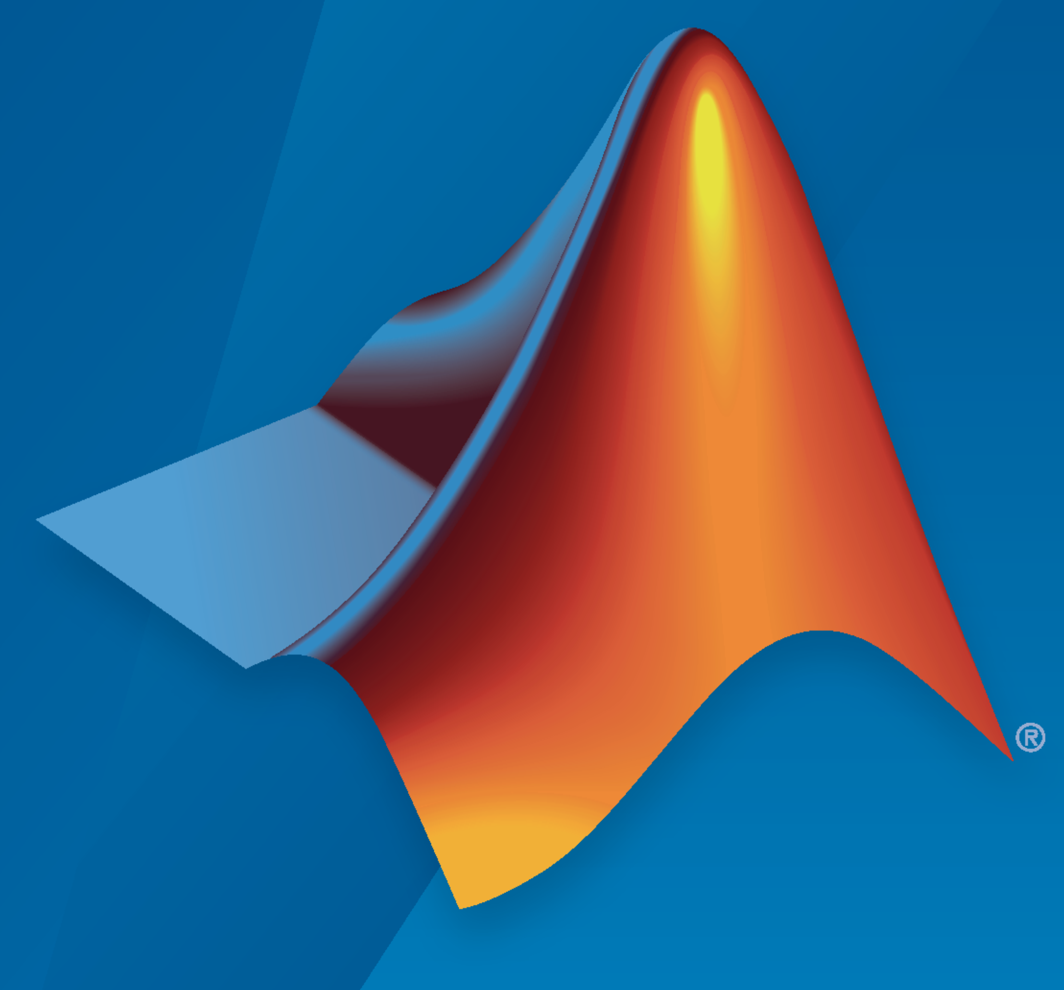
Polyspace® Products for Ada

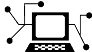
User's Guide



R2021b

MathWorks®

How to Contact MathWorks

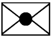


Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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



Phone: 508-647-7000

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

Polyspace® Products for Ada User's Guide

© COPYRIGHT 1999–2021 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 5.2 (Release 2008b)
March 2009	Online Only	Revised for Version 5.3 (Release 2009a)
September 2009	Online Only	Revised for Version 5.4 (Release 2009b)
March 2010	Online Only	Revised for Version 5.5 (Release 2010a)
September 2010	Online Only	Revised for Version 6.0 (Release 2010b)
April 2011	Online Only	Revised for Version 6.1 (Release 2011a)
September 2011	Online Only	Revised for Version 6.2 (Release 2011b)
March 2012	Online Only	Revised for Version 6.3 (Release 2012a)
September 2012	Online Only	Revised for Version 6.4 (Release 2012b)
March 2013	Online Only	Revised for Version 6.5 (Release 2013a)
September 2013	Online Only	Revised for Version 6.6 (Release 2013b)
March 2014	Online Only	Revised for Version 6.7 (Release 2014a)
October 2014	Online Only	Revised for Version 6.8 (Release 2014b)
March 2015	Online Only	Revised for Version 6.9 (Release 2015a)
September 2015	Online Only	Revised for Version 6.10 (Release 2015b)
March 2016	Online Only	Revised for Version 6.11 (Release 2016a)
September 2016	Online Only	Revised for Version 6.12 (Release 2016b)
March 2017	Online Only	Revised for Version 6.13 (Release 2017a)
September 2017	Online Only	Revised for Version 6.14 (Release 2017b)
March 2018	Online Only	Revised for Version 6.15 (Release 2018a)
September 2018	Online Only	Revised for Version 6.16 (Release 2018b)
March 2019	Online Only	Revised for Version 6.17 (Release 2019a)
September 2019	Online Only	Revised for Version 6.18 (Release 2019b)
March 2020	Online Only	Revised for Version 6.19 (Release 2020a)
September 2020	Online Only	Revised for Version 6.20 (Release 2020b)
March 2021	Online Only	Revised for Version 6.21 (Release 2021a)
September 2021	Online Only	Revised for Version 6.22 (Release 2021b)

Contents

1 Introduction to Polyspace Products

Overview of Polyspace Verification **1-2**

The Value of Polyspace Verification **1-3**
Enhance Software Reliability **1-3**
Decrease Development Time **1-3**
Improve the Development Process **1-4**

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

2 How to Use Polyspace Software

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

Implementing a Process for Polyspace Verification **2-4**
Overview of the Polyspace Process **2-4**
Defining Quality Goals **2-4**
Defining a Verification Process to Meet Your Goals **2-6**
Applying Your Verification Process to Assess Code Quality **2-6**
Improving Your Verification Process **2-7**

Sample Workflows for Polyspace Verification **2-8**
Overview of Verification Workflows **2-8**
Software Developers – Standard Development Process **2-8**
Software Developers – Rigorous Development Process **2-10**
Quality Engineers – Code Acceptance Criteria **2-12**
Project Managers — Integrating Polyspace Verification with Configuration Management Tools **2-13**

3 Setting Up a Verification Project

Create Project **3-2**
Create Project **3-2**

Specify Analysis Options . . . 3-3
Specify Results Folder . . . 3-4

Create Project Using Template . . . **3-5**
Use Predefined Template . . . 3-5
Create Your Own Template . . . 3-5

Update Project . . . **3-7**
Add Source and Include Folders . . . 3-7
Manage Include File Sequence . . . 3-8
Change Analysis Options . . . 3-8

Modularize Project . . . **3-10**
Create New Module . . . 3-10
Create Configurations in Module . . . 3-10

Organize Layout of Polyspace User Interface . . . **3-12**
Create Your Own Layout . . . 3-12
Save and Reset Layout . . . 3-12

Customize Results Location and Folder Name . . . **3-14**

Specify External Text Editor . . . **3-15**

Change Default Font Size . . . **3-16**

Choosing Contextual Verification Options . . . **3-17**

4 Emulating Your Run-Time Environment

Target & Compiler Overview . . . **4-2**

Specifying Target & Compiler Parameters . . . **4-3**

Predefined Target Processor Specifications . . . **4-4**

Main Generator Overview . . . **4-5**

Automatically Generating a Main . . . **4-6**

Manually Generating a Main . . . **4-7**

How Polyspace Verifies Generic Packages . . . **4-8**

Specifying Constraints Using Text Files . . . **4-9**
Constraint File Format . . . 4-9
Tips for Creating Constraint Files . . . 4-10
Example Constraint File . . . 4-10
Warning Messages Related to Constraints . . . 4-10

Effect of External Constraints on Polyspace Analysis **4-12**
Stubbed Functions **4-12**
Stubbed Procedures **4-13**

Performing Efficient Module Testing with Constraints **4-15**

Reducing Orange Checks with External Constraints **4-16**

Using Pragma Assert to Set Data Ranges **4-17**

Supported Ada Pragmas **4-18**

How Polyspace Evaluates Function and Procedure Parameters **4-19**

5 Preparing Source Code for Verification

Stubbing Overview **5-2**

Manual vs. Automatic Stubbing **5-3**
Deciding which Stub Functions to Provide **5-3**
Summary **5-4**

Automatic Stubbing **5-6**

Polyspace Software Assumptions **5-7**

Scheduling Model **5-8**
Example **5-8**
Launching Command **5-8**
Limitation **5-8**

Modelling Synchronous Tasks **5-9**
Problem **5-9**
Explanation **5-9**
Solution 1 . . . . **5-9**
Solution 2 . . . . **5-10**

Interruptions and Asynchronous Events/Tasks **5-11**
Problem **5-11**
Explanation **5-11**
My interrupts it1 and it2 cannot preempt each other **5-11**
My interruptions can preempt each other **5-11**

Are Interruptions Maskable or Preemptive by Default? **5-13**
Problem **5-13**
Explanation **5-13**
Solution **5-13**
Original Packages **5-13**
Extra Packages **5-13**
Command Line to Open Polyspace User Interface **5-14**

Mailboxes 5-15
Problem 5-15
Explanation 5-15
Solution 5-15
package mailboxes 5-16
package body mailboxes 5-16
procedure receive 5-16
task body task_1 5-16

Atomicity 5-18
Definitions 5-18
Instructional Decomposition 5-18
Critical Sections and Temporal Exclusion 5-18

Priorities 5-19

6 Running a Verification

Run Local Verification 6-2
Start Verification 6-2
Monitor Progress 6-2
Stop Verification 6-2
Open Results 6-3

Run Remote Verification 6-4
Start Verification 6-4
Monitor Progress 6-4
Stop Verification 6-5
Open Results 6-5

Phases of Verification 6-6

Run File-by-File Local Verification 6-7
Run Verification 6-7
Open Results 6-7

Run File-by-File Remote Verification 6-9
Run Verification 6-9
Open Results 6-9

Manage Job Monitor 6-10
Purge Server Queue 6-10
Change Job Monitor Password 6-10
Share Server Verifications Between Users 6-11

Run Local Verification at Command Line 6-13

Run Remote Verification at Command Line 6-14
Start Verification 6-14
Manage Verification 6-14
Download Verification Results from Server 6-15

Create Command-Line Script from Project File 6-16
Generate Scripting Files 6-16
Run an Analysis 6-16

7 Troubleshooting Verification

Hardware Does Not Meet Requirements 7-2

Location of Included Files Not Specified 7-3

Polyspace Software Cannot Find the Server 7-4

Limit on Assignments and Function Calls 7-6

Examining the Compile Log 7-7

Common Compile Errors 7-8
Missing specification for unit 7-8
Calendar not found 7-8
Not a predefined library unit 7-9
representation clause appears too late 7-9
Package system and standard include 7-10
Unsigned type 7-10
Function not declared in package 7-10
pre-elaborated unit 7-11
actual must be a definite subtype 7-11
'ref attribute 7-12
Cannot load s-dec.ads (unit not found) 7-12
Green Hills standard include 7-13
Package Analysis Limitation 7-13
Ambiguous Bounds in Discrete Range 7-14

Error from Special Characters 7-15
Issue 7-15
Cause 7-15
Workaround 7-15

Verification Time Considerations 7-16

Displaying Verification Status Information 7-17

Ideal Application Size 7-18

Optimum Size 7-19

Selecting a Subset of Code 7-20
Results 7-21
Examples of Removable Components 7-21
Subdivide According to Data Flow 7-21
Subdivide According to Real-Time Characteristics 7-22
Subdivide According to Files 7-23

Benefits of Methods . . . 7-24
When the Application is Incomplete . . . 7-24
Application Code Size . . . 7-24

Obtaining Configuration Information . . . 7-26

Reasons for Unchecked Code . . . 7-27
Issue . . . 7-27
Possible Cause: Early Red or Gray Check . . . 7-28
Possible Cause: Incorrect Options . . . 7-29

Storage of Temporary Files . . . 7-30

Disk Defragmentation and Antivirus Software . . . 7-31

Out-of-Memory Errors During Report Generation . . . 7-32

8 Reviewing Verification Results

Polyspace Check Colors . . . 8-2

Verification Following Red and Orange Checks . . . 8-3
Verification Following Red Check . . . 8-3
Green Check Following Orange Check . . . 8-3
Gray Check Following Orange Check . . . 8-4

Project and Results Folder Contents . . . 8-5
Files in the Results Folder . . . 8-5

Result Views in Polyspace User Interface . . . 8-6
Results List . . . 8-6
Source . . . 8-8
Result Details . . . 8-10
Variable Access . . . 8-11
Call Hierarchy . . . 8-13
Concurrency Modeling . . . 8-15

Why Review Dead Code Checks . . . 8-16
Functional Bugs in Gray Code . . . 8-16
Structural Coverage . . . 8-16

Review Red Checks . . . 8-18
Step 1: Interpret Check Information . . . 8-18
Step 2: Determine Root Cause of Check . . . 8-18

Review Gray Checks . . . 8-20

Review Orange Checks . . . 8-21
Step 1: Interpret Check Information . . . 8-21
Step 2: Determine Root Cause of Check . . . 8-21
Step 3: Trace Check to Polyspace Assumption . . . 8-23

Review Global Variable Usage **8-24**

CWE Coding Standard and Polyspace for Ada Results **8-25**
CWE and Polyspace for Ada **8-25**
Find CWE IDs from Polyspace Results **8-25**

Add Review Comments to Results **8-27**
Assign and Save Comments **8-27**
Import Review Comments from Previous Verifications **8-28**

Justify Results Through Code Annotations **8-30**
Add Annotations from the User Interface **8-30**
Type Annotations Directly in Your Code **8-32**
Syntax Examples **8-34**

Define Custom Annotation Format **8-36**
Define Annotation Syntax Format **8-38**
Map Your Annotation to the Polyspace Annotation Syntax **8-42**

Annotation Description Full XML Template **8-44**
Example **8-46**

Add Review Comments to Code **8-49**
Enter Code Comments in Specific Syntax **8-49**
Copy Comment Syntax from Polyspace User Interface **8-50**

Filter and Group Results **8-52**
Filter Results **8-52**
Group Results **8-53**

Prioritize Check Review **8-54**

Generate Report **8-55**
Specify Report Generation Before Verification **8-55**
Generate Report After Verification **8-56**

Export Results to Text File **8-58**
Export Results **8-58**
View Exported Results **8-58**
Generate Graphs from Results **8-59**

Export Global Variable List **8-60**
Export Variable List to Text File **8-60**
View Exported Variable List **8-61**

Customize Report Templates **8-62**
Create Custom Template **8-62**
Apply Global Filters in Template **8-62**
Override Global Filters **8-63**
Use Custom Template **8-64**

Set Character Encoding Preferences **8-65**

9 Managing Orange Checks

What Is an Orange Check? 9-2

Sources of Orange Checks 9-5
Orange Checks from Code 9-5
Orange Checks from Verification Limitations 9-5

Do I Have Too Many Orange Checks? 9-7

Limit Display of Orange Checks 9-8

Reduce Orange Checks 9-10
Improve Verification Precision 9-10
Apply Coding Guidelines 9-11
Stub Parts of the Code Manually 9-11
Specify Multitasking Behavior 9-14

10 Verifying Code in the Eclipse IDE

Install Polyspace Plug-In for Eclipse IDE 10-2

Configure Verification 10-5
Prerequisites 10-5
Specify Verification Options 10-5
Next Steps 10-5

Run Verification 10-6
Prerequisites 10-6
Start, Monitor and Stop Verification 10-6
Next Steps 10-7

Review Results 10-8
Prerequisites 10-8
Review Results 10-8
Save Multiple Results 10-8

Glossary

Introduction to Polyspace Products

- “Overview of Polyspace Verification” on page 1-2
- “The Value of Polyspace Verification” on page 1-3
- “How Polyspace Verification Works” on page 1-5

Overview of Polyspace Verification

Polyspace products verify C, C++, and Ada code by detecting run-time errors before code is compiled and executed. Polyspace verification uses formal methods not only to detect errors, but to prove mathematically that certain classes of run-time errors do not exist.

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

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

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

The Value of Polyspace Verification

In this section...

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

Enhance Software Reliability

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

Polyspace software can:

- Prove that your code has certain kinds of errors.
- Prove that your code does not have certain kinds of errors.
- Identify unreachable code.
- Identify code that can have an error along some execution paths.

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

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

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

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

Improve the Development Process

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

Polyspace verification software supports code verification throughout the development process:

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

How Polyspace Verification Works

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

What is Static Verification

Static verification is a broad term, and is applicable to any tool 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 erroneous. Polyspace verification provides deep-level verification identifying most run-time errors and possible access conflicts on global shared data.

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

For example, consider the following code:

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

To check that the variable `i` does not overflow the range of `tab`, a traditional approach would be to enumerate each possible value of `i`. One thousand checks would be required.

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

An approximation usually leads to information loss. For instance, the information that `i` is incremented by one every cycle in the loop is lost. However, the important fact is that this information is not required to ensure that a range error will not occur; it is only necessary to prove that the domain variation of `i` is smaller than the range of `tab`. Only one check is required to establish that — and hence the gain in efficiency compared to traditional approaches.

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

Exhaustiveness

Nothing is lost in terms of exhaustiveness. The reason is that Polyspace verification works by performing upper approximations. In other words, the computed variation domain of a program variable is a superset of its actual variation domain. The direct consequence is that a runtime error (RTE) item to be checked cannot be missed by Polyspace verification.

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 2-8

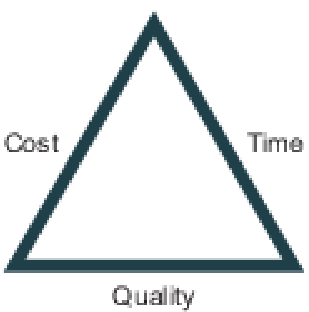
Polyspace Verification and the Software Development Cycle

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

Software Quality and Productivity

The goal of most software development teams is to maximize both quality and productivity. However, when developing software, you must consider the following factors:

- Cost
- Quality
- Time



Changing the requirements for one of these factors can impact 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 the modules in an application until each module meets the required quality level. Unfortunately, this process often ends before quality requirements are met, because the available time or budget has been exhausted.

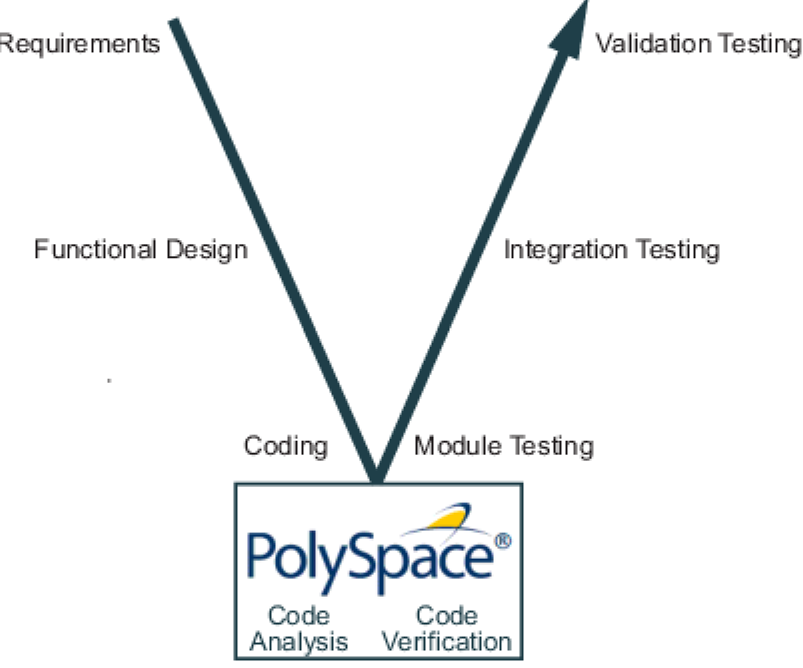
Polyspace verification allows a different process. Polyspace verification can support both productivity improvement and quality improvement at the same time, although you have to reach a balance between these goals.

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 obvious defects. Once the code is stable, you verify it again before module/unit testing, with more stringent verification and review criteria.

Using verification at this stage of 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 developer is familiar with their own code, and can quickly determine why the code contains defects. In addition, defects are cheaper to fix at this stage as 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 Goals" on page 2-4
"Defining a Verification Process to Meet Your Goals" on page 2-6
"Applying Your Verification Process to Assess Code Quality" on page 2-6
"Improving Your Verification Process" on page 2-7

Overview of the Polyspace Process

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

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

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

Defining Quality Goals

Before you can verify whether your code meets your quality goals, you must define those goals. This process involves:

- "Choosing Robustness or Contextual Verification" on page 2-4
- "Defining Software Quality Levels" on page 2-5

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

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

Robustness Verification

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

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

- Assumes function inputs are full range
- Initializes global variables to full range
- Automatically stubs missing functions

While this approach ensures that the software works under all verified conditions, it can lead to orange checks (unproven code) in your results. You must then manually inspect these orange checks in accordance with your software quality goals.

Contextual Verification

Contextual verification proves that the software 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 “Inputs & Stubbing”.
- Create a detailed main program to model the call sequence, instead of using the default main generator. For more information, see “Manually Generating a Main” on page 4-7.
- Provide manual stubs that emulate the behavior of missing functions, instead of using the default automatic stubs. For more information, see “Manual vs. Automatic Stubbing” on page 5-3.

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 for your application, from level SQL-1 (lowest) to level SQL-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			
	SQL1	SQL2	SQL3	SQL4
Document static information	X	X	X	X
Review all red checks	X	X	X	X
Review all gray checks	X	X	X	X
Review first criteria level for orange checks		X	X	X
Review second criteria level for orange checks			X	X
Perform dataflow analysis			X	X
Review third criteria level for orange checks				X

In the example above, the quality criteria include:

- **Static Information** – Includes information about the application architecture, the structure of each module and file. Full verification of your application requires the documentation of static information.
- **Red checks** – Represent errors that occur every time the code is executed.
- **Gray checks** – Represent unreachable code.
- **Orange checks** – Indicate unproven code, meaning a run-time error may occur. .
- **Dataflow analysis** – Identifies errors such as non-initialized variables and variables that are written but not subsequently 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 Goals

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

These actions include:

- Setting standards for code development, such as coding rules.
- Setting Polyspace Analysis options to match your quality goals. See “Specify Analysis Options” on page 3-3.
- Setting review criteria in the Polyspace user interface so that results are reviewed consistently. See “Review Results”.

Applying Your Verification Process to Assess Code Quality

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

This process includes:

1. Running a Polyspace verification for each software component as it is written.
2. Reviewing verification results consistently. See “Results Management”.
3. Saving review comments for each component, so they are available for future review. See “Add Review Comments to Results” on page 8-27.
4. Performing additional verifications on each component, as defined by your quality goals.

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

For more information, see “Reduce Orange Checks” on page 9-10.

Sample Workflows for Polyspace Verification

In this section...

"Overview of Verification Workflows" on page 2-8
"Software Developers – Standard Development Process" on page 2-8
"Software Developers – Rigorous Development Process" on page 2-10
"Quality Engineers – Code Acceptance Criteria" on page 2-12
"Project Managers — Integrating Polyspace Verification with Configuration Management Tools" on page 2-13

Overview of Verification Workflows

Polyspace verification supports two goals at the same time:

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

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

User Description

This workflow applies to software developers using a standard development process. Before implementing Polyspace verification, these users fit the following criteria:

- In Ada, unit test tools or coverage tools are not used – functional tests are performed just after coding.
- In C, either coding rules are not used, or rules are not followed consistently.

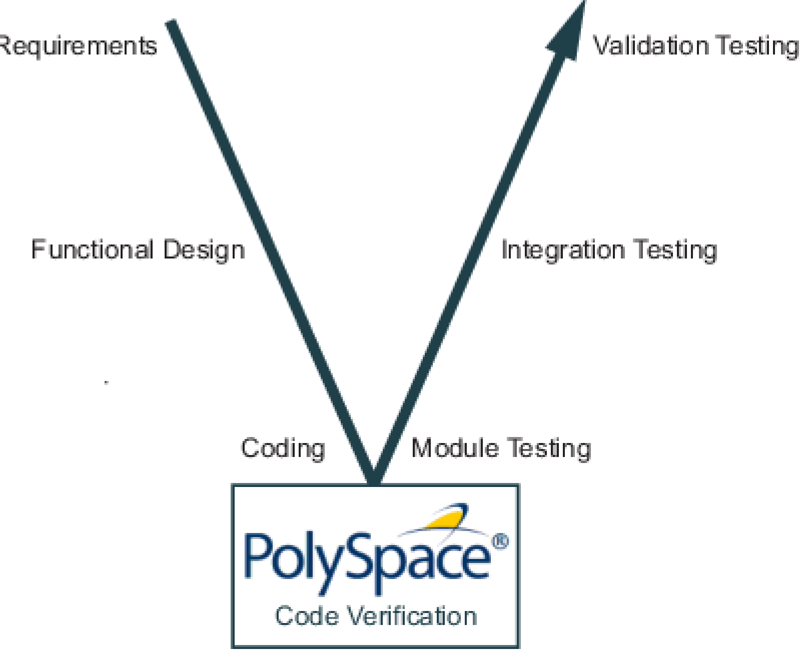
Quality

The main goal of Polyspace verification is to improve productivity while maintaining or improving software quality. Verification helps developers find and fix bugs more quickly than other processes. It also improves software quality by identifying bugs that otherwise might remain in the software.

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

Verification Workflow

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



The verification workflow consists of the following steps:

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

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

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

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

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

Costs and Benefits

When using verification to detect bugs:

- **Red and gray checks** — The number of bugs found in red and gray checks varies, but approximately 40% of verifications reveal one or more red errors or bugs in gray code.

- **Orange checks** — 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:

- **Setup time —** the time required to set up your verification will be higher if you do not use coding rules. You may have to make modifications to the code before starting the verification.

Software Developers – Rigorous Development Process

User Description

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

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

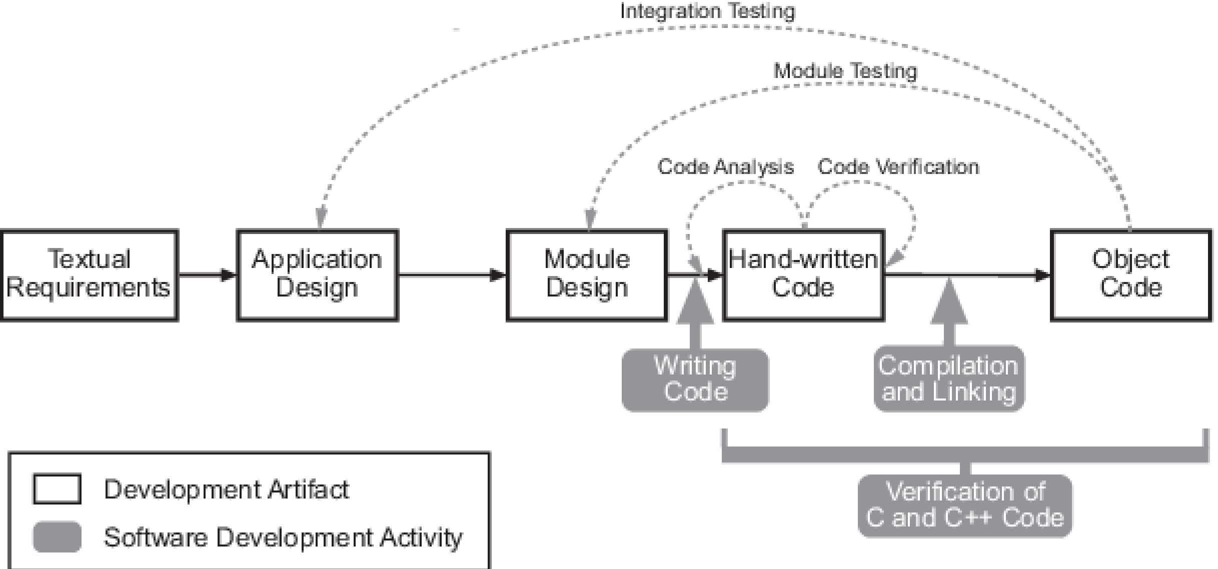
Quality

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

Verification can prove the absence of run-time errors, while helping developers find and fix 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:
   - Creates a “main” program to model call sequence, instead of using the automatically generated main.
   - Sets options to check the properties of some output variables. For example, if a variable `y` is returned by a function in the file and should always be returned with a value in the range 1 to 100, then Polyspace can flag instances where that range of values might be breached.
2. The project leader configures the project to check the required coding rules.
3. Each developer performs file-by-file verification as they write code, and reviews both coding rule violations and verification results.
4. The developer fixes coding rule violations, fixes **red** errors, examines **gray** code, and performs a selective orange review.
5. Until coding is complete, the developer repeats steps 2 and 3 as required.
6. Once a developer considers a file complete, they perform a final verification.
7. The developer 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:

- 3–5 orange and 3 gray checks per file, yielding an average of 1 bug. Often, 2 of the orange checks represent the same bug, and another represent an anomaly.
- Typically, each file requires two verifications before it can be checked-in to the configuration management system.
- The average verification time is about 15 minutes.

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

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

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

Quality Engineers – Code Acceptance Criteria

User Description

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

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

Quality

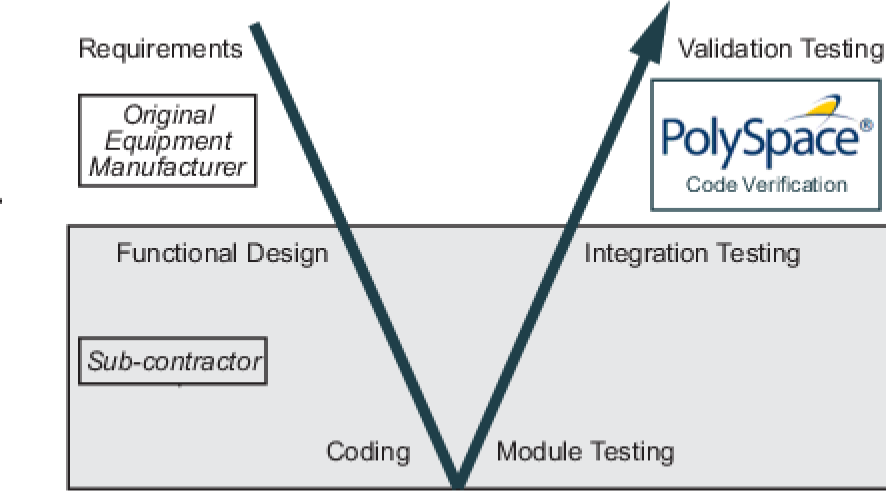
The main goal of Polyspace verification is to control and evaluate the safety of an application.

The criteria used to evaluate code can vary widely depending on the criticality of the application, from absence of red errors only 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 2-5.

Verification Workflow

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



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

The verification workflow consists of the following steps:

1. Quality engineering group defines clear quality goals for the code to be written, including specific quality levels for each version of the code to be delivered (first, intermediate, or final delivery) For more information, see “Defining Quality Goals” on page 2-4.
2. Development group writes code according to established standards.
3. Development group delivers software to the quality engineering group.
4. The project leader configures the Polyspace project to meet the defined quality goals, as described in “Defining a Verification Process to Meet Your Goals” on page 2-6.
5. Quality engineers perform verification on the code.
6. Quality engineers review **red** errors, **gray** code, and the number of orange checks defined in the process.

   Note The number of orange checks reviewed often depends on the version of software being tested (first, intermediate, or final delivery). This can be defined by quality level (see “Defining Software Quality Levels” on page 2-5).
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 fixing faults is higher, because verification takes place late in the development cycle.

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

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

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

Project Managers — Integrating Polyspace Verification with Configuration Management Tools

User Description

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

Quality

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

Verification Workflow

The verification workflow consists of the following steps:

1. Project manager defines quality goals, including individual quality levels for each stage of the development cycle.
2. Project leader configures a Polyspace project to meet quality goals.
3. Developers run verification at the following stages:

   - **Daily check-in** — On the files currently under development. Compilation must complete without the permissive option.
   - **Pre-unit test check-in** — On the files currently under development.
   - **Pre-integration test check-in** — On the whole project, ensuring that compilation can complete without the permissive option. This stage differs from daily check-in because link errors are highlighted.
   - **Pre-build for integration test check-in** — On the whole project, with multitasking aspects accounted for as required.
   - **Pre-peer review check-in** — On the whole project, with multitasking aspects accounted for as required.
4. Developers review verification results for each check-in activity to confirm that the code meets the required quality level. For example, the transition criterion could be: “No bug found within 20 minutes of selective orange review”

Setting Up a Verification Project

- "Create Project" on page 3-2
- "Create Project Using Template" on page 3-5
- "Update Project" on page 3-7
- "Modularize Project" on page 3-10
- "Organize Layout of Polyspace User Interface" on page 3-12
- "Customize Results Location and Folder Name" on page 3-14
- "Specify External Text Editor" on page 3-15
- "Change Default Font Size" on page 3-16
- "Choosing Contextual Verification Options" on page 3-17

Create Project

To create a project manually, you must know:

- Location of your source files
- Location of your include files

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

Create Project

This example shows how to create a new project.

1. Select **File** > **New Project**.
2. In the Project – Properties window, specify properties for your project:
   - **Project name**
   - **Location**: Folder where you will store the project file (`.psprj` file) and the results unless you specify otherwise. You can use the `.psprj` file to reopen the project.

     The software assigns a default location to your project called your Polyspace Workspace. You can change this default in the Polyspace Preferences on the **Project and Results Folder** tab.
   - Clear the **Use template** check box unless you have a template you want to use.
3. On the next screen, add source folders to your project.
   a. Use the **Browse** button to navigate to the folder containing the source files you want to analyze.

      By default, Polyspace looks for `.c`, `.cpp`, `.cxx`, or `.cc` files. If you use other file extensions, before closing the dialog box, change the **Files of types** option.
   b. If you chose a source folder with subfolders but do not want to analyze files in the subfolders, clear the check box **Add recursively**.
   c. (Linux® only) Often, compilers add symbolic links in your source folders during compilation. If your folder contains symbolic links to other folders but you do not want to add source files from the other folders, select **Exclude symbolic links**.
   d. Click **Add Source Folder**. All source files found under this folder are added to your Polyspace project.

      Tip To see the full path of your files, toggle the button.
   e. If you do not want to analyze all the files under your source folder, right-click the file or folder and select **Exclude Files**. The file appears with an symbol in your project indicating it is not considered for analysis. You can reinclude the files for analysis by right-clicking and selecting **Include Files**.
4. On the next screen, add include folders to your project. The analysis looks for include files relative to the folder paths that you specify. For instance, if your code contains the preprocessor directive `#include<../mylib.h>` and you include the folder:

`C:\My_Project\MySourceFiles\Includes`

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

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

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

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

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

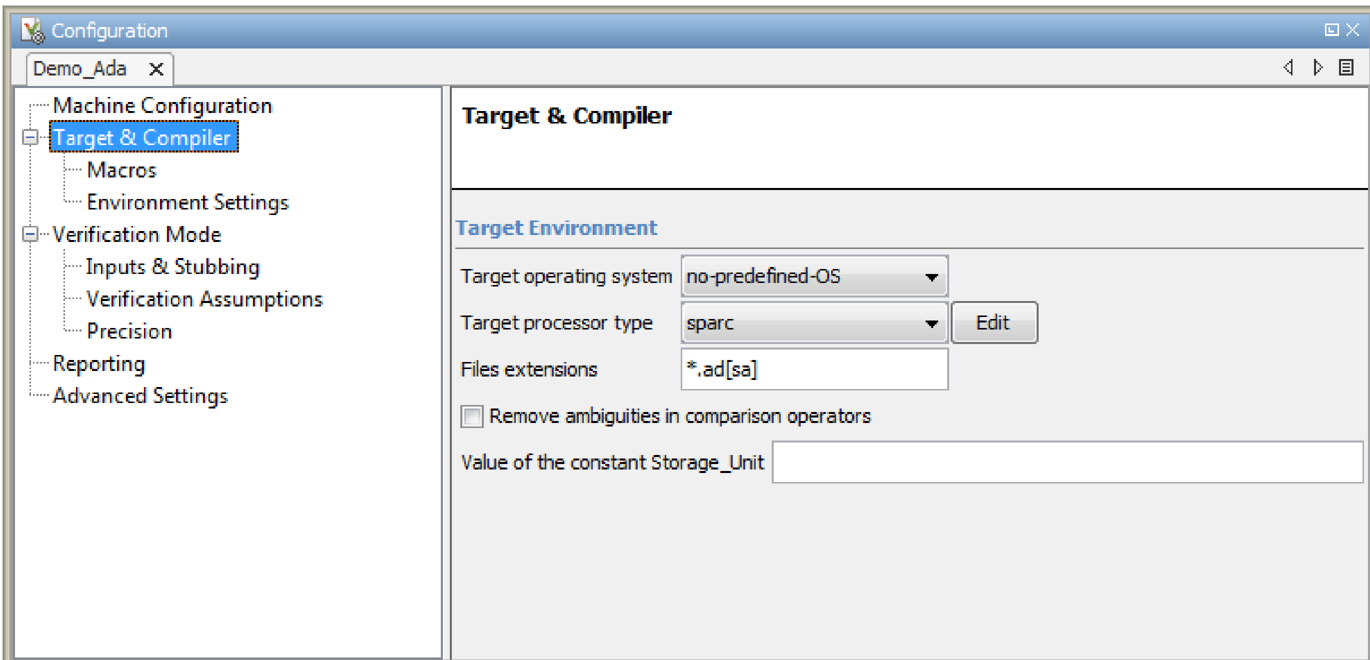
d Click **Add Include Folders**. The include folder is added to your Polyspace project.

Specify Analysis Options

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

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

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



For instance:

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

You can also create another configuration in your module. For more information, see "Create Configurations in Module" on page 3-10.

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

Specify Results Folder

This example shows how to specify a results folder. In the **Project Browser** pane, the folder appears as a node under the **Result** node of your project. By default, the software creates a new results folder for each analysis. Before starting an analysis, you can choose to overwrite an existing results folder. For example, if you stopped an analysis before completion and want to restart it, you can overwrite a results folder.

- To create a new folder for every run, on the **Project Browser** pane, select **Create new result folder**.
  - By default, the new folder is created in `Project_folder` / `Module_name`. `Project_folder` is the project location you specified when creating a new project.
  - You can also create a parent folder for storing your results. Select **Tools > Preferences** and enter the parent folder location on the **Project and Results Folder** tab. If you enter a parent folder location, any new result folder will be created under this parent folder.
- To overwrite an existing folder that is open in the **Project Browser** pane, clear **Create new result folder**. Before running verification, select the result that you want to overwrite.

Create Project Using Template

A **Project Template** is a predefined set of analysis options for a specific compilation environment. When creating a new project, you can do one of the following:

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

  Polyspace provides predefined templates for common compilers such as `Aonix`, `Rational`, and `Greenhills`. For additional templates, see Polyspace Compiler Templates .
- Set analysis options manually. You can then save your options as a template and reuse them later.

Use Predefined Template

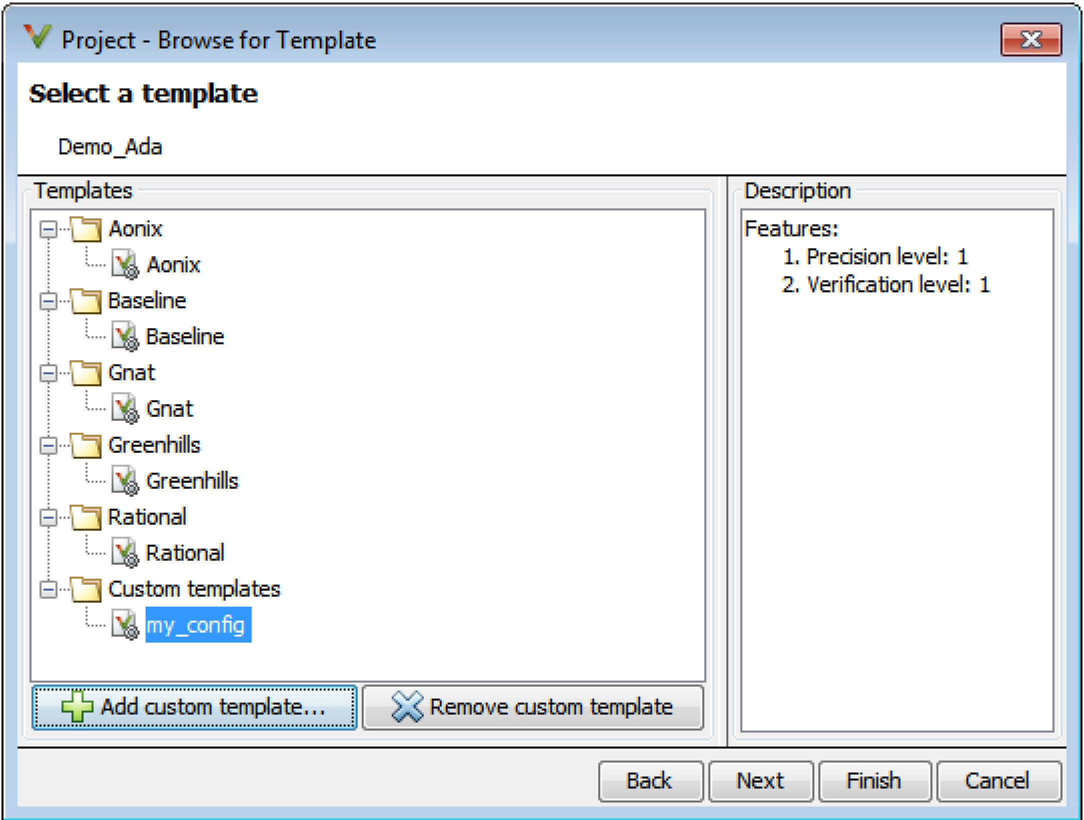
1. Select **File > New Project**.
2. On the Project – Properties dialog box, after specifying the project name and location, under **Project configuration**, select **Use template**.
3. On the next screen, select the template that corresponds to your compiler. For further details on a template, select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

   If your compiler does not appear in the list of predefined templates, select **Baseline**.
4. On the next screen, add your source files and include folders. For more information, see “Create Project” on page 3-2.

Create Your Own Template

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

- To create a template from a project that is open on the **Project Browser** pane:
  1. Right-click the project configuration that you want to use, and then select **Save As Template**.
  2. Enter a description for the template, then click **Proceed**. Save your template file.
- When you create a new project, to use a saved template:
  1. Select Add custom template... .
  2. Navigate to the template that you saved earlier, and then click **Open**. The new template appears in the **Custom templates** folder on the **Templates** browser. Select the template for use.


Project - Browse for Template
Select a template
Demo_Ada
Templates
Aonix
Aonix
Baseline
Baseline
Gnat
Gnat
Greenhills
Greenhills
Rational
Rational
Custom templates
my_config
Add custom template...
Remove custom template
Description
Features:
1. Precision level: 1
2. Verification level: 1
Back
Next
Finish
Cancel

Update Project

You can manually add source files and include folders to an existing project, or change the analysis options.

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

Add Source and Include Folders

If you want to change which files or folders are active in your project without removing them from your project tree:

1. Right-click the file or folder and select **Exclude Files**.

   The file appears with an symbol in your project indicating it is not considered for analysis. You can reinclude the files for analysis by right-clicking and selecting **Include Files**.

If you want to add additional source folders or include folders, follow these steps:

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

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

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

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

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

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

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

   ---

   e. If you do not want to analyze all the files under your source folder, right-click the file or folder and select **Exclude Files**. The file appears with an symbol in your project indicating it is not considered for analysis. You can reinclude the files for analysis by right-clicking and selecting **Include Files**.

   Repeat these steps as many times as necessary, then click **Next**.
4. Add include folders to your project.

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

      By default, Polyspace looks for `.h`, `.hpp`, or `.hxx` files. If you use other file extensions, before closing the dialog box, change the **Files of types** option.
   b. If you chose an include folder that contains subfolder and you want to add those include folders as well, select the check box **Include all subfolders**.
   c. (Linux only) Often, compilers add symbolic links in your folders during compilation. If your folder contains symbolic links to other folders but you do not want to add includes from the other folders, select **Exclude symbolic links**.
   d. Click **Add Include Folders**. The include folder is added to your Polyspace project.

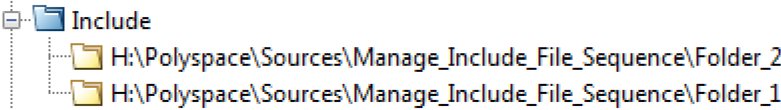
   Repeat these steps as many times as necessary, then click **Finish**. The new project opens in the **Project Browser** pane.
5. Click **Finish**.
6. Before running an analysis, you must copy the source files to a module.
   a. Select the source files that you want to copy. To select multiple files together, press the **Ctrl** key while selecting the files.
   b. Right-click your selection.
   c. Select **Copy to** > **Module_*n***. *n* is the module number.

Manage Include File Sequence

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

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

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



To change the order of include folders:

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

Change Analysis Options

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

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

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

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

  For more information, see “Verification Mode”.
- To allow Polyspace to prove more operations and therefore produce fewer non-critical orange checks, you can specify appropriate options.

  For more information, see “Reduce Orange Checks” on page 9-10.

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

Modularize Project

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

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

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

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

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

In this section...
"Create New Module" on page 3-10
"Create Configurations in Module" on page 3-10

Create New Module

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

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

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

   You see a new module, **Module_2**, in your project. To rename the module, right-click the module name.
2. In your project, below the **Source** node, right-click the files that you want to add to the module. From the context menu, select **Copy to** > **Module_2**.

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

Create Configurations in Module

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

- Change the analysis options in this configuration.

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

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

To create a new configuration in your module:

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

   - On the **Project Browser** pane, the software displays a new configuration *`project_name`*`_1`. To rename the configuration, right-click the configuration and select **Rename Configuration**.
   - On the **Configuration** pane, the new configuration appears as an additional tab.
2. On the **Configuration** pane, specify the analysis options for the new configuration.
3. To use this new configuration, double-click it.

   When you run a new verification on the module, it uses the analysis options in this configuration.
4. To see the configuration you used for a certain result, right-click the result on the **Project Browser**. Select **Open Configuration**.

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

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

Organize Layout of Polyspace User Interface

The Polyspace user interface has two default layouts of panes.

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

Project Browser	**Configuration**
	Output Summary

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

Results List	**Result Details**
	Dashboard

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

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

Create Your Own Layout

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

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

To move a pane to another location:

1. Float the pane in one of three ways:

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

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

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

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

Save and Reset Layout

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

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

Customize Results Location and Folder Name

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

1. Select **Tools** > **Preferences**.
2. On the **Project and Results Folder** tab, select **Create new result folder**.
3. In the **Parent results folder location** field, specify the location that you want.
4. If you require a subfolder, select the **Add a subfolder using the project name** check box. This subfolder takes the name of the project.
5. If required, specify additional formatting options for the folder name . The options allow you to incorporate the following information into the name of the results folder:

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

For each verification, the software now creates a new results folder *`ResultFolderPrefix_ProjectVariable_DateFormat_TimeFormat_Counter`*.

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

Specify External Text Editor

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

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

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

   Polyspace has already specified the command-line arguments for the following editors:

   - `Emacs`
   - `Notepad++` — Windows® only
   - `UltraEdit`
   - `VisualStudio`
   - `WordPad` — Windows only
   - `gVim`

   If you are using one of these editors, select it from the **Arguments** drop-down list.

   If you are using another text editor, select `Custom` from the drop-down list, and enter the command-line options in the field provided.

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

   a. In the **Text Editor** field, enter `/usr/bin/xterm`.
   b. From the **Arguments** drop-down list, select `Custom`.
   c. In the field to the right, enter `-e /usr/bin/vi $FILE`.
6. To revert back to the built-in editor, on the **Editors** tab, from the **Text editor** drop-down list, select **Built In**.

Change Default Font Size

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

1 Select **Tools** > **Preferences**.
2 On the **Miscellaneous** tab:

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

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

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

Choosing Contextual Verification Options

While creating your project, you must configure analysis options to match your quality goals. 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 “Defining Quality Goals” on page 2-4.

Note If you are aware of run-time errors in your code but still want to run a verification, you can annotate your code so that these known errors are highlighted in the Polyspace user interface. For more information, see “Add Review Comments to Code” on page 8-49.

---

1. On the **Configuration** pane, select **Verification Mode**. Select **Verify module**.
2. On the **Configuration** pane, select **Inputs & Stubbing**. In the **Variable/function range setup** field, specify a data range specification (DRS) file.
3. Specify how uninitialized global variables are initialized with `Initialization of uninitialized global variables`.

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

Emulating Your Run-Time Environment

- “Target & Compiler Overview” on page 4-2
- “Specifying Target & Compiler Parameters” on page 4-3
- “Predefined Target Processor Specifications” on page 4-4
- “Main Generator Overview” on page 4-5
- “Automatically Generating a Main” on page 4-6
- “Manually Generating a Main” on page 4-7
- “How Polyspace Verifies Generic Packages” on page 4-8
- “Specifying Constraints Using Text Files” on page 4-9
- “Effect of External Constraints on Polyspace Analysis” on page 4-12
- “Performing Efficient Module Testing with Constraints” on page 4-15
- “Reducing Orange Checks with External Constraints” on page 4-16
- “Using Pragma Assert to Set Data Ranges” on page 4-17
- “Supported Ada Pragmas” on page 4-18
- “How Polyspace Evaluates Function and Procedure Parameters” on page 4-19

Target & Compiler Overview

Many applications 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 influence whether errors (such as overflows) occur.

Some run-time errors are dependent on the target CPU and operating system. Therefore, before running a verification, you must specify the type of CPU and operating system for the target environment.

See Also

Related Examples

- "Specifying Target & Compiler Parameters" on page 4-3

Specifying Target & Compiler Parameters

To specify the target environment and compiler behavior for your application, in the Polyspace user interface, on the **Configuration** pane, select **Target & Compiler**.

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

1. For **Target operating system**, select the operating system on which your application is designed to run.
2. For **Target processor type**, select the processor on which your application is designed to run.

   For detailed specifications of each predefined target processor, see "Predefined Target Processor Specifications" on page 4-4.

See Also

More About

- "Target & Compiler Overview" on page 4-2

Predefined Target Processor Specifications

Polyspace software supports many processors. To specify a predefined processor:

1. On the **Configuration** pane, select **Target & Compiler**.
2. For **Target processor type**, select your processor.
3. If your processor is not specified in the drop-down list, use the following table to select a processor that shares the same characteristics as your processor.

Target	sparc	m68kCold Fire	1750a	powerpc32 bit	powerpc64 bit	i386
Character	8	8	16	8	8	8
short_integer	16	16	16	16	16	16
Integer	32	32	16	32	32	32
long_integer	32	32	32	32	64	32
long_long_integer	64	64	64	64	64	64
short_float	32	32	32	32	32	32
Float	32	32	32	32	32	32
long_float	64	64	48	64	64	64
long_long_float	64	64	48	64	64	64

In the following list, the largest default alignment of basic types within record/array for various targets is given:

- `powerpc32bits` — 64.
- `powerpc64bits` — 64.
- `i386` — 32.

4. To identify target processor characteristics, compile and run the following program. If none of the characteristics described in the preceding table match, contact MathWorks® technical support for advice.

```
with TEXT_IO;
procedure TEMP is
type T_
Ptr is access integer;
Ptr  :T_Ptr;
begin
TEXT_IO.PUT_LINE ( Integer'Image (Character'Size) );
TEXT_IO.PUT_LINE ( Integer'Image (Short_Integer'Size) );
TEXT_IO.PUT_LINE ( Integer'Image (Integer'Size));
TEXT_IO.PUT_LINE ( Integer'Image (Long_Integer'Size) );
-- TEXT _IO.PUT_LINE ( Integer'Image( Long_Long_Integer'Size) );
TEXT_IO.PUT_LINE ( Integer'Image (Float'Size) );
--  TEXT _IO.PUT_LINE ( Integer'Image(D_Float'Size) );
TEXT_IO.PUT_LINE ( Integer'Image (Long_Float'Size));
TEXT_IO.PUT_LINE ( Integer'Image (Long_Long_Float'Size) );
TEXT_IO.PUT_LINE( Integer'Image (T_Ptr'Size) );
end TEMP;
```

Main Generator Overview

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

When you run a verification on Polyspace Client™ for Ada software, the main is generated. When you run a verification on Polyspace Server™ for Ada software, you can choose to generate a main automatically.

Automatically Generating a Main

You can choose to automatically generate a main by selecting the **Verify module** (`-main-generator`) option. The `-main-generator` option automatically creates a procedure that calls every uncalled procedure within the code.

With Polyspace Client for Ada software, the software, by default, automatically generates a main. You can choose to manually generate a main using the `-main` option:

1. On the **Configuration** pane, select **Verification Mode**.
2. Select **Verify whole application**.
3. In the **Main entry point** field, the package that defines the main, for example, `INIT.MAIN`.

With Polyspace Server for Ada, the software sets the `-main` option by default. You can choose to automatically generate a main using the `-main-generator` option.

1. On the **Configuration** pane, select **Verification Mode**.
2. Select **Verify module**.

For more information on the main generator, see `Verify module`.

Manually Generating a Main

You might prefer to manually generate a main because it allows you to provide a more accurate model of the calling sequence to be generated.

To manually define the main:

1. Identify the API functions and extract their declaration.
2. Create a main containing declarations of a volatile variable for each type that is listed 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 that you created.

The following code shows the five steps:

```
-- Step 1: API function declarations
function func1(x in integer) return integer;
procedure func2(x in out float, y in integer);

-- Step 2: Create main with declarations of volatile variables
procedure main is
    a,b,c,d: integer;
    e,f: float;
pragma volatile (a);
pragma volatile (e);
begin

  --Step 3: Create loop
  loop
    f:=e;
    c:=a;
    d:=a;
    -- Steps 4 and 5
    if (a = 1) then b:= func1(c); end if;
    if (a = 1) then func2(e,d); end if;
  end loop
end main;
```

How Polyspace Verifies Generic Packages

Consider the following code, which instantiates a generic package.

```
with Ada.Numerics.Generic_Elementary_Functions;

Package Body Test is
   Pi : Constant := 3.141592;
   Buf_Length : constant  := 500;
   type Buffer_type is array(1 .. Buf_Length) of Float;
   Tab   : Buffer_type;

   -- Create instance of generic package
   package Trig is new Ada.Numerics.Generic_Elementary_Functions(float);

Procedure Main is
begin
  for i in Tab'First .. Tab'Last loop
    Tab(i) := float(1.0 - Trig.cos(2.0 * Pi * float(i - 1) / 1000.0));
  end loop;
end Main;

end Test;
```

Polyspace can only analyze packages that are explicitly instantiated. In the code, `Trig` represents a new instantiation of the generic package `Ada.Numerics.Generic_Elementary_Functions(float)`. If you specify the **Verify module** (`-main-generator`) option, Polyspace verifies the functions called by your code. In this case, Polyspace verifies only the function `cos` from the new package.

Specifying Constraints Using Text Files

By default, Polyspace software performs *robustness verification*, proving that the software does not generate run-time errors for all verification conditions. Robustness verification assumes that the data inputs are set to their full range. Therefore, most operations 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. You can set xternal constraints on data ranges, and verify the code within these ranges. This process can substantially reduce the number of orange checks in the verification results.

To specify external constraints, you must specify a file that constrains the range of values for global variables, values returned by stubbed functions, `out` or `in/out` parameters of stubbed procedures, or input parameters of user subprograms called by the main generator during verification. See “Constraint File Format” on page 4-9.

To configure a verification that applies the data range specifications in this text file:

1. In the Polyspace user interface, on the **Configuration** pane, select **Inputs & Stubbing**.
2. To the right of the **Constraint setup** row, click . The **Load a constraint file** dialog box opens.
3. Use this dialog box to navigate to the folder that contains your text file with constraints.
4. In the **File name** field, specify your constraint file.
5. Click **Open**. You see the file path in the **Constraint setup** field.
6. Select **File** > **Save** to save your project settings, including the text file location.

Constraint File Format

The constraint file contains a list of variables, functions, and parameter names together with associated data ranges. During verification, the point at which the range is applied, for example, to a variable, is controlled by the mode keyword: `reinit`, `init`, or `permanent`.

Each line of the file must have the following format:

var_func_param min_val max_val `<reinit|init|permanent>`

- *var_func_param* — A variable name, the name of a function that returns a value, or a subprogram parameter name.
- *min_val*, *max_val* — Constants that specify minimum and maximum range values. Data type of these values can be character, enumerator, integer, or float. The integer or float values may be binary, octal, decimal, or hexadecimal.
- `reinit` — Sets global variables to the specified range at the entry point for each subprogram called by the main generator, or the entry point for the user-defined main subprogram.
- `init` — Initializes subprogram input parameters to a specified range when the subprogram is called by the main generator.
- `permanent` — Sets the return, `out`, or `in/out` parameters to the specified range of a stubbed subprogram each time the subprogram is called.

Tips for Creating Constraint Files

- You can replace *min_val* and *max_val* by the words "min" or "max". In this case, the software uses the corresponding minimum and maximum value for the declared data subtype (true even for an enumeration type that has enumerated values min and max). For example, with a SPARC® processor, the minimum value for the integer data type is -2^31 and the maximum value is 2^31-1.
- You can use tab, comma, space, or semicolon as column separators.
- You can apply data range specification to variables and subprograms declared within a package specification or body, or subprograms outside a package. For subprograms outside a package, use the subprogram name as package name.
- You cannot apply data range specification to:
  - Local subprograms or task entries
  - Constant qualified variables, record discriminants, variables of access type, or variables defined in a protected type or task type

Example Constraint File

The following lines:

```
P.x 2#0001#E2 100 reinit    # x is (re)initialized between [4;100]
P.y min max reinit            # y is initialized with the full range.
P.s1.c 'a' max reinit        # s1.x is initialized between ['a';Character'Last]
P.bar -1.0 1.0 permanent    # stubbed function bar returns [-1.0;1.0]
P.bar1.outp -1.0 1.0 permanent      # stubbed procedure bar1's parameter
                                 # outp returns [-1.0;1.0]
P.proc.i -1.0 1.0 init        # main generator calls the user
                                            # procedure proc with the parameter
                                            # i initialized to [-1.0;1.0]
dummy_f.dummy_f -10 10 permanent    # stubbed free function dummy_f
                                                        # returns [-10;10].
```

are data range specifications for a scenario where:

- x and y are two global variables declared in the package P
- s1 is a variable of record type that has a character type component c
- bar is the name of a stubbed function
- bar1 is a stubbed procedure with outp as out parameter
- proc is a procedure defined with a parameter named i
- dummy_f is a function declared without a parent package

Warning Messages Related to Constraints

Polyspace produces a DRS warning message in the verification log file in the following situations.

- When a data range constraint is applied:

  ```
  Warning:  <symbol> has a range specified by DRS
  in [<min> .. <max>] (<mode>).
  ```

- If the constraint file contains a syntax error, Polyspace produces one of the following types of messages:

- *<DRS_file>*, *line <line#>*: Warning: data range specification with incorrect min that is greater than max
- *<DRS_file>*, *line <line#>*: Warning: data range specification with incorrect min or max type. <[Integer|Float|Enum]> value is expected
- *<DRS_file>*, *line <line#>*: Warning: data range specification with incorrect mode
- *<DRS_file>*, *line <line#>*: Warning: data range specification with incorrect [max|min] value
- *<DRS_file>*, *line <line#>*: Warning: data range specification with [min|min] out of range of ada type

- If the constraint file contains an unsupported data range specification, Polyspace produces one of the following types of messages:

  - *<DRS_file>*, *line <line#>*: Warning: data range specification with unsupported object type | DRS cannot be applied to constant variable, record discriminant or variant, access type, protected type, task entry and local subprogram
  - *<DRS_file>*, *line <line#>*: Warning: data range specification with unsupported variable scope | Variable must be defined within a package specification or body

See Also

More About

- “Effect of External Constraints on Polyspace Analysis” on page 4-12

Effect of External Constraints on Polyspace Analysis

Using external constraints, you can narrow down the assumptions that Polyspace makes about global variables, input arguments and return values of undefined functions. This topic shows how constraints apply to a Polyspace analysis.

Stubbed Functions

If a function body is not present, Polyspace uses a function stub for the analysis. The analysis assumes that the function stub can return any value within the range allowed by the data type of the return value. You can narrow down this assumption with external constraints.

Code Example

Analyze this code by specifying `custom_main` as the main entry point. See `Main entry point`.

```
with System;
use System;

package my_package is

function fun(p: Integer; q : Integer) return Integer;
function stub_fun (p : Integer; q : Integer) return Integer;

end my_package;

package body my_package is

function fun(p: Integer; q : Integer) return Integer is
begin
 return p + q;
end fun;

end my_package;


with my_package;

procedure custom_main is
i : Integer;
j : Integer;
begin
    i := my_package.fun(10,20);
    j := my_package.stub_fun(10, 20);
end;
```

If you place your cursor on the assignments to `i` and `j`, the tooltips show this:

- `i` has the value 30 from the function `fun`.
- `j` can have any value allowed for an integer. The function `stub_fun` is not defined, forcing Polyspace to make this assumption.

Constraints

Specify this constraint in a text file:

```
# function stub constraints
my_package.stub_fun  -10 10 permanent
```

Use the option `Constraint setup` to specify this text file. After analysis, you see from the tooltips that the variable `j` is confined to the range [-10,10].

Instead of using constrained Polyspace stubs, you can write your own stubs. If you write your own stubs, you can implement constraints at a more granular level. For instance, this stub for `stub_fun` constrains the return value to the range [-10,10], *excluding zero*.

```
function stub_fun(p: Integer; q : Integer) return Integer is
tmp: Integer;
random: Integer;
pragma volatile (random);
begin
  tmp := random;
  pragma assert (((tmp >= 10) and (tmp < 0)) or ((tmp > 0) and (tmp <= 10)));
end fun;
```

Along with fundamental data types, you can also constrain structured types. For instance, if a stubbed function `stub_fun` returns a variable of this type:

```
type Record_Type is
record
 a : Int_32;
 b : Float_64;
 c : CHAR;
end record;
```

You can constrain the return value with this set of constraints:

```
# function stub constraints
my_package.stub_fun.a -10 10 permanent
my_package.stub_fun.b -20.0 20.0 permanent
my_package.stub_fun.c 'A' 'B' permanent
```

Stubbed Procedures

Stubbed procedures involve the same assumptions as stubbed functions. Instead of a function return value, the analysis makes assumptions about `out` and `in out` parameters of the procedure. The analysis assumes that following the procedure call, these parameters can have any value allowed by their data types.

Code Example

Analyze this code with Polyspace.

```
with System;
use System;

package my_package is
procedure stub_proc (val :in Integer; res : out Integer);
end my_package;

package body my_package is
end my_package;


with my_package;

procedure custom_main is
i : Integer;
j : Integer;
begin
    i := 10;
    my_package.stub_proc(i,j);
end;
```

If you place your cursor on the argument `j` in the call to `stub_proc`, you see that it can have any value allowed for an integer. The procedure `stub_proc` is not defined, forcing Polyspace to make this assumption.

Constraints

Specify this constraint in a text file:

```
# function stub constraints
my_package.stub_proc.res  -10 10 permanent
```

Use the option `Constraint setup` to specify this text file. After analysis, you see from the tooltips that the variable `j` is confined to the range [-10,10].

As with functions, instead of using constrained Polyspace stubs, you can write your own stubs. If you write your own stubs, you can implement constraints at a more granular level.

See Also

More About

- “Specifying Constraints Using Text Files” on page 4-9

Performing Efficient Module Testing with Constraints

External constraints allows you to perform efficient static testing of modules. To do so, you add design level information, which is missing in the source code.

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

- Input preconditions of call are designed for subprograms to be tested
- Input global data is consumed when testing subprograms
- Output data is produced by missing (stubbed) subprograms

Using external constraints, you can define:

- The nominal range for input arguments as preconditions of subprogram calls
- The generic range for input global variables at the start point of each subprogram test
- The generic range for return parameters of stubbed functions, and out or in/out parameters of procedures

These definitions then allow Polyspace software to perform a single static verification task, answering questions about robustness and reliability.

In this context, you assign keywords according to the type of data (input argument of call, input global data, stubbed subprogram output).

Type of Data	DRS Mode	Effect on Results	Why?	Oranges	Selectivity
Input argument of call	`init`	**Reduces** the number of orange checks (compared to a standard Polyspace verification)	Input arguments that were full range are set to a smaller and realistic range.	↓	↑
Input global data	`reinit`	**Reduces** the number of orange checks (compared to a standard Polyspace verification)	Input data that was full range is set to a smaller and realistic range.	↑	↑
Stubbed subprogram output	`permanent`	**Reduces** the number of orange checks (compared to a standard Polyspace verification)	Output data, produced by a missing subprogram, that was full range is set to a smaller and realistic range.	↑	↓

See Also

More About

- “Specifying Constraints Using Text Files” on page 4-9
- “Effect of External Constraints on Polyspace Analysis” on page 4-12

Reducing Orange Checks with External Constraints

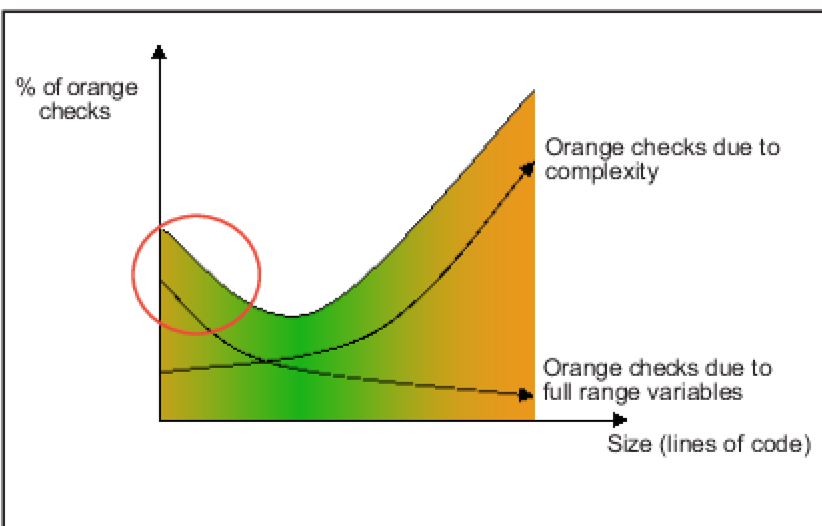
When performing robustness (worst case) verification, data inputs are set to their full range. Therefore, every operation on these inputs, even a simple “`one_input + 10`” can produce an overflow, as the range of `one_input` varies between the minimum value and the maximum value of the type.

If you use external constraints to restrict the range of “`one-input`” to the real functional constraints found in a 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 recognizes that:

- `one_input + 100` does not overflow
- the results of this operation are between `100` and `110`

This process eliminates the local overflow orange and results in more accuracy in the data. This accuracy is then propagated throughout the rest of the code.

The red circle indicates the orange checks that are removed by using constraints.



Removing orange checks caused by full-range (worst-case) data can significantly reduce the total number of orange checks, especially in the verification of small files or modules. However, the orange checks caused by code complexity does not change on applying constraints.

See Also

More About

- “Specifying Constraints Using Text Files” on page 4-9
- “Effect of External Constraints on Polyspace Analysis” on page 4-12

Using Pragma Assert to Set Data Ranges

You can use the construct `'pragma assert'` within your code to inform Polyspace of constraints imposed by the environment in which the software will run. A `"pragma assert"` function is:

`pragma assert(<integer expression>);`

If *`<integer expression>`* evaluates to zero, then the program is assumed to be terminated, therefore there is a “real” run-time error. This condition is why Polyspace produces checks for the assertions. The behavior matches the one exhibited during execution, because **execution paths for unsatisfied conditions are truncated** (red and then gray). Thus it can be assumed that a verification performed downstream of the assert uses value ranges which satisfy the assert conditions.

You can use the construct `'pragma assert'` in a procedure to inform Polyspace of constraints in the environment in which the software will be embedded. You can use user assertions to describe the physical properties of the environment, such as:

- The maximum and minimum speed limit (a car does not go faster than 200 miles per hour or slower than 0 miles per hour),
- The maximum duration of software exploitation (five years for a satellite and one hour for its launcher)

Example 4.1. Example

```
procedure main is
        counter: integer;
        -- counter is not initialized
        random: integer;
        pragma volatile (random);
begin
        counter:= random;
        -- counter~ [-2^31, 2^31-1]
        pragma assert (counter < 1000);
        pragma assert (counter > 100);
end;

end main;
```

Both assertions are orange because the conditions may or may not be fulfilled. From then on, counter ~ [101, 999] because execution paths that do not meet the conditions are halted.

Supported Ada Pragmas

Polyspace software provides verification support for many standard Ada or GNAT compiler pragmas.

Pragma	How Polyspace Software Processes Pragma
Import, Import_Function, and Import_Procedure	Stubs function or procedure
Interface and Interface_Name	Stubs function or procedure
Inspection_Point	Provides information about possible values for the variable. May display a range.
Volatile	Variable becomes full-range
Volatile_Components	If you specify Polyspace for Ada95, you get the same results as with the pragma Volatile. However, in this case, the pragma applies to arrays.
Assert	Produces a user assertion check, ASRT. See User Assertion.
Restrictions	Ignored for standard Ada or GNAT compiler restrictions. Other restriction pragmas produce a warning.
Ada_83 and Ada_95	Polyspace option -lang overwrites this pragma (option set by default to Ada95 when you use polyspace-ada).

Pure	Applies requirement that package has cross-dependencies only with other Pure packages. If requirement is not met, generates compilation errors.
	
	You can remove requirement by inserting pragma Not_Elaborated within package body. For example:
	
	`package System is`
	`pragma Pure;`
	`pragma Not_Elaborated;`
	
	`...`
	`end System;`

Prelaborate, Elaborate Elaborate_All, and Elaborate_Body	Provides order of elaboration and verification of packages by Polyspace
Storage_Unit	Polyspace option -storage_unit overwrites this pragma

Note If your code contains an unsupported pragma, Polyspace ignores the pragma and continues the verification. At the end of the compilation phase, Polyspace displays a message:

```
The following pragmas have been ignored...
```

How Polyspace Evaluates Function and Procedure Parameters

Polyspace applies by-copy semantics and a left-to-right evaluation order for parameter passing. You can use Polyspace to verify your Ada code provided your compiler implements:

- Left-to-right evaluation for subprogram parameters. Consider the following code.

```
1 with ada.integer_text_io;
2     use  ada.integer_text_io;
3     procedure test1 is
4        x,y,z,r : integer;
5
6       function f (x : integer) return integer
7       is
8        begin
9         z := 0;
10        return  x + 1;
11      end f;
12    begin
13      x := 10;
14      y := 20;
15      z := 10;
16      R := y / Z + F(x);
17      pragma assert(R = 13); -- green ASRT
18      put(R);
19    end;
```

  In this example, Polyspace verification implements left-to-right evaluation and generates a green ASRT check.

- By-copy semantics for subprogram parameters. Consider the following code.

```
1     procedure Test2
2     is
3
4      type Rec is
5       record
6        F,G: Integer;
7       end record;
8
9       R: Rec;
10      Result : Integer;
11
12      procedure Multiply (X, Y : in Rec; Z : out Rec)
13
14      is
15       begin
16        z := (0,0);
17        Z.F := X.F * Y.F;
18        Z.G := X.G * Y.G;
19       end Multiply;
20
21    begin
22     R := (10,10);
23     Result := 100;
24     Multiply (R,R,R);
25     Result := Result/R.F;
26     pragma assert (Result = 1); -- green ASRT
27    end Test2;
```

  In this example, Polyspace verification implements by-copy semantics and generates a green ASRT check.

The green checks generated indicate that the code conforms to the Ada standard, which states that *The execution of a program is erroneous if its effect depends on which mechanism is selected by the implementation*. See Formal Parameter Modes.

5

Preparing Source Code for Verification

- “Stubbing Overview” on page 5-2
- “Manual vs. Automatic Stubbing” on page 5-3
- “Automatic Stubbing” on page 5-6
- “Polyspace Software Assumptions” on page 5-7
- “Scheduling Model” on page 5-8
- “Modelling Synchronous Tasks” on page 5-9
- “Interruptions and Asynchronous Events/Tasks” on page 5-11
- “Are Interruptions Maskable or Preemptive by Default?” on page 5-13
- “Mailboxes” on page 5-15
- “Atomicity” on page 5-18
- “Priorities” on page 5-19

Stubbing Overview

A function stub is a small piece of code that emulates the behavior of a missing function. Stubbing is useful because it allows you to verify code before all functions have been fully developed.

Manual vs. Automatic Stubbing

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

There are two types of stubs in Polyspace verification:

- **Automatic stubs** – When you attempt to verify code that calls an unknown function, the software automatically creates a stub function based on the function's prototype (the function declaration). Automatic stubs generally 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 them in the verification with the rest of the source code.

Only advanced users should consider manual stubbing. Polyspace can automatically stub every missing function or procedure, leading to an efficient verification with a low loss in precision. However, in some cases you may want to manually stub functions instead. For example, when:

- Automatic stubbing does not provide an adequate representation of the code it represents— both in regards to missing functions and assembly instructions.
- The entire code is to be provided, which may be the case when verifying a large piece of 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 deal with a function that writes to global variables.

Deciding which Stub Functions to Provide

Stubs do not need to model the details of the functions or procedures involved. They only need to represent how the function interacts with the remainder of the code.

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, you can stub it to an empty action `begin null; end;`. Polyspace does not need a concept of timing because the software takes into account possible scheduling and interleaving of concurrent execution. You do not have to stub functions that set or reset a timer. Simply declare the variable representing time as volatile.
- an I/O access, such as to a hardware port, a sensor, read/write of a file, read of an EEPROM, write to a volatile variable:
  - You do not have to stub a write access. If you want to do so, you can stub it through an empty action `begin null; end;`.
  - You can stub read accesses using procedures that read volatile variables.
- a write to a global variable, consider which procedures or function write to it and why: do not stub the concerned *procedure_to_stub* if:
  - this variable is volatile;
  - this variable is a task list. Such lists are accounted for by default because tasks declared with the `-task` option are automatically started.

write a procedure_to_stub by hand if this variable is a regular variable read by other procedures or functions.

- a read from a global variable: if you want Polyspace to detect that it is a shared variable, you need to stub a read access as well. This is easy to achieve by copying the value into a local variable.

Generally speaking, follow the data flow and remember that:

- Polyspace only uses the Ada code which is provided.
- For multitasking code, Polyspace does not need to be informed of timing constraints through explicit time specification inside the code.

Example 5.1. Example

This example shows a header for a missing function (which might occur if, for example, the code is an incomplete subset or a project). The missing function copies the value of the src parameter to dest, so there would be a division by zero (RTE) at run time.

```
procedure a_missing_function
            (dest: in out integer,
            src  : in integer);
procedure test is
        a: integer;
        b: integer;
begin
        a: = 1;
        b: = 0;
        a_missing_function(a,b);
        b:= 1 / a;
        -- "/" with the default stubbing
end;
```

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 was commented out, then the division would be green.

A red division could only be achieved with a manual stub.

This example shows what might happen if the effects of assembly code are ignored.

```
procedure test is
begin
        a:= 1;
        b:= 0;
        -- b:= a
        pragma asm ("move: a,b")
        b:= 1 /a;
end;
```

Due to the reliance on the software's default stub, the assembly code is ignored and the division " **/"** is green. The red division "/" could only be achieved with a manual stub.

Summary

Stub manually: to gain precision by restricting return values generated by automatic stubs; to deal with a function which writes to global variables.

Stub automatically if you are sure that a run-time error will not be introduced by automatic stubbing; to minimize preparation time.

Automatic Stubbing

Some functions might not be included in the set of Ada source files because the functions are:

- External.
- Written in another programming language, for example, C.
- Part of the system libraries.

By default, Polyspace automatically stubs these functions.

Polyspace Software Assumptions

These are the rules followed by Polyspace. It is strongly recommended that the preceding sections should be read and understood before applying the rules described below. Some rules are mandatory; others facilitate improved selectivity.

The following describes the default behavior of Polyspace. If the code to be verified does not conform to these assumptions, then some minor modifications to the code or to the Polyspace run-time parameters will be required.

- The main procedure must terminate in order for entry-points (or tasks) to start.
- All tasks or entry-points start after the execution of the main has completed. They start simultaneously, without predefined assumptions regarding the sequence, priority and preemption.

If an entry-point is seen as dead code, it can be assumed that the main contains (a) red error(s) and therefore does not terminate. Polyspace does not assume any:

- “Atomicity” on page 5-18
- Timing constraints.

Scheduling Model

In the Polyspace model, the main procedure is executed first before other tasks are started. After it has finished, the task entry points are assumed to start concurrently in an interleaved manner. This is an accurate upper approximation model for most concurrent RTOS.

Tasks and main loops need to simply declare as entry points. It only concerns task not defined using keyword of the Ada language.

Example

```
procedure body back_ground_task is
begin
    loop -- infinite loop
-- background task body
-- operations
-- function call
my_original_package.my_procedure;
    end loop
end back_ground_task
```

Launching Command

```
polyspace-ada -entry-points package.other_task,package.back_ground_task
```

If the tasks are already infinite loops, simply declare them as mentioned above.

Limitation

- A main procedure using the `-main` option is required.
- **The tasks declared in `-entry-points` may not take parameters and may not have return values:** procedure MyTask is end MyTask;

  If it is not the case, it is mandatory to encapsulate with a new procedure. In this case, the real task will be called inside.
- The main procedure cannot be called in a defined or declared task.

Modelling Synchronous Tasks

Problem

My application has the following behavior:

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

My tasks do not interrupt each other. My tasks do not contain infinite loops.

```
procedure tsk_10ms;
begin do_things_and_exit();
    -- it's important it returns control
end;
```

Explanation

If each task was declared to Polyspace by using the option

```
polyspace-ada -entry-points pack_name.tsk_10ms, pack_name.tsk_30ms,
pack_name.tsk_50ms
```

then the results **would** be valid. However, because more scenarios than those encountered at execution time are modelled, there may be unnecessarily more warnings — the results are less precise.

In order to address this, Polyspace Server for Ada needs to be informed that the tasks are purely sequential. This can be achieved by writing a function to call each of the tasks in the right sequence, and then declaring this new function as a single task entry point.

Solution 1

Write a function that calls the cyclic tasks in the right order: this is an **exact sequencer**. This sequencer is then identified to the software as a single task.

This sequencer will be a single Polyspace task entry point. This solution:

- is more precise,
- but you need to know the exact sequence of events.

```
procedure body one_sequential_Ada_function is
begin
    loop
        tsk_10ms;
        tsk_10ms;
        tsk_10ms;
        tsk_30ms;
        tsk_10ms;
        tsk_10ms;
        tsk_50ms;
    end_loop
end one_sequential_Ada_function;

polyspace-ada -entry-points pack_name.one_sequential_Ada_function
```

Solution 2

Make an **upper approximation sequencer**, which takes into account every possible scheduling. This solution:

- is less precise,
- but is quick to code, especially for complicated scheduling.

```
procedure body upper_approx_Ada_function is
    random : integer;
    pragma volatile (random);
begin
    loop
        if (random = 1) than tsk_10ms; end if;
        if (random = 1) than tsk_30ms; end if;
        if (random = 1) than tsk_50ms; end if;
    end_loop
end upper_approx_Ada_function;

polyspace-ada -entry-points pack_name.upper_approx_Ada_function
```

Note If this is the only task, then it can be added at the end of the main.

Interruptions and Asynchronous Events/Tasks

Problem

Interrupt service routines appear gray (dead code) in the Polyspace user interface.

Explanation

The gray code indicates that this code is not executed and is not taken into account, so the interruptions are ignored by Polyspace Server for Ada.

The execution model is such that the main is executed initially. Only if the main terminates and returns control (i.e. if it is not an infinite loop) will the task entry points be started.

My interrupts it1 and it2 cannot preempt each other

You can group interruptions in a single function and declare that function as a task entry point if the following conditions are fulfilled:

- The functions `it1` and `it2` cannot interrupt each other.
- Each interrupt can be raised several times in a row.
- The functions do not contain infinite loops.

```
procedure it_1;
procedure it_2;

task body all_interruptions_and_events is
random: boolean;
pragma volatile (random);
begin
    loop
        if (random) then it_1; end if;
        if (random) then it_2; end if;
    end_loop
end all_interruptions_and_events;

polyspace-ada -entry-points package.all_interruptions_and_events
```

My interruptions can preempt each other

If two interruption can be interrupted, then:

- encapsulate each of them in a loop;
- declare each loop as a task entry point.

```
package body original_file is
    procedure it_1 is begin ... end;
    procedure it_2 is begin ... end;
    procedure one_task is begin ... end;
end;

package body new_poly is
procedure polys_it_1 is begin loop it_1; end loop; end;
procedure polys_it_2 is begin loop it_2; end loop; end;
```

```
procedure polys_one_task is begin loop one_task; end loop; end;

polyspace-ada -entry-points new_poly. polys_it_1,new_poly. polys_it_2,
new_poly.polys_one_task
```

Are Interruptions Maskable or Preemptive by Default?

Problem

In my main task I use a critical section but I still have unprotected shared data. My application contains interrupts. Why is my variable verified as unprotected?

Explanation

Polyspace Server for Ada does not distinguish between interrupt service routines and tasks. If you specify an interrupt to be an `-entry-point`, it will have the same priority level as other procedures that are also declared as tasks via the `-entry-point` option. Therefore, as Polyspace Server for Ada makes an **upper approximation of scheduling and interleaving**. This upper approximation **includes the possibility that the ISR can be interrupted by other tasks**. There are more paths modelled than can happen during execution.

Solution

Embed your interrupt in a specific procedure that uses the same critical section as the one you use in your main task. Then, each time this function is called, the task will enter a critical section which will be equivalent to a nonmaskable interruption.

Original Packages

```
package my_real_package is
    procedure my_main_task;
    procedure my_real_it;
    shared_X: INTEGER:= 0;
end my_real_package;

package body my_real_package is
    procedure my_main_task is
    begin
        mask_it;
        shared_x:= 12;
        unmask_it;
    end my_main_task;

    procedure my_real_it is
    begin
        shared_x:= 100;
    end my_real_it;
end my_real_package;
```

Extra Packages

An extra package that is required to embed the task with body my_real_package;

```
package extra_additional_pack is
    procedure polyspace_real_it;
end extra_additional_package;
```

```
package body extra_additional_pack is
    procedure polyspace_real_it is
    begin
        mask_it;
        my_real_package.my_real_it;
        unmask_it;
    end;
end extra_additional_package;
```

Command Line to Open Polyspace User Interface

```
polyspace-ada \
-entry-point my_real_package.my_main_task,extra_additional_pack\
polyspace_real_it
\
-main your_package.your_main
```

Mailboxes

Problem

My application has several tasks:

- some that post messages in a mailbox;
- others that read these messages asynchronously.

This communication mechanism is possible because the OS libraries provide send and receive procedures. I do not have the source files because these procedures are part of the OS libraries.

Explanation

By default, Polyspace Server for Ada will automatically stub these send/receive procedures. Such a stub will exhibit the following behavior:

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

Solution

You can provide similar mechanisms with different levels of precision.

Mechanism	Description
Let **Polyspace Server for Ada stub automatically**	• Quick and easy to code • **Imprecise** because between a mailbox sender and receiver are not directly connected. It means that even if the sender is only submitting data within a small range, the full data range for the type(s) will be used for the receiver data.
Provide a **real mailbox** mechanism	• Can be very costly (time consuming) to implement • Can introduce errors in the stubs • Is too much effort compared with the solution below • Precise, but does not provide a much better precision than the upper approximation
Provide an **upper approximation of the mailbox**	in which each new read to the mailbox reads **one** of the recently posted messages, but not necessarily the last one. • Quick and easy to code • Gives precise results • See detailed implementation below

package mailboxes

```
    type BIG_ARRAY is
        array (1..100)of INTEGER;
    type MESSAGE is
    record
        length: INTEGER;
        content: BIG_ARRAY;
    end MESSAGE;
    MAILBOX : MESSAGE;
    procedure send
        (X: in MAILBOX);
    procedure receive
        (X: out MAILBOX);
end mailboxes;
```

package body mailboxes

```
procedure send (X: in MESSAGE) is
    random : boolean;
    pragma Volatile_(random);
begin
    if (random) then
        MAILBOX:= X;
    end if;
     -- a potential write
     -- to the mailbox
end;
```

procedure receive

```
(X: out MESSAGE) is
begin
    X:= MAILBOX;
end;
```

task body task_1

```
    msg : MESSAGE;
begin
    for i in 1 .. 100 loop
        msg.content(i):= i;
    end loop;
    msg.length : = 100;
    send(msg);
end task_1;
task body task_2 is
    msg : MESSAGE;
begin
    receive(msg);
    if (msg.length = 100) ...
end;
```

Provided that each of these tasks is included in a package.

```
polyspace-ada -main a_package.a_procedure
```

Atomicity

Definitions

- *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 the pieces are committed or none are.

Instructional Decomposition

In general terms, Polyspace Server for Ada does not take into account either CPU instruction decomposition or timing considerations.

It is assumed by Polyspace that instructions are not atomic except in the case of read and write instructions. Polyspace Server for Ada makes an **upper approximation of scheduling and interleaving**. Because of this approximation, the software models more paths than could happen during execution.

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 was 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 0xFF00, 0x0055 or 0xFF55.

Polyspace Server for Ada considers write/read instructions atomic, so **task 2 can only read 0xFF55**, even if X is not protected.

Critical Sections and Temporal Exclusion

Polyspace Server for Ada does not model the concept of atomicity for critical sections and temporally exclusive tasks. A critical section implies that once the function associated with `-critical-section-begin` has been called, other functions making use of the same label will be blocked. Functions not using the label can continue to run.

Polyspace Server for Ada verification of run-time errors supposes that a conflict does not occur when writing the shared variables. Hence even if a shared variable is not protected, the run-time error verification is complete and correct.

More information about protection is available in `Critical section details` or `Temporally exclusive tasks`.

Priorities

Polyspace does not consider priorities of tasks during verification. In addition, Polyspace does not assume that priorities can protect shared variables.

Though you cannot implement different task priorities, the verification effectively takes all priorities into account because it assumes that:

- All task entry points that you specify on the **Configuration** pane start at the same time.
- They can interrupt each other in any order, regardless of the sequence of instructions.

For instance, if you have two tasks `t1` and `t2`, and `t1` has higher priority than `t2`, use `polyspace-ada -entry-points t1,t2`. Polyspace assumes that:

- `t1` can interrupt `t2` at arbitrary intervals, thus modelling the behavior at execution time.
- `t2` can also interrupt `t1` at arbitrary intervals. This behavior does not occur at execution time unless priority inversion takes place. Polyspace Server for Ada makes an upper approximation of scheduling and interruptions. Because of this approximation, the software models more paths than possible during actual execution.

Running a Verification

- "Run Local Verification" on page 6-2
- "Run Remote Verification" on page 6-4
- "Phases of Verification" on page 6-6
- "Run File-by-File Local Verification" on page 6-7
- "Run File-by-File Remote Verification" on page 6-9
- "Manage Job Monitor" on page 6-10
- "Run Local Verification at Command Line" on page 6-13
- "Run Remote Verification at Command Line" on page 6-14
- "Create Command-Line Script from Project File" on page 6-16

Run Local Verification

Before running verification on your source files, you must add them to a Polyspace project. For more information, see “Create Project”.

In this section...
“Start Verification” on page 6-2
“Monitor Progress” on page 6-2
“Stop Verification” on page 6-2
“Open Results” on page 6-3

Start Verification

To start a verification on your local desktop:

1. On the **Project Browser** pane, select the project module that you want to verify.
2. On the toolbar, click the Run button.

Tip To run verification on all modules in the project, expand the drop-down list beside the Run. Select **Run All Modules**.

Monitor Progress

To monitor the progress of a local verification, use the following panes. If you have closed a pane, to open it again, select **Window** > **Show/Hide View**.

- **Output Summary** — Displays progress of verification, compile phase messages and errors.
- **Run Log** — This tab displays messages, errors, and statistics for all phases of the verification.

Tip To search for a term in the **Output Summary** or **Run Log**, enter the term on the **Search** pane. Select **Output Summary** or **Run Log** from the drop-down list beside the search box.

If the **Search** pane is not open by default, select **Windows** > **Show/Hide View** > **Search**.

At the end of a local verification, the **Dashboard** tab displays statistics, for example, percentage of code checked for run-time errors and check distribution.

Stop Verification

To stop a local verification:

1. On the toolbar, click the **Stop** button.

   A warning dialog box opens asking whether you want to stop the execution.

2 Click **Yes**. The verification stops, and results are incomplete. If you start another verification, the verification starts from the beginning.

Open Results

After verification, the results open automatically on the **Results List** pane. If you are looking at previous results when a verification is over, you can load the new results or retain the previous results on the **Results List** pane.

To open the new results later:

1. On the **Project Browser** pane, navigate to the results set that you want to review.
2. Double-click the results set, for example, **Result_1**.

   The software loads the verification results in the **Results List** pane.

To open results of verification when the corresponding project is not open in the **Project Browser** pane:

1. Select **File** > **Open**.
2. In the Open File dialog box, navigate to the results folder. For example:

   `My_project\Module_1\Result_1`
3. Select the results file, for example, `My_project.rte`.
4. Click **Open**.

Run Remote Verification

Run remote verification when:

- You want to shut down your local machine but not interrupt the verification.
- You want to free execution time on your local machine.
- You want to transfer verification to a more powerful computer.

Before you run remote verification, you must do the following:

- Set up a server for this purpose. For more information, see “Polyspace Software Administration”.
- Add your source files to a Polyspace project. For more information, see “Create Project”.

In this section...
“Start Verification” on page 6-4
“Monitor Progress” on page 6-4
“Stop Verification” on page 6-5
“Open Results” on page 6-5

Start Verification

To start a remote verification:

1. On the **Project Browser** pane, select the module you want to verify.
2. On the **Configuration** pane, select **Machine Configuration**. Select **Send to Polyspace Server**.
3. On the toolbar, click the Run button.

   The verification starts. For information on the verification process, see “Phases of Verification” on page 6-6.

   Note If you see the message `Verification process failed`, click **OK** and go to “Troubleshooting in Polyspace Products for Ada”.

Monitor Progress

You can manage your verification through the Polyspace Job Monitor.

1. Select **Tools** > **Open Job Monitor**.
2. In the Polyspace Job Monitor, right-click your verification.
3. From the context menu, select your management task:

   - **View Log File** — Open the verification log.
   - **Download Results** — Download verification results from remote computer if the verification is complete.

Stop Verification

1. Select **Tools** > **Open Job Monitor**.
2. In the Polyspace Job Monitor, right-click your verification. From the context menu, select **Remove From Queue**.

Open Results

Your results are downloaded automatically after verification. To open them:

1. On the **Project Browser** pane, navigate to the results set.
2. Double-click the results set, for example, **Result_1**.

   The software loads the verification results in the **Results List** pane.

Phases of Verification

The verification has three main phases:

1. Checking syntax and semantics (the compile phase). Because Polyspace software is independent of a particular Ada compiler, it ensures that your code is portable, maintainable, and complies with ANSI® standards.
2. Generating a main if the verification does not find a main and you selected the **Verify module** option. For more information, see “Generate a main”.
3. Analyzing the code for run-time errors and generating color-coded diagnostics.

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 Polyspace user interface. 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** view gives you the identification number (Analysis ID) for the verification.

Run File-by-File Local Verification

This example shows how to run a local verification on each file independently of other files in the module. You need a Polyspace Server for Ada license to perform a local file-by-file verification on your desktop.

In this section...
"Run Verification" on page 6-7
"Open Results" on page 6-7

Run Verification

1. Select your project configuration. On the **Configuration** pane, specify that each file must be verified independently of other files.
   a. Select the **Verification Mode** node.
   b. Select **Verify module** and then **Verify files independently**.
   c. For **Common source files**, enter files that you want to include in the verification of each file. Enter the full path to a file. Enter one file path per row.
2. On the toolbar, click the button.

   As you can see on the **Output Summary** pane, you can see that after the **Compile** phase, each file is verified independently. After the verification is complete for a file, you can view the results while other files are still being verified.

Open Results

After verification, your results appear in the **Project Browser**. The results are grouped under a root node below the **Result** node of your module. The results for each source file has the same name as the source file.

After a source file is verified, to open the results, double-click the corresponding result file under the **Result** node. Alternatively, after all source files are verified, you can see a summary of results for all files and begin reviewing from files with more severe issues.

1. To open the results, click the root node below the **Result** node in your project module. For instance, click **Result_1** if your project uses the default result naming scheme.

   On the **Dashboard** pane, you can see a summary of results for all files. The files in the summary table are sorted by the severity of check colors. For instance, the files are sorted by the number of red checks. The files with the same number of red checks are sorted by the number of gray checks and so on.
2. To load the results for an individual file, double-click the file name on the table.

   After you load the results for an individual file, the **Dashboard** pane shows graphs for the current file. The summary table for all files appears on a separate **Unit by unit results synthesis** tab on this pane. You can use this tab to load results for other files.

See Also

`Verify files independently`

Run File-by-File Remote Verification

This example shows how to run a remote verification on each file independently of other files in the module.

Before you run remote verification, you must do the following:

- Set up a server for this purpose. For more information, see “Polyspace Software Administration”.
- Add your source files to a Polyspace project. For more information, see “Create Project”.

In this section...
“Run Verification” on page 6-9 “Open Results” on page 6-9

Run Verification

1. On the **Project Browser** pane, select the module you want to verify.
2. On the **Configuration** pane, select **Machine Configuration**. Select **Send to Polyspace Server**.
3. On the **Configuration** pane, specify that each file must be verified independently of other files.
   a. Select the **Verification Mode** node.
   b. Select **Verify module** and then **Verify files independently**.
   c. For **Common source files**, enter files that you want to include with verification of each file. Enter the full path to a file. Enter one file path per row.
4. On the toolbar, click the button.

   After the **Compile** phase, you can view the jobs in the Polyspace Job Monitor.
5. Select **Tools** > **Open Job Monitor**.

   Your files appear as child nodes under the main verification node. After the verification is complete for a file, you can download and view the results while other files are still being verified. Right-click the row corresponding to the file and select **Download Results**.

Open Results

Your results are automatically downloaded after verification.

To open result for each source file, double-click the corresponding result file under the **Result** node. The result file has the same name as the source file.

See Also

`Verify files independently`

Manage Job Monitor

In this section...

"Purge Server Queue" on page 6-10
"Change Job Monitor Password" on page 6-10
"Share Server Verifications Between Users" on page 6-11

Purge Server Queue

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

Note You must have the Job Monitor password to purge the server queue.

To purge the server queue:

1 Select **Tools** > **Open Job Monitor**.

   The Polyspace Job Monitor 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 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, the jobs are not removed until the entire group has been verified.
4 Enter the Job Monitor **Password**.
5 Click **OK**.

   The server queue is purged.

Change Job Monitor Password

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

Note The default password is `admin`.

To set the Job Monitor password:

1 Select **Tools** > **Open Job Monitor**.

   The Polyspace Job Monitor opens.

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

The Change Administrator Password dialog box opens.

3 Enter your old and new passwords. Then click **OK**.

The password is changed.

Note Passwords are limited to 8 characters.

Share Server Verifications Between Users

Security of Jobs in Server Queue

For security reasons, 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 (for example, download, kill, and remove), the Job Monitor 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` which is 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 6.1. 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 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, verification jobs you submit to the server queue use this key instead of a random one. 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 Job Monitor password to use Administrator Mode.

To use administrator mode:

1. Select **Tools** > **Open Job Monitor**.

   The Polyspace Job Monitor opens.
2. Select **Operations** > **Enter Administrator Mode**.
3. Enter the Job Monitor **Password**.
4. Click **OK**.

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

Run Local Verification at Command Line

At the Windows or Linux or command-line, append sources and analysis options to the `polyspace-ada` command.

For instance:

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

  ```
  polyspace-ada -sources "file.adb" -lang ada95 -target m68k
  ```
- To specify verification precision, use the `-O` option. For instance, to set precision level to 2 for your source file `file.adb`, use the command:

  ```
  polyspace-ada -sources "file.adb" -lang ada95 -O2
  ```

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

You can also enter the following at the command line:

```
polyspace-ada -help
```

Run Remote Verification at Command Line

In this section...

"Start Verification" on page 6-14
"Manage Verification" on page 6-14
"Download Verification Results from Server" on page 6-15

Start Verification

A set of commands allow you to run remote verifications.

These commands begin with the following prefixes:

- **Server verification** — `*Polyspace_Install*/polyspace/bin/polyspace-remote-ada`
- **Client verification** — `*Polyspace_Install*/polyspace/bin/polyspace-remote-ada -desktop`

For example,`polyspace-remote-ada -desktop -server [<*hostname*>:[<*port*>] | auto]` connects the client to the specified server. This connection allows you to run verifications remotely on the server.

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

Manage Verification

A set of commands allow you to manage verification jobs in the server queue. These commands begin with the prefix *Polyspace_Install*/polyspace/bin/psqueue-:

- psqueue-download <*id*> <*results dir*> — download an identified verification into a results folder. When downloading a unit-by-unit verification group, 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 Job Monitor.
  - [-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 completed verifications from the queue. For unit-by-unit verification jobs, the jobs are not removed until the entire group has been verified.
- psqueue-dump — gives the list of verifications in the queue associated with the default Job Monitor. 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 Job Monitor.
- `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 Polyspace user interface.
  - `[-full]` give full log file.
  - `psqueue-set-password` <*password*> <*new password*> — change administrator password.
- `psqueue-check-config` — check the configuration of Job Monitor.
  - `[-check-licenses]` check for licenses only.
- `psqueue-upgrade` — Allow to upgrade a client side. See “Software Installation”.
  - `[-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_Install*`/polyspace/bin/psqueue-` <*command*> `-h` provides information about available options for each command.

Download Verification Results from Server

You can download verification results at the command line using the `psqueue-download` command.

To download your results, enter the following command:

<*PolyspaceCommonDir*>`/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 later in the Polyspace user interface.

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 the 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.

Create Command-Line Script from Project File

In this section...
"Generate Scripting Files" on page 6-16
"Run an Analysis" on page 6-16

This example shows how to use a project file that you configured in the Polyspace interface to generate the necessary information to run from the command line. If you have already spent time configuring your project in the Polyspace interface, this command is useful to extract your setup work for scripting. For this example, you use the example shipped with Polyspace.

Generate Scripting Files

1. In the Polyspace interface, open the example project by selecting **Help** > **Examples** > **Demo_Ada**.

   This example has been set up and configured with analysis options.
2. Open a command-line terminal and navigate to your `Polyspace_Workspace` folder. By default it is:

   - Linux — `/home/USER/Polyspace_Workspace`
   - Windows — `Users\USER\Documents\Polyspace_Workspace`
   - Mac — `USER/Polyspace_Workspace`
3. Navigate down to the example project:

   ```
   cd Examples/R2017b/Demo_Ada
   ```
4. Run the script generation command . (*`matlabroot`* is your installed program folder, for example `C:\Program Files\MATLAB\R2017b`.)

   ```
   matlabroot/polyspace/bin/polyspace -generate-launching-script-for Demo_Ada.psprj
   ```

   Polyspace generates the following folder structure`Demo_Ada\Demo_Ada`. The lowest level `Demo_Ada` contains:

   - `source_command.txt` — List of source files
   - `options_command.txt` — List of the analysis options
   - `launchingCommand.sh` (UNIX) or `launchingCommand.bat` (DOS) — Shell script that calls the correct commands

For more details about what files are generated and how to use them, see `-generate-launching-script-for`.

Run an Analysis

After you have completed, "Generate Scripting Files" on page 6-16, you can use the files to run an analysis from the command line. The launching script makes integrating into continuous integration tools such as Jenkins, easier. Here are a few examples of how to use the generated files to run an analysis.

- Run the generated script locally by using the `launchingCommand.bat` file.

```
Demo_Ada\Demo_Ada\launchingCommand.bat
```

- Run the generated script and change the results folder.

```
Demo_Ada\Demo_Ada\launchingCommand.bat -results-dir Results_Demo_Ada_mine
```

  The extra `-results-dir` option overrides the results folder specified in the `options_command.txt` file.

- Send the analysis to a remove server using the options files.

```
matlabroot/polyspace/bin/polyspace-remote-ada -server ...
    -options-file Demo_Ada\Demo_Ada\options_command.txt
```

- Run the analysis from the command line with the `-options-file` option.

```
matlabroot/polyspace/bin/polyspace -options-file ...
    Demo_Ada\Demo_Ada\options_command.txt
```

See Also

`-generate-launching-script-for`

External Websites

- How do I use Polyspace with Jenkins?

Troubleshooting Verification

- "Hardware Does Not Meet Requirements" on page 7-2
- "Location of Included Files Not Specified" on page 7-3
- "Polyspace Software Cannot Find the Server" on page 7-4
- "Limit on Assignments and Function Calls" on page 7-6
- "Examining the Compile Log" on page 7-7
- "Common Compile Errors" on page 7-8
- "Error from Special Characters" on page 7-15
- "Verification Time Considerations" on page 7-16
- "Displaying Verification Status Information" on page 7-17
- "Ideal Application Size" on page 7-18
- "Optimum Size" on page 7-19
- "Selecting a Subset of Code" on page 7-20
- "Benefits of Methods" on page 7-24
- "Obtaining Configuration Information" on page 7-26
- "Reasons for Unchecked Code" on page 7-27
- "Storage of Temporary Files" on page 7-30
- "Disk Defragmentation and Antivirus Software" on page 7-31
- "Out-of-Memory Errors During Report Generation" on page 7-32

Hardware Does Not Meet Requirements

If your computer does not have the minimal hardware requirements, you see a warning during verification, but the verification continues. For information about hardware requirements for the Polyspace products, see:

`www.mathworks.com/products/polyspaceclientada/requirements.html`

To avoid this issue, upgrade your computer to meet the minimal requirements.

Location of Included Files Not Specified

If you see the following message, either the included files are missing or you did not specify the location of included files:

```
example.adb, line 12 (column 14): Error: "runtime_error (spec)" depends
on "types (spec)"
```

For information on how to specify the location of include files, see “Create Project”.

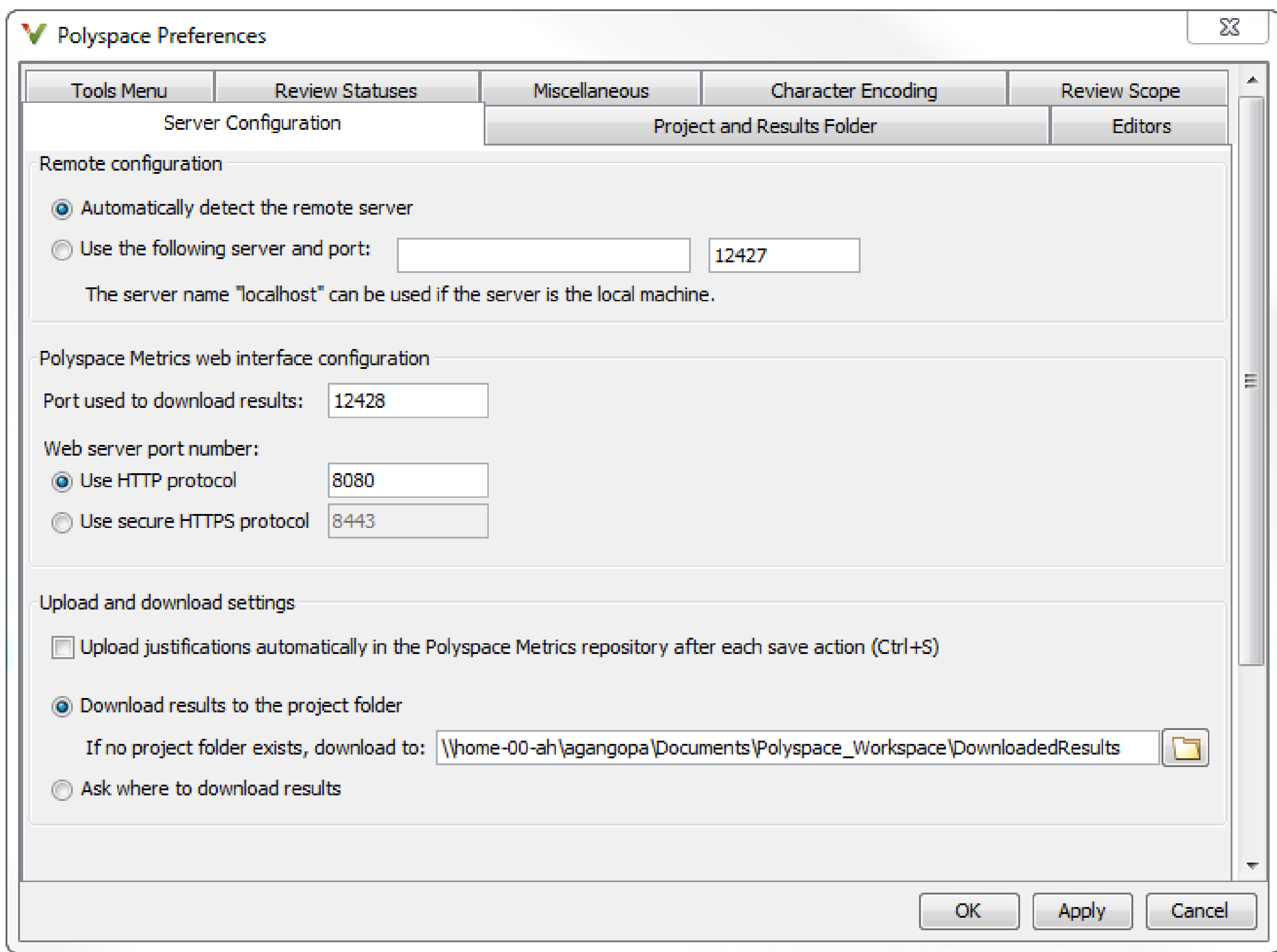
Polyspace Software Cannot Find the Server

If the Polyspace software cannot find the server, you see the following message in the log:

```
Error: Unknown host :
```

To find the server information:

1. Select **Tools** > **Preferences**.
2. Select the **Server configuration** tab.



How you handle 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	Check the server name and port number.

For information about setting up a server, see the *Polyspace Installation Guide*.

Limit on Assignments and Function Calls

If you start a client verification for a large file, the verification can stop with an error message stating that the number of assignments and function calls is too large. For example:

```
*** License error: number of assignments and function calls is too large
*** for the desktop mode (15462 v.s 2000).
*** Aborting.

-------------------------------------------------------------------
---                                                             ---
---  Verifier has encountered an internal error.            ---
---  Please contact your technical support.                 ---
---                                                             ---
-------------------------------------------------------------------

Failure at: Dec 21, 2009 18:21:42
User time for polyspace-ada: 1773real, 1097.1u + 101s (6.1gc)
Exiting because of previous error

***
*** End of Polyspace Verifier analysis
***
```

The Polyspace Client for Ada software can verify only Ada code with up to 2,000 assignments and calls.

To verify code containing more than 2,000 assignments and calls, run a server verification using Polyspace Server for Ada.

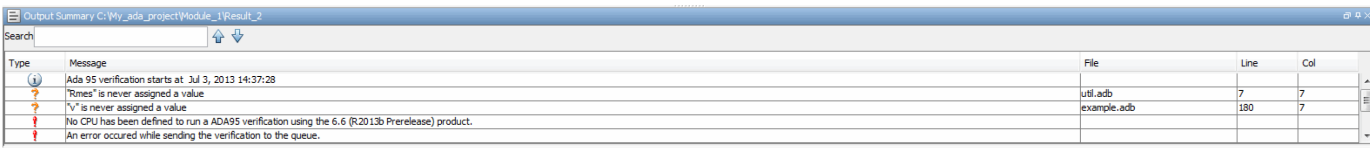
Examining the Compile Log

The compile log displays compile-phase messages and errors. You can search the log by entering search terms in the **Search** box.

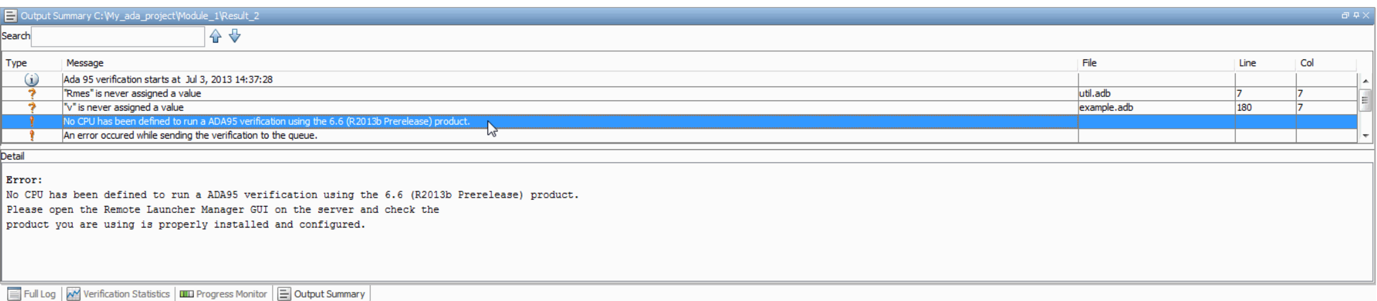
To examine errors in the Compile log:

1 In the log area of the Polyspace user interface, click **Compile**.

A list of compile-phase messages appear in the log part of the window.



2 Click a message to see message details, as well as the full path of the file containing the error.



3 To open the source file referenced by a message, right-click the row for the message. From the context menu, select **Open Source File**.

The file opens in your text editor.

4 Fix the error and run the verification again.

Common Compile Errors

In this section...

"Missing specification for unit" on page 7-8
"Calendar not found" on page 7-8
"Not a predefined library unit" on page 7-9
"representation clause appears too late" on page 7-9
"Package system and standard include" on page 7-10
"Unsigned type" on page 7-10
"Function not declared in package" on page 7-10
"pre-elaborated unit" on page 7-11
"actual must be a definite subtype" on page 7-11
"'ref attribute" on page 7-12
"Cannot load s-dec.ads (unit not found)" on page 7-12
"Green Hills standard include" on page 7-13
"Package Analysis Limitation" on page 7-13
"Ambiguous Bounds in Discrete Range" on page 7-14

Missing specification for unit

Problem

You must supply complete specifications associated with a package body verification to the Polyspace software. If you do not, you might encounter the following error message:

```
Verifying _pst_main

Verifying my_package

-> Verifier found an error
in ./My_Package.adb, line 2 (column 14):
Missing specification for unit "My_Package"
```

Solution/Workaround

Include the specifications of the package body in the list of supplied sources.

Explanation

When you supply a package body as the source, and the package body specification as one of the specifications in one of the `-ada-include-dir` folders, the Polyspace software reports this error.

Calendar not found

Problem

The compiler did not find the package calendar.

Solution/Workaround

In the `sources` folder, create a file with:

```
With ada.calendar;
package calendar renames ada.calendar
```

Explanation

For some compilers, the package calendar is on the top level. For the GNAT compiler, the calendar is a child of Ada.

Not a predefined library unit

Problem

You see the error message:

```
"machine_code" is not a predefined library unit
```

Solution/Workaround

In the `sources` folder, create a file with the following lines:

```
with System.Machine_Code;
package Machine_Code renames System.Machine_Code;
```

Explanation

Depending on the compiler that you are using, the subpackage of the package system can have a different name.

representation clause appears too late

Problem

The compilation phase stops and displays the warning:

```
representation clause appears too late
```

Solution/Workaround

Change:

```
type the_type is new Integer range 0 .. 10;
var : the_type;
for the_type'size use 16; -- Error : representation clause appears too late
```

to:

```
type the_type is new Integer range 0 .. 10;
for the_type'size use 16;
```

Explanation

If you use a type between its declaration clause and the representation clause, the Polyspace software displays this warning.

Package system and standard include

Problem

The standard include files are dependent on the compiler. You may see the following error message:

```
-> Verifier found an error in f1.ada, line 253 (column 29): "Offset" not
   declared(1) in "System"
-> Verifier found an error in f2.ada, line 758 (column 43): expected type
   "System.OFFSET"
```

Solution/Workaround

Copy the `system.ads` file from *`ada_include_path`* into your `sources` folder and insert the line:

```
type OFFSET is range -2**31 .. 2**31-1;
```

If your project source language is `Ada83`, the *`ada_include_path`* is *`product_root`*`\polyspace\verifier\ada\ada83include`.

If your project source language is `Ada95`, the *`ada_include_path`* is *`product_root`*`\polyspace\verifier\ada\ada95include\os-support`.

`product_root` is the folder where Polyspace Client for Ada is installed, for instance `C:\Program Files\Polyspace\PolyspaceForADA_R2017b\`

Explanation

This type definition is specific to the AONIX/Alsys Ada compiler.

Unsigned type

Problem

Some code uses unsigned types. The Polyspace compiler does not support unsigned types.

Solution/Workaround

Define unsigned types as follows:

```
type unsigned_integer is mod 4294967296;
type unsigned_short_integer is mod 65536;
type unsigned_tiny_integer is mod 256;
```

Function not declared in package

Problem

The package `operations` does not declare the function `New_ATCB`. The package `System.Tasking.Initialization` declares that function.

Solution/Workaround

Copy the file `s-taprop.ads` from `<product-dir>/adainclude/` into the `sources` folder. Into the `s-taprop.ads` file, insert the following line:

```
function New_ATCB (Self_ID : integer) return Task_ID;
```

Explanation

Add missing specifications to the package.

pre-elaborated unit

Problem

This package has a `pragma preelaborate` construct.

Solution/Workaround

Comment out the `pragma preelaborate` construct.

actual must be a definite subtype

Problem

The compile error message is:

```
actual for "SOURCE" must be a definite subtype
```

If the formal subtype is definite, the actual subtype must also be definite. This error is a valid compilation error in Ada 95 but is not valid in Ada 83. For more information, see the Ada 95 standard (12.5.1-6) and Ada 95 annotated (12.5.1-28.a).

The following example can be extended to other generic declarations. This example is based on the `unchecked_conversion` generic function.

The example code is:

```
generic
   type SOURCE is limited private;
   type TARGET is limited private;

function UNCHECKED_CONVERSION (S : SOURCE) return TARGET;


with UNCHECKED_CONVERSION;
package Test is
        type INDEX is new INTEGER;
    type DATA_INDEX is new INTEGER;

    type UNCONSTRAINED_DATA_TYPE is array
    (INDEX range <>) of INTEGER;

    subtype CONSTRAINED_DATA_TYPE is
         UNCONSTRAINED_DATA_TYPE (INDEX range INDEX'First..Index'LAST);

    function TO_DATA is new UNCHECKED_CONVERSION
       (SOURCE => UNCONSTRAINED_DATA_TYPE,
        TARGET => INTEGER);

    procedure Main;

end Test;
```

Solution/Workaround

Change the lines:

```
type SOURCE is limited private;
    type TARGET is limited private;
```

to:

```
type SOURCE (<>) is limited private;
type TARGET (<>) is limited private;
```

to match the Polyspace definitions.

Explanation

The Polyspace provides its own version of `Unchecked_Conversion` and its own definition of the `SOURCE` and the `TARGET`.

'ref attribute

Problem

The use of the `'ref` attribute is not standard. The Polyspace software does not support that attribute.

Two examples that cause a compile error are:

```
system.address'ref (16#FFFF_FFFF#)
```

```
a_var'ref
```

Solution/Workaround

In the preceding examples, use the following code instead:

```
system.address (16#FFFF_FFFF#)
```

```
var'address
```

Explanation

This attribute is dependent on the compiler.

Cannot load s-dec.ads (unit not found)

Problem

When compiling VMS Ada code, you may see the following error message:

```
cannot load s-dec.ads (unit not found)
```

Solution/Workaround

Comment out every line that uses the `AST_entry` or `Type_class` attribute.

Explanation

The `AST_entry` and `Type_class` attributes are specific to VMS Ada.

Green Hills standard include

Problem

When analyzing a Green Hills® application, you may see compile errors due to:

- The compatibility between the Polyspace and Green Hills include files
- A limitation the Polyspace Verifier encounters when compiling a Green Hills include file

Solution/Workaround

The Polyspace software now provides a specific option for the Green Hills Ada compiler. For more information, see `Target operating system`.

Explanation

The `$POLYSACE_ADA/adainclude/greenhills` folder contains the Green Hills compiler include files.

Package Analysis Limitation

Problem

Suppose you have a types package that defines a task to a pointer type. Other packages include this type package using the `with` clause. When you use that pointer type in the package, you cannot analyze that package.

Solution/Workaround

1. Copy package specifications that have unsupported construction from the `includes` folder to the `include-modified` folder.
2. In these files, comment out every unsupported construction.
3. Use the `-ada-include-dir` option to incorporate the modified files in the analysis.

   For example:

   ```
   polyspace-ada -lang ada95 \
   -ada-include-dir $HERE/includes \
   -ada-include-dir $HERE/includes-modified \
   -extensions-for-spec-files "*.a??"
   ```

   Note If a package is defined in two different folders, the file compiled and analyzed by Polyspace is the last one specified.

Explanation

By taking these steps, you do not have to modify the original files. You must maintain copies of the original files in the `includes-modified` folder. These types of include files do not change very often.

Use this workaround for an Ada compiler standard include file.

Ambiguous Bounds in Discrete Range

Problem

The type `System.address` must be declared `private` in the package `System` (file `system.ads`). Otherwise, your verification might fail with the following error:

```
Verifying _pst_main
Verifying mypackage
mypackage.ada, line xx (column yy): Error: ambiguous bounds in discrete range
Warning: Failed compilation of mypackage
```

Solution/Workaround

Rerun the verification with the following options:

```
-OS-target gnat -D PST_GNAT_SYSTEM_ADDRESS_TYPE_IS_PRIVATE
```

Error from Special Characters

Issue

Your file or folder names contain extended ASCII characters, such as accented letters or Kanji characters. You face file access errors during analysis. Error messages you might see include:

- `No source files to analyze`
- `Control character not valid`
- `Cannot create directory` *`Folder_Name`*

Cause

Polyspace does not fully support these characters. If you use extended ASCII in your file or folder names, your Polyspace analysis may fail due to file access errors.

Workaround

Change the unsupported ASCII characters to standard US-ASCII characters.

Verification Time Considerations

In relation to the verification time, consider the following factors:

- Size of the code
- Number of global variables
- Nesting depth of the variables (the more nested the variables are, the longer the verification takes)
- Depth of the application call tree
- “Intrinsic complexity” of the code, particularly the arithmetic manipulation

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 **Output Summary** and **Dashboard** tabs in the user interface.

For *server* verifications, use the Polyspace Job Monitor to follow the progress of your verification.

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.

For more information, see:

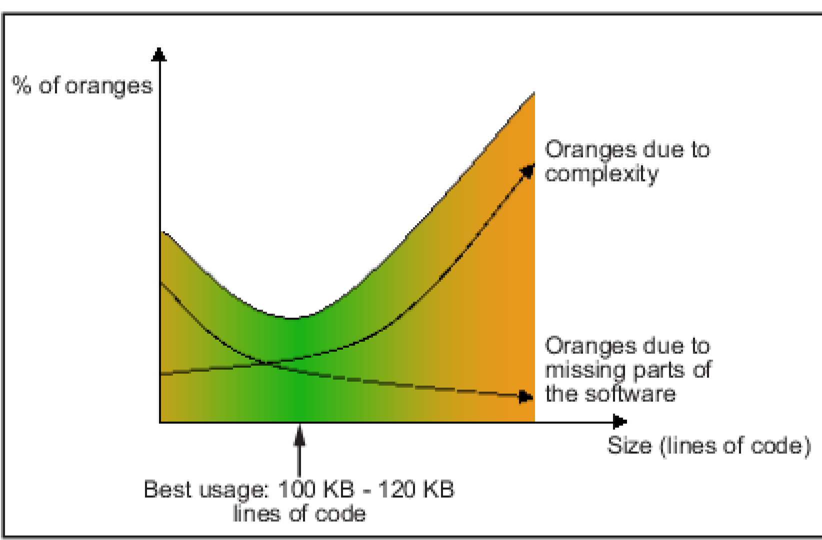
- Client verification: “Monitor Progress” on page 6-2
- Server verification: “Monitor Progress” on page 6-4

Ideal Application Size

There is a compromise between the time and resources required to verify an application, and the resulting selectivity. The larger the project size, the broader the approximations made by Polyspace. These approximations enable Polyspace to extend the range of project sizes that it can manage and to solve incomputable problems. You must balance the benefits from verifying the whole of a large application against the resulting loss of precision.

Begin with package by package verifications. The maximum recommended application size is 100,000 lines of code.

Subdividing an application prior to verification typically has a beneficial impact on selectivity—that is, more red, green, and gray checks, fewer orange warnings, and therefore more efficient bug detection.



A compromise between selectivity and size

Optimum Size

Polyspace software verifies numerous applications with greater than 100,000 lines of code. However, as project sizes become very large, the Polyspace Server:

- Makes broader approximations, producing more orange checks.
- Can take much more time to verify the application.

Before you use another form of testing, use the Polyspace software early on in the development process.

When a small module (file, piece of code, package) is verified using Polyspace, focus on the red and gray checks. **Orange** unproven checks at this stage are very useful, because most of them deal with robustness of the application. The checks change to red, gray, or green as the project progresses and more and more modules are integrated.

During the integration process, the code might become so large (100,000 lines of code or more) that the verification of the whole project is not achievable within a reasonable amount of time. You have several options:

- Keep using Polyspace only upstream in the process.
- Verify subsets of the code.
- Use the `-unit-by-unit` option, as described in "Subdivide According to Files" on page 7-23.

Selecting a Subset of Code

If a project is subdivided into logical sections by considering data flow, the total verification time is shorter than for the project considered in one pass. (See also “Volatile Variables”and “Automatic Stubbing” on page 5-6.)

In such an application, consider the following:

- Function entry points — Refer to the Polyspace execution model because the function entry points are started concurrently, without assumptions regarding sequence or priority. They represent the beginning of your call tree.
- Data entry points — Examine the lines in the code where data is acquired as “data entry points”

Consider the following examples.

Example 7.1. Example 1

```
Procedure complete_treatment_based_on_x(input : integer) is
begin
    thousand of line of computation...
end
```

Example 7.2. Example 2

```
procedure main is
begin
    x:= read_sensor();
    y:= complete_treatment_based_on_x(x);
end
```

Example 7.3. Example 3

```
REGISTER_1: integer;
for REGISTER_1 use at 16#1234abcd#;
procedure main is
begin
    x:= REGISTER_1;
    y:= complete_treatment_based_on_x(x);
end
```

In each example, the `x` variable is a data entry point, and `y` is the consequence of a data entry point. `y` may be formatted data, due to a very complex manipulation of `x`.

Because `x` is volatile, `y` contains all possible formatted data. You can completely remove the procedure `complete_treatment_based_on_x` and let automatic stubbing work. It then assigns a full range of data to `y` directly.

```
-- removed body of complete_treatment_based_on_x
procedure main is
begin
x:= ... -- what ever;
y:= complete_treatment_based_on_x(x); -- now stubbed!
end
```

Results

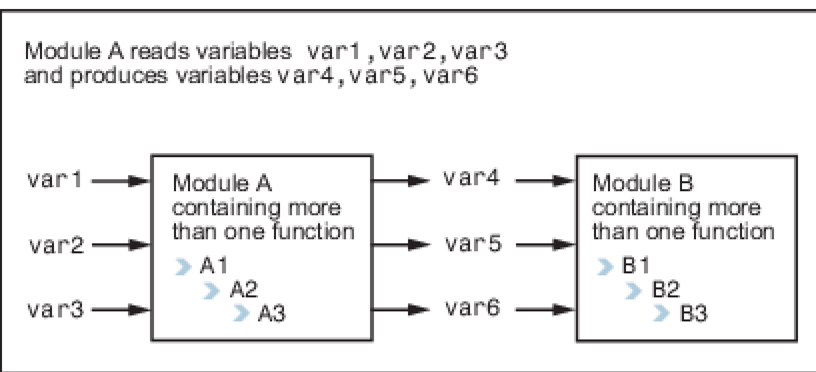
- (–) A slight loss of precision on `y`. Polyspace considers all possible values for `y`, including the formatted values present at the first verification.
- (+) A huge investigation of the code is not required to isolate a meaningful subset.
- (+) Functional modules are not lost.
- (+) The results are still valid, because you do not have to remove a thread that uses shared data.
- (+) The complexity of the code is considerably reduced.
- (+) A high precision level (for example, `O2`) can be maintained.

Examples of Removable Components

- **Error management modules**. Contain a large array of structures that are accessed through an API, but return only a Boolean value. By removing the API code and retaining the prototype, the automatically generated stub is assumed to return a value in the range `[-2^31, 2^31-1]`, which includes `1` and `0`. The procedure is considered to return the full range of possible results
- **Buffer management for mailboxes coming from missing code**. Suppose an application reads a huge buffer of 1024 char, and then uses it to populate three small arrays of data, using a very complicated algorithm before passing it to the main module. If the buffer is excluded from the verification and the arrays are initialized with random values instead, the verification of the remaining code is unaffected.

Subdivide According to Data Flow

Consider the following example.



In this application, `var1`, `var2`, and `var3` can vary between the following ranges.

`var1`	Between 0 and 10
`var2`	Between 1 and 100
`var3`	Between -10 and 10

Specification of Module A:

Module A consists of an algorithm that interpolates between `var1` and `var2`. That algorithm uses `var3` as an exponential factor. When `var1` is equal to 0, the result in `var4` is also equal to 0.

As a result, `var4`, `var5`, and `var6` are produced with the following specifications.

Ranges	`var4var5var6`	Between -60 and 110Between 0 and 12Between 0 and 100
Properties	A set of properties between variables	For example: • 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 modules `A` and `B` to be verified 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]`.

Results:

- (–) A slight loss of precision on the `B` module verification, because now the combinations for `var4`, `var5`, and `var6` are restricted by the `A` module verification.
- For instance, if the `B` module included the test:

  ```
  If var2 is equal to 0, then var4 > var5 > 5
  ```

  then the dead code on subsequent `else` clauses are undetected.
- (+) An in-depth investigation of the code is not required to isolate a meaningful subset.
- (+) The results remain valid, because you do not have to remove a thread that changes shared data.
- (+) The complexity of the code is reduced by a significant factor.
- (+) The maximum precision level can be retained.

Examples of removable components:

- Error management modules. A function `has_an_error_already_occurred` might return `TRUE` or `FALSE`. Such a module may contain a big array of structures that are accessed through an API. The removal of the API code with the retention of the prototype results in the Polyspace verification producing a stub which returns `[-2^31, 2^31-1]`, including 1 and 0. Therefore, the procedure `has_an_error_already_occurred` returns the full range of possible answers, as the code does at execution time.
- Buffer management for mailboxes coming from missing code. Suppose a large buffer of 1024 char is read, and the data is then collated into three small arrays of data using a complicated algorithm. This data is then given to a main module for processing. For the Polyspace Server verification, the buffer can be removed and the three arrays initialized with random values.
- Display modules.

Subdivide According to Real-Time Characteristics

Another way to split an application is to isolate files that contain only a subset of tasks and to verify each subset separately.

If a verification is initiated using only a few tasks, Polyspace Server loses information regarding the interaction between variables.

Suppose an application involves tasks T1 and T2, and a variable x.

If T1 modifies x, and T2 is scheduled to read x at a particular moment, subsequent operations in T2 are impacted by the values of x.

As an 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. There are two ways to achieve a sound standalone verification of T2:

- x can be declared as volatile to take into account all possible executions. Otherwise, x takes only its initial value or x remains constant, and T2 verification is a subset of possible execution paths. You might have precise results, but for only one *scenario* among possible states for the variable x.
- x can be initialized to the whole possible range [10;15], and then the T2 entry point called.

Subdivide According to Files

Extract a subset of files and perform a verification, in one of three ways:

- Use entry points.
- Create a main that calls randomly those functions that are not called by other functions within this subset of code.
- Relaunch your verification using the -unit-by-unit option. (For more information, see Verify files independently.)

When you want to find red errors and bugs in gray code, this method can produce good results.

Benefits of Methods

You might want to split the code:

- To reduce the verification time for a particular precision mode.
- To reduce the number of oranges (for details, see the following sections).

The problems that subdivision may create are:

- Orange checks from a lack of information regarding the relationship between modules, tasks, or variables.
- Orange checks from using too wide a range of values for stubbed functions.

When the Application is Incomplete

When the code consists of a small subset of a larger project, a lot of procedures are automatically stubbed. Automatic stubbing is done according to the specification or prototype of the missing functions. Therefore Polyspace assumes that all possible values for the parameter type can be returned.

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. The number of incidences of these “data set issue” orange checks can be reduced by precise stubbing.

Now consider a procedure `f` that modifies its input parameters `a` and `b`, both of which are passed by reference. Suppose that `a` might be modified to a value between 0 and 10, and `b` might be modified to a value between -10 and 10. In an automatically stubbed function, the combination `a` = 10 and `b` = 10 is possible, even though it might not be possible with the real function. This approach can introduce orange checks in a code snippet such as `1/(a*b - 100)`, where the division would be orange.

- Even where precise stubbing is used, verifying a small part of an application might introduce extra orange checks. However, the net result from reducing the complexity is to reduce the total number of orange checks.
- When using the default stubbing, the increase in the number of orange checks is more pronounced.

Application Code Size

Polyspace can make approximations when computing the possible values of variables in your code. Such an approximation uses a superset of the actual possible values.

For example, in a relatively small application, the Polyspace Server might 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 `VAR` is used to as a divisor, the division is green (because 0 is not a possible value).

If the program being verified is large, the Polyspace Server simplifies the internal data representation 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 on in the verification, the Polyspace Server might further simplify the `VAR` range to `[-2 ; 25]`.

When the size of the program becomes large, this phenomenon leads to the increase of the number of orange warnings.

Note The amount of simplification applied to the data representations also depends on the required precision level (`O0`, `O2`). The Polyspace Server adjusts the level of simplification, for example:

- `-O0` — Shorter computation time
- `-O1` — Fewer orange warnings
- `-O2` — Default and high-precision results
- `-O3` — Fewer orange warnings and longer computation time

Obtaining Configuration Information

Use the `polyspace-ada -ver` command to quickly gather information about your system configuration. You require this information when entering support requests.

Configuration information includes:

- Hardware configuration
- Operating system
- Polyspace licenses
- Specific version numbers for Polyspace products
- Installed Bug Report patches

To obtain configuration information, use the following command:

```
Polyspace_Install/polyspace/bin/polyspace-ada -ver
```

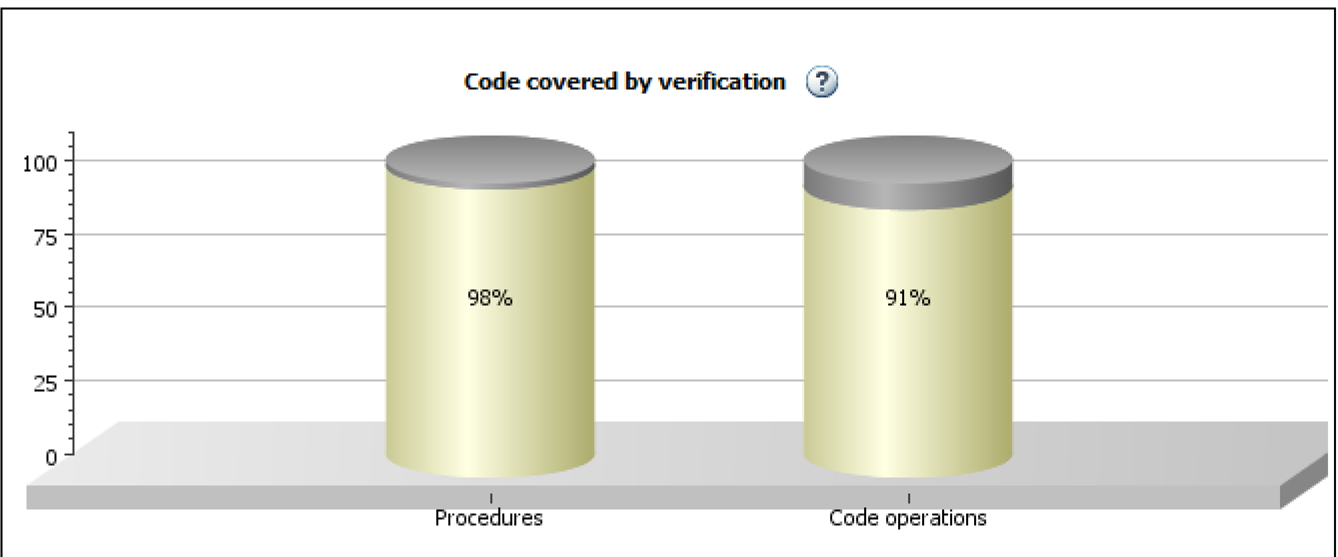
Note You can obtain the same configuration information by selecting **Help > About** in the Polyspace verification environment.

Reasons for Unchecked Code

Issue

After verification, you see in the **Code covered by verification** graphs that a significant portion of your code has not been checked for run-time errors.

For instance, in the following graph, the **Dashboard** pane shows that 2% of your procedures have not been checked for run-time errors. (In the procedures that were checked, 9% of operations have not been checked.)



The unchecked code percentage in the **Code covered by verification** graph covers:

- Procedures and operations that are not checked because they are proven to be unreachable.

  They appear gray on the **Source** pane.

```
function myabs (x : integer) return integer is
begin
   if (x >= 0) then
      return (X);
   else
      return (-X);
   end if;
end myabs;
```

- Functions and operations that are not proven unreachable but not checked for some other reason.

  They appear black on the **Source** pane.

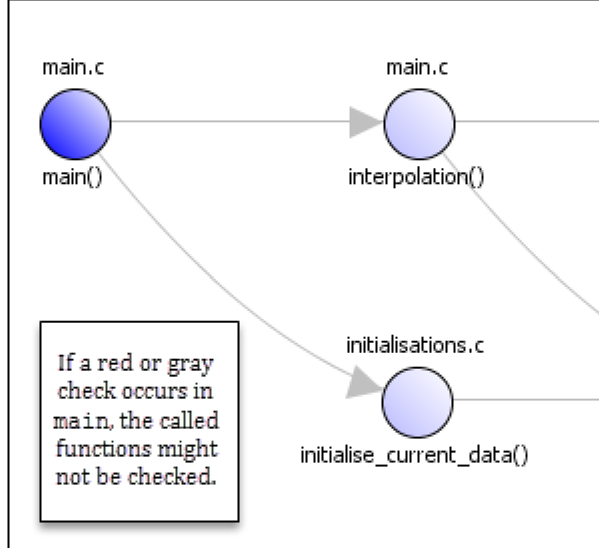
```
function Return_Code(Y : in Integer) return Integer is
begin
   return (2*Y);
end Return_Code;
```

Possible Cause: Early Red or Gray Check

You have a red or gray check towards the beginning of the function call hierarchy. Red or grey checks can lead to subsequent unchecked code.

- Red check: The verification does not check subsequent operations in the block of code containing the red check.
- Gray check: Gray checks indicate unreachable code. The verification does not check operations in unreachable code for run-time errors.

If you call functions from the unchecked block of code, the verification does not check those functions either. If you have a red or gray check towards the beginning of the call hierarchy, functions further on in the hierarchy might not be checked. You end up with a significant amount of unchecked code.



Solution

See if the `main` procedure or another entry point function has red or gray checks. See if you call most of your functions from the subsequent unchecked code.

To navigate from the `main` down the function call hierarchy and identify where the unchecked code begins, use the navigation features on the **Call Hierarchy** pane. If you do not see the pane by default, select **Window > Show/Hide View > Call Hierarchy**. For more information, see “Call Hierarchy” on page 8-13.

Alternatively, you can consider an arbitrary unchecked function and investigate why it is not checked. See if the same reasoning applies for many functions.

Review the red or gray checks and fix them. See “Review Red Checks” on page 8-18 and “Review Gray Checks” on page 8-20.

Possible Cause: Incorrect Options

You did not specify the necessary analysis options. When incorrectly specified, the following options can cause unchecked code:

- Multitasking options: If you are verifying multitasking code, through these options, you specify your entry point functions. You might not have specified all your entry points.
- Inputs and stubbing options: Through these options, you constrain variable ranges from outside your code or force stubbing of functions.

  Possible errors in specification include:

  - You specified variable ranges that are too narrow causing otherwise reachable code to become unreachable.
  - You stubbed some functions unintentionally.

Solution

Check your options in the preceding order. If your specifications are incorrect, fix them.

Storage of Temporary Files

If you specify the option `-tmp-dir-in-results-dir`, Polyspace does not use the standard `/tmp` or `C:\Temp` folder to store temporary files. Instead, Polyspace uses a subfolder of the results folder. If the results folder is mounted on a network drive, this action can increase verification time. Use this option only when the temporary folder partition is not large enough and you need to troubleshoot.

You can specify `-tmp-dir-in-results-dir` through a line command or the **Configuration** > **Advanced Settings** > **Extra Settings** field.

Disk Defragmentation and Antivirus Software

If a disk defragmentation tool or antivirus software runs on the machine on which your client or server verification is running, the verification might fail, generating an error message like the following:

```
Some stats on aliases use:
  Number of alias writes:      22968
  Number of must-alias writes: 3090
  Number of alias reads:       0
  Number of invisibles:        949
Stats about alias writes:
  biggest sets of alias writes: foo1:a (733), foo2:x (728), foo1:b (728)
  procedures that write the biggest sets of aliases: foo1 (2679), foo2 (2266), foo3 (1288)
**** C to intermediate language translation - 17 (P_PT) took 44real, 44u + 0s (1.4gc)
exception SysErr(OS.SysErr(name="Directory not empty", syserror=notempty)) raised.
unhandled exception: SysErr: No such file or directory [noent]


-------------------------------------------------------------------------
---                                                                                   ---
---  Verifier has encountered an internal error.                    ---
---  Please contact your technical support.                         ---
---                                                                                   ---
--------------------------------------------------------------------------
```

On your machine, you must do the following:

- Stop the disk defragmentation tool.
- Deactivate the antivirus software, or configure exception rules for the antivirus software that allow Polyspace to run without failure.

Out-of-Memory Errors During Report Generation

During generation of very large reports, the software might produce errors that indicate insufficient memory. For example:

```
Exporting views...
Initializing...
Polyspace Report Generator
Generating Report
 .....
    Converting report
Opening log file:  C:\Users\auser\AppData\Local\Temp\java.log.7512
Document conversion failed
.....
Java exception occurred:
java.lang.OutOfMemoryError: Java heap space
```

To increase the Java® heap size, modify the `-Mx` option in the *`Polyspace_Install`*`\polyspace\bin\`*`architecture`*`\java.opts` file. By default, the heap size is set to 512 MB. For 32-bit machines, you can increase the size to 1 GB. For 64-bit machines, you can specify a higher value, for example, 2 GB.

Reviewing Verification Results

- "Polyspace Check Colors" on page 8-2
- "Verification Following Red and Orange Checks" on page 8-3
- "Project and Results Folder Contents" on page 8-5
- "Result Views in Polyspace User Interface" on page 8-6
- "Why Review Dead Code Checks" on page 8-16
- "Review Red Checks" on page 8-18
- "Review Gray Checks" on page 8-20
- "Review Orange Checks" on page 8-21
- "Review Global Variable Usage" on page 8-24
- "CWE Coding Standard and Polyspace for Ada Results" on page 8-25
- "Add Review Comments to Results" on page 8-27
- "Justify Results Through Code Annotations" on page 8-30
- "Define Custom Annotation Format" on page 8-36
- "Annotation Description Full XML Template" on page 8-44
- "Add Review Comments to Code" on page 8-49
- "Filter and Group Results" on page 8-52
- "Prioritize Check Review" on page 8-54
- "Generate Report" on page 8-55
- "Export Results to Text File" on page 8-58
- "Export Global Variable List" on page 8-60
- "Customize Report Templates" on page 8-62
- "Set Character Encoding Preferences" on page 8-65

Polyspace Check Colors

Polyspace software presents verification results as colored entries in the source code. There are four main colors in the results:

- **Red** – Indicates code proven to contain an error
- **Gray** – Indicates unreachable code (dead code).
- **Orange** – Indicates unproven code (code might have a run-time error).
- **Green** – Indicates code proven not to have a run-time error

When reviewing verification results, remember these rules:

- An instruction is verified only if no run-time error is proven to occur in the previous instruction.
- The verification assumes that each run-time error causes a “core dump”. The corresponding instruction is considered to have stopped, even if the actual run-time execution of the code might not stop. With orange checks, only the green parts propagate through to subsequent checks.
- Focus on the message produced by the verification, and do not jump to false conclusions. You must understand the color of a check step by step, until you find the root cause of the issue.
- Determine the cause by examining the actual code. Do not focus on what the code is supposed to do.

Verification Following Red and Orange Checks

Verification Following Red and Orange Checks

Polyspace considers that all execution paths that contain a run-time error terminate at the location of the error. Therefore:

- Following a red check, Polyspace does not analyze the remaining code in the same scope as the check.
- Following an orange check, Polyspace analyzes the remaining code. But it considers only the subset of execution paths that did not contain the run-time error.

Use these two rules to understand your checks. The following examples show how the two rules can result in checks that can be misleading when viewed out of context. Understand the examples below thoroughly to practice reviewing checks in context of the remaining code.

In this section...
“Verification Following Red Check” on page 8-3
“Green Check Following Orange Check” on page 8-3
“Gray Check Following Orange Check” on page 8-4

Verification Following Red Check

Consider each line of the procedure `red`, which shows what happens after a **red** check.

```
procedure red is
X: integer;
begin
X:= 1 / X;
X:= X + 1;
end;
```

When Polyspace divides by X, X is not initialized. Therefore, the software generates a **red** NIV check for the non-initialized variable X. Execution paths following this statement are stopped. Checks are not generated for the statement `X:= X + 1;`

Green Check Following Orange Check

Now consider the procedure `propagate`, which shows how **green** checks propagate out of **orange** checks.

```
function read_an_input return integer;
procedure propagate is
X: Integer;
Y: array (0..99) of Integer;;
begin
X:= read_an_input;
Y(X):= 0;
Y(X):= 0;
end main;
```

For the `propagate` procedure:

- X is assigned the value of `read_an_input`. After this assignment, `X = [-2^31, 2^31-1]`.
- At the first array access, an “out of bounds” error is possible because X can be equal to, for example, -3 as well as 3.

- The conditions leading to a run-time error are truncated. They are not considered further in the verification. On the following line, the executions for which `X = [-2^31, -1]` and `[100, 2^31-1]` are stopped.
- At the next instruction, `X = [0, 99]`.
- At the second array access, the check is green because `X = [0, 99]`.

Therefore, green checks can propagate out of orange checks.

Note Through manual stubbing and by using `assert`, you can use value propagation to restrict input values for data.

See "Using Pragma Assert to Set Data Ranges" on page 4-17.

Gray Check Following Orange Check

Consider the following example, paying particular attention to the dead (gray) code following the "if" statement:

```
function Read_An_Input return integer;
procedure Main is
X: Integer;
Y: array (0..99) of Integer;
begin
X := Read_An_input;
Y(X) := 0; -- [array index may be without its bounds] [x is
initialized]
Y(X-1):= (1 / X) + X ; [array index is within its bounds]
if (X = 0) then
Y(X) := 1; -- this line is unreachable
end if;
end Main;
```

You can see that:

- The line containing the access to the `Y` array is unreachable.
- The line is unreachable only if the test for `x = 0` is always false.
- You can conclude that the test is false because the input data is not equal to 0. However, `Read_An_Input` can represent a 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 execution paths leading to a run-time error, meaning that subsequent lines deal with only x = [0, 99].
- The orange check on the division also truncates execution paths that lead to a run-time error, so instances where x = 0 are also stopped. Therefore, for the code execution path after the orange division sign, `x = [1; 99]`.
- x is not equal to 0 at this line. The array access is green (y (x – 1).

Project and Results Folder Contents

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

The organization of Polyspace files in the physical folder location follows the hierarchy displayed in the Polyspace user interface. The project folder contains a subfolder for each module. In each module folder, there is one or more result subfolder, named `Result_#`. The number of result folders depends on whether you overwrite or retain previous results for each new run. To use a different folder naming convention or different storage location for results, see “Specify Results Folder” on page 3-4.

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

Files in the Results Folder

Some of the files and folders in the results folder are described below:

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

See Also

Related Examples

- “Specify Results Folder” on page 3-4

Result Views in Polyspace User Interface

In this section...
"Results List" on page 8-6
"Source" on page 8-8
"Result Details" on page 8-10
"Variable Access" on page 8-11
"Call Hierarchy" on page 8-13
"Concurrency Modeling" on page 8-15

Results List

The **Results List** pane lists all checks along with their attributes. To organize your check review, from the list on this pane, select one of the following options.

- **None**: Lists all checks without grouping them. The checks are sorted in the following order:

  1. **Red**: Indicates code that is proven to contain an error. The check indicates that the code will fail every time it is executed.
  2. **Gray** — Indicates unreachable code.
  3. **Orange** — Indicates unproven code that might contain an error.
  4. **Green** — Indicates code that is proven to not contain an error.
- **Family**: Lists checks grouped by color. Within each color, the checks are organized by group. For more information on the check groups, see "Run-Time Checks".
- **File**: Lists checks grouped by file. Within each file, the checks are grouped by procedure.
- **Package**: Lists checks grouped by package. Within each package, the checks are grouped by procedure.

For each check, the **Results List** pane contains the check attributes, listed in columns:

Attribute	Description
Family	Group to which the check belongs. For instance, if you choose the grouping **File**, this column contains the name of the file and function containing the check.
ID	Unique identification number of the check. In the default view on the **Results List** pane, the checks appear sorted by this number.
Type	Check color
Group	Category of the check. For more information on the checks covered by a group, see the check reference pages.
Check	Description of the error

Attribute	Description
Information	For run-time errors, this attribute indicates whether the check is related to path or bounded input values. For coding rule violations, this attribute indicates whether the rule is `Required`.
File	File containing the instruction where the check occurs
Package	Package containing the instruction where the check occurs
Function	Function containing the instruction where the check occurs.
Line	Line number of the instruction where the check occurs.
Col	Column number of the instruction where the check occurs. The column number is the number of characters from the beginning of the line.
%	Percentage of checks that are not orange. This column is most useful when you choose the grouping `Checks by File/Function`. The entry in this column against a file or function indicates the percentage of checks in the file or function that are not orange.
Severity	Level of severity you have assigned to the check. The possible levels are: • `Unset` • `High` • `Medium` • `Low`
Status	Review status you have assigned to the check. The possible statuses are: • `Unreviewed` • `To investigate` • `To fix` • `Justified` • `No action planned` • `Not a defect` • `Other`
Justified	Check boxes showing whether you have justified the checks
Comments	Comments you have entered about the check

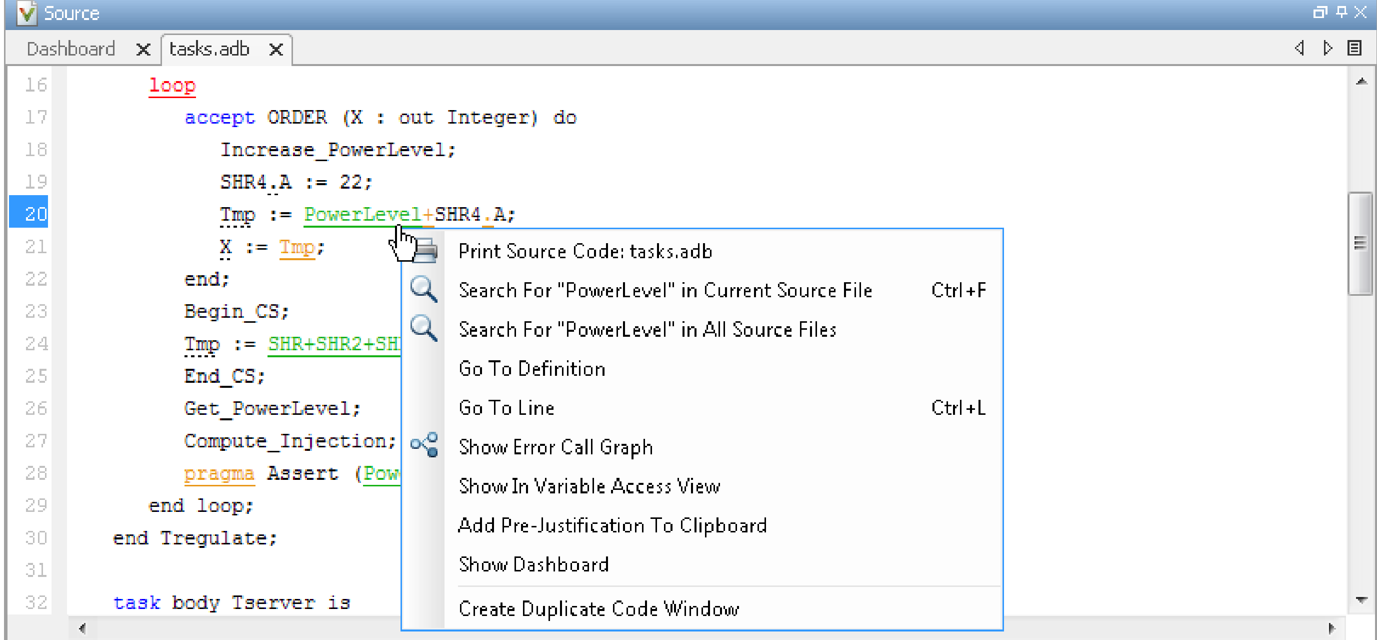
To show or hide a column, right-click anywhere on the column title. From the context menu, select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through checks. For more information, see “Add Review Comments to Results” on page 8-27.
- Organize your check review using column filters. For more information, see “Filter and Group Results” on page 8-52.

Source

The **Source** pane shows the source code with colored checks highlighted.

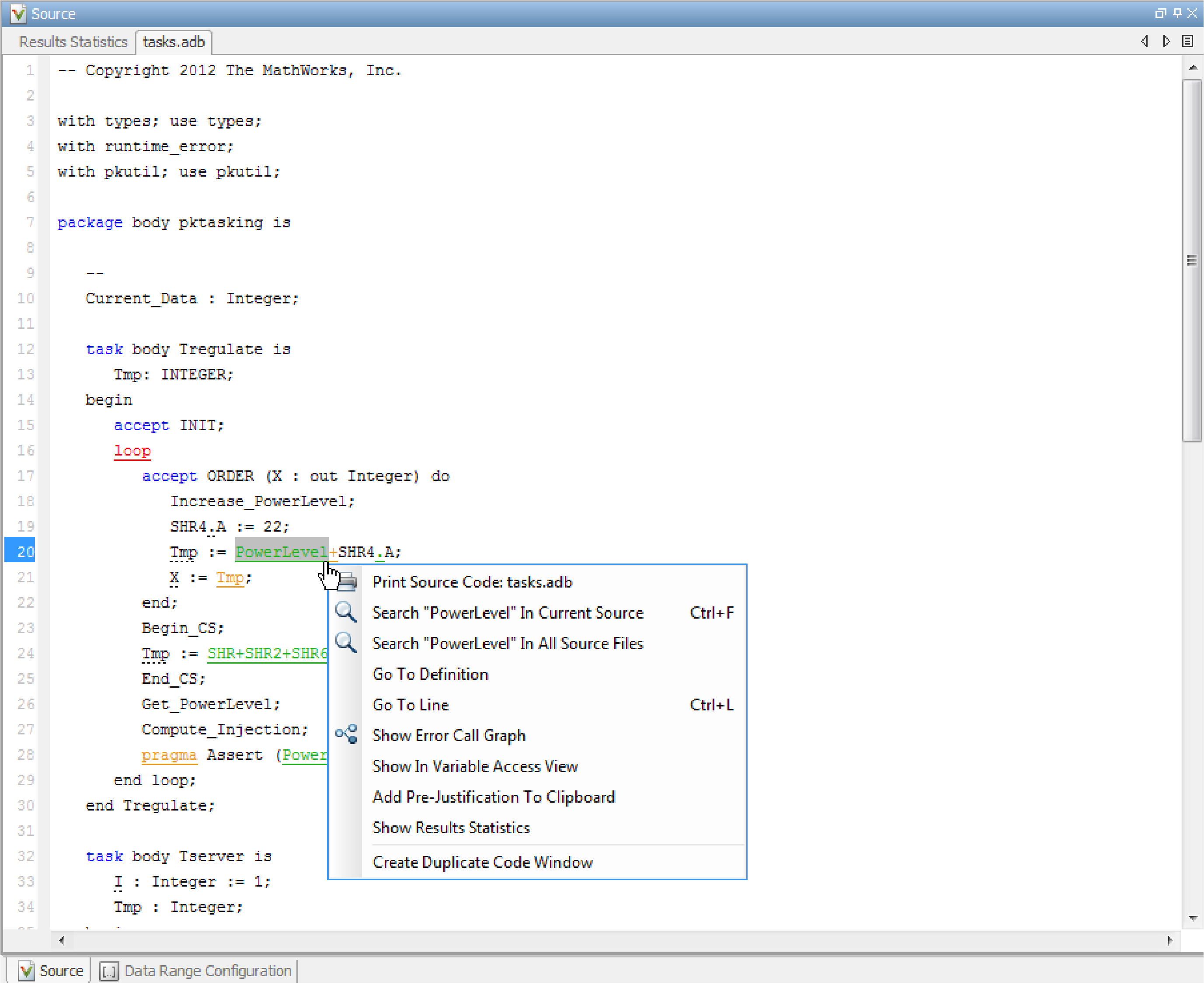


Tooltips

Placing your cursor over a check displays a tooltip that provides range information for variables, operands, function parameters, and return values.

Examine Source Code

In the **Source** pane, if you right-click a text string, the context menu provides options to examine your code. For example, right-click the variable `PowerLevel`:



Use the following options to examine and navigate through your code:

- **Search "PowerLevel" in Current Source** — List occurrences of the string within the current source file in the **Search** pane.
- **Search "PowerLevel" in All Source Files** — List all occurrences of the string in source files. The results appear on the **Search** pane.

  Go To Definition — Go to the line of code that contains the definition of `PowerLevel`. The software supports this feature for global and local variables, functions and types.
- **Go To Line** — Open the Go To Line dialog box. If you specify a line number and click **Enter**, the software displays the specified line of code.
- **Expand All Macros** or **Collapse All Macros** — Display or hide the content of macros in current source file.

Manage Multiple Files in Source Pane

You can view multiple source files in the **Source** pane.

On the **Source** pane toolbar, right-click a tab title to manage source files.

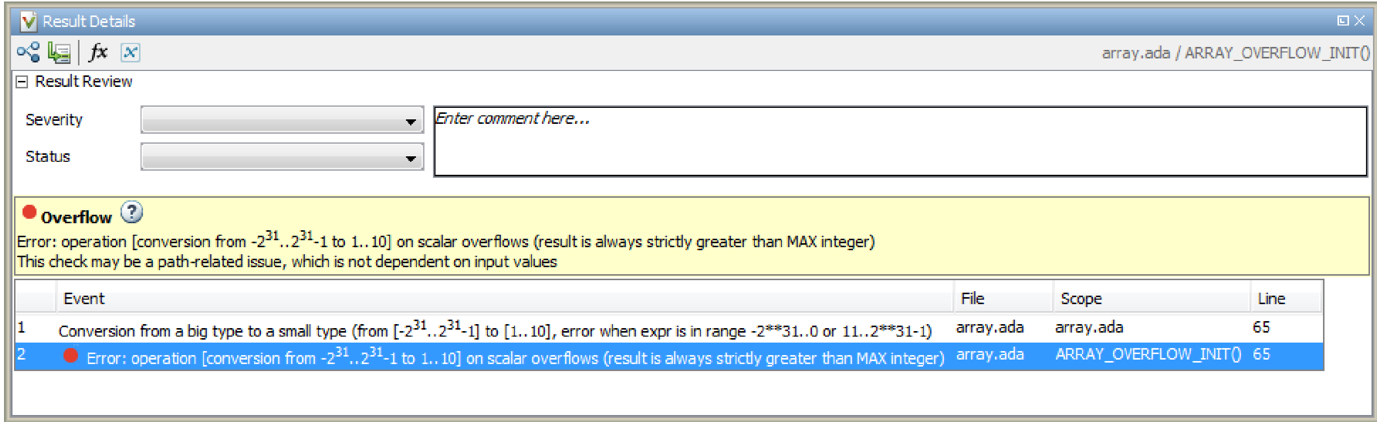
From the **Source** pane context menu, you can:

- **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 view.
- **Previous** – Display the previous view.
- **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.

Result Details

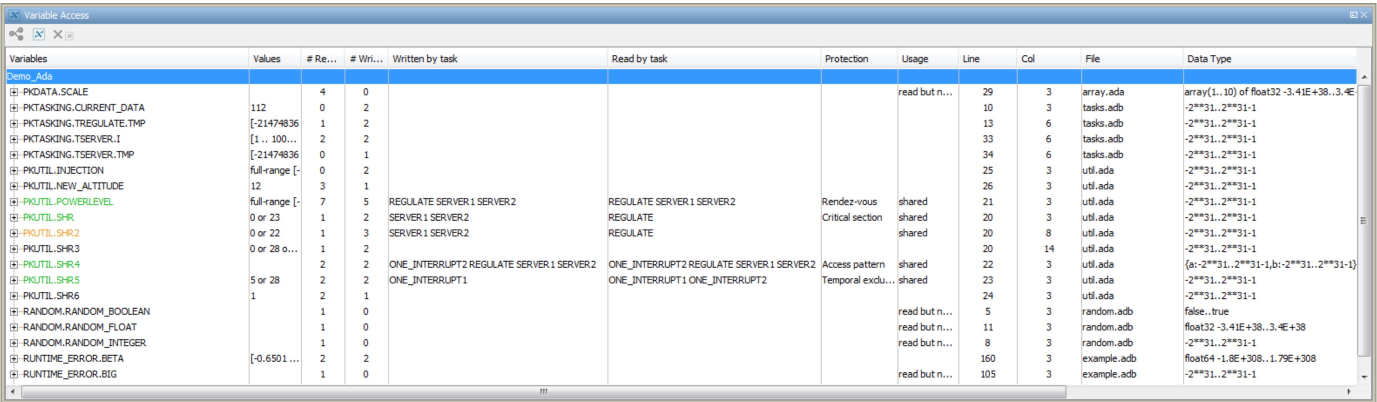
On the **Results List** pane, if you click a check, you see additional information on the **Result Details** pane.

On this pane, you can also assign a **Severity** and **Status** to each check. You can also enter comments to describe the results of your review. This action helps you track the progress of your review and avoid reviewing the same check twice.



Variable Access

The **Variable Access** pane displays global variables. For each global variable, the pane lists functions and tasks performing read/write operation on the variables, along with their attributes, such as values, read/write operations and shared usage.



For each variable and each read/write access, the **Variable Access** pane contains the relevant attributes. For the variables, the various attributes are listed in this table.

Attribute	Description
Variables	Name of Variable, *Package_Name. Variable_Name* *Package_Name*: Name of package where variable is declared
Values	Value (or range of values) of variable
# Reads	Number of times the variable is read
# Writes	Number of times the variable is written
Written by task	Name of tasks writing on variable
Read by task	Name of tasks reading variable

Attribute	Description
Protection	Whether shared variable is protected from concurrent access (Filled only when `Usage` column has entry, `Shared`) The possible entries in this column are: • `Critical Section`: If variable is accessed in critical section of code • `Temporal Exclusion`: If variable is accessed in mutually exclusive tasks For more details on these entries, see “Verification Mode”.
Usage	`Shared`, if variable is shared between tasks; otherwise, blank
Line	Line number of variable declaration
Col	Column number (number of characters from beginning of line) of variable declaration
File	Source file containing variable declaration
Data Type	• If the variable has a scalar data type, this column states the values allowed for the type. • If the variable is an array or a record, this column states the values allowed for the data types of its components.
Detailed Type	Data type of variable, if the variable has a scalar data type.

Double-click a variable name to view read/write access operations on the variable. The arrowhead symbols ▸ and ◂ in the **Variable Access** pane indicate functions performing read and write access respectively on the global variable. Likewise, tasks performing read and write access are indicated by the symbols ||▸ and ◂|| respectively.

For access operations on the variables, the various attributes described in the pane are listed in this table.

Attribute	Description
Variables	Names of procedure (or task) performing read/write access on the variable, *`Package_Name.Procedure_Name`* *`Package_Name`*: Name of package containing procedure (or task) definition
Values	Value or range of values of variable in the procedure or task performing read/write access

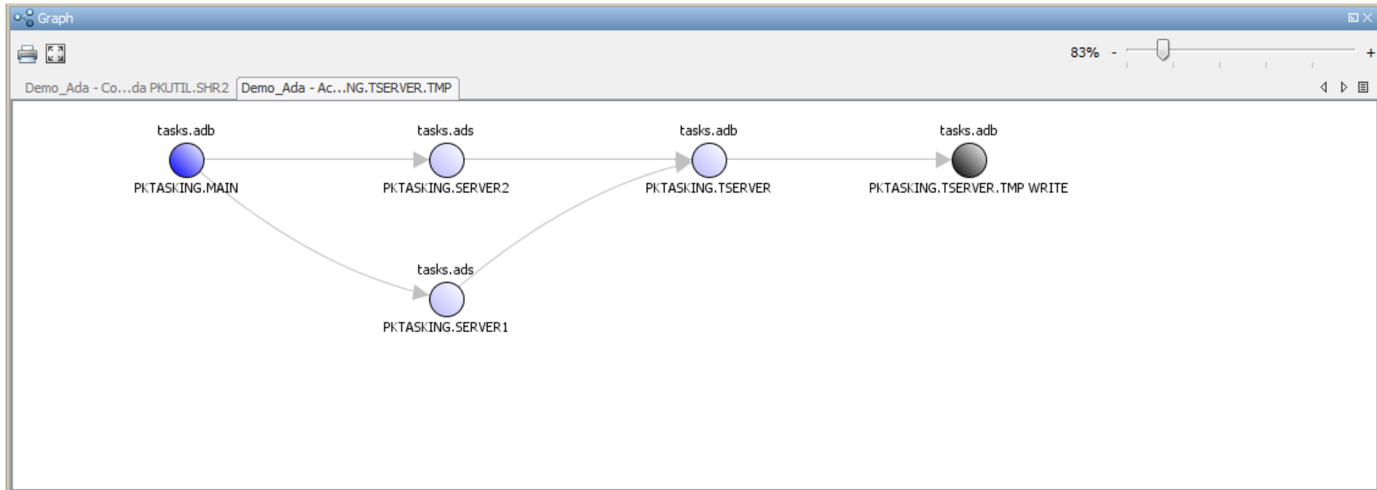
Attribute	Description
Written by task	*Only for tasks*: Name of task performing write access on variable
Read by task	*Only for tasks*: Name of task performing read access on variable
Line	Line number where procedure or task accesses variable
Col	Column number where procedure or task accesses variable
File	Source file containing access operation on variable

You can also perform the following actions from the **Variable Access** pane.

- **View Access Graph**

  View the access operations on a global variable in graphical format using the **Variable Access** pane. Select the global variable and click .

  Here is an example of an access graph:



- **Show/Hide Non-Shared Variables**

  Customize the **Variable Access** pane to show only the shared variables. On the Variable Access pane toolbar, click the Non-Shared Variables button to show or hide non-shared variables.
- **Hide Access in Unreachable Code**

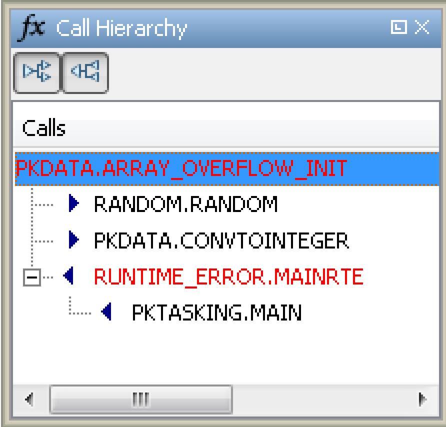
  Hide read/write access occurring in dead code by clicking the filter button .

Call Hierarchy

The **Call Hierarchy** pane displays the call tree of procedures in the source code.

For each procedure, `foo`, the **Call Hierarchy** pane lists the procedures and tasks that call `foo` (callers) and those called by `foo` (callees). The callers are indicated by ◀ (procedures), or ◀|| (tasks). The callees are indicated by ▶ (procedures) or ||▶ (tasks).

In the following example, the **Call Hierarchy** pane displays the procedure, `SORT_CALIBRATION`, in the package, `SENSITIVITY`. It also displays the callers and the callees of `SORT_CALIBRATION`.



Depending on the name, the corresponding line number in the **Call Hierarchy** pane refers to a different line in the source code:

- For a procedure name, the line number refers to the beginning of the procedure definition. In the preceding example, the definition of `SENSITIVITY.SORT_CALIBRATION` begins on line 97.
- For a callee name, the number refers to the line where the callee is called. In the preceding example, callee, `SENSITIVITY.POLYNOMIA`, is called by `SENSITIVITY.SORT_CALIBRATION` on line 108.
- For a caller name, the number refers to the line where the caller calls the procedure. In the preceding example, caller, `RUNTIME_ERROR.MAINRTE`, calls `SENSITIVITY.SORT_CALIBRATION` on line 222.

Tip Select a caller or callee name to navigate to the procedure call in the source code.

You can perform the following actions from the **Call Hierarchy** pane:

- **Show/Hide Callers and Callees**

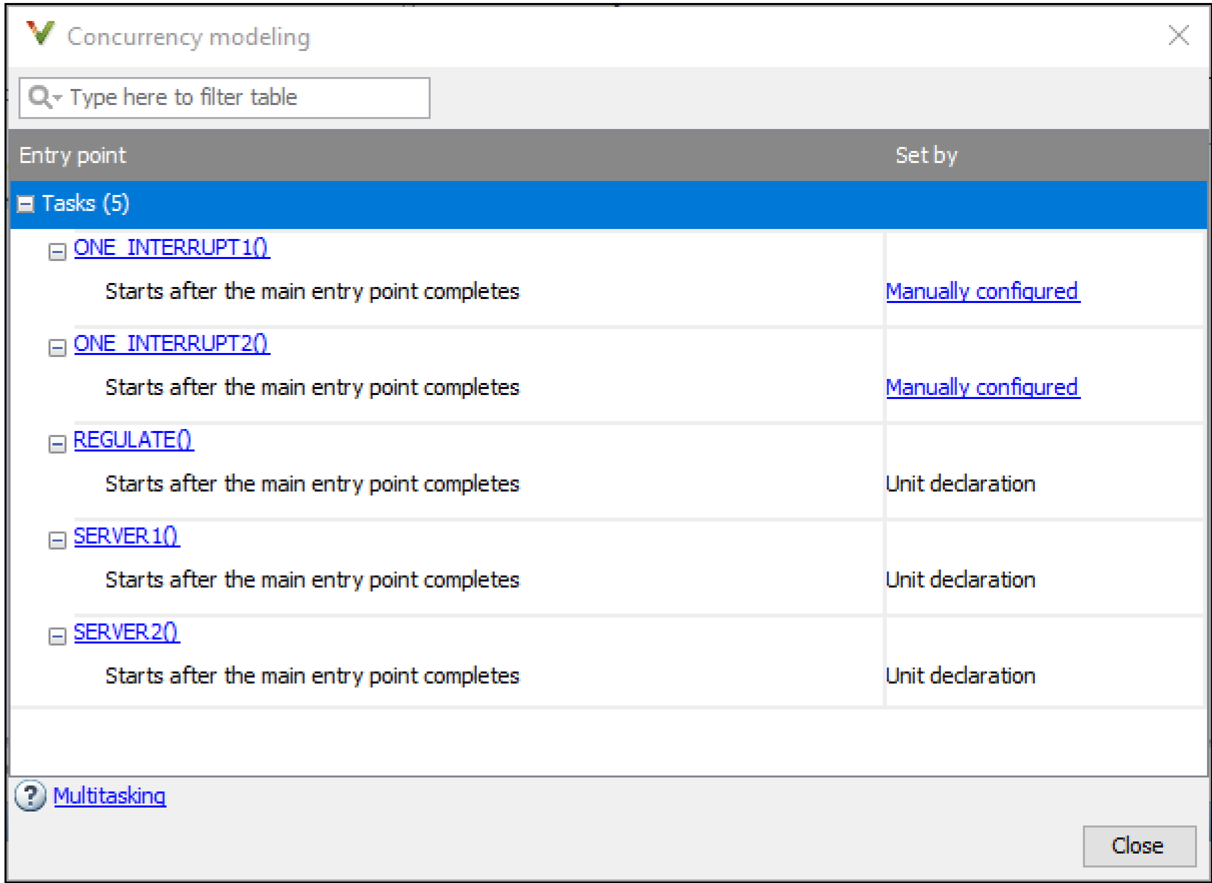
  Customize the view to display callers only or callees only. Show or hide callers and callees by clicking this button

- **Go to Caller/Callee Definition**

  Go directly to the definition of a caller or callee in the source code. Right-click the name of the caller or callee and select **Go To Definition**. .

Concurrency Modeling

The **Concurrency Modeling** view displays all the tasks and interrupts that the analysis extracts from your code and your Polyspace multitasking configuration.



in the table, the functions are listed in the first column by order of decreasing priority. The second column shows how Polyspace detects each task or interrupt that you manually configured in your Polyspace project configuration.

From this view, you can:

- Click a function name to go to its definition in the source code.
- Click **Manually configured**, for functions that are manually configured, to go to the **Multitasking** node on the **Configuration** pane.

Why Review Dead Code Checks

In this section...
"Functional Bugs in Gray Code" on page 8-16
"Structural Coverage" on page 8-16

Functional Bugs in Gray Code

Polyspace verification finds different types of dead code. Common examples include:

- Defensive code
- 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 an unreachable branch of a conditional statement.
- An unreachable "else" clause where the wrong variable is tested in the "if" statement.
- A variable that is supposed to be local to the file but instead is local to the function.
- Wrong variable prototyping leading to a comparison which is always false.

The consequences of dead code and the effort to deal with it is unpredictable. From a one-week effort of functional testing on target, trying to build a scenario going into that branch, and wondering why the functional behavior is altered to a three-minute code review discovering the bug.

Polyspace does not measure the impact of dead code.

The tool provides a list of dead code. A short code review enables you to place each entry from that list into one of the five categories. Doing so identifies known dead code and uncovers real bugs.

The Polyspace experience is that at least 30% of gray code reveals real bugs.

Structural Coverage

Polyspace software performs upper approximations of possible executions. Therefore, even if a line of code is shown in green, there remains a possibility that it is a dead portion of code. Because

Polyspace verification made an upper approximation, it could not conclude that the code was dead. Instead it concludes that a run-time error could not be found.

Polyspace verification finds around 80% of dead code that the developer would find by doing structural coverage.

Polyspace verification is intended to be a productivity aid in dead code detection. It detects dead code which might take days of effort to find by other means.

Review Red Checks

During verification, Polyspace checks each operation in your code for certain run-time errors. After verification, the software displays the checks on the **Results List** pane.

A red check indicates that the operation fails the check on all execution paths. For instance, a red **Division by Zero** check on a division operation indicates that a division by zero occurs every time the operation takes place. Therefore, you must fix the code containing a red check.

In this section...
“Step 1: Interpret Check Information” on page 8-18
“Step 2: Determine Root Cause of Check” on page 8-18

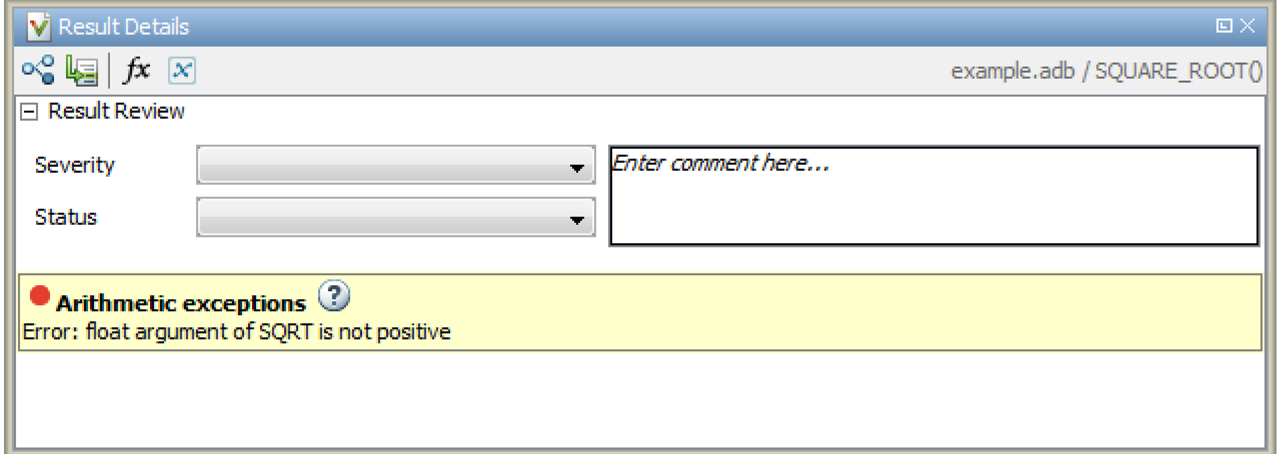
Step 1: Interpret Check Information

Select a check on the **Results List** pane.

- On the **Result Details** pane, view further information about the check.
- On the **Source** pane, the operation containing the check is highlighted.

  If you place your cursor on the operation, the tooltip provides further information about the check.

Sometimes, this information is enough to understand the root cause of the check. If you can determine a fix for your code from this information, you do not have to proceed further with this procedure.



Step 2: Determine Root Cause of Check

If you cannot determine the root cause based on the check information, using navigation shortcuts in the user interface, navigate to the root cause.

1. Using the tooltips on variables or operations, identify the variable `var` that causes the check. For instance, for a **Division by Zero** error, `var` can be the denominator variable.
2. Trace the data flow for `var`.

a Browse through the previous instances of `var`. On the **Source** pane, place your cursor on each instance of `var` to see its values.

<table>
<tr><th>Variable</th><th>How to trace data flow</th></tr>
<tr><td>Local Variable</td><td>Right-click the variable. Select Search For varname in Current Source File or Search For varname in All Source Files.</td></tr>
<tr><td>Global Variable

Right-click the variable. If the option Show In Variable Access View appears, the variable is a global variable.</td><td>i Select the right-click menu option Show In Variable Access View.

The current instance of the variable is shown on the Variable Access pane.
ii Select the previous instances of the variable on this pane.

Write operations on the variable are indicated with ◂ and read operations with ▸.

Tip On the Variable Access pane, drag the Line column to the left. You can then easily see the line numbers during navigation.</td></tr>
<tr><td>Procedure argument

<code>procedure func(..,arg: in float,..) is</code>
<code>.</code>
<code>.</code>
<code>end func;</code></td><td>i On the Result Details pane, select the fx button.

On the Call Hierarchy pane, you see the calling functions indicated with ◂.
ii Select a calling function name. You go to the call to <code>func</code> in your source.
iii Identify the variable in the call to <code>func</code> that maps to <code>arg</code>. This variable is your new variable to trace back.

Tip On the Call Hierarchy pane, drag the Line column to the left. You can then easily see the line numbers during navigation.</td></tr>
<tr><td>Function return value

<code>ret := func();</code></td><td>i Find the function definition.

Right-click <code>func</code> on the Source pane. Select Go To Definition, if the option exists.
ii In the definition of <code>func</code>, identify each <code>return</code> statement. The variable that the function returns is your new variable to trace back.</td></tr>
</table>

b Find the instance where `var` acquires the value that can cause the run-time error.

3 If `var` obtains values from another variable, trace the data flow for the second variable.

Continue this process until you identify the root cause of the check.

Review Gray Checks

Gray checks indicate code that cannot be reached during run-time.

If the gray check indicates defensive code, ignore the check. For instance, you can have error handling tests in your code. If the errors do not occur, the test blocks appear gray. However, you might want to retain the error handling test.

In some cases, unreachable code results from coding errors. Therefore, you must review the gray checks. Also, if you do not want to retain unnecessary code, review and fix gray checks.

Note Following a red check, Polyspace does not verify the remaining code in the same scope as the check. However, this code does not appear gray on the **Source** pane.

Review and fix the red checks so that Polyspace can verify the remaining code. For more information, see “Review Red Checks” on page 8-18.

---

1. After verification, see the code coverage metrics on the **Dashboard** pane.

   The coverage metrics are displayed through the **Code covered by verification** graph. The graph displays:

   - Percentage of code covered by verification.
   - Percentage of procedures covered by verification.
2. If the percentage of procedures covered is less than 100, investigate why there are unreachable procedures. Select the column graph to see the full list of unreachable procedures.
3. Investigate the **Unreachable code** checks further.
4. If you determine that the check represents defensive code, ignore the check. Add a comment and justification in your result or code explaining the rationale.

Review Orange Checks

During verification, Polyspace checks each operation in your code for certain run-time errors. After verification, the software displays the checks on the **Results List** pane.

An orange check indicates that the operation fails the check only on certain execution paths. Investigate whether the execution paths can occur during run time. If you determine that the execution paths can occur, you must fix the code containing the check.

In this section...
"Step 1: Interpret Check Information" on page 8-21
"Step 2: Determine Root Cause of Check" on page 8-21
"Step 3: Trace Check to Polyspace Assumption" on page 8-23

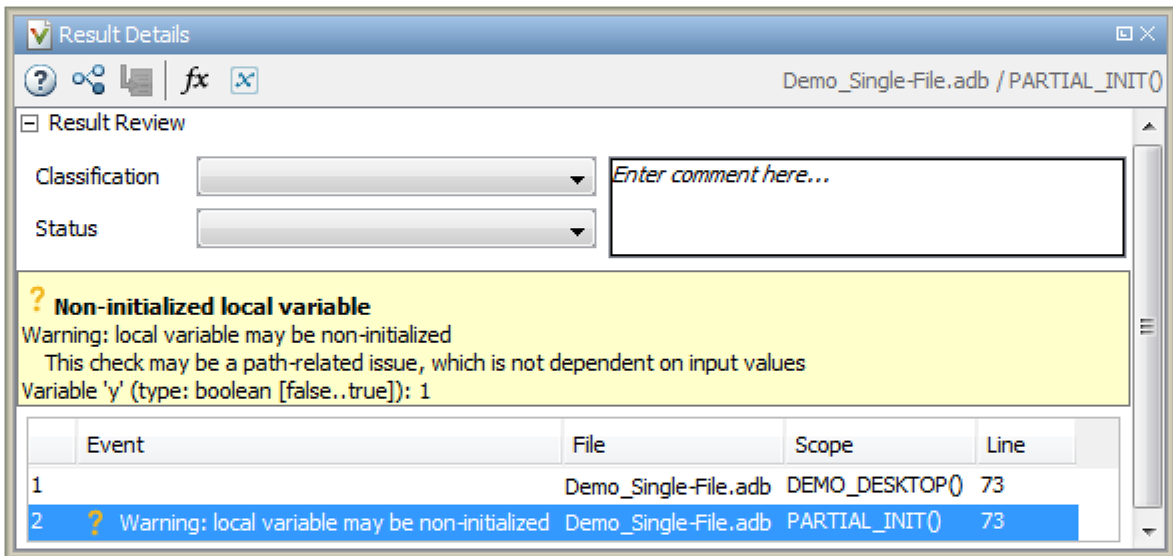
Step 1: Interpret Check Information

Select a check on the **Results List** pane.

- On the **Result Details** pane, view further information about the check.
- On the **Source** pane, the operation containing the check is highlighted.

  If you place your cursor on the operation, the tooltip provides further information about the check.

Sometimes, this information is enough to understand the root cause of the check. If you can determine a fix for your code from this information, you do not have to proceed further with this procedure.



Step 2: Determine Root Cause of Check

If you cannot determine the root cause based on the check information, using navigation shortcuts in the user interface, navigate to the root cause.

1 Using the tooltips on variables or operations, identify the variable `var` that causes the check. For instance, for a **Division by Zero** error, `var` can be the denominator variable.

2 Trace the data flow for `var`.

   a Browse through the previous instances of `var`. On the **Source** pane, place your cursor on each instance of `var` to see its values.

Variable	How to trace data flow
Local Variable	Right-click the variable. Select **Search For *varname* in Current Source File** or **Search For *varname* in All Source Files**.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	i Select the right-click menu option **Show In Variable Access View**. The current instance of the variable is shown on the **Variable Access** pane. ii Select the previous instances of the variable on this pane. Write operations on the variable are indicated with ◀ and read operations with ▶. **Tip** On the **Variable Access** pane, drag the **Line** column to the left. You can then easily see the line numbers during navigation.

Procedure argument
 	
`procedure func(..,arg: in float,..) is`
`.`
`.`
`end func;`
	i On the **Result Details** pane, select the *fx* button.
	
	On the **Call Hierarchy** pane, you see the calling functions indicated with ◀.
	ii Select a calling function name. You go to the call to `func` in your source.
	iii Identify the variable in the call to `func` that maps to `arg`. This variable is your new variable to trace back.
	
	Tip On the **Call Hierarchy** pane, drag the **Line** column to the left. You can then easily see the line numbers during navigation.

Function return value `ret := func();`	i Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. ii In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

   b Find the instance where `var` acquires the value that can cause the run-time error.

3. If `var` obtains values from another variable, trace the data flow for the second variable.

   Continue this process until you identify the root cause of the check.
4. For orange checks, you have an additional option that helps with root cause investigation. If a function is called several times and an error occurs only on certain calls, you can identify which function call caused the check. For more information, see `Sensitivity context`.

Step 3: Trace Check to Polyspace Assumption

If you cannot determine a coding error, try to trace the check to a Polyspace assumption earlier in the code. If the assumption is broader than what you expect, do one of the following:

- If you can use an analysis option to relax the assumption, rerun verification using that option.

  In particular, determine if you must specify constraints outside your code or provide other contextual information. See “Inputs & Stubbing”.
- See if you can improve your coding design to avoid the assumption.

  For instance, `goto` statements interrupt the flow and can cause orange checks during verification. Avoid `goto` statements in your code.

  To improve your coding design:

  - Enforce limits on code complexity metrics.
  - Observe coding rules.
- Ignore the orange check. Add a comment and justification in your result or code describing why you ignored the check.

Review Global Variable Usage

After verification, Polyspace displays a list of global variables in your source code. Using this list:

- You can remove variables that you define but do not use.

  Such variables appear gray on the **Results List** and **Source** pane.
- For code intended for multitasking, you can see which variables are not protected from concurrent access by multiple tasks.
  - If Polyspace proves that a variable is protected, it appears green on the **Results List** and **Source** pane.
  - Otherwise, it appears orange.

For more information, see “Global Variables”.

To review global variable usage:

1. On the **Results List** pane, from the list, select **Family**.

   The global variables appear together under one node.
2. Expand the **Global Variable** node. Review each result under the nodes:
   - **Shared** > **Potentially unprotected variable**.
   - **Not shared** > **Unused non-shared variable**.
3. For each potentially unprotected variable, select the variable name.
   1. On the **Result Details** pane, view which tasks can access the variable.
   2. Read and write operations on the variable appear in this pane. Select each operation to navigate to it in your source code.

      This action also displays more details of the operation on the **Variable Access** pane.
   3. To review your multitasking options, select **Window** > **Show/Hide View** > **Configuration**.

      Identify whether you can leverage some of the existing protection mechanisms to protect your variable. For more information on multitasking verification, see `Critical section details` or `Temporally exclusive tasks`.

CWE Coding Standard and Polyspace for Ada Results

Common Weakness Enumeration (CWE) is a dictionary of common software weakness types that can occur in software architecture, design, code, or implementation. These weaknesses can lead to security vulnerabilities.

CWE and Polyspace for Ada

The CWE dictionary assigns a unique identifier to each software weakness type. These identifiers serve as a common language for describing software security weaknesses and a standard for software security tools targeting these weaknesses. For more information, see Common Weakness Enumeration.

Polyspace for Ada results are mapped to CWE identifiers (IDs). Using the results, you can evaluate your code against the CWE standard. For instance, CWE ID 456 (Missing Initialization of a Variable) maps to the run-time check, **Non-initialized variable** and **Non-initialized local variable**.

For more information on the CWE Compatibility and Effectiveness Program, see CWE Compatibility.

Find CWE IDs from Polyspace Results

The following table lists the CWE IDs (version 2.8) addressed by Polyspace for Ada with its corresponding run-time checks.

CWE ID	CWE ID Description	Polyspace for Ada Check
120	Buffer copy without checking size of input	`Scalar and Float Overflow`
125	Out-of-bounds read	`Correctness Condition`
131	Incorrect calculation of buffer size	`Scalar and Float Overflow`
134	Use of externally-controlled format string	`Correctness Condition`
136	Type errors	`Correctness Condition`
137	Representation errors	`Correctness Condition`
189	Numeric errors	`Power Arithmetic`
190	Integer overflow or wraparound	`Scalar and Float Overflow`
191	Integer underflow (wrap or wraparound)	`Scalar and Float Overflow`
362	Concurrent execution using shared resource with improper synchronization ('race condition')	`Shared unprotected global variable`
366	Race condition within a thread	`Shared unprotected global variable`
369	Divide by zero	`Division by Zero`

CWE ID	CWE ID Description	Polyspace for Ada Check
456	Missing initialization of a variable	Non-Initialized Local Variable Non-Initialized Variable
457	Use of uninitialized variable	Non-Initialized Local Variable Non-Initialized Variable
476	NULL pointer dereference	Correctness Condition
561	Dead code	Unreachable Code
570	Expression is always false	Unreachable Code
571	Expression is always true	Unreachable Code
682	Incorrect calculation	Arithmetic Exceptions Scalar and Float Overflow
835	Loop with unreachable exit condition	Non Terminating Loop

Add Review Comments to Results

This example shows how to comment on results in the Polyspace user interface. When reviewing results, you can assign a status to them, and enter comments to describe the results of your review. These actions help you to track the progress of your review and avoid reviewing the same result twice.

In this section...
"Assign and Save Comments" on page 8-27
"Import Review Comments from Previous Verifications" on page 8-28

Assign and Save Comments

1. On the **Results List** pane, select the result that you want to review.
2. Investigate the result further.

   For more information, see:

   - "Review Red Checks" on page 8-18
   - "Review Gray Checks" on page 8-20
   - "Review Orange Checks" on page 8-21
   - "Review Global Variable Usage" on page 8-24
3. On the **Results List** or **Result Details** pane, provide the following review information for the result:

   - **Severity** to describe how critical you consider the issue.
   - **Status** to describe how you intend to address the issue.

     To justify the check, select one of the **Status** options, `Justified`, `No action planned` or `Not a defect`. You can view the percentage of results justified per file and function. On the **Results List** pane, from the list, select **File**. View the entries on the **Justified** column.

     You can also create your own status or associate justification with an existing status. Select **Tools > Preferences** and create or modify statuses on the **Review Statuses** tab.
   - **Comment** to describe any other information about the result.
4. To provide review information for a group of results, select the results in the group together. Then provide review information for a single result.

   To select the results:

   - If the results are contiguous, left-click the first result. Then **Shift** click the last result.
   - If the results are not contiguous, **Ctrl** click each result.
   - If the results belong to the same group and have the same color, right-click one result. From the context menu, select **Select All *`Color Type`* Results**.

     For instance, select **Select All Orange "Overflow" Results**.
5. To save your review comments, select **File > Save**. Your comments are saved with the verification results.

Import Review Comments from Previous Verifications

- “Import Comments” on page 8-28
- “Specify Automatic Comment Import from Last Verification” on page 8-28
- “View Imported Comments That Do Not Apply” on page 8-28

After you have reviewed verification results, you can reuse your review comments for subsequent verifications.

After you import checks and comments, clicking the icon skips justified checks. Therefore, you do not have to review checks twice.

Import Comments

1. Open your verification results.
2. Select **Tools** > **Import Comments**.
3. Navigate to the folder containing your previous results.
4. Select the results file with extension `.rte` and then click **Open**.

   The review comments from the previous results are imported into the current results, and the Import checks and comments report opens. For information on this report, see “View Imported Comments That Do Not Apply” on page 8-28.

Specify Automatic Comment Import from Last Verification

1. Select **Tools** > **Preferences**, which opens the Polyspace Preferences dialog box.
2. Select the **Project and Results Folder** tab.
3. Under **Import Comments**, select **Automatically import comments from last verification**.
4. Click **OK**.

   After you set this preference, for every run, the software imports review comments from the last run.

View Imported Comments That Do Not Apply

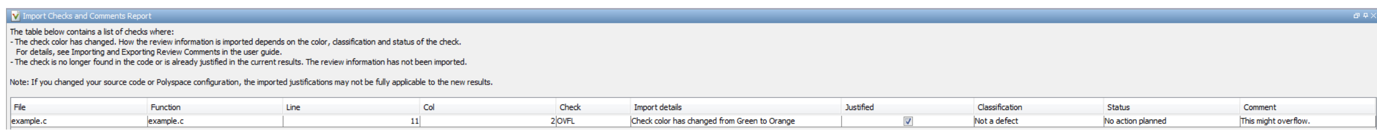
You can directly import review comments from another set of results into the current results. However, it is possible that your review comments do not apply to a subsequent verification because:

- You have changed your source code so that the check is no longer present.
- You have changed your source code so that the check color has changed.
- You have already entered different review comments for the same check.

The Import Checks and Comments Report highlights differences between two verification results. When you import comments from a previous verification, you can see this report. If you have closed the report after an import, to review the report again:

1. Select **Window** > **Show/Hide View** > **Import Comments Report**.

   The Import Checks and Comments Report opens, highlighting differences in the two results.



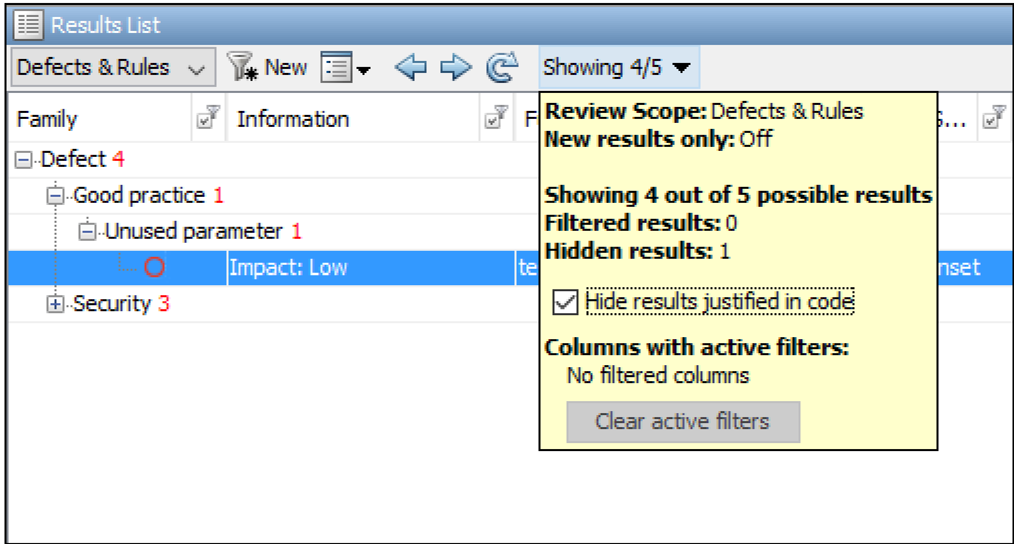
2 Review the differences between the two results.

- If the check color changes, Polyspace populates the **Comment** field but not the fields **Severity**, **Status** or **Justified**.
- If a check no longer appears in the code, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review comments from the previous result.
- If you have already entered different review comments for the same check, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review comments from the previous result.

Justify Results Through Code Annotations

If Polyspace finds a known or acceptable result in your code, you can suppress it in subsequent analyses. Add code annotations indicating that you have reviewed the issue and you do not intend to fix it. Polyspace hides results justified through annotations in **Results List** pane.

To make hidden results visible again, in the **Results List** pane header, click **Showing** and clear the appropriate box.

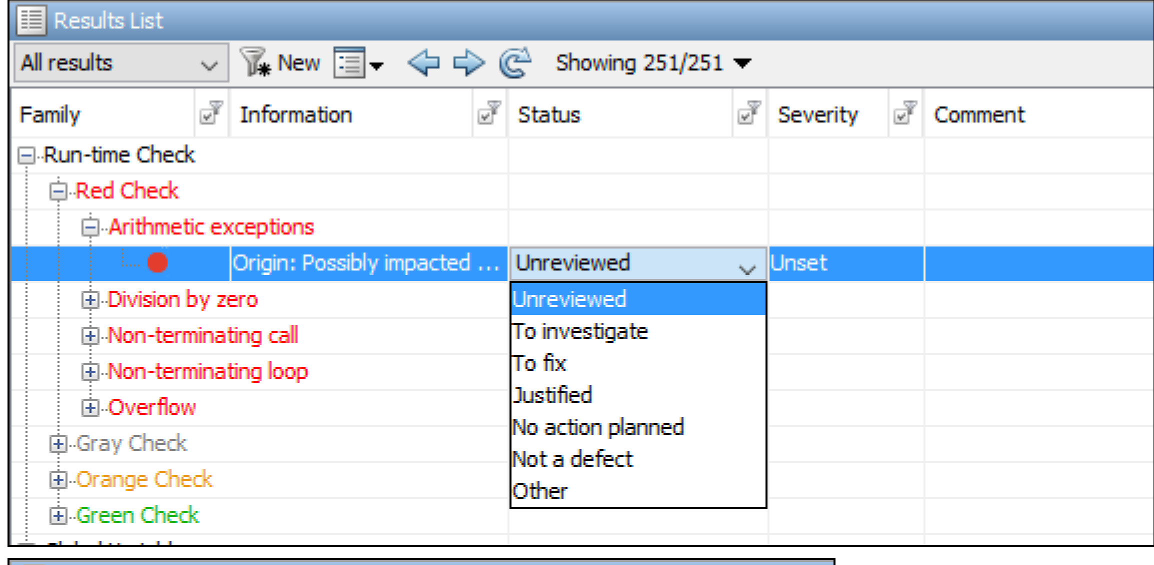


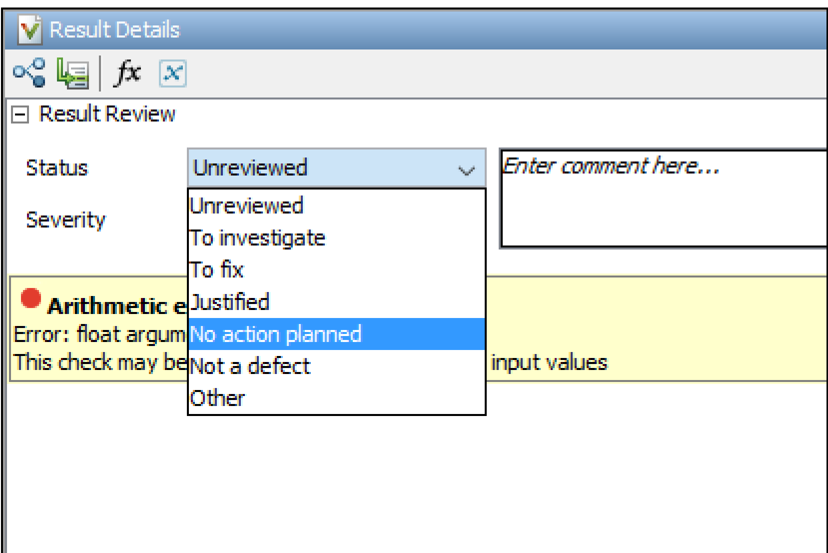
You can add annotations from the Polyspace user interface, or by typing them directly in your code.

Note Results suppressed through annotations still appear in generated reports.

Add Annotations from the User Interface

When you review an analysis result, you can assign a **Status** and **Severity**, and add a **Comment** from either the **Results List** or **Results Details** panes.





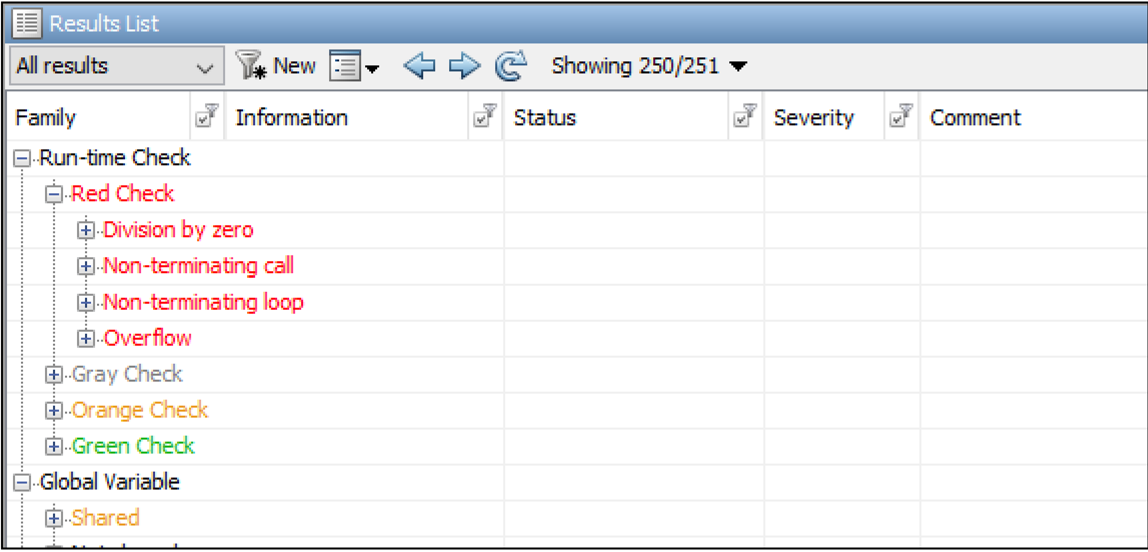
To convert the assigned **Status**, **Severity**, and **Comment** to a code annotation

1. Right-click the result in the **Results List** pane, and select **Add Pre-Justification to Clipboard**.
2. Right-click the result again and select **Open Editor**. The software opens the source file to the location of the defect.
3. Paste the contents of the clipboard on the line containing the defect or coding rule violation. The result **Status**, **Severity**, and **Comment** are converted to an annotation with a Polyspace syntax format.

```
Beta : Long_Float;
   procedure Square_Root is
      Alpha : Float := Random.random;
      Gamma : long_float;
   begin
      Square_Root_conv (Alpha, Beta);
      Beta := Beta - 0.75;
      Gamma := sqrt(Beta);  -- polyspace RTE:EXCP [No action planned:Low] "Additional com
   end Square_Root;

}
```

If you save your source file and rerun the analysis, annotated results with status `Justified`, `No action planned`, or `Not a defect` are hidden in the **Results List** pane.



Type Annotations Directly in Your Code

To add comments directly to your code, use the Polyspace annotation syntax. The syntax is not case-sensitive, and has the following format:

- Annotation for current line of code:

  `line of code; -- polyspace Family:Type`
- Annotation for current line of code and `n` following lines:

  `code; --  polyspace +n Family:Type`
- Annotation for block of code:

  ```
  -- polyspace-begin Family:Type
  code;
  -- polyspace-end Family:Type
  ```

Annotations begin with the keyword `polyspace`, and must include *`Family`* and *`Type`* field values. You can optionally specify *`Status`*, *`Severity`*, and *`Comment`* field values.

```
polyspace Family:Type [Status:Severity] "Comment"
```

If you do not specify a status, Polyspace considers the result justified, and assigns the status `No action planned` to the result.

Use this table to replace the different annotation fields with their allowed values, or see the examples on page 8-34.

Field	Allowed Value
Family	Type of analysis result: • `RTE` • `VARIABLE` Use the asterisk character to specify all analysis results `*:*`.
Type	For `RTE`, use the acronyms for run-time checks. see “Run-Time Checks”. For `VARIABLE`, the only allowed value is the asterisk character " `*` ". Use the asterisk character " `*` " to specify all types in a family `[RTE:*]`.
Status	Text to indicate how you intend to address the error in your code. This value populates the **Status** column in the **Results List** pane: • `Unreviewed` • `To investigate` • `To fix` • `Justified` • `No action planned` • `Not a defect` • `Other` Polyspace suppresses results annotated with status `Justified`, `No action planned`, or `Not a defect` in subsequent analyses. If you specify a status that is not an allowed value, Polyspace stores it as a custom status.

Field	Allowed Value
Severity	Text to indicate how critical you consider the error in your code. This value populates the **Severity** column in the **Results List** pane: • Unset • High • Medium • Low If you specify a severity that is not an allowed value, Polyspace appends it to the status field and stores it as a custom status. For example, `[To investigate:sporadic]` is displayed in the **Status** column of the **Results List** pane as `To investigate sporadic`.
Comment	Additional text, such as a keyword or an explanation for the status and severity. This value populates the **Comment** column in the **Results List** pane.

Syntax Examples

Suppress a Single Result

Enter an annotation on the same line as the result and specify the *Family* (RTE) and *Type* (ZDV). When you do not specify a status, Polyspace assigns the status No action planned, and suppresses the result in subsequent analyses.

```
code; -- polyspace RTE:EXCP
```

Suppress All Run-Time Errors Over Multiple Lines

Enter an annotation with +n between polyspace and the *Family*:*Type* entries.

The following annotation applies to lines 4–7. The line count includes code, comments, and blank lines.

```
4. code ; -- polyspace +3 RTE:*
5. -- comment
6.
7. code;
8. code;
```

Specify Multiple Families in the Same Annotation

Enter each family separated by a space. The following annotation applies to all run-time errors and to all global variable results.

```
some code; -- polyspace RTE:* VARIABLE:*
```

Specify Multiple Types in the Same Annotation

After you specify the *`Family`* (`RTE`), enter each *`Type`* separated by a comma.

```
code; -- polyspace RTE:NIVL, ZDV
```

Add Explanatory Comments to Annotation

After you specify a *`Family`* and *`Type`*, you can add a *`Comment`* with additional information for your justification. You can provide a comment for all families and types, or a comment for each family or type.

```
//Single comment
code; -- polyspace RTE:EXCP VARIABLE:* "Comment applies to RTE and global variables results"

//Multiple comments incorrect syntax:
code; -- polyspace RTE:* "Comment applies to RTE results" VARIABLE:* "Comment applies to global

//Multiple comments correct syntax:
code; -- polyspace RTE:* "Comment applies to RTE results" polyspace VARIABLE:* "Comment applies
```

In annotations, Polyspace ignores all text following a *`Comment`*. To specify additional *`Family`*:*`Type`*, [*`Status`*:*`Severity`*], or *`Comment`* entries, you must reenter the keyword `polyspace` after a comment.

Set Status and Severity

You can specify allowed values on page 8-32, or enter custom values for status and severity. A custom severity entry is appended to the status and stored as a custom **Status** in the user interface.

```
//Set Status only
code; -- polyspace RTE:* [To fix] "some comment"

//Set Status 'To fix' and Severity 'High'
code; -- polyspace RTE:EXCP [To fix: High] "some comment"

//Set custom status 'Assigned' and Severity 'Medium'
code; -- polyspace VARIABLE:* [Assigned: Medium]
```

See Also

`-xml-annotations-description`

More About

- “Define Custom Annotation Format” on page 8-36

Define Custom Annotation Format

This example shows how to create and edit an XML file to define an annotation format, and map it to the Polyspace annotation syntax.

To get started, copy the following code to a text editor and save it on your machine as `annotations_description.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>

<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
             Family="example XML">


 <Expressions Search_For_Keywords="myKeyword"
              Separator_Acronyms="," >
    <!-- Define annotation format in this
    section by adding <Expression/> elements -->

   <Expression Kind="SAME_LINE"
               Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)"
               Rules_Position="1"
               />

   <Expression Kind="GOTO_INCREMENT"
               Regex="myKeyword\s+(\+\d+\s)(\w+(\s*,\s*\w+)*)"
               Increment_Position="1"
          Rules_Position="2"
               />

  <Expression Kind="BEGIN"
               Regex="myKeyword\s*(\w+(\s*,\s*\w+)*)\s*Block_on"
               Rules_Position="1"
          Case_Insensitive="true"
               />

   <Expression Kind="END"
               Regex="myKeyword\s*(\w+(\s*,\s*\w+)*)\s*Block_off"
               Rules_Position="1"
               />
   <Expression Kind="END_ALL"
               Regex="myKeyword\sBlock_off_all"
               />


   <Expression Kind="SAME_LINE"
               Regex="myKeywords\s+(\w+(\s*,\s*\w+)*)(\s*\[(\w+\s*)*([:]\s*(\w+\s*)+)*\])*(\s*-\
               Rules_Position="1"
          Status_Position="4"
         Severity_Position="6"
         Comment_Position="8"
               />
    <!-- SAME_LINE example with more complex Regex.
        Matches the following annotations:
        -- myKeywords 50 [my_status:my_severity] -Additional comment-
        -- myKeywords 50 [my_status]
        -- myKeywords 50 [:my_severity]
        -- myKeywords 50 -Additional comment-
            -->

 </Expressions>

 <Mapping>
    <!-- Map your annotation syntax to the Polyspace annotation
```

```
    syntax by adding <Acronym_Mapping /> elements in this section -->

    <Acronym_Mapping Rule="100" Type="RTE" Acronym="ZDV"/>
    <Acronym_Mapping Rule="101" Type="RTE" Acronym="NIVL"/>
    <Acronym_Mapping Rule="50" Type="VARIABLE" Acronym="*"/>
 </Mapping>
</Annotations>
```

The XML file consists of two parts:

- `<Expressions>...</Expressions>` where you define the format of your annotation syntax.
- `<Mapping>...</Mapping>` where you map your syntax to the Polyspace annotation syntax.

After you are done editing this file, Polyspace can interpret your custom code annotation when you invoke the option `-xml-annotations-description`.

Define Annotation Syntax Format

To define an annotation syntax in Polyspace, your syntax must follow a pattern that you can represent with a regular expression. See “Regular Expressions” (MATLAB). It is recommended that you include a keyword in the pattern of your annotation syntax to help identify it. In this example, the keyword is `myKeyword`. Set the attribute `Search_For_Keywords` equal to this keyword.

Once you know the pattern of your annotation, you can define it in the XML by adding an `<Expression/>` element and specifying at least the attributes `Kind`, `Regex`, and `Rules_position`. For instance the first `<Expression />` element in `annotations_description.xml` defines an annotation with the following attributes:

- `Kind="SAME_LINE"`. The annotation applies to code on the same line.
- `Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)"`. Polyspace uses the regular expression (regex) to search for a string that begins with `myKeyword`, followed by a space `\s+`. Polyspace then searches for a capturing group `(\w+(\s*,\s*\w+)*)` that includes an alphanumeric rule identifier `\w+` and, optionally, additional comma-separated rule identifiers `(\s*,\s*\w+)*`.
- `Rules_Position="1"`. The integer value of this attribute corresponds to the number of opening parentheses preceding the relevant capturing group in the regex. In `myKeyword\s+(\w+(\s*,\s*\w+)*)`, one opening parenthesis precedes the capturing group of the rule identifier `(\w+(\s*,\s*\w+)*)`. If you wanted to match rule identifiers captured by `(\s*,\s*\w+)`, then you would set `Rules_Position="2"`, since two opening parentheses precede this capturing group.

The list of attributes and their values are listed in this table. The example column refers to the format defined in `annotations_description.xml`.

Attribute	Use	Value	Example
`Kind`	Required	`SAME_LINE`	Applies only on the same as the annotation. `code; -- myKeyword 100`

Attribute	Use	Value	Example

		`GOTO_INCREMENT`	Applies on the same line as the annotation, and the following n lines.
			`3. code; -- myKeyword +3 ALL_MIS`
			`4. -- commments`
			`5.`
			`6. code;`
			`7. code;`
			The preceding annotation applies to lines 3–6 only.

		`BEGIN`	Applies to same line and all following lines until a corresponding expression with attribute `Kind="END"` or `"END_ALL"`, or until the end of the file.
			`-- myKeyword 100, 101 Block_on`
			`Code block 1;`
			`...`

		`END`	Stops the application of a rule declared by a corresponding expression with attribute `Kind="BEGIN"`.
			`-- myKeyword 100, 101 Block_on`
			`Code block 1;`
			`...`
			`More code;`
			`-- myKeyword 100 Block_off`
			Only rule 100 is turned off. Rule 101 still applies.

Attribute	Use	Value	Example
		END_ALL	Stops all rules declared by an expression with attribute Kind="BEGIN". -- myKeyword 100, 101 Block_on Code block 1; ... More code; -- myKeyword Block_off_all Both rules 50 and 51 are turned off.
Regex	Required	Regular expression search string	See "Regular Expressions" (MATLAB). Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)" matches these expressions: -- myKeyword 100, 101 -- myKeyword 50
Rules_Position	Required, except when you set Kind="END_ALL"	Integer	The integer value of this attribute corresponds to the number of opening parentheses in the regex before the relevant search expression. <Expression Kind="GOTO_INCREMEN Regex="myKeyword\s+ Increment_Position= Rules_Position="2" /> The search expression for the rule, \w+(\s*,\s*\w+)*, is after the second opening parenthesis of the regex.

Attribute	Use	Value	Example
Increment_Position	Required only when you set Kind="GOTO_INCREMENT"	Integer	The integer value of this attribute corresponds to the number of opening parentheses in the regex before the relevant search expression. <Expression Kind="GOTO_INCREMEN Regex="myKeyword\s+ Increment_Position= Rules_Position="2" /> The search expression for the increment, \+\d+\s, is after the first opening parenthesis of the regex.
Status_Position	Optional	Integer	See Increment_Position example. When you use this attribute, the entry in your annotation is displayed in the **Status** column of the **Results List** pane of the user interface.
Severity_Position	Optional	Integer	See Increment_Position example. When you use this attribute, the entry in your annotation is displayed in the **Severity** column of the **Results List** pane of the user interface.

Attribute	Use	Value	Example
`Comment_Position`	Optional	Integer	See `Increment_Position` example. When you use this attribute, the entry in your annotation is displayed in the **Comment** column of the **Results List** pane of the user interface. Your comment is appended to the string `Justified by annotation in source:`
`Case_Insensitive`	Optional	true/false	When you set this attribute to "true", the regular expression is case insensitive, otherwise it is case-sensitive. If you do not declare this attribute in your expression, the regular expression is case-sensitive. For `Case_Insensitive="true"`, these annotations are equivalent: `-- MYKEYWORD ALL_MISRA BLOCK_ON` `-- mykeyword all_misra block_on`

Map Your Annotation to the Polyspace Annotation Syntax

After you define your annotation format, you can map the rule identifiers you are using to their corresponding Polyspace annotation syntax. You can do this mapping by adding an `<Acronym_Mapping />` element and specifying attributes `Rule`, `Type`, and `Acronym`. For instance if rule identifier 100 corresponds to a division by zero run-time error, map it to the Polyspace syntax `RTE:ZDV` using the following element:

```
<Acronym_Mapping Rule="100" Type="RTE" Acronym="ZDV"/>
```

The list of attributes and their values are listed in this table. The example column refers to the format defined in `annotations_description.xml`.

Attribute	Dependency	Value	Example
`Rule`	Required	User defined	See the mapping section of `annotations_description.xml`
`Type`	Required	Corresponds to Polyspace results family. For a list of allowed values, see allowed values on page 8-38.	See the mapping section of `annotations_description.xml`
`Acronym`	Required	Corresponds to Polyspace results rule. For a list of allowed values, see allowed values on page 8-38.	See the mapping section of `annotations_description.xml`

See Also
"Annotation Description Full XML Template" on page 8-44

Annotation Description Full XML Template

This table lists all the elements, attributes, and values of the XML you can use to define an annotation format and map it to the Polyspace annotation syntax. For an example of how to edit an XML to define annotation syntax, see “Define Custom Annotation Format” on page 8-36.

Element	Attribute	Use	Value
Annotations	Family	Required	User defined string. For example, "Custom Annotations"
Expressions	Search_For_Keywords	Required	User defined string. For example, "myKeyword"
	Separator_Acronyms	Required	User defined string. For example, ","
	Separator_Type_With_Acronym	Optional	User defined string. For example, " "
	Separator_Types_And_Acronym_List	Optional	User defined string. For example, ":"
Expression	Kind	Required	SAME_LINE
			GOTO_INCREMENT
			BEGIN
			END
			END_ALL
			NEXT_CODE_LINE The annotation applies to the next line of code. Comments and blank lines are ignored.
			GOTO_LABEL
			LABEL
			XML_START
			XML_CONTENT The annotation for this expression must be on a single line.
			XML_END
	Regex	Required	Regular expression search string that matches the pattern of your annotation.

Element	Attribute	Use	Value
	`Rules_Position`	Required, except when you set `Kind="END_ALL" or "LABEL"`	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regex before the relevant search expression.
	`Increment_Position`	Required only when you set `Kind="GOTO_INCREMENT"`	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regex before the relevant search expression.
	`Status_Position`	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regex before the relevant search expression.
	`Severity_Position`	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regex before the relevant search expression.
	`Comment_Position`	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regex before the relevant search expression.
	`Label_Position`	Required only when you set `Kind="GOTO_LABEL" or "LABEL"`	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regex before the relevant search expression.

Element	Attribute	Use	Value
	Case_Insensitive	Optional	true/false. When you do not declare this attribute, the default value is false.
	Is_Pragma	Optional	true/false. When you do not declare this attribute, the default value is false. Set this attribute to true if you want to declare your annotation using a pragma instead of a comment.
	Applies_Also_On_Same_Line	Optional	true/false. When you do not declare this attribute, the default value is true. Use this attribute to enable annotations with the old Polyspace syntax to apply on the same line.
Mapping	None	None	None
Acronym_Mapping	Rule	Required	User defined
	Type	Required	Corresponds to Polyspace results rule. For a list of allowed values, see allowed values on page 8-38.
	Acronym	Required	Corresponds to Polyspace results family. For a list of allowed values, see allowed values on page 8-38.

Example

The following example code covers some of the less commonly used attributes for defining annotations in XML.

```
<?xml version="1.0" encoding="UTF-8"?>


<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
             Family="XML Template">

  <Expressions Separator_Acronyms=","
               Search_For_Keywords="myKeyword">

    <Expression Kind="GOTO_LABEL"
                Regex="(\A|\W)myKeyword\s+S\s+(\d+(\s*,\s*\d+)*)\s+([a-zA-Z_-]\w+)"
                Rules_Position="2"
                Label_Position="4"

                />

     <Expression Kind="LABEL"
                Regex="(\A|\W)myKeyword\s+L:(\w+)"
                Label_Position="2"

                />
     <!-- Annotation applies starting current line until
        next declaration of label word "myLabel"
         Example:

         code; -- myKeyword S 100 myLabel
         ...
         more code;
         -- myKeyword L myLabel
    -->

    <Expression Kind="BEGIN"
                Regex="#\s*pragma\s+myKeyword_MESSAGES_ON\s+(\w+)"
                Rules_Position="1"
                Is_Pragma="true"
                />
    <!-- Annotation declared with pragma instead of comment
        Example:#pragma myKeyword_MESSAGES_ON 100 -->


    <!-- Comment declaration with XML format-->

    <!-- XML_START must be declared before XML_CONTENT -->
    <Expression Kind="XML_START"
                Regex="<\s*myKeyword_COMMENT\s*>"

                />
    <!-- Example: <myKeyword_COMMENT>  -->

    <Expression Kind="XML_CONTENT"
                Regex="<\s*(\d*)\s*>(((?![*]/)(?!<).)*)</\s*(\d*)\s*>"
                Rules_Position="1"
                Comment_Position="2"

                />
    <!-- Example: <100>This is my comment</100>
```

```
       XML_CONTENT must be declare on a single line.

       <100>This is my comment
       </100>
       is incorrect.
       -->

    <Expression Kind="XML_END"
                Regex="</\s*myKeyword_COMMENT\s*>"

                />
    <!--    Example: </myKeyword_COMMENT> -->
  </Expressions>

  <Mapping>

    <Acronym_Mapping Rule="100" Type="RTE" Acronym="ZDV"/>
      </Mapping>
</Annotations>
```

Add Review Comments to Code

Note Starting R2017b, Polyspace uses a simpler annotation format. See “Justify Results Through Code Annotations” on page 8-30 .

This example shows how to place review comments in your code for a particular result. If your code comments follow a particular syntax, in a later verification on the same code, Polyspace can read the comments. Using the comments, Polyspace automatically populates the **Severity**, **Status** and **Comment** fields on the **Results List** pane. After you have placed your comments in your code, you or another reviewer can avoid reviewing the same result twice.

In this section...
“Enter Code Comments in Specific Syntax” on page 8-49
“Copy Comment Syntax from Polyspace User Interface” on page 8-50

Enter Code Comments in Specific Syntax

You can manually enter comments in a specific syntax just before the line containing the result.

To comment:

- An individual line of code, use the following syntax:

```
-- polyspace<Type: RunTimeError1[,RunTimeError2[,…]] : [Severity] : [Status] >
 [Additional text]
```

- A section of code, use the following syntax:

```
-- polyspace:begin<Defect:Kind1[,Kind2] : [Severity] : [Status] >
[Additional text]

... Code section ...

-- polyspace:end<Type:Kind1[,Kind2] : [Severity] : [Status] >
```

The square brackets *[]* indicate optional information.

Replace	Replace with
Type	**Runtime errors:** RTE
	Global variables: VARIABLE
Kind1,Kind2,...	**Runtime errors:** Acronyms for checks such as ZDV, OVFL, etc.. If you want the comment to apply to all checks on the following line, specify ALL.

Replace	Replace with
	Global variables: `ALL`. For global variables, the same comment syntax applies irrespective of whether they are shared or used.
`Severity`	Text that indicates the severity of the defect. Enter one of the following: • `Unset` • `High` • `Medium` • `Low` This text populates the **Severity** column on the **Results List** pane.
`Status`	Text that indicates how you intend to correct the error in your code. Enter one of the following or any other text: • `Unreviewed` • `To investigate` • `To fix` • `Justified` • `No action planned` • `Not a defect` • `Other` This text populates the **Status** column on the **Results List** pane.
`Additional text`	Any text. This text populates the **Comment** column on the **Results List** pane.

- “Syntax Example: Run-time Checks” on page 8-50
- “Syntax Example: Global Variables” on page 8-50

Syntax Example: Run-time Checks

- Non-terminating call:

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

- Division by zero:

```
-- polyspace<RTE: ZDV : High : Fix > Denominator cannot be zero
```

Syntax Example: Global Variables

```
-- polyspace<VARIABLE: ALL : Low : Justify with annotations> Known issue
```

Copy Comment Syntax from Polyspace User Interface

Instead of manually entering the comment in a specific syntax, you can copy the comment syntax from the Polyspace user interface and paste in your code.

1. On the **Results List** or **Result Details** pane, assign a **Severity**, **Status** and **Comment** to a result.

   a. Select the result.

   b. Select options from the **Severity** and **Status** dropdown lists.

   c. In the **Comment** field, enter a comment that helps you recognize the result easily.

2. Copy the **Severity**, **Status** and **Comment**.

   a. On the **Results List** pane, right-click the result.

   b. Select **Add Pre-Justification to Clipboard**. The software copies the justification string to the clipboard.

3. Paste the **Severity**, **Status** and **Comment** in your source code.

   a. On the **Results List** pane, right-click the result and select **Open Editor**.

      Your source file opens on the **Code Editor** pane or an external text editor depending on your **Preferences**. The current line is the line containing the result.

   b. Using the paste option in the text editor, paste the justification template string on the line immediately before the line containing the result.

      You can see your **Severity**, **Status** and **Comment** as a code comment in a format that Polyspace can read later.

   c. Save your source file.

4. Run the verification again. Open your results.

   On the **Results List** pane, the software populates the **Severity**, **Status** and **Comment** fields for the result. You can either ignore these findings, or filter them from the **Results List** pane.

Filter and Group Results

This example shows how to filter and group results on the **Results List** pane. To organize your result review, use filters and groups when you want to:

- Review certain types of checks in preference to others. For instance, you first want to address only the `Non-terminating loop` checks.
- Review only new results found since the last verification.
- Not review checks you have already justified.

  Typically, in your second or later rounds of review, you would have some checks already justified.
- Review only those checks that you have already assigned a certain status. For instance, you want to review only those checks to which you have assigned the status, `Investigate`.
- Review all checks in the body of a particular file or function. Because of continuity of code, reviewing these checks together can help you organize your review process.

  You can also review the checks in one file alone if you have written the code for that file only and not the entire set of source files used for verification.

Filter Results

You can filter results using graphs on the **Dashboard** pane or filters on the **Results List** pane.

Filter Using Dashboard

The **Check Distribution** graph on the **Dashboard** pane provides a graphical overview of the results, divided by result color. You can select a color in the graph to view only checks of that color.

To clear filters from the **Dashboard** pane, select the link **View all results in this scope**. This action clears all filters and displays the available results in the scope that you choose in the upper left menu of the **Results List** toolbar.

Filter Using Results List

For all other filtering mechanisms, use filters on the **Results List** pane itself. To clear filters from the **Results List** pane, use the button **Clear active filters** in the **Showing** dropdown.

- To filter results, on the **Results List** pane, select the icon on the desired column. Clear **All**. Select the boxes for the results that you want displayed.

Item to Filter	Column
Results in a certain file or function	**File** or **Function**
Results with a certain severity or status	**Severity** or **Status**
Results that you have justified. If you assign the status `Justified`, `No action planned` or `Not a defect`, a result is justified.	**Justified**
Checks only	**Family**
Checks of a certain color	**Family**

Item to Filter	Column
Global variables of a certain type	**Family**
Code metrics	**Family**

- To review only new results found since the last verification, on the **Results List** pane, select .

Note You can also apply multiple filters. Once you apply a set of filters to your verification results, they are preserved for subsequent verifications on the same project module. The **Results List** pane shows the number of results filtered from display. If you place your cursor on the number, you can see which filters have been applied.

Group Results

On the **Results List** pane, from the list, select an appropriate option.

- To view ungrouped results , select **None**.
- To view results grouped by result type, select **Family**.

  The results are organized by type: checks, global variables, coding rule violations, code metrics. Within each type, they are grouped further.

  - The checks are grouped by color. Within each color, the checks are organized by check group. For more information on the groups, see “Run-Time Checks”.
  - The global variables are grouped by their usage. For more information, see “Global Variables”.
- To show results grouped by file, select **File**.

  Within each file, the results are grouped by procedures in the file.
- To show results grouped by package, select **Package**.

  Within each class, the results are grouped by method. The global variables are grouped under **_init_globals()**.

Prioritize Check Review

This example shows how to organize your review of orange checks.

1. Before beginning your check review, do the following:

   - See the **Code covered by verification** graph on the **Dashboard** pane. See if the **Procedure** and **Code operation** columns display a value closer to 100%. Otherwise, identify why Polyspace could not cover the code.

     For more information, see “Review Gray Checks” on page 8-20. If a substantial number of functions or code operations were not covered, after identifying and fixing the cause, run verification again.
   - See if you have used the right configuration. With the results open, select **Window > Show/ Hide View > Configuration**.

     Sometimes, especially if you are switching between multiple configurations, you can accidentally use the wrong configuration for the verification.
2. From the drop-down list in the left of the **Results List** pane toolbar, select **Critical checks**.

   This action retains only red, gray and critical orange checks.
3. Click the forward arrow to go to the first unreviewed check. Review this check.

   For more information, see “Results Review Process”.

   Continue to click the forward arrow until you have reviewed through all of the checks.
4. Before reviewing orange checks, review red and gray checks.
5. To check that you have addressed the red and critical orange checks, rerun the verification and view your results.
6. If you do not have red or unjustified critical orange checks, from the drop-down list in the left of the **Results List** pane toolbar, select **All results**.

   Depending on the quality level you want, you can choose whether to review the noncritical orange checks or not. For more information, see “Do I Have Too Many Orange Checks?” on page 9-7.
7. To see what percentage of checks you have justified:

   a. If you want the percentage broken down by color and type, on the **Results List** pane, from the list, select **Family**. If you want the percentage broken down by file and function, select **File**.

   b. View the entries in the **Justified** column.

Generate Report

This example shows how to generate a report from your verification results. Using a customizable template, the report presents your results in a concise manner for managerial review or other purposes. To generate a verification report, do one of the following:

- Specify certain options before verification so that the software automatically generates a report.
- Generate a report from your verification results.

You can also export your results to a text file and generate graphs and statistics. See "Export Results to Text File" on page 8-58.

Specify Report Generation Before Verification

User Interface	Command Line
1 Select your project configuration. On the **Configuration** pane, select **Reporting**. Specify report generation options. For more information, see "Reporting". **2** Run verification and open your results. **3** Select **Reporting** > **Open Report** **4** Navigate to the `Polyspace-Doc` subfolder in your results folder. You can see the generated report in this subfolder. Click **OK** to open the report.	Use the appropriate option with the `polyspace-ada` command. For more information on the options, see the section **Command-Line Information** in "Reporting". Additionally, you can also specify a report name using the option `-report-output-name`.

Generate Report After Verification

User Interface	Command Line

1 Open your verification results.
2 Select **Reporting** > **Run Report**.
 	
The Run Report dialog box opens.
3 Select the following options:
• In the **Select Reports** section, select the report templates you want to use. For example, you can select **Developer** and **Quality**.
 	
For more information, see `Report template`.
• Select an **Output folder** in which to save the reports.
• Select the **Output format** for the reports.
• If the display language (Windows) or locale (Linux) of your operating system is set to another language, you see an option to generate English reports. Select this option if you want an English report, otherwise the report is in another language.
• If you want to filter results from your report, use filters on the **Results List** pane to display only the results that you want to report. Then, when generating reports, select **Only include currently displayed results**.
 	
For more information on filtering, see “Filter and Group Results” on page 8-52.
• If you perform a file by file verification, you can generate a report of the verification results for each file or for all the files together. To generate a single report, select the option **Generate a single report including all unit results**.
 	
You can generate a single *filtered* report for all units. The report uses the filters applied to each unit and the review scope (`All results`, `Critical checks`, etc.) applied to the currently displayed unit.
4 Click **Run Report**.
	Use the appropriate option with the `polyspace-report-generator` command.
	
	The available options are:
	• `-template` *`path`*: Path to report template file. For more information, see `Report template`.
	
	The predefined report templates are in *`matlabroot`*`\toolbox\polyspace\psrptgen\templates\Developer.rpt`. Here, *`matlabroot`* is the MATLAB® installation folder such as `C:\Program Files\MATLAB\R2015a`.
	• `-format` *`type`*: Output format of report. The allowed *`type`*s are`HTML`, `PDF` and `WORD`.
	• `-output-name` *`filename`*: Name of report.
	• `-results-dir` *`folder_paths`*: Path to folder containing your verification results.
	
	To generate a single report for multiple verifications, specify *`folder_paths`* as follows:
	
	`"folder1, folder2, ..., folderN"`
	
	where *`folder1`*`,` *`folder2`*`, ...` are paths to the folders that contain verification results. For example,
	
	`"C:\My_project\Module_1\results, C:\My_project\Module_2\Results"`
	
	If you do not specify a folder path, the software uses verification results from the current folder.
	• `-set-language-english`: Use this option to generate English reports if the default report is in another language. The display language (Windows) or locale (Linux) of your operating system determines the default language in the report.

User Interface	Command Line
The software creates the specified reports and opens them.	

See Also

Related Examples

- "Customize Report Templates" on page 8-62

Export Results to Text File

You can export your verification results to a tab delimited text file. Using the text file, you can:

- Generate graphs or statistics about your results that you cannot readily obtain from the user interface by using MATLAB or Microsoft® Excel®. For instance, for each check type (**Division by zero**, **Overflow**), you can calculate how many checks are red, orange, or green.
- Integrate the verification results with other checks you perform on your code.

Export Results

You can export results from the user interface or command line.

<table>
<tr><th>User Interface</th><th>Command Line</th></tr>
<tr><td>1. Open your verification results.
2. Export all results or only a subset of the results.
• To export all results, select Reporting > Export > Export All Results.
• If you want to filter results from your report, use filters on the Results List pane to display only the results that you want to report. Then, when exporting results, select Reporting > Export > Export Currently Displayed Results.
For more information on filtering, see “Filter and Group Results” on page 8-52.
3. Select a location to save the text file and click OK.</td><td>Use appropriate options with the <code>polyspace-report-generator</code> command.
The available options are:
• <code>-generate-results-list-file</code>: Specifies that a text file must be generated.
• <code>-results-dir folder_paths</code>: Path to folder containing your verification results. If you do not specify a folder path, the software uses verification results from the current folder.
To generate text files for multiple verifications, specify <code>folder_paths</code> as follows:
<code>"folder1, folder2, ..., folderN"</code>
<code>folder1, folder2, ...</code> are paths to the folders that contain verification results. For example:
<code>"C:\My_project\Module_1\results, C:\My_project\Module_2\Results"</code>
To merge the text files, use the <code>join</code> function.</td></tr>
</table>

The exported text file uses the character encoding on your operating system. If special characters from your comments are not exported correctly in the text file, change the character encoding on your operating system before exporting.

View Exported Results

The text file contains the result information available on the **Results List** pane in the user interface (except for line and column information). See “Results List” on page 8-6. Though you cannot identify the location of a result in your source code using the text file, you can parse the file and generate graphs or statistics about your results.

The text file also contains a **Key** column. If the same file has the same result across multiple verifications, it has the same entry in this column. When you merge multiple verification results that might contain common files, use this entry to eliminate copies of a result. For instance, if you run coding-rule checking on multiple modules and merge the results, header files and coding rule violations in them appear in multiple module verification results. To eliminate copies of a coding rule violation, use the entry in the **Key** column.

Generate Graphs from Results

This example shows how to generate a pie chart from the generated text file showing the distribution of red, gray and orange run-time checks by check type. The text file has the name `Result_List.txt`.

```
% Read contents of text file into a table
resultsList = readtable('Result_List.txt', 'Delimiter', '\t');

% Eliminate results that are not run-time checks, also eliminate green checks
matches = ismember(resultsList.Family, {'Run-time Check'}) & ...
    ~ismember(resultsList.Color, {'Green'});
checkList = resultsList(matches, :);

% Create a pie chart showing distribution of checks
pie(categorical(checkList.Check))
```

The key functions used in the example are:

- `readtable`: Create table (MATLAB) from file.
- `pie`: Create pie chart from a categorical array (MATLAB).

When you execute the script, you see a distribution of checks by check type.

Export Global Variable List

You can export the list of global variables in your code to a tab delimited text file. The text file or the table contains the same information as the **Variable Access** pane in the Polyspace user interface.

Using the text file, you can:

- Generate graphs or statistics about global variables. For instance, you can see the percentage of shared global variables that are not protected against concurrent access.
- Use the range information to create external constraints for global variables. For instance, you can report that your code is free of certain run-time errors only for the extracted range of global variables.

  You can also use the range to specify external constraints on subsequent verifications or verification of other modules. See “Specifying Constraints Using Text Files” on page 4-9.

Export Variable List to Text File

You can export results from the user interface or command line.

User Interface	Command Line

1 Open your verification results.
2 Select **Reporting** > **Export** > **Export Variable Access**.
3 Select a location to save the text file and click **OK**.
	Use appropriate options with the `polyspace-report-generator` command.
	
	The available options are:
	• `-generate-variable-access-file`: Specifies that a text file must be generated.
	• `-results-dir` *`folder_paths`*: Path to folder containing your verification results. If you do not specify a folder path, the software uses verification results from the current folder.
	
	To generate text files for multiple verifications, specify *`folder_paths`* as follows:
	
	`"folder1, folder2, ..., folderN"`
	
	`folder1`, *`folder2`*, `...` are paths to the folders that contain verification results. For example:
	
	`"C:\My_project\Module_1\results, C:\My_project\Module_2\Results"`

View Exported Variable List

The text file or the table contains the result information available on the **Variable Access** pane in the user interface. See also “Variable Access” on page 8-11.

Some differences in presentation between the **Variable Access** pane and the text file are listed below.

- The **Access** column in the text file indicates whether the row shows information about the variable (**Aggregate**) or information about operations on the variable (**Write** or **Read**).
- The **Function** column in the text file shows the functions where the variable is read or written (and on the **Variable Access** pane).
- There are no rows corresponding to read and write operations from tasks (and on the **Variable Access** pane). This information is available in the **Written by task** and **Read by task** columns.
- The colors on the **Variable Access** pane are represented through the columns **Unreachable** and **Protected**:
  - If a shared variable is accessed in multiple tasks without a common protection, it is colored orange on the **Variable Access** pane. In the text file, the **Protected** column shows **Unprotected**.
  - If a shared variable is accessed in multiple tasks but with a common protection, it is colored green on the **Variable Access** pane. In the text file, the **Protected** column shows **Protected**.
  - If a shared variable is not accessed at all, it is colored gray on the **Variable Access** pane. In the text file, the **Unreachable** column shows **Is unreachable**.

See Also

Related Examples

- “Variable Access” on page 8-11
- “Export Results to Text File” on page 8-58

Customize Report Templates

This example shows how to customize the templates that you use for report generation. To customize the templates, you must have MATLAB Report Generator™ software installed on your system.

In this section...
“Create Custom Template” on page 8-62
“Apply Global Filters in Template” on page 8-62
“Override Global Filters” on page 8-63
“Use Custom Template” on page 8-64

Create Custom Template

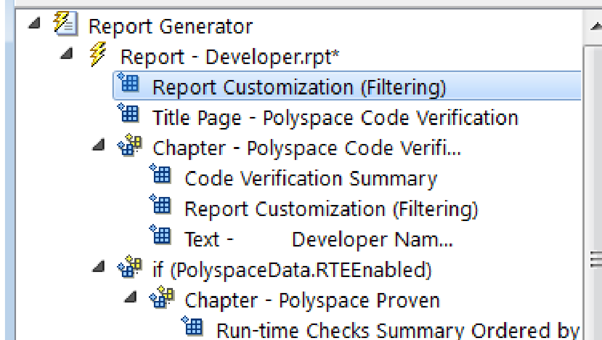
If you have Simulink® Report Generator software on your system:

1. Open the Report Explorer from the MATLAB command prompt:

   ```
   report
   ```
2. Select **File** > **Open** to open the template that you want to customize.
3. Navigate to `Matlab_Install/toolbox/polyspace/psrptgen/templates` where `Matlab_Install` is the MATLAB installation folder. Use the `matlabroot` command to find the folder location.
4. Modify the template using the options on the **Report Options** pane.
5. Save the modified template as a `.rpt` file.

Apply Global Filters in Template

1. In the Report Explorer, open the template that you want to customize. For instance, **Developer.rpt**.
2. On the **Name** pane, under the **Polyspace** node, select **Report Customization (Filtering)**.
3. Drag this component above the **Title Page** component that is located under the **Report-Developer.rpt** node.



4. On the **Report Customization (Filtering)** pane in the right side of the Report Explorer, specify your filters. For example:

- To include **Unreachable code** checks, under **Advanced filters**, in the **Check types to include** field, enter `Unreachable code`.
- To exclude **Unreachable code** checks, under **Advanced filters**, in the **Check types to include** field, enter the regular expression `^(?!Unreachable code).*`.
- To include the file `main.c`, under **Advanced filters**, in the **Files to include** field, enter `main.c`.
- To exclude the file `main.c`, under **Advanced filters**, in the **Files to include** field, enter the regular expression `^(?!main.c).*`.

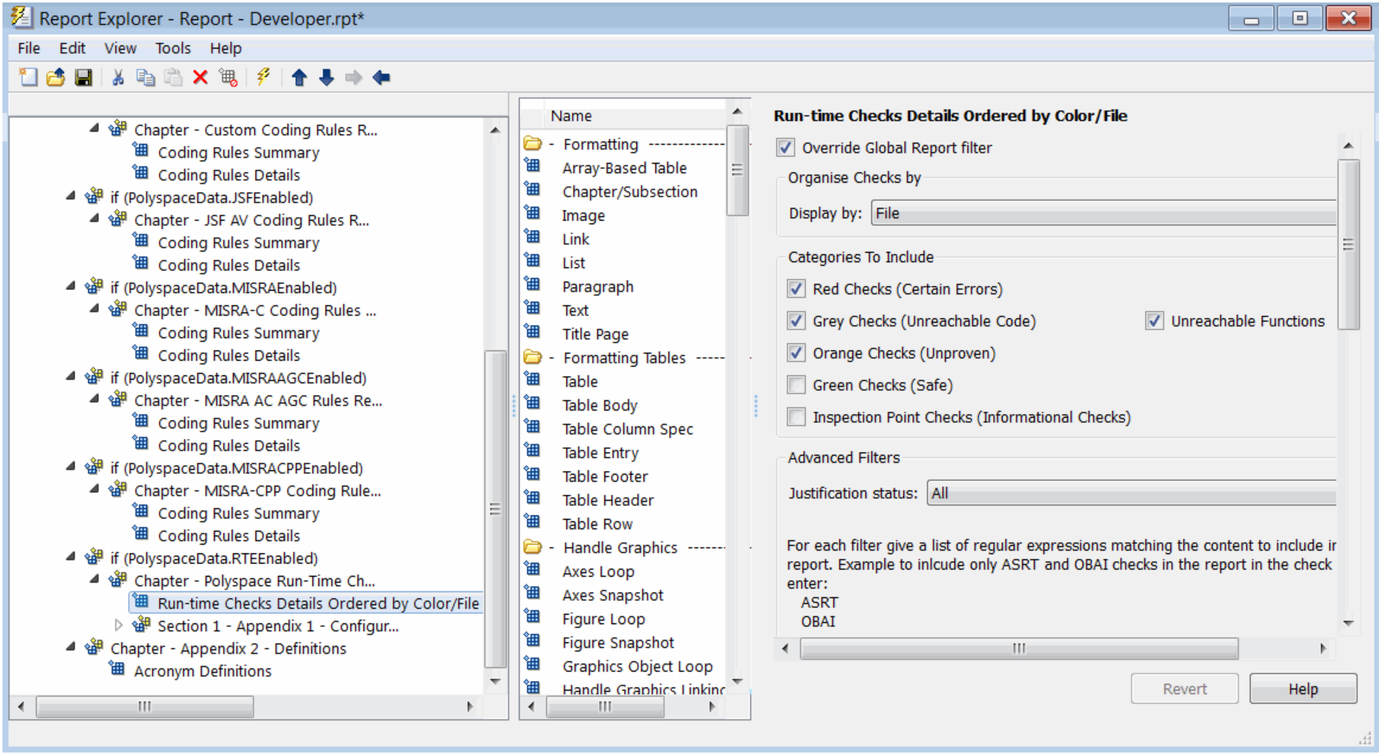
In each text box, specify one filter per line.

For more information, see “Regular Expressions” (MATLAB).

Override Global Filters

You can override some of the global filters using the **Run-time Check Details Ordered by Color/File** component. For example, you can have a report chapter that contains NIV checks even though NIV checks are excluded by the global filters.

1 Select the **Run-time Check Details Ordered by Color/File** component.



2 On the right of the dialog box, select the **Override Global Report filter** check box.

3 Specify your filters for this component. For example, in the **Check types to include** field, enter NIV.

4 Save the template.

For more information on the components available for customizing a report template, see “Generate Report”.

Use Custom Template

1. Open your results in the Polyspace interface.
2. Select **Reporting** > **Run Report**.
3. Click **Browse**.
4. Navigate to the location where you saved your template `.rpt` file.
5. Select the file and click **OK**. Under **Select Reports**, you see your template.
6. Select the template and click **Run Report**.

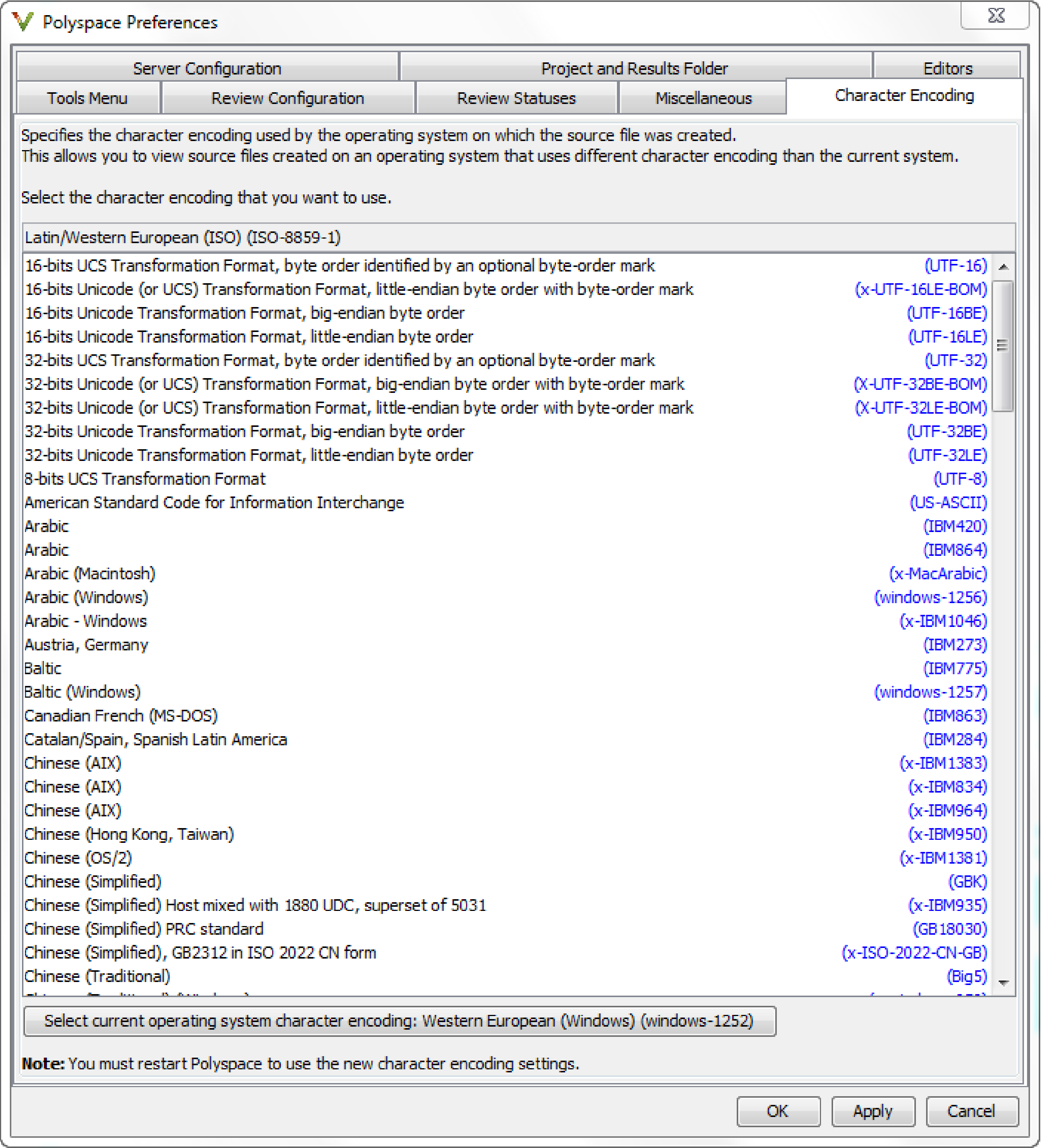
Set Character Encoding Preferences

If the source files that you want to verify are created on an operating system that uses different character encoding than your current system (for example, when viewing files containing Japanese characters), you receive an error message when you view the source file or run certain macros.

The **Character encoding** option allows you to view source files created on an operating system that uses different character encoding than your current system.

To set the character encoding for a source file:

1. Select **Tools** > **Preferences**.
2. In the Polyspace Preferences dialog box, select the **Character encoding** tab.



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.

Managing Orange Checks

- "What Is an Orange Check?" on page 9-2
- "Sources of Orange Checks" on page 9-5
- "Do I Have Too Many Orange Checks?" on page 9-7
- "Limit Display of Orange Checks" on page 9-8
- "Reduce Orange Checks" on page 9-10

What Is an Orange Check?

Orange checks indicate unproven code, which means the software cannot prove that the code:

- Produces a run-time error
- Does not produce a run-time error

Polyspace verification attempts to prove the absence or existence of run-time errors. Therefore, the software considers all code unproven before a verification. During a verification, the software attempts to prove that the code is:

- Without run-time errors (green)
- Certain to fail (red)
- Unreachable (gray)

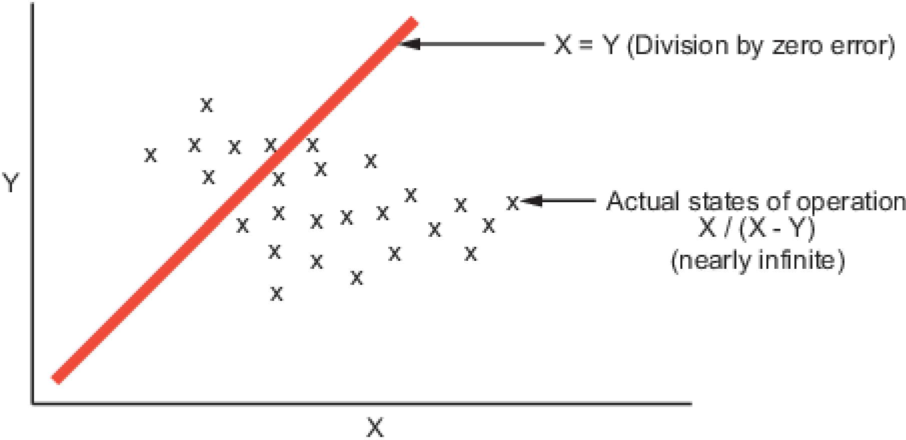
Code that is not assigned one of these categories (colors) stays unproven (orange).

Code often remains unproven in situations where some paths fail while others succeed. For example, consider the following instruction:

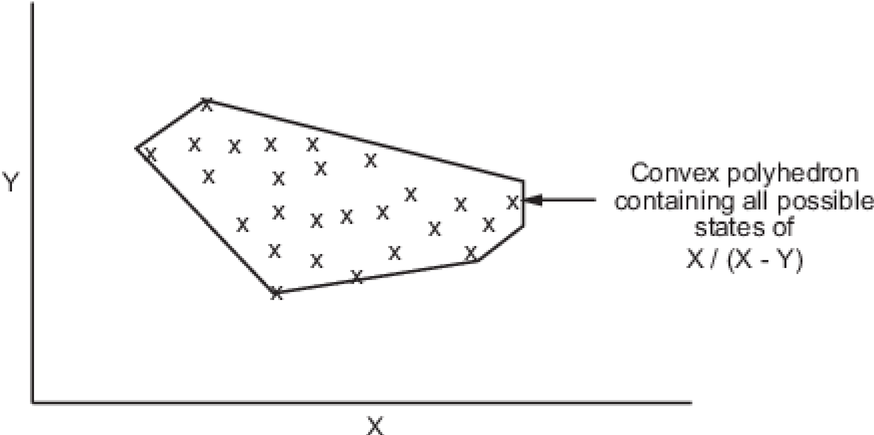
```
X = 1 / (X - Y);
```

Does a division-by-zero error occur?

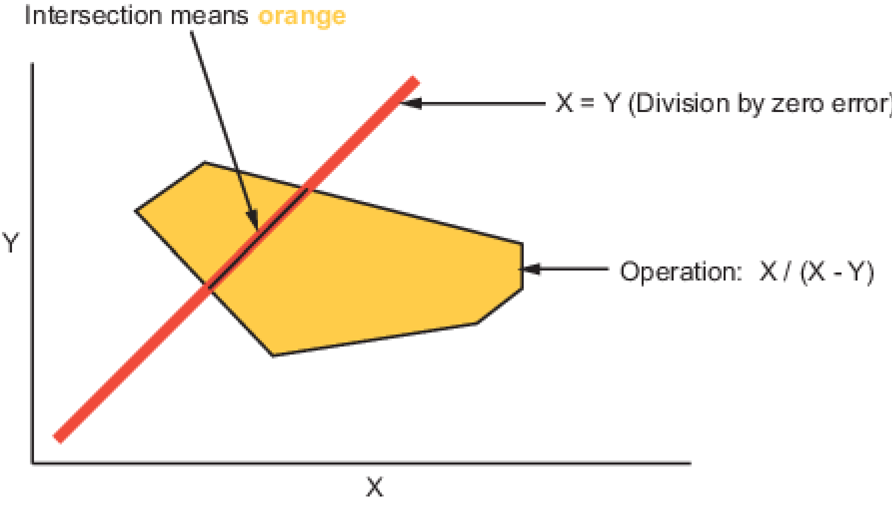
The answer depends on the values of *X* and *Y* . However, there are an almost infinite number of possible values. Creating test cases for all possible values is not practical.



Because 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.

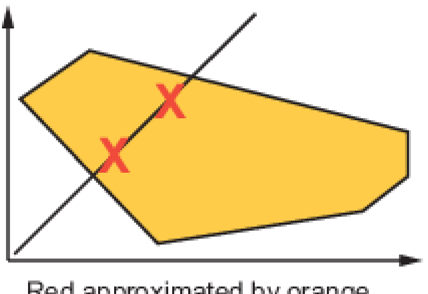


Polyspace verification then compares the data set represented by this polyhedron to the error zone. If the two data sets intersect, the check is orange.

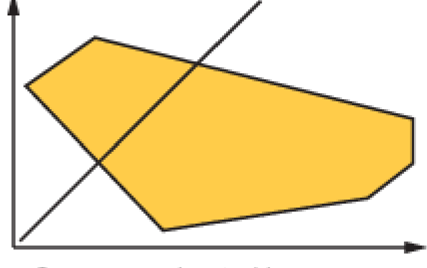


Graphical Representation of an Orange Check

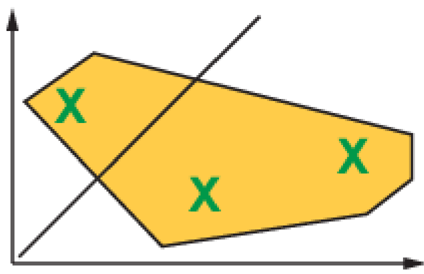
A true orange check represents a situation where some paths fail while others succeed. However, because the data set in the verification is an approximation of actual values, an orange check may actually represent a check of another color.



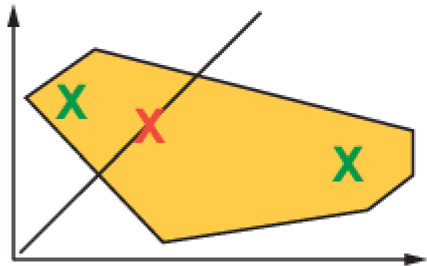
Red approximated by orange



Gray approximated by orange



Green approximated by orange



Any other situation (true orange)

Polyspace reports an orange check when 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 does not have run-time errors.

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:

In this section...
"Orange Checks from Code" on page 9-5
"Orange Checks from Verification Limitations" on page 9-5

Orange Checks from Code

Potential Bug

An orange check can reveal code which will fail under some circumstances. These types of orange checks 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 `Division by Zero`.

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 many combinations of input data rather than a 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 might know that the three variables cannot 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, you can usually identify the cause quickly. After identifying a data set issue, you might want to comment the code to flag the warning, or modify the code to take the constraints into account.

Orange Checks from Verification Limitations

Inconclusive Verification

An orange check can be caused by situations in which the verification is unable to conclude whether a problem exists.

In some code, it is impossible to conclude whether an error exists without additional information.

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.

Unless you define the call sequence, the verification cannot determine if an error will occur.

Most inconclusive orange checks take some time to investigate. An inconclusive orange check often results from complex code structure. Sometimes, such situations take an hour or more to understand. Depending on the criticality of the function and the required speed of execution, you may might want to rewrite the code to remove risk of failure.

Basic Imprecision

An orange check can be caused by imprecise approximation of the data set used for verification.

For example, consider a variable *X*:

- Before the function call, *X* is defined as having the following values: `-5`, `-3`, `8`, or a value in the range `[10...20]`. `0` has been excluded from the set of possible values for *X*.
- However, due to optimization at low precision levels (`-O0`), the verification approximates *X* in the range `[-5...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 might be able to resolve orange checks by increasing the precision level. If increasing the precision level does not resolve the orange check, verification cannot help directly. You must review the code to determine the problem.

For more information, see “Polyspace Software Assumptions”.

Do I Have Too Many Orange Checks?

If the goal of code verification is to prove the absence of run-time errors, you might be concerned by the number of orange checks in your results.

However, the presence of multiple orange checks need not be a cause for concern. The minimum number that you want depends on several factors:

- **Development Stage** – When verifying the first version of a software component, focus exclusively on resolving red checks. As development progresses, start considering the orange checks more and more.
- **Application Requirements** – Sometimes, to write provable code, you can compromise with properties such as code size, speed, and portability. Depending on the requirements of your application, you might optimize one or more of these properties at the expense of more orange checks.
- **Quality Goals** – Using Polyspace software, you can meet your quality goals. Therefore, before you verify code, you must define quality goals for your application. These goals should be based on the criticality of the application, as well as time and cost constraints. Based on your quality goals, you can choose to retain a specific minimum number of orange checks in your application.

It is these factors that ultimately determine how many orange checks are acceptable in your results, and what you must do with the orange checks that remain.

Limit Display of Orange Checks

This example shows how to control the number and type of orange checks displayed on the **Results List** pane using the dropdown menu on the upper left. To reduce your review effort, you can do one of the following:

- Display only the critical orange checks.

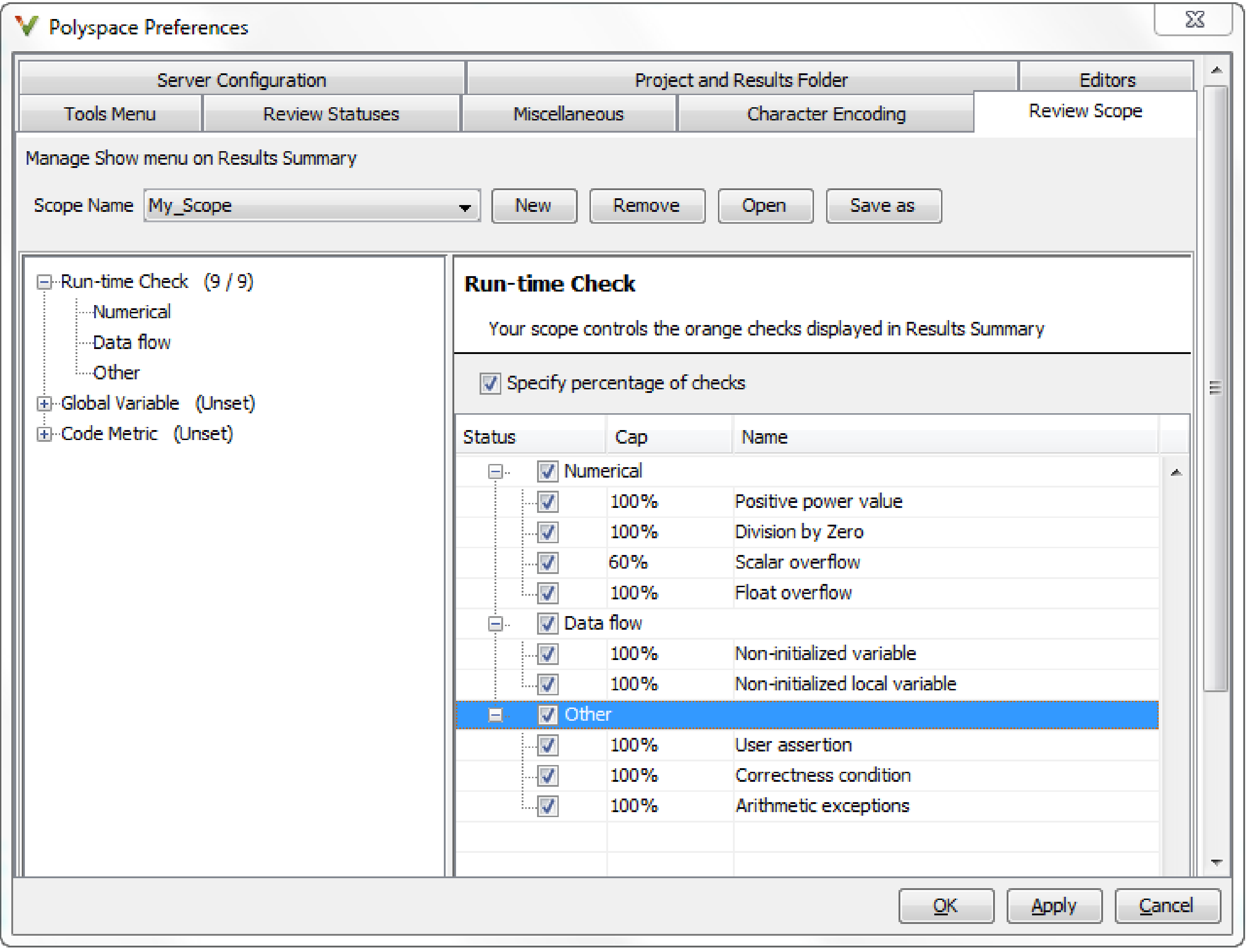
  Use the option **Show** > **Critical checks** on the **Results List** pane.
- Limit the number or suppress orange checks for certain check types, using additional options on the **Show** menu.

  You can create your own options. You can share the option files to help developers in your organization review at least a certain number or percentage of orange checks.

1. Select **Tools** > **Preferences**.
2. On the **Review Scope** tab, select **New**. Save your option file.
3. On the left pane, select **Run-time Check**. On the right pane, to suppress a check completely from display, clear the box next to the check. To suppress a check partly, specify a percentage less than 100 to display.

   To select all checks belonging to a category such as **Numerical**, select the box next to the category name. For more information on the categories, see “Run-Time Checks”. If only a fraction of checks in a category are selected, the check box next to the category name displays a ▣ symbol.

   Instead of a percentage, you can specify a number or the string `ALL`. To specify a number, clear the box **Specify percentage of checks**.



4 Select **Apply** or **OK**.

On the **Results List** pane, the menu in the left of the toolbar displays the additional options.

5 Select the option corresponding to the limits that you want. Only the number or percentage of orange checks that you specify remain on the **Results List** pane.

- If you specify an absolute number, Polyspace displays that number of orange checks.
- If you specify a percentage, Polyspace displays green and justified orange checks until they make up the percentage. If they do not make up the percentage, the software then displays unjustified orange checks.

  You can use a review scope with percentage specifications to ensure that you justify at least a certain percentage of checks.

Reduce Orange Checks

An orange check indicates that Polyspace detects a possible run-time error but cannot prove it. To help Polyspace produce more proven results, you can:

- Follow good coding practices.
- Specify the necessary verification options.

You can also limit the number and family of orange checks displayed on **Results List**. For more information, see "Limit Display of Orange Checks" on page 9-8.

In this section...

"Improve Verification Precision" on page 9-10
"Apply Coding Guidelines" on page 9-11
"Stub Parts of the Code Manually" on page 9-11
"Specify Multitasking Behavior" on page 9-14

Improve Verification Precision

Improving the precision of a verification can reduce the number of orange checks in your results.

There are a number of Polyspace options that can improve the precision of the verification. The compromise for this improved precision is increased verification time.

The following sections describe how to improve the precision of your verification:

- "Set the Analysis Precision Level" on page 9-10
- "Set Software Safety Analysis Level" on page 9-10

Set the Analysis Precision Level

The precision level specifies the mathematical algorithm used to compute the cloud of points (polyhedron) containing many possible states for the variables. Changing the precision level does not improve the quality of your code. However, orange checks caused by low precision can become green when verified with higher precision. The default precision level is 2. To set the precision level:

1. In the Polyspace user interface, on the **Configuration** pane, select **Precision**.
2. From the **Precision Level** drop-down list, select `0`, `1`, `2`, or `3`.

For more information, see `Precision level`.

Set Software Safety Analysis Level

The verification level specifies how many times the abstract interpretation algorithm passes through your code. Each pass results in a deeper level of propagation of calling and called context. The deeper the verification goes, the more precise it is. By default, verification proceeds to `Software Safety Analysis Level 4`. To set the verification level:

1. In the Polyspace user interface, on the **Configuration** pane, select **Precision**.
2. From the **Verification level** drop-down list, select the level that you want.

For more information, see `Verification level`.

Apply Coding Guidelines

The number of orange checks per file depends on the coding style used in the project.

The following coding guidelines improve Polyspace precision and selectivity in Ada code verification:

- Use constrained types. Use subtype and not standard type.
- Do not use "use at" clause.
- Minimize the use of big and complex types (record of record, array of record, etc.).
- Minimize the use of volatile variables.
- Minimize the use of assembler code.
- Do not mix assembly code and Ada. Gather assembly code in a procedure or function which can be automatically stubbed.

Stub Parts of the Code Manually

Manually stubbing parts of your code can reduce the number of orange checks in your results. Manual stubbing does not improve the quality of your code, but only changes the results.

Stubs do not need to model the details of the functions or procedures involved. They only need to represent how the code interacts with 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 take a value from the full range of an integer.

The following sections describe how to reduce orange checks using manual stubbing:

- “Manual vs. Automatic Stubbing” on page 9-11
- “Emulating Function Behavior with Manual Stubs” on page 9-12
- “Reducing Orange Checks with Empty Stubs” on page 9-13
- “Applying Constraints to Variables Using Stubs” on page 9-13

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 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 a run-time error.
- **Manual stubs** – You create these stub functions to emulate the behavior of the missing functions, and manually include them in the verification with the rest of the source code. Manual stubs can better emulate 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.

Example 9.1. Stubbing Example

```
procedure a_missing_function
(dest: in out integer,
src : in integer);
procedure test is
a: integer;
b: integer;
begin
a: = 1;
b: = 0;
a_missing_function(a,b);
b:= 1/a;
end;
```

Due to automatic stubbing, the verification assumes that *a* can be have take a value from the full range of integers, including 0. This assumption produces an orange check on the division.

If you provide an empty manual stub for the function, the division is green. This action 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.

You can also provide a detailed manual stub that emulates the behavior of the function.

Emulating Function Behavior with Manual Stubs

You can improve both the speed and selectivity of your verification by providing manual stubs that emulate 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 how the code interacts with the remainder of the system.

Example 9.2. Example

This example shows a header for a missing function (which may occur when the verified code is an incomplete subset of a project).

```
procedure a_missing_function
        (dest: in out integer,
        src  : in integer);
```

Applying fine-level modeling of constraints in primitives and outside functions at the application periphery propagates more precision throughout the application, which results in a higher selectivity rate (more proven colors, i.e. more red+ green + gray). For this function, you could add a simple body:

```
procedure a_missing_function
        (dest: in out integer,
        src : in integer)
begin
        dest := src;
end;
```

In this case, instead of considering the full range for the `dest` parameter, Polyspace considers the relation between input parameter `src` and the output parameter, propagating more precision throughout the application.

Reducing Orange Checks with Empty Stubs

Providing empty manual stubs can reduce the number of orange checks in your results.

For example, consider the following code:

```
package automatic_vs_manual_stub is

     procedure write_or_not1(x : in out Integer);
     procedure write_or_not2(x : in out Integer);
     procedure green;
     procedure orange;

end;

package body automatic_vs_manual_stub is

     procedure write_or_not2(x : in out Integer) is
     begin
           null;
     end;


     procedure orange is
           x : Integer;
           y : Integer;
     begin
           x := 12;
           y := 1;
           write_or_not1(x);
           y := y/x;   -- Orange ZDV due to automatic stub
     end;

     procedure green is
           x : Integer;
           y : Integer;
     begin
           x := 12;
           y := 1;
           write_or_not2(x);
           y := y/x;    -- Green due to empty stub
     end;

end;
```

The code for the two functions is identical, but the automatic stub produces an orange check, while the empty stub produces a green.

While the empty stub reduces the number of orange checks in your results, you must take additional steps to ensure that the actual function does not result in a run-time error.

Applying Constraints to Variables Using Stubs

Another way to increase the selectivity is to indicate to the Polyspace software that some variables may lie within smaller functional ranges instead of the full range of the considered type.

This smaller function range primarily concerns two items from the language:

- Parameters passed to functions.
- Variables' content, mostly globals, which might change from one execution to another. Typically, these might include things like calibration data or mission specific data. These variables might be read directly within the code, or read through an API of functions.

Reduce the cloud of points

If a function is supposed to return an integer, the default automatic stubbing stubs it on the assumption that it can potentially take a value from the full range of an integer.

Polyspace models data ranges throughout the code it verifies. It produces more precise, informative results provided that the data it considers from the “outside world” is representative of the data that can be expected when the code is implemented. There is a certain number of mechanisms available to model such a data range within the code itself, and there are three possible approaches.

with volatile and assert	**with assert and without volatile**	**without assert, without volatile, without "if"**

`function stub return INTEGER is`
`tmp: INTEGER;`
`random: INTEGER;`
`pragma volatile (random);`
`begin`
`tmp:= random;`
`pragma assert (tmp>=1);`
`pragma assert (tmp<=10);`
`return tmp;`
`end;`
	`function random return INTEGER;`
	`pragma Interface (C, random);`
	`function stub return INTEGER is`
	`tmp: INTEGER;`
	`begin`
	`tmp:= random;`
	`pragma assert (tmp>=1);`
	`pragma assert (tmp<=10);`
	`return tmp;`
	`end;`
		`function random return INTEGER;`
		`pragma Interface (C, random);`
		`function stub return INTEGER is`
		`tmp: INTEGER;`
		`begin`
		`tmp:= random;`
		`while (tmp<1 or tmp>10)`
		`loop`
		`tmp:=random;`
		`end loop;`
		`return tmp;`
		`end;`

The three approaches are equivalent (except, perhaps, that the assertions in the first two usually generate orange checks).

Specify Multitasking Behavior

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 `T2` to `X`.
- `T2` divides a local variable by `X`.
- A division by zero error is possible because `T1` can be started before or after `T2`, so the division causes an orange `Division by Zero` error.

The verification cannot determine if an error will occur without knowing the call sequence. Modeling the task differently could turn this orange check green or red.

For more information, see “Modelling Synchronous Tasks” on page 5-9.

Verifying Code in the Eclipse IDE

- “Install Polyspace Plug-In for Eclipse IDE” on page 10-2
- “Configure Verification” on page 10-5
- “Run Verification” on page 10-6
- “Review Results” on page 10-8

Install Polyspace Plug-In for Eclipse IDE

You can install the Polyspace plug-in only if you have already set up the Eclipse Integrated Development Environment (IDE). For information about downloading and installing the Eclipse IDE, go to www.eclipse.org.

In addition to the Eclipse IDE, you must have:

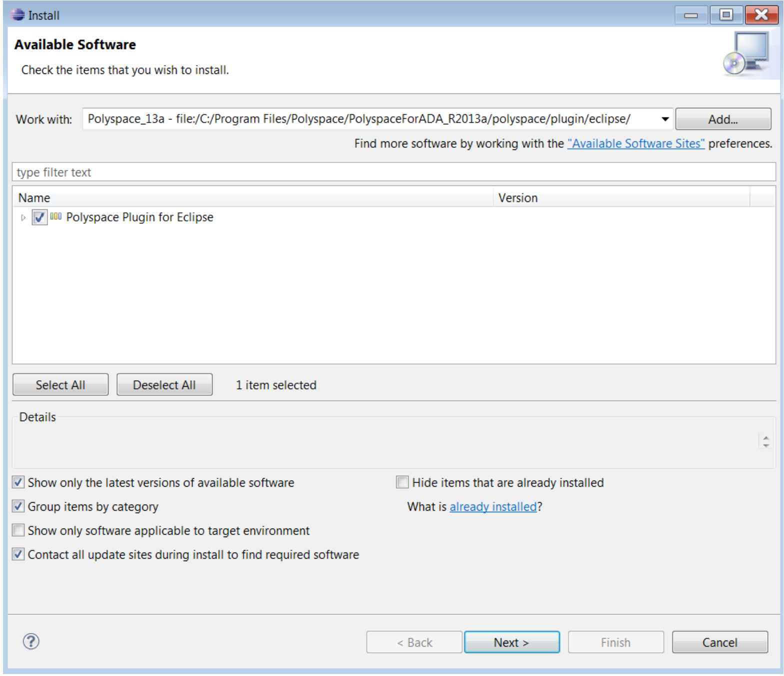
- The GNATbench 2.5.1 plug-in. For more information, go to www.adacore.com.
- A GNAT compiler, a free compiler for Ada95 that is integrated into the GCC compiler system. For more information, go to www.gnu.org/software/gnat/.

Note On a Windows system, the GNATbench plug-in supports only the 32-bit version of the Eclipse IDE. Therefore, on a 64-bit Windows machine, you must install the 32-bit version of the Polyspace product. From a DOS command window, run the following command:

```
DVD\Installer32bits\Windows\Disk1\InstData\VM\Polyspace.exe
```

To install the Polyspace plug-in:

1. From the Eclipse editor, select **Help > Install New Software**. The Install wizard opens, displaying the Available Software page.
2. Click **Add**, which opens the Add Repository dialog box.
3. In the **Name** field, specify a name for your Polyspace site, for example, `Polyspace_13a`.
4. Click **Local**, which opens the Browse for Folder dialog box.
5. Navigate to the *`Polyspace_Install`*`\polyspace\plugin\eclipse` folder. Then click **OK**.
6. Click **OK**, which closes the Add Repository dialog box.
7. On the Available Software page, select `Polyspace Plugin for Eclipse`.



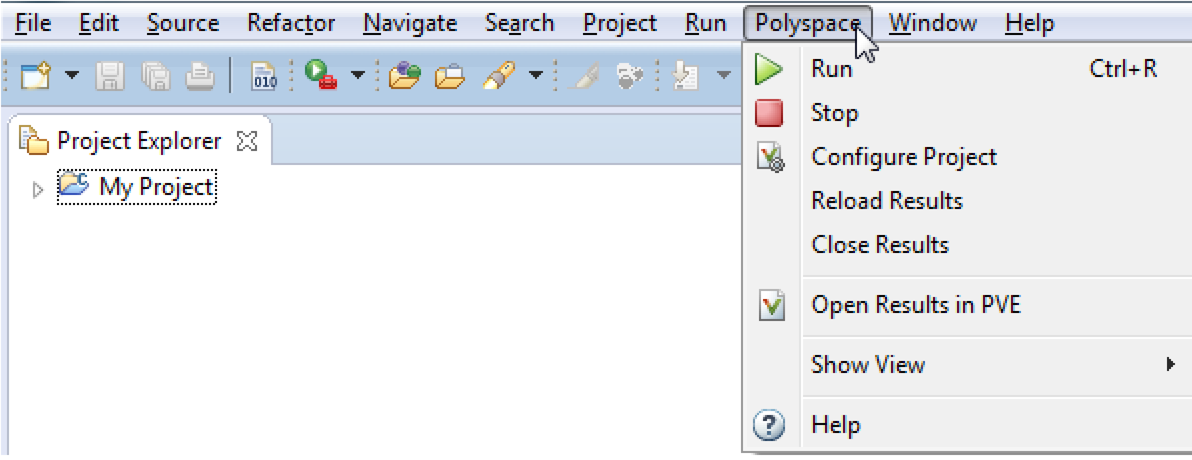
8 Click **Next**.

9 On the Install Details page, click **Next**.

10 On the Review Licenses page, review and accept the license agreement. Then click **Finish**.

Once you install the Polyspace plug-in, in the Eclipse editor, you have access to:

- A **Polyspace** menu
- A **Polyspace Run** view



See Also

Related Examples

- “Configure Verification” on page 10-5

- "Run Verification" on page 10-6
- "Review Results" on page 10-8

Configure Verification

This example shows how to set up a Polyspace verification within the Eclipse Integrated Development Environment (IDE). You can run verification on Ada code and review the verification results without leaving the Eclipse environment.

Before running verification, you can change the default values of the verification options.

Prerequisites

Before you configure your Polyspace verification, you must do the following:

- Install the Polyspace plugin for Eclipse.

  See “Polyspace Plugin Requirements” and “Install Polyspace Plug-In for Eclipse IDE” on page 10-2.
- Set up an Eclipse project containing the source code that you want to verify.

  See Eclipse documentation.

Specify Verification Options

To configure your verification:

1. In **Project Explorer**, select the project or files that you want to verify.
2. Select **Polyspace > Configure Project** to open the **Configuration** pane in the Polyspace user interface.
3. Select your verification options. For more information, see “Analysis Options”.
4. Save your options and close the pane.

Next Steps

After you configure your project, you are ready to run verification. See “Run Verification” on page 10-6.

Run Verification

This example shows how to run a Polyspace verification within the Eclipse Integrated Development Environment (IDE).

Prerequisites

Before you run Polyspace verification, you must do the following:

- Install the Polyspace plugin for Eclipse.

  See “Install Polyspace Plug-In for Eclipse IDE” on page 10-2.
- Set up an Eclipse project for the source code that you want to verify. Configure Polyspace verification for the project.

  See “Configure Verification” on page 10-5.

Start, Monitor and Stop Verification

You can start a Polyspace verification from the Eclipse editor.

1. Switch to the Polyspace perspective.
   a. Select **Window** > **Open Perspective** > **Other**.
   b. In the Open Perspective dialog box, select **Polyspace**.

   This allows you to view only the information related to a Polyspace verification.
2. To start a verification, do one of the following:
   - In the **Project Explorer**, right-click the project containing the files that you want to verify and select **Run Polyspace**.
   - In the **Project Explorer**, select the project containing the files that you want to verify. From the global menu, select **Polyspace** > **Run**.
3. Follow the progress of the verification in the **Polyspace Run** view.

   If you see an error or warning during the compilation phase, double-click it to go to the corresponding location in the source code. Once the verification is over, the results are displayed in the **Results List** view.
4. To stop a verification, select **Polyspace** > **Stop**. Alternatively you can use the button in the **Polyspace Run** view.

The Polyspace files for your Eclipse project, including results and Polyspace configuration files, are saved in the following folder:

`Polyspace_Workspace\Projects\EclipseProjects\Eclipse Project Name`

Here:

- *`Eclipse Project Name`* is the name of your Eclipse project.

- *`Polyspace_Workspace`* is the location where your Polyspace files are stored. You specify this location on the **Project and Results Folder** tab in your Polyspace preferences (**Tools > Preferences** in the Polyspace user interface).

Next Steps

After you run a verification in Eclipse, your results open automatically on the **Results List** view. You have to review each result and determine whether to fix your code or justify the result. See "Review Results" on page 10-8.

Review Results

This example shows how to review the results of a Polyspace verification within the Eclipse Integrated Development Environment (IDE).

After you run a verification in Eclipse, your results open automatically on the **Results List** view. You have to review each result and determine whether to fix your code or justify the result.

Prerequisites

To see results from a Polyspace verification, you must do the following:

- Install the Polyspace plugin for Eclipse.

  See “Install Polyspace Plug-In for Eclipse IDE” on page 10-2.
- Set up an Eclipse project for the source code that you want to verify. Configure Polyspace verification for the project and run verification.

  See “Run Verification” on page 10-6.

Review Results

1. Select a check to see detailed information on the **Result Details** view.
2. In the **Result Details** view, to see a brief description and examples of the result, click the button next to the result name.
3. If you close Eclipse or run Polyspace on another Eclipse project, your results are closed. To reopen the results for an Eclipse project, select the project in the **Project Explorer** and from the global menu, select **Polyspace > Reload Results**.

Save Multiple Results

The results in Eclipse are overwritten every time a new verification is performed. However, Polyspace automatically imports **Status**, **Severity**, and **Comment** information to the new verification results. If you want to save your earlier results:

1. Select **Polyspace > Open Results in PVE** to open your results in the Polyspace user interface.
2. Save your results from the Polyspace user interface.

In addition to the **Results List** and **Result Details** views available in Eclipse, in the Polyspace user interface, you can use other views to:

- View tooltips with information about variable ranges.
- Navigate the call hierarchy easily in your source code.

Glossary

Atomic — In computer programming, the adjective *atomic* describes a unitary action or object that is essentially indivisible, unchangeable, whole, and irreducible.

Batch mode — Execution of Polyspace from the command line rather than through the Polyspace 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 by Polyspace during a verification and subsequently colored red, orange, green or gray in the Run-Time Checks perspective.

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 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 Polyspace verification, so data values possible at execution time are represented by supersets including those values.

Orange check — A warning that represents a possible error which may be revealed upon further investigation.

Polyspace Approach — The manner of use of Polyspace to achieve a particular goal, with reference to a collection of techniques and guiding principles.

Precision — A verification which includes few inconclusive orange checks is said to be precise

Progress text — Output from Polyspace during verification that indicates what proportion of the verification has been completed. Could be considered to be 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 a Polyspace verification.

Scaling option — Option applied when an application submitted to Polyspace Server 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.