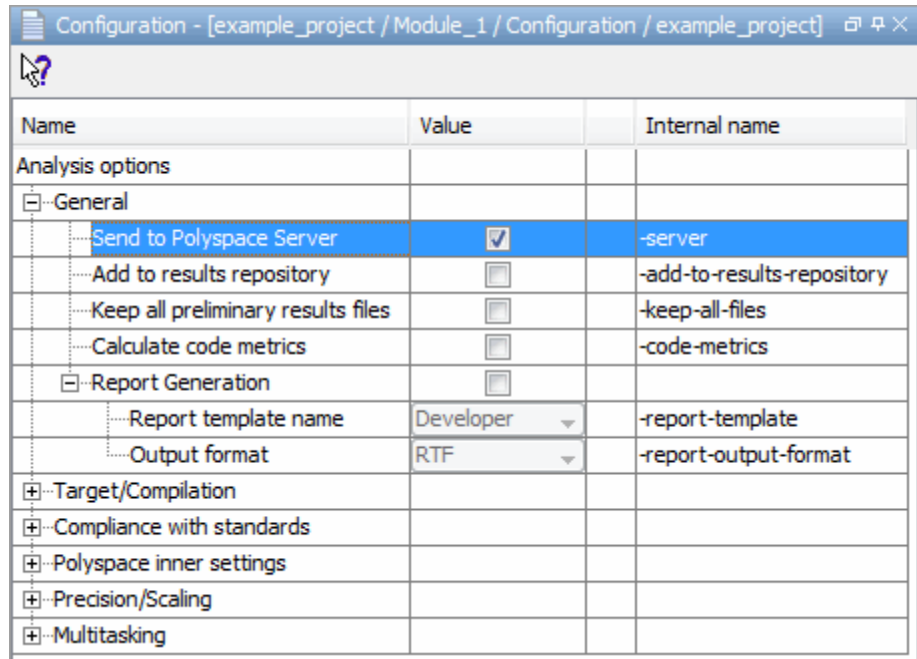
For the best performance, run verifications on a server. If the server is busy or you want to verify a small file, you can run a verification on a client.


Note Because a verification on a client can process only a limited number of variable assignments and function calls, the source code should have no more than 800 lines of code.

If you launch a verification on C or C++ code containing more than 2,000 assignments and calls, the verification will stop and you will receive an error message.

To start a verification that runs on a client:

- 1 In the Project Browser, select the module you want to verify.
- 2 Clear the **Send to Polyspace Server** check box in the General Analysis options.



- 3 Click the **Run** button  on the Project Manager toolbar.

The Output Summary and Progress Monitor windows become active, allowing you to monitor the progress of the verification.

Note If you see the message `Verification process failed`, click **OK** and go to “Verification Process Failed Errors” on page 7-2.

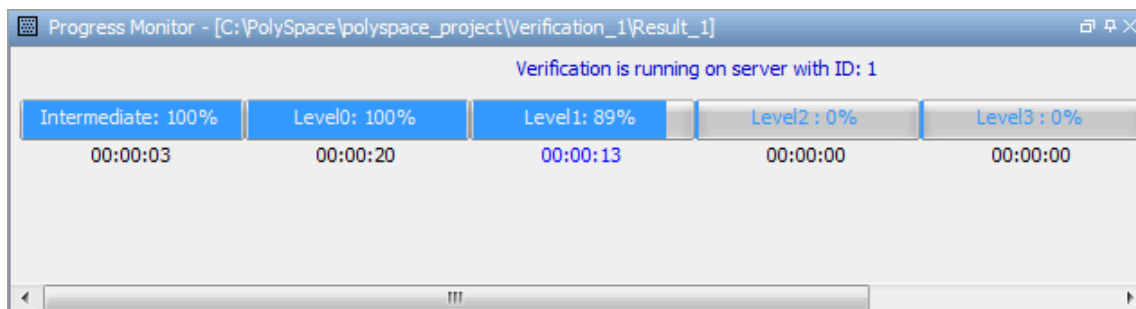
What Happens When You Run Verification

The verification has three main phases:

- 1 Checking syntax and semantics (the compile phase). Because Polyspace software is independent of any particular compiler, it ensures that your code is portable, maintainable, and complies with ANSI standards.
- 2 Generating a main if it does not find a main and the **Generate a Main** option is selected. For more information about generating a main, see “Main Generator Behavior for Polyspace Software” (C) or “Generate a main (-main-generator)” (C++) in the *Polyspace Products for C/C++ Reference*.
- 3 Analyzing the code for run-time errors and generating color-coded diagnostics.

Monitoring the Progress of the Verification

You can monitor the progress of the verification by viewing the progress monitor and logs at the bottom of the Project Manager perspective.



The progress monitor highlights the current phase in blue and displays the amount of time and completion percentage for that phase.

The logs report additional information about the progress of the verification. To view a log, click the button for that log. The information appears in the log display area at the bottom of the Project Manager window. Follow the next steps to view the logs:

- 1 Click the **Output Summary** tab to display compile phase messages and errors. You can search the log by entering search terms in the **Search in the log** box and clicking the left arrows to search backward or the right arrows to search forward.

2 Click the **Verification Statistics** tab to display statistics, such as analysis options, stubbed functions, and the verification checks performed.

3 Click the **Refresh** button  to update the display as the verification progresses.

4 Click the **Full Log** tab to display messages, errors, and statistics for all phases of the verification.

Note You can search the logs. In the **Search in the log** box, enter a search term and click the left arrows to search backward or the right arrows to search forward.

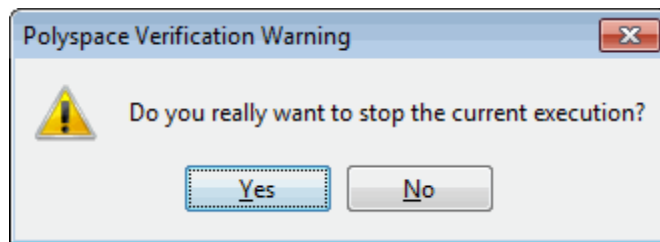
Stopping the Verification Before It is Complete

You can stop the verification before it is complete. If you stop the verification, results are incomplete. If you start another verification, the verification starts over from the beginning.

To stop a verification:

1 Click the **Stop** button  on the Project Manager toolbar.

A warning dialog box opens.



2 Click **Yes**.

The verification stops and the message `Verification process stopped` appears.

3 Click **OK** to close the **Message** dialog box.

Note Closing the Polyspace verification environment window does *not* stop the verification. To resume display of the verification progress, start the Polyspace software and open the project.

Running Verifications from Command Line

In this section...
“Launching Verifications in Batch” on page 6-37
“Managing Verifications in Batch” on page 6-37

Launching Verifications in Batch

A set of commands allow you to launch a verification in batch.

All these commands begin with the following prefixes:

- **Server verification** —
Polyspace_Install/Verifier/bin/polyspace-remote-c
or *Polyspace_Install/Verifier/bin/polyspace-remote-cpp*
- **Client verification** —*polyspace-remote-desktop-c*
or *polyspace-remote-desktop-cpp*.

These commands are equivalent to commands with a prefix *Polyspace_Install/bin/polyspace-*.

For example, *polyspace-remote-desktop-c -server [<hostname>:[<port>] | auto]* allows you to send a C client verification remotely.

Note If your Polyspace server is running on Windows, the batch commands are located in the */wbin/* folder. For example, *Polyspace_Install/Verifier/wbin/polyspace-remote-c.exe* or *Polyspace_Install/Verifier/wbin/polyspace-remote-cpp.exe*.

Managing Verifications in Batch

In batch, a set of commands allow you to manage verification jobs in the server queue.

On UNIX platforms, all these command begin with the prefix *Polyspace_Common/RemoteLauncher/bin/psqueue-*.

On Windows platforms, these commands begin with the prefix *Polyspace_Common/RemoteLauncher/wbin/psqueue-*:

- `psqueue-download <id> <results dir>` — download an identified verification into a results folder. When downloading a unit-by-unit verification group, all the unit results are downloaded and a summary of the download status for each unit is displayed.
 - `[-f]` force download (without interactivity)
 - `-admin -p <password>` allows administrator to download results.
 - `[-server <name>[:port]]` selects a specific Queue Manager.
 - `[-v|version]` gives release number.
- `psqueue-kill <id>` — kill an identified verification. For unit-by-unit verification groups, you can stop the entire group, or individual jobs within the group. Stopping an individual job does not kill the entire group.
- `psqueue-purge all|ended` — remove all completed verifications from the queue. For unit-by-unit verification jobs, no jobs are removed until the entire group has been verified.
- `psqueue-dump` — gives the list of all verifications in the queue associated with the default Queue Manager. Unit-by-unit verification groups are shown using a tree structure.
- `psqueue-move-down <id>` — move down an identified verification in the Queue. Individual jobs can be moved within a unit-by-unit verification group, but not outside of the group.
- `psqueue-remove <id>` — remove an identified verification in the queue. You cannot remove a single job that is part of a unit-by-unit verification group, you can only remove the entire group.
- `psqueue-get-qm-server` — give the name of the default Queue Manager.
- `psqueue-progress <id>`: give progression of the currently identified and running verification. This command does not apply to unit-by-unit verification groups, only the individual jobs within a group.
 - `[-open-launcher]` display the log in the graphical user interface.

- [-full] give full log file.
- psqueue-set-password <password> <new password> — change administrator password.
- psqueue-check-config — check the configuration of Queue Manager.
 - [-check-licenses] check for licenses only.
- PSQueueSpooler — open the Polyspace Queue Manager Interface (Spooler) graphical user interface.
 - [-server <hostname>] specify the name of a specific Polyspace server. The Spooler connects to the specified server instead of the default server.
- psqueue-upgrade — Allow to upgrade a client side (see the Polyspace Installation Guide in the *Polyspace_Common/Docs* folder).
 - [-list-versions] give the list of available release to upgrade.
 - [-install-version <version number> [-install-dir <folder>]] [-silent] allow to install an upgrade in a given folder and in silent.

Note *Polyspace_Common/bin/psqueue-<command> -h* gives information about all available options for each command.

Troubleshooting Verification Problems

- “Verification Process Failed Errors” on page 7-2
- “Compilation Errors” on page 7-9
- “C++ Dialect Issues” on page 7-21
- “C Link Errors” on page 7-30
- “C++ Link Errors” on page 7-36
- “Standard Library Function Stubbing Errors” on page 7-40
- “Automatic Stubbing Errors” on page 7-47
- “Troubleshooting Using Preprocessed Files” on page 7-50
- “Reducing Verification Time” on page 7-55
- “Obtaining Configuration Information” on page 7-74
- “Removing Preliminary Results Files” on page 7-77

Verification Process Failed Errors

In this section...
“Reasons Verification Can Fail” on page 7-2
“Viewing Error Information When Verification Stops” on page 7-2
“Hardware Does Not Meet Requirements” on page 7-3
“You Did Not Specify the Location of Included Files” on page 7-4
“Polyspace Software Cannot Find the Server” on page 7-4
“Limit on Assignments and Function Calls” on page 7-7

Reasons Verification Can Fail

If you see a message that includes `Verification process failed`, the Polyspace software could not perform the verification. The following sections present some possible reasons for a failed verification.

Message	See
Errors found when verifying host configuration	“Hardware Does Not Meet Requirements” on page 7-3
<code>include.h: No such file or folder (where include.h represents the included file)</code>	“You Did Not Specify the Location of Included Files” on page 7-4
<code>Error: Unknown host :</code>	“Polyspace Software Cannot Find the Server” on page 7-4
<code>License error: number-of assignments and function calls is too big for -unit mode</code>	“Limit on Assignments and Function Calls” on page 7-7

Viewing Error Information When Verification Stops

If the Polyspace software provides no graphical result, it lists the errors and their locations at the end of the log file. To find the errors, scroll through the verification log file, starting at the end and working backwards.

The following example shows the log file information that results if you use the C++ `-class-analyzer` *argument*, but the verification cannot find *argument* in the source code:

```

*****
***
*** Beginning Visual C++ 8 source normalization
***
*****
**** Visual C++ 8 source normalization - 1 (Loading)
-----
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-cpp: 25.6real, 25.6u + 0s (0gc)
Error: Exiting because of previous error
***
*** End of Polyspace Verifier analysis
***

```

Hardware Does Not Meet Requirements

Message

In the verification log:

```
Errors found when verifying host configuration.
```

Cause

The verification fails if your computer does not have the minimal hardware requirements. For information about the hardware requirements, see

www.mathworks.com/products/polyspaceclientc/requirements.html.

Solution

You can:

- Upgrade your computer to meet the minimal requirements.
- In the General section of the **Analysis** options, select **Continue with current configuration** and run the verification again.

You Did Not Specify the Location of Included Files

Message

In the verification log (where `include.h` represents the included file):

```
include.h: No such file or folder
```

Cause

Either the files are missing or you did not specify the location of included files.

Solution

Do one of the following:

- Include the file in the designated location.
- Specify the proper location of include files.

MathWorks recommends that you create a project file to store include files, as described in “Creating a Project” on page 3-2.

Polyspace Software Cannot Find the Server

Message

Search in the verification log for:

```
Error: Unknown host :
```

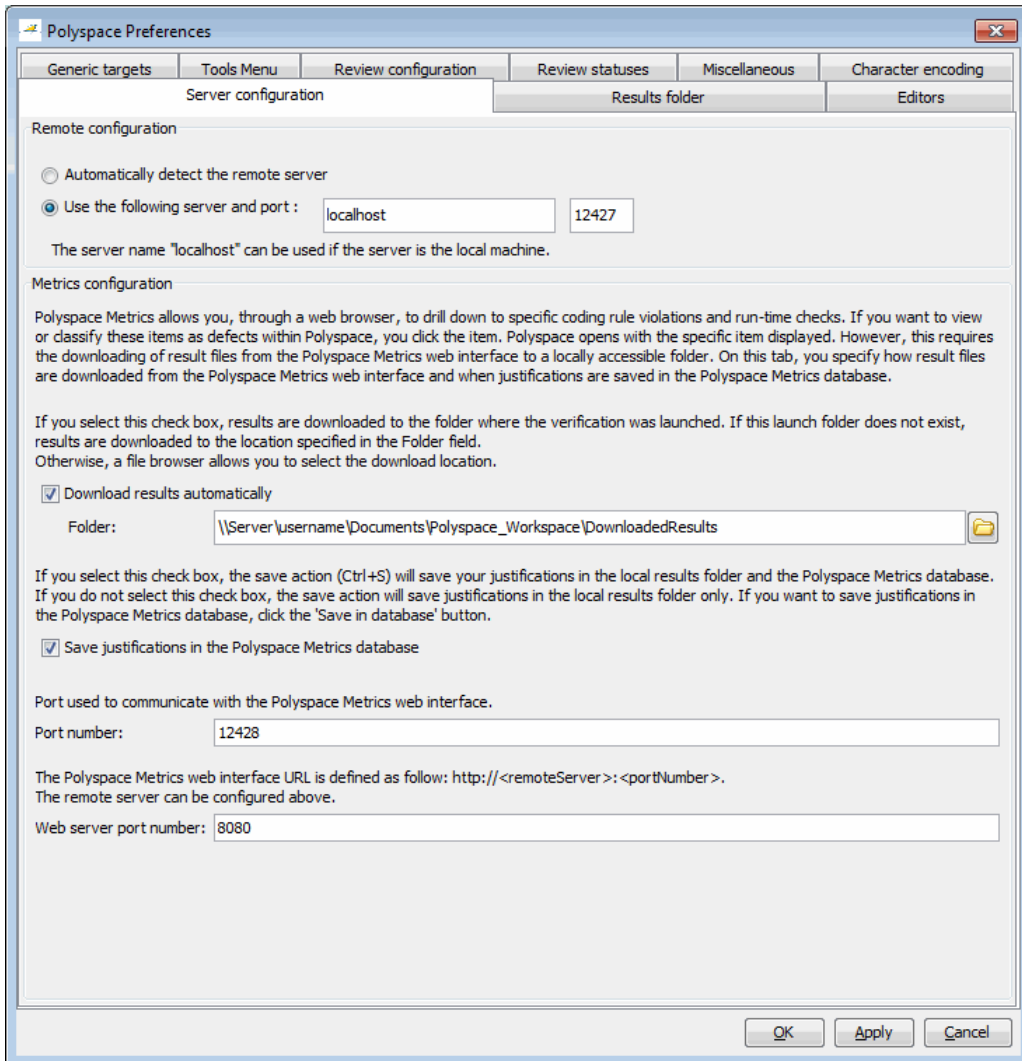
Cause

Polyspace software uses information in the preferences to locate the server. In this case, Polyspace software cannot find the server.

Solution

To find the server information in the preferences:

- 1** Select **Options > Preferences**.
- 2** Select the **Server configuration** tab.



How you deal with this error depends on the selected remote configuration option.

Remote Configuration Option	Solution
Automatically detect the remote server	Specify the server by selecting Use the following server and port and providing the server name and port.
Use the following server and port	Confirm the server name and port number are accurate.

For information about setting up a server, see the *Polyspace Installation Guide*.

Limit on Assignments and Function Calls

C Message

```
*****
Beginning C to intermediate language translation
*****
C to intermediate language translation 1 (P_SP)
...

*** License error: number of assignments and function calls is
too big for -unit mode (5534 v.s 2000).
*** Stopping.
```

C++ Message

```
*****
Beginning C++ to intermediate language translation
*****
C++ to intermediate language translation 1 (P_SP)
...

*** License error: number of assignments and function calls is
too big for -unit mode (5534 v.s 3000).
*** Aborting.
```

Cause

Polyspace Client for C/C++ software can verify:

- C code with up to 2,000 assignments and calls
- C++ code with up to 3,000 assignments and calls

Solution

To verify C code containing more than 2,000 assignments and calls, or C++ code containing more than 3,000 assignments and calls, launch your verification with Polyspace Server for C/C++.

Compilation Errors

In this section...

“Compilation Error Overview” on page 7-9
“Checking Compilation Before Running Verification” on page 7-10
“Examining Compile Log” on page 7-10
“Compilation Messages Described in This Section” on page 7-12
“Syntax Error” on page 7-13
“Undeclared Identifier” on page 7-14
“No Such File or Folder” on page 7-15
“#error directive” on page 7-16
“Class, Array, Struct or Union is Too Large” on page 7-16
“Unsupported Non-ANSI Keywords (C)” on page 7-17
“Initialization of Global Variables (C++)” on page 7-19

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)

Checking Compilation Before Running Verification

The Compilation Assistant allows you to check your project for compilation problems before launching a verification, allowing you to avoid many compilation errors. When the Compilation Assistant detects an error, it reports the problem and suggests possible solutions.

For information on using the Compilation Assistant, see “Checking for Compilation Problems” on page 6-5.







Examining Compile Log

The compile log displays compile phase messages and errors. To search the log, enter search terms in the **Search in the log** box. Click the left arrows to search backward or click the right arrows to search forward.







To examine errors in the Compile log:

- 1 Click the **Output Summary** tab at the bottom of the Project Manager perspective.

A list of compile phase messages appear in the log part of the window.







Status	Description	File	Line
	PolySpace Launcher for CPP verification start at Nov 30, 20...		
	invalid combination of type specifiers	MyCountClass.cpp	1
	first parameter of allocation function must be of type "size_t"	MyCountClass.cpp	8
	Failed compilation of MyCountClass.cpp		
	Verifier has detected compilation error(s) in the code.		
	Exiting because of previous error		

- 2 Click any of the messages to see message details, as well as the full path of the file containing the error.

Status	Description	File	Line
	PolySpace Launcher for CPP verification start at Nov 30, 20...		
	invalid combination of type specifiers	MyCountClass.cpp	1
	first parameter of allocation function must be of type "size_t"	MyCountClass.cpp	8
	Failed compilation of MyCountClass.cpp		
	Verifier has detected compilation error(s) in the code.		
	Exiting because of previous error		

Detail			
File C:\polyspace_project_new\sources\MyCountClass.cpp line 1			
Error:			
invalid combination of type specifiers			
typedef unsigned long wchar_t;			
^			

- 3** To open the source file referenced by any message, right click the row for the message, then select **Open Source File**.

Status	Description	File	Line
	PolySpace Launcher for CPP verification start at Nov 30, 200...		
	invalid combination of type specifiers	MyCountClass.cpp	1
	first parameter of allocation function must be	MyCountClass.cpp	8
	Failed compilation of MyCountClass.cpp		
	Verifier has detected compilation error(s) in t		
	Exiting because of previous error		

Detail			
File C:\polyspace_project_new\sources\MyCountClass.cpp line 1			
Error:			
invalid combination of type specifiers			
typedef unsigned long wchar_t;			
^			

The file opens in your text editor.

Note You must configure a text editor before you can open source files. See “Configuring Text and XML Editors”.

- 4** If you do not understand the error information in the **Detail** pane, right-click the row for the message and select **Open Preprocessed File**.

This action opens the `.ci` file that the Polyspace software uses to compile the source file. The contents of this file helps you understand the compilation error.

- 5 The two compilation errors in this example occurred because the compiler encountered unexpected type definitions. To correct the errors in this example:

- invalid combination of type specifiers

in the line

```
typedef unsigned long wchar_t;
```

By default, `wchar_t` is a C++ keyword. For some C++ compilers, `wchar_t` is not defined, and you can use it as a `typedef`. The Polyspace compiler allows you to define `wchar_t` as a `typedef` using the `-wchar-t-is-typedef` option.

- first parameter of allocation function must be of type `"size_t"`

in the line

```
void * operator new(unsigned long mysize);
```

the `new` operator has been prototyped with an unsigned long parameter, which requires that `size_t` be unsigned long. To fix this compile error, set the `-size-t-is-unsigned-long` option.

- 6 Rerun the verification.

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 7-13
undeclared identifier	"Undeclared Identifier" on page 7-14

Phrase Found in Message	See
No such file or folder or Catastrophic error: could not open source file	“No Such File or Folder” on page 7-15
#error: directive	“#error directive” on page 7-16

This section also describes error messages triggered by unsupported keywords. See “Unsupported Non-ANSI Keywords (C)” on page 7-17.

This section includes sample code that triggers the example message.

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 “How to Gather Compilation Options Efficiently” on page 4-30.

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 “How to Gather Compilation Options Efficiently” on page 4-30.

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, all 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.

For more information, see “Setting Up a Target” on page 4-2.

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`. All 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
bin 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 %POLYSPACE_C%\Verifier\tools\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 Client for C/C++, accept the default initialization of static data member during the declaration.

C++ Dialect Issues

In this section...

“ISO versus Default Dialects” on page 7-21

“CFront2 and CFront3 Dialects” on page 7-23

“Visual Dialects” on page 7-24

“GNU Dialect” on page 7-26

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 ;
```

All 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 a correct 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

The GNU dialect is based on GCC 3.4. 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)) ;

}
```

- All 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 correctly 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 7-30
“Function: Wrong Argument Type” on page 7-31
“Function: Wrong Argument Number” on page 7-31
“Variable: Wrong Type” on page 7-32
“Variable: Signed/Unsigned” on page 7-32
“Variable: Different Qualifier” on page 7-33
“Variable: Array Against Variable” on page 7-33
“Variable: Wrong Array Size” on page 7-34
“Missing Required Prototype for varargs” on page 7-34

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`. All 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: Wrong Argument 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 'arg 1' type incompatible with definition

```
int f(float y)          int f(int *y);
{
int r;                  void main(void)
r=12;                   {
}                        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

Verifying cross-files ANSI C compliance ...

Error: global declaration of 'f' function has incompatible type with its definition
declared function type has incompatible args. number with definition

```
int f(float y)          int f(int *y);
{
int r;                  void main(void)
r=12;                   {
}                        int r;
                        r = f(&r);
                        }
```

Solution

These two functions have a different number of arguments. This mismatch in the number of arguments results in a nondeterministic execution.

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 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 7-36
“Lib C Stubbing Errors” on page 7-37

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 Standard Library Functions and Polyspace Stubs” on page 7-40

“_polyspace_stdstubs.c Compilation Errors” on page 7-40

“Troubleshooting Approaches for Standard Library Function Stubs” on page 7-42

“Restart with the -I option” on page 7-42

“Include Files with Stubs to Replace Automatic Stubbing” on page 7-43

“Create a _polyspace_stdstubs.c File with Necessary Includes” on page 7-44

“Provide a .c file Containing a Prototype Function” on page 7-45

“Ignore _polyspace_stdstubs.c” on page 7-46

Conflicts Between Standard Library Functions and Polyspace Stubs

A code set can compile successfully for a target, but during the `_polyspace_stdstubs.c` compilation phase for that same code, Polyspace software can generate 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 that function

For more information about errors generated during automatic stub creation, see “Automatic Stubbing Errors” on page 7-47.

`_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 any of the techniques in any order. Consider trying the simplest approaches first, and trying other techniques as necessary 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 7-42
- “Include Files with Stubs to Replace Automatic Stubbing” on page 7-43
- “Create a `_polyspace_stdstubs.c` File with Necessary Includes” on page 7-44
(Use when precision is important enough to justify extensive code preparation time)
- “Provide a `.c` file Containing a Prototype Function” on page 7-45
(Use when you do not want to invest much time for code preparation time)
- “Ignore `_polyspace_stdstubs.c`” on page 7-46

If the problem persists after trying all 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.

Include Files with Stubs to Replace Automatic Stubbing

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 correct 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 a `_polyspace_stdstubs.c` File with Necessary Includes” on page 7-44
- “Provide a `.c` file Containing a Prototype Function” on page 7-45
- “Ignore `_polyspace_stdstubs.c`” on page 7-46

Create a `_polyspace_stdstubs.c` File with Necessary 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 Do one of the following (MathWorks recommends the first approach):
 - Provide the `string.h` file that contains the real prototype and type definitions for the stubbed function.
 - 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-c -I /homemade_include
```

or

```
polyspace-c -I /our_target_include_path
```

Provide a .c file Containing a 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 “How to Gather Compilation Options Efficiently” on page 4-30.

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-c -D POLYSPACE_NO_STANDARD_STUBS
```

- Deactivate all stubbed extensions to ANSI C standard by using `-D POLYSPACE_STRICT_ANSI_STANDARD_STUBS`. For example:

```
polyspace-c -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 any 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 a proper 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 7-47

“Function Pointer Error” on page 7-47

“Unknown Prototype Error” on page 7-49

“Parameter -entry-points Error” on page 7-49

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 7-40.

Function Pointer Error

Message

```
Fatal error: function 'f' refers to a function pointer either  
much too complex or in a too-complex data-structure, or with  
unknown parameters.
```

```
It cannot be stubbed automatically.
```

Solutions

Consider a prototype `f` that contains a function pointer as a parameter.

If the function pointer prototype only contains scalars and/or floats, the Polyspace software automatically stubs `f`.

For example, the verification process automatically stubs the following function:

```
int f()  
void (*ptr_ok)(int, char, float),
```

```
other_type1 other_param1);
```

If this function pointer prototype also contains pointers, you get the error message and have to stub the `f` function manually.

For example, stub the following function manually (unless you use the `-permissive-stubber` option):

```
int f()
void (*ptr_ok)(int *, char, float),
other_type1 other_param1);
```

If you use the `-permissive-stubber` option on the following function `f()`, you still see the function pointer error. The Polyspace software does not recognize if the `f()` calls the function pointer `ptr`.

```
typedef void (*ptr_func_T) (int, int*, float);
extern ptr_func_T* extern_function_ptr(void);
extern int f(ptr_func_T, int other_param1);

void function_link_stubber(void)
{
    ptr_func_T ptr = extern_function_ptr();
    f(ptr,10);
    extern_function_ptr();
}
```

In this case, to resolve the error, you can provide a manual stub of `f()` that does not call the function pointer `ptr`. Add this stub to the verification. The code for this solution is:

```
typedef void (*ptr_func_T) (int, int*, float);
extern ptr_func_T* extern_function_ptr(void);
extern int pst_random(void);
int f(ptr_func_T ptrf, int other_param1)
{
    return pst_random();
}
```

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.

Troubleshooting Using Preprocessed Files

In this section...
“Overview of Preprocessed (.ci) Files” on page 7-50
“Example .ci File” on page 7-50
“Troubleshooting Methodology” on page 7-52

Overview of Preprocessed (.ci) Files

The preceding sections explained common types of compile and linking error messages. However, sometimes the error messages are not sufficient to find the cause of a problem, because the problem does not correspond to any of the common error messages listed above.

Polyspace software, like other compilers, transforms source code into preprocessed code. These preprocessed files are located in the folder: *<results folder>/CPP-ALL/SRC/MACROS* or *<results folder>/ALL/SRC/MACROS*. They have a .ci extension, and they can help you understand the cause of an error.

Example .ci File

A *.ci file is a copy of original file containing whole header files inside a unique file:

- Compile flags activate some parts of code,
- Macro commands are expanded,
- Arguments which are described as `#define xxx`, are replaced by their owned definition,
- etc.

Extension.cpp	Extension.h
<pre> #include "Extension.h" Extension::Extension(int val) { m_val = 0; ABS(val); if (val > MAX_VALUE) m_val = -1; } #ifdef _DEBUG void Extension::message(char*) {} #else void print(char*) {} #endif </pre>	<pre> #define MAX_VALUE 10 #define ABS(x) ((x)<0?(x):- (x)) class Extension { public: int m_val; Extension(int val); #ifdef _DEBUG void message(char*); #else void print(char*); #endif }; </pre>

The associated file `Extension.ci` uses the compile flag `_DEBUG`:

```

# 1 "../sources/extension.cpp"
# 1 "<Product>/Verifier/cinclude/polyspace_std_decls.h" 1

# 1 "../sources/extension.cpp" 2
# 1 "../sources/extension.h" 1
class Extension
{
public:
    int m_val;
    Extension(int val);

    message(char*);    // _DEBUG activates the message member function

};

# 2 "../sources/extension.cpp" 2

Extension::Extension(int val)
                    
```

```
{
  m_val = 0;
  ((val)<0?(val): -(val)); // EXPANDED MACRO ABS

  if (val > 10 ) // MAX_VALUE REPLACED BY 10
    m_val = -1;
}

void Extension::message(char*) {}
```

Analyzing these files with the compile flag `-D _DEBUG` expands the code fully and may help you find problems quickly.

Troubleshooting Methodology

This section is designed to help you understand error messages, and the differences between your compiler and Polyspace compilation:

- 1** Check whether the compile error messages come from a dialect problem.
- 2** Verify that link error messages are related or not to:
 - A C++ stubbing error which could be resolved by an option (like `-no-stl-stubs`)
 - C stubbing error which could be resolved by an option or a compilation flag like `POLYSPACE_NO_STANDARD_STUBS` or `POLYSPACE_STRICT_ANSI_STANDARD_STUBS`.
- 3** Check the preprocessed `*.ci` files to see the expanded files. Looking at the preprocessed code can help you find errors faster.

Example with these original codes:

Child1.c	Child2.c	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: int m_val; };</pre>

Error message:

```
Pre-linking C++ sources ...
"../sources/test.h", line 4: error: declaration of function
"Test::Test(const Test &)" does not match function
"Test::algorithm" during compilation of "CPP-ALL/SRC/
MACROS/Child2.cpp" (one may have been removed due to #define)
    class Test
      ^
    detected during compilation of secondary translation unit
"CPP-ALL/SRC/MACROS/Child2.cpp"
```

In this example it is clear that DEBUG is defined in child1.c but not in child2.c, which creates two different definition of the class test.

The solution can also come up by comparing the two *.ci files:

Test.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; }; # 2 "../sources/Test.cpp" 2 </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; }; # 2 "../sources/Child2.h" 2 </pre>

Looking at the preprocessed code can help to find errors faster.

Reducing Verification Time

In this section...

- “Factors Impacting Verification Time” on page 7-55
- “Displaying Verification Status Information” on page 7-56
- “Techniques for Improving Verification Performance” on page 7-57
- “Turning Antivirus Software Off” on page 7-59
- “Tuning Polyspace Parameters” on page 7-59
- “Subdividing Code” on page 7-60
- “Reducing Procedure Complexity” on page 7-70
- “Reducing Task Complexity” on page 7-72
- “Reducing Variable Complexity” on page 7-72
- “Choosing Lower Precision” on page 7-73

Factors Impacting Verification Time

These factors affect how long it takes to run a verification:

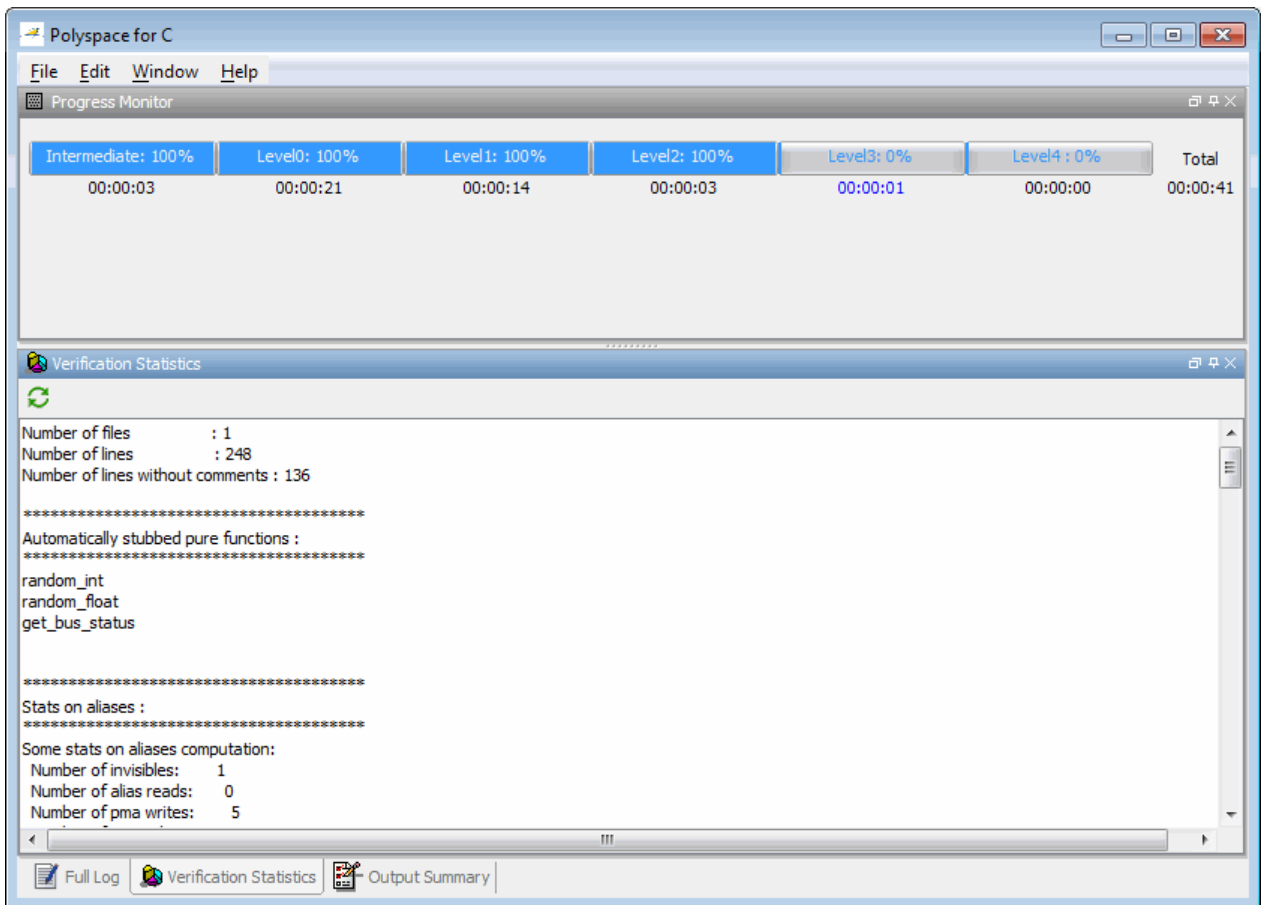
- The size of the code
- The number of global variables
- The nesting depth of the variables (the more nested they are, the longer it takes)
- The depth of the call tree of the application
- The intrinsic complexity of the code, particularly with regards to pointer manipulation

Because many factors impact 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.

Displaying Verification Status Information

For *client* verifications, monitor the progress of your verification using the **Progress Monitor** and **Verification Statistics** tabs in the Project Manager. For more information, see “Monitoring the Progress of the Verification” on page 6-34.

For *server* verifications, use the Polyspace Queue Manager to follow the progress of your verification. For more information, see “Monitoring Progress of Server Verification” on page 6-16.



The progress bar highlights each completed phase and displays the amount of time for that phase. You can estimate the remaining verification time by extrapolating from this data, and considering the number of files and passes remaining.

Techniques for Improving 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).

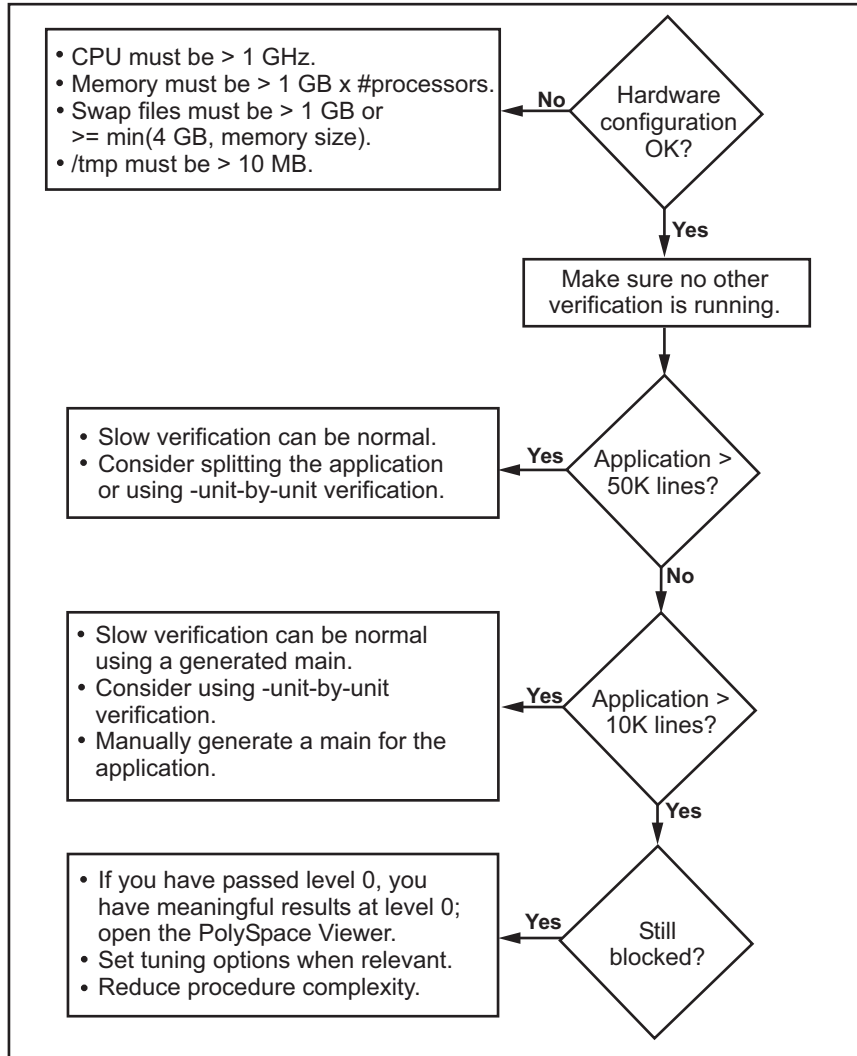
You can use several techniques to reduce the amount of time required for a verification, including

- “Turning Antivirus Software Off” on page 7-59
- “Tuning Polyspace Parameters” on page 7-59
- “Subdividing Code” on page 7-60
- “Reducing Procedure Complexity” on page 7-70
- “Reducing Task Complexity” on page 7-72
- “Reducing Variable Complexity” on page 7-72
- “Choosing Lower Precision” on page 7-73

You can combine these techniques. See the following performance-tuning flow charts:

- “Standard Scaling Options Flow Chart” on page 7-58
- “Reducing Code Complexity” on page 7-59

Standard Scaling Options Flow Chart



Reducing Code Complexity

To reduce code complexity, MathWorks recommends that you try the following techniques, in the order listed:

- “Reducing Procedure Complexity” on page 7-70
- “Reducing Task Complexity” on page 7-72
- “Reducing Variable Complexity” on page 7-72

After you use any of these techniques, restart the verification.

Turning Antivirus Software Off

Disabling or switching off any third-party antivirus software for the duration of a verification can reduce the verification time by up to 40%.

Tuning 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

MathWorks suggests that you 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, an appropriate launching command is

```
polyspace-c -00 -respect-types-in-globals -k-limiting 0
```

Subdividing Code

- “An Ideal Application Size” on page 7-60
- “Benefits of Subdividing Code” on page 7-60
- “Possible Issues with Subdividing Code” on page 7-61
- “Recommended Approach” on page 7-62
- “Selecting a Subset of Code” on page 7-64

An Ideal Application Size

People have used Polyspace software to analyze numerous applications with greater than 100,000 lines of code.

There always is a compromise 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
- Get correct results are correct because you do not need to remove any thread affecting change 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):

$$[-2 ; 20]$$

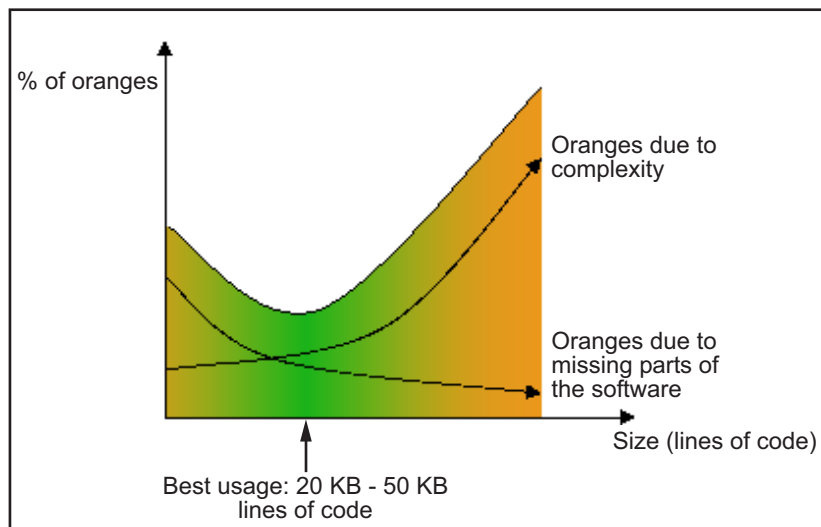
This phenomenon increases the number of orange warnings when the size of the program becomes large.

Recommended Approach

MathWorks recommends that you 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.

Selecting 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 any assumption 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 (see “Stubbing” on page 5-2).

```

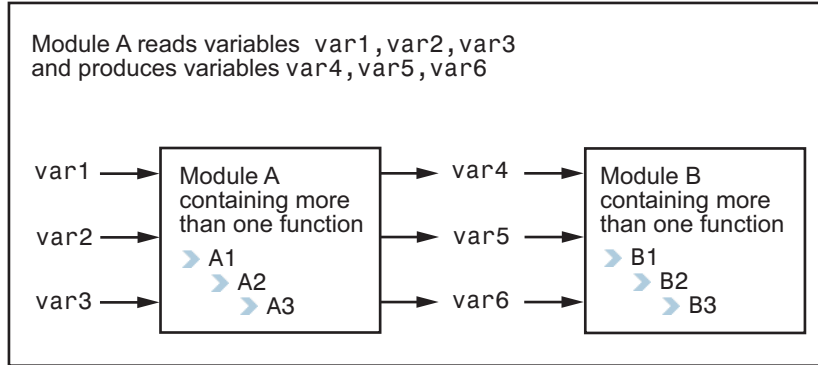
//removed definition of complete_treatment_based_on_x
void main(void)
{
  x = ... // what ever
  y = complete_treatment_based_on_x(x); // now stubbed!
}

```

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 answers, just like reality.
- **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 necessary to isolate a meaningful subset. It means that a logical split is possible for any application, in accordance with the logic of the data.
- The results remain valid, because there 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 answers, just like the code would at execution time.

- 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. This approach is accurate 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 red errors and bugs 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 all functions that the subset of the code does not call.

Reducing 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.

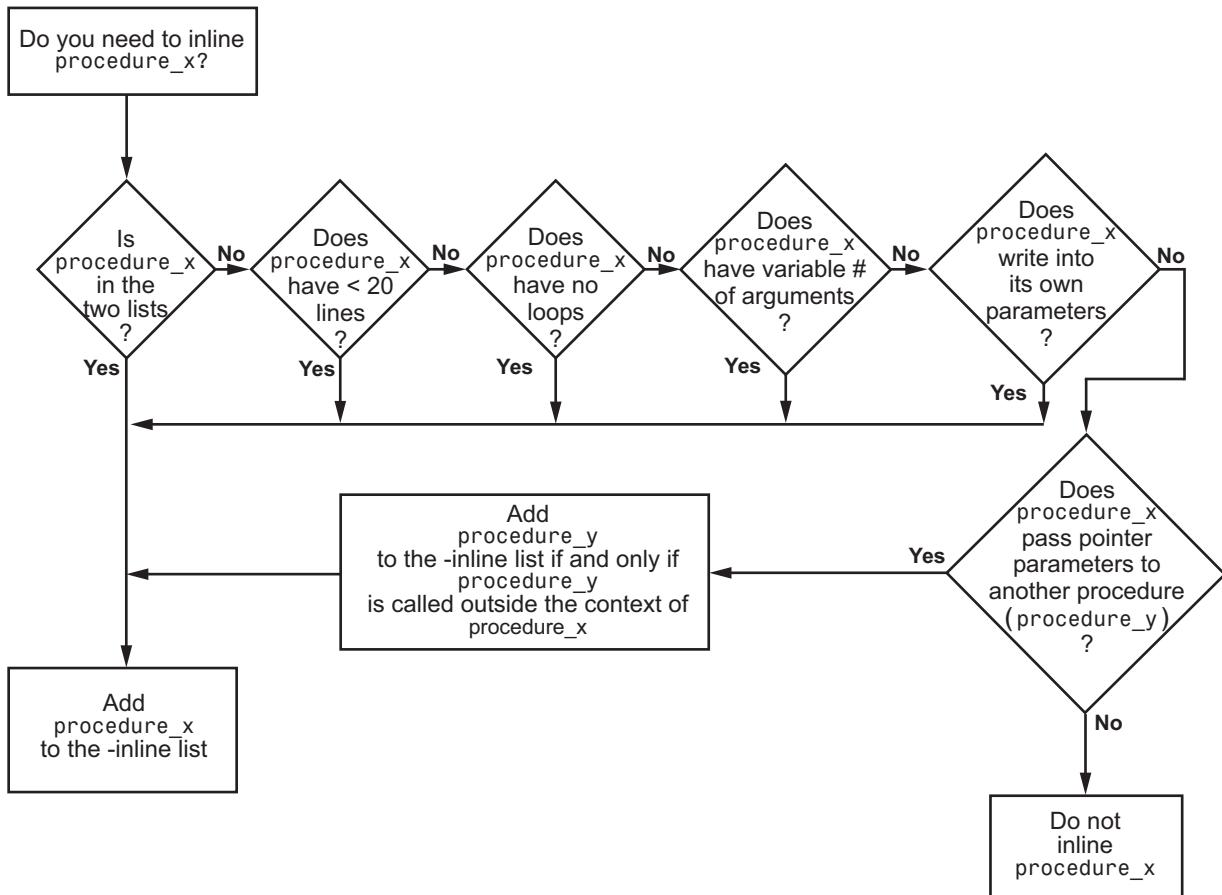
The `-inline` option creates clones of each specified procedure for each call to it. This option reduces the number of aliases in a procedure, and can improve precision in some situations.

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.

Reducing 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.

Reducing 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.

Choosing 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:

- -O0 — shorter computation time
- -O2 — less orange warnings
- -O3 — less orange warnings and longer computation time. MathWorks recommends using this option only for projects containing less than 1,000 lines of code.

Obtaining Configuration Information

The `polyspace-ver` command allows you to gather information quickly about your system configuration. Use this information when entering support requests.

Configuration information includes:

- Hardware configuration
- Operating system
- Polyspace licenses
- Specific version numbers for Polyspace products
- Any installed Bug Report patches

To obtain your configuration information, enter the following command:

- **UNIX/Linux** — `Polyspace_Install/Verifier/bin/polyspace-ver`
- **Windows** — `Polyspace_Install/Verifier/wbin/polyspace-ver.exe`

The configuration information appears.

```

Administrator: C:\Windows\system32\cmd.exe
C:\PolySpace\PolySpaceForCandCPP_R2011a\Verifier\wbin>polyspace-ver.exe
-----
Machine Hardware Configuration:
* Number of CPUs      : 4
* CPU frequency      : 2.400GHz
* CPU type           : i686
* Memory             : 5.98GB
* Swap               : 11.96GB
* /tmp free space    : 168.37GB
-----

Machine Software Configuration:
Windows 7 x64 (WOW64) <>
-----

Polyspace Licenses:
PolySpace_Client_C_CPP:
  License Number: 5173
  Expiration date: 01-mar-2012

PolySpace_Server_C_CPP:
  License Number: 5173
  Expiration date: 01-mar-2012

PolySpace_Model_Link_SL:
  License Number: 5173
  Expiration date: 01-mar-2012

PolySpace_Model_Link_TL:
  License Number: 5173
  Expiration date: 01-mar-2012

PolySpace_UML_Link_RH:
  License Number: 5173
  Expiration date: 01-mar-2012
-----

Polyspace Versions:
Polyspace Version R2011a
* Kernel                CC-8.1.0.1
* Launcher              IHML-R2011a-U3
* Remote Launcher       RL-R2011a-U6
* Polyspace In One Click POC-R2011a-U3
* MBD Plugin            MBD-R2011a-U6
* Automatic Orange Tester AOT-R2011a-U6

Remote Launcher configuration
* Compatibility version 5_0_4

Server :
AH-SRUNS
-----

C:\PolySpace\PolySpaceForCandCPP_R2011a\Verifier\wbin>

```

Note You can obtain the same configuration information by selecting **Help > About** in the Polyspace Verification Environment.

Removing Preliminary Results Files

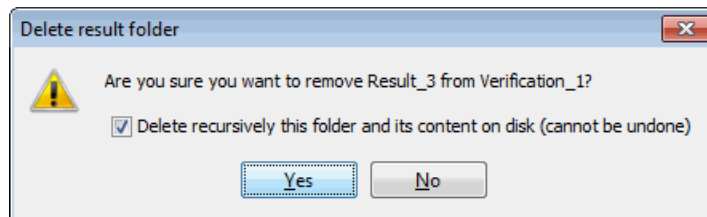
By default, the software automatically deletes preliminary results files when they are no longer needed by the verification. However, if you run a client verification using the option `-keep-all-files`, preliminary results files are retained in the results folder. This allows you to restart the verification from any stage, but can leave unnecessary files in your results folder.

If you later decide that you no longer need these files, you can remove them.

To remove preliminary results files,

- 1 Open the project containing the results you want to delete in the Project Manager.
- 2 Select the results you want to delete.
- 3 Press the **Delete** key on your keyboard.

The Delete Results folder dialog box opens.



- 4 If you want to delete the entire results folder, select **Delete recursively this folder**, otherwise clear the check-box.
- 5 Click **Yes**.

The results files are deleted.

Reviewing Verification Results

- “Before You Review Polyspace Results” on page 8-2
- “Opening Verification Results” on page 8-8
- “Reviewing Results Systematically” on page 8-34
- “Reviewing All Checks” on page 8-44
- “Tracking Review Progress” on page 8-55
- “Importing and Exporting Review Comments” on page 8-64
- “Generating Reports of Verification Results” on page 8-68
- “Using Polyspace Results” on page 8-80

Before You Review Polyspace Results

In this section...

“Overview: Understanding Polyspace Results” on page 8-2

“Why Gray Follows Red and Green Follows Orange” on page 8-3

“The Message and What It Means” on page 8-4

“The Code Explanation” on page 8-5

Overview: Understanding Polyspace Results

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 that never has a run-time error (safe code).

When you analyze these colors, 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. This means that red checks are always followed by gray checks, and orange checks only propagate the green parts 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.

Why Gray Follows Red and Green Follows Orange

Gray checks follow **red** checks, and **green** checks are propagated out of **orange** checks.

In the following example, consider why:

- The gray checks follow the **red** in the red function.
- There are **green** checks relating to the array.

```

void red(void)                extern int Read_An_Input(void);
{                               void propagate(void)
int x;                         {
x = 1 / x ;                   int X;
x = x + 1;                    int y[100];
}                               X = Read_An_Input();
                               y[X] = 0; // [array index within bounds]
                               y[X] = 0;
                               }

```

Consider each line of code for the red function:

- When Polyspace verification divides by X , X is not initialized. Therefore, the corresponding check (Non Initialized Variable) on X is red.
- As a result, Polyspace verification stops all possible execution paths because they all produce an RTE. Therefore, the subsequent instructions are gray (unreachable code).

Now, consider each line of code for the propagate function:

- X is assigned the return value of `Read_An_Input`. After this assignment, $X = [-2^{31}, 2^{31}-1]$.
- At the first array access, you might see an “out of bounds” error because X can equal -3 as well as 3 .
- Subsequently, all conditions leading to an RTE are truncated — they are no longer considered in the verification. On the following line, all executions in which $X = [-2^{31}, -1]$ and $[100, 2^{31}-1]$ are stopped.

- At the next instruction, $X = [0, 99]$.
- Therefore, at the second array access, the check is green because $X = [0, 99]$.

Summary

Green checks can be propagated out of orange checks.

The Message and What It Means

Polyspace software numbers checks to correspond to the code execution order.

Consider the instruction `x++`;

The verification first checks for a potential NIV (Non Initialized Variable) for `x`, and then checks the potential OVFL (overflow). This action mimics the actual execution sequence.

Understanding these sequences can help you understand the message presented by the verification, and what that message means.

Consider an orange NIV on `x` in the test:

```
if (x > 101);
```

You might conclude that the verification does not keep track of the value of `x`. However, consider the context in which the check is made:

```
extern int read_an_input(void);

void main(void)
{
  int x;
  if (read_an_input()) x = 100;
  if (x > 101) // [orange on the NIV : non initialised variable ]
    { x++; } // gray code
}
```

Explanation

You can see the category of each check by clicking it in the Run-Time Checks perspective. When you examine an orange check, you see that any value of a variable that would result in a run-time error (RTE) is not considered further. However, as this example NIV (Non Initialized Variable) shows, any value that does not cause an RTE is verified on subsequent lines.

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 may be different from what you first suspect.

Summary

In summary:

- " $x > 100$ " does **NOT** mean that the verification does not know anything about x .
- " $x > 100$ " **DOES** mean that the verification does not know whether x is initialized.

When you review results, remember:

- Focus on the message provided in the results.
- Do not assume any conclusions.

The Code Explanation

Verification results depend entirely on the code that you are verifying. When interpreting the results, do not consider:

- Any physical action from the environment in which the code operates.
- Any configuration that is not part of the verification.
- Any reason other than the code itself.

The only thing that the verification considers is the code submitted to it.

Consider the following example, paying particular attention to the dead (gray) code following the "if" statement:

```
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
}
```

You can see that:

- The line containing the access to the y array is unreachable.
- Therefore, the test to assess whether $x = 0$ is always false.
- **The initial conclusion is that "the test is always false."** You might conclude that this results from input data that is not equal to 0. However, Read_An_Input can be 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 the array access ($y[x]$) truncates any execution path leading to a run-time error, meaning that subsequent lines deal with only $x = [0, 99]$.
- The orange check on the division also truncates all execution paths that lead to a run-time error, so all instances where $x = 0$ are also stopped. Therefore, for the code execution path after the orange division sign, $x = [1; 99]$.
- x is never equal to 0 **at this line**. The array access is green ($y[x-1]$).

Summary

In this example, all the results are located in the same procedure. However, by using the call tree, you can follow the same process even if an orange check results from a procedure at the end of a long call sequence. Follow the "called by" call tree, **and concentrate on explaining the issues by reference to the code alone.**

Opening Verification Results

In this section...

“Downloading Results from Server to Client” on page 8-8

“Downloading Server Results Using Command Line” on page 8-10

“Downloading Results from Unit-by-Unit Verifications” on page 8-11

“Opening Verification Results from Project Manager Perspective” on page 8-12

“Opening Verification Results from Run-Time Checks Perspective” on page 8-13

“Exploring the Run-Time Checks Perspective” on page 8-14

“Selecting Review Level” on page 8-28

“Searching Results in Run-Time Checks Perspective” on page 8-29

“Setting Character Encoding Preferences” on page 8-30

“Opening Results for Generated Code ” on page 8-32

Downloading Results from Server to Client

When you run a verification on a Polyspace server, the Polyspace software automatically downloads the results to the client system that launched the verification. In addition, the results are stored on the Polyspace server. You can then download the results from the server to other client systems.

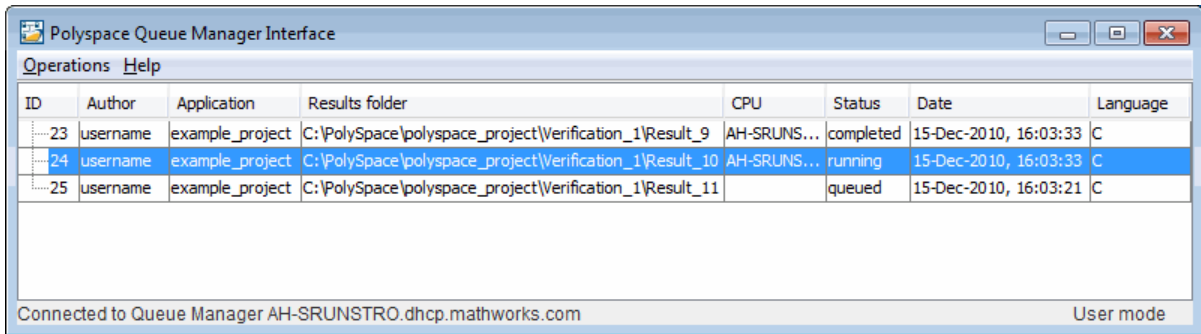
Note If you download results before the verification is complete, you get partial results and the verification continues.

To download verification results from a server to a client system:

- 1 Double-click the **Polyspace Spooler** icon.



The **PolySpace Queue Manager Interface** opens.



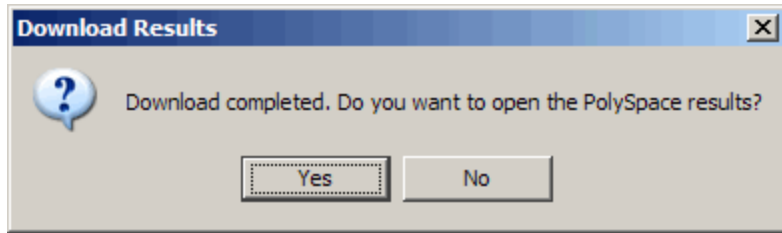
- 2 Right-click the job that you want to view, and select **Download Results** .

Note To remove the job from the queue after downloading your results, from the context menu, select **Download Results And Remove From Queue** .

The Save dialog box opens.

- 3 Select the folder into which you want to download results.
- 4 Click **Save** to download the results and close the dialog box.

When the download is complete, a dialog box opens asking if you want to open the Polyspace results.



5 Click **Yes** to open the results.

Once you download results, they remain on the client, and you can review them at any time using the Polyspace Run-Time Checks perspective.

Downloading Server Results Using Command Line

You can download verification results from the command line using the `psqueue-download` command.

To download your results, enter the following command:

```
Polyspace_Common/RemoteLauncher/bin/psqueue-download <id>  
<results dir>
```

The verification `<id>` is downloaded into the results folder `<results dir>`.

Note If you download results before the verification is complete, you get partial results and the verification continues.

Once you download results, they remain on the client, and you can review them at any time using the Polyspace Run-Time Checks perspective.

The `psqueue-download` command has the following options:

- `[-f]` force download (without interactivity)
- `-admin -p <password>` allows administrator to download results.
- `[-server <name>[:port]]` selects a specific Queue Manager.

- [-v|version] gives release number.

Note When downloading a unit-by-unit verification group, all the unit results are downloaded and a summary of the download status for each unit is displayed.

For more information on managing verification jobs from the command line, see “Managing Verifications in Batch” on page 6-37.

Downloading Results from Unit-by-Unit Verifications

If you run a unit-by-unit verification, each source file is sent to Polyspace Server individually. The queue manager displays a job for the full verification group, as well as jobs for each unit (using a tree structure).

You can download and view verification results for the entire project, or for individual units.

To download the results from unit-by-unit verifications:

- To download results for an individual unit, right-click the job for that unit, then select **Download Results**.

The individual results are downloaded and can be viewed as any other verification results.

- To download results for a verification group, right-click the group job, then select **Download Results**.

The results for all unit verifications are downloaded, as well as an HTML summary of results for the entire verification group.

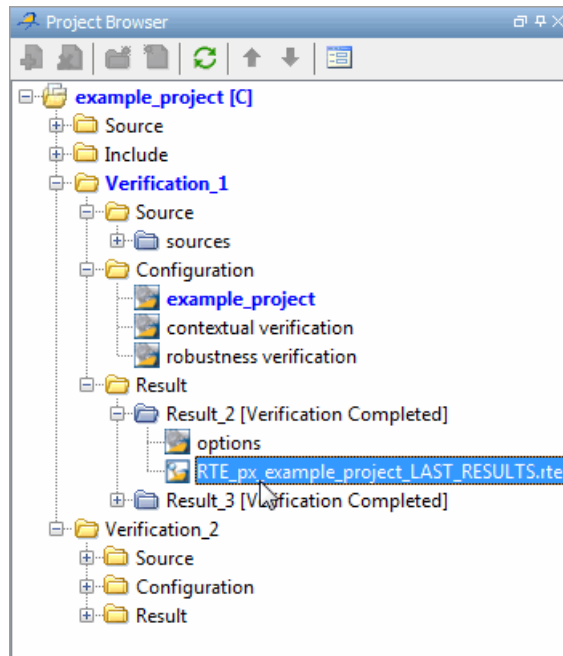
PolySpace Unit By Unit Results Synthesis										
	Green	Orange	Inputs Orange	Dark Orange	Red	Grey	Total	Selectivity	Results	Log file
Source compliance phase results										Open log file
Unit single_file_analysis	97	8	8		2	4	111	93%	Open results	Open log file
Unit main	12	5	3				17	75%	Open results	Open log file
Unit example	99	10	10		5	77	191	95%	Open results	Open log file
Unit tasks2	30	2	2				32	94%	Open results	Open log file
Unit initialisations	52	6				3	61	90%	Open results	Open log file
Unit tasks1	33	5	1				38	87%	Open results	Open log file
	323	36	24	0	7	84	450	92%		

Opening Verification Results from Project Manager Perspective

You can open verification results directly from the Project Browser in the Project Manager perspective. Since each Polyspace project can contain multiple verifications, the Project Browser allows you to quickly identify and open the results you want to review.

To open verification results from the Project Manager:

- 1 Open the project containing the results you want to review.
- 2 In the Project Browser Source tree, navigate to the results you want to review.



3 Double-click the results file.

The results open in the Run-Time Checks perspective.

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

Opening Verification Results from Run-Time Checks Perspective

You use the Run-Time Checks perspective to review verification results. If you know the location of the results file you want to review, you can open it directly from the Run-Time Checks perspective.

Note You can also browse and open results from the Project Browser in the Project Manager perspective.

To open verification results from the Run-Time Checks perspective:

1 Select the **Run Time Checks** button  in the Polyspace Verification Environment toolbar.

2 Select **File > Open Result**

The Please select a file dialog box opens.

3 Select the results file that you want to view.

4 Click **Open**.

The results open in the Run-Time Checks perspective.

Exploring the Run-Time Checks Perspective

- “Overview” on page 8-14
- “Run-Time Checks Pane” on page 8-16
- “Source Pane” on page 8-19
- “Review Statistics Pane” on page 8-23
- “Check Review Pane” on page 8-24
- “Variable Access Pane” on page 8-25
- “Call Hierarchy Pane” on page 8-27

Overview

The Run-Time Checks perspective looks like the following figure.

The screenshot displays the Polyspace IDE interface with the following components:

- Run-Time Checks:** A tree view on the left showing the project structure and the status of various checks. A red icon indicates a failed check (NTC.1).
- Check Review:** A central panel showing details for a specific check, including the source code snippet and a classification table.

Classification	Status	Justified
- Review Statistics:** A table on the right summarizing the overall progress of the review.

Coding review progress	Count	Progr...
Red NTC justified / to justify	0/4	0
Red justified / to justify	0/8	0
Gray justified / to justify	0/6	0
Orange justified / to justify	0/18	0
Software reliability indicator	259/297	87
- Source code:** The central editor showing the C code for the `Recursion_caller` function, with the error location highlighted at line 157, column 5.
- Call Hierarchy:** A tree view on the right showing the sequence of function calls, with the current function highlighted.
- Variable Access:** A panel on the right showing the variables accessed during the execution, including their types and values.

The Run-Time Checks perspective has six sections below the toolbar. Each section provides a different view of the results. The following table describes these views.

This Pane...	Displays...
Run-Time Checks (Procedural entities view)	List of the checks (diagnostics) for each file and function in the project
Source (Source code view)	Source code for a selected check in the procedural entities view
Review Statistics (Coding review progress view)	Statistics about the review progress for checks with the same type and category as the selected check
Check Review (Selected check view)	Details about the selected check
Variable Access (Variables view)	Information about global variables declared in the source code
Call Hierarchy (Call tree view)	Tree structure of function calls

You can resize or hide any of these sections.

Run-Time Checks Pane

The Run-Time Checks pane displays a table with information about the diagnostics for each file in the project. The Run-Time Checks pane is also called the Procedural entities view.






Procedural entities	!	X	?	✓	Line	Col	%	Details
Demo_C (unp: 19/48, cov: 94%)	8	6	20	215				92
-example.c	4	2	8	83	1			92 example.c
-Close_To_Zero ()			3	10	37	12		77 example.c
-Non_Infinite_Loop ()				11	86	11		100 example.c
-Pointer_Arithmetic ()	1	1	1	19	89	12		95 example.c
-RTE ()	1			3	222	5		100 example.c
-IRV.1				✓	1	229	6	Function returns an initialized
-IRV.0				✓	1	231	12	Function returns an initialized
-IRV.3				✓	1	238	6	Function returns an initialized
-NTC.2	1			!		240	6	The called function example.:
-Recursion ()			1	14	137	12		93 example.c
-Recursion_caller ()	1			4	151	12		100 example.c
-Square_Root ()	1			4	185	12		100 example.c
-Square_Root_conv ()				8	179	12		100 example.c
-Unreachable_Code ()		1	1	8	199	12		90 example.c
-get_oil_pressure ()			2	2	21	11		50 example.c
-initialisations.c		1	1	41	1			98 initialisations.c
-main.c	2	1	3	9	1			80 main.c
-single_file_analyse.c	2	2	8	82	1			81 single_file_analyse.c

The checks in the Procedural entities view are colored as follows:


- **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 that never has a run-time error (safe code).

Polyspace software assigns files and functions the color of the most severe error found in that file. For example, the file `example.c` is red because it has a run-time error.

The first column of the table is the procedural entity (the file or function). The following table describes some of the other columns in the procedural entities view.

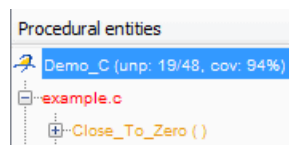
Column Heading	Indicates
	Number of red checks (operations where an error always occurs)
	Number of gray checks (unreachable code)
	Number of orange checks (warnings for operations where an error might occur)
	Number of green checks (operations where an error never occurs)
	Selectivity of the verification (percentage of checks that are not orange) This is an indication of the level of proof.

You can select which columns appear in the procedural entities view by right-clicking the Procedural entities column heading, and selecting the columns you want to display.

Tip If you see three dots in place of a heading, , resize the column until you see the heading. Resize the procedural entities view to see additional columns.

Code Coverage Metrics. The software displays two metrics for the project in the Procedural entities view:

- `unp` — Number of unreachable procedures (functions) as a fraction of the total number of procedures (functions)
- `cov` — Percentage of elementary operations in executable procedures (functions) covered by verification



These metrics provide:

- A measure of the code coverage achieved by the Polyspace verification.
- Indicators about the validity of your Polyspace configuration. For example, a large un_p value and a low cov value may indicate an early red check or missing function call.

Unreachable Functions. If the verification detects functions that cannot be reached from the main program, it does not verify them. The software considers these functions to be unreachable, and highlights them in gray in the procedural entities view.

Demo_C (unp: 19/48, cov: 94%)		8	6	20	215			92
-example.c		4	2	8	83	1		92
-Close_To_Zero ()				3	10	37	12	77
-Non_Infinite_Loop ()					11	66	11	100
-Pointer_Arithmetic ()		1	1	1	19	89	12	95
-RTE ()		1			3	222	5	100
-IRV.1	✓				1	229		6
-IRV.0	✓				1	231	12	
-IRV.3	✓				1	238		6
-NTC.2	✗	1				240		6
-Recursion ()				1	14	137	12	93
-Recursion_caller ()		1			4	151	12	100
-Square_Root ()		1			4	185	12	100
-Square_Root_conv ()					8	179	12	100
-Unreachable_Code ()			1	1	8	199	12	90
-get_oil_pressure ()					2	2	11	50

In this example, the function `Unreachable_Code` is considered unreachable, and therefore is not verified.

Source Pane

The Source pane shows the source code with colored checks highlighted. The Source Pane is also called the Source code view.

The screenshot shows a source code editor window titled "Source" with a file named "example.c". The code is as follows:

```

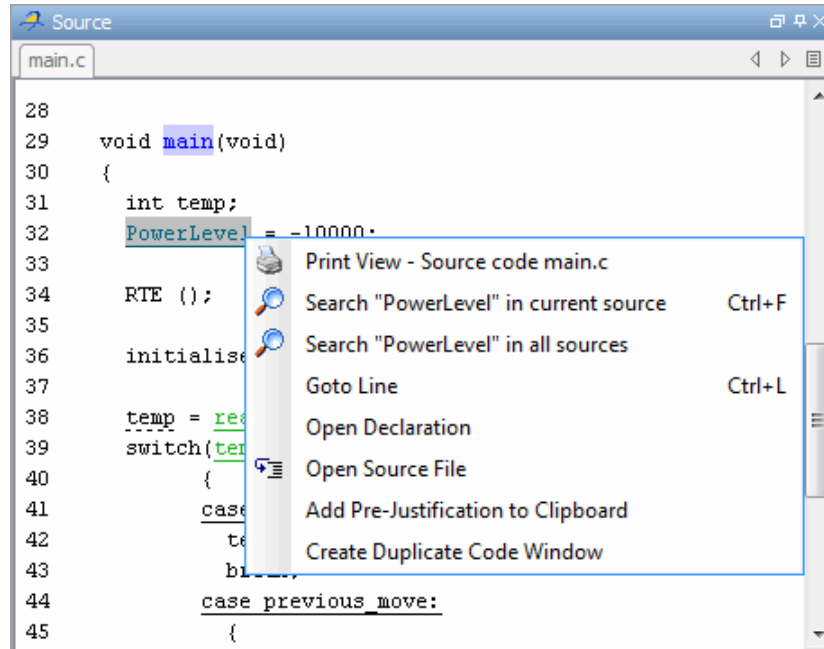
137 static void Recursion (int* depth)
138     /* if depth<0, recursion will lead to division by zero */
139 {   float advance;
140
141     *depth = *depth + 1;
142     advance = 1.0f/(float)(*depth); /* potential division by zero */
143
144
145     if (*depth <= 50)
146     {
147         Recursion(depth);
148     }
149 }
150
151 static void Recursion_caller(void)
152 {   int x=random_int();
153
154     if ((x>=4) && (x <= -1))
155     {
156         Recursion( &x ); // always encounters a division by zero
157     }
158 }

```

A tooltip is displayed over the `random_int()` function call on line 152. The tooltip text is: "returned value of random_int (int 32): full-range [-2³¹ .. 2³¹-1]".

Tooltips. Placing your cursor over a check displays a tooltip that provides ranges for variables, operands, function parameters, and return values. For more information on tooltips, see “Using Range Information in Run-Time Checks Perspective” on page 8-83.

Examining Source Code. In the Source pane, if you right-click a text string, the context menu provides options that help you to examine your code. For example, right-click the global variable `PowerLevel`:

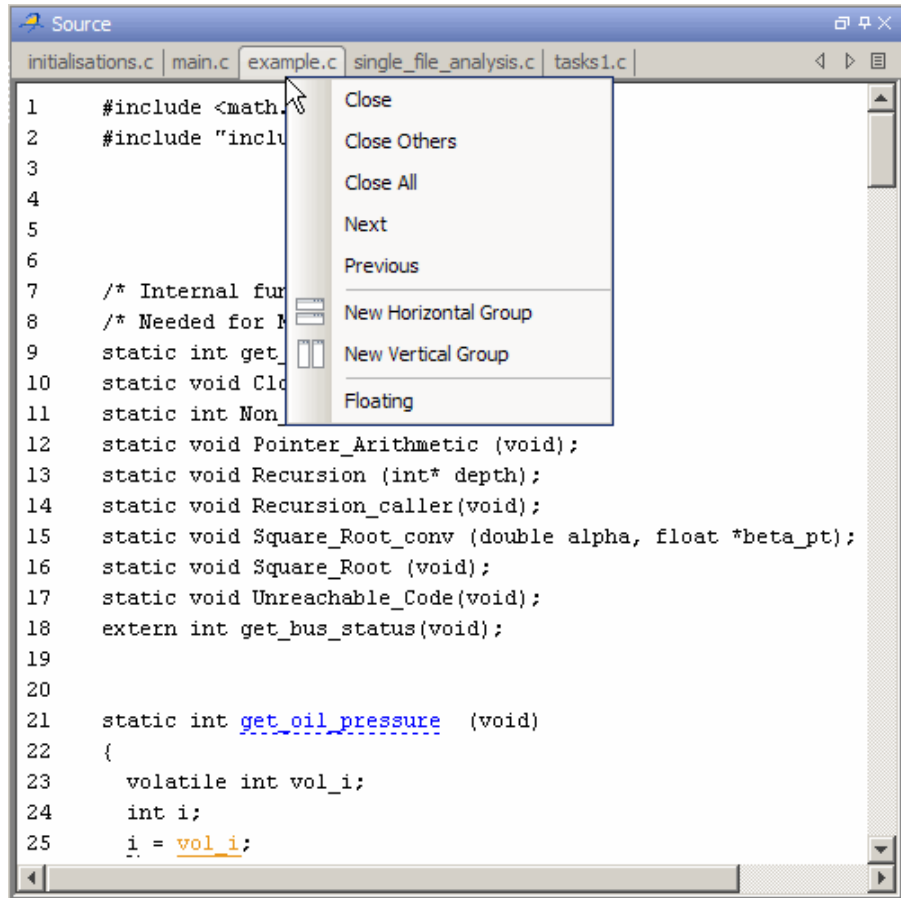


Use the following options to examine and navigate through your code:

- **Search “PowerLevel” in current source code** — List all occurrences of the string in the Search pane.
- **Goto Line** — Open the Goto Line dialog box. If you specify a line number and click **Enter**, the software displays the specified line of code.
- **Open Declaration** — If the selected text is a global variable or function, display the line of code that contains the declaration.
- **Open Source File** — Open the source file with your text editor.

Managing Multiple Files in Source Pane. You can view multiple source files in the Source pane. By default, the files are displayed as tabs in the Source pane.

Right-click any tab in the Source pane toolbar to manage source files.



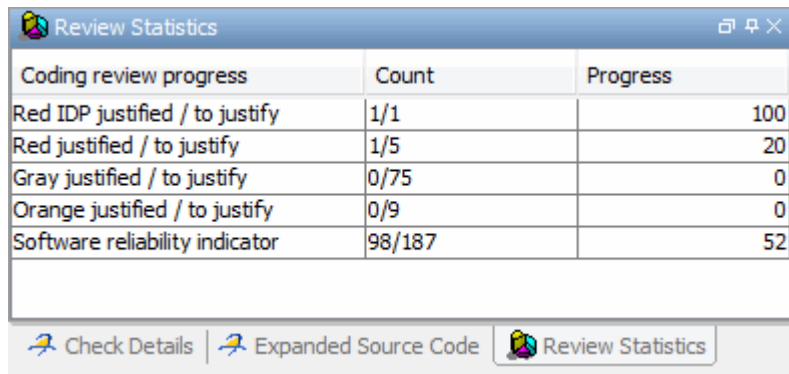
From the Source pane context menu, you can:

- **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 tab.
- **Previous** – Display the previous tab.

- **New Horizontal Group** – Split the Source window horizontally to display the selected source file below another file.
- **New Vertical Group** – Split the Source window vertically to display the selected source file side-by-side with another file.
- **Floating** – Display the current source file in a new window, outside the Source pane.

Review Statistics Pane

The Review Statistics pane displays statistics about how many checks you have reviewed. As you review checks, the software updates these statistics. The Review Statistics pane is also called the Coding review progress view.



Coding review progress	Count	Progress
Red IDP justified / to justify	1/1	100
Red justified / to justify	1/5	20
Gray justified / to justify	0/75	0
Orange justified / to justify	0/9	0
Software reliability indicator	98/187	52

Check Details | Expanded Source Code | Review Statistics

The **Count** column displays a ratio and the **Progress** column displays the equivalent percentage.

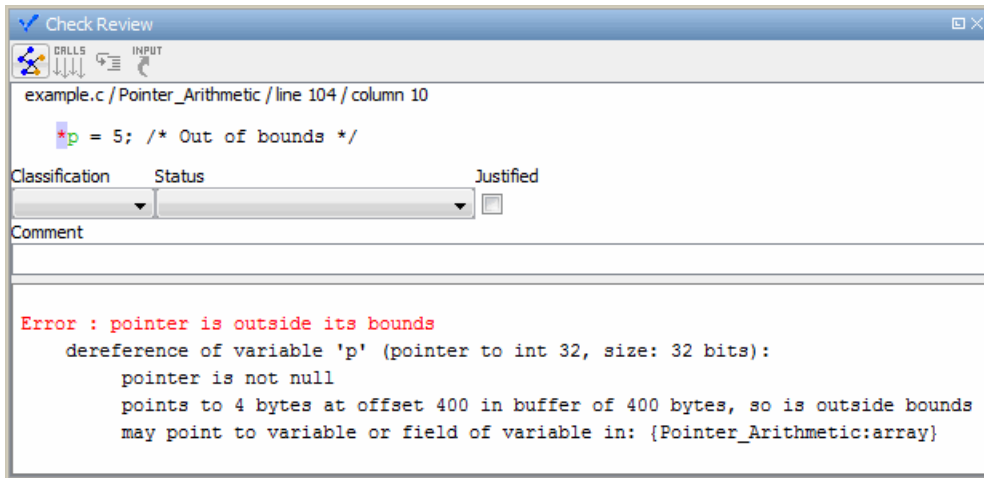
The first row displays the ratio of justified checks to total checks that have the same color and category of the current check. In this example, the first row displays the ratio of reviewed red IDP checks to total red IDP errors in the project.

The second row displays the ratio of justified checks to total checks that have the color of the current check. In this example, this is the ratio of red errors reviewed to total red errors in the project.

The last row displays the ratio of the number of green checks to the total number of checks, providing an indicator of the reliability of the software.


Check Review Pane

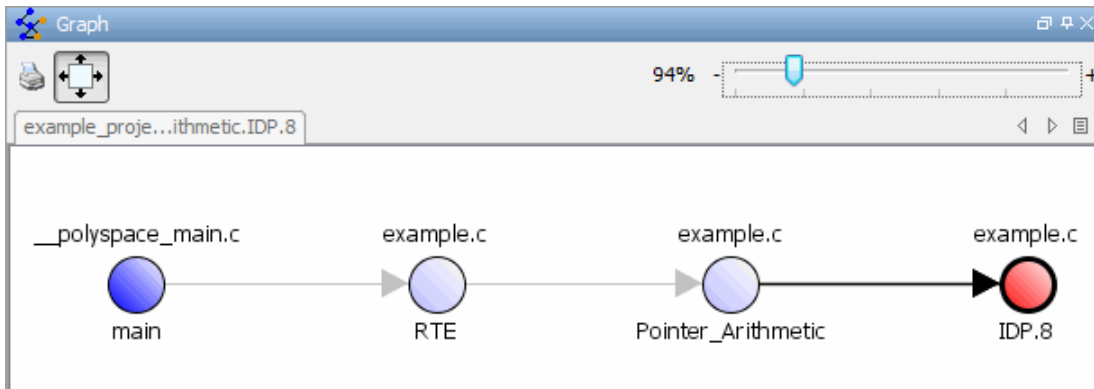
The Check Review Pane displays information about the current check. The Check Review pane is also called the Selected check view.



When reviewing checks, you use the Selected check view to mark checks as **Justified**, and enter comments to describe the results of your review. This helps you track the progress of your review and avoid reviewing the same check twice.

For more information, see “Reviewing and Commenting Checks” on page 8-56.

Error Call Graph. Click the **Show error call graph** icon  in the Check Review pane toolbar to display the call sequence that leads to the code associated with a check.




For more information, see “Displaying the Call Sequence for a Check” on page 8-48.

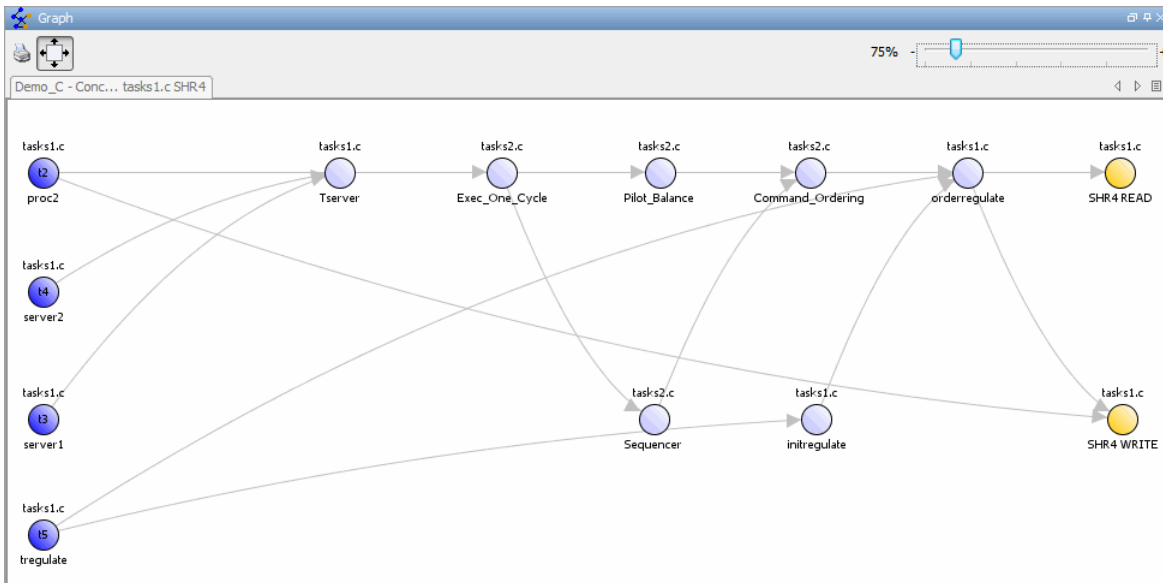
Variable Access Pane

The Variable Access pane, which is also called the Variables view, displays global variables. The pane shows where in the source code the variables are read or written to, and provides:


- Information about the variables and associated fields. For example, number of read/write access operations, data type, and value range. For global variables that are integers (signed and unsigned) or floating point variables (`float` and `double`), the software provides range information for both the individual read/write access operations within a file and the union of these operations.
- A hierarchical view of structured variables.


Variables	Detailed Type	Values	# Read	# Write	W.T.	R.T.	Protection	Usage	Scalar	Line	Col	File	Type
single_file_analysis.output_v7	int 32	[253..555]	3	2						18	11	single_file_analysis.c	
single_file_analysis_init_globals		0								18	11	single_file_analysis.c	
single_file_analysis_generic_validation		[253..555]								107	2	single_file_analysis.c	
single_file_analysis_generic_validation		[253..555]								108	22	single_file_analysis.c	
single_file_analysis_generic_validation		[253..555]								128	6	single_file_analysis.c	
single_file_analysis_generic_validation		[0..555]								130	19	single_file_analysis.c	
single_file_analysis.saved_values	array(0..126) of int 16		0	2						21	11	single_file_analysis.c	
single_file_analysis_init_globals										21	11	single_file_analysis.c	
single_file_analysis_generic_validation										130	6	single_file_analysis.c	
single_file_analysis.v0	unsigned int 16	[0..20624]	1	2						9	11	single_file_analysis.c	
single_file_analysis_init_globals		0								9	11	single_file_analysis.c	
single_file_analysis_functional_ranges		[0..20624]								40	2	single_file_analysis.c	
single_file_analysis_generic_validation		[0..20624]								83	23	single_file_analysis.c	
single_file_analysis.v1	int 16	[0..23040]	3	2						10	11	single_file_analysis.c	
single_file_analysis.v2	int 16	[25920..48000]	1	2						11	11	single_file_analysis.c	
single_file_analysis.v3	unsigned int 8	[0..216]	2	2						12	10	single_file_analysis.c	

Concurrent Access Graph. Click the Show Access Graph button  in the Variable Access pane toolbar to display a graph of read and write access for the selected variable.

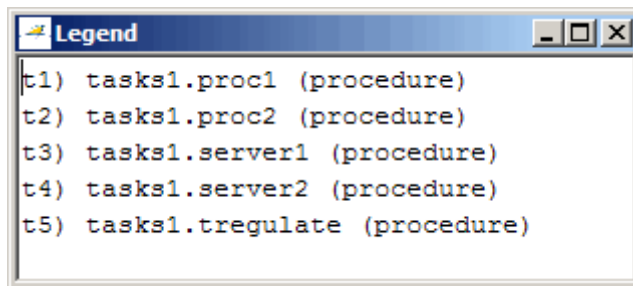


For more information, see “Displaying the Access Graph for Variables” on page 8-49.

Non Shared Variables. Click the Non-Shared Variables button  in the Variable Access pane toolbar to show or hide non-shared variables.

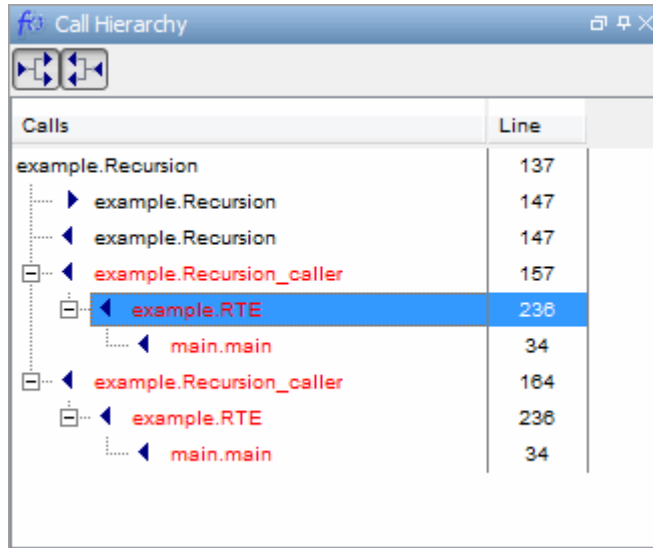
Read and Write Access in Dead Code. If a read or write access operation on a global variable lies within dead code, then Polyspace colors the operation gray in the **Variable Access** pane. When you examine verification results, you can hide these operations by clicking the filter button .

Legend Information. To display the legend for a variable, right-click the variable and select **Show legend**.

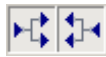


Call Hierarchy Pane

The Call Hierarchy pane displays the call tree of functions in the source code. You can use the call tree view to easily navigate up and down the call tree. The Call Hierarchy pane is also called the Call Tree view.



Calls	Line
example.Recursion	137
▶ example.Recursion	147
◀ example.Recursion	147
◀ example.Recursion_caller	157
◀ example.RTE	236
◀ main.main	34
◀ example.Recursion_caller	164
◀ example.RTE	236
◀ main.main	34

Callers and Callees. Click the buttons  in the Call Tree View toolbar to show or hide callers and callees.

Function Definitions. To go directly to the definition of a function, right-click the function call and select **Go to definition**.

Selecting Review Level

Use the slider on the Run-Time Checks toolbar to specify the review level.



You can review results from five different levels:

- 0 — Display red and gray checks (default). You can also display orange checks that are potential run-time errors. On the **Polyspace Preferences > Review Configuration** tab, specify the type of potential run-time errors that you are interested in. See “Reviewing Checks at Level 0” on page 8-35.

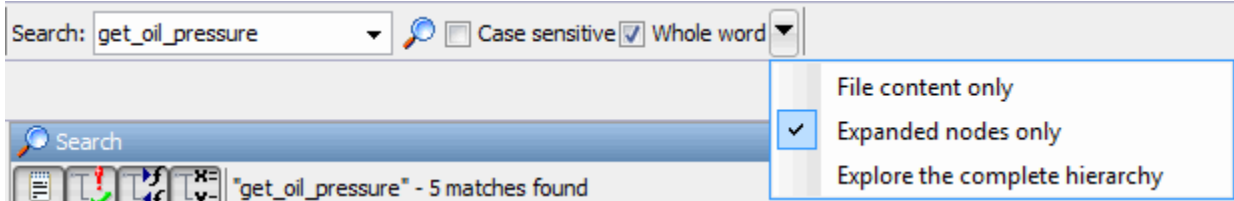
- 1, 2, and 3 — Display red, gray, and green checks. In addition, display orange checks according to values specified on the **Polyspace Preferences > Review Configuration** tab. You can use either a predefined or custom methodology to specify the number of orange checks per category. See “Viewing Methodology Requirements for Levels 1, 2, and 3” on page 8-38 and “Defining a Custom Methodology for Levels 1, 2, and 3” on page 8-39.
- All — Display all checks.

The default setting is 1.

Searching Results in Run-Time Checks Perspective

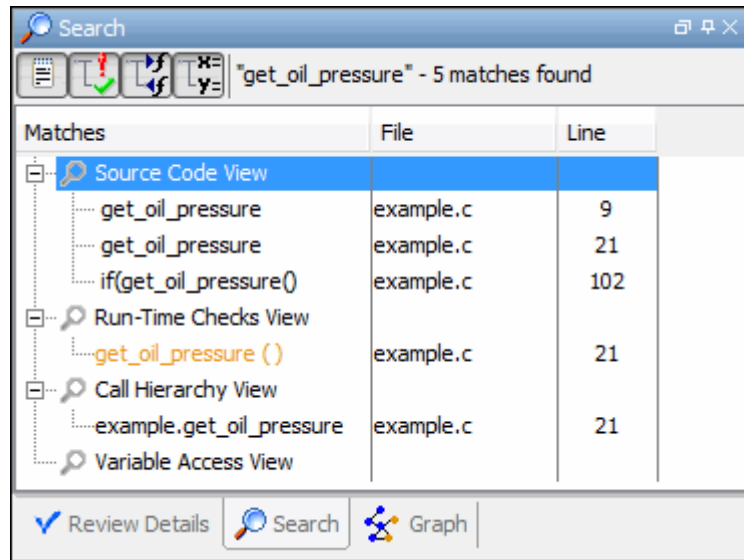
You can search your results and source code using the Search feature in the Run-Time Checks perspective toolbar.

The Search toolbar allows you to quickly enter search terms, specify search options, and set the scope for your search.



You can limit the scope of your search to only file content, only expanded nodes, or you can search the complete hierarchy.

When you perform a search, your search results are reported in the Search pane.



Search results are organized by location:

- Source Code View
- Run-Time Checks View
- Call Hierarchy View
- Variable Access View

You can use the four filter buttons in the Search pane toolbar to hide results from any of these locations.

Setting 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.

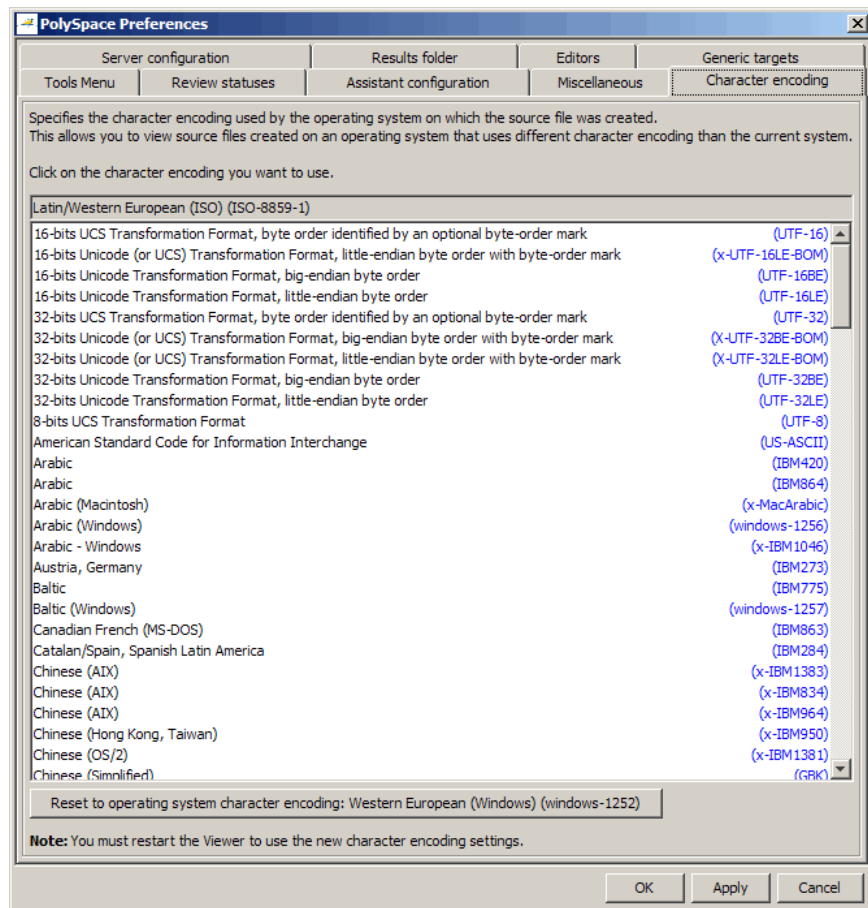
The **Character encoding** option allows you to view source files created on an operating system that uses different character encoding than your current system.

To set the character encoding for a source file:

1 Select Options > Preferences .

The **Polyspace Preferences** dialog box opens.

2 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.

Opening 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 Run-Time checks perspective.

If you launched the verification using the Polyspace Model Link SL or Polyspace Model Link TL products, the software automatically creates a file in the results folder called `code_generator_used.txt` to provide this information. However, if you did not use these products to launch verification, you must provide this information manually.

To manually specify which code generator created the code:

- 1 Open your results in the Run-Time Checks perspective.
- 2 Select **Review > Code Generator Support > *code_generator***

Manually Creating 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 will then automatically use this file each time you open the results.

The format of this file is the following:

```
<Code generator>
MATLABROOT=<Path to MATLAB>
ModelVersion=<model name>:<model version>
```

where `<Code generator>` can be either `RTWEmbeddedCoder` or `TargetLink`.

For Example:

```
RTWEmbeddedCoder
MATLABROOT=C:\MATLAB\R2010b
ModelVersion=demo_ml:1.94
```

Reviewing Results Systematically

In this section...

“What are Review Levels?” on page 8-34

“Reviewing Checks at Level 0” on page 8-35

“Reviewing Checks at Levels 1, 2, and 3” on page 8-36

“Viewing Methodology Requirements for Levels 1, 2, and 3” on page 8-38

“Defining a Custom Methodology for Levels 1, 2, and 3” on page 8-39

“Reviewing Checks Progressively” on page 8-41

“Saving Review Comments” on page 8-43

What are Review Levels?

To facilitate your review of verification results, Polyspace allows you to control the type and number of orange checks displayed in the **Procedural entities** and **Source** views of the Run-Time Checks perspective. There are five levels at which you can review your results:

- 0 — The software displays red and gray checks. In addition, you can configure the software to displays orange checks that are potential run-time errors. Through the **Polyspace Preferences > Review Configuration** tab, specify the categories of potential run-time errors that you want the software to display. By default, the software does not display any orange checks at this level. See “Reviewing Checks at Level 0” on page 8-35.

This level is suitable for the review of fresh code.

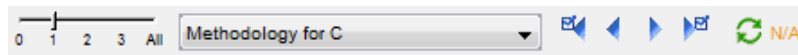
- 1, 2, and 3 — The software displays red, gray, and green checks. In addition, the software displays orange checks according to values specified on the **Polyspace Preferences > Review Configuration**. You can use either a predefined methodology or a custom methodology to specify the number of orange checks per check category. See “Viewing Methodology Requirements for Levels 1, 2, and 3” on page 8-38 and “Defining a Custom Methodology for Levels 1, 2, and 3” on page 8-39.

For a predefined methodology, these levels are suitable for reviews at the following stages of the development process.

Level	Development Stage
1	Fresh code
2	Unit tested code
3	Code Review

- All — In addition to red, gray, and green checks, the software displays *all* orange checks. Use this level when you want to carry out an exhaustive review of your verification results.

The toolbar in the Run-Time Checks perspective provides controls specific to review levels.



The controls include:

- A slider for selecting the review level. By default, the Run-Times Checks perspective opens at level 1.
- A menu for selecting the review methodology for levels 1, 2, and 3.
- Arrows for navigating through checks.

Reviewing Checks at Level 0

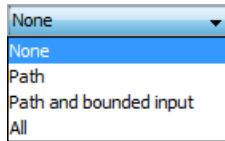
At this level, in addition to red and gray checks, you can focus on orange checks that Polyspace identifies as potential run-time errors. These potential run-time errors fall into three categories:

- Path – The software identifies orange checks that are path-related issues, which are not dependent on input values.
- Path and bounded input – In addition to orange checks that are path-related issues, the software identifies orange checks that are related to bounded input values.

- All – In addition to path-related and bounded input orange checks, the software identifies orange checks that are related to unbounded input values.

To specify the potential run-time error category for level 0:

- 1** In the Polyspace verification environment, select **Options > Preferences**. The Polyspace Preferences dialog box opens.
- 2** Select the **Review configuration** tab.
- 3** From the **Level** drop-down list, select your category.



The default is None, that is, the software displays only red and gray checks.

- 4** Click **OK** to save your options and close the Polyspace Preferences dialog box.

To select review level 0, in the Run-Time Checks toolbar, move the Review Level slider to **0**.

Reviewing Checks at Levels 1, 2, and 3

In addition to red, gray, and green checks, the software displays orange checks according to values specified on the **Review Configuration** tab in the Polyspace Preferences dialog box.

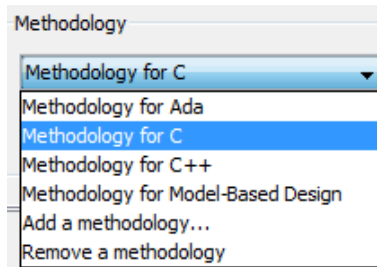
Levels 1, 2, and 3			
	Level 1	Level 2	Level 3
Common			
ZDV	5	20	ALL
NIVL	10	50	ALL
S-OVFL	10	50	ALL
COR		10	10
NIV		0	10
F-OVFL	5	10	20
ASRT		5	20
C & C++ only			
OBAI	10	20	ALL
SHF	5	10	ALL
IDP		10	20
NIP		10	20
STD_LIB			
C only			
IRV	5	20	ALL
C++ only			
NNT			
CPP			
FRV			
OOP			
EXC			
Ada only			
EXCP			
POW			

You can use either a predefined methodology or a custom methodology to specify the number of orange checks per check category. See “Defining a Custom Methodology for Levels 1, 2, and 3” on page 8-39.

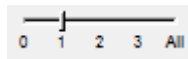
Each predefined methodology has three levels, with increasing review requirements. As the level increases, you review more checks.

To select, for example, a predefined methodology and level:

- 1 From the Run-Time Checks perspective, select **Options > Preferences**. The Polyspace Preferences dialog box opens.
- 2 Select the **Review configuration** tab.
- 3 From the **Methodology** drop-down list, select Methodology for C or Methodology for C++.



- 4 Use the Review Level slider to select the appropriate level.



Viewing Methodology Requirements for Levels 1, 2, and 3

A methodology specifies the orange checks that you review, and you can view the requirements of the methodology through the Polyspace Preferences dialog box.

Note You cannot change the parameters specified in predefined methodologies, for example, Methodology for C, but you can create your own custom methodologies. See “Defining a Custom Methodology for Levels 1, 2, and 3” on page 8-39.

To examine the configuration for Methodology for C:

- 1 In the Polyspace verification environment, select **Options > Preferences**.
The Polyspace Preferences dialog box opens.

- 2 Select the **Review configuration** tab.
- 3 From the **Methodology** drop-down list, select **Methodology for C**.

In the section **Levels 1, 2, and 3**, a table shows the number of orange checks that you review for a given level and check category.

Levels 1, 2, and 3			
	Level 1	Level 2	Level 3
Common			
ZDV	5	20	ALL
NIVL	10	50	ALL
S-OVFL	10	50	ALL
COR		10	10
NIV		0	10
F-OVFL	5	10	20
ASRT		5	20
C & C++ only			
OBAI	10	20	ALL
SHF	5	10	ALL
IDP		10	20
NIP		10	20
STD_LIB			
C only			
IRV	5	20	ALL

For example, the table specifies that you review five orange ZDV checks when you select level 1. The number of checks increases as you move from level 1 to level 3, reflecting the changing review requirements as you move through the development process.

- 4 Click **OK** to close the dialog box.

Defining a Custom Methodology for Levels 1, 2, and 3

A methodology specifies the orange checks that you review. For review levels 1, 2, and 3, you cannot change the predefined methodologies, for example, **Methodology for C**, but you can define your own methodology.

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. This percentage is given by:

$(\text{green checks} + \text{reviewed orange checks}) \times 100 / (\text{green checks} + \text{total orange checks})$

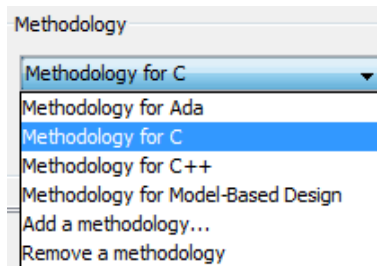
To define a custom methodology:

- 1 In the Polyspace verification environment, select **Options > Preferences**.

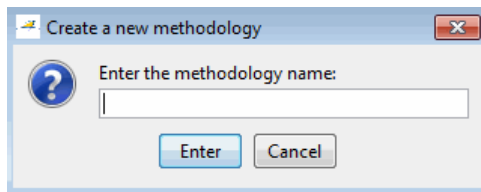
The **Polyspace Preferences** dialog box opens.

- 2 Select the **Review configuration** tab.

- 3 From the **Methodology** drop-down list, select **Add a methodology**.



The Create a new methodology dialog box opens.



- 4 Enter a name for the new methodology, for example, `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.

- 6 Enter the total number of checks (or percentage of checks) to review for each type of check at levels 1, 2, and 3.

Levels 1, 2, and 3			
	Level 1	Level 2	Level 3
Common			
ZDV	10	30	100
NIVL	10	30	100
S-OVFL	0	30	100
COR			
NIV			
F-OVFL			
ASRT			
C & C++ only			
OBAI	5	10	100
SHF	0	10	100
IDP			
NIP			
STD_LIB			
C only			
IRV			

7 Click **OK** to save the methodology and close the dialog box.

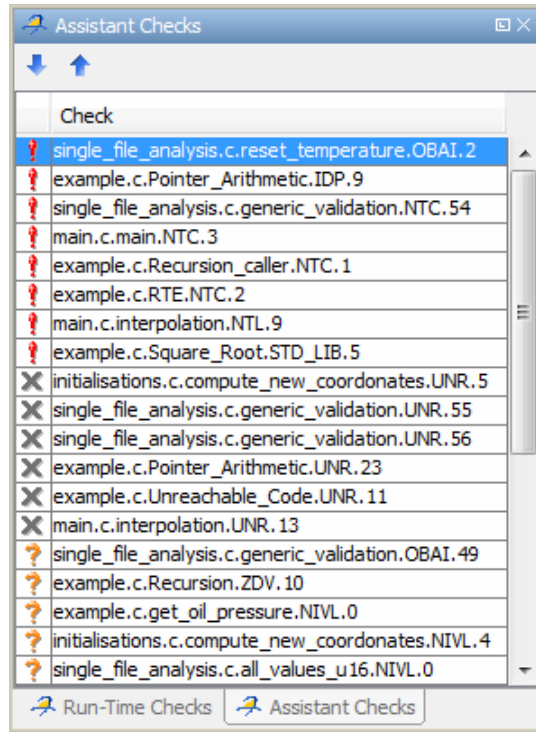
Reviewing Checks Progressively


On the Run-Time Checks perspective toolbar, use the forward arrow  or back arrow  to move to the next or previous unjustified check. The software takes you through checks in the following order:

- All red checks
- All gray checks (the first check in each unreachable function). If you want to skip gray checks when using the forward or back arrows, on the **Polyspace Preferences > Review Configuration** tab, select **Skip gray checks review**.
- Orange checks — the number of orange checks is determined by the methodology and review level that you select

To review checks:

- 1 Select the **Assistant Checks** tab.



- 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 Review pane displays information about this check.

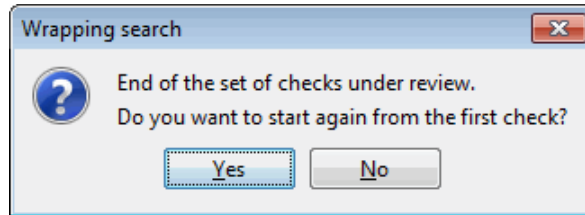
Note You can also display the call sequence for a check. See “Displaying the Call Sequence for a Check” on page 8-48.

- 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. For more information, see “Tracking Review Progress” on page 8-55.

- 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 .

Saving Review Comments

After you have reviewed your results, you can save your comments with the verification results. Saving your comments makes them available the next time you open the results file, allowing you to avoid reviewing the same check twice.

To save your review comments:

- 1 Select **File > Save**.

Your comments are saved with the verification results.

Note Saving review comments also allows you to import those comments into subsequent verifications of the same module, allowing you to avoid reviewing the same check twice.

Reviewing All Checks

In this section...
<p>“Selecting a Check to Review” on page 8-44</p> <p>“Displaying the Call Sequence for a Check” on page 8-48</p> <p>“Displaying the Access Graph for Variables” on page 8-49</p> <p>“Filtering Checks” on page 8-50</p> <p>“Saving Review Comments” on page 8-53</p>

Selecting a Check to Review

To display all checks, on the Run-Time Checks perspective toolbar, move the Review Level slider to **All**:

To review a check:

- 1 In the procedural entities section of the Run-Time Checks pane, expand any file containing checks.
- 2 Expand the procedure containing the check that you want to review.

You see a color-coded list of the checks:

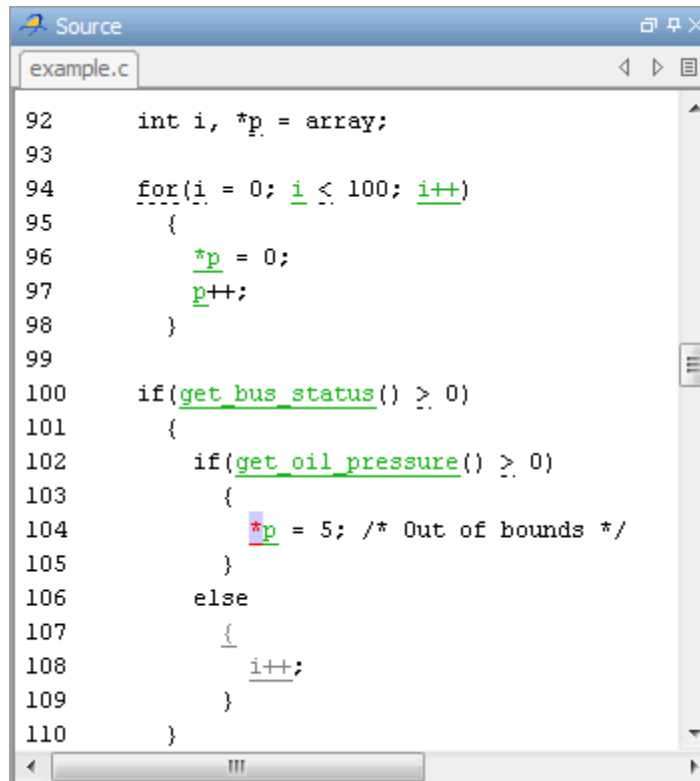
	1	2	1	6	89	12	90	example.c
✓ OVFL.2				1	94	23		Operation [+] on scalar does not ...
✓ IDP.3				1	96	6		Pointer is within its bounds
✓ IRV.6				1	100	5		Function returns an initialized v...
✓ IRV.7				1	102	9		Function returns an initialized v...
! IDP.8	1				104	10		Error : pointer is outside its bounds
✗ UNR.10		1			107	8		Unreachable code
✗ OVFL.12		1			108	11		Unreachable check : overflow o...
✓ IRV.13				1	112	6		Function returns an initialized v...
? IDP.15			1		114	16		Warning : pointer may be outsid...
✓ IDP.22				1	119	6		Pointer is within its bounds

Each item in the list of checks has an acronym that identifies the type of check and a number. For example, IDP.8, IDP stands for Illegal Dereferenced Pointer.

For more information about different types of checks, see “Check Descriptions for C Code” in the *Polyspace Products for C/C++ Reference*.

- 3 Click the check that you want to review.

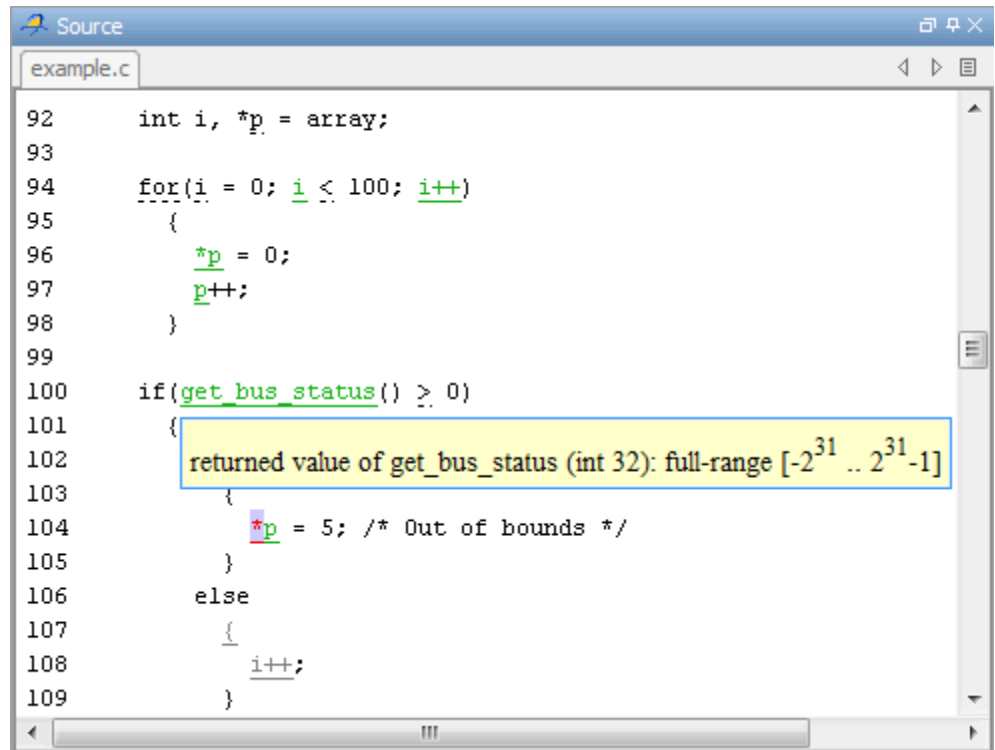
The Source pane displays the section of source code where this error occurs.



```
92     int i, *p = array;
93
94     for(i = 0; i < 100; i++)
95     {
96         *p = 0;
97         p++;
98     }
99
100    if(get_bus_status() >= 0)
101    {
102        if(get_oil_pressure() >= 0)
103        {
104            *p = 5; /* Out of bounds */
105        }
106        else
107        {
108            i++;
109        }
110    }
```

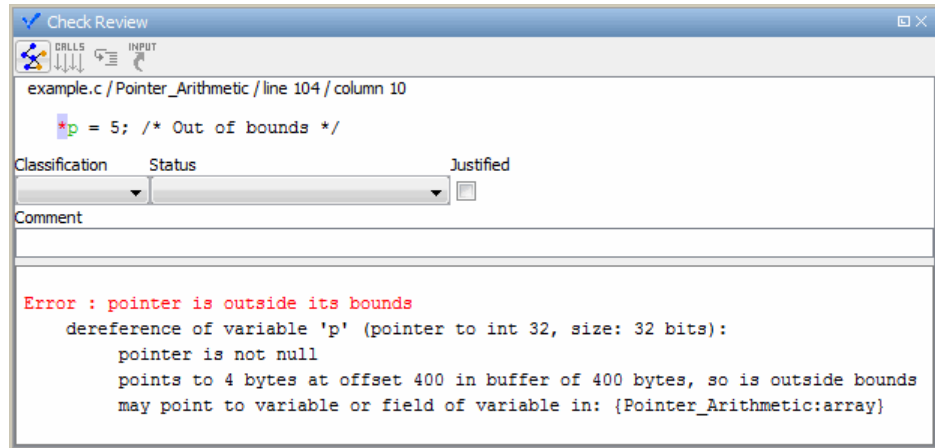
- 4 Place your cursor over any colored check in the code.

A tooltip provides ranges for variables, operands, function parameters, and return values. For more information on these tooltips, see “Using Range Information in Run-Time Checks Perspective” on page 8-83.



```
92     int i, *p = array;
93
94     for(i = 0; i < 100; i++)
95     {
96         *p = 0;
97         p++;
98     }
99
100    if(get_bus_status() >= 0)
101    {
102        returned value of get_bus_status (int 32): full-range [-231 .. 231-1]
103    {
104        *p = 5; /* Out of bounds */
105    }
106    else
107    {
108        i++;
109    }
```


- 5 In the code, click the red check. You see details about the check in the Check Review pane.



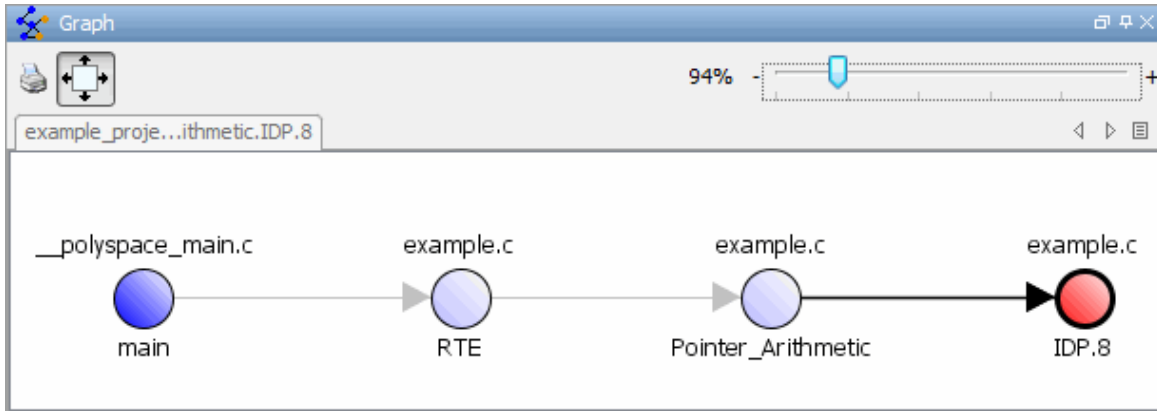
You can enter comments to describe the results of your review. You can also mark the check as Justified to help track your review progress. For more information, see “Tracking Review Progress” on page 8-55.

Displaying the Call Sequence for a Check

You can display the call sequence that leads to the code associated with a check. To see the call sequence for a check:

- 1 In the procedural entities window, expand the procedure containing the check that you want to review.
- 2 Select the check that you want to review.
- 3 In the Check Review pane toolbar, click the error call graph button. 

A window displays the call graph.



The call graph displays the code associated with the check.

Displaying the Access Graph for Variables

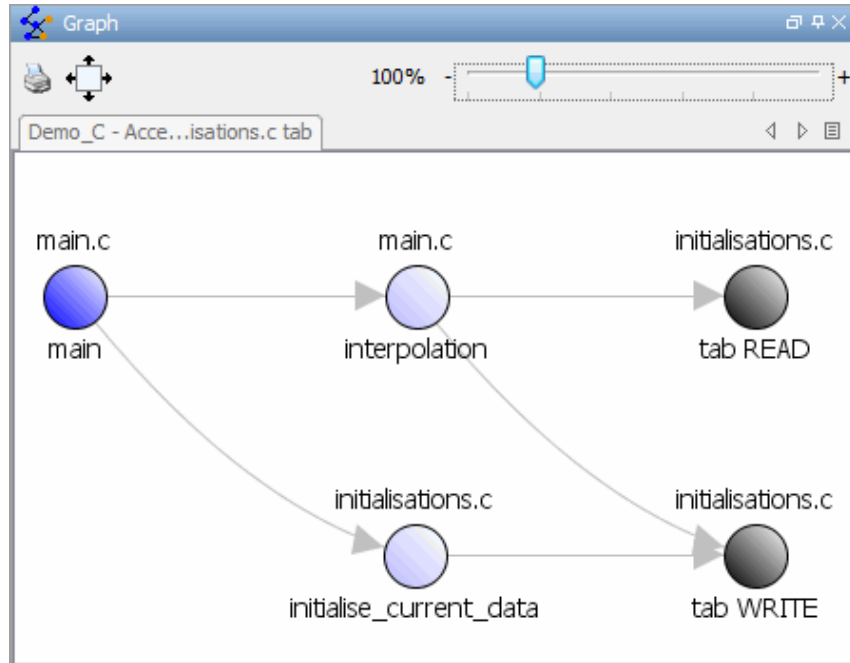
You can display the access sequence for any variable that is read or written in the code.

To see the access graph:

- 1 Select the Variables View.
- 2 Select the variable that you want to view.
- 3 In 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 access for the variable.

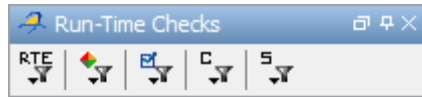
- 4 Click any object in the graph to navigate to that function in the Procedural entities view and Source code view.

Filtering Checks

You can filter the checks that you see in the Run-Time Checks perspective so that you can focus on certain checks. Polyspace software allows you to filter your results in several ways. You can filter by:

- Check category (ZDV, IDP, NIP, etc.)
- Color of check (gray, orange, green)
- Justified or unjustified
- Classification
- Status

To filter checks, select one of the filter buttons in the Run-Time checks toolbar.



Tip The tooltip for a filter button describes what filter the button activates.

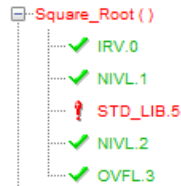
Example: Filtering IRV Checks


You can use an RTE filter to hide a given check category, such as IRV. When a filter is enabled, you do not see that check category.

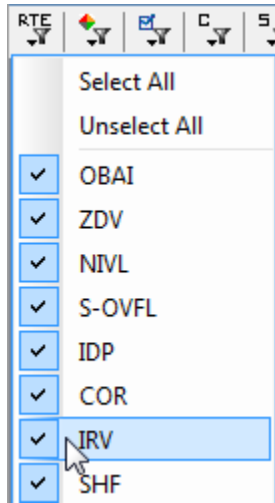
To filter IRV checks in the `Square_Root()` procedure:

- 1 Expand `Square_Root()`.

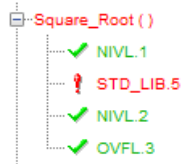
`Square_Root()` has five checks: four are green and one is red.



- 2 Click the **RTE filter** icon .
- 3 Clear the **IRV** option.



The software hides the IRV check for `Square_Root()`.



4 Select the IRV option to redisplay the IRV check.

Note When you filter a check category, red checks of that category are not hidden.

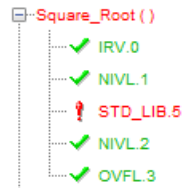
Example: Filtering Green Checks

You can use a Color filter to hide certain color checks. When a filter is enabled, you do not see that color check.

To filter green checks in the `Square_Root()` procedure:

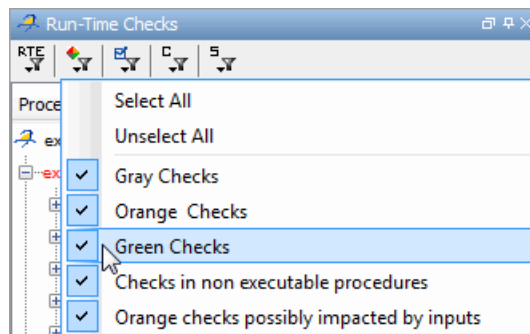
1 Expand `Square_Root()`.

Square_Root () has five checks: four are green and one is red.



2 Click the **Color filter** icon .

3 Clear the **Green Checks** option.



The software hides the green checks.



Saving Review Comments

After you have reviewed your results, you can save your comments with the verification results. Saving your comments makes them available the next time you open the results file, allowing you to avoid reviewing the same check twice.

To save your review comments:

1 Select **File > Save**.

Your comments are saved with the verification results.

Note Saving review comments also allows you to import those comments into subsequent verifications of the same module, allowing you to avoid reviewing the same check twice.

Tracking Review Progress

In this section...

“Checking Coding Review Progress ” on page 8-55

“Reviewing and Commenting Checks ” on page 8-56

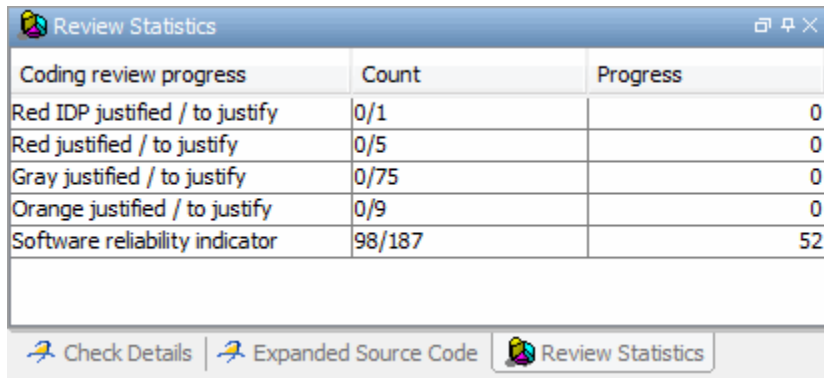
“Defining Custom Status ” on page 8-58

“Tracking Justified Checks in Procedural Entities View” on page 8-60

“Commenting Code to Justify Known Checks” on page 8-61

Checking Coding Review Progress

When you select a check in either Assistant or Manual mode, the Review Statistics pane displays statistics about the review progress for that category and severity of error.



Coding review progress	Count	Progress
Red IDP justified / to justify	0/1	0
Red justified / to justify	0/5	0
Gray justified / to justify	0/75	0
Orange justified / to justify	0/9	0
Software reliability indicator	98/187	52

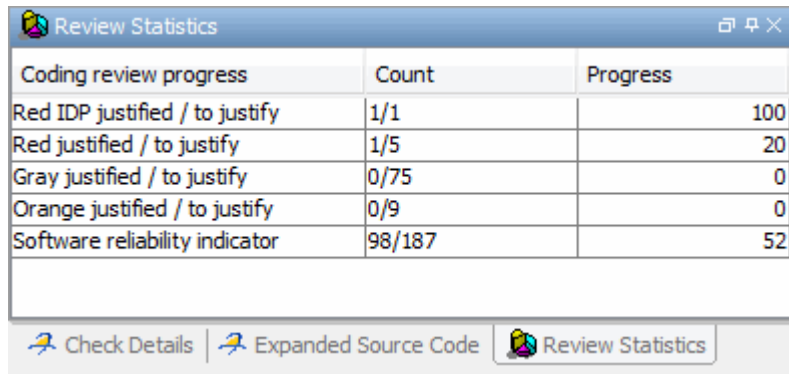
The **Count** column displays a ratio and the **Progress** column displays the equivalent percentage.

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 example, the first row displays the ratio of reviewed red IDP checks to total red IDP errors in the project.

The second row displays the ratio of justified checks to total checks that have the color of the current check. In this example, this is the ratio of red errors reviewed to total red errors in the project.

The last row displays the ratio of the number of green checks to the total number of checks, providing an indicator of the reliability of the software.

When you select the **Justified** checkbox for the check, the software updates the ratios of errors reviewed to total errors in the **Coding review progress** part of the window.



Coding review progress	Count	Progress
Red IDP justified / to justify	1/1	100
Red justified / to justify	1/5	20
Gray justified / to justify	0/75	0
Orange justified / to justify	0/9	0
Software reliability indicator	98/187	52

Check Details | Expanded Source Code | Review Statistics

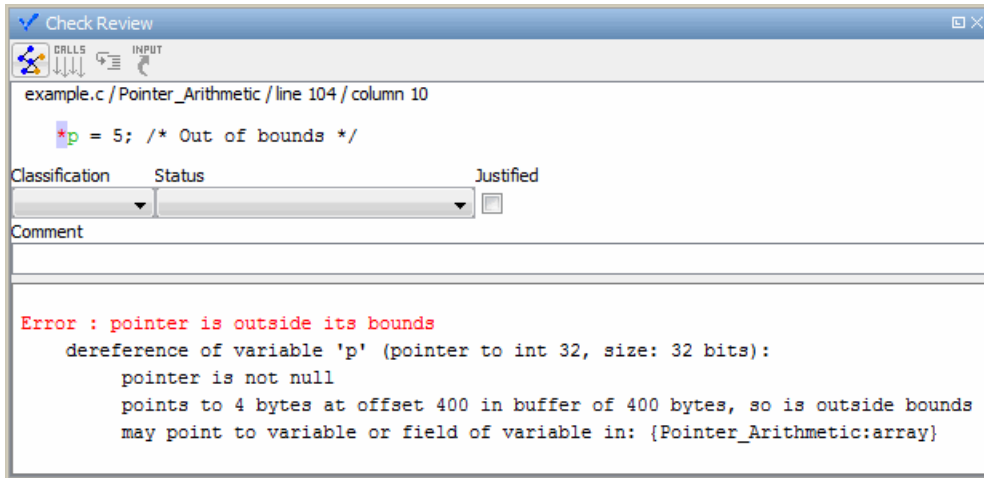
Reviewing and Commenting Checks

When reviewing checks in either Assistant or Manual mode, you can mark checks **Justified**, and enter comments to describe the results of your review. This helps you track the progress of your review and avoid reviewing the same check twice.

To review and comment a check:

- 1 Select the check that you want to review.

The Check Review pane displays information about the current check.



2 After you review the check, select a **Classification** to describe the severity of the issue:

- High
- Medium
- Low
- Not a defect

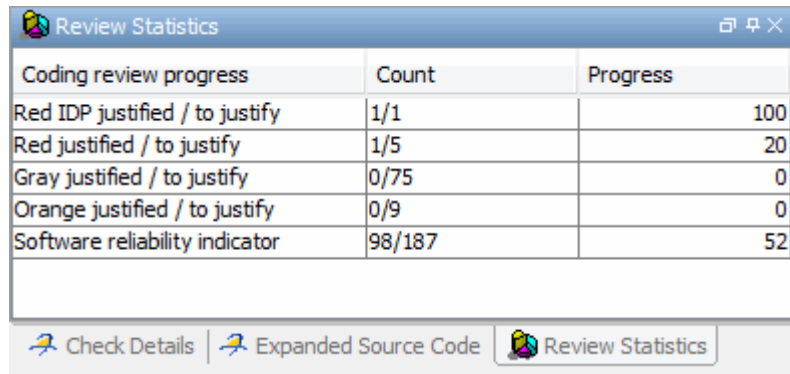
3 Select a **Status** to describe how you intend to address the issue:

- Fix
- Improve
- Investigate
- Justify with annotations
- No Action Planned
- Other
- Restart with different options
- Undecided

Note You can also define your own statuses. See “Defining Custom Status” on page 8-58.

- 4 In the comment box, enter additional information about the check.
- 5 Select the check box to indicate that you have justified this check.

The software updates the ratios of errors justified to total errors in the **Review Statistics** pane of the Run-Time Checks perspective window.



Coding review progress	Count	Progress
Red IDP justified / to justify	1/1	100
Red justified / to justify	1/5	20
Gray justified / to justify	0/75	0
Orange justified / to justify	0/9	0
Software reliability indicator	98/187	52

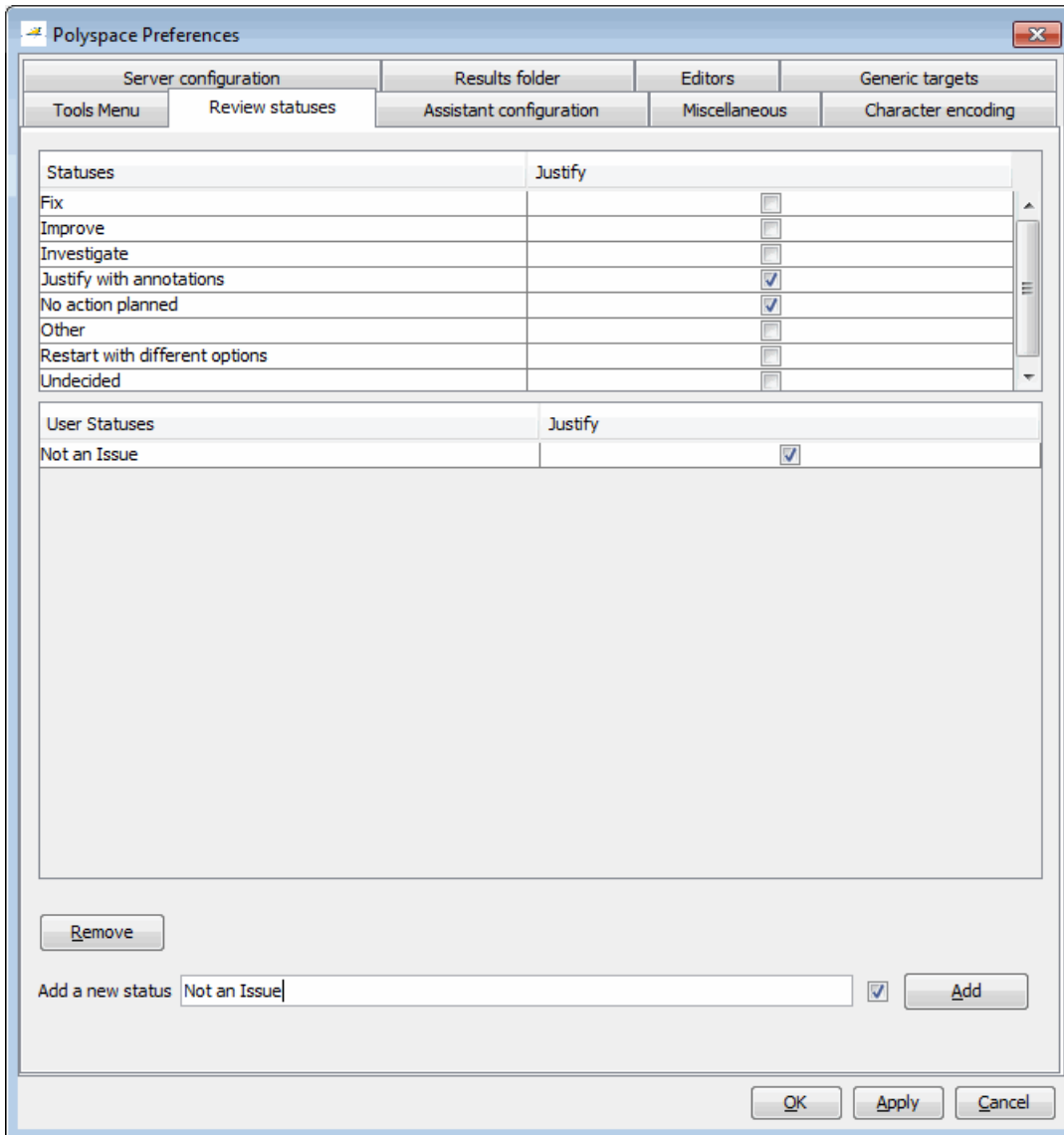
Check Details | Expanded Source Code | Review Statistics

Defining Custom Status

In addition to the Predefined statuses for reviewing checks, you can define your own statuses. Once you define a status, you can select it from the **Status** menu in the Selected check view.

To define custom statuses:

- 1 Select **Options > Preferences**.
- 2 Select the **Review Statuses** tab.

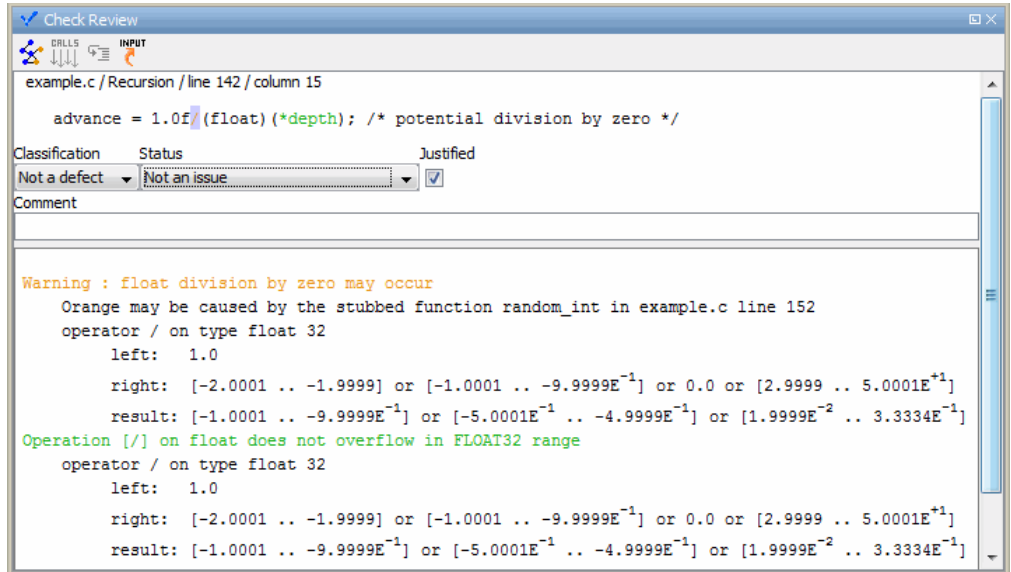


3 Enter your new status at the bottom of the dialog box, then click **Add**.

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 **Status** menu in the Selected check view.



Tracking Justified Checks in Procedural Entities View

You can use the Run-Time checks pane to display which checks have been justified and the Status used to describe each check.

Procedural entities	?	X	?	✓	Line	Col	%	Justified	Status
Demo_C	8	42	14	65			89	<input type="checkbox"/>	
example.c	4	5	6	15	1		80	<input type="checkbox"/>	
Close_To_Zero ()			3	2	37	12	40	<input type="checkbox"/>	
Non_Infinite_Loop ()				5	66	11	100	<input type="checkbox"/>	
Pointer_Arithmetic ()	1	2		1	89	12	100	<input type="checkbox"/>	
RTE ()	1				222	5	100	<input type="checkbox"/>	
Recursion ()			1	2	137	12	67	<input type="checkbox"/>	
OVFL.5				1	141	17		<input type="checkbox"/>	
OVFL.6				1	142	15		<input type="checkbox"/>	
2DV.7				1	142	15		<input checked="" type="checkbox"/>	Not an Issue
Recursion_caller ()	1				151	12	100	<input type="checkbox"/>	
Square_Root ()	1	1		1	185	12	100	<input type="checkbox"/>	
Square_Root_conv ()				4	179	12	100	<input type="checkbox"/>	
Unreachable_Code ()	2	1			199	12	67	<input type="checkbox"/>	
get_oil_pressure ()				1	21	11	0	<input type="checkbox"/>	
initialisations.c		1		10	1		100	<input type="checkbox"/>	

Tip If you do not see the **Justified** column, resize the **Procedural entities** view to display the column. If it does not appear, right click the **Procedural entities** column heading and select **Justified**.

You can select the **Justified** check box to mark a check as justified. Selecting this check box also automatically:

- Selects the **Justified** check box for that check in the Check Review pane.
- Updates the counts in the Review Statistics pane.

Commenting Code to Justify Known Checks

You can place comments in your code that inform Polyspace software of known checks. This allows you to highlight and categorize checks identified in

previous verifications, so that you can focus on new checks when reviewing your verification results.

The Run-Time Checks perspective displays the information that you provide within your code comments, and marks the checks as **Justified**.

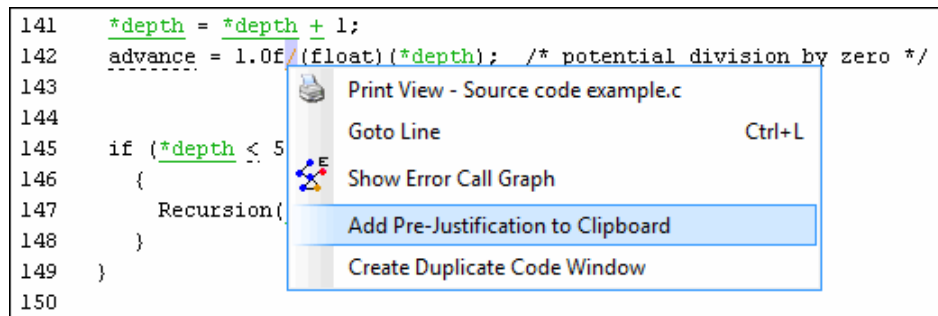
For more information, see “Annotating Code to Indicate Known Coding Rule Violations” on page 5-47.

Copying and Pasting Justifications

Instead of typing the full syntax of an annotation comment in your source code, you can copy an annotation template from the Run-Time Checks perspective, paste it into your source code, and modify the template to comment the check.

To copy the justification template to the clipboard:

- 1 Right-click anywhere in the Code pane, then select **Add Pre-Justification to Clipboard**.



The justification string is copied to the clipboard.

- 2 Open the source file containing the check you want to justify.
- 3 Navigate to the code you want to comment, and paste the justification template string on the line immediately before the line you want to comment.
- 4 Modify the template text to comment the code appropriately.

```
if (random_int() > 0)
{
    /* polyspace<RTE: NTC : Low : No Action Planned > This run-time error was discovered previously */
    Square_Root();
}

Unreachable_Code();
```

5 Save the file.

Importing and Exporting Review Comments

In this section...
“Reusing Review Comments” on page 8-64
“Importing Review Comments from Previous Verifications” on page 8-65
“Exporting Review Comments to Spreadsheet” on page 8-66
“Viewing Checks and Comments Report” on page 8-66

Reusing Review Comments

After you have reviewed verification results on a module, you can reuse your review comments with subsequent verifications of the same module. This allows you to avoid reviewing the same check twice, or to compare results over time.

The Run-Time Checks perspective allows you to either:

- Import review comments from another set of results into the current results.
- Export review comments from the current results to a spreadsheet.

You can also generate a report that compares the source code and verification results from two verifications, and highlights differences in the results.

Note If the code has changed since the previous verification, the imported comments may not be applicable to your current results. For example, the justification for an orange check may no longer be relevant to the current code. Open the Import/Export Report to see changes that affect your review comments.

Importing Review Comments from Previous Verifications



If you have previously reviewed verification results for a module and saved your comments, you can import those comments into the current verification, allowing you to avoid reviewing the same check twice.

Caution The comments you import replace any existing comments in the current results.

To import review comments from a previous verification:

- 1 Open your most recent verification results in the Run-Time Checks perspective.
- 2 Select **Review > Import > Import Comments**.
- 3 Navigate to the folder containing your previous results.
- 4 Select the results (.RTE) file, 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 “Viewing Checks and Comments Report” on page 8-66.

Once you import checks and comments, the **go to next check**  icon in assistant mode will skip any justified checks, allowing you to review only checks that you have not justified previously. If you want to view reviewed checks, click the **go to next reviewed check**  icon.

Note If the code has changed since the previous verification, the imported comments may not be applicable to your current results. For example, the justification for an orange check may no longer be relevant to the current code.

Exporting Review Comments to Spreadsheet

After you have reviewed verification results, you can export your review comments to .CSV format, for use with the PolyspaceMacro Excel® Report.

To export review comments to spreadsheet format:

- 1 Select **Review > Export in Spreadsheet Format**.
- 2 Navigate to the Polyspace-Doc folder within your results folder. For example:
`polyspace_project\Verification_(1)\Result_(1)\Polyspace-Doc`
- 3 Select **Export**.

The review comments from the current results are exported into .CSV format.

For information on generating the Excel Report, see “Generating Excel Reports” on page 8-74.

Viewing Checks and Comments Report

Importing review comments from a previous verification can be extremely useful, since it allows you to avoid reviewing checks twice, and to compare verification results over time.

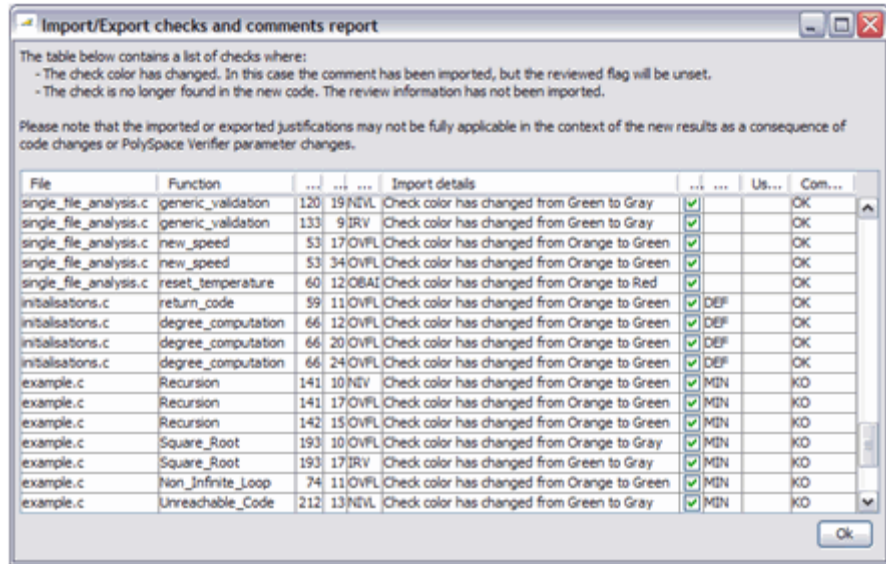
However, if your code has changed since the previous verification, or if you have upgraded to a new version of the software, the imported comments may not be applicable to your current results. For example, the color of a check may have changed, or the justification for an orange check may no longer be relevant to the current code.

The Import/Export checks and comments report allows you to compare the source code and verification results from a previous verification to the current verification, and highlights differences in the results.

To view the Import/Export checks and comments report:

- 1 Select **Review > Import > Open Import Report**.

The Import/Export checks and comments report opens, highlighting differences in the two results, such as unmatched lines and changes to the color of checks.



If the color of a check changes, the previous review comments are imported, but the check is not marked as reviewed.

If a check no longer appears in the code, the report highlights the change, but the software does not import any comments on the check.

Generating Reports of Verification Results

In this section...

“Polyspace Report Generator Overview” on page 8-68

“Generating Verification Reports” on page 8-70

“Running the Report Generator from the Command Line” on page 8-72

“Automatically Generating Verification Reports” on page 8-73

“Customizing Verification Reports” on page 8-73

“Generating Excel Reports” on page 8-74

Polyspace Report Generator Overview

The Polyspace Report Generator allows you to generate reports about your verification results, using predefined report templates.

Report Templates

The Polyspace Report Generator provides the following report templates:

- **Coding Rules Report** – Provides information about compliance with MISRA C, MISRA C++, or JSF C++ Coding Rules, as well as Polyspace configuration settings for the verification.
- **Developer Report** – Provides information useful to developers, including summary results, detailed lists of red, orange, and gray checks, and Polyspace configuration settings for the verification. Detailed results are sorted by type of check (Proven Run-Time Violations, Proven Unreachable Code Branches, Unreachable Functions, and Unproven Run-Time Checks).
- **Developer Review Report** – Provides the same information as the Developer Report, but reviewed results are sorted by review classification (High, Medium, Low, Not a defect) and status, and untagged checks are sorted by file location.
- **Developer with Green Checks Report** – Provides the same content as the Developer Report, but also includes a detailed list of green checks.
- **Quality Report** – Provides information useful to quality engineers, including summary results, statistics about the code, graphs showing

distributions of checks per file, and Polyspace configuration settings for the verification.

- **Software Quality Objectives Report** – Provides comprehensive information on software quality objectives (SQA), including code metrics, code analysis (coding-rules checker results), code verification (run-time checks), and the configuration settings for the verification. The code metrics section provides is the same information displayed in the Polyspace Metrics web interface.

Report Formats

The Polyspace Report Generator allows you to generate verification reports in the following formats:

- HTML
- PDF
- RTF
- DOC (Microsoft® Word)
- XML

Note Microsoft Word format is not available on UNIX platforms. RTF format is used instead.

Note You must have Microsoft Office installed to view .RTF format reports containing graphics, such as the Quality report.

Gray Checks Reported in Summary vs. Check Details

When you generate a report, the number of gray checks reported in the Code Verification Summary section may differ from the number of gray checks listed in the Run-Time Checks Results section.

This happens because the summary provides the total number of gray checks in your results, while the detailed tables in the Run-Time Checks Results section does not list every gray check.

In the details section:

- Only UNR checks are listed, since all gray checks derived from a UNR check do not have to be justified.
- All gray checks derived from a red check are not listed, since they all have the same cause.

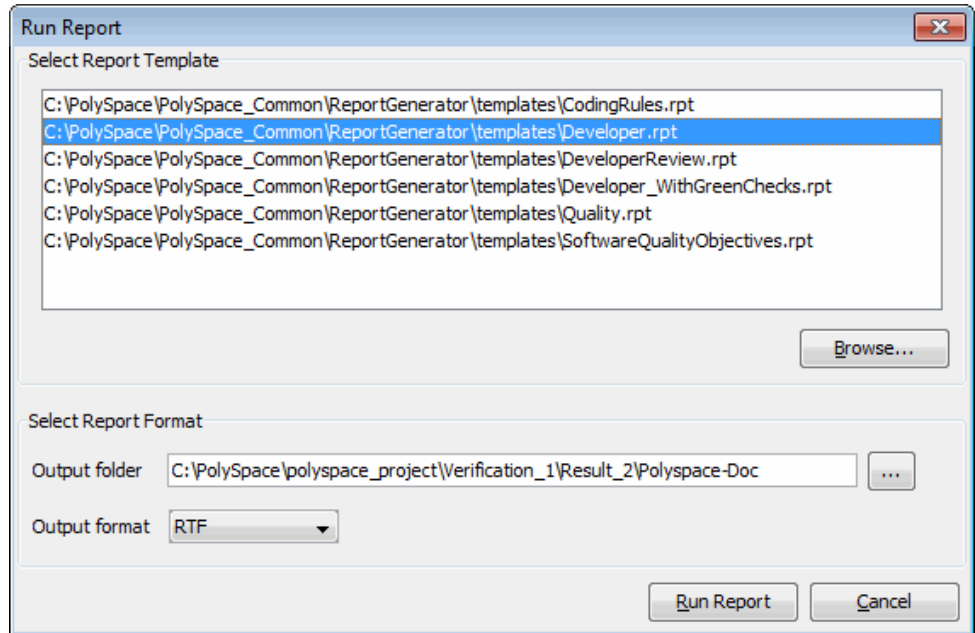
Generating Verification Reports

You can generate reports for any verification results using the Polyspace Report Generator.

To generate a verification report:

- 1** In the Run-Time Checks perspective, open your verification results.
- 2** Select **Run > Run Report > Run Report**.

The Run Report dialog box opens.



- 3** In the Select Report Template section, select the type of report that you want to run.
- 4** If your results are part of a unit-by-unit verification, you can generate a report for the current unit results, or for the entire project. Select **Generate a single report including all unit results** to combine all unit results in the report.
- 5** Select the Output folder in which to save the report.
- 6** Select the Output format for the report.
- 7** Click **Run Report**.

The software creates the specified report.

Note If you generate an RTF format report on a Linux system, the software does not open the report at the end of the generation process.

Running the Report Generator from the Command Line

You can also run the Report Generator, with options, from the command line, for example:

```
C:\>\Polyspace\Polyspace_Common\ReportGenerator\wbin\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,

```
C:\Polyspace\Polyspace_Common\ReportGenerator\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.

Automatically Generating Verification Reports

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 In the Analysis options section of the Configuration pane, expand **General**.
You see the General options.
- 3 Select **Report Generation**.
- 4 Select the **Report template name**.
- 5 Select the **Output format** for the report.
- 6 Save your project.

Customizing Verification Reports

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** – *Polyspace_Common/ReportGenerator/components*
- **Report templates** – *Polyspace_Common/ReportGenerator/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:
- 3 Set the current folder in MATLAB to the Polyspace reports template folder, using the following command:

```
addpath(`Polyspace_Common/ReportGenerator/components')
```

```
cd('Polyspace_Common/ReportGenerator/templates')
```

- 4 Start the Report Editor GUI using the following command:

```
report
```

For more information on using the MATLAB Report Generator software, refer to the *MATLAB Report Generator User's Guide*.

Note To access custom reports in the Polyspace Report Generator, you must save the report template in:*Polyspace_Common/ReportGenerator/templates*.

Generating Excel Reports

You can generate Microsoft Excel reports of your verification results. These reports contain a summary of the information displayed in the Run-Time Checks perspective. Excel Reports can also contain review comments, if you export your comments to spreadsheet format before generating the report.

Note Excel reports do not use the Polyspace Report Generator.

The Excel report contains the following sheets:

- **RTE Checks** – List of all checks, as displayed in the Procedural Entities view.
- **Launching Options** – Analysis options used for the verification.
- **Check Synthesis** – Statistics showing RTE checks by category, including the Selectivity of each category.
- **Checks by file** – Statistics showing distribution of RTE checks by file.
- **Orange Check Distribution** – Graph showing orange check distribution by category.
- **Checks per file** – Graph showing check distribution by file.
- **Global Data Dictionary** – Full list of global variables and where in the source code they are read or written to, as displayed in the Variables Access pane. .
- **Shared Globals** – Protected and non-protected global variables.
- **Application Call Tree** – Full call-tree of functions in the source code, as displayed in the Call Hierarchy pane.

To generate an Excel report of your verification results:

- 1** (Optional) If you want your report to contain review comments, export your review comments to spreadsheet format. For more information, see “Exporting Review Comments to Spreadsheet” on page 8-66.
- 2** In your results folder, navigate to the Polyspace-Doc folder. For example: `polyspace_project\results\Polyspace-Doc`.

The folder should have the following files:

```
Example_Project_Call_Tree.txt
Example_Project_RTE_View.txt
Example_Project_Variable_View.txt
```

Example_Project-NON-SCALAR-TABLE-APPENDIX.ps
Polyspace_Macros.xls

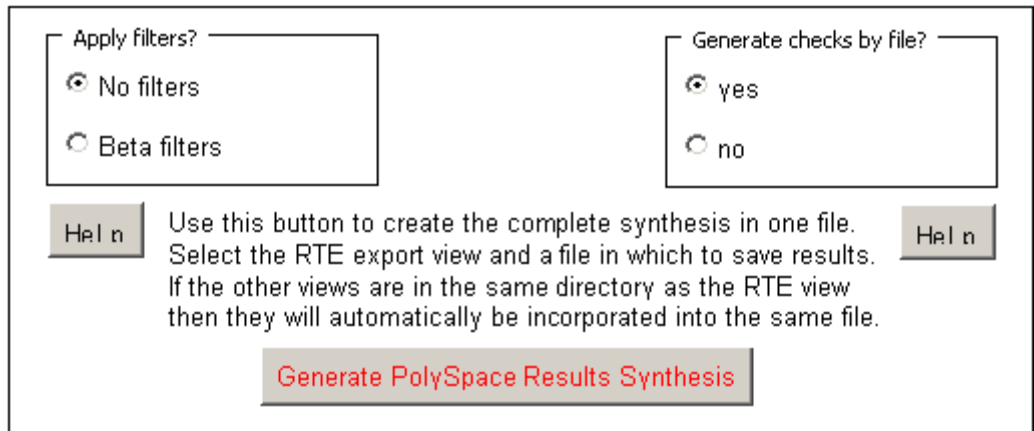
The first three files correspond to the call tree, RTE, and variable views in the Polyspace Run-Time Checks Perspective window.

- 3 Open the macros file Polyspace_Macros.xls.

You see a security warning dialog box.

- 4 Click **Enable Macros**.

A spreadsheet opens. The top part of the spreadsheet looks like the following figure.



- 5 Specify the report options that you want, then click **Generate Polyspace Results Synthesis**.

The synthesis report combines the RTE, call tree, and variables views into one report.

The **Where is the Polyspace RTE View text file** dialog box opens.

- 6 In **Look in**, navigate to the Polyspace-Doc folder in your results folder. For example: polyspace_project\results\Polyspace-Doc.
- 7 Select *Project_RTE_View.txt*.

8 Click **Open** to close the dialog box.

The **Where should I save the analysis file?** dialog box opens.

9 Keep the default file name and file type.

10 Click **Save** to close the dialog box and start the report generation.

Microsoft Excel opens with the spreadsheet that you generated. This spreadsheet has several worksheets.

Example_Project-Synthesis.xls	
A	
1	Call Graph of ll tree
2	
3	all tree
4	__polyspace_main.main
5	- > example.RTE
6	- > example.Close_To_Zero
7	> pst_stubs_0.random_float
8	> pst_stubs_0.random_float
9	> pst_stubs_0.random_int
10	> example.Non_Infinite_Loop
11	- > example.Pointer_Arithmetic
12	> pst_stubs_0.get_bus_status
13	> example.get_oil_pressure
14	> pst_stubs_0.get_bus_status
15	- > example.Recursion_caller
16	> pst_stubs_0.random_int
17	- > example.Recursion
18	** RecursiveCall to example.Recursion:
19	> pst_stubs_0.random_int
20	- > example.Recursion
21	Already displayed above
22	> pst_stubs_0.random_int
23	- > example.Square_Root
24	> pst_stubs_0.random_float
25	- > example.Square_Root_conv
26	> ?extern.cos
27	> ?extern.sqrt
28	- > example.Unreachable_Code
29	> pst_stubs_0.random_int
30	> pst_stubs_0.random_int

Application Call Tree / Shared Globals / Global Data Dictionary / Checks by file

- 11 Select the **Check Synthesis** tab to view the worksheet showing statistics by check category.

Example_Project-Synthesis.xls						
	A	B	C	D	E	F
1	RTE Statistics					
2	Check category	Check detail	R	O	Gy	Gr
3	OBAI	Out of Bounds Array Index	0	0	0	0
4	NIVL	Uninitialized Local Variable	0	1	2	32
5	IDP	Illegal Dereference of Pointer	1	1	0	7
6	NIP	Uninitialized Pointer	0	0	0	12
7	NIV	Uninitialized Variable	0	0	0	6
8	IRV	Initialized Value Returned	0	0	0	13
9	COR	Other Correctness Conditions	0	0	0	2
10	ASRT	User Assertion Failure	0	1	0	0
11	POW	Power Must Be Positive	0	0	0	0
12	ZDV	Division by Zero	0	1	0	4
13	SHF	Shift Amount Within Bounds	0	0	0	0
14	OVFL	Overflow	0	2	3	5
15	UNFL	Underflow	0	0	3	7
16	UOVFL	Underflow or Overflow	0	3	0	5
17	EXCP	Arithmetic Exceptions	0	0	0	0
18	NTC	Non Termination of Call	3	0	0	0
19	k-NTC	Known Non Termination of Call	0	0	0	0
20	NTL	Non Termination of Loop	0	0	0	0
21	UNR	Unreachable Code	0	0	1	0
22	UNP	Uncalled Procedure	0	0	0	0
23	IPT	Inspection Point	0	0	0	0
24	OTH	other checks	0	0	0	0
25	EXC	Exception handling	0	0	0	0
26	CCP	Control Flow	0	0	0	0

Using Polyspace Results

In this section...

“Review Runtime Errors: Fix Red Errors” on page 8-80

“Red Checks Where Gray Checks were Expected” on page 8-81

“Using Range Information in Run-Time Checks Perspective” on page 8-83

“Using Pointer Information in Run-Time Checks Perspective” on page 8-88

“Why Review Dead Code Checks” on page 8-92

“Reviewing Orange Checks” on page 8-94

“Integration Bug Tracking” on page 8-94

“How to Find Bugs in Unprotected Shared Data” on page 8-95

“Dataflow Verification” on page 8-96

“Data and Coding Rules” on page 8-96

“Potential Side Effect of a Red Error” on page 8-97

“Relationships Between Variables” on page 8-98

Review Runtime Errors: Fix Red Errors

All Runtime Errors highlighted by Polyspace verification are determined by reference to the language standard, and are sometimes implementation dependant — that is, they may be acceptable for a particular compiler but unacceptable according to the language standard.

Consider an overflow on a type restricted from -128 to 127. The computation of $127+1$ cannot be 128, but 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. Any deviation from the recommendations of the language standard is treated as a red error, and must therefore be corrected.

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 particular compiler specific behavior . An example of a Polyspace option to permit compiler specific behavior is the option to force “IN/OUT” ADA function parameters to be initialized. 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 bugs.

Most of the bugs you find are easy to correct once the software identifies them. Polyspace verification identifies bugs regardless of their consequence, or how difficult they may be to correct.

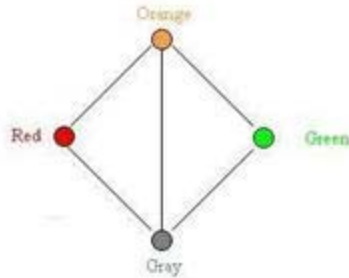
Red Checks Where Gray Checks were Expected

By default, Polyspace verification continues when it finds a red error. This is used to deal with two primary circumstances:

- A red error appears in code which was expected to be dead code.
- A red error appears which was expected, but the verification is required to continue.

Polyspace verification performs an upper approximation of variables. Consequently, it may be true that Polyspace software verifies a particular branch of code as though it was accessible, despite the fact that it could never be reached during “real life” execution. In the example below, there is an attempt to compare elements in an array, and the verification is not able to conclude that the branch was unreachable. Polyspace verification may conclude that an error is present in a line of code, even when that code cannot be reached.

Consider the figure below.



As a result of imprecision, each color shown can be approximated by a color immediately above it in the grid. It is clear that green or red checks can be approximated by orange ones, but the approximation of gray checks is less obvious.

During Polyspace verification, data values possible at execution time are represented by supersets including those values - and possibly more besides.

Gray code represents a situation where no valid data values exist. Imprecision means that such situation can be approximated

- by an empty superset;
- by a nonempty super set, members of which may generate checks of any color.

Therefore, the verification cannot be guaranteed to find all dead code.

However, there is no problem in having gray checks approximated by red ones. Where any red error is encountered, all instructions which follow it in the relevant branch of execution are aborted as usual. At execution time, it is also true that those instructions would not be executed.

Consider the following example:

```
if (condition) then action_producing_a_red;
```

After the "if" statement, the only way execution can continue is if the condition is false; otherwise a **red check** would be produced. Therefore, after this branch the condition is always false. For that reason, the code verification continues, even with a specific error. Remember that this propagates values throughout your application. None of the execution paths leading to a run-time error will continue after the error and if the **red check** is a real problem rather than an approximation of a gray check, then the verification will not be representative of how the code will behave when the red error has been addressed.

It is applicable on the current example:

```
1 int a[] = { 1,2,3,4,5,7,8,9,10 };
2 void main(void)
3 {
4   int x=0;
5   int tmp;
6   if (a[5] > a[6])
7     tmp = 1 /x; // RED ERROR [scalar division by zero] in gray code
8 }
```

Using Range Information in Run-Time Checks Perspective

- “Viewing Range Information” on page 8-83
- “Interpreting Range Information” on page 8-84
- “Diagnosing Errors with Range Information” on page 8-85

Viewing Range Information

You can see range information associated with variables and operators within the Source pane. Place your cursor over an operator or variable. A tooltip message displays the range information, if it is available.

Note The displayed range information represents a superset of dynamic values, which the software computes using static methods.

In the Source pane, if a line of code contains colored checks, then selecting a check displays the error or warning message along with range information in the selected check view.

Note Computing range information for reads and operators may take a long time. You can reduce verification time by limiting the amount of range information displayed in verification results. See “Less range information (-less-range-information)” in the *Polyspace Products for C/C++ Reference Guide*.

Interpreting 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
```

In the following example,

```
30 {
31   int temp;
32   PowerLevel = -10000;
33
34   RTE assignment of variable 'PowerLevel' (int 32): -10000
35
```

the tooltip message indicates the variable PowerLevel is a 32-bit integer with the value -10000.

In the next example,

```
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]
```

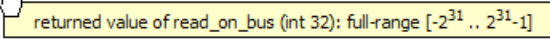
the tooltip message indicates that the variable advance is a 32-bit float that lies between either -1.0001 and -4.6566E-10 or 1.9999E-2 and 3.3334E-1

The tooltip message also indicates whether the variable occupies the full range:

```

37
38  temp = read_on_bus();
39  switch(temp)
40  {

```



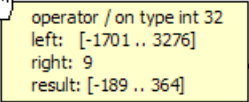
The tooltip message indicates that the returned value of the function `read_on_bus` is a 32-bit integer that occupies the full range of the data type, -2147483648 to 2147483647.

With operators, the software displays associated information. Consider the following example:

```

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  {

```



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

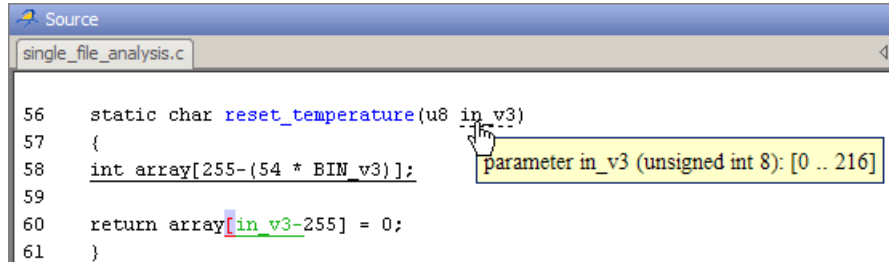
Diagnosing Errors with Range Information

You can use range information to diagnose errors. Consider the function `reset_temperature()` in the following example:

Procedural entities	!	X	?	✓	Line	g
Demo_C	9	78	25	272	93	
-example.c	4	8	8	83	1	92
-initialisations.c	3	1	41	1	98	
-main.c	2	6	3	9	1	85
-single_file_analysis.c	2	4	8	82	1	92
..._init_globals ()					1	0
+..._all_values_s16 ()			2	5	25	71
+..._all_values_s32 ()			2	5	24	71
+..._all_values_u16 ()			2	5	26	71
+...functional_ranges ()				6	37	100
+...generic_validation ()	1	4	2	50	64	98
+...new_speed ()				9	51	100
+...reset_temperature ()	1			2	56	100
... OBAI.0	1				60	
... NIVL.1				1	60	
... OVFL.2				1	60	
...unused_function ()					137	0
+...tasks1.c			3	26	1	90
+...tasks2.c			2	17	1	89
+...polyspace_stdsubs.c	1	57		14	1	100

Clicking the red check, OBAI.0 in the **Procedural entities** view or [on line 60 in the source code view, displays an error message and range information in the Check Review pane:

Although `in_v3` is green (as a local variable), it is in the range 0 - 216. This results in a negative index range. Moving the cursor to the beginning of the function reveals the cause of the red check: the input argument is between 0 and 216:



```

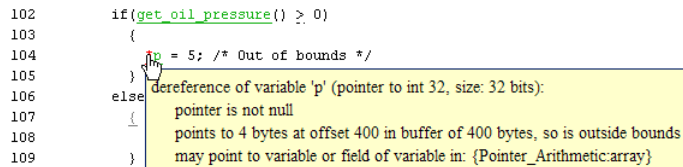
Source
single_file_analysis.c

56 static char reset_temperature(u8 in_v3)
57 {
58 int array[255-(54 * BIN_v3)];
59
60 return array[in_v3-255] = 0;
61 }
parameter in_v3 (unsigned int 8): [0 .. 216]

```

Using Pointer Information in Run-Time Checks Perspective

Within the Source pane, you can see information about pointers to variables or functions. If you place the cursor over a pointer, dereference character ([, ->, *), function call, or function declaration, a tooltip message displays pointer information. For example:



```

102 if(get_oil_pressure() >= 0)
103 {
104     p = 5; /* Out of bounds */
105 }
106 else
107 {
108     [
109     }
dereference of variable 'p' (pointer to int 32, size: 32 bits):
pointer is not null
points to 4 bytes at offset 400 in buffer of 400 bytes, so is outside bounds
may point to variable or field of variable in: {Pointer_Arithmetic:array}

```

If you click the pointer check (IDP, NIP), dereference character, function call, or function declaration, the software also displays the pointer information in **Check Review**.


```
example.c / Pointer_Arithmetic / line 104 / column 10
⊞ *p = 5; /* Out of bounds */
```

Classification	Status	Justified Comment
		<input type="checkbox"/>
Error : pointer is outside its bounds dereference of variable 'p' (pointer to int 32, size: 32 bits): pointer is not null points to 4 bytes at offset 400 in buffer of 400 bytes, so is outside bounds may point to variable or field of variable in: {Pointer_Arithmetic:array}		

For a pointer to a 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 8-91.
- 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 8-92.

Note Tooltip messages display only lines that contain meaningful information. For example, when a pointer is initialized by the main generator, the tooltip does not display lines for offset and aliases.

For a pointer to a function, 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).

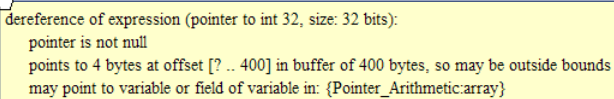
Note Computing pointer information may take a long time. You can disable the display of pointer information by selecting the option `no-pointer-information`. See “No pointer information (`-no-pointer-information`)” in the *Polyspace Products for C/C++ Reference Guide*.

You can use pointer information when analyzing, for example, red and orange IDP and NIP checks. In the following example, placing the cursor over the orange check shows that offset position may lie outside the bounds of the pointer.

```

112  i = get_bus_status();
113
114  if (i >= 0) {*(p-i) = 10;}
115
116  if ((0 < i) &&
117      {
118         p = p - i;
119         *p = 5;

```



dereference of expression (pointer to int 32, size: 32 bits):
 pointer is not null
 points to 4 bytes at offset [? .. 400] in buffer of 400 bytes, so may be outside bounds
 may point to variable or field of variable in: {Pointer_Arithmetic:array}

Messages on Dereferences

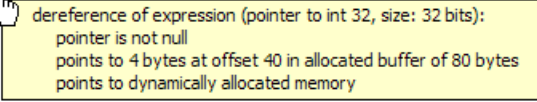
Tooltip messages on dereferences give information about the expression that is dereferenced.

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
```



On `p[10]`, the expression dereferenced is `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 40 ($10 * \text{sizeof(int)}$).
- The dereferenced pointer is not null.
Explanation: `p` is null, but `p+10` is not null ($0+40 \neq 0$).

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 `-allow-ptr-arith-on-struct`. See “Enable pointer arithmetic out of bounds of fields (`-allow-ptr-arith-on-struct`)” in the *Polyspace Products for C/C++ Reference Guide*.

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
```

Why Review Dead Code Checks

- “Functional Bugs in Gray Code” on page 8-92
- “Structural Coverage” on page 8-94

Functional Bugs in Gray Code

Polyspace verification finds different types of dead code. Common examples include:

- Defensive code which is never reached.
- Dead code due to a particular configuration.

- Libraries which are not used to their full extent in a particular context.
- Dead code resulting from bugs in the source code.

The causes of dead code listed in the following examples are taken from critical applications of embedded software by Polyspace verification.

- A lack of parenthesis and operand priorities in the testing clause can change the meaning significantly.
- Consider a line of code such as:

```
IF NOT a AND b OR c AND d
```

Now consider how misplaced parentheses might influence how that line behaves:

```
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 of variable inside a branch where the conditions are never met
- An unreachable “else” clause where the wrong variable is tested in the “if” statement
- A variable that should be local to the file but instead is local to the function
- Wrong variable prototyping leading to a comparison which is always false (say)

As is the case for red errors, the consequences of dead code and how much time you must spend on it is unpredictable. For example, it can be:

- A one-week effort of functional testing on target, trying to build a scenario going into that branch.
- A three-minute code review discovering the bug.

Again, as for red errors, Polyspace does not measure the impact of dead code.

The tool provides a list of dead code. A short code review enables you to identify known dead code and uncover real bugs.

In general, at least 30% of gray code reveals real bugs.

Structural Coverage

Polyspace software always performs upper approximations of all possible executions. Therefore, if a line of code is shown in green, there is a possibility that it is a dead portion of code. Because Polyspace verification makes an upper approximation, it does not conclude that the code is dead, but it could conclude that no run-time error is found.

Polyspace verification finds around 80% of dead code that the developer finds by doing structural coverage.

Use Polyspace verification as a productivity aid in dead code detection. It detects dead code which might take days of effort to find by any other means.

Reviewing Orange Checks

Orange checks indicate *unproven code*. This means that the code can neither be proven safe, nor can it be proven to contain a runtime error.

The number of orange checks you review is determined by several factors, including:

- The stage of the development process
- Your quality objectives

There are also actions you can take to reduce the number of orange checks in your results.

For information on managing orange checks in your results, see Chapter 9, “Managing Orange Checks”.

Integration Bug Tracking

By default, you can achieve integration bug tracking by applying the selective orange methodology to integrated code. Each error category reveals integration bugs, depending on the coding rules that you choose for the project.

For instance, consider a function that receives two unbounded integers. The presence of an overflow can be checked only at integration phase because at unit phase the first mathematical operation reveals an orange check.

Consider these two circumstances:

- When you carry out integration bug tracking in isolation, a selective orange review highlights most integration bugs. A Polyspace verification is performed integrating tasks.
- When you carry out integration bug tracking together with an exhaustive orange review at unit phase, a Polyspace verification is performed on one or more files.

In this second case, an exhaustive orange review already has been performed, file by file. Therefore, at integration phase, assess **only checks that have turned from green to another color** .

For instance, if a function takes a structure as an input parameter, the standard hypothesis made at unit level is that the structure is well initialized. This consequentially displays a green NIV check at the first read access to a field. But this might not be true at integration time, where this check can turn orange if any context does not initialize these fields.

These orange checks reveal integration bugs.

How to Find Bugs 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.

A shared data item is a global data item that is read from or written to by two or more tasks. It is unprotected from concurrent access when one task

can access it while another task is in the process of doing so. Consider all the possible situations:

- A scenario which would lead to such a conflict for a particular variable; then a bug exists and you must provide protection.
- No such scenarios; then one of the following explanations may apply:
 - The compilation environment guarantees an atomic read/write access on variables of type less than 1 or, 2 bytes. Therefore, all conflicts concerning a particular variable type still guarantee the integrity of the variables content. Be careful when you port the code.
 - The variable is protected by a critical section or a mutual temporal exclusion. You may want to include this information in the Polyspace launching parameters and reverify.

Consider checking whether variables are modified when they are supposed to be constant. Use the variables dictionary.

Dataflow Verification

Data flow verification is often performed within certification processes — typically in the avionic, aerospace, or transport markets.

This activity makes use of two features of Polyspace results, which are available any time after the Control and Data Flow verification phase:

- Call tree computation
- Dictionary containing read/write access to global variables. (You can also use this to build a database listing for each procedure, for its parameters, and for its variables.)

Polyspace software can help you to build these results by extracting information from both the call tree and the dictionary.

Data and Coding Rules

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

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 design problems, such as:

- File APIs (or functions accessible from outside the file) with no 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 output values.

Potential Side Effect of a Red Error

When the software finds a red error, you can continue the verification but proceed with caution. Consider this piece of code:

```
int *global_ptr;
int variable_it_points_to;

void big_red(void)
{
int r;
int my_zero = 0;
if (condition==1)
    r = 1 / my_zero; // red ZDV
...
... // hundreds of lines

global_ptr = &variable_it_points_to;

other_function();
}
```

```
void other_function(void)
{
if (condition==1)
    *global_ptr = 12;
}
```

Polyspace verification works by propagating data sets representing ranges of possible values throughout the call tree, and throughout the functions in that call tree. Sometimes, Polyspace software internally subdivides the functions for verification, and the propagation of the data ranges need several

iterations (or integration levels) to be complete. You can observe that effect by examining the color of the checks upon completion of each of those levels.

- The verification detects gray code which exists due to a terminal RTE which is not be flagged in red until a subsequent integration level.
- The verification flags an **NTC** in red with the content in gray. This red NTC is the result of an imprecision; it should be gray.

Suppose that an NTC is hard to understand at a given integration level (level 4):

- If other **red checks** exist at level 4, fix them and restart the verification
- Otherwise, look through the results from each previous level to see whether you can locate other red errors. If so, fix them and restart the verification

Relationships Between Variables

Abstract

A red error can hide a bug which occurred on previous lines.

```
%% file1.c %%                                %% file2.c %%
1 void f(int);                                1 #include <math.h>
2 int read_an_input(void);                    2
3                                              3 void f(int a)
4 int main(void)                              4 {
5 {                                            5 int tmp;
6 int x,old_x;                                6 tmp = sqrt(0-a);
7                                              7 }
8 x = read_an_input();
9 old_x = x;
10
11 if (x<0 || x>10)
12 return 0;
13
14 f(x);
15
```

```
16 x = 1 / old_x; // division is red
17
18 }
```

Explanation 1

- When `old_x` is assigned to `x` (file 1, line 9), the verification retains the following information:
 - `x` and `old_x` are equivalent to the full range of an integer: $[-2^{31}; 2^{31}-1]$.
 - `x` and `old_x` are equal.
- After the `if` clause (file 1, line 11), `X` is equivalent to `[0; 10]`. Because `x` and `old_x` are equal, **`old_x` is equivalent to `[0;10]` as well**. Otherwise the return statement is executed.
- When `X` is passed to `"f"` (file 1, line 14), the only possible conclusion for `sqrt` is that `x=0`. All other values lead to a run-time exception (file 2, line 6) `tmp = sqrtt(0 a);`.
- A red error occurs (file 1, line 16) because `x` and `old_x` are equal, therefore `old_x = 0`.

Explanation 2

- Suppose that the verification **exits** immediately when encountering a run-time error. Introduce a print statement that writes to the standard output after the `"f"` procedure is called (file 1, line 14), to show the current value of `x` and `old_x`.
- The only way the program can reach the print statement is when `X = 0`. So, if `X=0`, `old_x` must also have been assigned to `0`, which makes the division **red**.

Summary

Polyspace verification builds relationships between variables and propagates the consequence of these relationships backwards and forwards.

Managing Orange Checks

- “Understanding Orange Checks” on page 9-2
- “Too Many Orange Checks?” on page 9-12
- “Reducing Orange Checks in Your Results” on page 9-14
- “Reviewing Orange Checks” on page 9-31
- “Automatically Testing Orange Code” on page 9-52

Understanding Orange Checks

In this section...
“What is an Orange Check?” on page 9-2
“Sources of Orange Checks” on page 9-6

What is an Orange Check?

Orange checks indicate *unproven code*. This means that the code can neither be proven safe, nor can it be proven to contain a runtime error.

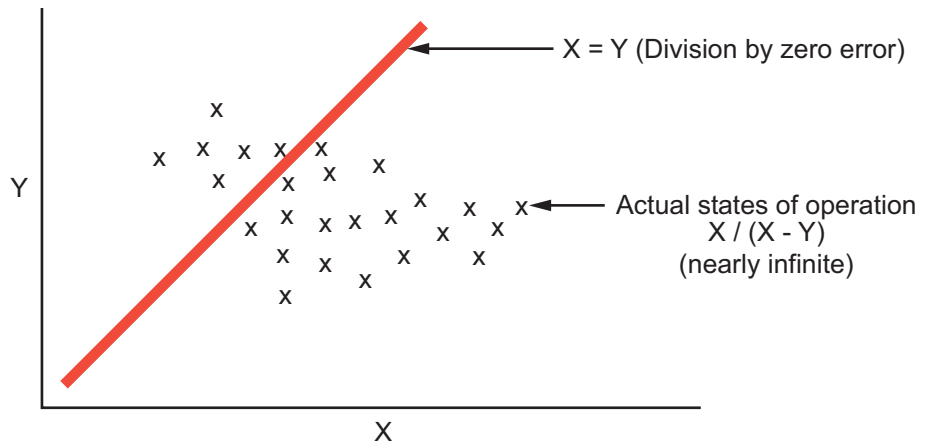
Polyspace verification does not simply try to find bugs, it attempts to prove the absence or existence of run time errors. Therefore, all code starts out as unproven prior to verification. The verification then attempts to prove that the code is either correct (green), is certain to fail (red), or is unreachable (gray). Any remaining code stays unproven (orange).

Code often remains unproven in situations where some paths fail while others succeed. For example, consider the following instruction:

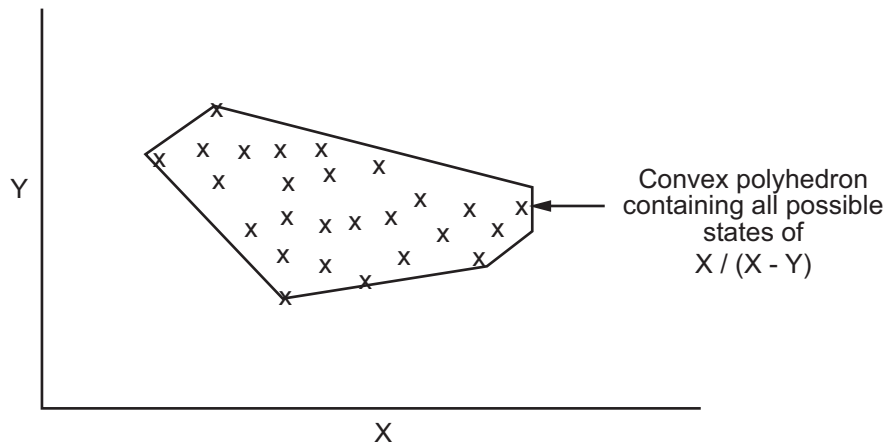
```
X = 1 / (X - Y);
```

Does a division-by-zero error occur?

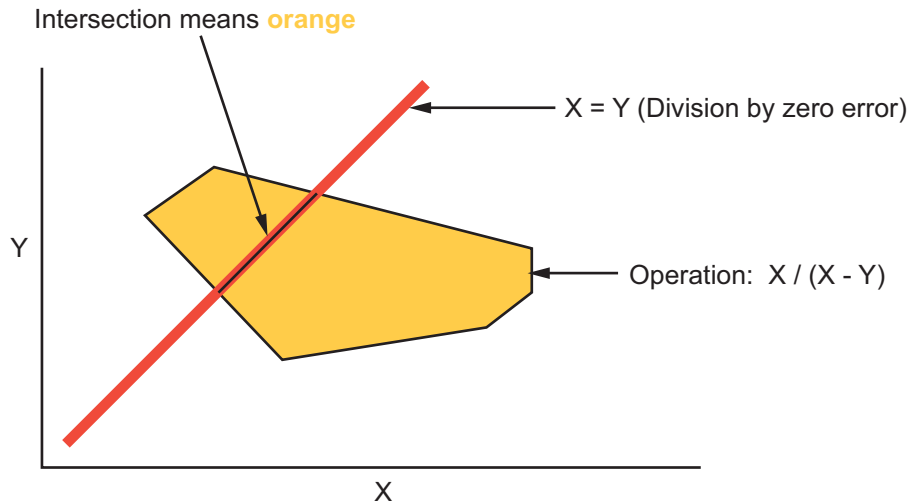
The answer clearly depends on the values of X and Y . However, there are an almost infinite number of possible values. Creating test cases for all possible values is not practical.



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.

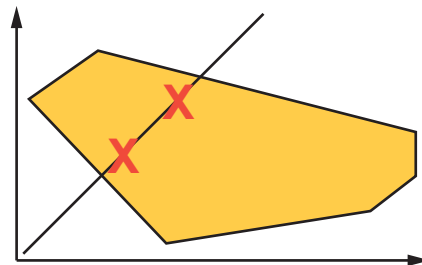


Polyspace verification then compares the data set represented by this polyhedron to the error zone. If the two data sets intersect, the check is orange.

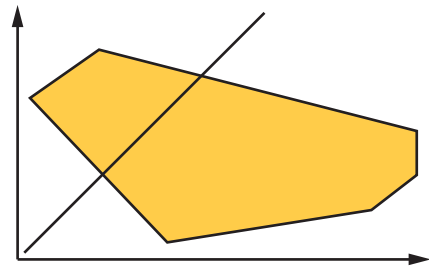


Graphical Representation of an Orange Check

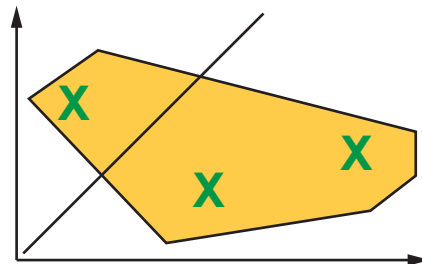
A true orange check represents a situation where some paths fail while others succeed. However, because the data set used in the verification is an approximation of actual values, an orange check may actually represent a check of any other color, as shown below.



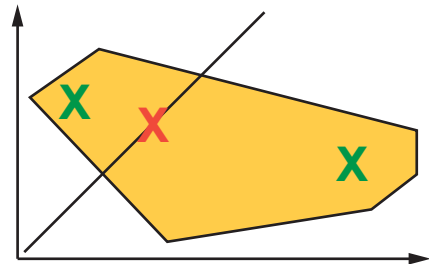
Red approximated by orange



Gray approximated by orange



Green approximated by orange



Any other situation (true orange)

Polyspace verification reports an orange check any time the two data sets intersect, regardless of the actual values. Therefore, you may find orange checks that represent bugs, while other orange checks represent code that is safe.

You can resolve some of these orange checks by increasing the precision of your verification, or by adding execution context, but often you must review the results to determine the source of an orange check.

Sources of Orange Checks

Orange checks can be separated into two categories:

- “Orange Checks Due to Code Issues” on page 9-6
- “Orange Checks Due to Tool Issues” on page 9-9

Orange Checks Due to Code Issues

Most orange checks are caused by issues in the code. These oranges may represent real bugs, or could indicate theoretical issues that cannot actually occur in your application.

Orange checks due to code issues can be caused by:

- “Potential Bug” on page 9-6
- “Data Set Issue” on page 9-7
- “Function Sequence” on page 9-8

Potential Bug. An orange check can reveal code which will fail in some circumstances. These types of orange checks are called true orange, and often represent real bugs.

For example, consider a function `Recursion()`:

- `Recursion()` takes a parameter, increments it, then divides by it.
- This sequence of actions loops through an indirect recursive call to `Recursion_recurse()`.

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 ZDV.

When an orange check indicates a potential bug, you can usually identify the cause quickly. The range information provided in the Run-Time Checks perspective can help you identify whether the orange represents a bug that should be fixed. See “Using Range Information in Run-Time Checks Perspective” on page 8-83.

If the orange represents a situation that cannot actually occur (for example, the initial value above cannot be negative), you have several options:

- Comment the orange check and ignore it.
- Modify the code to take constraints into account.
- Constrain the data ranges used in the verification using DRS (contextual verification).

Data Set Issue. An orange check can result from a theoretical set of data that cannot actually occur.

Polyspace verification uses an *upper approximation* of the data set, meaning that it considers all combinations of input data rather than any particular combination. Therefore, an orange check may result from a combination of input values that is not possible at execution time.

For example, consider three variables *X*, *Y*, and *Z*:

- Each of these variables is defined as being between 1 and 1,000.
- The code computes $X*Y*Z$ on a 16-bit data type.
- The result can potentially overflow, so it causes an orange OVFL.

When developing the code, you may know that the three variables cannot all take the value 1,000 at the same time, but this information is not available to the verification. Therefore, the multiplication is orange.

When an orange check is caused by a data set issue, it is usually possible to identify the cause quickly. The range information provided in the Run-Time Checks perspective can help you identify whether the orange represents a bug that should be fixed. See “Using Range Information in Run-Time Checks Perspective” on page 8-83.

After identifying a data set issue, you have several options:

- Comment the orange check and ignore it.
- Modify the code to take data constraints into account.

- Constrain the data ranges that are verified using DRS (contextual verification).

Function Sequence. An orange check can occur if the verification cannot conclude whether a problem exists.

In some code, it is impossible to conclude whether an error exists without additional information, such as the function sequence.

For example, consider a variable X , and two functions, $F1$ and $F2$:

- $F1$ assigns $X = 12$.
- $F2$ divides a local variable by X .
- The automatically generated main ($F0$) initializes X to 0.
- The generated main then randomly calls the functions, similar to the following:

```
If (random)
  Call F1
  Call F2
Else
  Call F2
  Call F1
```

A division by zero error is possible because $F1$ can be called before or after $F2$, so the division causes an orange ZDV. 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.
- Use `-main-generator` options to describe the function call sequence, or to specify a function called before the main.
- Write defensive code to prevent potential problems.
- Comment the orange check and ignore it.

Orange Checks Due to Tool Issues

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 oranges may suggest design issues with the code.

Orange checks due to tool issues can be caused by:

- “Code Complexity” on page 9-9
- “Basic Imprecision” on page 9-10

Code Complexity. An orange check can occur when the code structure is too complicated to be verified by Polyspace software.

When code is extremely complex, the verification cannot conclude whether a problem exists, and therefore reports an inconclusive orange check in the results.

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.

Polyspace verification reports an orange OVFL on the addition.

This type of orange check is a false positive, because the scenario does not cause a real bug. However, it does suggest that the code may 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. These checks are often

related, and analysis identifies a single cause — perhaps a function or a variable modified many times.

In these cases, you may want to recode to ensure there is no risk, depending on the criticality of the function and the required speed of execution.

To limit the number of orange checks caused by code complexity, you can:

- Enforce coding rules during development
- Perform unit-by-unit verification to verify smaller sections of code.

Note MathWorks recommends enforcing compliance with coding standards to reduce code complexity. For more information, see Chapter 11, “Checking Coding Rules”.

Basic Imprecision. 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, and therefore 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 $[10 \dots 20]$.
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 $[-5 \dots 20]$, instead of the previous set of values.
- Therefore, calling the function $x = 1/x$ causes an orange ZDV.

Polyspace verification is unable to prove the absence of a run-time error in this case.

In cases of basic imprecision, you may be able to resolve orange checks by increasing the precision level. If this does not resolve the orange check, however, verification cannot help directly. You need to review the code to determine if there is an actual problem.

To limit the number of orange checks caused by basic imprecision, avoid code constructions that cause imprecision.

For more information, see “Approximations Used During Verification” in the *Polyspace Products for C/C++ Reference*.

Too Many Orange Checks?

In this section...
“Do I Have Too Many Orange Checks?” on page 9-12
“How to Manage Orange Checks” on page 9-13

Do I Have Too Many Orange Checks?

If the goal of code verification is to prove the absence of run time errors, you may be concerned by the number of orange checks (unproven code) in your results.

In reality, asking “Do I have too many orange checks?” is not the right question. There is not an ideal number of orange checks that applies for all applications, not even zero. Whether you have too many orange checks depends on:

- **Development Stage** – Early in the development cycle, when verifying the first version of a software component, you may want to focus exclusively on finding red errors, and not consider orange checks. As development of the same component progresses, however, you may want to focus more on orange checks.
- **Application Requirements** – There are actions you can take during coding to produce more provable code. However, writing provable code often involves compromises with code size, code speed, and portability. Depending on the requirements of your application, you may decide to optimize code size, for example, at the expense of more orange checks.
- **Quality Goals** – Polyspace software can help you meet quality goals, but it cannot define those goals for you. 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.

It is these factors that ultimately determine how many orange checks are acceptable in your results, and what you should do with the orange checks that remain.

Thus, a more appropriate question is “How do I manage orange checks?”

This question leads to two main activities:

- Reducing the number of orange checks
- Working with orange checks

How to Manage Orange Checks

Polyspace verification cannot magically produce quality code at the end of the development process. Verification is a tool that helps you measure the quality of your code, identify issues, and ultimately achieve the quality goals you define. To do this, however, you must integrate Polyspace verification into your development process.

Similarly, you cannot successfully manage orange checks simply by using Polyspace options. To manage orange checks effectively, you must take actions while coding, when setting up your verification project, and while reviewing verification results.

To successfully manage orange checks, perform each of the following steps:

- 1** Define your quality objectives to set overall goals for application quality. See “Defining Quality Objectives” on page 2-5.
- 2** Set Polyspace analysis options to match your quality objectives. See “Specifying Options to Match Your Quality Objectives” on page 3-26.
- 3** Define a process to reduce orange checks. See “Reducing Orange Checks in Your Results” on page 9-14.
- 4** Apply the process to work with remaining orange checks. See “Reviewing Orange Checks” on page 9-31.

Reducing Orange Checks in Your Results

In this section...
“Overview: Reducing Orange Checks” on page 9-14
“Applying Coding Rules to Reduce Orange Checks” on page 9-15
“Considering Generated Code” on page 9-20
“Improving Verification Precision” on page 9-21
“Stubbing Parts of the Code Manually” on page 9-26
“Describing Multitasking Behavior Properly” on page 9-28
“Considering Contextual Verification” on page 9-29
“Considering the Effects of Application Code Size” on page 9-30

Overview: Reducing Orange Checks

There are several actions you can take to reduce the number of orange checks in your results.

However, it is important to understand that while some actions increase the quality of your code, others simply change the number of orange checks reported by the verification, without improving code quality.

Actions that reduce orange checks and improve the quality of your code:

- **Apply coding rules** – Coding rules are the most efficient means to reduce oranges, and can also improve the quality of your code.
- **Move to generated code** – Generated code can reduce orange checks and eliminate certain types of coding errors.

Actions that reduce orange checks through increased verification precision:

- **Set precision options** – There are several Polyspace options that can increase the precision of your verification, at the cost of increased verification time.
- **Implement manual stubbing** – Manual stubs that accurately model the behavior of missing functions can increase the precision of the verification.

- **Specify multitasking behavior** – Accurately defining call sequences and other multitasking behavior can increase the precision of the verification.

Options that reduce orange checks but do not improve code quality or the precision of the verification:

- **Constrain data ranges** – You can use data range specifications (DRS) to limit the scope of a verification to specific variable ranges, instead of considering all possible values. This reduces the number of orange checks, but does not improve the quality of the code. Therefore, DRS should be used specifically to perform contextual verification, not simply to reduce orange checks.

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 objectives, as described in Chapter 2.

It is your quality objectives that determine how many orange checks are acceptable in your results, what actions you should take to reduce the number of orange checks, and what you should do with any orange checks that remain.

Applying Coding Rules to Reduce Orange Checks

The number of orange checks in your results depends strongly on the coding style used in the project. Applying coding rules can both reduce the number of orange checks in your verification results, and improve the quality of your code. Coding rules are the most efficient way to reduce orange checks.

Polyspace software allows you to check MISRA C coding rules during verification. If your code complies with the first subset of MISRA rules (coding rules with a direct impact on selectivity), the total number of orange checks will decrease substantially, and the percentage of orange checks representing real bugs will increase.

In addition, some code constructions are known to produce orange checks. If your design avoids these constructions, you will see fewer orange checks in your verification results. The second subset of MISRA rules (coding rules with an indirect impact on selectivity), checks for these constructions.

The following coding rules are recommended to reduce oranges:

- “Coding Rules with a Direct Impact on Selectivity (SQ0-subset1)” on page 9-16
- “Coding Rules with an Indirect Impact on Selectivity (SQ0-subset2)” on page 9-18

For more information on checking MISRA C coding rules, see Chapter 11, “Checking Coding Rules”.

Coding Rules with a Direct Impact on Selectivity (SQ0-subset1)

The following set of coding rules will typically improve the selectivity of your verification results.

Rule I	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 I	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.

Coding Rules with an Indirect Impact on Selectivity (SQ0-subset2)

Good design practices generally lead to less code complexity, which can improve the selectivity of your verification results. The following set of coding rules help address design issues that can impact selectivity.

Rule #	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
MISRA 12.5	The operands of a logical && or shall be primary-expressions

Rule #	Description
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
MISRA 19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives

Rule #	Description
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; }
```

Considering Generated Code

Moving to generated code can reduce the number of orange checks in your results, and improve the overall quality of your software.

Generated code has a well-defined set of coding rules, and eliminates certain types of coding errors by construction. This results in higher ratio of green checks in your verification results.

The Polyspace Model Link SL, Polyspace Model Link TL, and Polyspace UML Link™ RH products allow you to integrate Polyspace verification into a generated code workflow.

For more information, see the *Polyspace Model Link Products User's Guide*.

Improving Verification Precision

Improving the precision of a verification can reduce the number of orange checks in your results, although it does not affect the quality of the code itself.

There are a number of Polyspace options that affect the precision of the verification. The trade off for this improved precision is increased verification time.

The following sections describe how to improve the precision of your verification:

- “Balancing Precision and Verification Time” on page 9-21
- “Setting the Analysis Precision Level” on page 9-22
- “Setting Software Safety Analysis Level” on page 9-23
- “Other Options that Can Improve Precision” on page 9-24

Balancing Precision and Verification Time

When performing code verification, you must find the right balance between precision and verification time. Consider the two following extremes:

- If a verification runs in one minute but contains only orange checks, the verification is not useful because each check must be reviewed manually.
- If a verification contains no orange checks (only gray, red, and green), but takes six months to run, the verification is not useful because of the time spent waiting for the results.

Higher precision yields more proven code (red, green, and gray), but takes longer to complete. The goal is therefore to get the most precise results in the time available. Factors that influence this compromise include the time available for verification, the time available to review results, and the stage in the development cycle.

For example, consider the following scenarios:

- **Unit testing** – Before going to lunch, a developer starts verification. After returning from lunch the developer reviews verification results for one hour.

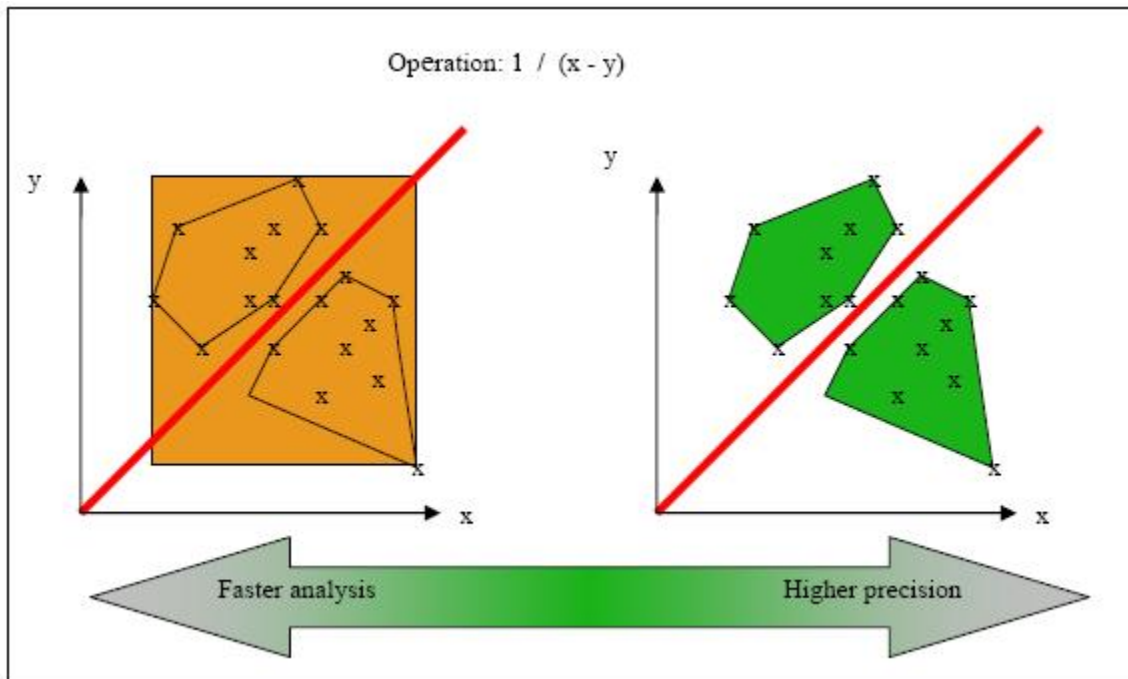
- **Integration testing** – Verification runs automatically on nightly builds of modules or software components.

These scenarios require a developer to use Polyspace software in different ways. Generally, the first verification should use the lowest precision mode, while subsequent verifications increase the precision level.

Setting the Analysis Precision Level

The analysis **Precision Level** specifies the mathematical algorithm used to compute the cloud of points (polyhedron) containing all possible states for the variables.

Although changing the precision level does not affect the quality of your code, orange checks caused by low precision become green when verified with higher precision.



Affect of Precision Rate on Orange Checks

To set the precision level:

- 1** In the Analysis options section of the Project Manager perspective, select **Precision/Scaling > Precision**.
- 2** Select the -00, -01, -02 or -03 precision level the Precision Level drop-down list.

For more information, see “Precision Level (-0)” in the *Polyspace Products for C/C++ Reference*.

Setting Software Safety Analysis Level

The Software Safety Analysis level of your verification specifies how many times the abstract interpretation algorithm passes through your code. The deeper the verification goes, the more precise it is.

There are 5 Software Safety Analysis levels (pass0 to pass4). By default, verification proceeds to pass4, although it can go further if required. Each iteration results in a deeper level of propagation of calling and called context.

To set the Software Safety Analysis level:

- 1** In the Analysis options section of the Project Manager perspective, select **Precision/Scaling > Precision**.
- 2** Select the appropriate level in the **To end of** drop-down list.

For more information, see “To end of (-to)” in the *Polyspace Products for C/C++ Reference*.

Example: Orange Checks and Software Safety Analysis Level

The following example shows how orange checks are resolved as verification proceeds through Software Safety Analysis levels 0 and 1.

Safety Analysis Level 0	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 this example, division by an input parameter of a function produces an orange during Level 0 verification, but turns to green during level 1. The verification gains more accurate knowledge of x as the value is propagated deeper.

Other Options that Can Improve Precision

The following options can also improve verification precision:

- “Improve precision of interprocedural analysis” on page 9-25
- “Sensitivity context” on page 9-25
- “Inline” on page 9-25

Note Changing these options does not affect the quality of the code itself. Improved precision can reduce the number of orange checks, but will increase verification time.

Improve precision of interprocedural analysis. This option causes the verification to propagate information within procedures earlier than usual. This improves the precision within each Software Safety Analysis level, meaning that some orange checks are resolved in level 1 instead of later levels.

However, using this option increases verification time exponentially. In some cases this could cause a level 1 verification to take longer than a level 4 verification.

For more information, see “Improve precision of interprocedural analysis (-path-sensitivity-delta)” in the *Polyspace Products for C/C++ Reference*.

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.

For more information, see “Sensitivity context (-context-sensitivity)” in the *Polyspace Products for C/C++ Reference*.

Inline. This option creates clones of a each specified procedure for each call to it. This reduces the number of aliases in a procedure, and can improve precision in some situations.

However, using this option can duplicate large amounts of code, leading to increased verification time and other scaling problems.

For more information, see “Inline (-inline)” in the *Polyspace Products for C/C++ Reference*.

Stubbing Parts of the Code Manually

Manually stubbing parts of your code can reduce the number of orange checks in your results. However, manual stubbing generally does not improve the quality of your code, it only changes the results.

Stubs do not need to model the details of the functions or procedures involved. They only need to represent the effect that the code might have on the remainder of the system.

If a function is supposed to return an integer, the default automatic stubbing will stub it on the assumption that it can potentially take any value from the full type of an integer.

The following sections describe how to reduce orange checks using manual stubbing:

- “Manual vs. Automatic Stubbing” on page 9-26
- “Emulating Function Behavior with Manual Stubs” on page 9-27

Manual vs. Automatic Stubbing

There are two types of stubs in Polyspace verification:

- **Automatic stubs** – The software automatically creates stubs for unknown functions based on the function’s prototype (the function declaration). Automatic stubs do not provide insight into the behavior of the function, but are very conservative, ensuring that the function does not cause any runtime errors.
- **Manual stubs** – You create these stub functions to model the behavior of the missing functions, and manually include them in the verification with the rest of the source code. Manual stubs can better model missing functions, or they can be empty.

By default, Polyspace software automatically stubs functions. However, because automatic stubs are conservative, they can lead to more orange checks in your results.

Stubbing Example

The following example shows the effect of automatic stubbing.

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

Due to automatic stubbing, the verification assumes that *a* can be any integer, including 0. This produces an orange check on the division.

If you provide an empty manual stub for the function, the division would be green. This reduces the number of orange checks in the result, but does not improve the quality of the code itself. The function could still potentially cause an error.

However, if you provide a detailed manual stub that accurately models the behavior of the function, the division could be any color, including red.

Emulating Function Behavior with Manual Stubs

You can improve both the speed and selectivity of your verification by providing manual stubs that accurately model the behavior of missing functions. The trade-off is time spent writing the stubs.

Manual stubs do not need to model the details of the functions or procedures involved. They only need to represent the effect that the code might have on the remainder of the system.

Example

This example shows a header for a missing function (which may occur when the verified code is an incomplete subset of a project).

```
int a,b;
int *ptr;
void a_missing_function(int *dest, int src);
```

```
/* should copy src into dest */  
void main(void)  
{  
    a = 1;  
    b = 0;  
    a_missing_function(&a, b);  
    b = 1 / a;  
}
```

The missing function copies the value of the `src` parameter to `dest`, so there is a division by zero error.

However, automatic stubbing always shows an orange check, because `a` is assumed to have any value in the full integer range. Only an accurate manual stub can reveal the true **red** error.

Using manual stubs to accurately model constraints in primitives and outside functions propagates more precision throughout the application, resulting in fewer orange checks.

Describing Multitasking Behavior Properly

The asynchronous characteristics of your application can have a direct impact on the number of orange checks. Properly describing 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** ZDV.

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**.

Refer to “*Preparing Multitasking Code*” on page 5-32 for information on tasking facilities, including:

- Shared variable protection:
 - Critical sections,
 - Mutual exclusion,
 - Tasks synchronization,
- Tasking:
 - Threads, interruptions,
 - Synchronous/asynchronous events,
 - Real-time OS.

Considering Contextual Verification

By default, Polyspace software performs *robustness verification*, proving that the software works under all 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.

Polyspace software also allows you to perform *contextual verification*, proving that the software works under normal working conditions. When performing contextual verification, you use the data range specifications (DRS) module to set external constraints on global variables and stub function return values, and the code is verified within these ranges.

Contextual verification can substantially reduce the number of orange checks in your verification results, but it does not improve the quality of your code.

Note DRS should be used specifically to perform contextual verification, it is not simply a means to reduce oranges.

For more information, see “Specifying Data Ranges for Variables and Functions (Contextual Verification)” on page 4-56.

Considering the 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 the amount of simplification applied to the data representations also depends on the required precision level (O0, O2), Polyspace verification adjusts the level of simplification:

- -O0: shorter computation time. Focus only red and gray.
 - -O2: less orange warnings.
 - -O3: less orange warnings and bigger computation time.
-

Reviewing Orange Checks

In this section...

- “Overview: Reviewing Orange Checks” on page 9-31
- “Defining Your Review Methodology” on page 9-31
- “Performing Selective Orange Review” on page 9-32
- “Importing Review Comments from Previous Verifications” on page 9-37
- “Commenting Code to Provide Information During Review” on page 9-38
- “Viewing Sources of Orange Checks” on page 9-39
- “Working with Orange Checks Caused by Input Data” on page 9-40
- “Refining Data Range Specifications” on page 9-44
- “Performing an Exhaustive Orange Review” on page 9-47

Overview: Reviewing Orange Checks

After you define a process that matches your quality objectives, you do not have too many orange checks. You have the correct number of orange checks for your quality model.

At this point, the goal is not to eliminate orange checks, it is to work efficiently with them.

Working efficiently with orange checks involves:

- Defining a review methodology to work consistently with orange checks
- Reviewing orange checks efficiently
- Importing comments to avoid duplicating review effort
- Dynamically testing orange checks

Defining Your Review Methodology

Before reviewing verification results, you should configure a methodology for your project. The methodology specifies both the type and number of orange checks you need to review at review levels 0, 1, 2, and 3.

As part of the process for defining quality objectives for your project, you should:

- For review level 0, establish the categories of potential run-time errors that must be reviewed before moving to the next level. See “Reviewing Checks at Level 0” on page 8-35.
- For review levels 1, 2, and 3, specify the number of checks per check category using either a predefined or custom methodology. See “Reviewing Checks at Levels 1, 2, and 3” on page 8-36.

Note For information on setting the quality levels for your project, see “Defining Software Quality Levels” on page 2-8.

After you configure a methodology, each developer uses the methodology to review verification results. This approach ensures that all users apply the same standards when reviewing orange checks at each stage of the development cycle.

For more information on defining a methodology, see “Reviewing Results Systematically” on page 8-34.

Performing Selective Orange Review

Once you have defined a methodology for your project, you can perform a *selective orange review*.

The number and type of orange checks you review is determined by your methodology and the quality level you are trying to achieve. As a project progresses, the quality level (and number of orange checks to review) generally increases.

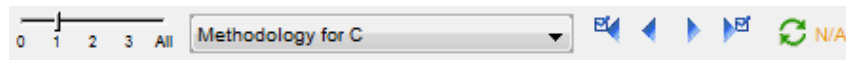
For example, you may perform a review at level 0 and 1 in the early stages of development, to improve the quality of newly written code. Later, you may perform a level 2 review as part of unit testing.


In general, 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. Therefore, to maximize the number of bugs you can identify, you should focus on those checks you can understand quickly, spending no more than 5 minutes on each check. Checks that take longer to understand are left for later analysis.

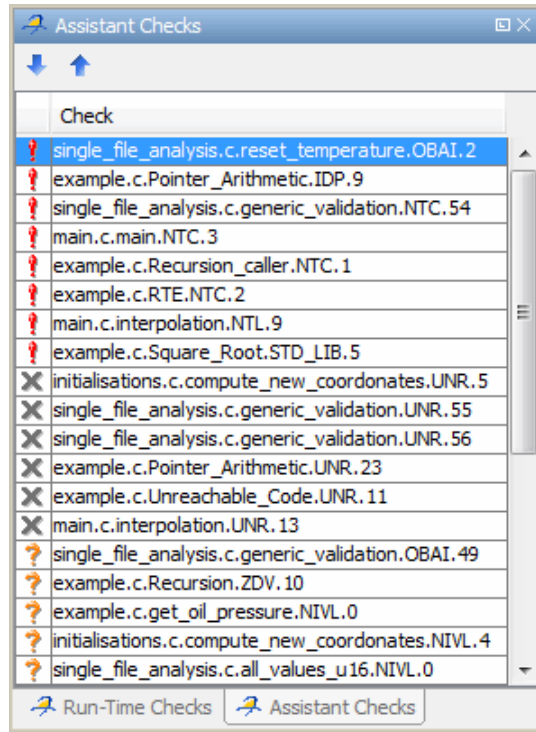
To perform a selective orange review, for example, at level 1:

- 1 On the Run-Time Checks perspective toolbar, move the Review Level slider to **1**.



- 2 Select the methodology for your project from the methodology menu, for example, **Methodology for C**.
- 3 Click the forward arrow  to select the first check to review.

The **Assistant Checks** tab shows the current check, and the Source pane displays the source code for this check.



- 4 Perform a quick code review on each orange check, spending no more than 5 minutes on each.

Your goal is to quickly 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.

See “Sources of Orange Checks” on page 9-6 for more information on each of these causes.

Note If an orange check is too complicated to explain quickly, it may be an inconclusive check caused by complex code structure, or the result of basic imprecision (approximation of the data set used for verification). These types of checks often take a substantial amount of time to understand.

5 If you cannot identify a cause within 5 minutes, move on to the next check.

Note Your goal is to find the maximum number of bugs in a short period of time. Therefore, you want to identify the source of as many orange checks as possible, while leaving more complex situations for future analysis.

6 Once you understand the cause of an orange check, use the Check Review pane to record your review.

a Select a **Classification** to describe the severity of the issue:

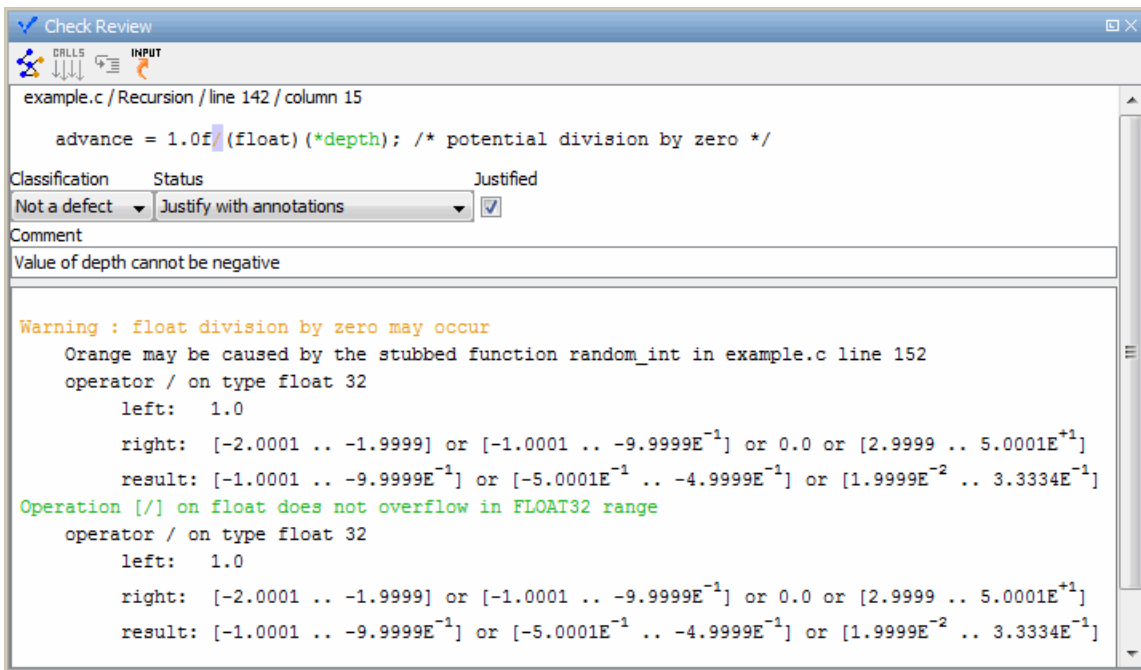
- High
- Medium
- Low
- Not a defect


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

Note You can also define your own statuses, which then appear in the user-defined acronym menu. For more information see “Defining Custom Status” on page 8-58.

- c Select the **Justified** check box to indicate that you have reviewed the check.
- d Enter a comment for the reviewed check in the text box.



- 7 Click the forward arrow  to navigate to the next check, and repeat steps 5 and 6.
- 8 Continue to click the forward arrow until you have reviewed all of the checks identified by the assistant.
- 9 Select **File > Save** to save your review comments.



Importing Review Comments from Previous Verifications

Once you have reviewed verification results for a module and saved your comments, you can import those comments into subsequent verifications of the same module, allowing you to avoid reviewing the same check twice.

To import review comments from a previous verification:

- 1** Open your most recent verification results in the Run-Time Checks perspective.
- 2** Select **Review > Import > Import Comments**.
- 3** Navigate to the folder containing your previous results.
- 4** Select the results (.RTE) file, then click **Open**.

The review comments from the previous results are imported into the current results, and the Import checks and comments report opens.

Once you import checks and comments, the **go to next check**  icon in assistant mode will skip any reviewed checks, allowing you to review only checks that you have not reviewed previously. If you want to view reviewed checks, click the **go to next reviewed check**  icon.

Note If the code has changed since the previous verification, the imported comments may not be applicable to your current results. For example, the justification for an orange check may no longer be relevant to the current code. To see the changes that affect your review comments, open the Import/Export Report.

For more information, see “Importing and Exporting Review Comments” on page 8-64.

Commenting Code to Provide Information During Review

You can place comments in your code to provide information on known issues. When reviewing results, you can use these comments to:

- Highlight and quickly understand issues identified in previous verifications
- Identify and skip previously reviewed checks.

This allows you to avoid reviewing the same check twice, and focus your review on new issues.

You must annotate your code before running a verification:

```
if (random_int() > 0)
{
    /* polyspace<RTE: NTC : Low : No Action Planned > This run-time error was discovered previously */
    Square_Root();
}

Unreachable_Code();
```

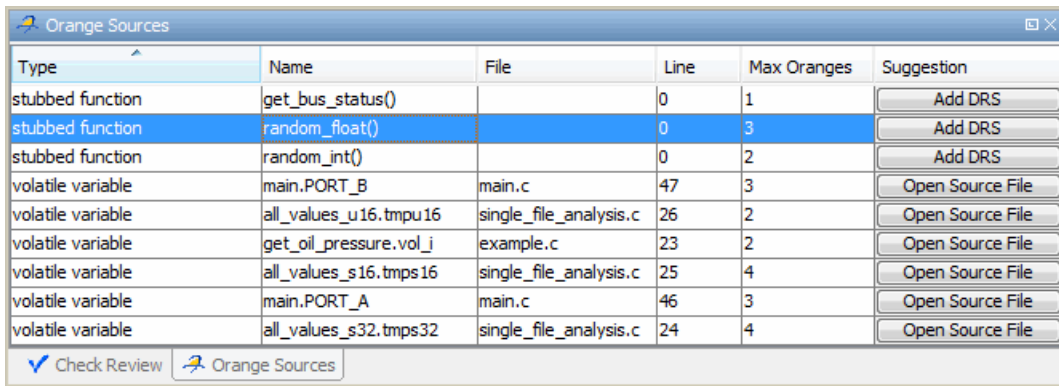
In the Run-Time Checks pane, the **Comment**, **Classification**, and **Status** columns display your code comments. In addition, in the **Justified** column, the check box is selected.

Procedural entities					Line	%	Justified	Comment	Classification	Status
example_project	5	64	8	99		95	<input type="checkbox"/>			
example.c	4	8	8	83	1	92	<input type="checkbox"/>			
Close_To_Zero ()			3	10	37	77	<input type="checkbox"/>			
Non_Infinite_Loop ()				11	66	100	<input type="checkbox"/>			
Pointer_Arithmetic ()	1	3	1	19	89	98	<input type="checkbox"/>			
RTE ()	1			3	222	100	<input type="checkbox"/>			
IRV.0				1	229		<input type="checkbox"/>			
IRV.1				1	231		<input type="checkbox"/>			
IRV.2				1	238		<input type="checkbox"/>			
NTC.3	1				241		<input checked="" type="checkbox"/>	This run-time error was discovered previously	Low	No Action Planned
Recursion ()			1	14	137	93	<input type="checkbox"/>			

For more information, see “Highlighting Known Coding Rule Violations and Run-Time Errors” on page 5-47.

Viewing Sources of Orange Checks

During a verification, the software identifies code that may be the source of orange checks. To view these possible sources of orange checks, in the Run-Time Checks perspective, Select **Window > Show/Hide View > Orange Sources**. The Orange Sources tab appears.



Type	Name	File	Line	Max Oranges	Suggestion
stubbbed function	get_bus_status()		0	1	Add DRS
stubbbed function	random_float()		0	3	Add DRS
stubbbed function	random_int()		0	2	Add DRS
volatile variable	main.PORT_B	main.c	47	3	Open Source File
volatile variable	all_values_u16.tmpu16	single_file_analysis.c	26	2	Open Source File
volatile variable	get_oil_pressure.vol_j	example.c	23	2	Open Source File
volatile variable	all_values_s16.tmps16	single_file_analysis.c	25	4	Open Source File
volatile variable	main.PORT_A	main.c	46	3	Open Source File
volatile variable	all_values_s32.tmps32	single_file_analysis.c	24	4	Open Source File

You 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 may resolve the orange check. See “Refining Data Range Specifications” on page 9-44.

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**.

To go to the code in the Source view, select the appropriate row in the Orange Sources display.


Working with Orange Checks Caused by Input Data

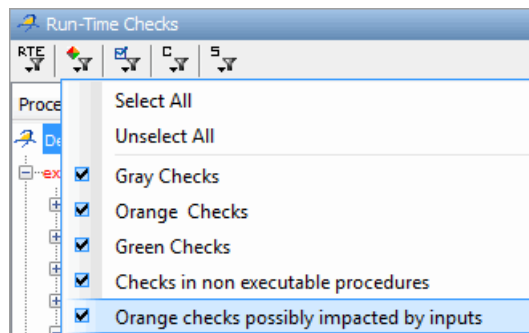
Polyspace verification tries to identify orange checks that may be caused by input data, in particular, checks that may be caused by:

- Stubs
- Main-generator
- Volatile variables
- External variables
- Absolute addresses

Filtering Orange Checks Caused by Inputs

The Run-Time Checks perspective allows you to hide orange checks that may be caused by inputs. To hide these orange checks:

- 1 In the Run-Time Checks pane toolbar, click the **Color filter** icon .
- 2 Clear the **Orange checks possibly impacted by inputs** option.

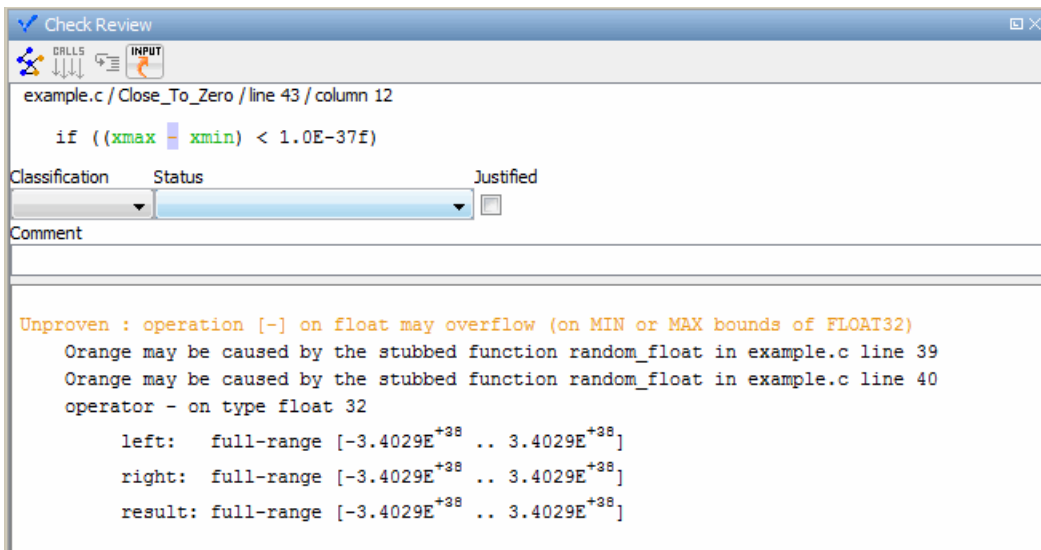


The software hides orange checks that may be caused by inputs.

Additional Information on Orange Checks Caused by Inputs

When the verification identifies orange checks that may be caused by inputs, it provides additional information about the cause of the orange check. This information can help you review results more efficiently.

To see information about the source of the orange check, click the check in the Source code view. Additional information appears in the Check Review pane.



Note Although the internal method used here and the internal method used by Review Level 0 may identify the same orange check, these methods generate different messages in the Check Review pane. In the two examples that follow, the message generated by the internal method for Review Level 0 is in blue:

- Orange may be caused by the stubbed function stub in *filename.c* line *number*

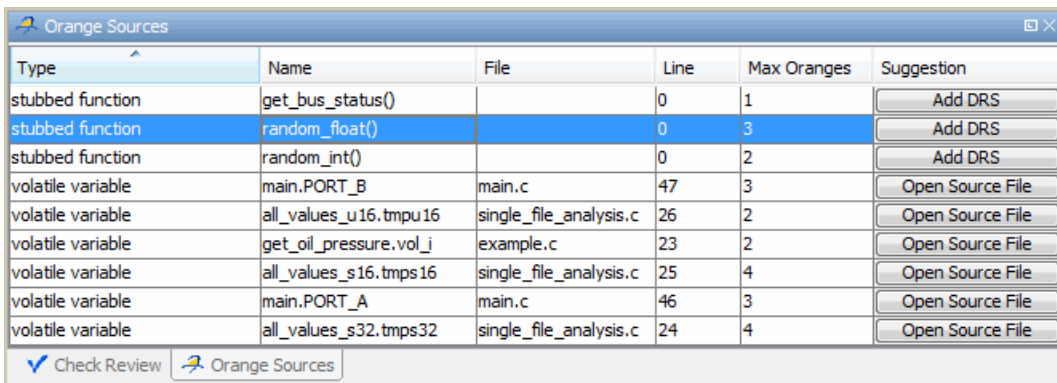
This check may be an issue related to unbounded input values

- Orange may be caused by the stubbed function stub in *filename.c* line *number*

This check may be a path-related issue, which is not dependent on input values

You can also view this information on the Orange Sources tab. Use one of the following:

- On the Check Review toolbar, click . The Orange Sources tab appears.



Type	Name	File	Line	Max Oranges	Suggestion
stubbed function	get_bus_status()		0	1	Add DRS
stubbed function	random_float()		0	3	Add DRS
stubbed function	random_int()		0	2	Add DRS
volatile variable	main.PORT_B	main.c	47	3	Open Source File
volatile variable	all_values_u16.tmpu16	single_file_analysis.c	26	2	Open Source File
volatile variable	get_oil_pressure.vol_j	example.c	23	2	Open Source File
volatile variable	all_values_s16.tmps16	single_file_analysis.c	25	4	Open Source File
volatile variable	main.PORT_A	main.c	46	3	Open Source File
volatile variable	all_values_s32.tmps32	single_file_analysis.c	24	4	Open Source File

- In the Source pane, when you right-click an orange check, you see **Show Orange Source Information** in the context menu if an input is a possible cause of the orange check.

```

42
43   if ((xmax - xmin) < 1.0E-37f)
44   {
45       y = 1.0
46   }
47   else
48   { /* div
49       y = (xi
50   }
51 }
52
53
54

```

To open the Orange Sources tab, select **Show Orange Source Information**.

- In the Procedural entities view, when you right-click an orange check, you see **Show Orange Source Information** in the context menu if an input is a possible cause of the orange check.

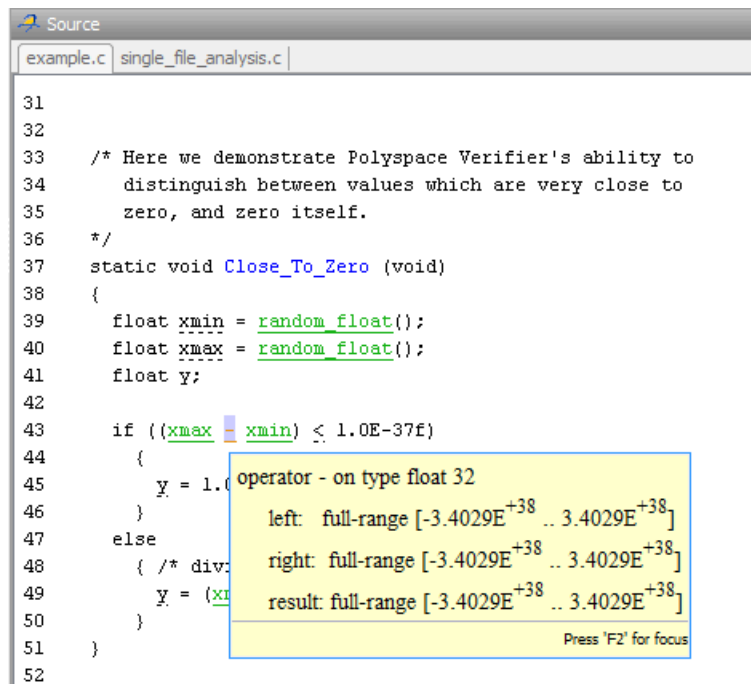
Node	4	2	5	15	81	1	12	12	Unp
example.c									
Close_To_Zero ()			3	2	40	37	12	12	Unp
OVFL.12			1			43	12	Unp	
OVFL.4			1			49	18	Unp	

To open the Orange Sources tab, select **Show Orange Source Information**.

For more information about the Orange Sources tab, see “Viewing Sources of Orange Checks” on page 9-39.

Refining Data Range Specifications

During verification, the software tries to identify code that may be the source of orange checks. You can see this information through the Orange Sources tab. See “Viewing Sources of Orange Checks” on page 9-39. For some orange checks, the software may display the suggestion **Add DRS** or **Modify DRS**, which indicates that you may be able to resolve the check by applying or refining data range specifications. Consider the following example, where the subtraction operator - is orange.



```


31
32
33  /* Here we demonstrate Polyspace Verifier's ability to
34     distinguish between values which are very close to
35     zero, and zero itself.
36  */
37  static void Close_To_Zero (void)
38  {
39     float xmin = random_float();
40     float xmax = random_float();
41     float y;
42
43     if ((xmax - xmin) <= 1.0E-37f)
44     {
45         y = 1.0E-37f;
46     }
47     else
48     { /* div
49         y = (x
50     }
51 }
52

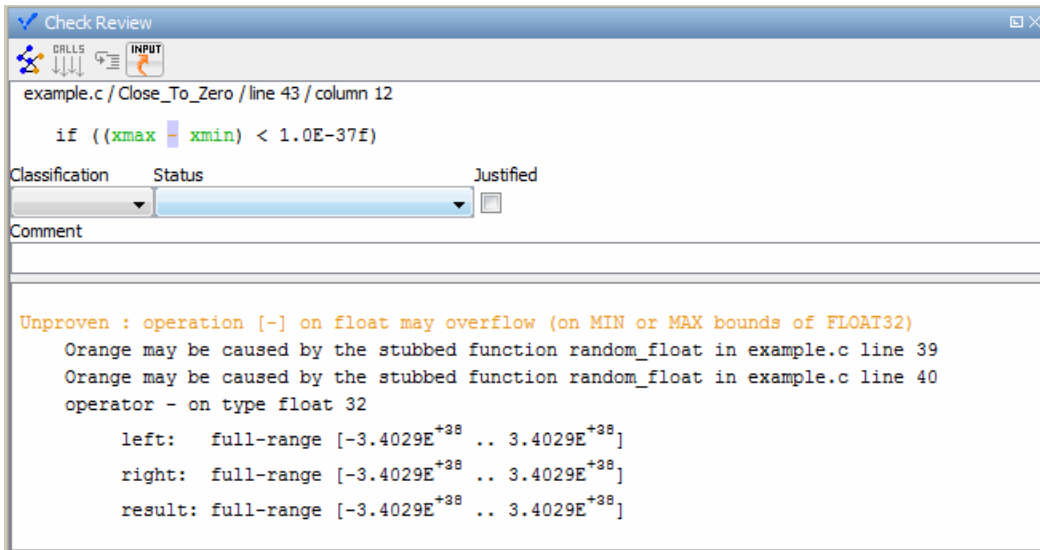
```

operator - on type float 32

left:	full-range [-3.4029E ⁺³⁸ .. 3.4029E ⁺³⁸]
right:	full-range [-3.4029E ⁺³⁸ .. 3.4029E ⁺³⁸]
result:	full-range [-3.4029E ⁺³⁸ .. 3.4029E ⁺³⁸]

Press 'F2' for focus

- 1 In the Source pane, click  to see details of the check in the Check Review pane.



example.c / Close_To_Zero / line 43 / column 12

```
if ((xmax - xmin) < 1.0E-37f)
```

Classification Status Justified

Comment

Unproven : operation [-] on float may overflow (on MIN or MAX bounds of FLOAT32)


Orange may be caused by the stubbed function random_float in example.c line 39

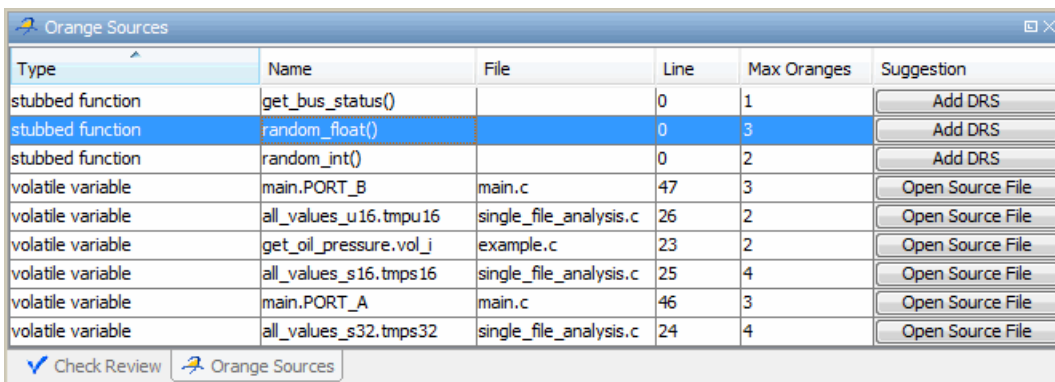
Orange may be caused by the stubbed function random_float in example.c line 40

operator - on type float 32

```
left:  full-range [-3.4029E+38 .. 3.4029E+38]
right: full-range [-3.4029E+38 .. 3.4029E+38]
result: full-range [-3.4029E+38 .. 3.4029E+38]
```

The operands `xmax` and `xmin` are both values returned by the function `random_float`. The check is orange because the result of the subtraction operation may lie outside the range of the data type `float 32`.

- 2 To open the Orange Sources tab, on the Check Review toolbar, click .



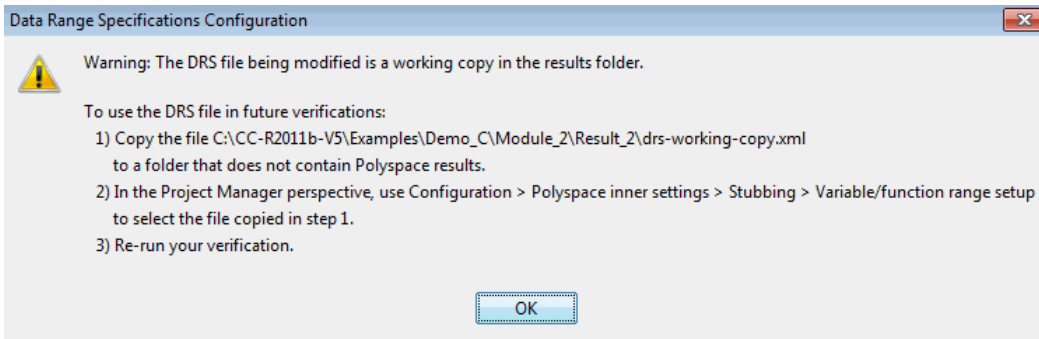
Type	Name	File	Line	Max Oranges	Suggestion
stubbed function	get_bus_status()		0	1	Add DRS
stubbed function	random_float()		0	3	Add DRS
stubbed function	random_int()		0	2	Add DRS
volatile variable	main.PORT_B	main.c	47	3	Open Source File
volatile variable	all_values_u16.tmpu16	single_file_analysis.c	26	2	Open Source File
volatile variable	get_oil_pressure.vol_j	example.c	23	2	Open Source File
volatile variable	all_values_s16.tmps16	single_file_analysis.c	25	4	Open Source File
volatile variable	main.PORT_A	main.c	46	3	Open Source File
volatile variable	all_values_s32.tmps32	single_file_analysis.c	24	4	Open Source File

Check Review Orange Sources

- 3** In the highlighted row, click **Add DRS**. The Data Range Configuration tab appears, displaying a copy of the existing DRS configuration file.

Name	File	Attributes	Type	Main Generator Called	Init Mode	Init Range	Initialize Pointer	Init Allocated	# Allocated Objects	Global Assert	Global Assert Range
Global Variables											
User Defined Functions											
Stubbed Functions											
SEND_MESSAGE()	include.h	extern									
get_bus_status()	example.c	extern									
random_float()	include.h	extern									
random_float.return	include.h		float32		PERMANENT	min..max					
random_int()	include.h	extern									
read_bus_status()	include.h	extern									
read_on_bus()	include.h	extern									
Non Applicable											

- 4** You can specify a range for the value returned by the function `random_float`. In the **Init Range** column, replace `min..max`, for example, by `-10..10`. The software writes the values to the file `drs-working-copy.xml` in the results folder. The first time you modify this file, the software warns you that you are not modifying the DRS configuration file used during the verification but a copy.



- 5** To rerun the verification with the modified DRS configuration file:
- a** Copy `drs-working-copy.xml` to a folder that does not contain Polyspace results.
 - b** In the Project Manager perspective, from the Configuration pane, use **Analysis Options > Polyspace inner settings > Stubbing > Variable/function range setup** to

select `drs-working-copy.xml`. See “Specifying Data Ranges Using Existing DRS Configuration” on page 4-64.

- c Rerun your verification. The software replaces the orange check by a green check.

```

37 static void Close_To_Zero (void)
38 {
39     float xmin = random_float();
40     float xmax = random_float();
41     float y;
42
43     if ((xmax - xmin) <= 1.0E-37E)
44     {
45         y = 1.
46     }
47     else
48     { /* div
49         y = (x
50     }
51 }
52

```

operator - on type float 32
left: [-1.0E⁺¹ .. 1.0E⁺¹]
right: [-1.0E⁺¹ .. 1.0E⁺¹]
result: [-2.0001E⁺¹ .. 2.0001E⁺¹]
Press 'F2' for focus

Performing an Exhaustive Orange Review

Up to 80% of orange checks can be resolved using multiple iterations of the process described in “Performing Selective Orange Review” on page 9-32. However, for extremely critical applications, you may want to resolve all orange checks. Exhaustive orange review is the process for resolving the remaining orange checks.

Note To display all orange checks, on the Run-Time Checks perspective toolbar, move the Review Level slider to **All**.

An exhaustive orange review is generally conducted later in the development process, during the unit testing or integration testing phase. The purpose of an exhaustive orange review is to analyze any orange checks that were not resolved during previous selective orange reviews, to identify potential bugs in those orange checks.

You must balance the time and cost of performing an exhaustive orange review against the potential cost of leaving a bug in the code. Depending on your quality objectives, you may or may not want to perform an exhaustive orange review.

Cost of Exhaustive Orange Review

During an exhaustive orange review, each orange check takes an average of 5 minutes to review. This means that 400 orange checks require about four days of code review, and 3,000 orange checks require about 25 days.

However, if you have already completed several iterations of selective orange review, the remaining orange checks are likely to be more complex than average, increasing the average time required to resolve them.

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 See “Sources of Orange Checks” on page 9-6 for more information on each of these causes.

Although you must review each check individually, there are some general guidelines to follow.

- 1 Start your review with the modules that have the highest selectivity in your application.

If the verification finds only one or two orange checks in a module 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 bugs. In general, these types of orange checks can also be resolved more quickly.

- 2 Next, examine files that contain a large percentage of orange checks compared to the rest of the application. These files may highlight design issues.

Often, when you examine modules containing the most orange checks, those checks will prove inconclusive. 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 9-49.

- 3 For all files you review, spend the first 10 minutes identifying checks that you can quickly categorize (such as potential bugs and data set issues), similar to what you do in a selective orange review.

Even after performing a selective orange review, a significant number of checks can be resolved quickly. These checks are more likely than average to reflect actual bugs.

- 4 Spend the next 40 minutes of each hour tracking more complex bugs.

If an orange check is too complicated to explain quickly, it may be an inconclusive check caused by complex code structure, or the result of basic imprecision (approximation of the data set used for verification). These types of checks often take a substantial amount of time to understand. See “Resolving Orange Checks Caused by Basic Imprecision” on page 9-50.

- 5 Depending on the results of your review, correct the code or comment it to identify the source of the orange check.

Inconclusive Verification and Code Complexity

The most interesting type of inconclusive check occurs when verification reveals that 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 reports 20 orange overflows (OVFL).

This scenario does not cause a real bug, but the development team may know that this variable caused trouble during development and earlier testing phases. Polyspace verification also identified a problem, suggesting that the code is poorly designed.

Resolving Orange Checks Caused by Basic Imprecision

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 basic imprecision, you may be able to 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 only assist you through the call tree and dictionary. The code needs to 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 9-6.

Automatically Testing Orange Code

In this section...

“Automatic Orange Tester Overview” on page 9-52
“How the Automatic Orange Tester Works” on page 9-53
“Selecting the Automatic Orange Tester” on page 9-54
“Starting the Automatic Orange Tester Manually” on page 9-56
“Reviewing Test Results After Manual Run” on page 9-60
“Refining Data Ranges with Automatic Orange Tester” on page 9-64
“Saving and Reusing Your Configuration” on page 9-68
“Exporting Data Ranges for Polyspace Verification” on page 9-68
“Configuring Compiler Options” on page 9-69
“Starting the Automatic Orange Tester Manually” on page 9-56
“Reviewing Test Results After Manual Run” on page 9-60

“Saving and Reusing Your Configuration” on page 9-68
“Exporting Data Ranges for Polyspace Verification” on page 9-68
“Configuring Compiler Options” on page 9-69
“Technical Limitations” on page 9-70

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.

Select the Automatic Orange Tester before you run a verification. See “Selecting the Automatic Orange Tester” on page 9-54. When the software

runs the Automatic Orange Tester at the end of static verification, the software may categorize some orange checks as potential run-time errors. See “Reviewing Checks at Level 0” on page 8-35. 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 Level 0 review, 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

...
```

Note 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 “Starting the Automatic Orange Tester Manually” on page 9-56.

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 Run-Time Checks perspective (see “What is an Orange Check?” on page 9-2).

If you select the Automatic Orange Tester (through the option `-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.

Selecting the Automatic Orange Tester

You must run a verification with the `-automatic-orange-tester` option selected, 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 Under **Analysis Options**, expand the **Polyspace inner settings** node.
- 2 Select the **Automatic Orange Tester** check box.

Name	Value	Internal name
Analysis options		
⊕ General		
⊕ Target/Compilation		
⊕ Compliance with standards		
⊖ Polyspace inner settings		
⊕ Run a verification unit by unit	<input type="checkbox"/>	-unit-by-unit
⊕ Generate a main	<input type="checkbox"/>	-main-generator
⊕ Stubbing		
⊕ Assumptions		
Continue with compile error	<input type="checkbox"/>	-continue-with-compile-error
Verification time limit		-timeout
⊖ Automatic Orange Tester	<input checked="" type="checkbox"/>	-automatic-orange-tester
Number of automatic tests	500	-automatic-orange-tester-tests-number
Maximum loop iterations	1000	-automatic-orange-tester-loop-max-iteration
Maximum test time	5	-automatic-orange-tester-timeout
Run verification in 32 or 64-bit mode	auto	-machine-architecture
Number of processes for multiple CPU core systems	4	-max-processes
Other options		
⊕ Precision/Scaling		
⊕ Multitasking		

3 Specify values for the following options:

- **Number of automatic tests** — Total number of test cases you want to run. Running more tests increases the chances of finding a run-time error, but takes more time to complete. 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. Default is 1000, which is also the maximum value that the software supports.

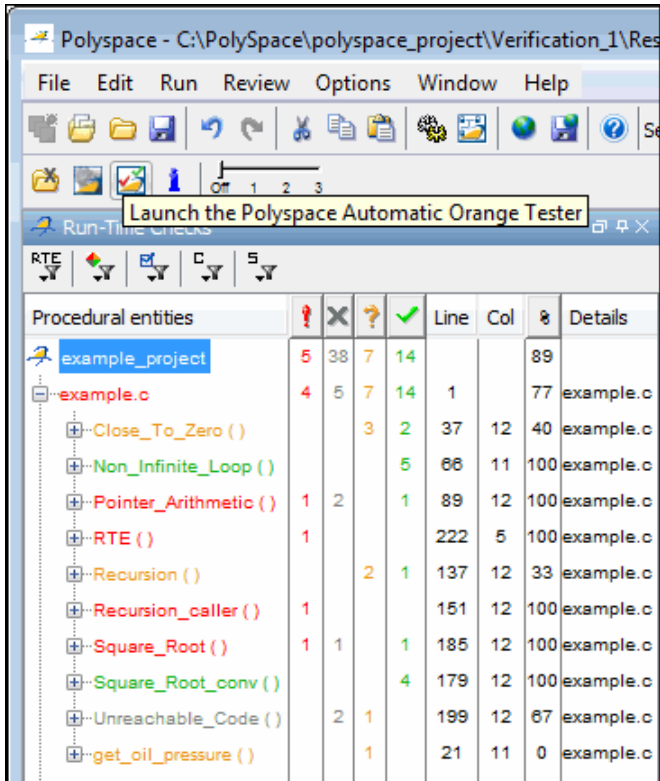
- **Maximum test time** — Maximum time (in seconds) allowed for a test before Automatic Orange Tester moves on to next test. Increasing test time reduces number of tests that time out, but increases total verification time. Default is 5 seconds. The maximum value that the software supports is 60.

Starting 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 “Selecting the Automatic Orange Tester” on page 9-54.

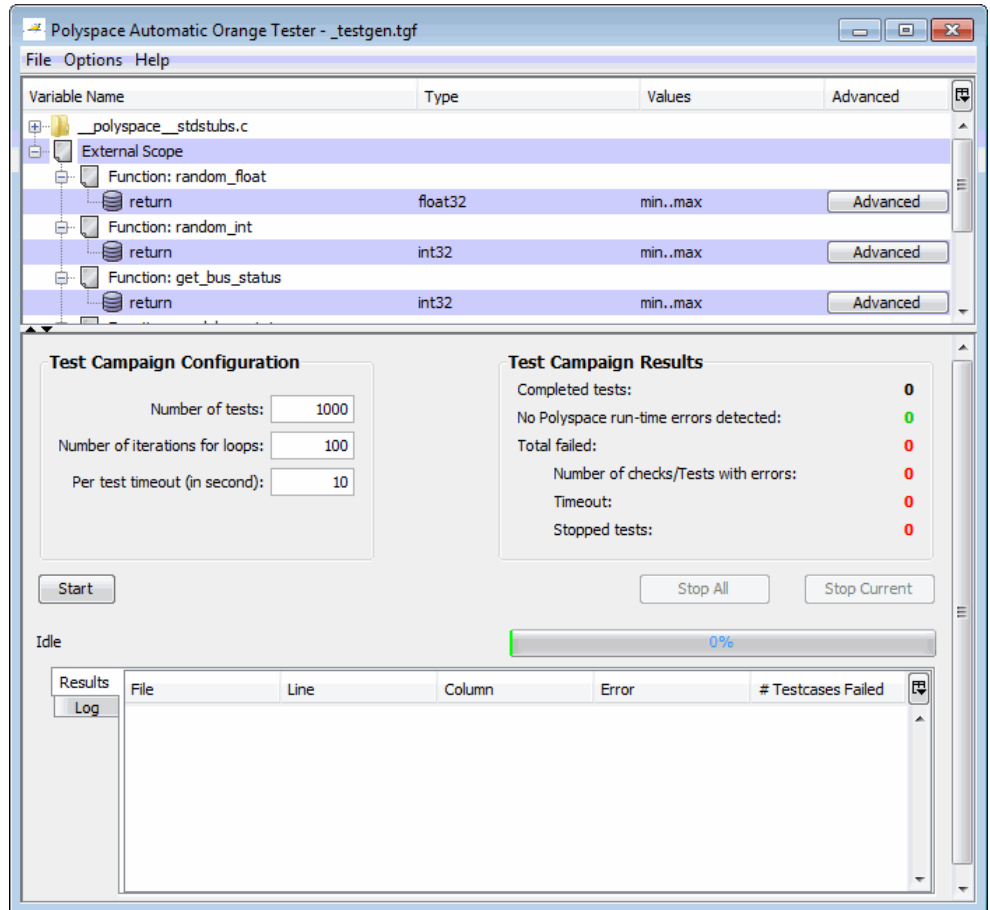
To start the Automatic Orange Tester:

- 1 Open your results in the Run-Time Checks perspective.



- 2 Click  (Launch the Polyspace Automatic Orange Tester) in the toolbar to open the Automatic Orange Tester.

The Automatic Orange Tester opens.



3 In the Test Campaign Configuration window, specify the following parameters:

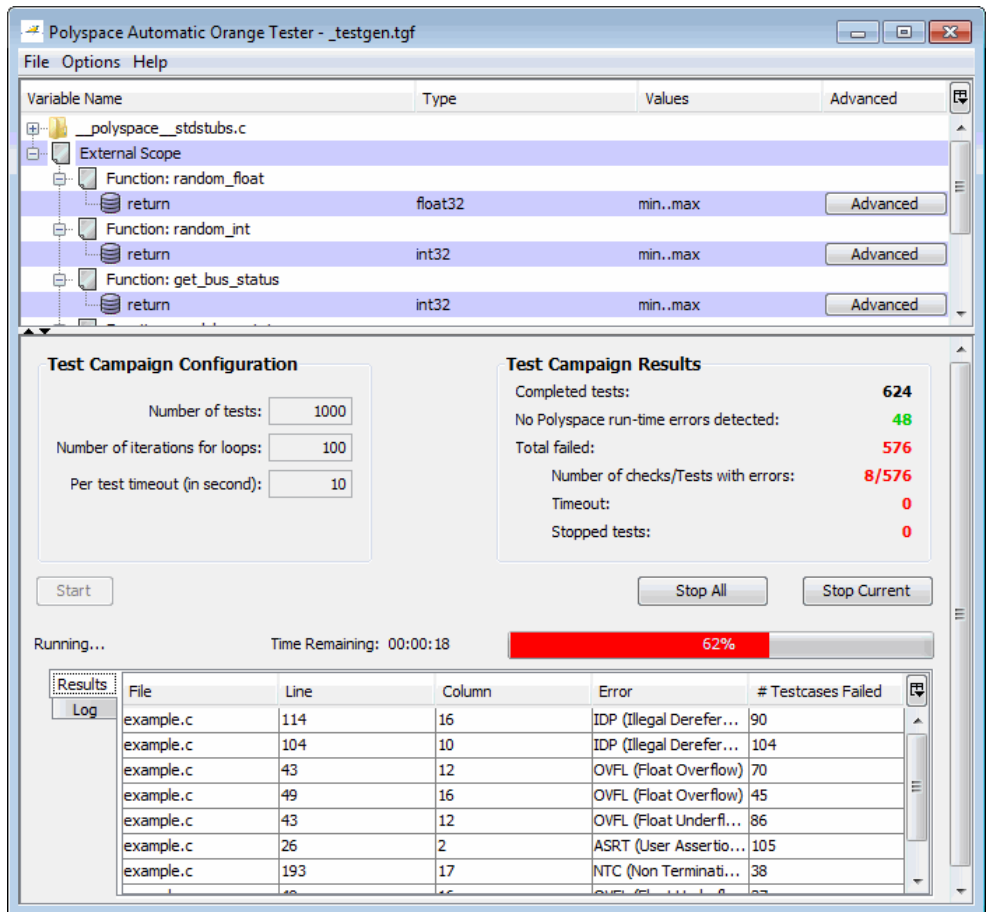
- **Number of tests** – Specifies the total number of test cases you want to run. Running more tests increases the chances of finding a runtime error, but also takes more time to complete.
- **Number of iterations for infinite loops** – Specifies the maximum number of loop iterations to perform before the Automatic Orange Tester identifies an infinite loop. A larger number of iterations decreases the

chances of incorrectly identifying an infinite loop, but also may take more time to complete.

- **Per test timeout** – 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 timeout, 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 completes:

- Click **Stop Current** to stop the current test and move on to the next one.
- Click **Stop All** to immediately stop all tests.

Reviewing Test Results After Manual Run

When testing is complete, the Automatic Orange Tester displays an overview of the testing results, along with detailed information about each failed test.

Test Campaign Configuration

Number of tests:

Number of iterations for loops:

Per test timeout (in second):

Test Campaign Results

Completed tests: **1000**

No Polyspace run-time errors detected: **73**

Total failed: **927**

Number of checks/Tests with errors: **8/927**

Timeout: **0**

Stopped tests: **0**

Start Stop All Stop Current

Test Completed Time Remaining: 00:00:00 **100%**

File	Line	Column	Error	# Testcases Failed
example.c	114	16	IDP (Illegal Derefer...	151
example.c	104	10	IDP (Illegal Derefer...	155
example.c	43	12	OVFL (Float Overflow)	119
example.c	49	16	OVFL (Float Overflow)	69
example.c	43	12	OVFL (Float Underfl...	150
example.c	26	2	ASRT (User Assertio...	166
example.c	193	17	NTC (Non Terminati...	57

Test Campaign Results

The Test Campaign Results window displays overview information about the results of your dynamic tests, including:

- **Completed tests** – Displays the total number of tests completed.
- **No Polyspace runtime errors detected** – Displays the number of tests that did not produce a runtime error.
- **Total failed** – Displays the number of tests that produced a runtime error.
- **Number of checks/Tests with errors** – Displays the number of Polyspace checks that produced at least one failed test, as well as the total number of tests that produced a runtime error.
- **Timeout** – Displays the number of tests that exceeded the specified **Per test timeout** limit.
- **Stopped tests** – The number of tests that were stopped manually.

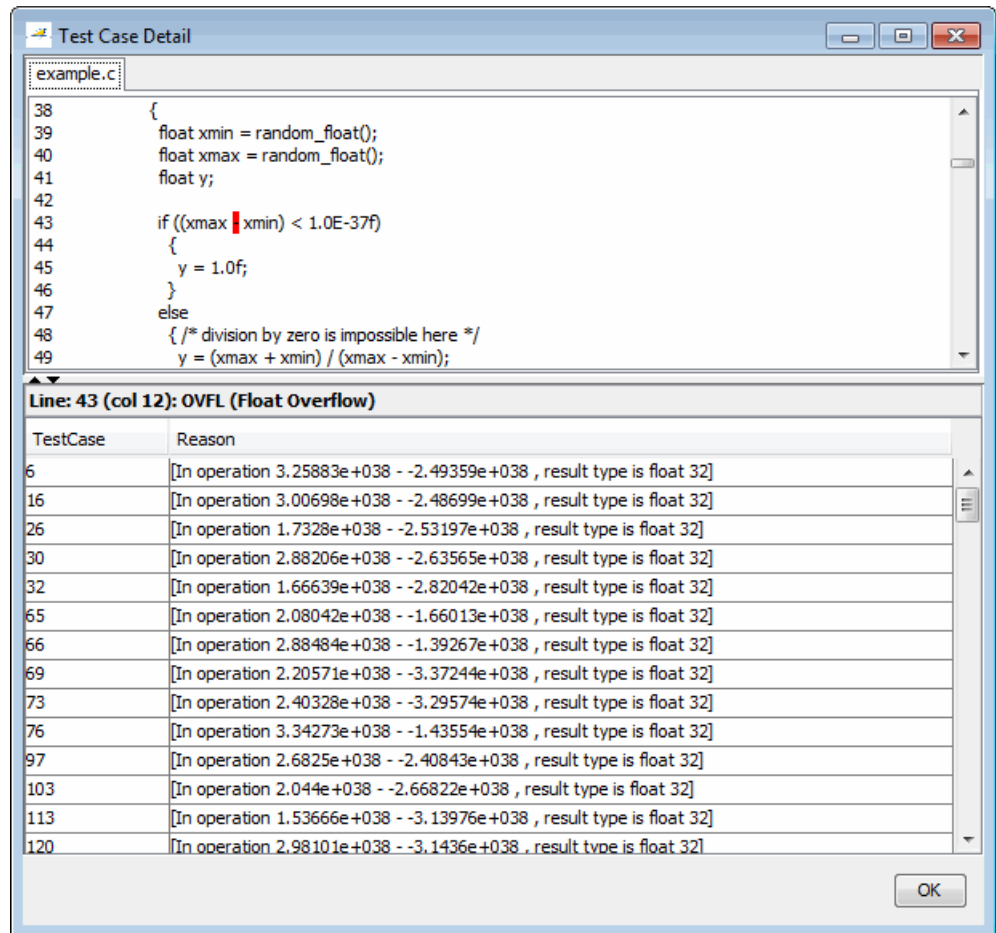
Use the Test Campaign Results Window to see an overall assessment of your test results, as well as to decide if you need to increase the **Per test timeout** value.

Results Table

The Results table displays detailed information about each failed test, to help you identify the cause of the runtime error. This information includes:

- The filename, 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.

In addition, You can view more details about any failed test by clicking on the appropriate 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. Since the Automatic Orange Tester performs runtime 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 if there is an actual bug in the code.

Log

The Log window displays a complete list of all 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 screenshot displays the Orange Tester interface. On the left, the 'Test Campaign Configuration' section includes input fields for 'Number of tests' (1000), 'Number of iterations for loops' (100), and 'Per test timeout (in second):' (10). A 'Start' button is located below these fields. On the right, the 'Test Campaign Results' section shows a summary: 'Completed tests: 1000', 'No Polyspace run-time errors detected: 73', 'Total failed: 927', 'Number of checks/Tests with errors: 8/927', 'Timeout: 0', and 'Stopped tests: 0'. Below the results are 'Stop All' and 'Stop Current' buttons. A progress bar at the bottom indicates 'Test Completed' and 'Time Remaining: 00:00:00' with a red bar showing '100%'. A 'Log' window is open, displaying a list of test results and a 'Test Summary' table.

Test Summary	
Number of tests	1000
Completed tests	1000
No Polyspace run-time errors detected	73
Total failed	927
Number of checks/Tests with errors	8/927
Timeout	0
Stopped tests	0

Test duration: 47 seconds
Test ended at: Tue Dec 21 17:23:36 EST 2010

The log file is also saved in the Polyspace-Instrumented folder with the following filename:

`TestGenerator_day_month_year-time.out`

Refining Data Ranges with Automatic Orange Tester

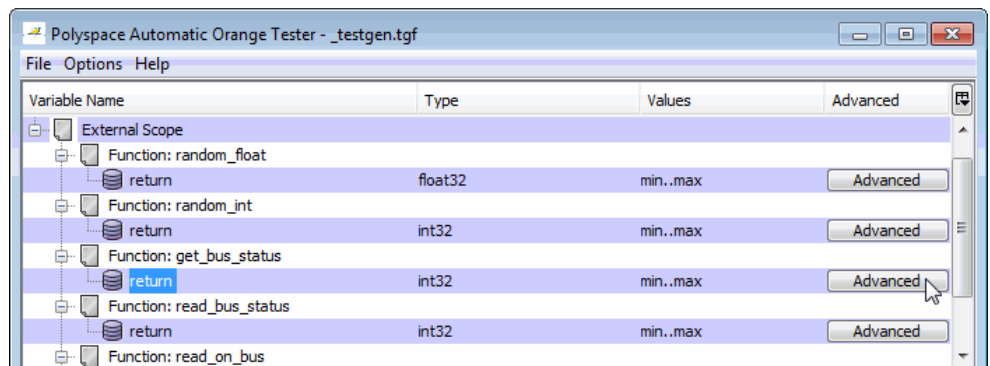
The Automatic Orange Tester allows you to specify ranges for external variables. This allows you to perform runtime tests using real-world values for your variables, rather than randomly selected values.

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

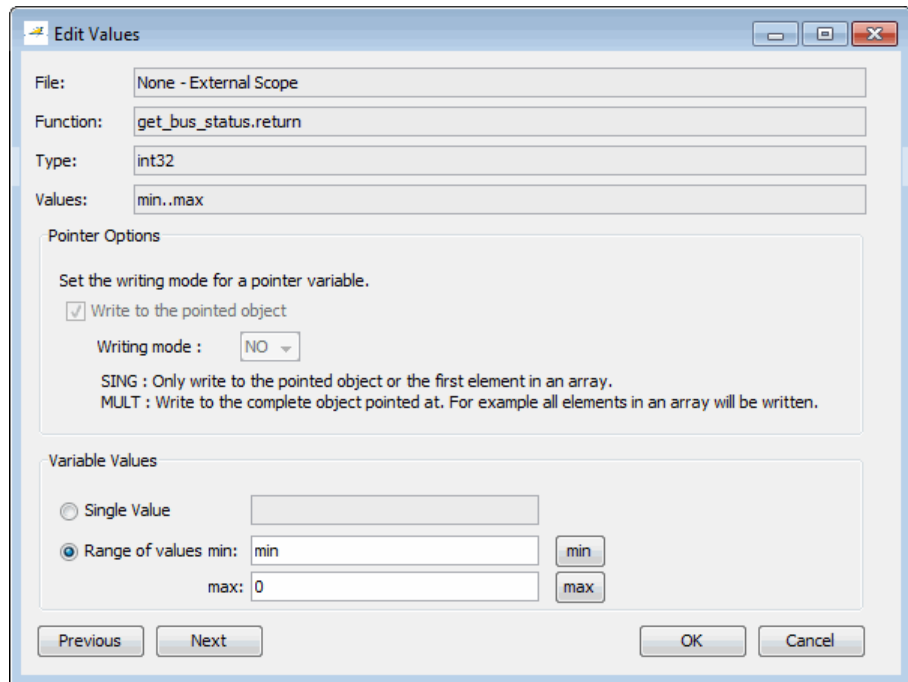
than purely theoretical problems. Once you set data ranges, you can export them to a DRS file for use in future verifications, reducing the number of orange checks in your results (see “Exporting Data Ranges for Polyspace Verification” on page 9-68).

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**. The Edit Values dialog box opens.



3 Set the appropriate values for the variable:

Single Value – Specifies a constant value for the variable.

Range of values, – Specifies a minimum and maximum value for the variable.

Note For pointers, you can also specify the writing mode:

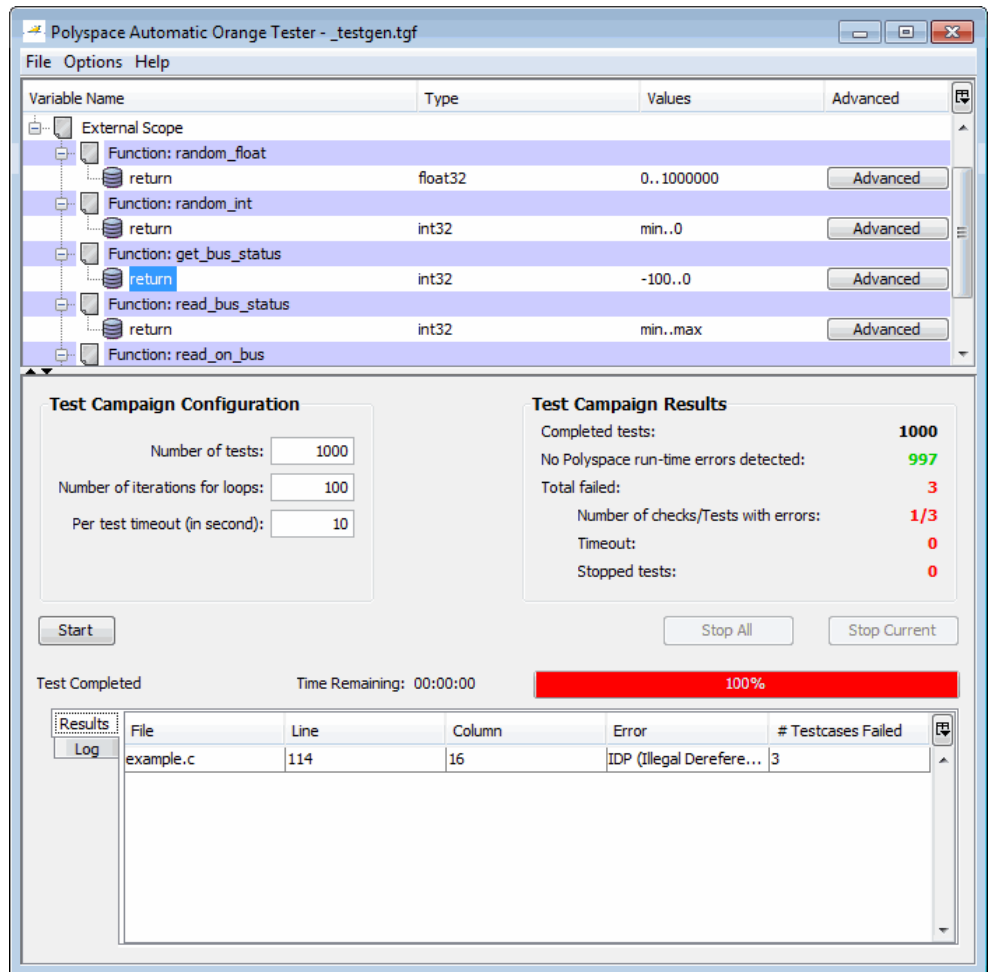
SING – The tests only write the object or first element in the array.

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 in on any actual code problems.

Saving and Reusing 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 an appropriate 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 appropriate `.tgf` file, then click **Open**.

The configuration is opened.

When you open a previously saved configuration, the **Log** window displays any 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).

Exporting 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. This allows you to reduce the number of orange checks identified in the Run-Time Checks perspective.

To export your data ranges:

- 1 Set the appropriate values for each variable you want to specify.
- 2 Select **File > Export DRS**.
- 3 Enter an appropriate name and click **Save**.

The DRS file is saved.

For information on using a DRS file for Polyspace verifications, see “Specifying Data Ranges for Variables and Functions (Contextual Verification)” on page 4-56.

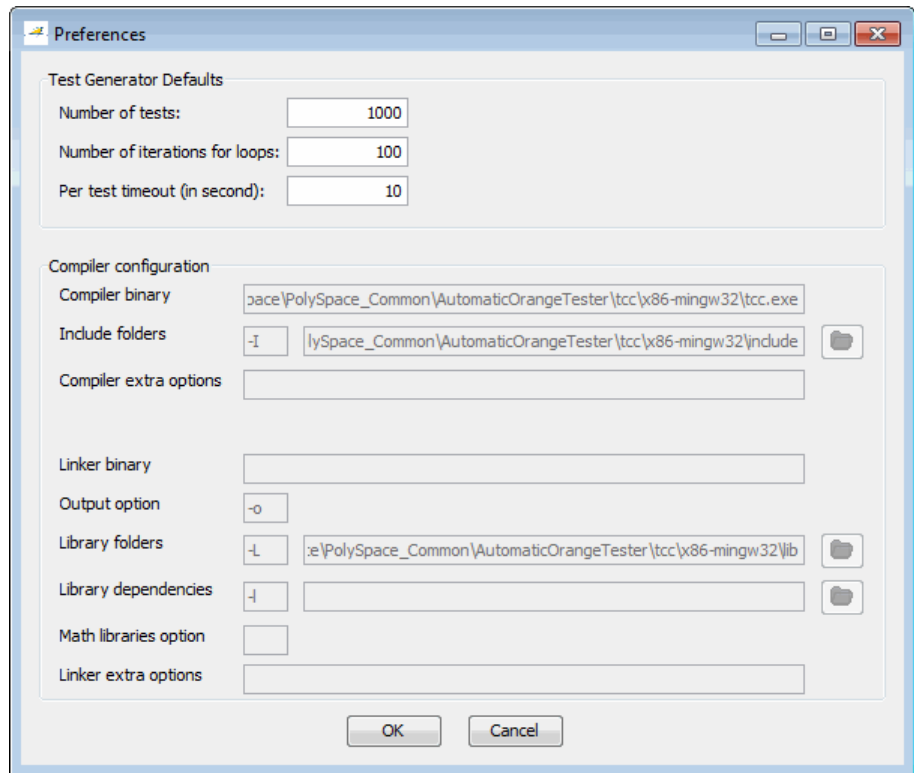
Configuring Compiler Options

On UNIX, Solaris, or Linux systems, you must configure your compiler and linker options before using the Automatic Orange Tester.

Note On Windows systems, the compiler options cannot be modified. You can only configure the library dependencies.

To set compiler and linker options:

- 1 Open the Automatic Orange Tester, as described above.
- 2 Select **Options > Configure**.
- 3 The Preferences dialog box opens.



4 Set the appropriate parameters for your compiler.

Technical Limitations

The Automatic Orange Tester has the following limitations:

- “Unsupported Polyspace Options” on page 9-70
- “Options with Restrictions” on page 9-71
- “Unsupported C Routines” on page 9-71

Unsupported Polyspace Options

The software does not support the following options with `-automatic-orange-tester`.

- `-div-round-down`
- `-char-is-16bits`
- `-short-is-8bits`

In addition, with the Automatic Orange Tester, the software does not support Global asserts in the code of the form `Pst_Global_Assert(A,B)` .

Options with Restrictions

You must not specify the following with `-automatic-orange-tester`:

- `-target [c18 | tms320c3c | x86_64 | sharc21x61]`
- `-data-range-specification` (in global assert mode)

In addition, the `-target mcpu` option must be used 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

Day to Day Use

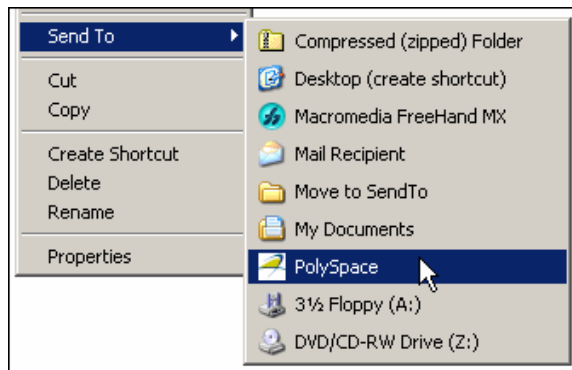
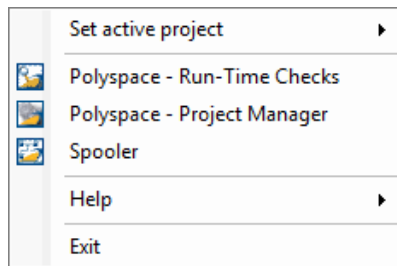
- “Polyspace In One Click Overview” on page 10-2
- “Using Polyspace In One Click” on page 10-3

Polyspace In One Click Overview

Most developers verify the same files multiple times (writing new code, unit testing, integration), and usually need to run verifications on multiple project files using the same set of options. In a Microsoft Windows environment, Polyspace In One Click provides a convenient way to streamline your work when verifying several files using the same set of options.

Once you have set up a project file with the options you want, you designate that project as the *active project*, and then send the source files to Polyspace software for verification. You do not have to update the project with source file information.

On a Windows systems, the plug-in provides a Polyspace Toolbar in the Windows Taskbar, and a **Send To** option on the desktop pop-up menu:



Using Polyspace In One Click

In this section...

“Polyspace In One Click Workflow” on page 10-3

“Setting the Active Project” on page 10-3

“Launching Verification” on page 10-5

“Using the Taskbar Icon” on page 10-7

Polyspace In One Click Workflow

Using Polyspace In One Click involves two steps:

- 1 Setting the active project.
- 2 Sending files to Polyspace software for verification.

Setting the Active Project

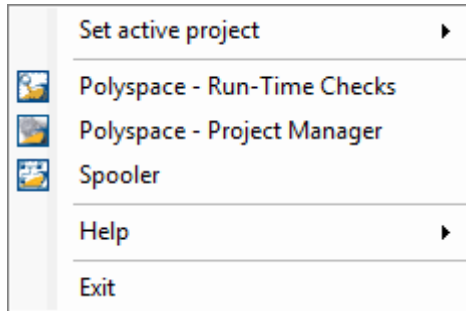
The active project is the project that Polyspace In One Click uses to verify the files that you select. Once you have set an active project, it remains active until you change the active project. Polyspace software uses the analysis options from the project; it does not use the source files or results folder from the project.

To set the active project:

- 1 In the taskbar area of your Windows desktop, right-click the Polyspace In One Click icon:

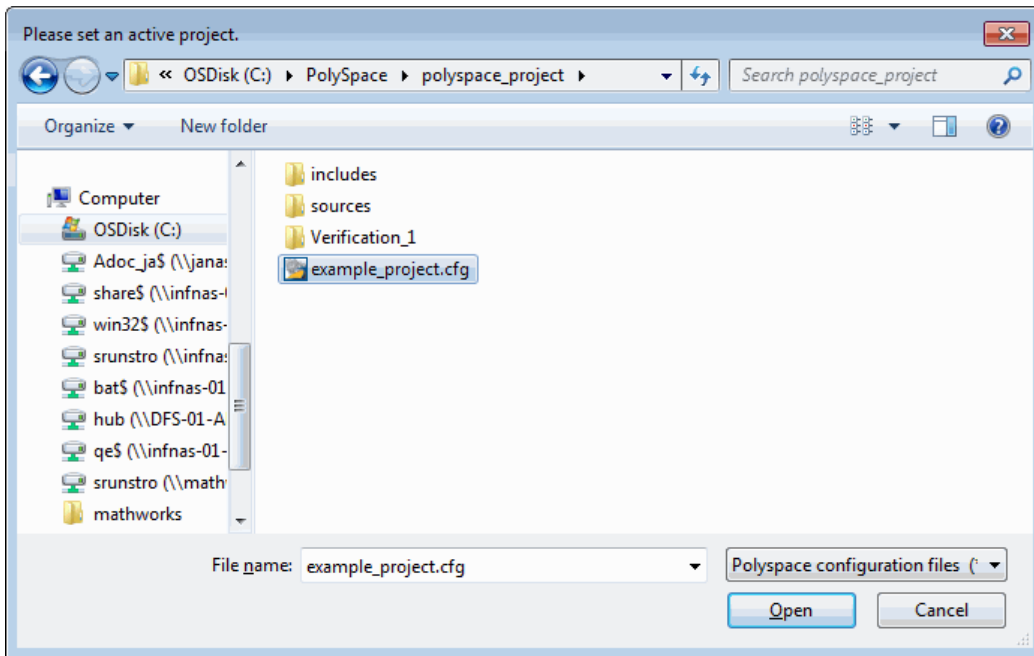


The context menu appears.



2 Select **Set active project** > **Browse** from the menu.

The Please set an active project dialog box opens:



3 Select the project you want to use as the active project.

4 Click **Open** to apply the changes and close the dialog box.

Note You can also set the active project by right-clicking on a project file (.cfg or .dsk) file and selecting **Send To > Polyspace**.

Launching Verification

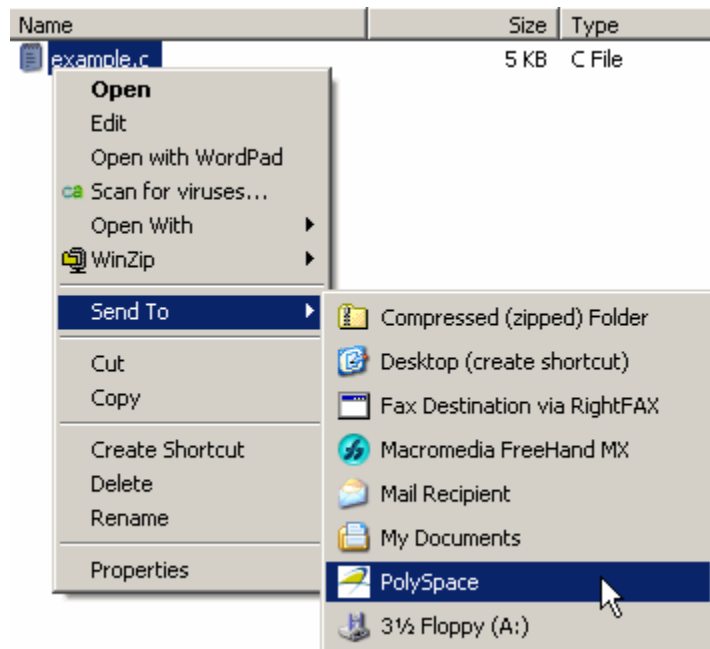
Polyspace in One Click allows you to send multiple files to Polyspace software for verification.

To send a file to Polyspace software for verification:

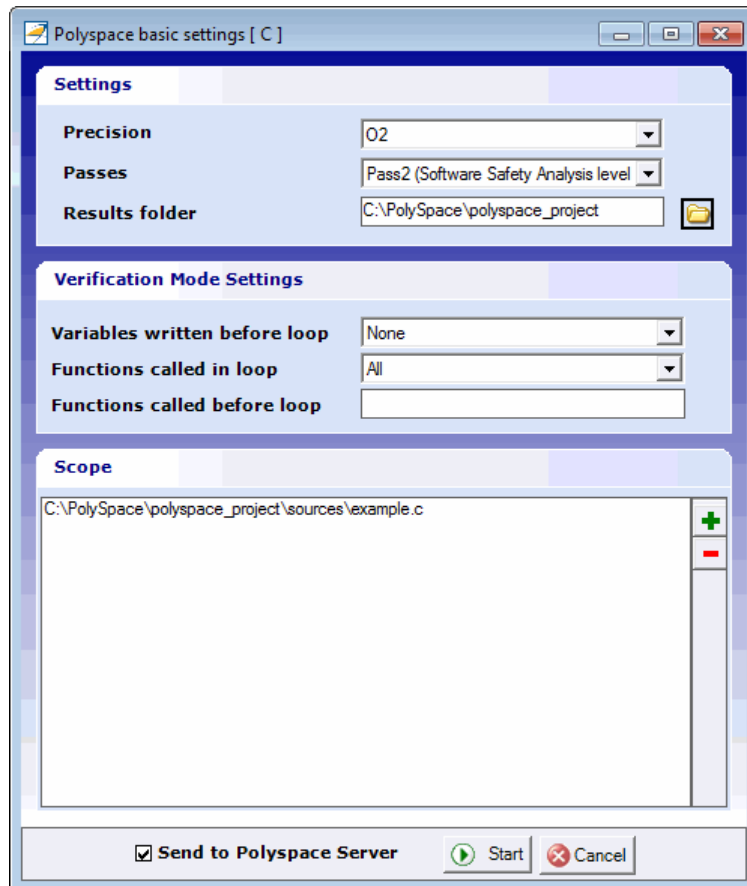
- 1 Navigate to the folder containing the source files you want to verify.
- 2 Right-click the file you want to verify.

The context menu appears.

- 3 Select **Send To > Polyspace**.



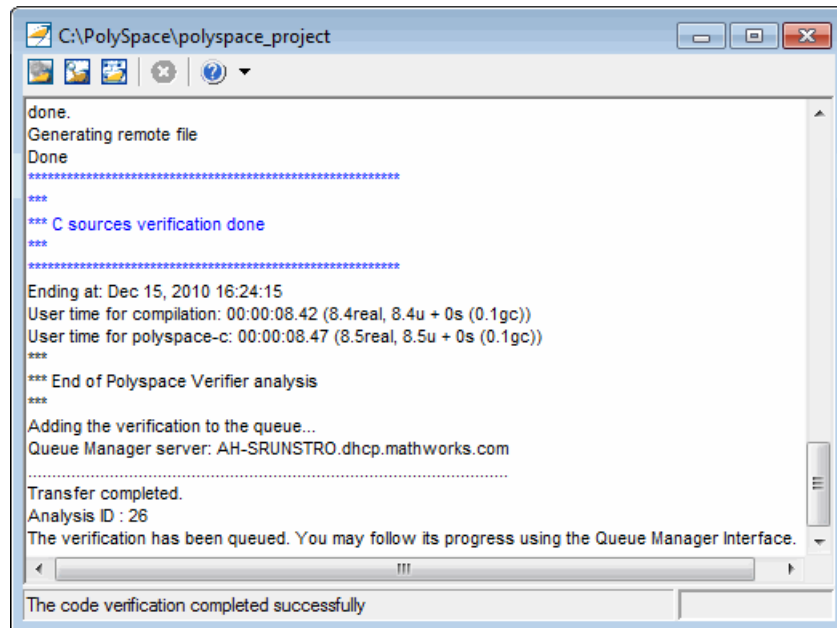
The **Polyspace basic settings** dialog box appears.



Note The options you specify the basic settings dialog box override any options set in the configuration file. These options are also preserved between verifications.

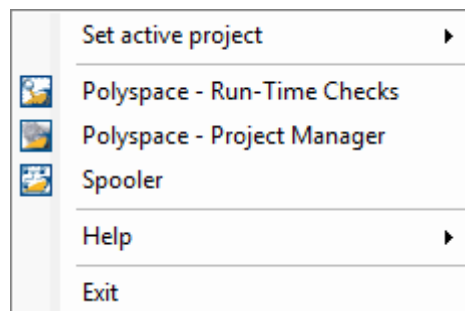
- 4** Enter the appropriate parameters for your verification.
- 5** Click **Start**.

The verification starts and the verification log appears.



Using the Taskbar Icon

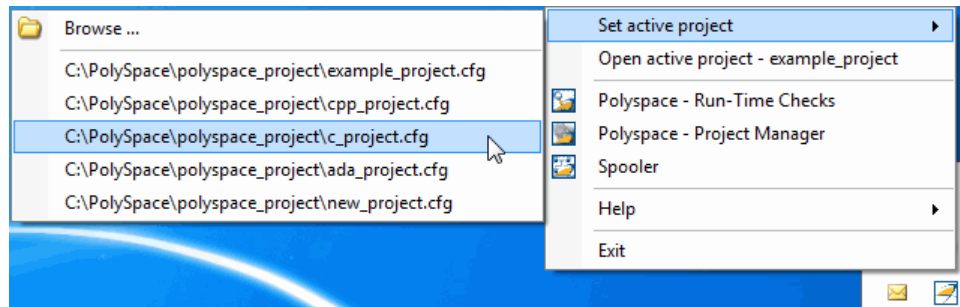
The Polyspace in One Click Taskbar icon allows you to access various software features.



Click the Polyspace Taskbar Icon, then select one of the following options:

- **Set active project** — Allows you to set the active configuration file. Before you start, you have to choose a Polyspace configuration file which contains the common options. You can choose a template of a previous project and move it to your working folder.

A standard file browser allows you to choose the configuration file. If you have multiple configuration files, you can quickly switch between them using the browse history.



Note No configuration file is selected by default. You can create an empty file with a .cfg extension.

- **Open active project** — Opens the active configuration file. This allows you to update the project using the Polyspace Verification Environment Project Manager perspective. It allows you to specify all Polyspace common options, including directives of compilation, options, and paths of standard and specific headers. It does not affect the precision of a verification or the results folder.
- **Polyspace - Run-Time Checks** — Opens the Polyspace Verification Environment, Run-Time Checks perspective. This allows you to review verification results in the standard graphical interface.
- **Polyspace - Project Manager** — Opens the Polyspace Verification Environment, Project Manager perspective. This allows you to launch a verification using the standard Polyspace graphical interface.

- **Spooler** — Opens the Polyspace Queue Manager Interface. If you selected a server verification in the “Polyspace Preferences” dialog box, the spooler allows you to follow the status of the verification.

Checking Coding Rules

- “Overview of Polyspace Code Analysis” on page 11-2
- “Polyspace MISRA C Checker Overview” on page 11-2
- “Setting Up Coding Rules Checking” on page 11-5
- “Viewing Coding Rules Checker Results” on page 11-18
- “Coding Rules Assistant” on page 11-28
- “Software Quality Objective Subsets of Coding Rules (C)” on page 11-33
- “Supported Coding Rules” on page 11-38

Overview of Polyspace Code Analysis

In this section...
“Code Analysis Overview ” on page 11-2
“Polyspace MISRA C Checker Overview” on page 11-2
“Polyspace MISRA C++ Checker Overview” on page 11-3
“Polyspace JSF C++ Checker Overview” on page 11-4

Code Analysis Overview

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 both reduce the number of orange checks in your verification results, and improve the quality of your code. Coding rules are the most efficient way to reduce orange checks.

While creating a project, you specify both the coding standard, and individual rules to enforce. Polyspace software then performs code analysis before starting verification, and reports any errors or warnings in the Coding Rules perspective.

Note If any source files in the verification do not compile, coding rules checking will be incomplete. The coding rules checker results will not contain results for files that did not compile, and may not contain full results for the files that did compile, since some rules are checked only after compilation completes.

Polyspace MISRA C Checker Overview

The Polyspace MISRA C checker helps you comply with the MISRA C 2004 coding standard.¹⁰

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

The MISRA C checker enables Polyspace software to provide messages when MISRA C rules are violated. Most messages are reported during the compile phase of a verification.

The MISRA C checker can check nearly all of the **142** MISRA C:2004 rules. In addition to the MISRA rules, the software checks one additional rule (15.0), to improve precision.

There are two subsets of MISRA coding rules that can have a direct or indirect impact on the selectivity (reliability percentage) of your verification results. You can select these subsets directly when setting up MISRA C Checking. These subsets are defined in “Software Quality Objective Subsets of Coding Rules (C)” on page 11-33.

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

Polyspace MISRA C++ Checker Overview

The Polyspace MISRA C++ checker helps you comply with the MISRA C++:2008 coding standard.¹¹

The Polyspace MISRA C++ checker enables Polyspace software to provide messages when MISRA C++ rules are not respected. Most messages are reported during the compile phase of a verification. The MISRA C++ checker can check 167 of the 228 MISRA C++ coding rules .

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.

Polyspace JSF C++ Checker Overview

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, and are designed to improve the robustness of C++ code, and improve maintainability.

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

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.

Setting Up Coding Rules Checking

In this section...

- “Activating the MISRA C Checker” on page 11-5
- “Activating the MISRA C++ Checker” on page 11-7
- “Activating the JSF C++ Checker” on page 11-7
- “Creating a MISRA C Rules File” on page 11-8
- “Creating a C++ Coding Rules File” on page 11-10
- “Excluding Files from Rules Checking” on page 11-12
- “Excluding All Include Folders from Coding Rules Checking” on page 11-13
- “Redefine Data Types as Boolean” on page 11-14
- “Configuring Text and XML Editors” on page 11-14
- “Commenting Code to Indicate Known Rule Violations” on page 11-16

Activating the MISRA C Checker

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.

To set the MISRA C checking option:

- 1** In the Analysis options part of the Configuration pane, expand the **Compliance with standards** option.

The Compliance with standards options appear.

- 2** Select the **Check MISRA C rules** check box.

- 3** Expand the **Check MISRA C rules** option.

Three options, **MISRA C rules configuration**, **Files and folders to ignore** and **Effective boolean types**, appear.

Name	Value	Internal name
Keep all preliminary results files	<input type="checkbox"/>	-keep-all-files
Calculate code metrics	<input type="checkbox"/>	-code-metrics
Report Generation	<input type="checkbox"/>	
Report template name	Developer	-report-template
Output format	RTF	-report-output-format
Target/Compilation		
Compliance with standards		
Code from DOS or Windows filesystem	<input checked="" type="checkbox"/>	-dos
Embedded assembler		
Strict	<input type="checkbox"/>	-strict
Permissive	<input type="checkbox"/>	-permissive
Check MISRA C rules	<input checked="" type="checkbox"/>	
MISRA C rules configuration	all-rules	-misra2
Files and folders to ignore		-includes-to-ignore
Effective boolean types		-boolean-types
Dialect support	none	-dialect

4 In the **MISRA C rules configuration** drop-down list, select which MISRA C rules to check:

- **all-rules** – Checks all possible MISRA C coding rules. All violations are reported as warnings.
- **SQO-subset1** – Checks a subset of MISRA C rules that have a direct impact on the selectivity of verification. All violations are reported as warnings. For more information, see “SQO Subset 1 – Coding Rules with a Direct Impact on Selectivity” on page 11-33.
- **SQO-subset2** – Checks a second subset of MISRA C rules that have an indirect impact on the selectivity of verification, as well as the rules contained in SQO-subset1. All violations are reported as warnings. For more information, see “SQO Subset 2 – Coding Rules with an Indirect Impact on Selectivity” on page 11-35.
- **custom** – Checks a specified set of coding rules. When you select this option, you must create a rules file that specifies which rules to check and whether to report an error or warning for violations of each rule. For more information, see “Creating a MISRA C Rules File” on page 11-8.

5 Specify any files to exclude from MISRA C checking. For more information, see “Excluding Files from Rules Checking” on page 11-12.

- 6 Specify data types that you want Polyspace to consider as Boolean. For more information, see “Redefine Data Types as Boolean” on page 11-14.

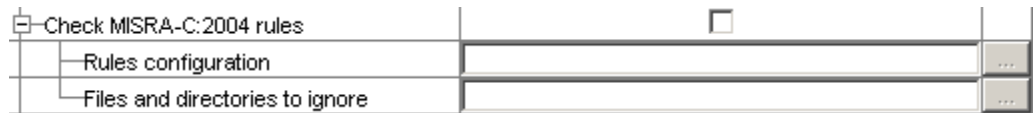
Activating the MISRA C++ Checker

You activate the MISRA C++ checker using the options `misra-cpp` and `includes-to-ignore`. These options can be set at the command line, or through the Project Manager user interface.

To activate the MISRA C++ Checker:

- 1 Open the project you want to use in the Project Manager perspective.
- 2 In the Analysis options part of the Configuration pane, select **Compliance with standards > Coding rules checker**.

The software displays the two MISRA C++ rules checker options:
`misra-cpp` and `includes-to-ignore`.



These options allow you to specify which coding rules to check and any files to exclude from the checker.

- 3 Select the **Check MISRA C++ rules** check box.

Activating the JSF C++ Checker

You activate the JSF C++ checker using the options `jsf-coding-rules` and `includes-to-ignore`. These options can be set at the command line, or through the Project Manager user interface.

To activate the JSF C++ Checker:

- 1 Open the project you want to use in the Project Manager perspective.
- 2 In the Analysis options part of the Configuration pane, eselect **Compliance with standards > Coding rules checker**.

The software displays the JSF C++ rules checker options, `-jsf-coding-rules` and `-includes-to-ignore`.

<input type="checkbox"/> Coding rules checker			
<input checked="" type="checkbox"/> Check JSF C++ rules	<input checked="" type="checkbox"/>		
JSF C++ rules configuration		...	<code>-jsf-coding-rules</code>
<input type="checkbox"/> Check MISRA C++ rules	<input type="checkbox"/>		
MISRA C++ rules configuration		...	<code>-misra-cpp</code>
Files and folders to ignore		...	<code>-includes-to-ignore</code>


These options allow you to specify which coding rules to check and any files to exclude from the checker.

- 3 Select the **Check JSF C++: rules** check box.

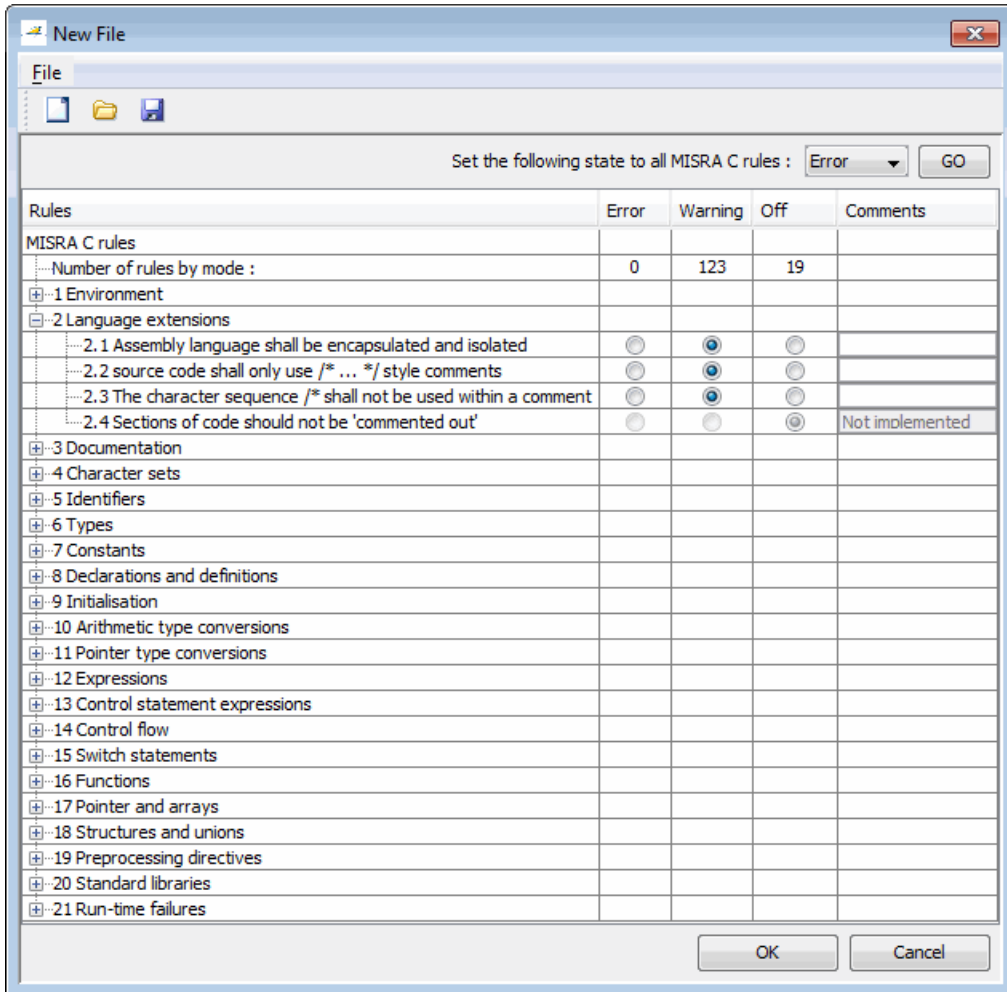
Creating a MISRA C Rules File

If you specify `custom` in the **MISRA C rules configuration** drop-down list, you must provide a rules file to specify which MISRA C rules to check.

To create a custom rules file:

- 1 In the **MISRA C rules configuration** drop-down list, select `custom`.
- 2 Click the browse button  to the right of the **MISRA C rules configuration** option.

The New File window opens, displaying a table of rules.



3 For each rule, you specify one of these states:

State	Causes the verification to...
Error	End after the compile phase when this rule is violated.
Warning	Display warning message and continue verification when this rule is violated.
Off	Skip checking of this rule.

Note The default state for most rules is **Warning**. The state for rules that have not yet been implemented is **Off**. Some rules always have state **Error** (you cannot change the state of these).

4 Click **OK** to save the rules and close the window.

The **Save as** dialog box opens.


5 In **File**, enter a name for the rules file.

6 Click **OK** to save the file and close the dialog box.

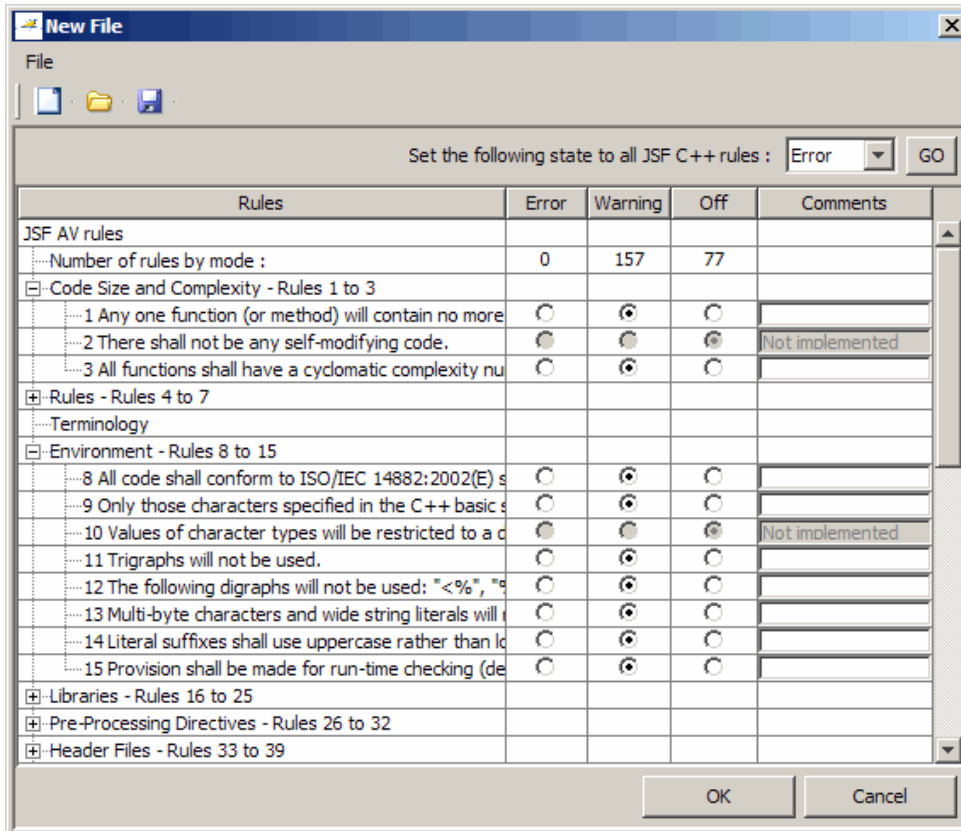
Creating a C++ Coding Rules File

You must have a rules file to run a verification with MISRA C++ or JSF++ checking. You can use an existing file or create a new one.

To create a new rules file:

1 Click the browse button  to the right of the **JSF C++ rules configuration** or **MISRA C++ rules configuration** option.

The New File window opens, allowing you to create a new rules file, or open an existing file.



2 For each MISRA C++ or JSF++ rule, specify one of these states:

State	Causes the verification to...
Error	End after the compile phase when this rule is violated.
Warning	Display warning message and continue verification when this rule is violated.
Off	Skip checking of this rule.

Note The default state for most rules is **Warning**. The state for rules that have not yet been implemented is **Off**. Some rules always have state **Error** (you cannot change the state of these).

3 Click **OK** to save the rules and close the window.

The **Save as** dialog box opens.

4 In **File**, enter a name for your rules file.

5 Click **OK** to save the file and close the dialog box.


Note If your project uses a dialect other than ISO, some JSF++ coding rules may not be completely checked. For example, AV Rule 8: “All code shall conform to ISO/IEC 14882:2002(E) standard C++.”

Excluding Files from Rules Checking

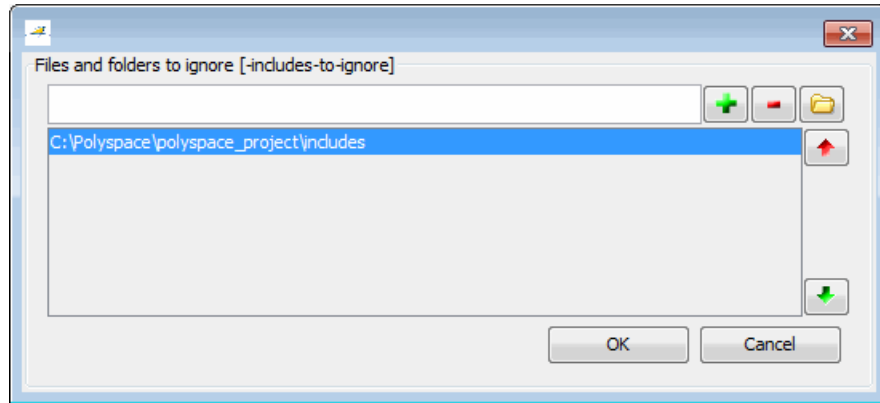
You can exclude individual files or folders from coding rules checking. For example, you might want to exclude include files.


Note You can also exclude all include folders from MISRA C checking. See “Excluding All Include Folders from Coding Rules Checking” on page 11-13.

To exclude files from coding rules checking:

1 Click the browse button  to the right of the **Files and folders to ignore** option.

The Files and folders to ignore dialog box opens.



- 2 Click the folder icon .
- 3 Select the files or folders (such as include files) you want to ignore.
- 4 Click **OK**.

The selected files appear in the list of files to ignore.

- 5 Click **OK** to close the dialog box.

Excluding All Include Folders from Coding Rules Checking

You can exclude all include folders from MISRA C checking. If you are checking a large code base (especially when using standard or Visual headers), excluding all include folders can significantly improve the speed of code analysis.

To exclude all include folders from MISRA C checking:

- 1 In the Analysis options section of the Project Manager, select **Compliance with standards > Check MISRA C rules > Files and folders to ignore**.
- 2 Enter `all`.


All include folders are excluded from checking.

Redefine Data Types as Boolean

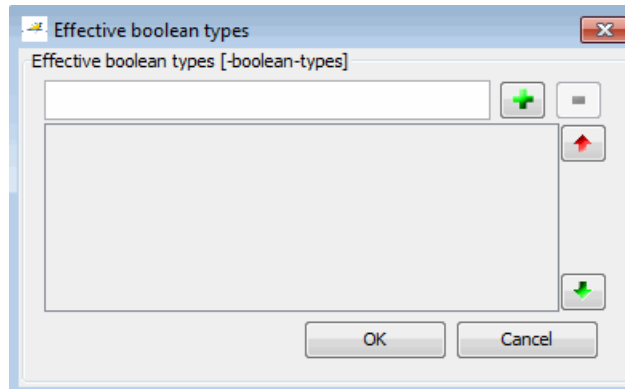
You can specify data types that you want Polyspace to consider as Boolean during rule checking. The software applies this redefinition only to data types defined by typedef statements.


Note The use of this option may affect the checking of MISRA C rules 12.6, 13.2, and 15.4.

To redefine a data type as Boolean:

- 1 Click the browse button  to the right of the **Effective boolean types** option.

The Effective boolean types dialog box opens.



- 2 In the text field, enter the data type.
- 3 Click . The data type appears in the list below the text field.
- 4 Click **OK** to close the dialog box.

Configuring Text and XML Editors

Before you check coding rules, you should configure your text and XML editors in the Preferences. Configuring text and XML editors allows you to

view source files and coding rules reports directly from the Coding Rules perspective.

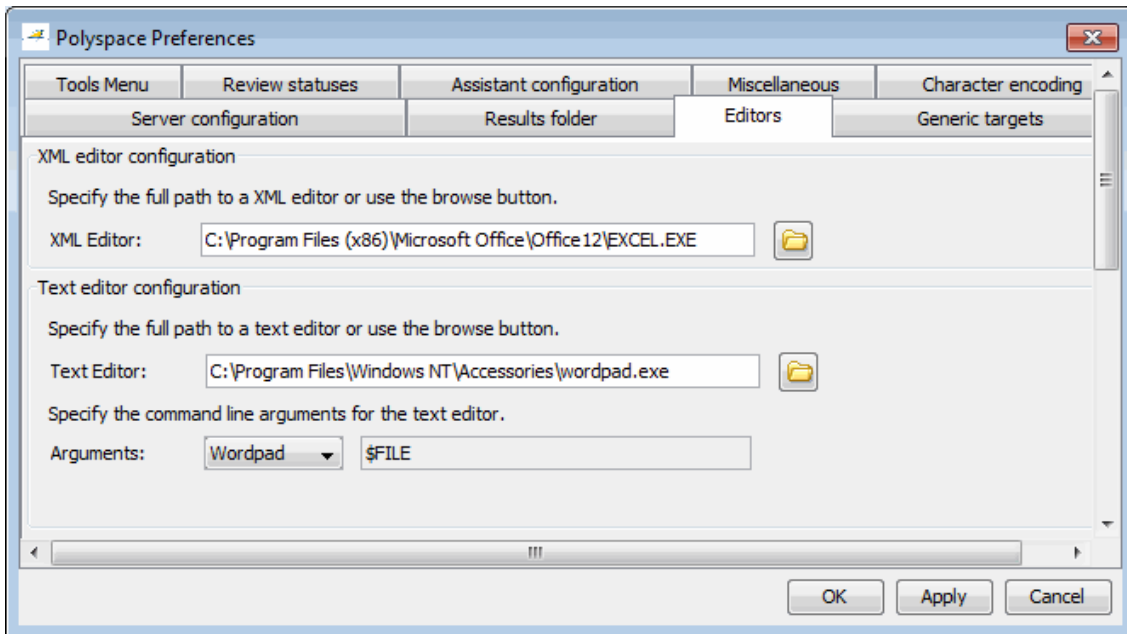
To configure your text and .XML editors:

1 Select **Options > Preferences**.

The Preferences dialog box opens.

2 Select the **Editors** tab.

The Editors tab opens.



3 Specify an XML editor to use to view MISRA-C reports. For example:

C:\Program Files\MSOffice\Office12\EXCEL.EXE

4 Specify a Text editor to use to view source files from the Project Manager logs. For example:

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

5 Select your text editor in the Arguments drop-down menu to automatically specify the command line arguments for that editor.

- Emacs
- Notepad++
- UltraEdit
- VisualStudio
- Wordpad

If you are using another text editor, select **Custom** from the drop-down menu, and specify the command line arguments for the text editor.

6 Click **OK**.

Commenting Code to Indicate Known Rule Violations

You can place comments in your code that inform Polyspace software of known or acceptable coding rule violations. The software uses the comments to highlight, in the Coding Rules perspective, errors or warnings related to the coding rule violations. Using this functionality, you can:

- Hide known coding rule violations while analyzing new coding rule violations.
- Inform other users of known coding rule violations.

The Coding Rules perspective displays the information that you provide within your code comments, and marks the violations as **Justified**.

For more information, see “Annotating Code to Indicate Known Coding Rule Violations” on page 5-47.

Note Source code annotations do not apply to code comments. Therefore, the following coding rules cannot be annotated:

- MISRA-C Rules 2.2 and 2.3
 - MISRA-C++ Rule 2-7-1
 - JSF++ Rules 127 and 133
-

Viewing Coding Rules Checker Results

In this section...

“Running a Verification with Coding Rules Checking” on page 11-18

“Examining Rule Violations” on page 11-19

“Commenting and Justifying Coding Rule Violations ” on page 11-22

“Opening Source Files from Coding Rules Perspective” on page 11-24

“Opening Coding Rules Report” on page 11-25

“Generating Coding Rules Report” on page 11-26

“Copying and Pasting Justifications” on page 11-27


Running a Verification with Coding Rules Checking

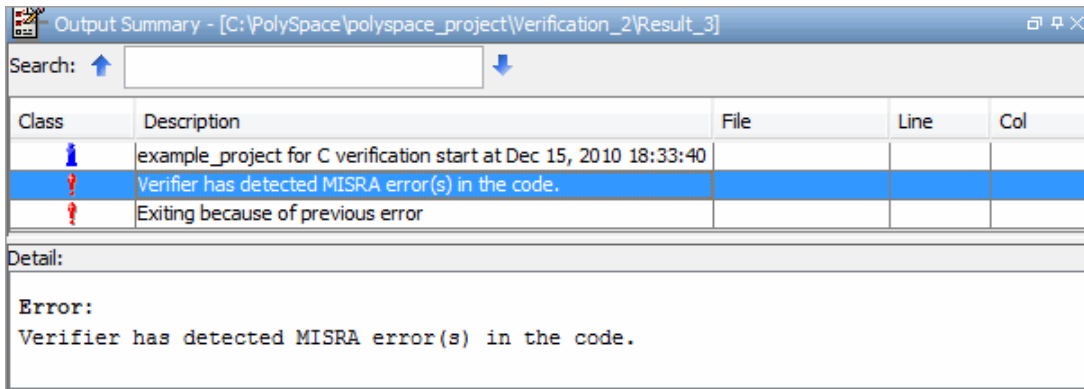
When you run a verification with the **MISRA C**, **Check MISRA C++ rules** or **Check JSF C++ rules** option selected, the verification checks most of the rules during the compile phase. If there is a violation of a rule with state **Error**, the verification stops.

Note Some rules address run-time errors.

The verification stops if there is a violation of a rule with state **Error**.

To start the verification:

- 1 Click the **Run** button  on the Project Manager toolbar.
- 2 Code analysis begins.
 - If the coding rules checker detects violations of coding rules set to **error**, the message “**Verification Failed**” appears at the bottom of the Project Manager perspective, and the Output Summary indicates that the verification has detected coding rules violations.



- If there are no violations of rules set to error, code verification continues after rules checking is complete.

3 When rules checking is complete, the file **MISRA-C-report.xml**, **MISRA-CPP-report.xml**, or **JSF-report.xml** appears in the Project Browser Results folder. This file contains the results from the coding rules checker.

Note If any source files in the verification do not compile, coding rules checking will be incomplete. The coding rules checker results will not contain results for files that did not compile, and may not contain full results for the files that did compile, since some rules are checked only after compilation completes.

Examining Rule Violations

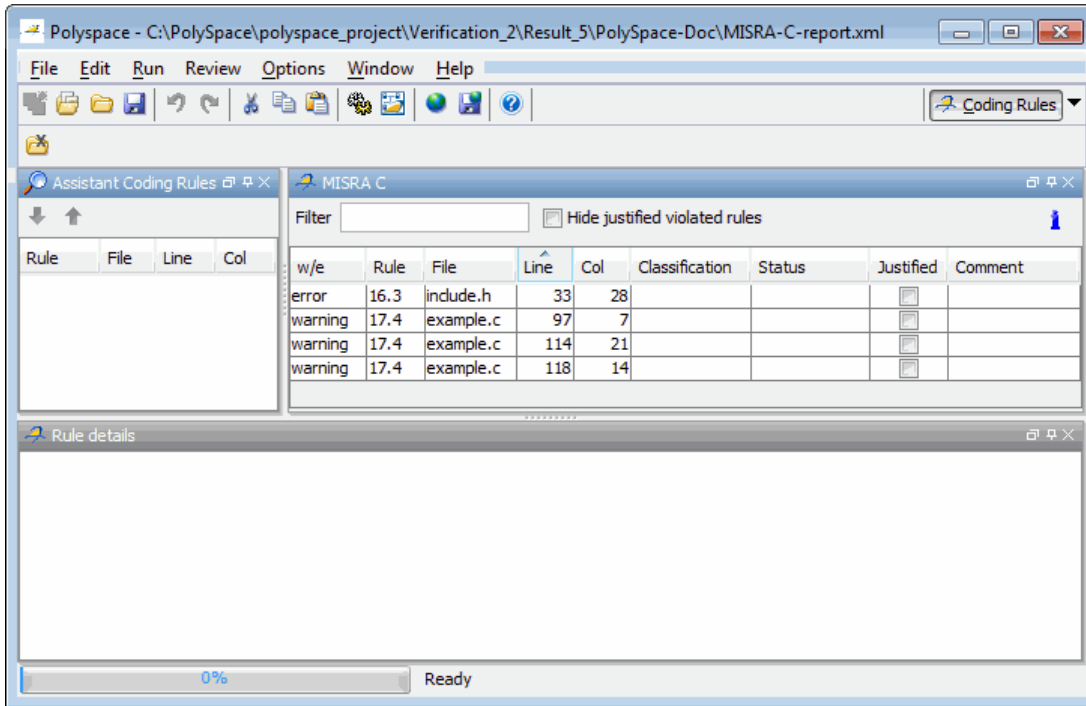
When code analysis is complete, you can view results in the Coding Rules perspective.

To examine MISRA C violations:

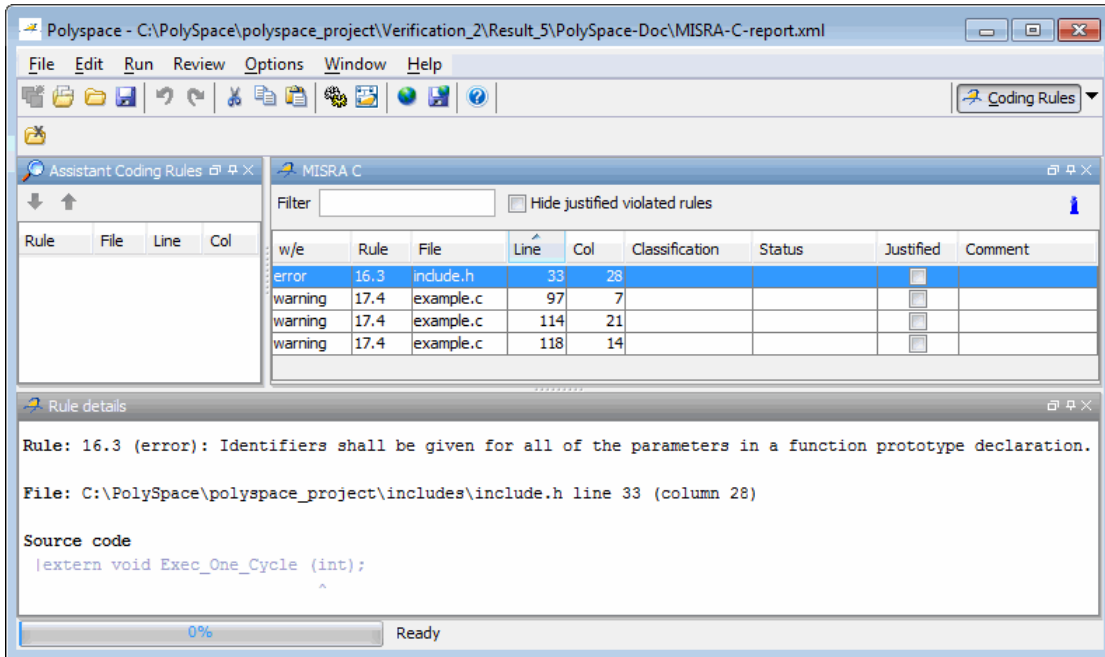
1 Double-click **MISRA-C-report.xml**, **MISRA-CPP-report.xml** or **JSF-report.xml** in the Project Browser Result folder.

The Coding Rules perspective appears, displaying a list of coding rule violations.

11 Checking Coding Rules

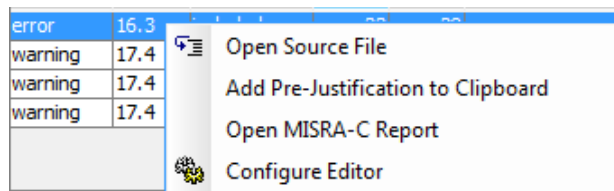


- 2 Click any of the violations to see a description of the violated rule, the full path of the file in which the violation was found, and the source code containing the violation.



In this example, the log reports a violation of rule 16.3. A function prototype declaration in `include.h` is missing an identifier.

- To open the source file containing the coding rule violation, right-click the row containing the violation, then select **Open Source File**.



The appropriate file opens in your text editor.

Note Before you can open source files, you must configure a text editor. See “Configuring Text and XML Editors” on page 11-14.

- 4 Correct any violations reported in the log, and run the verification again.

Commenting and Justifying Coding Rule Violations

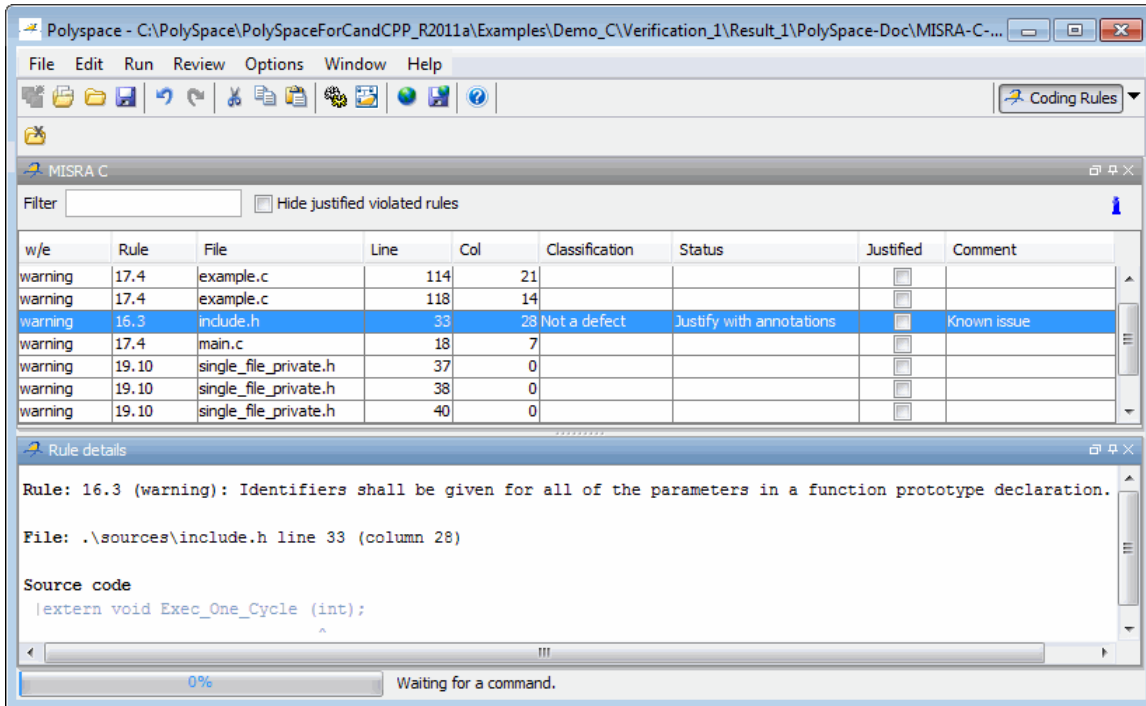
When reviewing coding rules violations in the Coding Rules perspective, you can classify the seriousness of each violation, mark violations as **Justified**, and enter comments to describe the results of your review.

After you mark violations as Justified, you can hide them. This helps you track the progress of your review and avoid reviewing the same violation twice.

To review, comment, and justify a violation:

- 1 In the Coding Rules perspective, select the violation you want to review.

The rule details pane displays a description of the violated rule, the full path of the file in which the violation was found, and the source code containing the violation.



2 After you review the violation, select a **Classification** to describe the seriousness of the issue:

- 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

Note You can also define your own statuses. See “Defining Custom Status” on page 8-58.

4 In the comment box, enter additional information about the violation.

5 Select the **Justified** check box to indicate that you have justified this check.

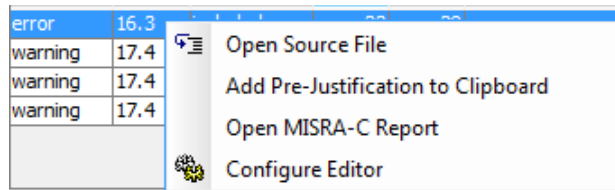
Note To hide coding rule violations that you justify, select the **Hide justified violated rules** check box.

Opening Source Files from Coding Rules Perspective

You can use the Coding Rules perspective to open the source file containing a coding rule violation.

To open the source file containing a coding rules violation:

- 1** In the Coding Rules perspective, select the violation you want to review.
- 2** Right-click the row containing the violation, then select **Open Source File**.



The appropriate file opens in your text editor.

Note Before you can open source files, you must configure a text editor. See “Configuring Text and XML Editors” on page 11-14.

Opening Coding Rules Report

After you check coding rules, you can generate an XML report containing all the errors and warnings reported by the coding rules checker.

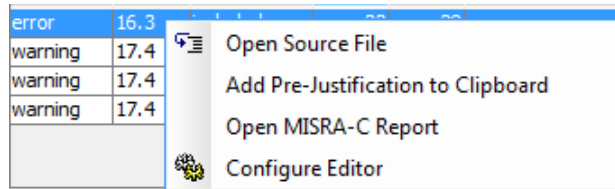
Note You must configure an XML editor before you can open a MISRA-C report. See “Configuring Text and XML Editors” on page 11-14.

To view the coding rules report:

- 1 Click the **Coding Rules** button in the Polyspace Verification Environment toolbar.

A list of MISRA C violations appear in the log part of the window.

- 2 Right click any row in the log, and select **Open MISRA-C Report**, **Open MISRA-CPP Report**, or **Open JSF Report**.



The report opens in your XML editor.

Name	Mode	Report	file	Line	Column	Message
16.3	required	error	C:\PolySpace\polyspace_project\includes\include.h	33	0	Identifiers shall be given for all of the parameters in a function prot
17.4	required	warning	example.c	97	0	Array indexing shall be the only allowed form of pointer arithmetic.
17.4	required	warning	example.c	114	0	Array indexing shall be the only allowed form of pointer arithmetic.
17.4	required	warning	example.c	118	0	Array indexing shall be the only allowed form of pointer arithmetic.

Note If any source files in the verification do not compile, the verification fails with compilation errors, and coding rules checking is incomplete. If this happens, the coding rules report is not exhaustive, since it does not contain results for files that did not compile, and may not contain full results (not all rules are checked) for the files that did compile.

Generating Coding Rules Report

You can use the Polyspace Report Generator to generate reports about compliance with Coding Rules, as well as other reports.

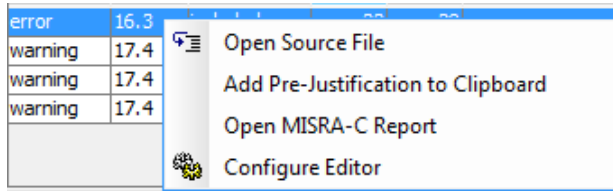
For information on using the Polyspace Report Generator, see “Generating Reports of Verification Results” on page 8-68.

Copying and Pasting Justifications

Instead of typing the full syntax of an annotation comment in your source code, you can copy an annotation template from the Coding Rules perspective, paste it into your source code, and modify the template to comment the check.

To copy the justification template to the clipboard:

- 1 In the Coding Rules perspective, select any violation.
- 2 Right-click the violation, then select **Add Pre-Justification to Clipboard**.



The justification string is copied to the clipboard.

- 3 Open the source file containing the violation you want to justify.
- 4 Navigate to the code you want to comment, and paste the justification template string on the line immediately before the line you want to comment.
- 5 Modify the template text to comment the code appropriately.

```
int    random_int ^ (void);
float  random_float(void);
extern void partial_init(int *new_alt);
extern void RTE(void);
/* polyspace<MISRA-C:16.3: Low : Justify with annotations > Known issue */
extern void Exec_One_Cycle (int);
extern int orderregulate (void);
extern void Begin_CS (void);
```

- 6 Save the file.

Coding Rules Assistant

In this section...
“Polyspace Metrics and Coding Rules Assistant” on page 11-28
“Reviewing Assistant Coding Rules” on page 11-28

Polyspace Metrics and Coding Rules Assistant

If you use the Polyspace Metrics Web interface to track coding rule violations, you can use the Coding Rules Assistant to review only the rule violations appropriate for your current Software Quality Objective level (SQO-level).

The Coding Rules Assistant displays only the rule violations you need to review to meet the requirements of your current SQO level. For example, if your quality level is set to SQO-1 in the Web interface, the Assistant Coding Rules pane displays only violations of the rules specified by “SQO Subset 1 – Coding Rules with a Direct Impact on Selectivity” on page 11-33.

Note The Assistant Coding Rules pane displays rule violations only if you open results from the Polyspace Metrics Web interface.

For more information on the Polyspace Metrics Web interface, see Chapter 12, “Software Quality with Polyspace Metrics”.

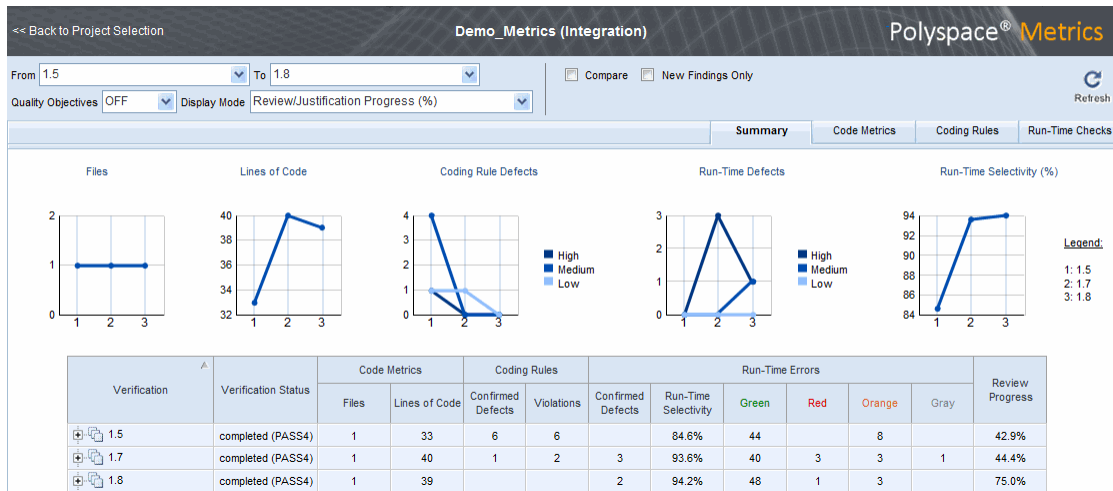
Reviewing Assistant Coding Rules

If you use the Polyspace Metrics Web interface to track coding rule violations, you can use the Coding Rules Assistant to review only the rule violations appropriate for your current Software Quality Objective level (SQO-level).

For more information on the Polyspace Metrics Web interface, see Chapter 12, “Software Quality with Polyspace Metrics”.

To use the Coding Rules Assistant:

- 1 Open your project in the Polyspace Metrics Web interface.



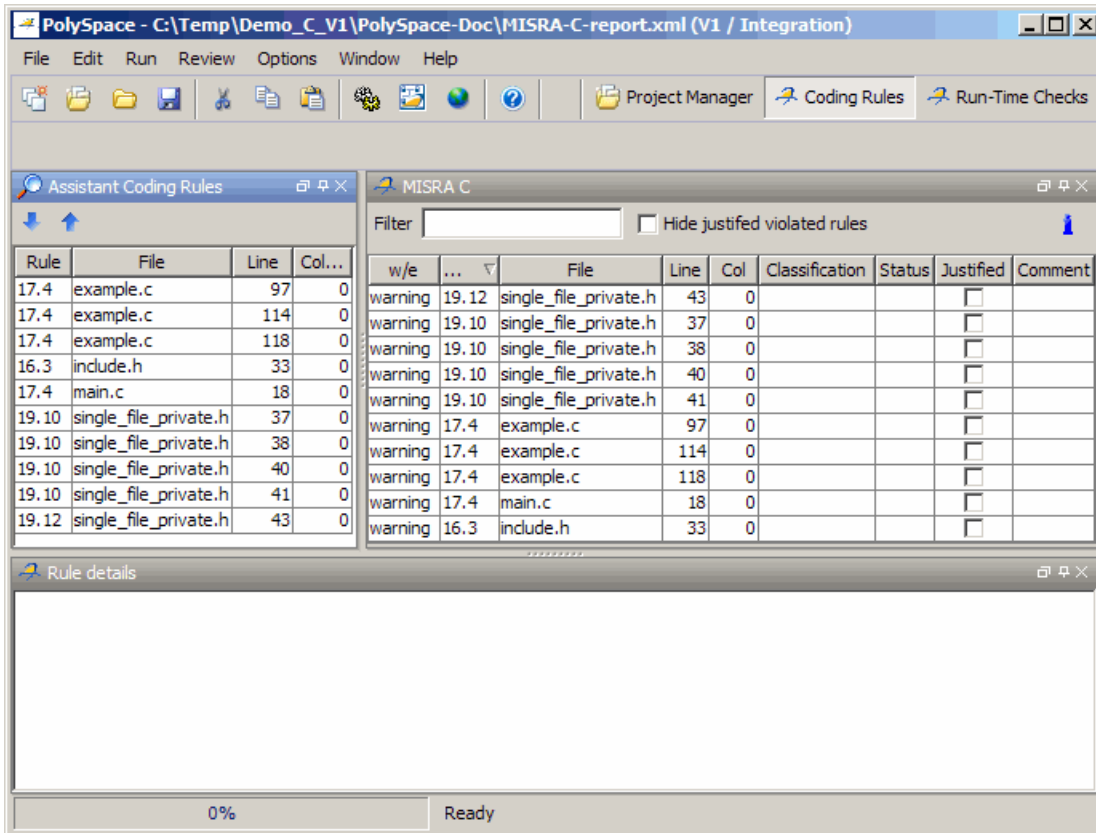
2 Select the Coding Rules tab.

A list of files and corresponding rule violations appears.

Verification	Coding Rules		
	Confirmed Defects	Reviewed	Violations
V4		0.0%	4
example.c		0.0%	3
include.h		100.0%	
main.c		0.0%	1
single_file_private		100.0%	

3 Click the value in the Violations column.

The Polyspace verification environment opens, showing the rule violations in the Coding Rules perspective.



4 In the Assistant Coding Rules toolbar, click the **Show Next Match** icon



- The Assistant Coding Rules pane shows the current rule violation.
- The MISRA C pane displays the current rule violation.
- The rule details pane displays a description of the violated rule, the full path of the file in which the violation was found, and the source code containing the violation.

The screenshot shows the PolySpace Coding Rules Assistant interface. The main window displays a list of violations in the MISRA C panel. The 'Rule details' panel shows the details for rule 17.4, which is a warning about array indexing. The source code snippet shows a pointer arithmetic operation: `| p++;`.

Rule	File	Line	Col...	w/e	...	File	Line	Col	Classification	Status	Justified	Comment
17.4	example.c	97	0	warning	19.12	single_file_private.h	43	0			<input type="checkbox"/>	
17.4	example.c	114	0	warning	19.10	single_file_private.h	37	0			<input type="checkbox"/>	
17.4	example.c	118	0	warning	19.10	single_file_private.h	38	0			<input type="checkbox"/>	
16.3	include.h	33	0	warning	19.10	single_file_private.h	40	0			<input type="checkbox"/>	
17.4	main.c	18	0	warning	19.10	single_file_private.h	41	0			<input type="checkbox"/>	
19.10	single_file_private.h	37	0	warning	17.4	example.c	97	0	Low	Improve	<input checked="" type="checkbox"/>	NXT
19.10	single_file_private.h	38	0	warning	17.4	example.c	114	0			<input type="checkbox"/>	
19.10	single_file_private.h	40	0	warning	17.4	example.c	118	0			<input type="checkbox"/>	
19.10	single_file_private.h	41	0	warning	17.4	main.c	18	0			<input type="checkbox"/>	
19.12	single_file_private.h	43	0	warning	16.3	include.h	33	0			<input type="checkbox"/>	

Rule details

Rule: 17.4 (warning): Array indexing shall be the only allowed form of pointer arithmetic.


File: example.c line 97 (column 0)

Source code

```
| p++;
```

0% Ready

- 5 Review the current check.
- 6 If you want to classify the violation as a defect, from the **Classification** cell drop-down list, select High, Medium, or Low . This will increment the **Confirmed Defect** value in Polyspace Metrics.
- 7 In the **Status** drop-down list, assign a status to this violation. For example, Fix or No action planned. When you assign a status to a violation, the software considers the violation to be *reviewed*.
- 8 If you consider the presence of a violation justifiable, select the **Justified** check box. In the **Comments** column, you can enter remarks justifying this violation.

- 9 Continue to click the the **Show Next Match** icon  until you have gone through all of the rule violations.
- 10 Save the project. The software updates review and justification information in the Polyspace Metrics repository. When you return to Polyspace Metrics, click **Refresh** to view the updated information.

Note Classifying a coding rule violation as a defect or assigning a status for an unreviewed violation in the Polyspace window, increases the corresponding metric values (**Confirmed Defects** and **Review Progress**) in the **Summary** and **Coding Rules** views of Polyspace Metrics.

Software Quality Objective Subsets of Coding Rules (C)

In this section...

“SQO Subset 1 – Coding Rules with a Direct Impact on Selectivity” on page 11-33

“SQO Subset 2 – Coding Rules with an Indirect Impact on Selectivity” on page 11-35

SQO Subset 1 – Coding Rules with a Direct Impact on Selectivity

The following set of coding rules will typically improve the selectivity of your verification results.

Rule I	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

Rule I	Description
MISRA 13.4	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type
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.

SQO Subset 2 – Coding Rules with an Indirect Impact on Selectivity

Good design practices generally lead to less code complexity, which can improve the selectivity of your verification results. The following set of coding rules help address design issues that can impact selectivity.

Note Specifying SQO-subset2 in your MISRA C rules configuration checks both the rules listed in SQO Subset 1 and SQO Subset 2.

Rule #	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 #	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 #	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.

Supported Coding Rules

In this section...
“MISRA C Rules Supported” on page 11-38
“MISRA C Rules Not Checked” on page 11-75
“Supported MISRA C++ Coding Rules” on page 11-79
“MISRA C++ Rules Not Checked” on page 11-99
“Supported JSF C++ Coding Rules ” on page 11-105
“JSF++ Rules Not Checked” on page 11-130

MISRA C Rules Supported

The following tables list MISRA C 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.

The Polyspace coding rules checker also 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, and 13.5.

The software reports most violations during the compile phase of a verification. However, the software detects violations of rules 9.1 (NIV checks), 12.11 (OVFL check using `-scalar-overflows-checks signed-and-unsigned`), 13.7 (gray checks), 14.1 (gray checks), 16.2 (Call graph) and 21.1 during code verification, 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 verification.

Environment

N.	MISRA Definition	Messages in log 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 ‘#include_next’ • ANSI C does not allow macros with variable arguments list • ANSI C does not allow ‘#assert’ • ANSI C does not allow ‘#unassert’ • ANSI C does not allow testing assertions • ANSI C does not allow ‘#ident’ • ANSI C does not allow ‘#scs’ • text following ‘#else’ violates ANSI standard. • text following ‘#endif’ violates ANSI standard. 	All the supported extensions lead to a violation of this MISRA rule. Standard compilation error messages do not lead to a violation of this MISRA rule and remain unchanged. Can be turned to Off (see -misra2 option).

N.	MISRA Definition	Messages in log 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 log 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 log file	Detailed Polyspace Specification
			asm statements even if it is encapsulated by a MACRO).
2.2	source code shall only use /* */ style comments	C++ comments shall not be used.	C++ comments are handled as comments but lead to a violation of this MISRA rule. Note: This rule cannot be annotated in the source code.
2.3	The character sequence /* shall not be used within a comment	The character sequence /* shall not appear within a comment.	This rule violation is also raised when the character sequence /* inside a C++ comment. Note: This rule cannot be annotated in the source code.

Character Sets

N.	MISRA Definition	Messages in log 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 log 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
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

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
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)	Warning on other conflicts (including member names)

Types

N.	MISRA Definition	Messages in log 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. 	Warning if value of type plain char is implicitly converted to value of type signed char or unsigned char.

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
		<ul style="list-style-type: none"> • Value of type signed char is implicitly converted to plain char. • Value of type unsigned char is implicitly converted to plain 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 log 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 log 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
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. 	During link phase, errors are converted into warnings with <code>-permissive-link</code> option.

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
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).
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 	Tentative of definitions are considered as definitions, No warning on undefined objects with <code>-allow-undef-variables</code> option, No warning on predefined symbols.

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
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.	Not checked if <code>-main-generator</code> option is set. 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 log file	Detailed Polyspace Specification
9.1	All automatic variables shall have been assigned a value before being used.		Checked during code verification. Violations displayed as NIV checks in the verification 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 <code>=</code> construct shall not be used to explicitly initialize	In an enumerator list, the <code>=</code> construct shall not be used to explicitly initialize	

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
	members other than the first, unless all items are explicitly initialized.	members other than the first, unless all items are explicitly initialized.	

Arithmetic Type Conversion

N.	MISRA Definition	Messages in log 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 • the expression is not constant and is a function argument, or • the expression is not constant and is a return expression 	<ul style="list-style-type: none"> • Implicit conversion of the expression of underlying type ?? to the type ?? that is not a wider integer type of the same signedness. • Implicit conversion of one of the binary operands whose underlying types are ?? and ?? • Implicit conversion of the binary right hand operand of underlying type ?? to ?? that is not an integer type. • Implicit conversion of the binary left hand operand of underlying type ?? to ?? that is not an integer type. • Implicit conversion of the binary right hand operand of underlying type ?? to ?? that is not a wider integer type of 	<ol style="list-style-type: none"> 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. 2 An expression of bool or enum types has int as underlying type. 3 Plain char may have signed or unsigned underlying type (depending on Polyspace target configuration or option setting). 4 The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
		the same signedness or Implicit conversion of the binary ? left hand operand of underlying type ?? to ??, but it is a complex expression.	bitfield width is not taken into account and it assumes that only signed unsigned int are used for bitfield (Rule 6.4).

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
10.1 (cont.)		<ul style="list-style-type: none"> • Implicit conversion of complex integer expression of underlying type ?? to ??. • Implicit conversion of non-constant integer expression of underlying type ?? in function return whose expected type is ??. • Implicit conversion of non-constant integer expression of underlying type ?? as argument of function whose corresponding parameter type is ??. 	
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 ?? to ?? that is not a wider floating type. • Implicit conversion of the binary ? right hand operand from ?? to ??, but it is a complex expression. • Implicit conversion of the binary ? right hand operand from ?? to ?? that is not a wider floating type or Implicit conversion of the binary ? left hand operand from ?? 	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.

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
		<p>to ??, but it is a complex expression.</p> <ul style="list-style-type: none"> • Implicit conversion of complex floating expression from ?? to ??. • Implicit conversion of floating expression of ?? type in function return whose expected type is ??. • Implicit conversion of floating expression of ?? type as argument of function whose corresponding parameter type is ??. 	
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 ?? may only be cast to narrower integer type of same signedness, however the destination type is ??.</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 log 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 taken into account and it assumes that only signed, unsigned int are used for bitfield (Rule 6.4).
10.4	The value of a complex expression of float type may only be cast to narrower floating type	Complex expression of ?? type may only be cast to narrower floating type, however the destination type is ??.	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 log 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 log 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 log 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 log 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).
12.3	The <code>sizeof</code> operator should not be used on expressions that contain side effects.	The <code>sizeof</code> operator should not be used on expressions that contain side effects.	No warning on volatile accesses

N.	MISRA Definition	Messages in log 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</p>

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
			<p>operator should be effectively Boolean.</p> <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 ??. • Bitwise [<< >>] on left hand operand of signed underlying type ??. • Bitwise [& ^] on two operands of s 	<p>The underlying type for an integer used in a re-processor expression 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 log 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 ?? 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 ??. • Minus operator applied to an expression whose underlying type is unsigned 	<p>The underlying type for an integer used in a re-processor expression 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.	

N.	MISRA Definition	Messages in log 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 on casts with float pointers (excepted with void *).
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 log 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, unless the operand is effectively Boolean	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	No warning is given on integer constants. Example: if (2) The use of the option -boolean-types may increase or decrease the number of warnings generated.
13.3	Floating-point expressions shall not be tested for equality or inequality.	Floating-point expressions shall not be tested for equality or inequality.	Warning on directs tests only.

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
13.4	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type	If <i>for</i> index is a variable symbol, checked that it is not a float.
13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control	<ul style="list-style-type: none"> • 1st expression should be an assignment. • Bad type for loop counter (XX). • 2nd expression should be a comparison. • 2nd expression should be a comparison with loop counter (XX). • 3rd expression should be an assignment of loop counter (XX). • 3rd expression: assigned variable should be the loop counter (XX). • The following kinds of <i>for</i> loops are allowed: <ul style="list-style-type: none"> (a) all three expressions shall be present; (b) the 2nd and 3rd expressions shall be present with prior initialization of the loop counter; 	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 log file	Detailed Polyspace Specification
		(c) all three expressions shall be empty for a deliberate infinite loop.	
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 log 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	<ul style="list-style-type: none"> • All non-null statements shall either: 	

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
	executed, or cause control flow to change	<ul style="list-style-type: none"> • 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	
14.7	A function shall have a single point of exit at the end of the function	A function shall have a single point of exit at the end of the function	

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
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 log 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 4 case 1: ... </pre> <p>The code between switch statement and first case is checked as gray by Polyspace verification. 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 log 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 log 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 Run-Time Checks perspective gives the information). Polyspace verification 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 log 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 const if the pointer is not used to modify the addressed object.	Pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.	Detected with simple heuristic algorithm.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Missing return value for non-void function XX.	Warning when a non-void function is not terminated with an unconditional return with an expression.
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.	Function identifier XX should be preceded by a & or followed by a parameter list.	

Pointers and Arrays

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
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 log 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.
18.4	Unions shall not be used	Unions shall not be used.	

Preprocessing Directives

N.	MISRA Definition	Messages in log 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" 	
19.3	The #include 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> 	

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.	Macro '<name>' does not expand to a compliant construct.	<p>We assume that a macro definition does not violate this rule when it expands to:</p> <ul style="list-style-type: none"> • a braced construct (not necessarily an initializer) • a parenthesized construct (not necessarily an expression) • a number • a character constant • a string constant (can be the result of the concatenation of string field arguments and literal strings) • the following keywords: typedef, extern, static, auto, register, const, volatile, __asm__ and __inline__ • a do-while-zero construct
19.5	Macros shall not be #defined and #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.	

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
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)
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.	Parameter instance shall be enclosed in parentheses.	If x is a macro parameter, the following instances of x as an operand of the # and ## operators do not generate a warning: #x, ##x, and x##. Otherwise, parentheses are needed around x.

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	'<name>' is not defined.	
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.	More than one occurrence of the # or ## preprocessor operators.	
19.13	The # and ## preprocessor operators should not be used	Message on definitions of macros using # or ## operators	
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.	'defined' without an identifier.	
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.	Precautions shall be taken in order to prevent multiple inclusions.	<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</p>

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
			inclusions. Otherwise, a violation of this MISRA rule is detected.
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	directive is not syntactically meaningful.	
19.17	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> or <code>#ifdef</code> directive to which they are related.	<ul style="list-style-type: none"> • <code>#elif</code> not within a conditional. • <code>#else</code> not within a conditional. • <code>#elif</code> not within a conditional. • <code>#endif</code> not within a conditional. • unbalanced <code>#endif</code>. • unterminated <code>#if</code> conditional. • unterminated <code>#ifdef</code> conditional. • unterminated <code>#ifndef</code> conditional. 	

Standard Libraries

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.	<ul style="list-style-type: none"> • The macro '<name>' shall not be redefined. • The macro '<name>' 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	Warning for argument in library function call if the following are all true: <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.

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
			The library function can be one of the following : sqrt, tan, pow, log, log10, fmod, acos, asin, acosh, atanh, or atan2.
20.4	Dynamic heap memory allocation shall not be used.	<ul style="list-style-type: none"> • The macro '<name>' shall not be used. • Identifier XX should not be used. 	In case the dynamic heap memory allocation functions are actually macros and the macro is expanded in the code, this rule is detected as violated. Assumes rule 20.2 is not violated.
20.5	The error indicator errno shall not be used	The error indicator errno shall not be used	Assumes that rule 20.2 is not violated
20.6	The macro <i>offsetof</i> , in library <stddef.h>, shall not be used.	<ul style="list-style-type: none"> • The macro '<name>' shall not be used. • Identifier XX should not be used. 	Assumes that rule 20.2 is not violated
20.7	The <i>setjmp</i> macro and the <i>longjmp</i> function shall not be used.	<ul style="list-style-type: none"> • The macro '<name>' shall not be used. • Identifier XX should not be used. 	In case the longjmp function is actually a macro and the macro is expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.8	The signal handling facilities of <signal.h> shall not be used.	<ul style="list-style-type: none"> • The macro '<name>' shall not be used. • Identifier XX should not be used. 	In case some of the signal functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated

N.	MISRA Definition	Messages in log file	Detailed Polyspace Specification
20.9	The input/output library <stdio.h> shall not be used in production code.	<ul style="list-style-type: none"> • The macro ‘<name> shall not be used. • Identifier XX should not be used. 	In case the input/output library functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.10	The library functions atof, atoi and 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 log 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 verification (runtime error checks).

MISRA C Rules Not Checked

The Polyspace coding rules checker does not check the following MISRA C coding rules. These rules cannot be enforced because they are outside the scope of Polyspace verification. 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	It is a process rule method.

Rule	Description	Comments
	language/compiler/assemblers conform.	
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.
3.4 (Required)	All uses of the <i>#pragma</i> directive shall be documented and explained.	The documentation of compiler must be checked.
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.

Expressions

Rule	Description	Comments
12.11 (Required)	Evaluation of constant unsigned expression should not lead to wraparound.	This rule is partially implemented using the option <code>-scalar-overflows-checks signed-and-unsigned</code> in Polyspace software. Concerning possible preprocessing overflows, Polyspace preprocessor does not take into account target basic types and considers always 32-Bit long int.

Functions

Rule	Description	Comments
16.10 (Required)	If a function returns error information, then that error information shall be tested.	Not statically checkable unless type defining error is standardized.

Pointers and Arrays

Rule	Description	Comments
17.1 (Required)	Pointer arithmetic shall only be applied to pointers that address an array or array element.	Not statically checkable unless the pointer verification has been done

Structures and Unions

Rule	Description	Comments
18.2 (Required)	An object shall not be assigned to an overlapping object.	Not statically checkable unless the data dynamic properties is taken into account
18.3 (Required)	An area of memory shall not be reused for unrelated purposes.	"purpose" is functional design issue.

Supported MISRA C++ Coding Rules

- “Language Independent Issues” on page 11-80
- “General” on page 11-80
- “Lexical Conventions” on page 11-80
- “Basic Concepts” on page 11-82
- “Standard Conversions” on page 11-83
- “Expressions” on page 11-84
- “Statements” on page 11-87
- “Declarations” on page 11-89
- “Declarators” on page 11-91
- “Classes” on page 11-92
- “Derived Classes” on page 11-92
- “Member Access Control” on page 11-93
- “Special Member Functions” on page 11-93
- “Templates” on page 11-94
- “Exception Handling” on page 11-95
- “Preprocessing Directives” on page 11-96
- “Library Introduction” on page 11-98

- “Language Support Library” on page 11-98
- “Diagnostic Library” on page 11-99
- “Input/output Library” on page 11-99

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 Verifier.

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.	

N.	MISRA Definition	Comments
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.	
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 ptrdiff_t is signed integer
5-0-4	An implicit integral conversion shall not change the signedness of the underlying type.	Assumes that ptrdiff_t is signed integer If the conversion is to a narrower integer with a different sign then Misra Cpp 5-0-4 takes precedence over Misra Cpp 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 Cpp 5-0-4 takes precedence over Misra Cpp 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.	
5-0-10	If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	

N.	MISRA Definition	Comments
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. (p+I, I+p and p-I, where p is a pointer and I an integer, p[i] accepted).
5-0-18	>, >=, <, <= 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 && or shall be a postfix - expression.	During preprocessing, violations of this rule are detected on the expressions in #if directives. Allowed exception on associativity (a && b && c), (a b c).
5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.	
5-2-3	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.	

N.	MISRA Definition	Comments
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.	
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.	

N.	MISRA Definition	Comments
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.	
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.	

N.	MISRA Definition	Comments
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.	
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).

N.	MISRA Definition	Comments
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.	

N.	MISRA Definition	Comments
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.	
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 member-declarator respectively.	
8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.	
8-4-1	Functions shall not be defined using the ellipsis notation.	
8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.	
8-4-3	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	
8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.	
8-5-1	All variables shall have a defined value before they are used.	NIV given by verifier and error messages for obvious cases
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
10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.	
10-2-1	All accessible entity names within a multiple inheritance hierarchy should be unique.	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.

N.	MISRA Definition	Comments
10-3-2	Each overriding virtual function shall be declared with the virtual keyword. 1.	
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtua	

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.	
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.	
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. No detection if no "main" (desktop mode?).
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.	

N.	MISRA Definition	Comments
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.	
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 ##.	

N.	MISRA Definition	Comments
16-0-7	Undefined macro identifiers shall not be used in <code>#if</code> or <code>#elif</code> preprocessor directives, except as operands to the defined operator.	
16-0-8	If the <code>#</code> 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 <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.	
16-2-1	The pre-processor shall only be used for file inclusion and include guards.	The rule is raised for <code>#ifdef</code> / <code>#define</code> 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 <code>'</code> , <code>"</code> , <code>/*</code> or <code>//</code> characters shall not occur in a header file name.	
16-2-5	The <code>\</code> character should not occur in a header file name.	
16-2-6	The <code>#include</code> directive shall be followed by either a <code><filename></code> or <code>"filename"</code> sequence.	
16-3-1	There shall be at most one occurrence of the <code>#</code> or <code>##</code> operators in a single macro definition.	
16-3-2	The <code>#</code> and <code>##</code> 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 <string> shall not be used.	
18-2-1	The macro offsetof shall not be used.	
18-4-1	Dynamic heap memory allocation shall not be used.	
18-7-1	The signal handling facilities of <signal> 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 <stdio> shall not be used.	

MISRA C++ Rules Not Checked

- “Language Independent Issues” on page 11-100
- “General” on page 11-101
- “Lexical Conventions” on page 11-101
- “Standard Conversions” on page 11-102
- “Expressions” on page 11-102
- “Declarations” on page 11-103
- “Classes” on page 11-103
- “Templates” on page 11-104
- “Exception Handling” on page 11-104
- “Preprocessing Directives” on page 11-105
- “Library Introduction” on page 11-105

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 effect(s).	
0-1-9	There shall be no dead code.	Not checked by the coding rules checker. Can be enforced through detection of gray code during verification.
0-1-11	There shall be no unused parameters (named or unnamed) in non- virtual functions.	
0-1-12	There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.	
0-2-1	An object shall not be assigned to an overlapping object.	
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
5-0-11	The plain char type shall only be used for the storage and use of character values.	
5-0-12	signed char and unsigned char type shall only be used for the storage and use of numeric values.	
5-0-13	The condition of an if-statement and the condition of an iteration- statement shall have type bool.	
5-0-16	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.	
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 that point.	
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++.	

Supported JSF C++ Coding Rules

- “Code Size and Complexity” on page 11-106
- “Environment” on page 11-106
- “Libraries” on page 11-107
- “Pre-Processing Directives” on page 11-108
- “Header Files” on page 11-109
- “Style” on page 11-110
- “Classes” on page 11-113
- “Namespaces” on page 11-117
- “Templates” on page 11-117
- “Functions” on page 11-118
- “Comments” on page 11-119
- “Declarations and Definitions” on page 11-119
- “Initialization” on page 11-120
- “Types” on page 11-121

- “Constants” on page 11-121
- “Variables” on page 11-121
- “Unions and Bit Fields” on page 11-122
- “Operators” on page 11-122
- “Pointers and References” on page 11-124
- “Type Conversions” on page 11-125
- “Flow Control Standards” on page 11-126
- “Expressions” on page 11-128
- “Memory Allocation” on page 11-129
- “Fault Handling” on page 11-129
- “Portable Code” on page 11-129

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 log file: <i><function name></i> has <i><num></i> logical source lines of code.
3	All functions shall have a cyclomatic complexity number of 20 or less.	Message in log file: <i><function name></i> has cyclomatic complexity number equal to <i><num></i>

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.	

N.	JSF++ Definition	Comments
11	Trigraphs will not be used.	
12	The following digraphs will not be used: <%, %>, <:, :>, %:, %:%:.	Message in log file: The following digraph will not be used: <i><digraph></i> Reports the digraph. If the rule level is set to warning, the digraph will be allowed even if it is not supported in <code>-dialect iso</code>
13	Multi-byte characters and wide string literals will not be used.	Report L 'c' and L "string" and use of <code>wchar_t</code> .
14	Literal suffixes shall use uppercase rather than lowercase letters.	
15	Provision shall be made for run-time checking (defensive programming).	Done with RTE checks in the Verifier.

Libraries

N.	JSF++ Definition	Comments
17	The error indicator <code>errno</code> shall not be used.	<code>errno</code> should not be used as a macro or a global with external "C" linkage.
18	The macro <code>offsetof</code> , in library <code><stddef.h></code> , shall not be used.	<code>offsetof</code> should not be used as a macro or a global with external "C" linkage.
19	<code><locale.h></code> and the <code>setlocale</code> function shall not be used.	<code>setlocale</code> and <code>localeconv</code> should not be used as a macro or a global with external "C" linkage.
20	The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.	<code>setjmp</code> and <code>longjmp</code> should not be used as a macro or a global with external "C" linkage.
21	The signal handling facilities of <code><signal.h></code> shall not be used.	<code>signal</code> and <code>raise</code> should not be used as a macro or a global with external "C" linkage.

N.	JSF++ Definition	Comments
22	The input/output library <code><stdio.h></code> shall not be used.	all standard functions of <code><stdio.h></code> should not be used as a macro or a global with external "C" linkage.
23	The library functions <code>atof</code> , <code>atoi</code> and <code>atol</code> from library <code><stdlib.h></code> shall not be used.	<code>atof</code> , <code>atoi</code> and <code>atol</code> should not be used as a macro or a global with external "C" linkage.
24	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> from library <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 pre-processor 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> pre-processor 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> .

N.	JSF++ Definition	Comments
29	The <code>#define</code> pre-processor 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 log file: <ul style="list-style-type: none"> • 29.1 : The <code>#define</code> pre-processor directive shall not be used to create inline macros. • 29.2 : Inline functions shall be used instead of inline macros
30	The <code>#define</code> pre-processor 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> pre-processor 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> pre-processor directive will only be used to include header (*.h) 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 (*.h) 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.	
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	Identifiers will not differ by: <ul style="list-style-type: none"> • Only a mixture of case • The presence/absence of the underscore character • The interchange of the letter 'O'; with the number '0' or the letter 'D' • The interchange of the letter 'I'; with the number '1' or the letter 'l' 	Checked regardless of scope. Not checked between macros and other identifiers. Messages in log file: <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.

N.	JSF++ Definition	Comments
	<ul style="list-style-type: none"> • 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' 	<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 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'.
50	The first word of the name of a class, structure, namespace, enumeration, or type created with typedef will begin with an uppercase letter. All others letters will be lowercase.	Messages in log 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 log 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 log 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 ".	

N.	JSF++ Definition	Comments
54	Implementation files will always have a file name extension of ".cpp".	Not case sensitive if you set the option <code>-dos</code> .
57	The public, protected, and private sections of a class will be declared in that order.	
58	When declaring and defining functions with more than two parameters, the leading parenthesis and the first argument will be written on the same line as the function name. Each additional argument will be written on a separate line (with the closing parenthesis directly after the last argument).	Detects that two parameters are not on the same line, The first parameter should be on the same line as function name. Does not check for the closing parenthesis.
59	The statements forming the body of an if, else if, else, while, do ... while or for statement shall always be enclosed in braces, even if the braces form an empty block.	<p>Messages in log 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.

N.	JSF++ Definition	Comments
60	Braces ("{}") which enclose a block will be placed in the same column, on separate lines directly before and after the block.	Detects that statement-block braces should be in the same columns.
61	Braces ("{}") which enclose a block will have nothing else on the line except comments.	
62	The dereference operator '*' and the address-of operator '&' will be directly connected with the type-specifier.	Reports when there is a space between type and "*" "&" for variables, parameters and fields declaration.
63	Spaces will not be used around '.' or '->', nor between unary operators and operands.	<p>Reports when the following characters are not directly connected to a white space:</p> <ul style="list-style-type: none"> • . • -> • ! • ~ • - • ++ • — <hr/> <p>Note A violation will be reported for "." used in float/double definition.</p> <hr/>

Classes

N.	JSF++ Definition	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.

N.	JSF++ Definition	Comments
71.1	A class's virtual functions shall not be invoked from its destructor or any of its constructors.	Reports when a constructor or destructor directly calls a virtual function.
74	Initialization of nonstatic class members will be performed through the member initialization list rather than through assignment in the body of a constructor.	<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 log 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 log 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.	

N.	JSF++ Definition	Comments
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/>
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 log file:</p>

N.	JSF++ Definition	Comments
		<ul style="list-style-type: none"> • An assignment operator shall return a reference to <code>*this</code>. • An assignment operator shall return a reference to its first arg.
83	An assignment operator shall assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).	Reports when a copy assignment does not assign all data members. In a derived class, it also reports when a copy assignment does not call inherited copy assignments.
88	Multiple inheritance shall only be allowed in the following restricted form: <i>n</i> interfaces plus <i>m</i> private implementations, plus at most one protected implementation.	Messages in log file: <ul style="list-style-type: none"> • Multiple inheritance on public implementation shall not be allowed: <code><public_base_class></code> is not an interface. • Multiple inheritance on protected implementation shall not be allowed : <code><protected_base_class_1></code> • <code><protected_base_class_2></code> are not interfaces.
88.1	A stateful virtual base shall be explicitly declared in each derived class that accesses it.	
89	A base class shall not be both virtual and non-virtual in the same hierarchy.	
94	An inherited nonvirtual function shall not be redefined in a derived class.	Does not report for destructor. Message in log file: Inherited nonvirtual function %s shall not be redefined in a derived class.

N.	JSF++ Definition	Comments
95	An inherited default parameter shall never be redefined.	
96	Arrays shall not be treated polymorphically.	Reports pointer arithmetic and array like access on expressions whose pointed type is used as a base class.
97	Arrays shall not be used in interface.	Only to prevent array-to-pointer-decay, Not checked on private methods
97.1	Neither operand of an equality operator (<code>==</code> or <code>!=</code>) shall be a pointer to a virtual member function.	Reports <code>==</code> and <code>!=</code> 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	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	Comments
104	A template specialization shall be declared before its use.	Reports the actual compilation error message.

Functions

N.	JSF++ Definition	Comments
107	Functions shall always be declared at file scope.	
108	Functions with variable numbers of arguments shall not be used.	
109	A function definition should not be placed in a class specification unless the function is intended to be inlined.	Reports when there is no "inline" 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 Verifier. Message in log 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	Comments
126	Only valid C++ style comments (<code>//</code>) 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	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	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	Comments
142	All variables shall be initialized before use.	Done with NIV and LOCAL_NIV checks in the Verifier.
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	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	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	Comments
152	Multiple variable declarations shall not be allowed on the same line.	

Unions and Bit Fields

N.	JSF++ Definition	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	Comments
157	The right hand operand of a && or operator shall not contain side effects.	<p>Assumes rule 159 is not violated.Messages in log 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 log 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	Comments
		Exception for: X Y Z , Z&&Y &&Z
159	Operators , &&, and unary & shall not be overloaded.	Messages in log 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 +. Verifier used 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 IDP checks in Verifier.
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. (No special case for visitor pattern.)
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 Verifier.
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 the SCAL-OVFL and FLOAT-OVFL checks in the Verifier.
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: <code>malloc / calloc / realloc / free</code> and all <code>new/delete</code> operators in functions or methods.

Fault Handling

N.	JSF++ Definition	Comments
208	C++ exceptions shall not be used.	Reports <code>try</code> , <code>catch</code> , <code>throw spec</code> , and <code>throw</code> .

Portable Code

N.	JSF++ Definition	Comments
209	The basic types of <code>int</code> , <code>short</code> , <code>long</code> , <code>float</code> and <code>double</code> shall not be used, but specific-length equivalents should be <code>typedef</code> 'd accordingly for each compiler, and these type names used in the code.	Only allows use of basic types through direct <code>typedefs</code> .
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: <code>p + I</code> <code>p - I</code> <code>p++</code> <code>p--</code> <code>p+=</code> <code>p-=</code> Allows <code>p[i]</code> .

JSF++ Rules Not Checked

- “Code Size and Complexity” on page 11-131
- “Rules” on page 11-131
- “Environment” on page 11-131
- “Libraries” on page 11-131
- “Header Files” on page 11-132
- “Style” on page 11-132
- “Classes” on page 11-132
- “Namespaces” on page 11-134
- “Templates” on page 11-134
- “Functions” on page 11-135
- “Comments” on page 11-136
- “Initialization” on page 11-136
- “Types” on page 11-136
- “Unions and Bit Fields” on page 11-137
- “Operators” on page 11-137
- “Type Conversions” on page 11-137
- “Expressions” on page 11-137
- “Memory Allocation” on page 11-138
- “Portable Code” on page 11-138
- “Efficiency Considerations” on page 11-138
- “Miscellaneous” on page 11-138
- “Testing” on page 11-139

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 itself 2 the right-hand side of an assignment 3 a condition 4 the only argument expression with a side-effect in a function call 5 condition of a loop 6 switch condition 7 single part of a chained operation

Memory Allocation

N.	JSF++ Definition
207	Unencapsulated global data will be avoided.

Portable Code

N.	JSF++ Definition
210	Algorithms shall not make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.).
210.1	Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.
211	Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.
212	Underflow or overflow functioning shall not be depended on in any special way.
214	Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.

Efficiency Considerations

N.	JSF++ Definition
216	Programmers should not attempt to prematurely optimize code.

Miscellaneous

N.	JSF++ Definition
217	Compile-time and link-time errors should be preferred over run-time errors.
218	Compiler warning levels will be set in compliance with project policies.

Testing

N.	JSF++ Definition
219	All tests applied to a base class interface shall be applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions /invariants shall be substituted in the derived class tests.
220	Structural coverage algorithms shall be applied against flattened classes.
221	Structural coverage of a class within an inheritance hierarchy containing virtual functions shall include testing every possible resolution for each set of identical polymorphic references.

Software Quality with Polyspace Metrics

- “About Polyspace Metrics” on page 12-2
- “Setting Up Verification to Generate Metrics” on page 12-4
- “Accessing Polyspace Metrics” on page 12-12
- “What You Can Do with Polyspace Metrics” on page 12-15
- “Customizing Software Quality Objectives” on page 12-30
- “Tips for Administering Results Repository” on page 12-45

About 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 “Accessing Polyspace Metrics” on page 12-12.
- Observe trends over time, by project or module. See “Review Overall Progress” on page 12-15.
- Compare metrics of one project version with those of another. See “Compare Project Versions” on page 12-21.

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 appropriate comments. See “Review Coding Rule Violations and Run-Time Checks” on page 12-22.

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 12-27.

Polyspace calculates metrics that are used to determine whether your code fulfills predefined software quality objectives. You can redefine these software quality objectives. See “Customizing Software Quality Objectives” on page 12-30.

Setting Up Verification to Generate Metrics

You can run, either manually or automatically, verifications that generate metrics. In each case, the Polyspace product 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 Polyspace verification environment:

- 1** Select the Project Manager perspective.
- 2** In **Project Browser**, select the project that you want to verify.
- 3** In the **Configuration** view, under **Analysis options > General**, select the following check boxes:
 - **Send to Polyspace Server**
 - **Add to results repository**
 - **Calculate code metrics** — Generates metrics about code complexity for the **Code Metrics** view. See “Review Code Complexity” on page 12-29.

For more information, see Chapter 3, “Setting Up a Verification Project” and Chapter 6, “Running a Verification”.

To set up scheduled, automatic verification runs, see “Specifying Automatic Verification” on page 12-4.

Specifying 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 there are 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.pspj` 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>
  <SmtConfiguration>
  </SmtConfiguration>
</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 12-6.

After creating `Projects.pspj`, place the file in the following folder on the Polyspace server:

```
/var/Polyspace/results-repository
```

Note If the flag `process_automation` in your configuration file `polyspace.conf` is set to `yes`, then, when you start your Polyspace Queue Manager server, Polyspace generates two template files in the results repository folder:

- `ProcessAutomationWindowsTemplate.psproj` for Windows
- `ProcessAutomationLinuxTemplate.psproj` for Linux

Use the appropriate template to create your `Projects.psproj` file.

For more information about the configuration file `polyspace.conf`, see “Manual Configuration of the Polyspace Server” in the *Polyspace Installation Guide*.

Element and Attribute Data for `Projects.psproj`

The following topics describe the data required to configure automatic verification.

Project. Specify three attributes:

- `name`. Project name as it appears in Polyspace Metrics.
- `language`. C, Cpp, Ada, or Ada95. Case insensitive.
- `verificationKind`. Mode, which is either Integration or Unit-by-Unit. Case insensitive.

For example,

```
<Project name="Demo_C" language="C" verificationKind="Integration">
```

The Project element also contains the following elements:

- “Options” on page 12-7
- “LaunchingPeriod” on page 12-7
- “Commands” on page 12-8
- “Users” on page 12-9

Options. Specify a list of all 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
  -code-metrics
  -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 12-9.

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 there are no 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:

```
<SmtplibConfiguration server="smtp.yourcompany.com" port="25">
```

Example of Projects.psproj

The following is an example of Projects.psproj:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Polyspace Metrics Automatic Verification Project File -->
<Configuration>
<Project name="Demo_C" language="C" verificationKind="Integration">
  <Options>
    -02
    -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
    -code-metrics
    -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>
```

Accessing Polyspace Metrics

In this section...

“Monitoring Verification Progress” on page 12-13

“Web Browser Support” on page 12-14

To go to the Polyspace Metrics project index, in the address bar of your Web browser, enter the following URL:

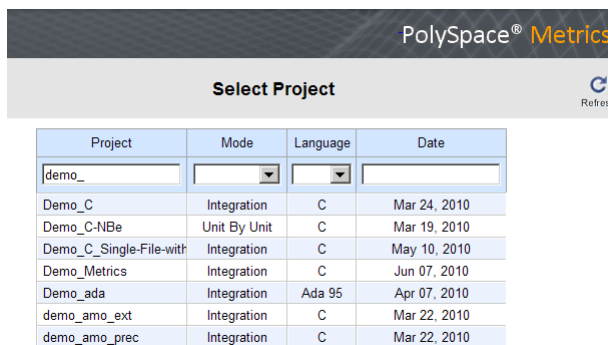
```
http:// ServerName: PortNumber
```

where

- *ServerName* is the name or IP address of the server that is your Polyspace Queue Manager.
- *PortNumber* is the Web server port number (default 8080)

See also “Configuring Polyspace Software” in the *Polyspace Installation Guide*.

The following graphic is an example of a project index.



Project	Mode	Language	Date
demo_			
Demo_C	Integration	C	Mar 24, 2010
Demo_C-NBe	Unit By Unit	C	Mar 19, 2010
Demo_C_Single-File-with	Integration	C	May 10, 2010
Demo_Metrics	Integration	C	Jun 07, 2010
Demo_ada	Integration	Ada 95	Apr 07, 2010
demo_amo_ext	Integration	C	Mar 22, 2010
demo_amo_prec	Integration	C	Mar 22, 2010


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 anywhere to go to the **Summary** view for that project.

Monitoring Verification Progress

In the **Summary > Verification Status** column, Polyspace Metrics provides status information for each verification in the project. The status can be queued, running, or completed.

If the verification mode is **Unit By Unit**, the software provides status information in each unit row. If the verification mode is **Integration**, the software provides status information in the parent row only.

If the verification status is **running** (and you have installed the Polyspace product on your computer), you can monitor progress of the verification with the Polyspace Queue Manager.

To open the Progress Monitor of the Polyspace Queue Manager:

- 1 In the **Summary > Verification Status** column, right-click the parent or unit cell with the status **running**.

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 (8)	queued (ID=39)											
1.0 (7)	queued (ID=38)											
1.0 (6)	running (ID=13)											
1.0 (5)	completed (P/)				12		83.8%	176	3	40	19	0.0%
1.0 (4)	completed (P/)				8		83.8%	176	3	40	19	0.0%
1.0 (3)	completed (P/)				12		83.8%	176	3	40	19	0.0%

- 2 From the context menu, select **Follow Progress**.

The Progress Monitor opens in the Polyspace verification environment.

For information, see “Monitoring Progress Using Queue Manager” on page 6-17.

Web Browser Support

Polyspace Metrics supports the following Web browsers:

- Internet Explorer® 7, Internet Explorer 6
- Firefox®

Polyspace Metrics is optimized for Internet Explorer 7 and Firefox, and has also been tested — but not optimized — for Internet Explorer 6.

Note To use Polyspace Metrics, you must install on your computer Java™, version 1.4 or later.

What You Can Do with Polyspace Metrics

In this section...

“Review Overall Progress” on page 12-15

“Displaying Metrics for Single Project Version” on page 12-19

“Creating a File Module and Specifying Quality Level” on page 12-20

“Compare Project Versions” on page 12-21

“Review New Findings” on page 12-21

“Review Coding Rule Violations and Run-Time Checks” on page 12-22

“Fix Defects” on page 12-27

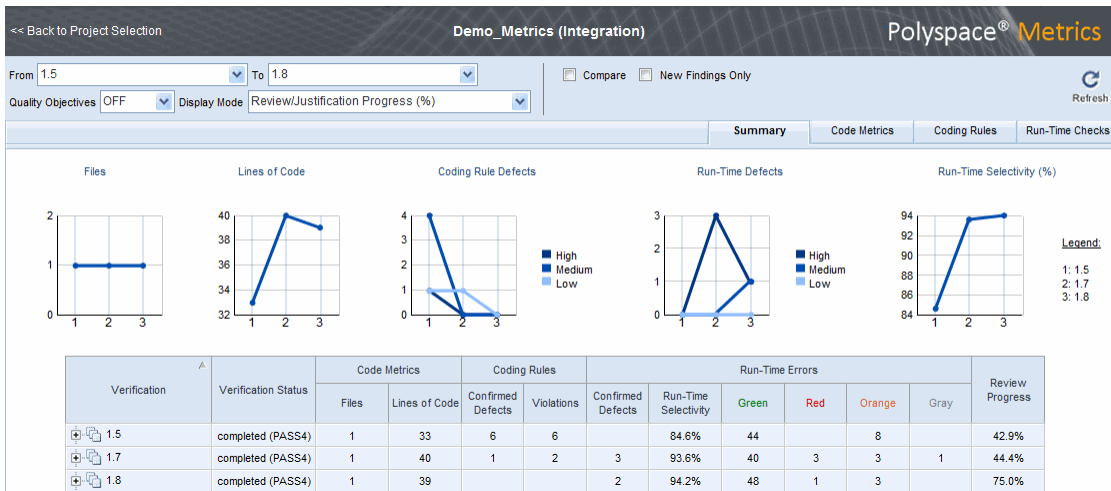
“Review Code Complexity” on page 12-29

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 “Accessing Polyspace Metrics” on page 12-12.
- 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 “Creating a File Module and Specifying Quality Level” on page 12-20.

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, SQO-3. Under **Software Quality Objectives**, in the **Level** cell, select SQO-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 “Customizing Software Quality Objectives” on page 12-30.

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	Quality Status	Level	Review Progress
V4	3	99.4%	272	66.7%	3	100.0%	6	100.0%	27	100.0%	6	FAIL	SQO-4	100.0%
└─ _polyspace__stdstubs.c		100.0%	14	100.0%	1	100.0%		100.0%	2	100.0%		PASS	SQO-4	100.0%
└─ _example.c	2	99.0%	83	0.0%	1	100.0%	2	100.0%	8	100.0%	3	FAIL	SQO-4	100.0%
└─ _initialisations.c	1	97.7%	41	100.0%		100.0%	1	100.0%	1	100.0%		PASS	SQO-4	100.0%
└─ _main.c		100.0%	9	100.0%		100.0%	1	100.0%	3	100.0%	2	PASS	SQO-4	100.0%
└─ _single_file_analysis.c		100.0%	82	100.0%	1	100.0%	2	100.0%	8	100.0%	1	PASS	SQO-4	100.0%
└─ _tasks1.c		100.0%	26	100.0%		100.0%		100.0%	3	100.0%		PASS	SQO-4	100.0%
└─ _tasks2.c		100.0%	17	100.0%		100.0%		100.0%	2	100.0%		PASS	SQO-4	100.0%

To view any of these checks in the Polyspace verification environment, in the appropriate cell, click the numeric value for the check. The Polyspace verification environment opens with the Run-Time Check perspective displaying verification information for this check.

Note If you update any check information through the Polyspace verification environment (see “Review Coding Rule Violations and Run-Time Checks” on page 12-22), 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 Coding Rule Violations and Run-Time Checks” on page 12-22.

Displaying 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:

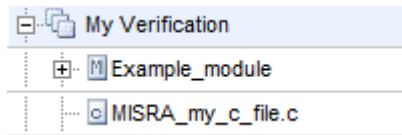
- Bar charts with file defect information, ordering the files according to the number of defects in each file
- A table with information about the selected project version

Creating a File Module and Specifying 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.

V3 vs V4	Code Metrics			Coding Rules			Run-Time Errors			Overall Evolution
	Files	Lines of Code	All Metrics Evolution	Confirmed Defects	Violations	All Metrics Evolution	Confirmed Defects	Run-Time Reliability	All Metrics Evolution	
V4	6	463	▼	1 (+1) ▼	4	◆	3 (+3) ▼	99.7% (+9.8%) ▲	◆	◆
		82						100.0%	◆	◆
		49						100.0%	◆	◆
		45		1 (+1) ▼	1	◆		100.0% (+40.0%) ▲	▲	◆
		80						100.0% (+1.7%) ▲	▲	▲
		136	▼		3		1 (+1) ▼	100.0% (+11.3%) ▲	◆	◆
		71					1 (+1) ▼	97.7% (+2.3%) ▲	◆	◆
							1 (+1) ▼	100.0% (+4.6%) ▲	◆	◆

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 Coding Rule Violations and Run-Time Checks

If you have installed Polyspace on your computer, you can use Polyspace Metrics to review and add information about coding rule violations and run-time checks produced by a verification.

Note By default, Polyspace Metrics does not use coding rule violations to evaluate software quality for C++ projects. However, you can customize software quality objectives (SQO) to incorporate coding rule violations. See “Customizing Software Quality Objectives” on page 12-30.

You may use the **Review Progress** metric in the **Summary** view to decide when your team of developers should start work on the next version of the software. For example, you may wait until the review is complete (**Review Progress** cell displays 100%), before informing your development team.

Coding Rule Violations

Consider an example, where you see the following in the **Summary** view.

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	Orange	Red		Gray
 V4	completed (PASS4)	6	463		4	3	93.2%	272	27	3	90	31.5%

Review progress is incomplete (31.5%), and there are four coding rule violations. In the **Violations** cell, click 4, which takes you to Polyspace Metrics **Coding Rules** view.

Verification	Coding Rules		
	Confirmed Defects	Reviewed	Violations
V4		0.0%	4
example.c		0.0%	3
include.h		100.0%	
main.c		0.0%	1
single_file_private		100.0%	

The **Reviewed** column reveals the files that you have not reviewed completely. In this example, `example.c` is unreviewed (0.0%). To continue reviewing violations in this file, expand `example.c`.

V4		0.0%	4
example.c		0.0%	3
17.4 Array indexi		0.0%	3

You see that there are three violations of rule 17.4.

Note If you want to review coding rule violations with reference to the coding rules, in the Polyspace Metrics **Coding Rules** view, from the **Group by** drop-down list, select **Coding Rules** and expand a specific coding rule.

On the row corresponding to rule 17.4, click the value in the **Review Progress** cell, 3. This action opens the Polyspace verification environment and takes you to the Coding Rules perspective. In **Assistant Coding Rules**, you see the list of unreviewed violations.

Rule	File	Line	Column
17.4	example.c	97	7
17.4	example.c	114	21
17.4	example.c	118	14

Double-click a row. In **Rule details**, you see information about the location of this violation.

Rule details
Rule: 17.4 (warning): Array indexing shall be the only allowed form of pointer arithmetic.
File: /mathworks/home/nbertolo/PolyspaceStudio/TESTS/Demo_C/sources/example.c line 118 (column 14)

Select the **MISRA C** view.

w/e	Rule	File	Line	Col	Classification	Status	Justified	Comment
warning	17.4	example.c	97		7 No action planned	Improve	<input checked="" type="checkbox"/>	
warning	17.4	example.c	114		21 No action planned	Justify with annotations	<input type="checkbox"/>	
warning	17.4	example.c	118		14 Justify with annotations	Restart with different options	<input type="checkbox"/>	

If you want to classify the violation as a defect, from the **Classification** cell drop-down list, select **High**, **Medium**, or **Low**. This will increment the **Confirmed Defect** value in Polyspace Metrics.

You can assign a status to this violation. From the **Status** drop-down list, select a status, for example, **Fix** or **No action planned**. When you assign a status to a violation, the software considers the violation to be *reviewed*.

If you consider the presence of a violation justifiable, select the **Justified** check box. In the **Comments** column, you can enter remarks justifying this violation.

Save the review. See “Saving Review Comments and Justifications” on page 12-27.

Note Classifying a coding rule violation as a defect or assigning a status for an unreviewed violation in the Polyspace verification environment, increases the corresponding metric values (**Confirmed Defects** and **Review Progress**) in the **Summary** and **Coding Rules** views of Polyspace Metrics.

Run-Time Checks

Consider an example, where you see the following in the **Summary** view.

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	Orange		Red	Gray
V4	completed (PASS4)	6	483		4	3	93.2%	272	27	3	90	31.5%

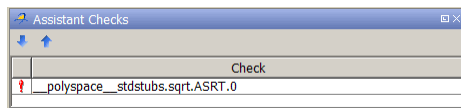
Under **Run-Time Errors**, click any cell value. This action takes you to the **Run-Time Checks** view.

Verification	Confirmed Defects	Run-Time Selectivity	Green Code	Systematic Runtime Errors (Red Checks)		Unreachable Code (Gray Checks)		Other Runtime Errors (Orange Checks)		Non-terminating Constructs		Review Progress
			Checks	Reviewed	Checks	Reviewed	Checks	Reviewed	Checks	Reviewed	Checks	
V4	3	93.2%	272	66.7%	3	6.7%	90	100.0%	27	100.0%	6	32.5%
└─ __polyspace__stdc		97.7%	14	0.0%	1	0.0%	69	100.0%	2	100.0%		2.8%
└─ example.c	2	92.2%	83	100.0%	1	25.0%	8	100.0%	8	100.0%	3	70.0%
└─ initialisations.c	1	97.8%	41	100.0%		33.3%	3	100.0%	1	100.0%		50.0%
└─ main.c		85.0%	9	100.0%		16.7%	6	100.0%	3	100.0%	2	54.5%
└─ single_file_analys		91.7%	82	100.0%	1	50.0%	4	100.0%	8	100.0%	1	85.7%
└─ tasks1.c		89.7%	26	100.0%		100.0%		100.0%	3	100.0%		100.0%
└─ tasks2.c		89.5%	17	100.0%		100.0%		100.0%	2	100.0%		100.0%

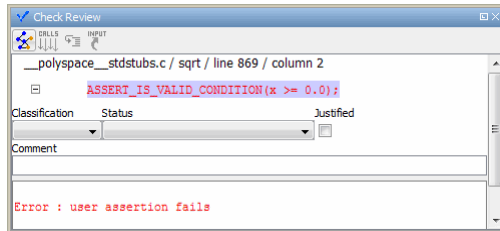
The **Review Progress** column reveals the progress level for each file, for example, 2.8% for `__polyspace__stdstubs.c`. Expand `__polyspace__stdstubs.c`.

Verification	Confirmed Defects	Run-Time Selectivity	Green Code	Systematic Runtime Errors (Red Checks)		Unreachable Code (Gray Checks)		Other Runtime Errors (Orange Checks)		Non-terminating Constructs		Review Progress
			Checks	Reviewed	Checks	Reviewed	Checks	Reviewed	Checks	Reviewed	Checks	
V4	3	93.2%	272	66.7%	3	6.7%	90	100.0%	27	100.0%	6	32.5%
└─ __polyspace__stdstubs.c		97.7%	14	0.0%	1	0.0%	69	100.0%	2	100.0%		2.8%
└─ ASRT - failure of user assertion		86.7%		0.0%	1	0.0%	12	100.0%	2			13.3%
└─ IRV - function returns an initializ		100.0%	1	100.0%		0.0%	3	100.0%				0.0%
└─ NVL - non-initialized global variat		100.0%	1	100.0%		100.0%		100.0%				100.0%
└─ NVL - non-initialized local variat		100.0%	12	100.0%		0.0%	31	100.0%				0.0%
└─ OVFL (Float) - overflow		100.0%		100.0%		0.0%	18	100.0%				0.0%
└─ ZDV (Float) - denominator must		100.0%		100.0%		0.0%	5	100.0%				0.0%

In the row containing the ASRT check, click the value in the red **Checks** cell, which opens the Polyspace verification environment with the Run-Time Checks perspective. The software displays the ASRT check in **Assistant Checks**.



Double-click the row with the ASRT check, which brings the check into **Check Review**.



Using the drop-down list for the **Classification** field, you can classify the check as a defect (High, Medium, or Low) or specify that the check is **Not** a defect.

Using the drop-down list for the **Status** field, you can assign a status for the check, for example, **Fix** or **Investigate**. When you assign a status, the software considers the check to be *reviewed*.

If you think that the presence of the check in your code can be justified, select the check box **Justified**. In the **Comment** field, enter remarks that justify this check.

Save the review. See “Saving Review Comments and Justifications” on page 12-27.

Note Classifying a run-time check as a defect or assigning a status for an unreviewed check in the Polyspace verification environment increases the corresponding metric values (**Confirmed Defects** and **Review Progress**) in the **Summary** and **Run-Time Checks** views of Polyspace Metrics.

Specifying Download Folder for Polyspace Metrics

When you click a coding rule violation or run-time check, Polyspace downloads result files from the Polyspace Metrics web interface to a local folder. You can specify this folder as follows:


- 1 Select **Options > Preferences > Server configuration**.
- 2 If you want to download result files to the folder from which the verification is launched, select the check box **Download results automatically**.

3 If this launch folder does not exist, specify another path in the **Folder** field.

If you do not specify a folder using step 2 or 3, when you click a violation or check, the software opens a file browser. Use this browser to specify the download location.

Saving Review Comments and Justifications

By default, when you save your project (**Ctrl+S**), the software saves your comments and justifications to a local folder. See “Specifying Download Folder for Polyspace Metrics” on page 12-26.

If you want to save your comments and justifications to a local folder *and* the Polyspace Metrics repository, on the Run-Time Checks 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 database**.

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

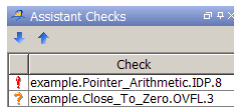
You can use Polyspace Metrics to access defects that you must fix.

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.

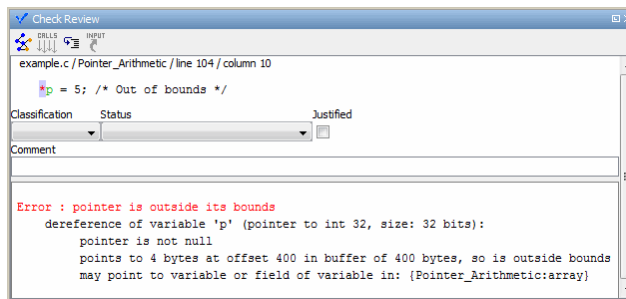
Verification	Confirmed Defects	Run-Time Selectivity	Green Code	Systematic Runtime Errors (Red Checks)	
			Checks	Reviewed	Checks
V4	4	93.2%	272	100.0%	3
└─ __polyspace_stc	1	97.7%	14	100.0%	1
└─ example.c	2	92.2%	83	100.0%	1
└─ initialisations.c	1	97.8%	41	100.0%	
└─ main.c		85.0%	9	100.0%	
└─ single_file_analys		91.7%	82	100.0%	1
└─ task1.c		89.7%	26	100.0%	
└─ task2.c		89.5%	17	100.0%	

In the **Confirmed Defects** column, click a non-zero cell value. For example, if you click 2, the Polyspace verification environment opens with the checks visible in **Assistant Checks**.



Check
example.Pointer_Arithmetic.IDP.8
example.Close_To_Zero.OVFL.3

Double-click the row containing a check. In **Check Review**, 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 “SQO Level 1” on page 12-32.
- How Polyspace evaluates these metrics and how you can customize code complexity metrics, see “About Customizing Software Quality Objectives” on page 12-30 and “SQO Level 1” on page 12-32.

Customizing Software Quality Objectives

In this section...

“About Customizing Software Quality Objectives” on page 12-30

“SQO Level 1” on page 12-32

“SQO Level 2” on page 12-35

“SQO Level 3” on page 12-35

“SQO Level 4” on page 12-36

“SQO Level 5” on page 12-36

“SQO Level 6” on page 12-36

“SQO Exhaustive” on page 12-37

“Coding Rules Set 1” on page 12-37

“Coding Rules Set 2” on page 12-39

“Run-Time Checks Set 1” on page 12-41

“Run-Time Checks Set 2” on page 12-42

“Run-Time Checks Set 3” on page 12-43

“Status Acronyms” on page 12-43

About Customizing 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. See “Configuring Polyspace Server Software” in the *Polyspace Installation Guide*.

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 12-37.

The following topics provide information about `MetricsDefinitions` elements and how SQO levels are calculated. Use this information when you modify or create elements.

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 SQO Level 1.

By default, Polyspace Metrics does not evaluate C++ coding rule violations for SQO Level 1. However, you can incorporate coding rule violations by adding the sub-element `Num_Unjustified_Violations` to the list of sub-elements in *SQO 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 12-37.

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
comf	20	Comment density of a file
path	80	Number of paths through a function
goto	0	Number of <code>goto</code> statements
vg	10	Cyclomatic complexity
calling	5	Number of calling functions
calls	7	Number of calls
param	5	Number of parameters per function
stmt	50	Number of instructions per function
level	4	Number of call levels in a function
return	1	Number of return statements in a function
vocf	4	Language scope
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 12-37	Number of unjustified violations of MISRA C rules specified by “Coding Rules Set 1” on page 12-37

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 appropriate 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>
</SQO>

```

SQO Level 2

The default SQO Level 2 element is:

```

<SQO ID="SQO-2" ParentID="SQO-1">
    <Num_Unjustified_Red>0</Num_Unjustified_Red>
    <Num_Unjustified_NT_Constructs>0</Num_Unjustified_NT_Constructs>
</SQO>

```

To fulfill requirements of SQO Level 2, the code must meet the requirements of SQO 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)

SQO Level 3

The default SQO Level 3 element is:

```

<SQO ID="SQO-3" ParentID="SQO-2">
    <Num_Unjustified_UNR>0</Num_Unjustified_UNR>
</SQO>

```

To fulfill requirements of SQO Level 3, the code must meet the requirements of SQO Level 2 **and** the number of unjustified UNR checks must not exceed the threshold (default is zero).

SQO Level 4

The default SQO Level 4 element is:

```
<SQO ID="SQO-4" ParentID="SQO-3">  
  <Percentage_Proven_Or_Justified>Runtime_Checks_Set_1</Percentage_Proven_Or_Justified>  
</SQO>
```

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 12-41.

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 12-39.
- Percentage of green and justified checks must not be less than the thresholds specified by “Run-Time Checks Set 2” on page 12-42

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 12-43.

SQO Exhaustive

The default Exhaustive element is:

```
<SQO ID="Exhaustive" ParentID="SQO-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>
</SQO>
```

To fulfill the requirements for this level, the code must meet the requirements of SQO 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 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 appropriate `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 appropriate `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 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>
  <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:

$$(\text{green checks} + \text{justified orange checks}) / (\text{green checks} + \text{all orange checks})$$

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 12-41, 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 12-41, 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**. See “Run-Time Checks” on page 12-24.

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 all users of your Polyspace server.

Tips for Administering Results Repository

In this section...

“Through the Polyspace Metrics Web Interface” on page 12-45

“Through Command Line” on page 12-46

“Backup of Results Repository” on page 12-48

Through the Polyspace Metrics Web Interface

You can rename or delete projects and verifications.

Project Renaming

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.

Project Deletion

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**.

Verification Renaming

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.

Verification Deletion

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**.

Through Command Line

You can run the following batch command with various options.

```
Polyspace_Common/RemoteLauncher/[w]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. Can be omitted if, in the Polyspace Preferences dialog box, on the **Server configuration** tab, you have specified a server name or clicked **Automatically detect the remote server**. MathWorks does not recommend the latter. See “Configuring Polyspace Client Software” in the *Polyspace Installation Guide*.
 - *old_prog* — Current project name
 - *new_prog* — New project name
 - *old_version* — Old version name
 - *new_version* — New version name
 - `-f` — Specifies that no confirmation is 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. Can be omitted if, in the Polyspace Preferences dialog box, on the **Server configuration** tab, you have specified a server name or clicked **Automatically detect the remote server**. MathWorks does not recommend the latter. See “Configuring Polyspace Client Software” in the *Polyspace Installation Guide*.
- *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 no confirmation is 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 Deletion Examples

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 of 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 “Configuring the Polyspace Server Software” in the *Polyspace Installation Guide*.

For example, on a UNIX 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 `$cd PolyspaceRLDatas`
- 2 `$unzip Path_to_backup_folder/results-repository.zip`

Using Polyspace Software in the Eclipse IDE

-
- “Verifying Code in the Eclipse IDE” on page 13-2

Verifying Code in the Eclipse IDE

In this section...

“Creating an Eclipse Project” on page 13-3

“Setting Up Polyspace Verification with Eclipse Editor” on page 13-4

“Launching Verification from Eclipse Editor” on page 13-6

“Reviewing Verification Results from Eclipse Editor” on page 13-6

“Using the Polyspace Spooler” on page 13-7

You can apply the powerful code verification of Polyspace software to code that you develop within the Eclipse Integrated Development Environment (IDE).

A typical workflow is:

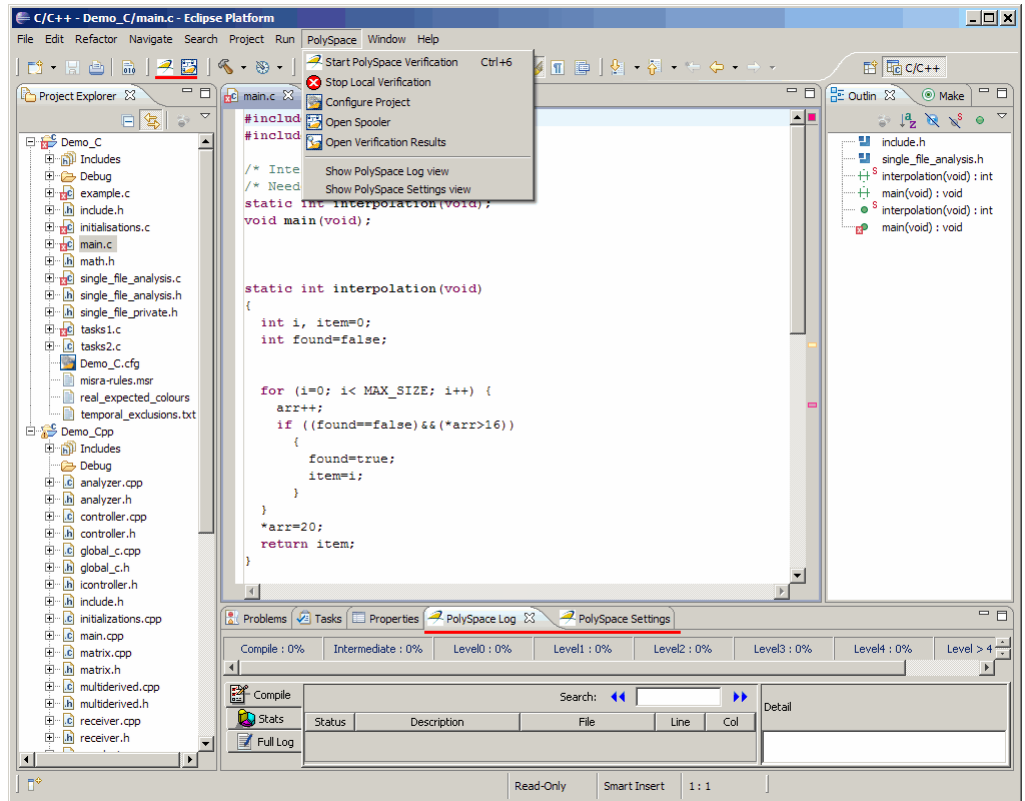
- 1** Use the Eclipse™ editor to create an Eclipse project and develop code within your project.
- 2** Set up the Polyspace verification by configuring analysis options and settings.
- 3** Start the verification and monitor the process.
- 4** Review the verification results.

Install the Polyspace plug-in for Eclipse IDE before you verify code in Eclipse IDE. For more information, see “Polyspace Plug-In Requirements” and “Installing the Polyspace Plug-In for Eclipse IDE” in the *Polyspace Installation Guide*.

Once you have installed the plug-in, in the Eclipse editor, you have access to:

- A **Polyspace** menu
- Toolbar buttons you use to launch a verification and open the Polyspace spooler

- Polyspace Log and Polyspace Setting views



Creating an Eclipse Project

If your source files do not belong to an Eclipse project, then create one 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.

Setting Up Polyspace Verification with Eclipse Editor

Analysis Options

To specify analysis options for your verification:

- 1** In **Project Explorer**, select the project or files that you want to verify.
- 2** Select **Polyspace > Configure Project** to open the Project Manager perspective of the Polyspace Verification Environment.
- 3** Under **Analysis options**, select your options for the verification process.
- 4** Save your options.

For information on *how* to choose your options, see “Options Description” in the *Polyspace Products for C Reference Guide*

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.
-

Other Settings

In the **Polyspace Settings** view, specify:

- In the **Results folder** field, the location of your results folder .
- The required **Verification level**, for example, `Level4`.

If the item that you select in the **Project Explorer** is not a C++ class, then you can also do the following in the **Polyspace Settings** view

- Generate a main (if the item you select does not contain one) by selecting the **Generate a main** check box. If you want to change the default behavior of the main generator, specify advanced settings through the `-main-generator-writes-variables` and `-main-generator-calls` options in the Project Manager perspective of the Polyspace Verification Environment. Select **Polyspace > Configure Project** to open this window.
- Specify the `-function-called-before-main` option. In the **Startup function to call** field, enter the name of the function that you want to call before all selected functions in main.

Setting Up Verification for a Single C++ Class

You can use the **Polyspace Settings** view to configure verification of a single class:

- 1 In **Project Explorer**, select your class.
- 2 In the **Polyspace Settings** view, select the **Verify the class contents only** check box.

This approach is equivalent to specifying the `-class-analyzer` and `-class-only` options. If necessary, you can use the Project Manager perspective of the Polyspace Verification Environment (**Polyspace > Configure Project**) to specify other options, for example, `-class-analyzer-calls`.

Launching Verification from Eclipse Editor

To launch 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 Verification**, or select **Polyspace > Start Polyspace Verification**.

You can see the progress of the verification in the **Polyspace Log** view. If you see an error or warning, double-click it to go to the corresponding location in the source code.

To stop a verification, select **Polyspace > Stop Local Verification**.

For more information on monitoring the progress of a verification, see Chapter 6, “Running a Verification” in the *Polyspace Products for C User Guide*.

Reviewing Verification Results from Eclipse Editor

Use the Run-Time Checks perspective of the Polyspace Verification Environment to examine results of the verification:

- 1 Select **Polyspace > Open Verification Results** to open the Run-Time Checks perspective of the Polyspace Verification Environment.
- 2 If results are available in the specified **Results folder**, then these results appear automatically in the Run-Time Checks perspective.

For information on reviewing and understanding Polyspace verification results, see Chapter 8, “Reviewing Verification Results” in the *Polyspace Products for C User Guide*.

Using the Polyspace Spooler

Use the Polyspace spooler to manage jobs running on remote servers. To open the spooler, select **Polyspace > Open Spooler** .

For more information, see “Managing Verification Jobs Using the Polyspace Queue Manager” on page 6-14 in the *Polyspace Products for C User Guide*.

Using Polyspace Software in Visual Studio

-
- “Verifying Code in Visual Studio” on page 14-2
-

Verifying Code in Visual Studio

In this section...

“Creating a Visual Studio Project” on page 14-4

“Setting Up and Starting a Polyspace Verification in Visual Studio” on page 14-5

“Monitoring a Verification” on page 14-13

“Reviewing Verification Results in Visual Studio” on page 14-15

“Using the Polyspace Spooler” on page 14-15

You can apply the powerful code verification functionality of Polyspace software to code that you develop within the Visual Studio Integrated Development Environment (IDE).

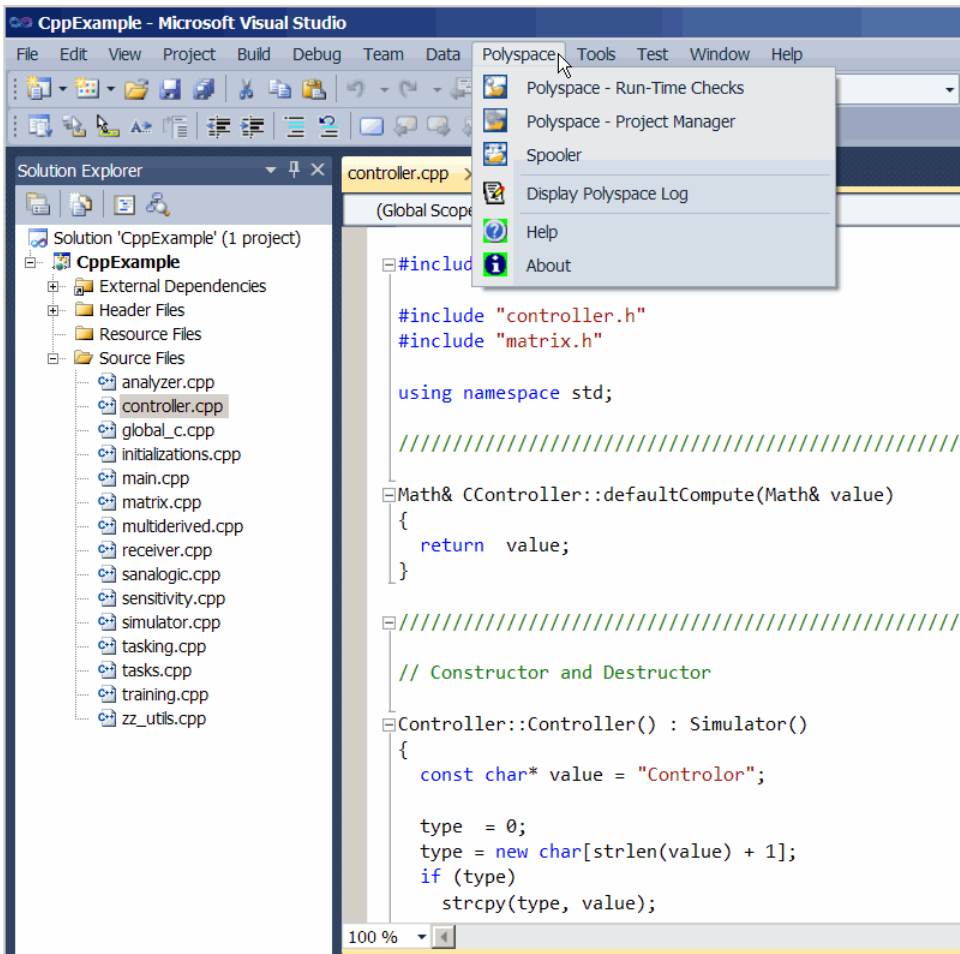
A typical workflow is:

- 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 plug-in for Visual.NET. For more information, see “Polyspace Plug-In Requirements” and “Installing the Polyspace C++ Add-In for Visual Studio” in the *Polyspace Installation Guide*.

Once you have installed the plug-in, in the Visual Studio editor, you can access:

- A **Polyspace** menu
- A **Polyspace Log** view



Creating a Visual Studio Project

If your source files do not belong to a Visual Studio project, you can create one 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 an appropriate location, for example, C:\Polyspace\Visual. The software creates some files and a Project Console Win32.

To add files to your project:

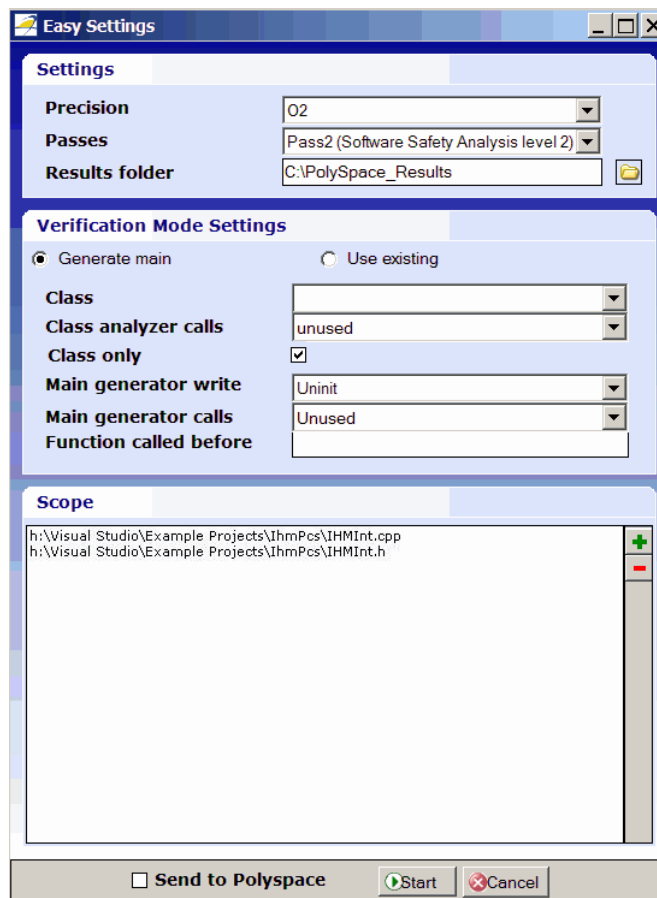
- 1 Select the **Browse the solution** tab.
- 2 Right-click the project name. From the pop-up menu, select **Add > Add existing element** .
- 3 Add the files you want (for example, `matrix.cpp` and `matrix.h` in `Polyspace_Install/Examples/Demo_Cpp_Long/sources`) to the project (for example, CppExample).

Setting Up and Starting a Polyspace Verification 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 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 “Option Descriptions for C++ Code” in the *Polyspace Products for C/C++ Reference*.

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 “Setting Standard Polyspace Options” on page 14-11.

- 4** Click **Start** to start the verification.

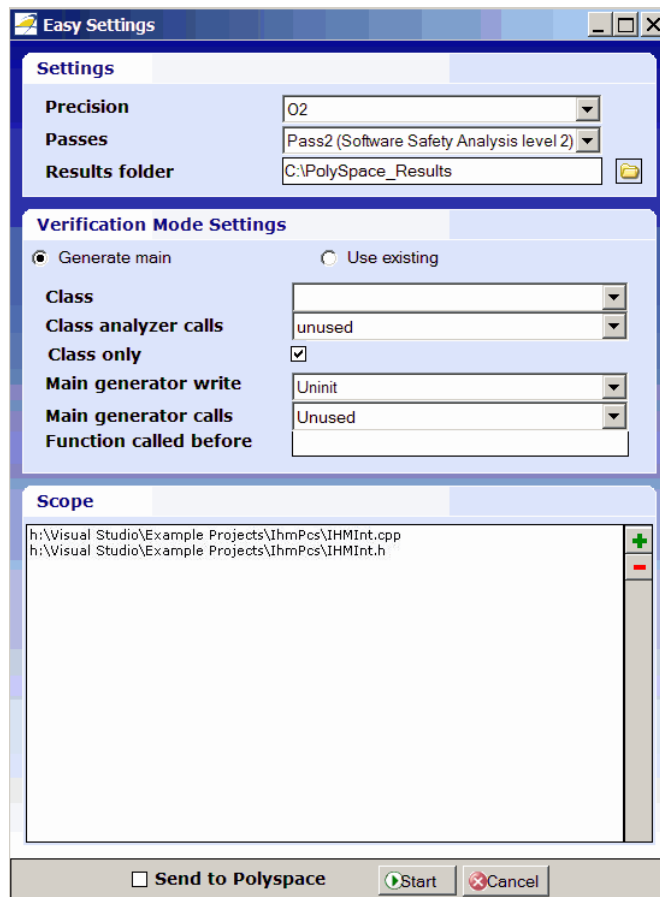
Verifying Classes


In the Easy Settings dialog box, you can verify a C++ class by modifying the scope option.

To verify a class:

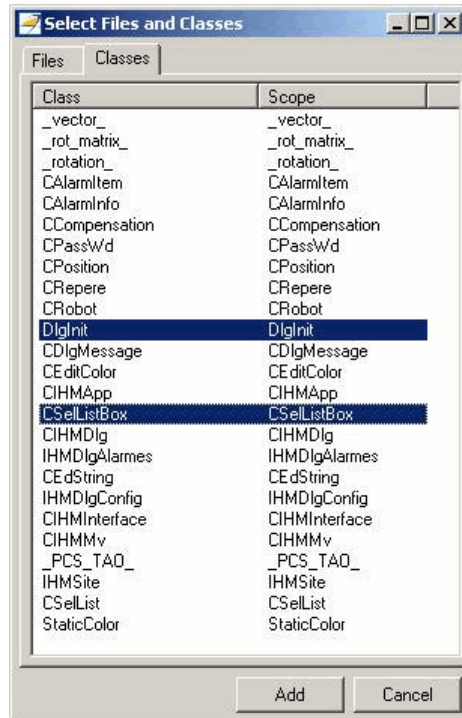
- 1 In the Visual Studio Solution Explorer, right-click a file and select **Polyspace Verification**.

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.

Verifying 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 Chapter 6, “Running a Verification” in the Polyspace® Products for C/C++ User’s Guide on page 1.

Importing Visual Studio Project Information into Polyspace Project

You can extract informations from a Visual Studio project file (vcproj) to help configure your Polyspace project.

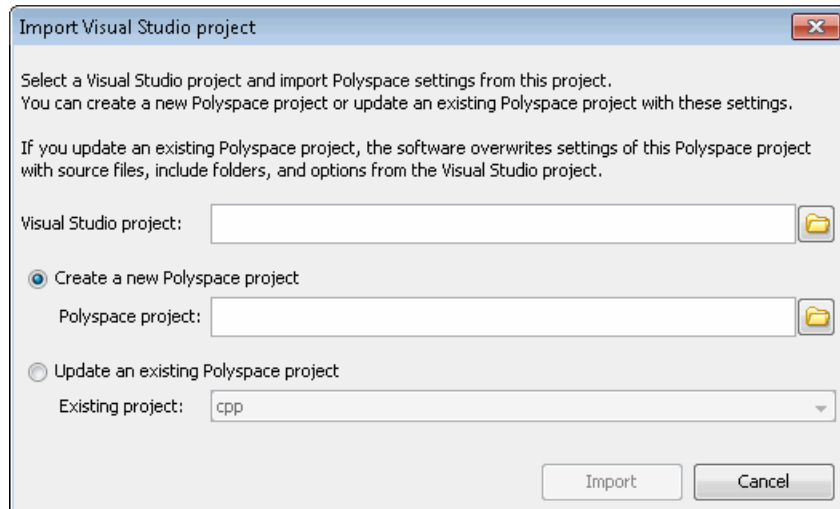
The Visual Studio import can retrieve the following information from a Visual Studio project:

- Source files
- Include folders
- Preprocessing directives (-D, -U)
- Polyspace specific options about dialect used

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.

Setting 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 (.cfg) 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 .cfg file used by the Project Manager does not correspond to the .cfg file of the current project.

For information on *how* to choose your options, see “Option Descriptions for C++ Code” in the *Polyspace Products for C/C++ Reference*.

The 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:

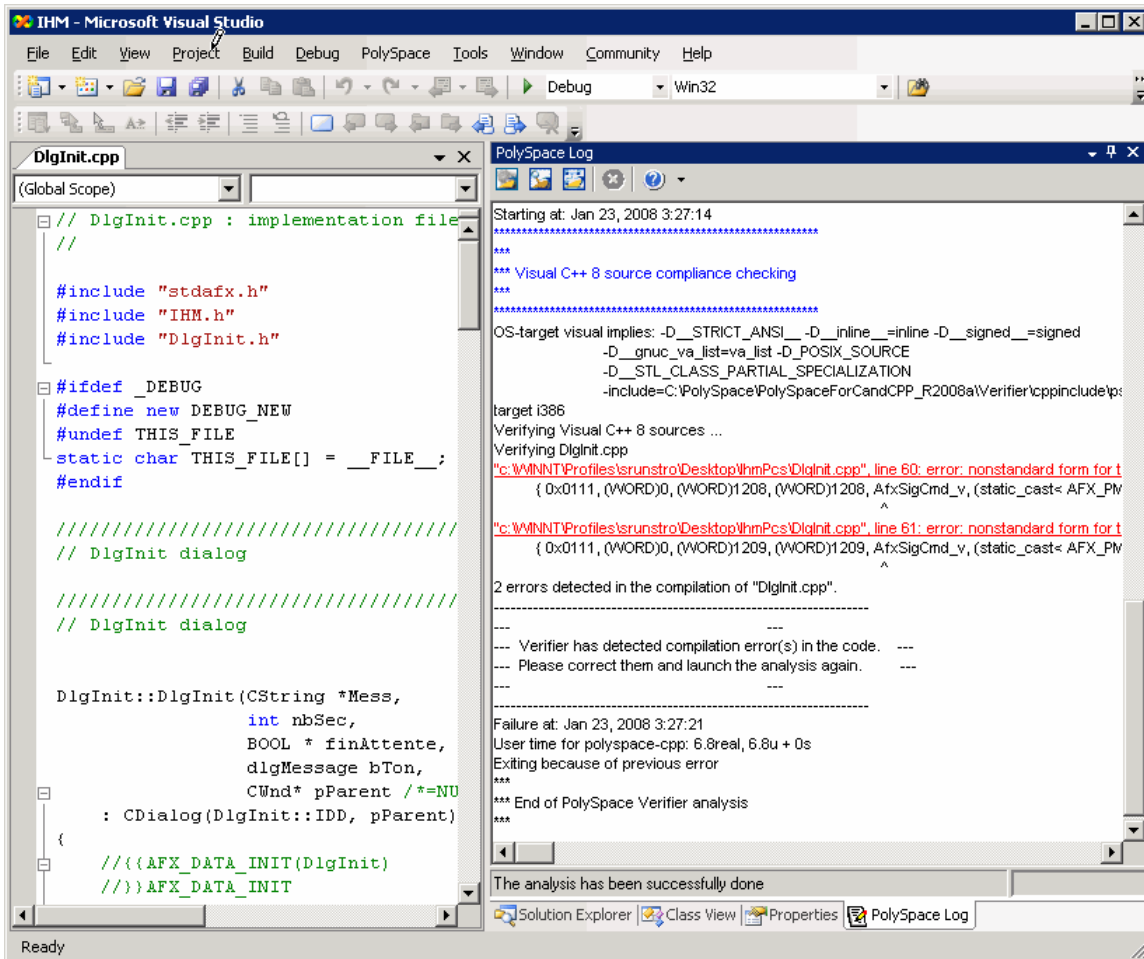
Visual Studio Option	Polyspace Option
/D <name>	-D <name>
/U <name>	-U <name>
/MT	-D_MT
/MTd	-D_MT -D_DEBUG
/MD	-D_MT -D_DLL
/MDd	-D_MT -D_DLL -D_DEBUG
/MLd	-D_DEBUG
/Zc:wchar_t	-wchar-t-is keyword
/Zc:forScope	-for-loop-index-scope in
/FX	-support-FX-option-results
/Zp[1,2,4,8,16]	-pack-alignment-value [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

Monitoring a Verification

Once you launch 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 “Managing Verification Jobs Using the Polyspace Queue Manager” on page 6-14 in the Polyspace® Products for C/C++ User’s Guide on page 1.

Reviewing Verification Results in Visual Studio

Select **Polyspace > Open Verification Results** to open the Run-Time Checks 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 Run-Time Checks perspective.

For information on reviewing and understanding Polyspace verification results, see Chapter 8, “Reviewing Verification Results” in the Polyspace® Products for C/C++ User’s Guide on page 1.

Using the Polyspace Spooler

You can use the Polyspace spooler to manage jobs that are run on remote servers. To open the spooler, select **Polyspace > Spooler** .

For more information, see “Managing Verification Jobs Using the Polyspace Queue Manager” on page 6-14 in the Polyspace® Products for C/C++ User’s Guide on page 1.

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 none are.

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.

A

- access sequence graph 8-49
- active project
 - definition 10-3
 - setting 10-3
- analysis options 3-22 3-26
 - generic targets 4-9
 - JSF++ compliance 3-37
 - MISRA C compliance 3-35 11-5
 - MISRA C++ compliance 3-37
- ANSI compliance 6-9

B

- batch run
 - verification 6-13

C

- call graph 8-48
- call tree view 8-14
- calling sequence 8-48
- cfg. *See* configuration file
- client 1-8 6-2
 - installation 1-15
 - verification on 6-31
- code view 8-19
- coding review progress view 8-14 8-56
- coding rule violations
 - justifying 11-22
- Coding Rules compliance 1-3
- Coding Rules perspective 1-8
- color-coding of verification results 1-2 1-4 8-16
- compile
 - log 7-10
- compile log
 - Project Manager 6-16 6-34
 - Spooler 6-14 6-17
- compile phase 6-9
- compliance

- ANSI 6-9
- JSF++ 3-37
- MISRA C 3-35 11-5
- MISRA C++ 3-37

- configuration file
 - definition 3-2
- contextual verification 2-5
- criteria
 - quality 2-8

D

- data range specifications 2-6
- downloading
 - results 8-8
 - results using command line 8-10
 - unit-by-unit verification results 8-11
- DRS 2-6

E

- error call graph 8-48

F

- files
 - includes 3-12 3-14 3-20
 - results 3-14 3-20
 - source 3-10 3-14 to 3-16 3-20
- filters 8-50
- folders
 - includes 3-12 3-14 3-20
 - results 3-14 3-20
 - sources 3-10 3-12 3-14 to 3-16 3-20

G

- generic target processors
 - definition 4-9
 - deleting 4-12
- global variable graph 8-49

H

hardware requirements 7-3

I

installation

Polyspace Client for C/C++ 1-15

Polyspace products 1-15

Polyspace Server for C/C++ 1-15

J

JSF C++ compliance

file exclusion 11-12

include folder exclusion 11-13

rules file 11-10

JSF++ compliance

analysis option 3-37

checking 3-37

violations 11-19

justifying coding rule violations 11-22

L

level

quality 2-8

licenses

obtaining 1-15

logs

compile

Project Manager 6-16 6-34

Spooler 6-14 6-17

full

Project Manager 6-16 6-34

Spooler 6-14 6-17

stats

Project Manager 6-16 6-34

Spooler 6-14 6-17

viewing

Project Manager 6-16 6-34

Spooler 6-14 6-17

M

manual mode

filters 8-50

use 8-44

MISRA C compliance

analysis option 3-35 11-5

checking 3-35 11-5

file exclusion 11-12

include folder exclusion 11-13

rules file 11-8

violations 11-19

MISRA C++ compliance

analysis option 3-37

checking 3-37

file exclusion 11-12

include folder exclusion 11-13

MISRA® C++ compliance

rules file 11-10

violations 11-19

O

objectives

quality 2-5

P

Polyspace Client for C/C++

installation 1-15

license 1-15

Polyspace In One Click

active project 10-3

overview 10-2

sending files to Polyspace software 10-5

starting verification 10-5

use 10-2

Polyspace products for C/C++

components 1-8

- installation 1-15
 - licenses 1-15
 - overview 1-2
 - related products 1-16
 - user interface 1-8
 - value 1-3
 - Polyspace Queue Manager Interface. *See* Spooler
 - Polyspace Server for C/C++
 - installation 1-15
 - license 1-15
 - Polyspace verification environment
 - opening 3-3
 - preferences
 - default server mode 6-9
 - server detection 7-4
 - Status 8-58
 - preprocessing files
 - opening 7-11
 - troubleshooting with 7-50
 - procedural entities view 8-14 8-16
 - reviewed column 8-60
 - product overview 1-2 to 1-3
 - progress bar
 - Project Manager window 6-16 6-34
 - project
 - creation 3-2
 - definition 3-2
 - file types
 - configuration file 3-2
 - folders
 - includes 3-3
 - results 3-3
 - sources 3-3
 - saving 3-25
 - Project Manager
 - batch run verification 6-13
 - monitoring verification progress 6-16 6-34
 - opening 3-3
 - overview 3-3
 - perspective 3-3
 - starting verification on client 6-31
 - starting verification on server 6-9
 - viewing logs 6-16 6-34
 - window
 - progress bar 6-16 6-34
 - Project Manager perspective 1-8
- Q**
- quality level 2-8
 - quality objectives 2-5 3-26
- R**
- related products 1-16
 - Polyspace products for linking to Models 1-16
 - Polyspace products for verifying Ada code 1-16
 - reports
 - generation 8-68
 - results
 - downloading from server 8-8
 - downloading using command line 8-10
 - folder 3-14 3-20
 - opening 8-12 to 8-13
 - report generation 8-68
 - unit-by-unit 8-11
 - reviewed column 8-60
 - reviewing coding rule violations,
 - classification 11-22
 - reviewing results, classification 8-56
 - reviewing results, status 8-56 8-58
 - robustness verification 2-5
 - rte view. *See* procedural entities view
 - Run Time Checks perspective
 - opening 8-12 to 8-13
 - Run-Time Checks perspective 1-8
 - call tree view 8-14
 - coding review progress view 8-14
 - modes

- selection 8-28
- procedural entities view 8-14
- selected check view 8-14
- source code view 8-14
- variables view 8-14
- window
 - overview 8-14

S

- selected check view 8-14 8-24 8-56
- server 1-8 6-2
 - detection 7-4
 - information in preferences 7-4
 - installation 1-15 7-4
 - verification on 6-9
- source code view 8-14 8-19
- Spooler 1-8
 - monitoring verification progress 6-14 6-17
 - removing verification from queue 8-8
 - use 6-14 6-17
 - viewing log 6-14 6-17
- Status, user defined 8-58

T

- target environment 3-21
- troubleshooting failed verification 7-2

V

- variables view 8-14 8-25 8-27
- verification
 - Ada code 1-16

- batch run 6-13
- C/C++ code 1-2 to 1-3
- client 6-2
- compile phase 6-9
- contextual 2-5
- failed 7-2
- monitoring progress
 - Project Manager 6-16 6-34
 - Spooler 6-14 6-17
- phases 6-9
- results
 - color-coding 1-2 1-4
 - opening 8-12 to 8-13
 - report generation 8-68
 - reviewing 8-8
- robustness 2-5
- run all 6-13
- running 6-2
- running on client 6-31
- running on server 6-9
- starting
 - from Polyspace In One Click 6-2 10-5
 - from Project Manager 6-2 6-32
- stopping 6-35
- troubleshooting 7-2
- with JSF C++ checking 11-18
- with MISRA C checking 11-18
- with MISRA C++ checking 11-18

W

- workflow
 - setting quality objectives 2-5