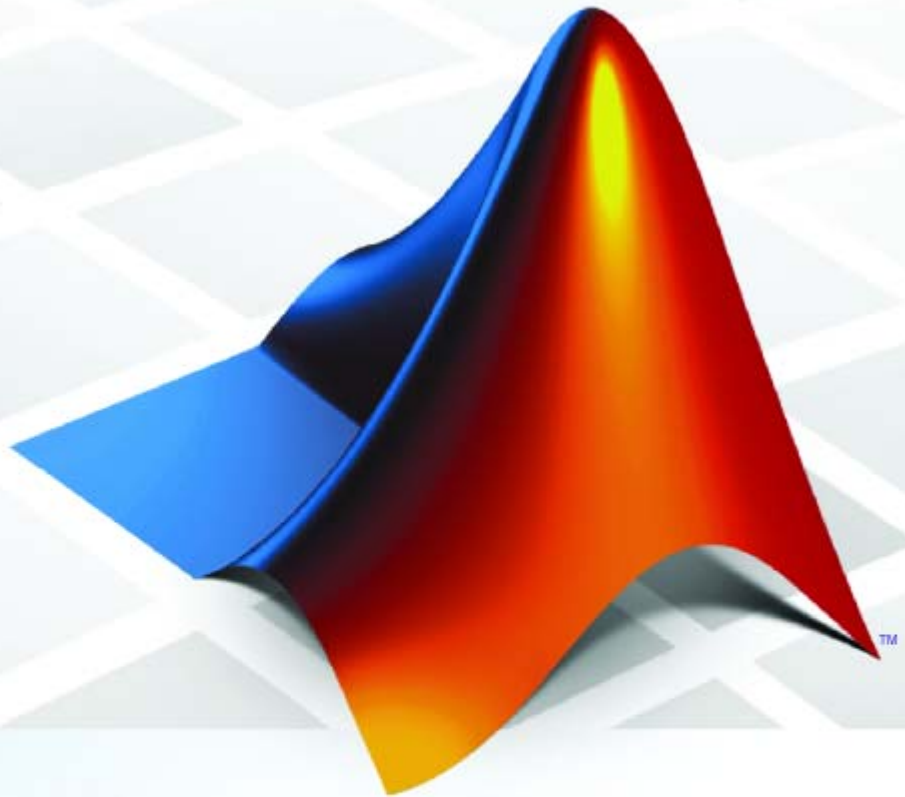


# PolySpace® Products for Ada 5

## User's Guide



## How to Contact The MathWorks



[www.mathworks.com](http://www.mathworks.com) Web  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab) Newsgroup  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html) Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com) Product enhancement suggestions  
[bugs@mathworks.com](mailto:bugs@mathworks.com) Bug reports  
[doc@mathworks.com](mailto:doc@mathworks.com) Documentation error reports  
[service@mathworks.com](mailto:service@mathworks.com) Order status, license renewals, passcodes  
[info@mathworks.com](mailto:info@mathworks.com) Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

*PolySpace® Products for Ada User's Guide*

© COPYRIGHT 1999–2009 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](http://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

The MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://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)

## Introduction to PolySpace Products

### 1

<b>Introduction to PolySpace Products</b> .....	1-2
The Value of PolySpace Verification .....	1-2
How PolySpace Verification Works .....	1-4
Product Components .....	1-5
Installing PolySpace Products .....	1-6
Related Products .....	1-6
<b>PolySpace Documentation</b> .....	1-8
About this Guide .....	1-8
Related Documentation .....	1-8

## How to Use PolySpace Software

### 2

<b>PolySpace Verification and the Software Development Cycle</b> .....	2-2
Software Quality and Productivity .....	2-2
Best Practices for Verification Workflow .....	2-3
<b>Implementing a Process for PolySpace Verification</b> ...	2-4
Overview of the PolySpace Process .....	2-4
Defining Quality Objectives .....	2-5
Defining a Verification Process to Meet Your Objectives ..	2-9
Applying Your Verification Process to Assess Code Quality .....	2-10
Improving Your Verification Process .....	2-10
<b>Sample Workflows for PolySpace Verification</b> .....	2-11
Overview of Verification Workflows .....	2-11
Software Developers – Standard Development Process ...	2-12
Software Developers – Rigorous Development Process ...	2-15

Quality Engineers – Code Acceptance Criteria .....	2-19
Quality Engineers – Certification/Qualification .....	2-22
Model-Based Design Users — Verifying Generated Code ..	2-24
Project Managers — Integrating PolySpace Verification with Configuration Management Tools .....	2-28

## Setting Up a Verification Project

### 3

<b>Creating a Project</b> .....	3-2
What Is a Project? .....	3-2
Project Directories .....	3-3
Opening PolySpace Launcher .....	3-3
Specifying Default Directory .....	3-6
Creating New Projects .....	3-8
Opening Existing Projects .....	3-10
Specifying Source Files .....	3-10
Specifying Include Directories .....	3-13
Specifying Results Directory .....	3-15
Specifying Analysis Options .....	3-16
Configuring Text and XML Editors .....	3-17
Saving the Project .....	3-18
<b>Specifying Options to Match Your Quality</b>	
<b>Objectives</b> .....	3-19
Quality Objectives Overview .....	3-19
Choosing Contextual Verification Options .....	3-19
Choosing Strict or Permissive Verification Options .....	3-20

## Emulating Your Runtime Environment

### 4

<b>Setting Up a Target</b> .....	4-2
Target/Compiler Overview .....	4-2
Specifying Target/Compilation Parameters .....	4-2

Predefined Target Processor Specifications (size of char, int, float, double...) .....	4-3
<b>Verifying an Application Without a “Main” .....</b>	<b>4-6</b>
Main Generator Overview .....	4-6
Automatically Generating a Main .....	4-6
Manually Generating a Main .....	4-7
Example .....	4-7
<b>Using Pragma Assert to Set Data Ranges .....</b>	<b>4-8</b>

## Preparing Source Code for Verification

# 5

<b>Stubbing .....</b>	<b>5-2</b>
Stubbing Overview .....	5-2
Manual vs. Automatic Stubbing .....	5-2
Automatic Stubbing .....	5-5
<b>Preparing Code for Variables .....</b>	<b>5-7</b>
Float Rounding .....	5-7
Expansion of Sizes .....	5-8
Volatile Variables .....	5-8
Shared Variables .....	5-10
<b>Preparing Multitasking Code .....</b>	<b>5-15</b>
PolySpace Software Assumptions .....	5-15
Scheduling Model .....	5-16
Modelling Synchronous Tasks .....	5-17
Interruptions and Asynchronous Events/Tasks .....	5-19
Are Interruptions Maskable or Preemptive by Default? ...	5-21
Mailboxes .....	5-22
Atomicity .....	5-26
Priorities .....	5-27

## Running a Verification

# 6

<b>Types of Verification</b> .....	<b>6-2</b>
<b>Running Verifications on PolySpace Server</b> .....	<b>6-3</b>
Starting Server Verification .....	<b>6-3</b>
What Happens When You Run Verification .....	<b>6-4</b>
Running Verification Unit-by-Unit .....	<b>6-5</b>
Managing Verification Jobs Using the PolySpace Queue Manager .....	<b>6-6</b>
Monitoring Progress of Server Verification .....	<b>6-8</b>
Viewing Verification Log File on Server .....	<b>6-11</b>
Stopping Server Verification Before It Completes .....	<b>6-13</b>
Removing Verification Jobs from Server Before They Run .....	<b>6-14</b>
Changing Order of Verification Jobs in Server Queue .....	<b>6-15</b>
Purging Server Queue .....	<b>6-16</b>
Changing Queue Manager Password .....	<b>6-17</b>
Sharing Server Verifications Between Users .....	<b>6-18</b>
<b>Running Verifications on PolySpace Client</b> .....	<b>6-21</b>
Starting Verification on Client .....	<b>6-21</b>
What Happens When You Run Verification .....	<b>6-22</b>
Monitoring the Progress of the Verification .....	<b>6-23</b>
Stopping Client Verification Before It Completes .....	<b>6-24</b>
<b>Running Verifications from Command Line</b> .....	<b>6-26</b>
Launching Verifications in Batch .....	<b>6-26</b>
Managing Verifications in Batch .....	<b>6-26</b>

## Troubleshooting Verification Problems

# 7

<b>Verification Process Failed Errors</b> .....	<b>7-2</b>
Overview .....	<b>7-2</b>
Hardware Does Not Meet Requirements .....	<b>7-2</b>
You Did Not Specify the Location of Included Files .....	<b>7-2</b>

PolySpace Software Cannot Find the Server .....	7-3
Limit on Assignments and Function Calls .....	7-4
<b>Compile Errors</b> .....	7-6
Overview .....	7-6
Examining the Compile Log .....	7-6
Unit Verification .....	7-8
<b>Reducing Verification Time</b> .....	7-9
PolySpace Verification Duration .....	7-9
An Ideal Application Size .....	7-9
Why Should there be an Optimum Size? .....	7-10
Selecting a Subset of Code .....	7-11
What are the Benefits of these Methods? .....	7-17
<b>Obtaining Configuration Information</b> .....	7-20
<b>Removing Preliminary Results Files</b> .....	7-22

## Reviewing Verification Results

# 8

<b>Before You Review PolySpace Results</b> .....	8-2
Overview: Understanding PolySpace Results .....	8-2
Why Gray Follows Red and Green Follows Orange .....	8-3
What is the Message and What does it Mean? .....	8-4
What is the Ada Explanation? .....	8-5
<b>Opening Verification Results</b> .....	8-8
Downloading Results from Server to Client .....	8-8
Downloading Server Results to UNIX or Linux Clients .....	8-11
Downloading Results from Unit-by-Unit Verifications .....	8-12
Opening Verification Results .....	8-12
Exploring the Viewer Window .....	8-13
Selecting Viewer Mode .....	8-17
Setting Character Encoding Preferences .....	8-17
<b>Reviewing Results in Assistant Mode</b> .....	8-21

What Is Assistant Mode? .....	8-21
Switching to Assistant Mode .....	8-21
Selecting the Methodology and Criterion Level .....	8-22
Exploring Methodology for Ada .....	8-23
Defining a Custom Methodology .....	8-25
Reviewing Checks .....	8-26
Saving Review Comments .....	8-28
<b>Reviewing Results in Expert Mode .....</b>	<b>8-29</b>
What Is Expert Mode? .....	8-29
Switching to Expert Mode .....	8-29
Selecting a Check to Review .....	8-30
Displaying the Calling Sequence .....	8-31
Displaying the Access Sequence for Variables .....	8-32
Tracking Review Progress .....	8-33
Making the Reviewed Column Visible .....	8-35
Filtering Checks .....	8-38
Types of Filters .....	8-38
Creating a Custom Filter .....	8-40
Saving Review Comments .....	8-41
<b>Importing and Exporting Review Comments .....</b>	<b>8-42</b>
Reusing Review Comments .....	8-42
Exporting Review Comments to Other Verification Results .....	8-42
Importing Review Comments from Previous Verifications .....	8-43
<b>Generating Reports of Verification Results .....</b>	<b>8-45</b>
PolySpace Report Generator Overview .....	8-45
Generating Verification Reports .....	8-46
Automatically Generating Verification Reports .....	8-47
Generating Excel Reports .....	8-48
<b>Using PolySpace Results .....</b>	<b>8-52</b>
Review Runtime Errors: Fix Red Errors .....	8-52
Review Dead Code Checks: Why Gray Code is Interesting .....	8-53
Reviewing Orange: Automatic Methodology .....	8-55
Reviewing Orange Checks .....	8-57
Integration Bug Tracking .....	8-57
How to Find Bugs in Unprotected Shared Data .....	8-58



Dataflow Verification .....	8-59
Potential Side Effect of a Red Error .....	8-59
Checks on Procedure Calls with Default Parameters .....	8-60
_INIT_PROC Procedures .....	8-62

## Managing Orange Checks

# 9

<b>Understanding Orange Checks</b> .....	9-2
What is an Orange Check? .....	9-2
Sources of Orange Checks .....	9-6
<b>Too Many Orange Checks?</b> .....	9-9
Do I Have Too Many Orange Checks? .....	9-9
How to Manage Orange Checks .....	9-10
<b>Reducing Orange Checks in Your Results</b> .....	9-11
Overview: Reducing Orange Checks .....	9-11
Applying Coding Rules to Reduce Orange Checks .....	9-12
Improving Verification Precision .....	9-12
Stubbing Parts of the Code Manually .....	9-16
Describing Multitasking Behavior Properly .....	9-21
<b>Reviewing Orange Checks</b> .....	9-23
Overview: Reviewing Orange Checks .....	9-23
Defining Your Review Methodology .....	9-23
Performing Selective Orange Review .....	9-24
Importing Review Comments from Previous Verifications .....	9-27
Performing an Exhaustive Orange Review .....	9-28

## Day to Day Use

# 10

<b>PolySpace In One Click Overview</b> .....	10-2
--	------

<b>Using PolySpace In One Click</b> .....	<b>10-3</b>
PolySpace In One Click Workflow .....	<b>10-3</b>
Setting the Active Project .....	<b>10-3</b>
Launching Verification .....	<b>10-5</b>
Using the Taskbar Icon .....	<b>10-9</b>

## **Glossary**

|

## **Index**

|

# Introduction to PolySpace Products

---

- “Introduction to PolySpace Products” on page 1-2
- “PolySpace Documentation” on page 1-8

## Introduction to PolySpace Products

In this section...
“The Value of PolySpace Verification” on page 1-2
“How PolySpace Verification Works” on page 1-4
“Product Components” on page 1-5
“Installing PolySpace Products” on page 1-6
“Related Products” on page 1-6

### The Value 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.

PolySpace verification can help you to:

- “Ensure Software Reliability” on page 1-2
- “Decrease Development Time” on page 1-3
- “Improve the Development Process” on page 1-4

### Ensure Software Reliability

PolySpace software ensures 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.

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

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

- Might have an error

With this information, you can be confident that you know how much of your code is run-time error free, 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.

A graphical user interface helps you to efficiently review verification results. Results are color-coded:

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

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

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

Reviewing 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 runtime errors. Static verification derives the dynamic properties of a program without actually executing it. This differs significantly from other techniques, such as runtime debugging, in that the verification it provides is not based on a given test case or set of test cases. The dynamic properties obtained in the PolySpace verification are true for all executions of the software.

## What is Static Verification

Static Verification is a broad term, and is applicable to any tool which derives dynamic properties of a program without actually executing it. However, most Static Verification tools only verify the complexity of the software, in a search for constructs which may be potentially dangerous. PolySpace verification provides deep-level verification identifying almost all runtime errors and possible access conflicts on global shared data.

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

For example, consider the following code:

```
for (i=0 ; i<1000 ; ++i)
```

```
{   tab[i] = foo(i);  
}
```

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

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

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

Static code verification has an exact solution but it is generally not practical, as it would in general require the enumeration of all possible test cases. As a result, approximation is required if a usable tool is to result.

## **Exhaustiveness**

Nothing is lost in terms of exhaustiveness. The reason is that PolySpace works by performing upper approximations. In other words, the computed variation domain of any program variable is always a superset of its actual variation domain. The direct consequence is that no runtime error (RTE) item to be checked can be missed by PolySpace.

## **Product Components**

The PolySpace products for verifying Ada code are:

- “PolySpace® Client for Ada Software” on page 1-6
- “PolySpace® Server for Ada Software” on page 1-6

## **PolySpace Client for Ada Software**

PolySpace Client software is the management and visualization tool of PolySpace products. You use the client to submit jobs for execution by PolySpace Server, and to review verification results. The PolySpace Client software includes the Launcher, Viewer, and Report Generator features.

PolySpace client software is typically installed on developer workstations that will send verification jobs to the PolySpace server.

## **PolySpace Server for Ada Software**

PolySpace Server software is the computational engine of PolySpace products. You use it to run jobs posted by PolySpace Clients, and to manage multiple servers and queues. The PolySpace Server software includes the Remote Launcher, Spooler, Report Generator, and HTML Generator features.

PolySpace Server software is typically installed on machines dedicated to PolySpace software that will receive verifications coming from PolySpace clients.

## **Installing PolySpace Products**

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

## **Related Products**

- “PolySpace Products for Verifying C and C++ Code” on page 1-6
- “PolySpace Products for Linking to Models” on page 1-7

## **PolySpace Products for Verifying C and C++ Code**

For information about PolySpace products that verify C and C++ code, see the following:

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

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



## **PolySpace Products for Linking to Models**

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

<http://www.mathworks.com/products/polyspacemodels1/>

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

## PolySpace Documentation

In this section...
“About this Guide” on page 1-8
“Related Documentation” on page 1-8

### About this Guide

This document describes how to use PolySpace software to verify Ada code, and provides detailed procedures for common tasks. It covers both PolySpace® Client™ for Ada and PolySpace® Server™ for Ada products.

This guide is intended for both novice and experienced users.

---

**Note** This document covers both the **Ada83** and **Ada95** languages. References are simply made to **Ada** throughout the document. When the document invokes a `polyspace-ada` command, you may wish to refer to the `polyspace-ada95` command with the same characteristics.

---

### Related Documentation

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

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

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

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

`/www.mathworks.com/access/helpdesk/help/toolbox/polyspace/polyspace.html`

### **The MathWorks Online**

For additional information and support, see:

`www.mathworks.com/products/polyspace`



# How to Use PolySpace Software

---

- “PolySpace Verification and the Software Development Cycle” on page 2-2
- “Implementing a Process for PolySpace Verification” on page 2-4
- “Sample Workflows for PolySpace Verification” on page 2-11

## PolySpace Verification and the Software Development Cycle

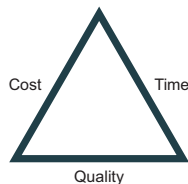
### In this section...

“Software Quality and Productivity” on page 2-2

“Best Practices for Verification Workflow” on page 2-3

### Software Quality and Productivity

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



Changing the requirements for one of these variables always impacts the other two.

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

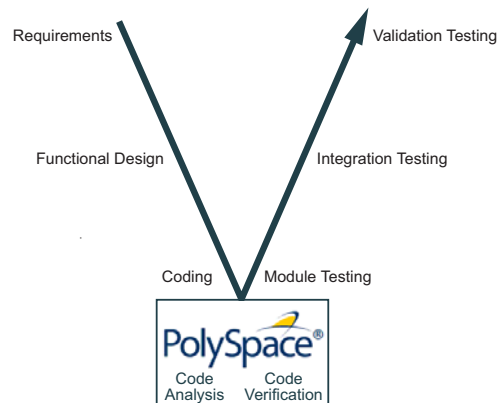
PolySpace verification allows a different process. PolySpace verification can support both productivity improvement and quality improvement at the same time, although there is always a balance between these goals.

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

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

## Best Practices for Verification Workflow

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



### PolySpace Verification in the Development Cycle

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

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

## Implementing a Process for PolySpace Verification

In this section...
“Overview of the PolySpace Process” on page 2-4
“Defining Quality Objectives” on page 2-5
“Defining a Verification Process to Meet Your Objectives” on page 2-9
“Applying Your Verification Process to Assess Code Quality” on page 2-10
“Improving Your Verification Process” on page 2-10

### Overview of the PolySpace Process

PolySpace verification cannot magically produce quality code at the end of the development process. Verification is a tool that helps you measure the quality of your code, identify issues, and ultimately achieve your own quality goals. To do this, however, you must integrate PolySpace verification into your development process.

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

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



## Defining Quality Objectives

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

This process involves:

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

## Choosing Robustness or Contextual Verification

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

- **Robustness Verification** – Prove software works under all conditions.
- **Contextual Verification** – Prove software works under normal working conditions.

---

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

---

**Robustness Verification.** Robustness verification proves that the software works under all conditions, including “abnormal” conditions for which it was not designed. This can be thought of as “worst case” verification.

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

- Assumes function inputs are full range
- Initializes global variables to full range

- Automatically stubs missing functions

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

**Contextual Verification.** Contextual verification proves that the software works under predefined working conditions. This limits the scope of the verification to specific variable ranges, and verifies the code within these ranges.

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

- Create a detailed main program to model the call sequence, instead of using the default main generator. For more information, see “Verifying an Application Without a “Main”” on page 4-6.
- Provide manual stubs that emulate the behavior of missing functions, instead of using the default automatic stubs. For more information, see “Stubbing” on page 5-2.

### Choosing Strict or Permissive Verification Objectives

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

Options that make verification more strict include:

- **-strict**

Options that make verification more permissive include:

- **-permissive**

For more information on these options, see “Options Description” in the *PolySpace Products for Ada Reference*.

## Defining Software Quality Levels

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

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

### Software Quality Levels

Criteria	Software Quality Levels			
	QL1	QL2	QL3	QL4
Document static information	X	X	X	X
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

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

- **Static Information** – Includes information about the application architecture, the structure of each module, and all files. This information must be documented to ensure that your application is fully verified.
- **Red checks** – Represent errors that occur every time the code is executed.
- **Gray checks** – Represent unreachable code.
- **Orange checks** – Indicate unproven code, meaning a run-time error may occur. PolySpace software allows you to define three criteria levels for

reviewing orange checks in the PolySpace Viewer. For more information, see “Reviewing Results in Assistant Mode”.

- **Dataflow analysis** – Identifies errors such as non-initialized variables and variables that are written but never read. This can include inspection of:
  - Application call tree
  - Read/write accesses to global variables
  - Shared variables and their associated concurrent access protection

## **Defining a Verification Process to Meet Your Objectives**

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

These actions include:

- Setting standards for code development, such as coding rules.
- Setting PolySpace Analysis options to match your quality objectives. See “Creating a Project”.
- Setting review criteria in the PolySpace Viewer to ensure results are reviewed consistently. See “Defining a Custom Methodology”.

### **Applying Your Verification Process to Assess Code Quality**

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

This process includes:

- 1** Launching PolySpace verification on each software component as it is written. See “Using PolySpace In One Click” on page 10-3.
- 2** Reviewing verification results consistently. See “Reviewing Results in Assistant Mode” on page 8-21.
- 3** Saving review comments for each component, so they are available for future review. See “Importing Review Comments from Previous Verifications”.
- 4** Performing additional verifications on each component, as defined by your quality objectives.

### **Improving Your Verification Process**

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

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

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

For more information, see Chapter 9, “Managing Orange Checks”.

## Sample Workflows for PolySpace Verification

### In this section...

“Overview of Verification Workflows” on page 2-11

“Software Developers – Standard Development Process” on page 2-12

“Software Developers – Rigorous Development Process” on page 2-15

“Quality Engineers – Code Acceptance Criteria” on page 2-19

“Quality Engineers – Certification/Qualification” on page 2-22

“Model-Based Design Users — Verifying Generated Code” on page 2-24

“Project Managers — Integrating PolySpace Verification with Configuration Management Tools” on page 2-28

### Overview of Verification Workflows

PolySpace verification supports two objectives at the same time:

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

You can use PolySpace verification in different ways depending on your development context and quality model. The primary difference being how you exploit verification results.

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

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

## Quality Objectives

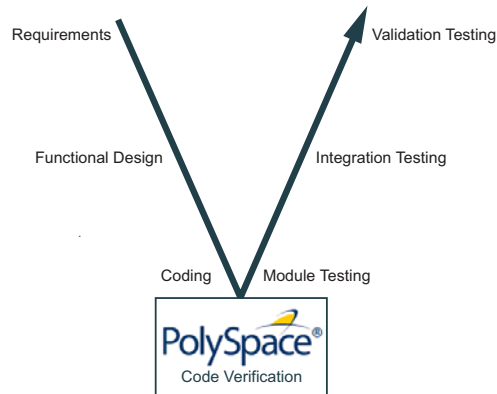
The main goal of PolySpace verification is to improve productivity while maintaining or improving software quality. Verification helps developers find and fix bugs more quickly than other processes. It also improves software quality by identifying bugs that otherwise might remain in the software.

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

## Verification Workflow

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





The verification workflow consists of the following steps:

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

---

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

---

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

---

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

---

Using this approach, it is possible that some bugs may remain in unchecked oranges. However, the verification process represents a significant improvement from 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 needed to set up your verification will be higher if you do not use coding rules. You may need to make modifications to the code before launching 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.

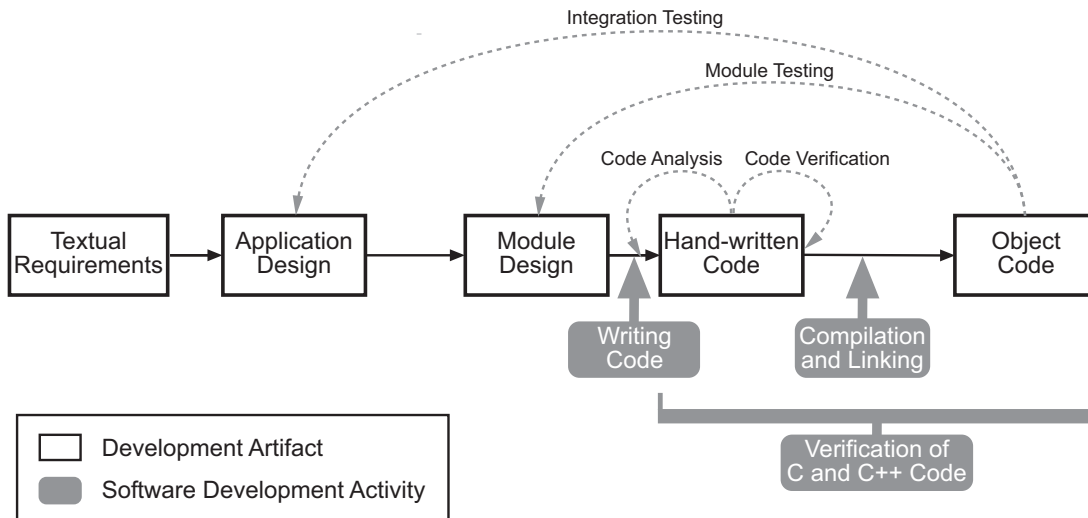
### **Quality Objectives**

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

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

### **Verification Workflow**

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



### Workflow for Code Verification

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

The verification workflow consists of the following steps:

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

- 2 The project leader configures the project to check appropriate coding rules.
- 3 Each developer performs file-by-file verification as they write code, and reviews both coding rule violations and verification results.
- 4 The developer fixes any coding rule violations, fixes all **red** errors, examines **gray** code, and performs a selective orange review.
- 5 The developer repeats steps 2 and 3 as needed, while completing the code.
- 6 Once a developer considers a file complete, they perform a final verification.
- 7 The developer 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. This represents a step towards an exhaustive orange check review. However, spending more time is unlikely to be efficient, and will not guarantee that no bugs remain.
- An exhaustive orange review 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 Objectives**

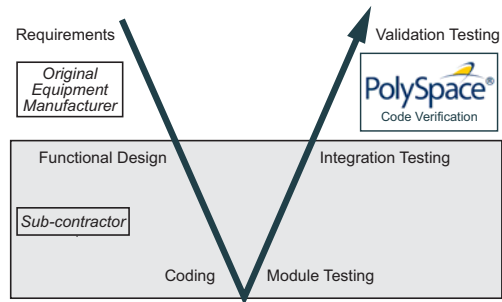
The main goal of PolySpace verification is to control and evaluate the safety of an application.

The criteria used to evaluate code can vary widely depending on the criticality of the application, from no red errors to exhaustive oranges review. Typically, these criteria become increasingly stringent as a project advances from early, to intermediate, and eventually to final delivery.

For more information on defining these criteria, see “Defining Software Quality Levels” on page 2-7.

### **Verification Workflow**

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



---

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

---

The verification workflow consists of the following steps:

- 1** Quality engineering group defines clear quality objectives for the code to be written, including specific quality levels for each version of the code to be delivered (first, intermediate, or final delivery) For more information, see “Defining Quality Objectives” on page 2-5.
- 2** Development group writes code according to established standards.
- 3** Development group delivers software to the quality engineering group.
- 4** The project leader configures the PolySpace project to meet the defined quality objectives, as described in “Defining a Verification Process to Meet Your Objectives” on page 2-9.
- 5** Quality engineers perform verification on the code.
- 6** Quality engineers review all **red** errors, **gray** code, and the number of orange checks defined in the process.



---

**Note** The number of orange checks reviewed often depends on the version of software being tested (first, intermediate, or final delivery). This can be defined by quality level (see “Defining Software Quality Levels” on page 2-7).

---

- 7 Quality engineers create reports documenting the results of the verification, and communicate those results to the supplier.
- 8 Quality engineers repeat steps 5–7 for each version of the code delivered.

### **Costs and Benefits**

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

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

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

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

# Quality Engineers – Certification/Qualification

## User Description

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

These users generally receive code late in the development cycle, and must perform a set of activities to meet certification requirements.

---

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

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

---

## Quality Objectives

The main goal of PolySpace verification is to improve productivity by replacing other qualification activities.

In this context, software quality is already extremely high, so verification is not intended to improve quality. Instead, it is intended to reduce the cost of achieving such quality.

PolySpace verification can increase productivity by replacing existing activities, such as:

- Data and control flow verification
- Shared data conflict detection
- Robustness unit tests

These activities are often performed by hand, or with classical testing methods, which can be time consuming. PolySpace verification can complete

the same tasks more efficiently, bringing improved productivity and reducing the cost of the process.

### **Verification Workflow**

The verification workflow consists of the following steps:

- 1** Developers write code using both coding and data rules.
- 2** The project leader configures the PolySpace project to meet the quality objectives of the certified process.
- 3** Quality engineers perform verification at the unit test stage.
- 4** Quality engineers review all **red** errors, **gray** code, and the number of orange checks defined in the certified process.
- 5** Quality engineers review verification results for data and control flow verification, and shared data detection.
- 6** Optionally, quality engineers perform an additional verification at the integration test phase.

### **Costs and Benefits**

The replacement of these activities can lead to significant cost reductions. For example, the time spent on data and control flow verification can decrease from 3 months to 2 weeks.

Quality is also more consistent since the process is more automated. PolySpace tools are equally efficient on a Friday afternoon and on a Tuesday morning.

# Model-Based Design Users – Verifying Generated Code

## User Description

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

These users generally use PolySpace software in combination with several other Mathworks products, including Simulink, Real-Time Workshop Embedded Coder, and Simulink Design Verifier. In many cases, these customers combine application components that are hand-written code with those created using generated code.

## Quality Objectives

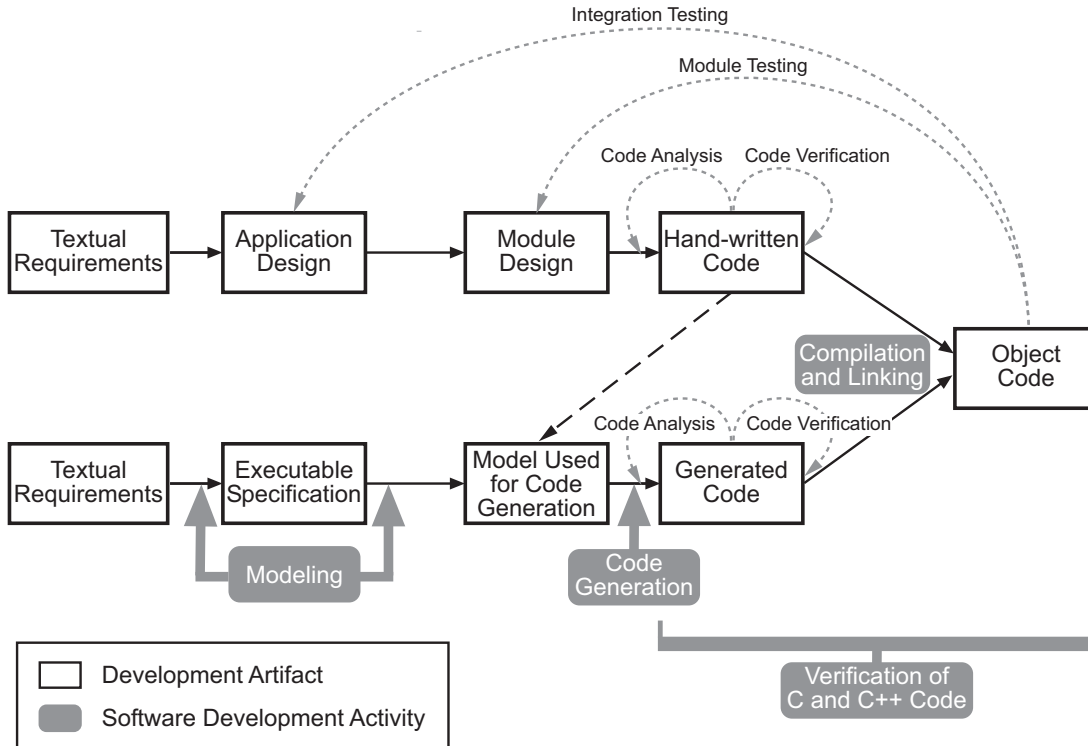
The goal of PolySpace verification is to improve the quality of the software by identifying implementation issues in the code, and ensuring the code is both semantically and logically correct.

PolySpace verification allows you to find run time errors:

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

## Verification Workflow

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



### Workflow for Verification of Generated and Mixed Code

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

The verification workflow consists of the following steps:

- 1** The project leader configures a PolySpace project to meet defined quality objectives.
- 2** Developers write hand-coded sections of the application.
- 3** Developers perform **PolySpace verification** on any hand-coded sections within the generated code, and review verification results according to the established quality objectives.
- 4** Developers create Simulink® model based on requirements.
- 5** Developers validate model to ensure it is logically correct (using tools such as Simulink Model Advisor, and the Simulink® Verification and Validation™ and Simulink® Design Verifier™ products).
- 6** Developers generate code from the model.
- 7** Developers perform **PolySpace verification** on the entire software component, including both hand-written and generated code.
- 8** Developers review verification results according to the established quality objectives.

---

**Note** The PolySpace Model Link™ SL product allows you to quickly track any issues identified by the verification back to the appropriate block in the Simulink model.

---

## Costs and Benefits

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

### Examples of Common Run-Time Errors

Type of Error	Design or Model Errors	Code Errors
Arithmetic errors	<ul style="list-style-type: none"> <li>• Incorrect Scaling</li> <li>• Unknown calibrations</li> <li>• Untested data ranges</li> </ul>	<ul style="list-style-type: none"> <li>• Overflows/Underflows</li> <li>• Division by zero</li> <li>• Square root of negative numbers</li> </ul>
Memory corruption	<ul style="list-style-type: none"> <li>• Incorrect array specification in state machines</li> <li>• Incorrect legacy code (look-up tables)</li> </ul>	<ul style="list-style-type: none"> <li>• Out of bound array indexes</li> <li>• Pointer arithmetic</li> </ul>
Data truncation	<ul style="list-style-type: none"> <li>• Unexpected data flow</li> </ul>	<ul style="list-style-type: none"> <li>• Overflows/Underflows</li> <li>• Wrap-around</li> </ul>
Logic errors	<ul style="list-style-type: none"> <li>• Unreachable states</li> <li>• Incorrect Transitions</li> </ul>	<ul style="list-style-type: none"> <li>• Non initialized data</li> <li>• Dead code</li> </ul>

# Project Managers – Integrating PolySpace Verification with Configuration Management Tools

## User Description

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

## Quality Objectives

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

## Verification Workflow

The verification workflow consists of the following steps:

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



# Setting Up a Verification Project

---

- “Creating a Project” on page 3-2
- “Specifying Options to Match Your Quality Objectives” on page 3-19

## Creating a Project

In this section...
“What Is a Project?” on page 3-2
“Project Directories” on page 3-3
“Opening PolySpace Launcher” on page 3-3
“Specifying Default Directory” on page 3-6
“Creating New Projects” on page 3-8
“Opening Existing Projects” on page 3-10
“Specifying Source Files” on page 3-10
“Specifying Include Directories” on page 3-13
“Specifying Results Directory” on page 3-15
“Specifying Analysis Options” on page 3-16
“Configuring Text and XML Editors” on page 3-17
“Saving the Project” on page 3-18

### What Is a Project?

In PolySpace software, a project is a named set of parameters for a verification of your software project’s source files. You must have a project before you can run a PolySpace verification of your source code.

A project includes:

- The location of source files and include directories
- The location of a directory for verification results
- Analysis options

You can create your own project or use an existing project. You create and modify a project using the Launcher graphical user interface.

A project file has one of the following file types:

Project Type	File Extension	Description
configuration	cfg	Required for running a verification. Does not include generic target processors.
PolySpace Project Model	ppm	For populating a project with analysis options, including generic target processors.
Desktop	dsk	In earlier versions of PolySpace software, for running a verification on a client computer.

## Project Directories

Before you begin verifying your code with PolySpace software, you must know the location of your Ada source package and any other specifications upon which it may depend either directly or indirectly. You must also know where you want to store the verification results.

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

```
polyspace_project/
```

- sources
- includes
- results

## Opening PolySpace Launcher

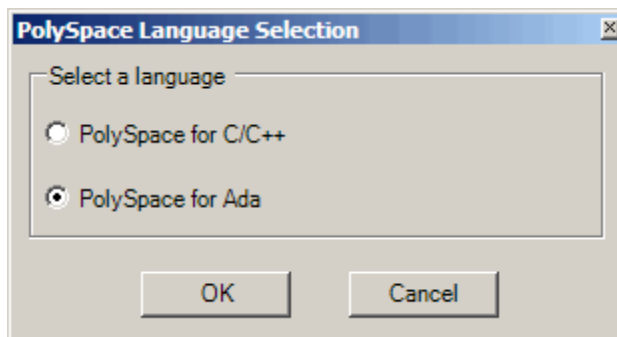
You use the PolySpace Launcher to create a project and start a verification.

To open the PolySpace Launcher:

- 1 Double-click the PolySpace Launcher icon.

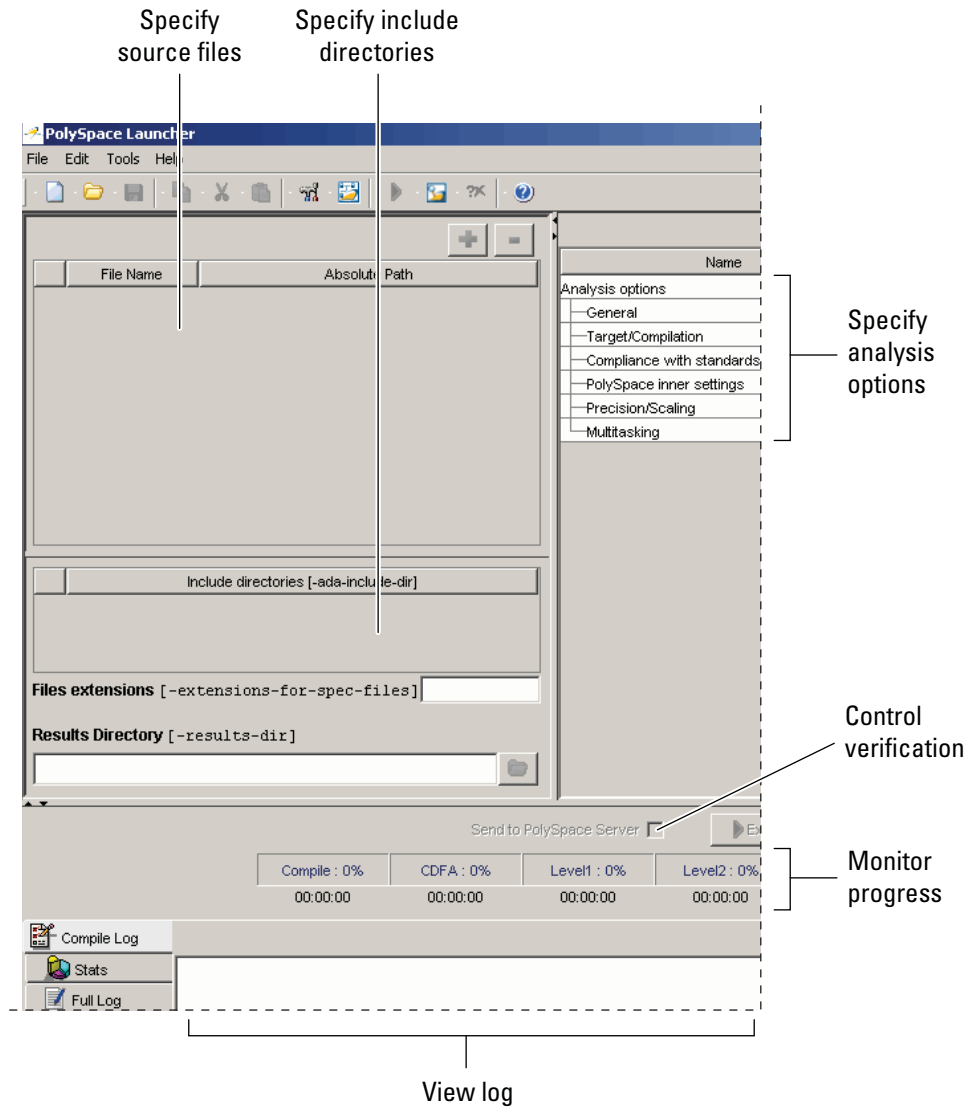


- 2 If you have both PolySpace for C/C++ and PolySpace for Ada products on your system, the **PolySpace Language Selection** dialog box will appear.



Select **PolySpace for Ada**, then click **OK**.

The PolySpace Launcher window appears:



The Launcher window has three main sections.

Use this section...	For...
Upper-left	Specifying: <ul style="list-style-type: none"><li>• Source files</li><li>• Include directories</li><li>• Results directory</li></ul>
Upper-right	Specifying analysis options
Lower	Controlling and monitoring a verification

You can resize or hide any of these sections. You learn more about the Launcher window later in this tutorial.

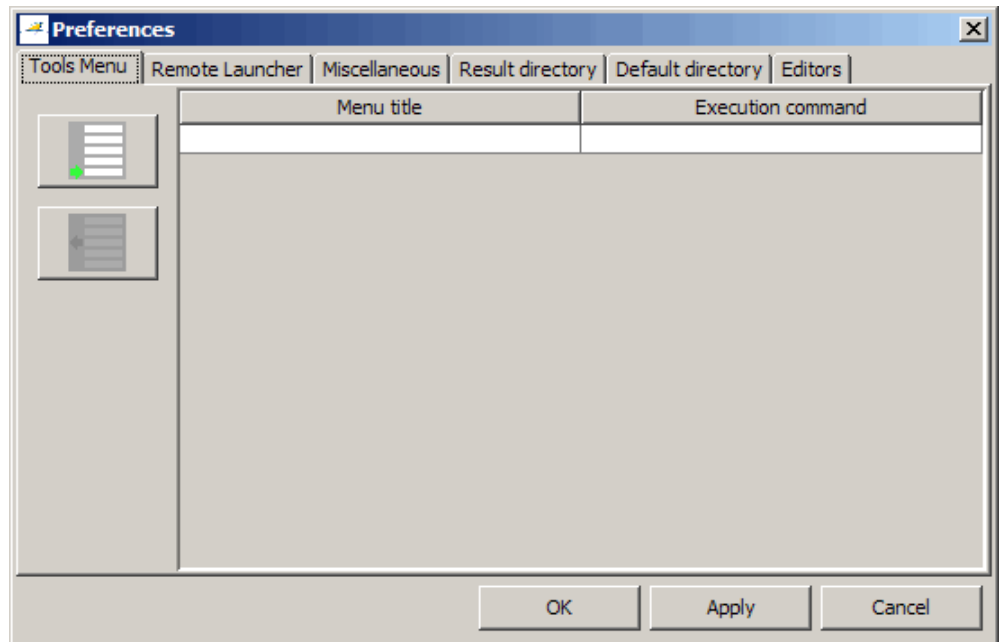
## Specifying Default Directory

PolySpace software allows you to specify the default directory that appears in directory browsers in dialog boxes. If you do not change the default directory, the default directory is the installation directory. Changing the default directory to the project directory makes it easier for you to locate and specify source files and include directories in dialog boxes.

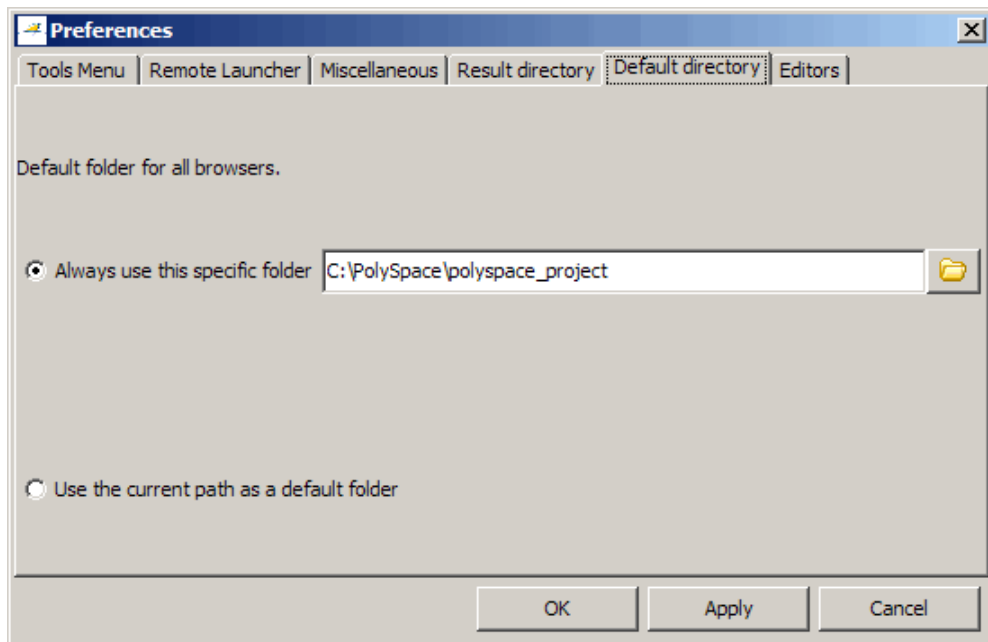
To change the default directory to the project directory:

- 1 Select **Edit > Preferences**.

The **Preferences** dialog box appears.



**2** Select the **Default directory** tab.



**3** Select **Always use this specific folder** if it is not already selected.

**4** Enter or navigate to the project directory you want to use.

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

## Creating New Projects

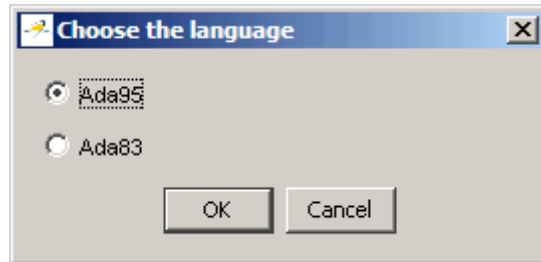
You must have a project, saved with file type `.cfg`, to run a verification.

To create a new project:

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

The **Choose the language** dialog box appears:

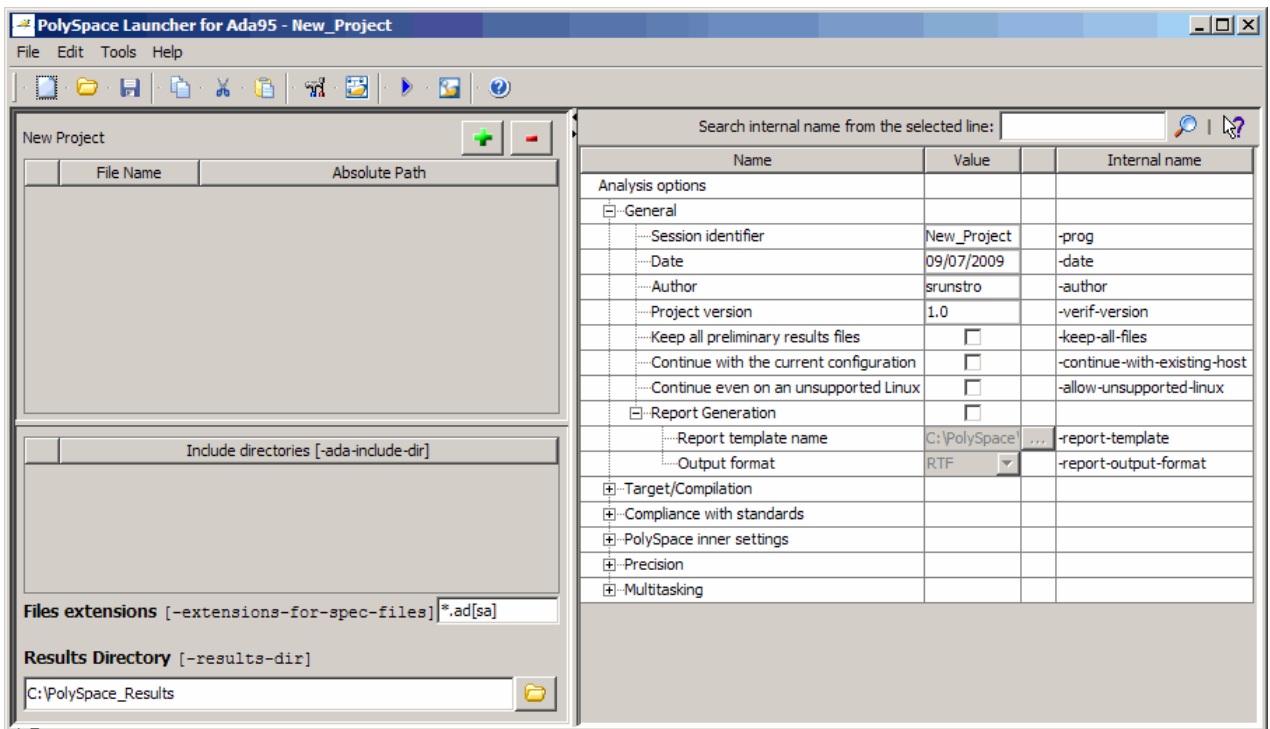




- 2 Select your code type, then click **OK**.

The default project name, `New_Project`, appears in the title bar.

In the **Analysis options** section, the **General** options node expands with default project identification information and options.



## Opening Existing Projects

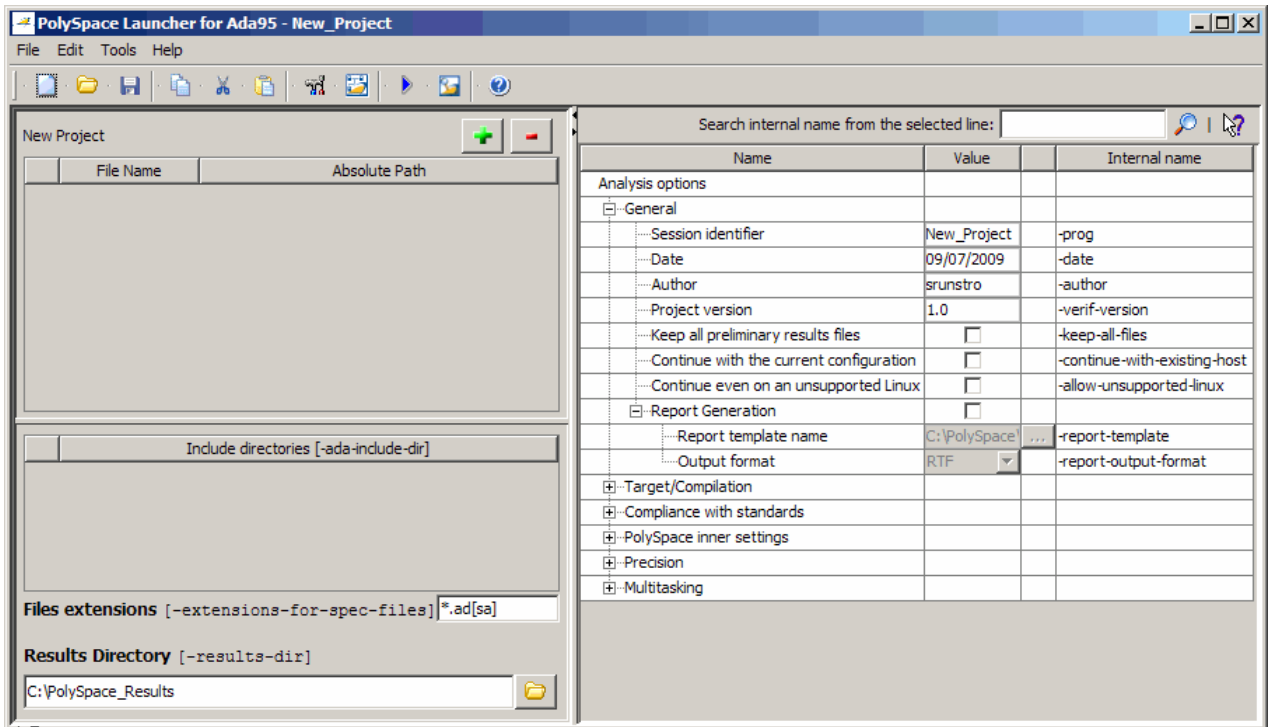
To open an existing project:

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

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

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

The selected project opens in the Launcher.



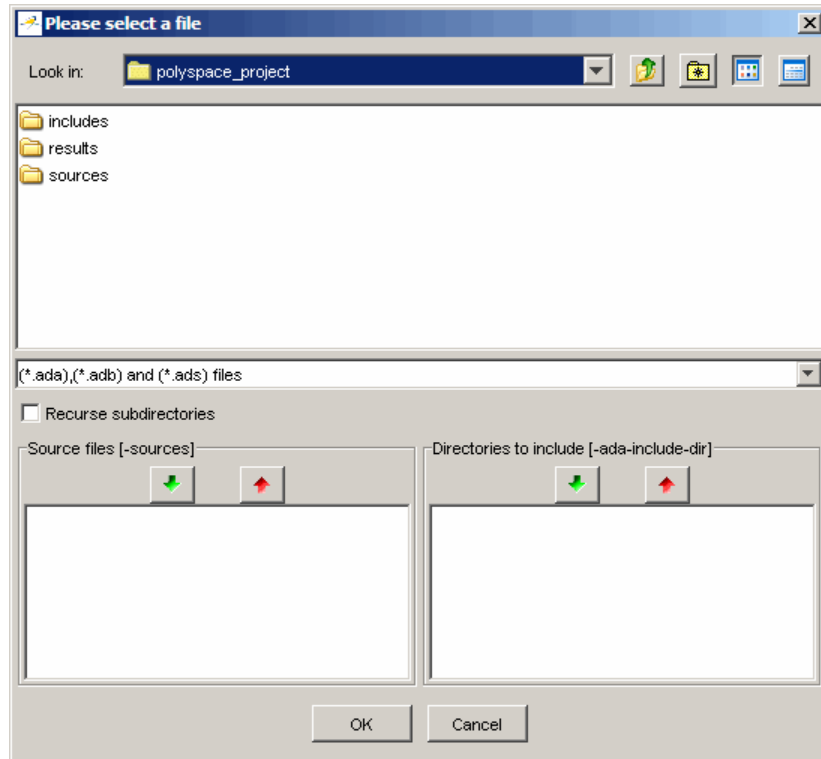
## Specifying Source Files

To specify the source files for your project:

- 1 Click the green plus sign button in the upper right of the files section of the Launcher window.



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



- 2** In the **Look in** field, navigate to your project directory containing your source files.
- 3** Select the files you want to verify, then click the green down arrow button in the **Source files** section.

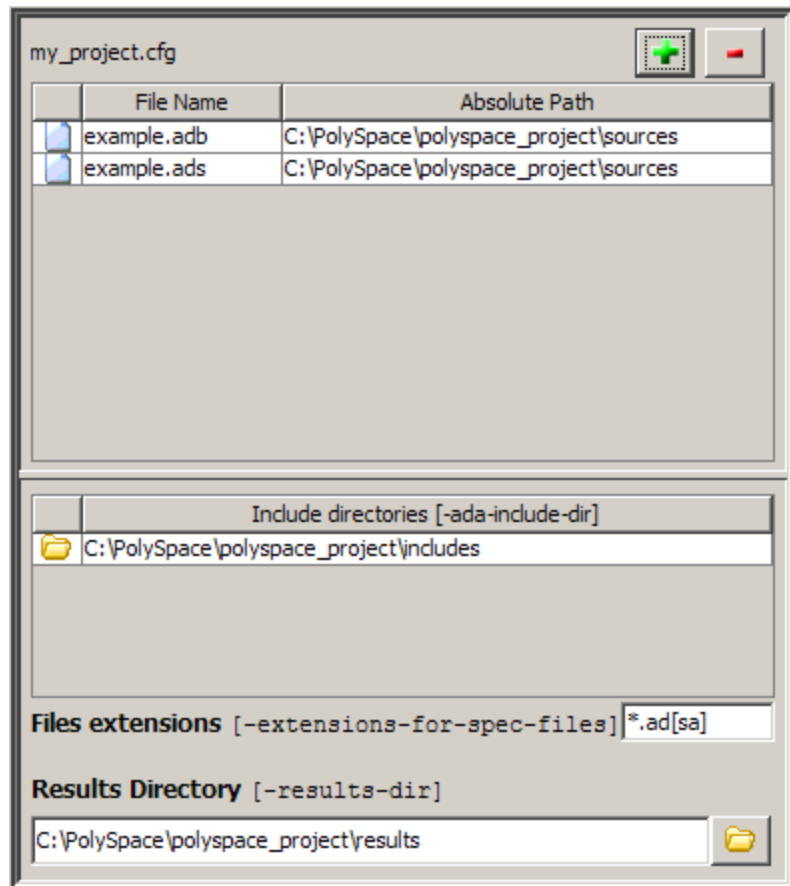


The path of each source files appear in the source files list.

**Tip** You can also drag directory and file names from an open directory directly to the source files list or include list.

- 4 lick **OK** to apply the changes and close the dialog box.

The source files you selected appear in the files section in the upper left of the Launcher window.



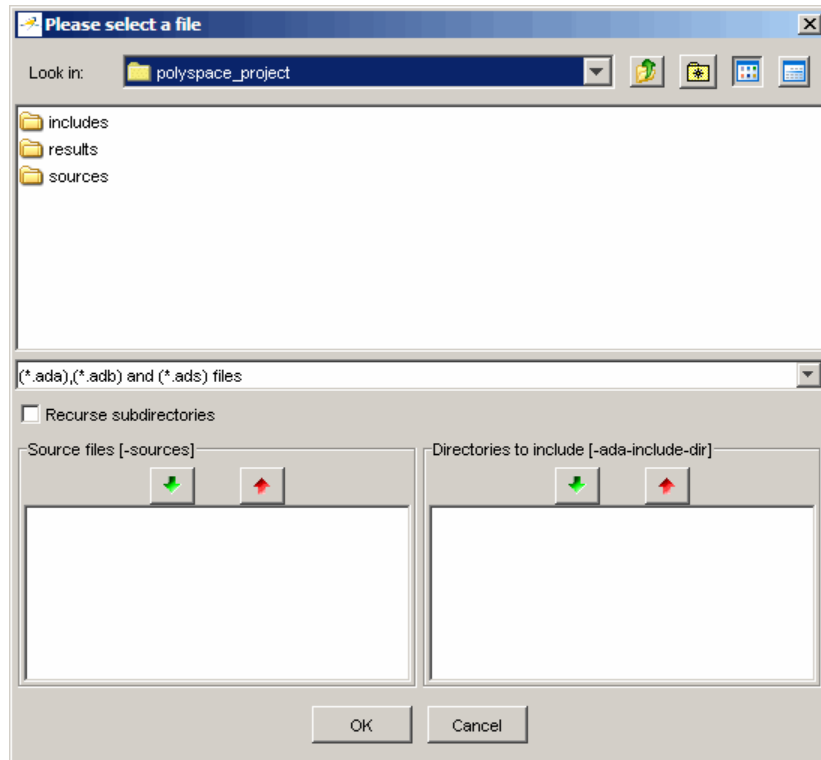
## Specifying Include Directories

To specify the include directories for the project:

- 1 lick the green plus sign button in the upper right of the files section of the Launcher window.



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



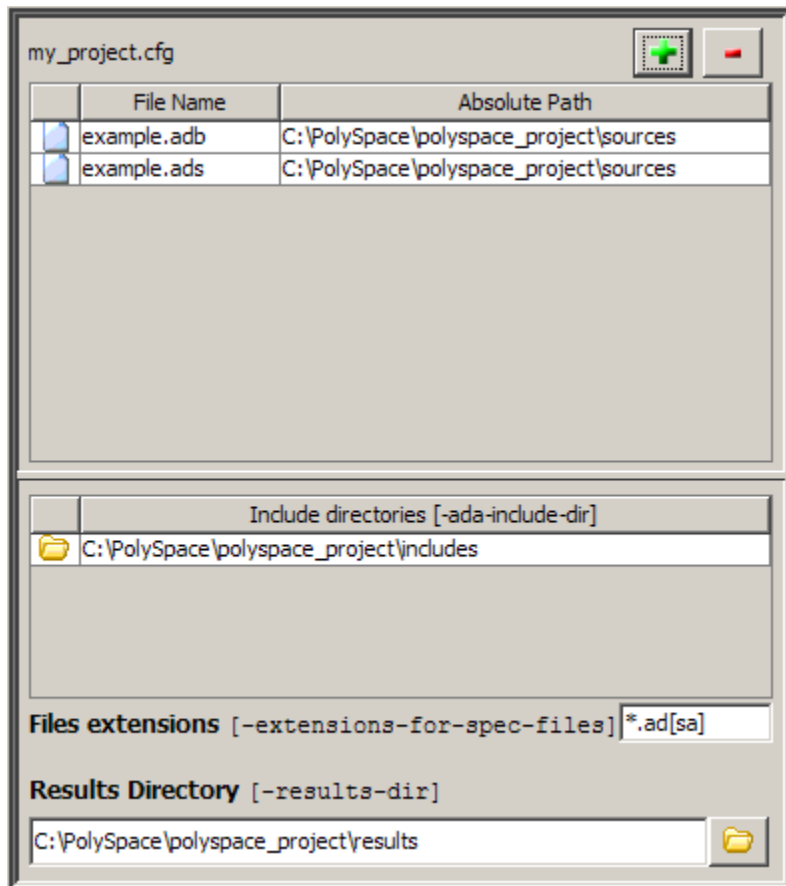
- 2 In the **Look in** field, navigate to your project directory.
- 3 Select the directory containing the include files for your project, then click the green down arrow button in the **Directories to include** section.



The path for each include directory appears in the source files list.

- 4 lick **OK** to apply the changes and close the dialog box.

The include directories you selected appear in the Include directories section on the left side of the Launcher window.

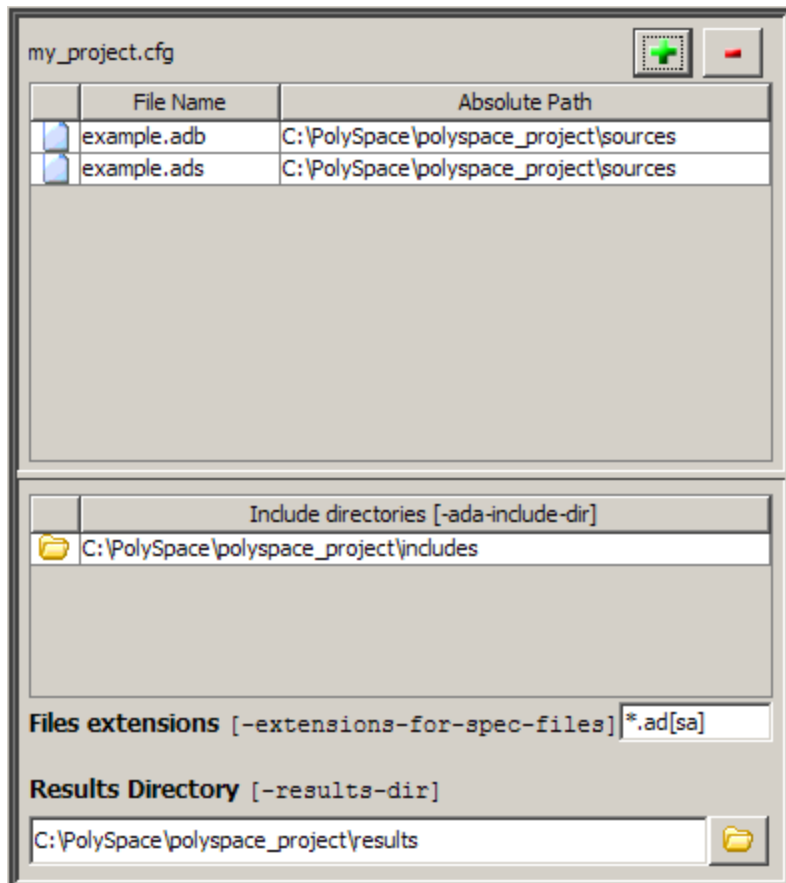


## Specifying Results Directory

To specify the results directory for the project:

- 1 In the **Results Directory** section of the Launcher window, specify the full path of the directory that will contain your verification results. For example: `:\polyspace_project\results`.

The files section of the Launcher window now looks like:



## Specifying Analysis Options

The analysis options in the upper-right section of the Launcher window include identification information and parameters that PolySpace software uses during the verification process.

To specify General parameters for your project:

- 1** In the Analysis options section of the Launcher window, expand **General**.
- 2** The General options appear.

Search internal name from the selected line: <input type="text"/>			
Name	Value		Internal name
Analysis options			
[-] General			
...Session identifier	New_Project		-prog
...Date	07/07/2009		-date
...Author	user_name		-author
...Project version	1.0		-verif-version
...Keep all preliminary results files	<input type="checkbox"/>		-keep-all-files
...Continue with the current configuration	<input type="checkbox"/>		-continue-with-existing-host
...Continue even on an unsupported Linux distribution	<input type="checkbox"/>		-allow-unsupported-linux
[-] Report Generation	<input type="checkbox"/>		
...Report template name	C:\PolySpace\...		-report-template
...Output format	RTF		-report-output-format
[+] Target/Compilation			
[+] Compliance with standards			
[+] PolySpace inner settings			
[+] Precision			
[+] Multitasking			

- 3** Specify the appropriate general parameters for your project.

For detailed information about specific analysis options, see “Option Descriptions” in the *PolySpace Products for Ada Reference*.



## Configuring Text and XML Editors

Before you run a verification, you should configure your text and XML editors in the Launcher. Configuring text and XML editors allows you to view source files directly from the Launcher logs.

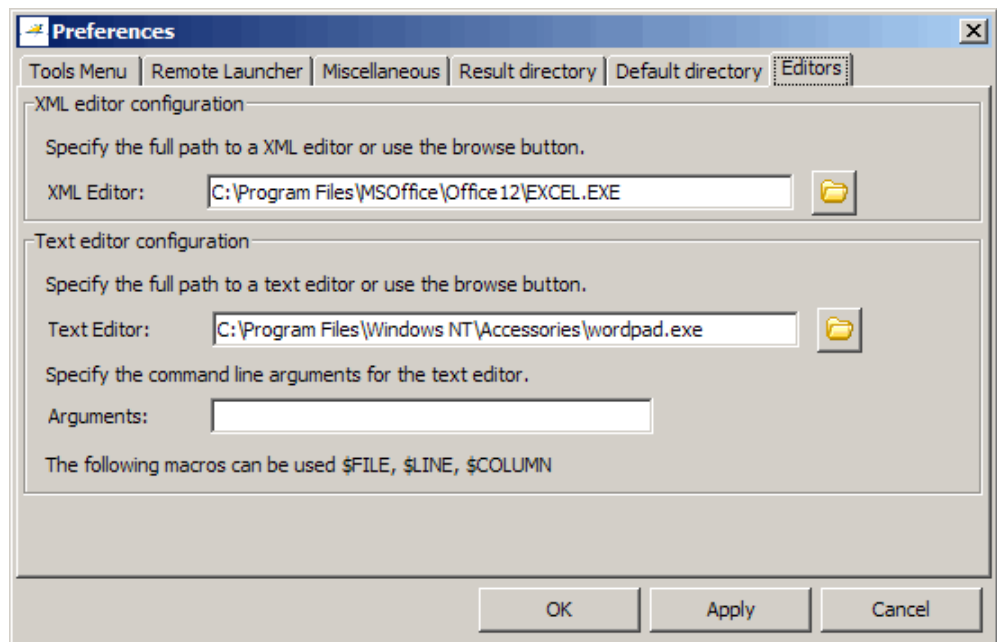
To configure your text and .XML editors:

- 1 Select **Edit > Preferences**.

The Preferences dialog box opens.

- 2 Select the **Editors** tab.

The Editors tab opens.



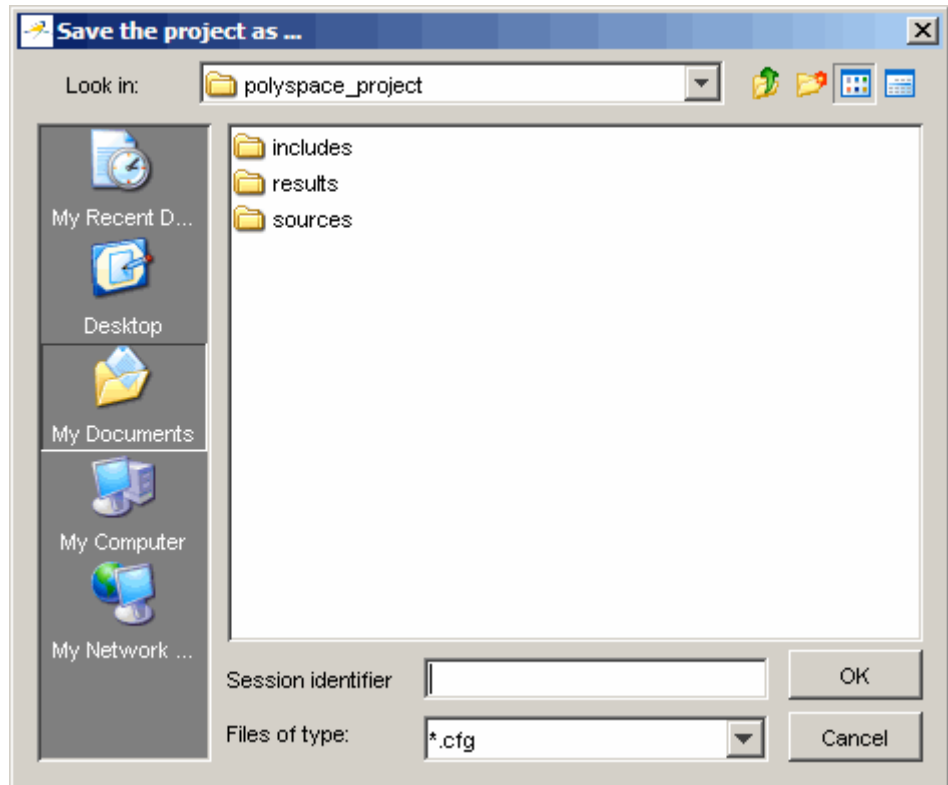
- 3 Specify a Text editor to use to view source files from the Launcher logs.

- 4 lick **OK**.

## Saving the Project

To save the project:

- 1 Select **File > Save project**. The **Save the project as** dialog box appears.



- 2 In **Look in**, select your project directory.
- 3 In **Session identifier**, enter a name for your project.
- 4 lick **OK** to save the project and close the dialog box.

## Specifying Options to Match Your Quality Objectives

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

This includes:

In this section...
“Quality Objectives Overview” on page 3-19
“Choosing Contextual Verification Options” on page 3-19
“Choosing Strict or Permissive Verification Options” on page 3-20

### Quality Objectives Overview

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

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

---

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

---

### Choosing Contextual Verification Options

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

For more information on robustness and contextual verification, see “Choosing Robustness or Contextual Verification” on page 2-5.

To specify contextual verification for your project:

- 1 In the Analysis options section of the Launcher window, expand **PolySpace Inner Settings**.

#### 2 Expand the **Generate a main** and **Stubbing** options.

PolySpace inner settings		
Run a verification unit by unit	<input type="checkbox"/>	-unit-by-unit
Name of the main subprogram	INIT.MAIN	-main
Generate a main	<input checked="" type="checkbox"/>	-main-generator
Stubbing		
Treat import as non volatile	<input type="checkbox"/>	-import-are-not-volatile
Treat export as non volatile	<input type="checkbox"/>	-export-are-not-volatile
No automatic stubbing	<input checked="" type="checkbox"/>	-no-automatic-stubbing
Initialisation of uninitialized global variables	No initi... ▾	-init-stubbing-vars-rando...

3 To control main generation behavior, specify the following options:

- **Generate a main (-main-generator)** – Specifies whether the software automatically generates a main.

4 To control stubbing behavior, use the following options:

- **No automatic stubbing (-no-automatic-stubbing)** – Specifies that the software will not automatically stub functions. The software list the functions to be stubbed and stops the verification.
- **Initialization of uninitialized global variables (-init-stubbing-vars-random)** – Specifies how uninitialized global variables are initialized.

For more information on these options, see “Options Description” in the *PolySpace Products for Ada Reference*.

## Choosing Strict or Permissive Verification Options

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

To specify the strictness of your verification:

- 1 In the Analysis options section of the Launcher window, expand **Compliance with standards**.

**2** In the Analysis mode drop-down menu, select **Strict** or **Permissive**.

Name	Value	Internal name
Analysis options		
+ General		
+ Target/Compilation		
- Compliance with standards		
Value of the constant Storage_Unit		-storage-unit
Remove comparison operators ambiguities	<input type="checkbox"/>	-base-type-directly-visible
<b>Analysis Mode</b>	Permis... ▾	<b>-strict/-permissive</b>
+ PolySpace inner settings	Strict	
+ Precision	Customize	
+ Multitasking	Permissive	

For more information on these options, see “Options Description” in the *PolySpace Products for Ada Reference*.



# Emulating Your Runtime Environment

---

- “Setting Up a Target” on page 4-2
- “Verifying an Application Without a “Main”” on page 4-6
- “Using Pragma Assert to Set Data Ranges” on page 4-8

# Setting Up a Target

In this section...
“Target/Compiler Overview” on page 4-2
“Specifying Target/Compilation Parameters” on page 4-2
“Predefined Target Processor Specifications (size of char, int, float, double...)” on page 4-3

## Target/Compiler Overview

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

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

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

## Specifying Target/Compilation Parameters

The Target/Compilation options in the Launcher allow you to specify the target processor and operating system for your application.

To specify target parameters for your project:

- 1 In the Analysis options section of the Launcher window, expand **Target/Compilation**.
- 2 The Target/Compilation options appear.



Name	Value		
Analysis options			
+ General			
- Target/Compilation			
Target processor type	sparc		-target
Operating system target for Standard Libraries compatibility	no-predefined-OS		-OS-target
Command/script to apply before start of the code verification		...	-pre-analysis-command
Command/script to apply after the end of the code verification		...	-post-analysis-command
+ Compliance with standards			
+ PolySpace inner settings			
+ Precision			
+ Multitasking			

**3** Specify the appropriate parameters for your target CPU and operating system.

For detailed information on each Target/Compilation option, see “Target/Compiler Options” in the PolySpace Products for Ada Reference.

## Predefined Target Processor Specifications (size of char, int, float, double...)

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

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

PolySpace supports some of the most commonly used processors, as listed in the table below. Even if the processor used in a target environment is not explicitly mentioned, it is safe to specify one from the table which shares the same listed characteristics.

Target	sparc	m68k ColdFire	1750a	powerpc 32bits	powerpc 64bits	I386
Character	8	8	16	8	8	8

Target	sparc	m68k ColdFire	1750a	powerpc 32bits	powerpc 64bits	i386
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

- Target powerpc32bits: The largest default alignment of basic types within record/array is 64.
- Target powerpc64bits: The largest default alignment of basic types within record/array is 64.
- Target i386: The largest default alignment of basic types within record/array is 32.

To identify a target processor's characteristics, compile and run the program below. If none of the characteristics described above match, please contact MathWorks Technical Support (<http://www.mathworks.com/support>).

```

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;
```

## Verifying an Application Without a “Main”

In this section...
“Main Generator Overview” on page 4-6
“Automatically Generating a Main” on page 4-6
“Manually Generating a Main” on page 4-7
“Example” on page 4-7

### Main Generator Overview

When your application is a function library (API) or a single module, you must provide a main that calls each non-called procedure within the code, because of the execution model used by PolySpace. You can either manually provide a main, or have PolySpace generate one for you automatically.

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

### Automatically Generating a Main

You can choose to automatically generate a main by selecting the **Generate a main** (-main-generator) option. The -main-generator option will create automatically a procedure which calls every non called procedure within the code, avoiding for instance to create manually a main.

- **PolySpace Client for Ada software** – By default, the software automatically generates a main. You can choose to manually generate a main using the -main option.
- **PolySpace Server for Ada software** – The -main option is set by default. You can choose to automatically generate a main using the -main-generator option.

## Manually Generating a Main

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

There are three steps involved in manually defining the main.

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

## Example

The API spec are:

```
function func1(x in integer) return integer;
```

```
procedure func2(x in out float, y in integer);
```

The main you'll have to create is the following :

```
procedure main is
```

```
  a,b,c,d: integer;
```

```
  e,f: float;
```

```
  pragma volatile (a);
```

```
  pragma volatile (e);
```

```
  -- We need an integer and float variable as a function parameter
```

```
begin
```

```
  loop
```

```
  f := e;
```

```
  c:=a;
```

```
  d:=a;
```

```
    if (a = 1) then b:= func1(c); end if;
```

```
    if (a = 1) then func2(e,d); end if;
```

```
  end loop
```

```
end main;
```

### 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” runtime error. This is why PolySpace will produce checks for them. The behavior matches the one exhibited during execution, because **all execution paths for unsatisfied conditions are truncated** (red and then gray). Thus it can be assumed that any verification performed downstream of the assert uses value ranges which satisfy the assert conditions.

It is therefore possible to use the construct 'pragma assert' in a procedure to inform PolySpace of constraints in the environment in which the software will be embedded. User assertions can be used to describe the physical properties of the environment such as:

- the maximum and minimum speed limit (a car never goes faster than 200 miles per hour or slower than 0),
- the maximum duration of software exploitation (five years for a satellite and one hour for its launcher),
- and so on ...

#### 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. But, from then on, counter ~ [101, 999] because any execution paths that does not meet the conditions are halted.





# Preparing Source Code for Verification

---

- “Stubbing” on page 5-2
- “Preparing Code for Variables” on page 5-7
- “Preparing Multitasking Code” on page 5-15

## Stubbing

In this section...
“Stubbing Overview” on page 5-2
“Manual vs. Automatic Stubbing” on page 5-2
“Automatic Stubbing” on page 5-5

### 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 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 the effect that the code might have on the remainder of the system.

Consider *procedure\_to\_stub*, If it represents:

- a timing constraint, such as a timer set/reset, a task activation, a delay or a counter of ticks between two precise locations in the code, then you can stub it to an empty action (begin null; end; ). PolySpace has no timing constraints and already takes into account all possible scheduling and interleaving and enhances all timing constraints: there is no need to stub functions that set or reset a timer. Simply declare the variable representing time as volatile.
- an I/O access: to a hardware port, a sensor, read/write of a file, read of an eeprom, write to a volatile variable, then: there is no need to stub a write access or simply stub a write access to an empty action (see above), stub read accesses as "I read all possible values (volatile)".
- a write to a global variable, you may need to 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 all 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 cares about the Ada code which is provided.
- PolySpace does not need to be informed of timing constraints because all possible sequencing is taken into account.

### 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;
  -- copy "b" to "a":
  -- 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 in the knowledge that no runtime error will be ever introduced by automatic stubbing; to minimize preparation time.

## Automatic Stubbing

### Problem

What is the default behavior for missing functions?

### Explanation

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

- they are external,
- they are written in C, or any other language than Ada,
- they are part of the system libraries.

PolySpace relies on and trusts their specifications when stubbing them.

### **Solution**

Add the `-automatic-stubbing` option to your launching script and PolySpace will stub missing code as follows:

- for an **in** parameter, nothing happens;
- for an **out** (or **in out**) parameter, the variable will be given the full range of its type;
- for a **return** parameter, it will be the full range of its type.

A procedure with this specification:

```
procedure a_missing_function (a: in out type_1, b: in integer);
```

will be stubbed like so:

```
a_missing_function (var_1, var_2)
```

That is - the "var\_1" variable will be overwritten with the full range of type\_1.

## Preparing Code for Variables

### In this section...

“Float Rounding” on page 5-7

“Expansion of Sizes” on page 5-8

“Volatile Variables” on page 5-8

“Shared Variables” on page 5-10

### Float Rounding

PolySpace handles float rounding by following the ANSI/IEEE 754-1985 standard. Using the `-ignore-float-rounding` option, PolySpace computes exact values of floats. Some paths will be reachable or not for PolySpace while they are not (or are) depending of the compiler and target. So it can potentially give approximate results: green should be unproven. Using the option allows to first have a look on remaining unproven check OVFL.

The Following example shows the board effect of such option:

```
package float_rounding is
  procedure main;
end float_rounding;
package body float_rounding is
  procedure main is
    x : float := float'last;
    random : boolean;
    pragma import(C,random);
  begin
    if random then
      x := x + 5.0 - float'last;
      -- with -ignore-float-rounding : overflow red on + 5.0
      -- without -ignore-float-rounding : overflow orange and x is
      very close to zero
    else
      x := x - 5.0 - float'last;
      -- with -ignore-float-rounding : x is now equal to 5.0
      -- without -ignore-float-rounding : x is very close to zero
    end if;
  end;
```

```
end;  
end float_rounding;
```

### Expansion of Sizes

The `-array-expansion-size` option forces PolySpace to verify each cell of global variable arrays having length less or equal to number as a separate variable.

#### Example

```
Package body Test is  
  Glob_Array_3 : array(1..3) of Integer := (1,2,3);  
  Glob_Array_8 : array(1..8) of Integer := (1,2,3,4,5,6,7,8);  
  procedure Main is  
  begin  
    pragma Assert (Glob_Array_3(3) = 3);  
    pragma Assert (Glob_Array_8(3) = 3);  
  end Main;  
end Test;
```

The `-variable-to-expand` option is used to specify aggregate variables (record, etc.) that will be split into independent variables for the purpose of verification. This option has an impact on the Global Data Dictionary results:

- Each variable specified in this option will have its fields verified separately;
- The data dictionary will distinguish fields accessed by different tasks.

The depth of the variable to expand is controlled by the `-variable-to-expand`.

---

**Note** Expansion options have an impact on the duration of a verification.

---

### Volatile Variables

#### Problem

A volatile variable can be defined as a variable which does not respect the "RAM axiom".



This axiom is:

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

### Explanation

As the value of a volatile variable is "unknown", it can take any value (that can be) represented by the type of the variable and can change even between 2 successive memory accesses.

A volatile variable is viewed as a "permanent random" by PolySpace because the value can change within its whole range between one read access and the next.

---

**Note** Even if the volatile characteristic of a variable is also commonly used by programmers to avoid compiler optimization, it has no consequence for PolySpace.

---

```
function test return integer is
  random: Integer;
  pragma volatile (random);
  y: Integer;  -- random ~ [-2^31, 2^31-1] ,
              -- although random is not initialized
begin
  y:= 1 /random; -- division and init orange
              -- because random
~ [-2^31, 2^31-1]
  random:= 100;
  y:= 1 /random; -- division and init orange
              -- because random~ [-2^31,2^31-1]
  return random; -- random ~ [-2^31, 2^31-1]
end;
```

### Shared Variables

#### Abstract

All of my shared variables appear in orange in the variable dictionary.

#### Explanation

When you launch PolySpace Server without any option all tasks are examined at the same level, making no assumptions about priorities, sequence order, or timing. In this context, shared variables will always be considered as unprotected.

#### Solution

You can use the following mechanisms to protect your variables.

- Critical section and mutual exclusion (explicit protection mechanisms);
- Access pattern (implicit protection);
- Rendezvous.

#### Critical Sections

These are the most common protection mechanism in applications and they are simple to use in PolySpace Server:

- if one task makes a call to a particular critical section, all other tasks will be blocked on the "critical-section-begin" function call until the originating task calls the "critical-section-end" function;
- this doesn't mean the code between two critical sections is atomic;
- It is a binary semaphore: you only have one token per label (in the example below CS1). Unlike many implementations of semaphores, it is not a decrementing counter that can keep track of a number of attempted accesses.

Also refer to "Atomicity" on page 5-26

**package my\_tasking.**

```
procedure proc1;
procedure proc2;
procedure my_main;
X: INTEGER;
Y: INTEGER;
end my_tasking;
```

**package body my\_tasking.**

```
with pkutil; use pkutil;
package body my_tasking is
  procedure proc1 is
  begin
    begin_cs;
    X = 12; -- X is protected
    Y = 100;
    end_cs;
  end;
  procedure proc2 is
  begin
    begin_cs;
    X = 11; -- X is protected
    end_cs;
    Y = 101; -- Y is not protected
  end;
  procedure my_main is
  begin
    X := 0;
    Y := 0;
  end
end my_tasking;
```

**package pkutil.**

```
procedure begin_cs;
procedure end_cs;
end pkutil;
```

### **package body pkutil.**

```
procedure Begin_CS is
begin
  null;
end Begin_CS;
procedure End_CS is
begin
  null;
end end_cs;
end pkutil;
```

### **Launching command.**

```
polyspace-ada \  
-automatic-stubbing \  
-main my_tasking.my_main \  
-entry-points my_tasking.proc1,my_tasking.proc2 \  
-critical-section-begin "pkutil.begin_cs:CS1" \  
-critical-section-end "pkutil.end_cs:CS1"
```

### **Mutual Exclusion**

Mutual exclusion between tasks or interrupts can be implemented while preparing PolySpace Server for launch setting.

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

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

These entry-points cannot overlap:

- t1 and t3
- t2, t3 and t4

These entry-points can overlap:

- t1 and t2
- t1 and t4

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

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

The myExclusions.txt is also required in the current directory. This will contain:

```
t1 t3
```

```
t2 t3 t4
```

## Rendezvous

All Ada rendezvous are taken into account without any input from the user. This is the only way to synchronize tasks. PolySpace Server does not handle atomicity, so other task synchronization mechanisms (including the use of critical sections) are not recognized by PolySpace Server.

package_first_task	other tasks
<pre>package first_task is   task task_1 is     entry INIT;     entry ORDER (X: out Integer);   end task_1; end first_task; package body first_task is   task body task_1 is   begin     accept INIT;     -- do things     accept ORDER (X: out Integer)     do       -- do things</pre>	<pre>with first_task; use first_task; package other_tasks is   task task_2 is   end task_2;   procedure main; end other_tasks; package body other_tasks is   task body task_2 is     X: INTEGER;   begin     task_1.init;     task_1.Order(X);   end task_2;   procedure main is</pre>

<b>package_first_task</b>	<b>other tasks</b>
<pre>-- call functions X:= 12; end; -- end accept -- return to main execution end task_1; end first_task;</pre>	<pre>begin;   null; end; end other_tasks;</pre>

The use of explicit tasks makes it unnecessary to use the `entry-points` option in your launching script.

```
polyspace-ada -main other_task.main
```

### **Semaphores**

Although it is possible to implement in ada, it is not possible to take into account a semaphore system call in PolySpace Server. Nevertheless, Critical sections may be used to model the behavior.

## Preparing Multitasking Code

### In this section...

“PolySpace Software Assumptions” on page 5-15

“Scheduling Model” on page 5-16

“Modelling Synchronous Tasks” on page 5-17

“Interruptions and Asynchronous Events/Tasks” on page 5-19

“Are Interruptions Maskable or Preemptive by Default?” on page 5-21

“Mailboxes” on page 5-22

“Atomicity” on page 5-26

“Priorities” on page 5-27

### 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 runtime 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 all start simultaneously, without any 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 assumes:

- no atomicity,
- no timing constraints.

### Scheduling Model

A problem can occur when some code is verified and the results suggest that all background tasks are dead code. In the same way, the problem could be the same (gray code) if several tasks (infinite loops) are defined and run concurrently in an RTOS.

In the PolySpace model, the main procedure is executed first before any other task is started. After it has finished, all task entry points are assumed to start concurrently, meaning they can interrupt each other at any time. 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 is always needed using `-main` option.
- **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 never interrupt each other. My tasks are not infinite loops - they always return control to the calling context.

```
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 - but there may be more warnings than necessary (that is, the results are less precise) because more scenarios than could actually happen at execution time are modelled.

In order to address this, PolySpace Server needs to be informed that the tasks are purely sequential - that is, that they are functions to be called in a deterministic order. This can be achieved by writing a function to call each

of the tasks in the correct sequence, and then declaring this new function as a single task entry point.

### Solution 1

Write a function that calls the cyclic tasks in the right order: 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

I have interrupt service routines which appear in gray (dead code) in the Viewer.

### Explanation

The gray code indicates that this code is not executed and is not taken into account, so all interruptions and tasks are ignored by PolySpace Server.

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, with all potential starting sequences being modeled.

### My interrupts it1 and it2 cannot preempt each other

If these 3 following conditions are fulfilled:

- the it1 and it2 functions can never interrupt each other;
- each interrupt can be raised several times, at any time;
- they are returning functions, and not infinite loops.

Then you can group non preemptive interruptions in a single function and declare that function as a task entry point.

```
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 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 any other procedures that are also declared as tasks via the -entry-point option. Therefore, as PolySpace Server makes an **upper approximation of all scheduling and all interleaving**, it **includes the possibility that the ISR might be interrupted by any other task**. There are more paths modelled than can happen during execution, but this has no adverse effect on the results obtained;

### 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;
  end;
```

```
    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 necessary 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 Launch PolySpace Viewer

```
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 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 appropriate to that data type.

### Solution

You can provide similar mechanisms with different levels of precision.

Mechanism	Description
Let PolySpace Server stub automatically	<ul style="list-style-type: none"> <li>• Quick and easy to code</li> <li>• <b>Imprecise</b> because there is no direct connection between a mailbox sender and receiver. It means that even if the sender is only submitting data within a small range, the full data range appropriate for the type(s) will be for the receiver data.</li> </ul>

<b>Mechanism</b>	<b>Description</b>
Provide a <b>real mailbox</b> mechanism	<ul style="list-style-type: none"> <li>• Can be very costly (time consuming) to implement</li> <li>• Can introduce errors in the stubs</li> <li>• Is too much effort compared with the solution below</li> <li>• Precise, but does not provide a much better precision than the upper approximation</li> </ul>
Provide an <b>upper approximation of the mailbox</b>	<p>in which each new read to the mailbox reads <b>one</b> of the recently posted messages, but not necessarily the last one.</p> <ul style="list-style-type: none"> <li>• Quick and easy to code</li> <li>• Gives precise results</li> <li>• See detailed implementation below</li> </ul>

**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 of the pieces are committed or none are.

## Instructional Decomposition

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

It is assumed by PolySpace that instructions are never atomic except in the case of read and write instructions. PolySpace Server makes an **upper approximation of all scheduling and all interleaving**. There are more paths modelled than could happen during execution, but given that **all possible paths are always verified**, this has no adverse effect on the results obtained.

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 any of 0xFF00, 0x0055 or 0xFF55.

PolySpace Server considers write/read instructions atomic, so **task 2 can only read 0xFF55**, even if X is not protected (refer to “Shared Variables” on page 5-10).

## Critical Sections

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

PolySpace Server's verification of Runtime Errors (RTEs) supposes that there was no conflict when writing the shared variables. Hence even if a shared variable is not protected, the RTE verification is complete and correct.

More information is available in "Critical Sections" on page 5-10.

## Priorities

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

In practice, while there is no facility to specify differing task priorities, all priorities **are** taken into account because of the default behavior of PolySpace Server assumes that:

- all task entry points (as defined with the option `-entry-points`) start potentially at the same time;
- they can interrupt each other in any order, no matter the sequence of instructions - and so all possible interruptions will be accounted for, in addition to some which can never occur in practice.

If you have two tasks `t1` and `t2` in which `t1` has higher priority than `t2`, simply use `polyspace-ada -entry-points t1,t2` in the usual way.

- `t1` will be able to interrupt `t2` at any stage of `t2`, which models the behavior at execution time;

- t2 will be able to interrupt t1 at any stage of t1, which models a behavior which (ignoring priority inversion) would never take place during execution. PolySpace Server has made an **upper approximation of all scheduling and all interleaving**. There are more paths modelled than could happen during execution, but this has no adverse effect on the results obtained.

# Running a Verification

---

- “Types of Verification” on page 6-2
- “Running Verifications on PolySpace Server” on page 6-3
- “Running Verifications on PolySpace Client” on page 6-21
- “Running Verifications from Command Line” on page 6-26

## Types of Verification

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

Use...	For...
Server	<ul style="list-style-type: none"><li>• Best performance</li><li>• Large files (more than 800 lines of code including comments)</li><li>• Multitasking</li></ul>
Client	<ul style="list-style-type: none"><li>• An alternative to the server when the server is busy</li><li>• Small files with no multitasking</li></ul> <hr/> <p><b>Note</b> Verification on a client takes more time. You might not be able to use your client computer when a verification is running on it.</p> <hr/>

## Running Verifications on PolySpace Server

### In this section...

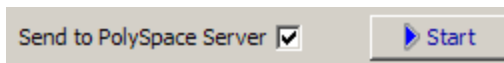
- “Starting Server Verification” on page 6-3
- “What Happens When You Run Verification” on page 6-4
- “Running Verification Unit-by-Unit” on page 6-5
- “Managing Verification Jobs Using the PolySpace Queue Manager” on page 6-6
- “Monitoring Progress of Server Verification” on page 6-8
- “Viewing Verification Log File on Server” on page 6-11
- “Stopping Server Verification Before It Completes” on page 6-13
- “Removing Verification Jobs from Server Before They Run” on page 6-14
- “Changing Order of Verification Jobs in Server Queue” on page 6-15
- “Purging Server Queue” on page 6-16
- “Changing Queue Manager Password” on page 6-17
- “Sharing Server Verifications Between Users” on page 6-18

### Starting Server Verification

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

To start a verification that runs on a server:

- 1 Open the Launcher.
- 2 Open the project containing the files you want to verify. For more information, see Chapter 3, “Setting Up a Verification Project”.
- 3 Select the **Send to PolySpace Server** check box next to the **Start** button in the middle of the Launcher window.



---

**Note** If you select **Set this option to use the server mode by default in every new project** in the Remote Launcher pane of the preferences, the **Send to PolySpace Server** check box is selected by default when you create a new project.

---

#### 4 Click **Start**.

The verification starts. For information on the verification process, see “What Happens When You Run Verification” on page 6-4.

---

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

---

#### 5 When you see the message *Verification process completed*, click **OK** to close the message dialog box.

#### 6 For information on downloading and viewing your results, see “Opening Verification Results” on page 8-8.

## What Happens When You Run Verification

The verification has three main phases:

- 1 Checking syntax and semantics (the compile phase). Because PolySpace software is independent of any particular Ada compiler, it ensures that your code is portable, maintainable, and complies with ANSI® standards.
- 2 Generating a main if it does not find a main and the **Generate a Main** option is selected. For more information about generating a main, see the section “Generate a main” in the “Options Description” chapter of the *PolySpace Products for Ada Reference*.
- 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 completes:



- A message dialog box tells you that the verification completed. This message means 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 log area tells you that the verification was transferred to the server and gives you the identification number (Analysis ID) for the verification. For the following verification, the identification number is 1.

The screenshot shows a log window with a sidebar on the left containing 'Compile', 'Stats', and 'Full Log'. The main area is a table with columns: Status, Description, File, Line, and Col. Two log entries are visible, both with an information icon (i) in the Status column.

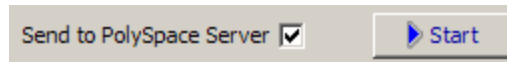
Status	Description	File	Line	Col
i	PolySpace Launcher for Ada95 verification start at Jan 15, 2009...			
i	The analysis has been queued with ID=2			

## Running Verification Unit-by-Unit

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

To run a unit-by-unit verification:

- 1 In the Launcher, ensure that the **Send to PolySpace Server** check box is selected.



- 2 In the Analysis options, expand **PolySpace inner settings**.
- 3 Select the **Run a verification unit by unit** check box.

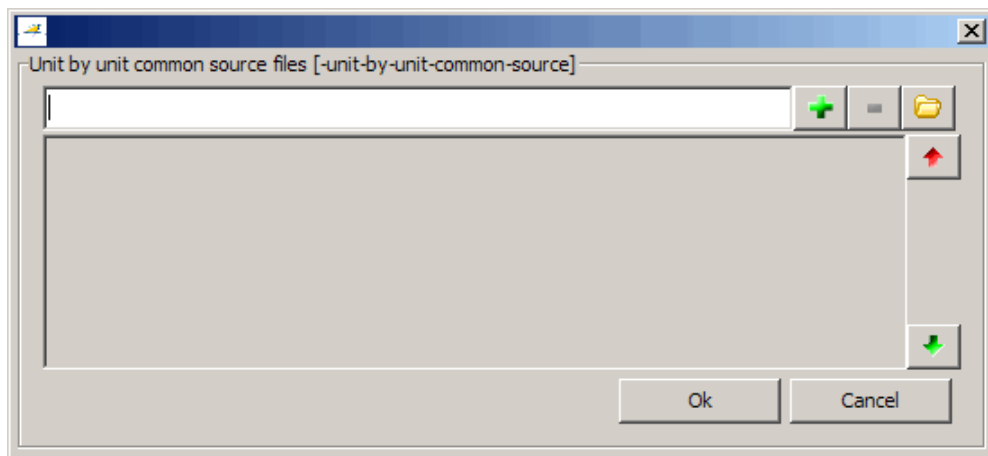
The screenshot shows a dialog box with a tree view under 'PolySpace inner settings'. The 'Run a verification unit by unit' option is checked. Below it, 'Unit by unit common source files' has an ellipsis button.

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

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

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

The Unit by unit common source dialog box opens.



- 6 Click the folder icon.

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

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

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

- 8 Click **Ok**.

- 9 Click **Start**.

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

## Managing Verification Jobs Using the PolySpace Queue Manager

You manage all server verifications using the PolySpace Queue Manager (also called the PolySpace Spooler). The PolySpace Queue Manager allows you to

move jobs within the queue, remove jobs, monitor the progress of individual verifications, and download results.

---

**Note** The PolySpace Queue Manager is not available on UNIX® or Linux® systems. To manage server verifications on UNIX or Linux systems, you must use batch commands. For information on managing verification jobs from the command line, see “Managing Verifications in Batch” on page 6-26.

---

To manage verification jobs on the PolySpace Server:

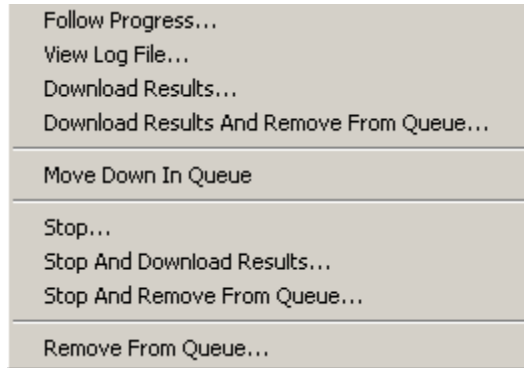
- 1 Double-click the **PolySpace Spooler** icon:



The **PolySpace Queue Manager Interface** opens.


PolySpace Queue Manager Interface							
Operations Help							
ID	Author	Application	Results directory	CPU	Status	Date	La
1	your_name	Example_Project	C:\polyspace_project\results	anse	running	'008,	

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



- 3 Select the appropriate option from the context menu.

---

**Tip** You can also open the Polyspace Queue Manager Interface by clicking the PolySpace Queue Manager icon  in the PolySpace Launcher toolbar.

---

### Monitoring Progress of Server Verification

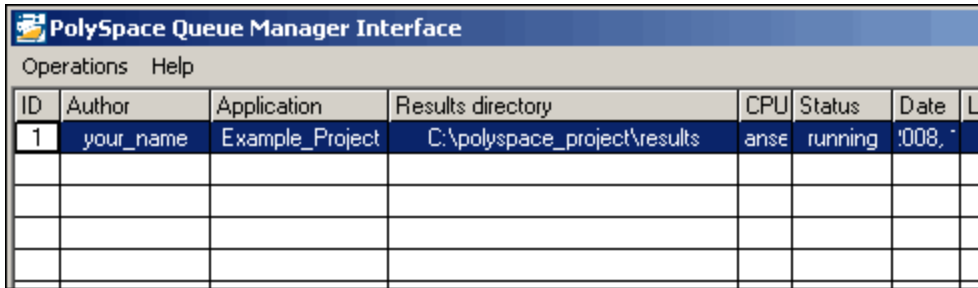
You can view the log file of a server verification using the PolySpace Queue Manager.

To view a log file on the server:

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



The **PolySpace Queue Manager Interface** opens.



The screenshot shows the PolySpace Queue Manager Interface. At the top, there is a title bar with the PolySpace logo and the text "PolySpace Queue Manager Interface". Below the title bar is a menu bar with "Operations" and "Help". The main area contains a table with the following columns: ID, Author, Application, Results directory, CPU, Status, Date, and Location. The first row of the table is highlighted in blue and contains the following data: ID: 1, Author: your\_name, Application: Example\_Project, Results directory: C:\polyspace\_project\results, CPU: anse, Status: running, Date: '008, and Location: .

ID	Author	Application	Results directory	CPU	Status	Date	Lo
1	your_name	Example_Project	C:\polyspace_project\results	anse	running	'008,	.

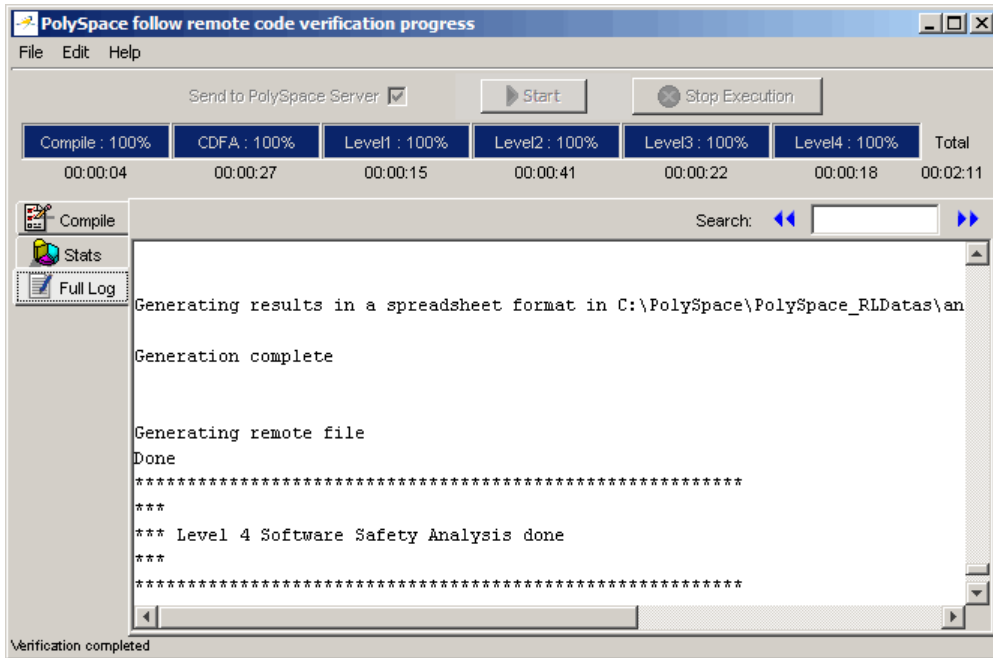
- 2 Right-click the job you want to monitor, and select **Follow Progress** from the context menu.

---

**Note** This option does not apply to unit-by-unit verification groups, only the individual jobs within a group.

---

A Launcher window labeled **PolySpace follow remote analysis progress for C** appears.




You can monitor the progress of the verification by watching the progress bar and viewing the logs at the bottom of the window. The word **processing** appears under the current phase. The progress bar highlights each completed phase and displays the amount of time for that phase.

The logs report additional information about the progress of the verification. The information appears in the log display area at the bottom of the window. The full log displays by default. It displays messages, errors, and statistics for all phases of the verification. You can search the full log by entering a search term in the **Search in the log** box and clicking the left arrows to search backward or the right arrows to search forward.

- 3 Click the **Compile Log** button to display compile phase messages and errors. You can search the log by entering search terms in the **Search in the log** box and clicking the left arrows to search backward or the right arrows to search forward. Click on any message in the log to get details about the message.

**4** Click the **Stats** button to display statistics, such as analysis options, stubbed functions, and the verification checks performed.

**5** Click the refresh button  to update the stats log display as the verification progresses.

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

When the verification completes, the status in the **PolySpace Queue Manager Interface** changes from running to completed.

PolySpace Queue Manager Interface							
Operations Help							
ID	Author	Application	Results directory	CPU	Status	Date	L
1	your_name	Example_Project	C:\polyspace_project\results	anse	completed	:008,	

## Viewing Verification Log File on Server

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

To view a log file on the server:

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



The **PolySpace Queue Manager Interface** opens.

PolySpace Queue Manager Interface							
Operations Help							
ID	Author	Application	Results directory	CPU	Status	Date	La
1	your_name	Example_Project	C:\polyspace_project\results	anse	running	'008,	

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

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

```

C:\PolySpace\PolySpace_Common\RemoteLauncher\wbin\psqueue-progress.exe
GUI files generation complete.
Generating remote file
Done

Certain (red) errors have been detected in the analysed code due
se.
Analysis continuing because the option -continue-with-red-error

*****
***
*** Level 4 Software Safety Analysis done
***
*****
Ending at: Apr 11, 2008 12:29:8
User time for pass4: 35.8real, 35.8u + 0s
User time for polyspace-c: 176.5real, 176.5u + 0s

***
*** End of PolySpace Verifier analysis
***
Press enter to close the window ...
  
```

3 Press **Enter** to close the window.



## Stopping Server Verification Before It Completes

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

To stop a server verification:

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



The **PolySpace Queue Manager Interface** opens.

PolySpace Queue Manager Interface							
Operations Help							
ID	Author	Application	Results directory	CPU	Status	Date	La
1	your_name	Example_Project	C:\polyspace_project\results	anse	running	'008,	

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

- **Stop** — Stops a unit-by-unit verification job without removing it. The status of the job changes from “running” to “aborted”. All jobs in the unit-by-unit verification group remain in the queue, and other jobs in the group will continue to run.
- **Stop and download results** — Stops the verification job immediately and downloads any preliminary results. The status of the verification changes from “running” to “aborted”. The verification remains in the queue.

- **Stop and remove from queue** — Stops the verification immediately and removes it from the queue. If the job is part of a unit-by-unit verification group, the entire verification is removed, not just the individual job.

## Removing Verification Jobs from Server Before They Run

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

---

**Note** If the job has started running, you must stop the verification before you can remove the job (see “Stopping Server Verification Before It Completes” on page 6-13). Once you have aborted a verification, you can remove it from the queue.

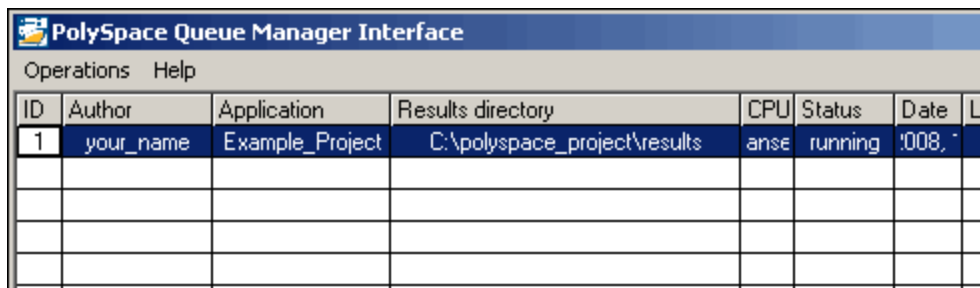
---

To remove a job from the server queue:

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



The **PolySpace Queue Manager Interface** opens.


 The screenshot shows a window titled "PolySpace Queue Manager Interface" with a menu bar containing "Operations" and "Help". Below the menu bar is a table with the following columns: ID, Author, Application, Results directory, CPU, Status, Date, and Location. The first row of the table is highlighted in blue and contains the following data: ID: 1, Author: your\_name, Application: Example\_Project, Results directory: C:\polyspace\_project\results, CPU: anse, Status: running, Date: '008, and Location: (partially visible).
 

ID	Author	Application	Results directory	CPU	Status	Date	Lo
1	your_name	Example_Project	C:\polyspace_project\results	anse	running	'008,	

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

The job is removed from the queue.

## Changing Order of Verification Jobs in Server Queue

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

To move a job within the server queue:

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



The **PolySpace Queue Manager Interface** opens.

PolySpace Queue Manager Interface							
Operations Help							
ID	Author	Application	Results directory	CPU	Status	Date	La
1	your_name	Example_Project	C:\polyspace_project\results	anse	running	'008,	

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

The job is moved down in the queue.

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

---

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

---

## Purging Server Queue

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

---

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

---

To purge the server queue:

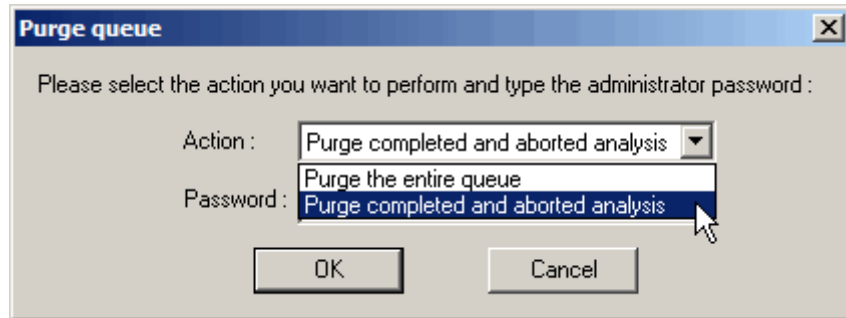
- 1 Double-click the **PolySpace Spooler** icon:



The **PolySpace Queue Manager Interface** opens.

PolySpace Queue Manager Interface							
Operations Help							
ID	Author	Application	Results directory	CPU	Status	Date	La
1	your_name	Example_Project	C:\polyspace_project\results	anse	running	'008,	

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



**3** Select one of the following options:

- **Purge completed and aborted analysis** — Removes all completed and aborted jobs from the server queue.
- **Purge the entire queue** — Removes all jobs from the server queue.

**4** Enter the Queue Manager **Password**.

**5** Click **OK**.

The server queue is purged.

## Changing Queue Manager Password

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

---

**Note** The default password is administrator.

---

To set the Queue Manager password:

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

The PolySpace Queue Manager Interface opens.

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

The Change Administrator Password dialog box opens.

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

The password is changed.

## Sharing Server Verifications Between Users

### Security of Jobs in Server Queue

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

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

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

### analysis-keys.txt File

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

- **UNIX** — `/home/<username>/PolySpace`
- **Windows®** — `C:\Documents and Settings\<username>\Application Data\PolySpace`

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

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

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

The fields in the file are tab-separated.

The file cannot contain blank lines.

**Example:**

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

**Sharing Verifications Between Accounts**

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

To share a verification with another user:

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

The second user can then download or manage the verification.

**Magic Key to Share Verifications**

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

The format for a magic key is as follows:

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

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

---

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

---

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

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

---

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

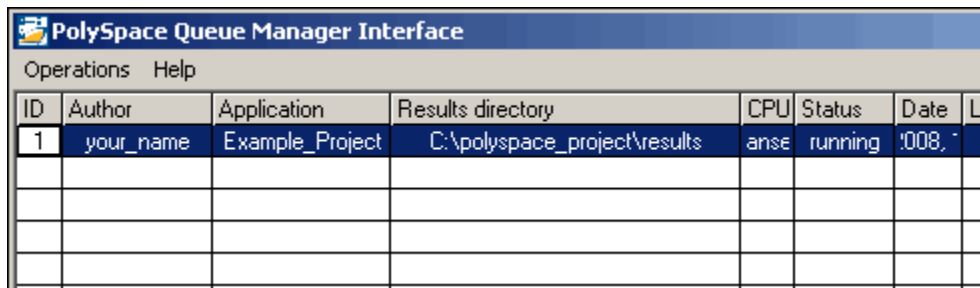
---

To use administrator mode:

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



The **PolySpace Queue Manager Interface** opens.

The screenshot shows the PolySpace Queue Manager Interface window. It has a title bar with the PolySpace logo and the text "PolySpace Queue Manager Interface". Below the title bar is a menu bar with "Operations" and "Help". The main area contains a table with the following data:

ID	Author	Application	Results directory	CPU	Status	Date	La
1	your_name	Example_Project	C:\polyspace_project\results	anse	running	'008,	

- 2 Select **Operations > Enter Administrator Mode**.
- 3 Enter the Queue Manager **Password**.
- 4 Click **OK**.

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



## Running Verifications on PolySpace Client

### In this section...

“Starting Verification on Client” on page 6-21

“What Happens When You Run Verification” on page 6-22

“Monitoring the Progress of the Verification” on page 6-23

“Stopping Client Verification Before It Completes” on page 6-24

### Starting Verification on Client

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

---

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

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

---

To start a verification that runs on a client:

- 1 Open the Launcher.
- 2 Open the project containing the files you want to verify. For more information, see Chapter 3, “Setting Up a Verification Project”.
- 3 Ensure that the **Send to PolySpace Server** check box is not selected.
- 4 If you see a warning that multitasking is not available when you run a verification on the client, click **OK** to continue and close the message box. This warning only appears when you clear the **Send to PolySpace Server** check box.
- 5 Click the **Start** button.



- 6 If you see a caution that PolySpace software will remove existing results from the results directory, click **Yes** to continue and close the message dialog box.

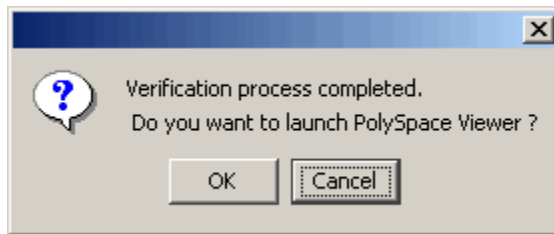
The progress bar and logs area of the Launcher window become active.

---

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

---

- 7 When the verification completes, a message dialog box appears telling you that the verification is complete and asking if you want to open the Viewer.



- 8 Click **OK** to open your results in the Viewer.

For information on viewing your results, see “Opening Verification Results” on page 8-8.

## What Happens When You Run Verification

The verification has three main phases:

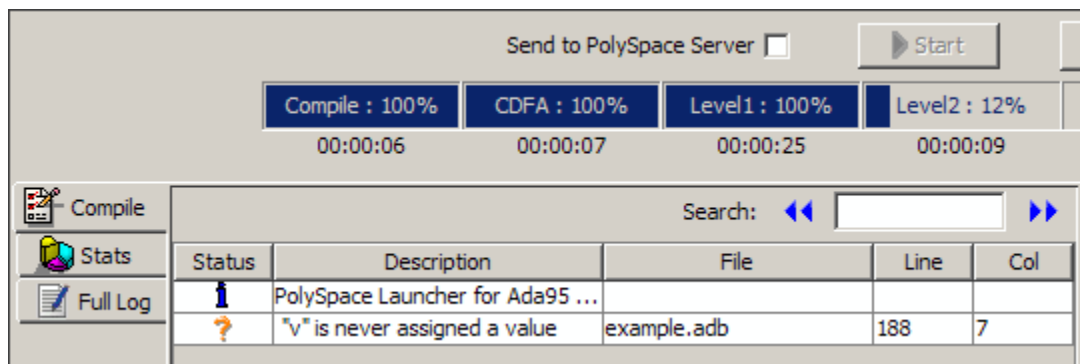
- 1 Checking syntax and semantics (the compile phase). Because PolySpace software is independent of any particular Ada compiler, it ensures that your code is portable, maintainable, and complies with ANSI standards.
- 2 Generating a main if it does not find a main and the **Generate a Main** option is selected. For more information about generating a main, see the

section “Generate a main” in the “Options Description” chapter of the *PolySpace Products for Ada Reference*.

- 3 Analyzing the code for run-time errors and generating color-coded diagnostics.

## Monitoring the Progress of the Verification

You can monitor the progress of the verification by watching the progress bar and viewing the logs at the bottom of the Launcher window.



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


The logs report additional information about the progress of the verification. To view a log, click the button for that log. The information appears in the log display area at the bottom of the Launcher window.

To view the logs:

- 1 The compile log is displayed by default.

This log displays compile phase messages and errors. You can search the log by entering search terms in the **Search in the log** box and clicking the left arrows to search backward or the right arrows to search forward. Click on any message in the log to get details about the message.

**2** Click the **Stats** button to display statistics, such as analysis options, stubbed functions, and the verification checks performed.

**3** Click the refresh button  to update the stats log display as the verification progresses.

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

You can search the full log by entering a search term in the **Search in the log** box and clicking the left arrows to search backward or the right arrows to search forward.

---

**Note** Closing the Launcher window does *not* stop the verification. To resume display of the verification progress, open the Launcher window and open the project that you were verifying when you closed the Launcher window.

---

### Stopping Client Verification Before It Completes

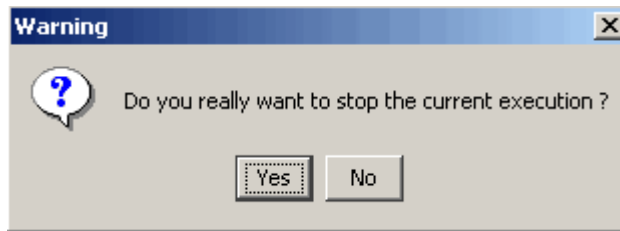
You can stop the verification before it completes. If you stop the verification, results will be incomplete, and if you start another verification, the verification starts over from the beginning.

To stop a verification:

**1** Click the **Stop Execution** button.



A warning dialog box appears.



**2** Click **Yes**.

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

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

---

**Note** Closing the Launcher window does *not* stop the verification. To resume display of the verification progress, open the Launcher window and open the project that you were verifying when you closed the Launcher window.

---

## Running Verifications from Command Line

In this section...
“Launching Verifications in Batch” on page 6-26
“Managing Verifications in Batch” on page 6-26

### Launching Verifications in Batch

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

All these commands begin with the following prefixes:

- **Server verification** —  
`<PolySpaceInstallDir>/Verifier/bin/polyspace-remote-ada95`
- **Client verification** —`polyspace-remote-desktop-ada95`

These commands are equivalent to commands with a prefix  
`<PolySpaceInstallDir>/bin/polyspace-`.

For example, `polyspace-remote-desktop-ada95 -server  
[<hostname>:[<port>] | auto]` allows you to send a Ada desktop  
verification remotely.

---

**Note** If your PolySpace server is running on Windows, the batch  
commands are located in the `/wbin/` directory. For example,  
`<PolySpaceInstallDir>/Verifier/wbin/polyspace-remote-ada95.exe`

---

### Managing Verifications in Batch

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

On UNIX platforms, all these command begin with the prefix  
`<PolySpaceCommonDir>/RemoteLauncher/bin/psqueue-`.

On Windows platforms, these commands begin with the prefix `<PolySpaceCommonDir>/RemoteLauncher/wbin/psqueue-:`

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

- `psqueue-set-password <password> <new password>` — change administrator password.
- `psqueue-check-config` — check the configuration of Queue Manager.
  - `[-check-licenses]` check for licenses only.
- `psqueue-upgrade` — Allow to upgrade a client side (see the PolySpace Installation Guide in the `<PolySpace Common Dir>/Docs` directory).
  - `[-list-versions]` give the list of available release to upgrade.
  - `[-install-version <version number> [-install-dir <directory>]] [-silent]` allow to install an upgrade in a given directory and in silent.

---

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

---



# Troubleshooting Verification Problems

---

- “Verification Process Failed Errors” on page 7-2
- “Compile Errors” on page 7-6
- “Reducing Verification Time” on page 7-9
- “Obtaining Configuration Information” on page 7-20
- “Removing Preliminary Results Files” on page 7-22

## Verification Process Failed Errors

In this section...
“Overview” on page 7-2
“Hardware Does Not Meet Requirements” on page 7-2
“You Did Not Specify the Location of Included Files” on page 7-2
“PolySpace Software Cannot Find the Server” on page 7-3
“Limit on Assignments and Function Calls” on page 7-4

### Overview

If you see a message that saying `Verification process failed`, it indicates that PolySpace software could not perform the verification. The following sections present some possible reasons for a failed verification.

### Hardware Does Not Meet Requirements

The verification fails if your computer does not have the minimal hardware requirements. For information about the hardware requirements, see

[www.mathworks.com/products/polyspaceclientada/requirements.html](http://www.mathworks.com/products/polyspaceclientada/requirements.html).

To determine if this is the cause of the failed verification, search the log for the message:

Errors found when verifying host configuration.

You can:

- Upgrade your computer to meet the minimal requirements.
- Select the **Continue with current configuration option** in the General section of the Analysis options and run the verification again.

### You Did Not Specify the Location of Included Files

If you see a message in the log, such as the following, either the files are missing or you did not specify the location of included files.

Verifier found an error in example.adb:23:14: "runtime\_error (spec)" depends on "types (spec)"

For information on how to specify the location of include files, see “Creating New Projects” on page 3-8.

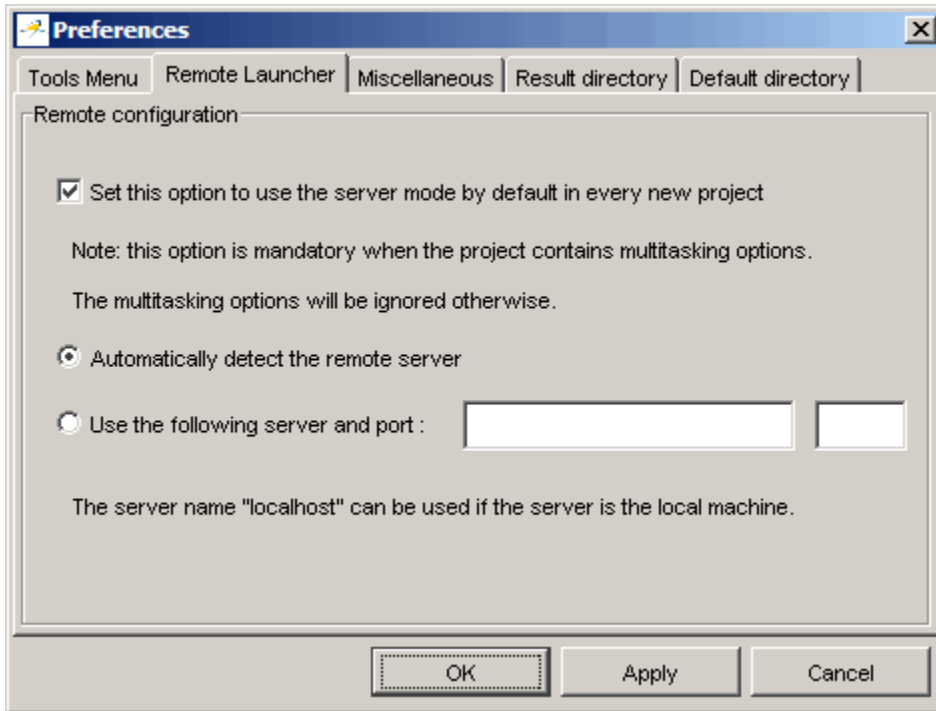
## **PolySpace Software Cannot Find the Server**

If you see the following message in the log, PolySpace software cannot find the server.

Error: Unknown host :

PolySpace software uses information in the preferences to locate the server. To find the server information in the preferences:

- 1** Select **Edit > Preferences**.
- 2** Select the **Remote Launcher** tab.



By default, PolySpace software automatically finds the server. You can specify the server by selecting **Use the following server and port** and providing the server name and port. For information about setting up a server, see the *PolySpace Installation Guide*.

## Limit on Assignments and Function Calls

If you launch a client verification on a large file, the verification may stop and you may receive an error message saying the number of assignments and function calls is too big. For example:

```
*****
Beginning C to intermediate language translation
*****
C to intermediate language translation 1 (P_SP)
...
```

```
*** License error: number of assignments and function calls is too  
big for -unit mode (5534 v.s 2000).  
*** Aborting.
```

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

To verify code containing more than 2,000 assignments and calls, launch your verification on the PolySpace Server for Ada.

## Compile Errors

In this section...
“Overview” on page 7-6
“Examining the Compile Log” on page 7-6
“Unit Verification” on page 7-8

### Overview

PolySpace software may be used instead of your chosen compiler to make syntactical, semantic and other static checks. These errors will be detected during the standard compliance checking stage, which takes about the same amount of time to run as a compiler. The use of PolySpace software this early in development yields a number of benefits:

- detection of link errors, plus errors which are only apparent with reference to two or more files;
- objective, automatic and early control of development work (perhaps to avoid errors prior to checking code into a configuration management system).

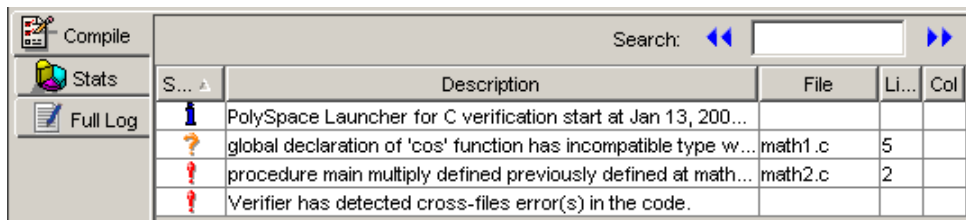
### Examining the Compile Log

The compile log displays compile phase messages and errors. You can search the log by entering search terms in the **Search in the log** box and clicking the left arrows to search backward or the right arrows to search forward.

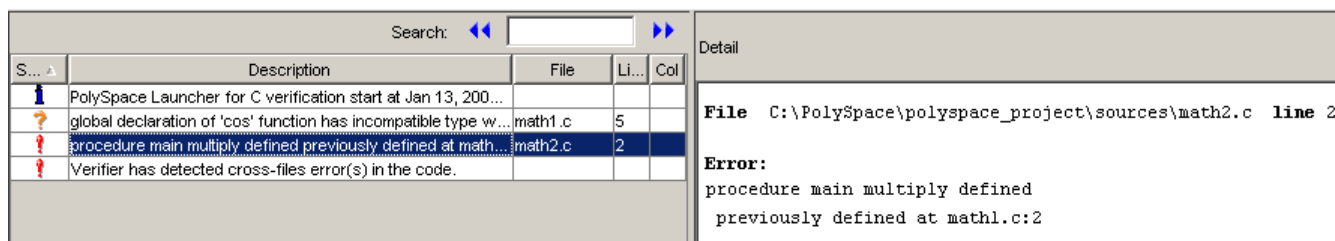
To examine errors in the Compile log:

- 1 Click the **Compile** button in the log area of the Launcher window.

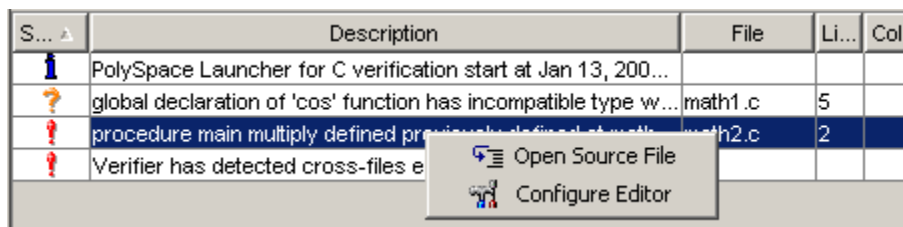
A list of compile phase messages appear in the log part of the window.



- 2 Click on any of the messages to see message details, as well as the full path of the file containing the error.



- 3 To open the source file referenced by any message, right click the row for the message, then select Open Source File.



The file opens in your text editor.

**Note** You must configure a text editor before you can open source files. See “Configuring Text and XML Editors” on page 3-17.

- 4 Correct the error and run the verification again.

### Unit Verification

PolySpace requires the complete specifications associated with a package body verification. Sometimes you might face this kind of obvious error message:

```
Verifying _pst_main  
  
Verifying my_package  
  
-> Verifier found an error  
in ./My_Package.adb:2:14:  
Missing specification for unit "My_Package"
```

PolySpace reports this kind of error when a package body is supplied as the source and the specification is supplied as one of the specifications in one of the `-ada-include-dir` directories.

Specifications of the package body needs to be included in the list of supplied sources.



## Reducing Verification Time

In this section...
“PolySpace Verification Duration” on page 7-9
“An Ideal Application Size” on page 7-9
“Why Should there be an Optimum Size?” on page 7-10
“Selecting a Subset of Code” on page 7-11
“What are the Benefits of these Methods?” on page 7-17

### PolySpace Verification Duration

The duration of a verification is impacted by:

- The size of the code
- The number of global variables
- The nesting depth of the variables (the more nested they are, the longer it takes)
- The depth of the call tree of the application
- The “intrinsic complexity” of the code, particularly with regards to arithmetic manipulation.

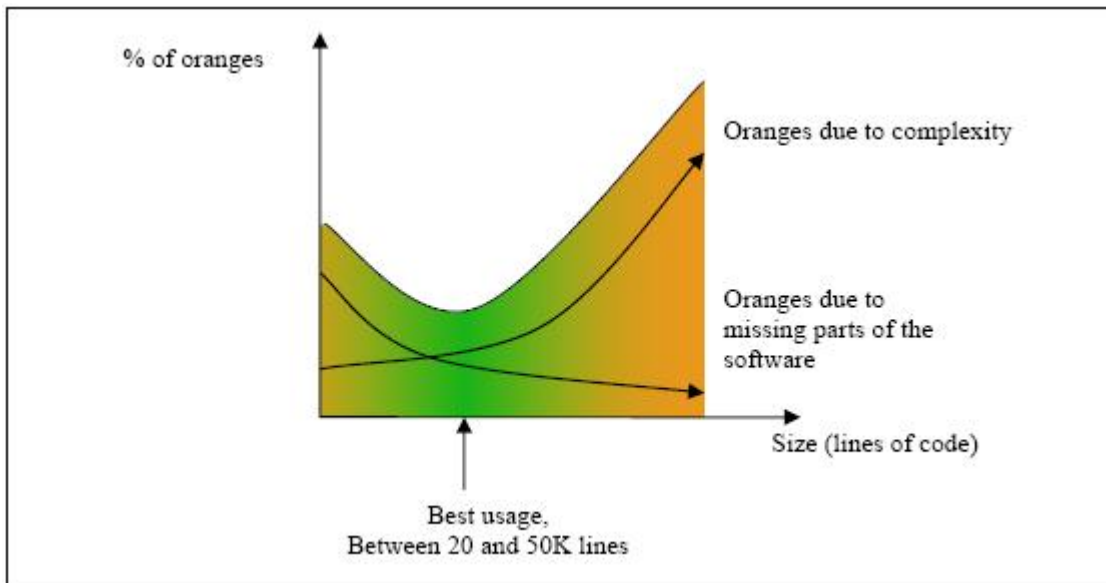
The fact that so many factors are involved makes it impossible to derive a precise formula to calculate verification duration. Following sub section try to give some hints to reduce time of a verification.

### An Ideal Application Size

There always 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 it can manage and to solve traditionally incomputable problems. However, they also mean that the benefits derived from verifying the whole of a large application have to be balanced against the loss of precision which results.

**This is why we recommend that you begin with package by package verifications.** The **maximum** recommended application size fifty thousand lines of code. For such applications, approximations should not be too significant. Take care that sometimes the duration of a verification may **not be reasonable**.

Experience suggests that subdividing an application prior to verification will typically have 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**

### **Why Should there be an Optimum Size?**

PolySpace has been used to verify numerous applications with greater than one hundred thousand lines of code. However, as project sizes become very large PolySpace Server

- makes broader approximations, producing more oranges
- can take much more time to verify the application.

PolySpace is most effective when it is used **as early as possible** in the development process, i.e. **BEFORE** any other form of testing.

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

During the integration process, there might be a point where the code becomes so large (maybe 50000 lines of code or more) that the verification of the whole project is not achievable within a reasonable amount of time. Then there are two options.

- Stop the use of PolySpace at this stage (many of the benefits have been achieved already.)
- Verify subsets of the code.

## Selecting a Subset of Code

If a project is subdivided into logical sections by considering data flow, the total verification time will be considerably shorter than for the project considered in one pass. (See also: “Volatile Variables” on page 5-8 , “Automatic Stubbing” on page 5-5)

In such an application, there are two distinct concepts to consider:

- function entry-points — Function entry-points refer to the PolySpace execution model since they are started concurrently, without any assumption regarding sequence or priority. They represent the beginning of your call tree;
- data entry-points — Regard lines in the code where data is acquired as "data entry points".

Consider the examples below.

### Example 1

```
Procedure complete_treatment_based_on_x(input : integer) is
begin
```

```
    thousand of line of computation...
end
```

### Example 2

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

### 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 case, the "x" variable is a data entry point and "y" is the consequence of such an entry point. "y" may be formatted data, due to a very complex manipulation of x.

Since x is volatile, a probable consequence will be that y will contain all possible formatted data. An approximation could be to completely remove the procedure `complete_treatment_based_on_x` and let automatic stubbing work: it will then assign a full range 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
```

## Some Consequences

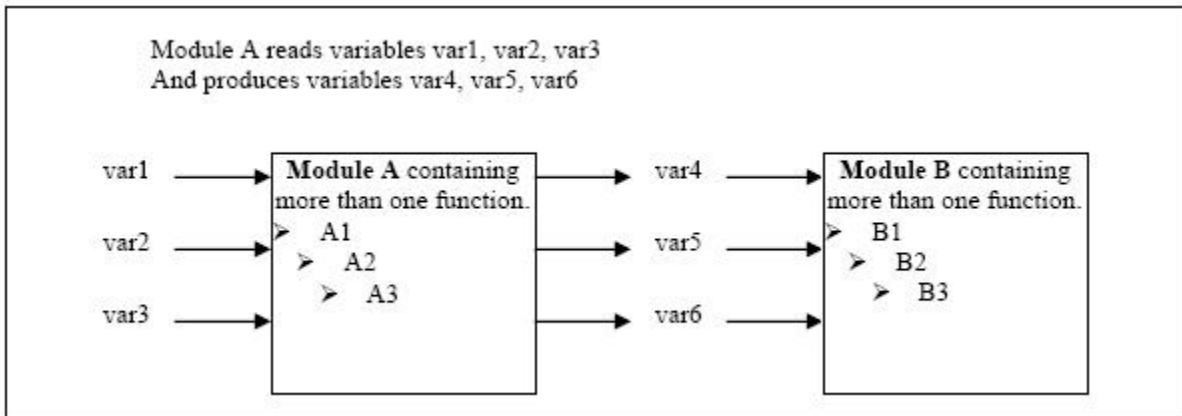
- (-) A slight loss of precision on  $y$ . PolySpace will now consider all possible values for  $y$ , including the formatted ones that were present at the first verification.
- (+) A huge investigation of the code is not necessary to isolate a meaningful subset. Any application can be split logically in this way.
- (+) No functional modules are lost.
- (+) The results will still be correct because there is no need to remove any thread affecting (change) shared data.
- (+) The complexity of the code is considerably reduced.
- (+) A high precision level (say O2) can be maintained.

## Typical Examples of Removable Components, According to the Logic of the Data

- **Error management modules.** These modules often contain a big 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 will be assumed to return a value in the range  $[-2^{31}, 2^{31}-1]$ , which includes 1 and 0. The procedure will be considered to return all possible answers, just like reality;
- **Buffer management for mailboxes coming from missing code.** Suppose an application reads a huge buffer of 1024 char, and then uses it to populate 3 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, then the verification of the remaining code will just be the same.

## Subdivide According to Data-Flow

Consider the following example.



In this application, variables 1, 2 and 3 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 which interpolates between var1 and var2. That algorithm uses var3 as an exponential factor, so when var1 is equal to 0, the result in var4 is also equal to 0.

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

Ranges	var4 var5 var6	Between -60 and 110 Between 0 and 12 Between 0 and 100
Properties	And a set of properties between variables	<ul style="list-style-type: none"> <li>• If var2 is equal to 0, than <math>\text{var4} &gt; \text{var5} &gt; 5</math>.</li> <li>• If var3 is greater than 4, than <math>\text{var4} &lt; \text{var5} &lt; 12</math></li> <li>• ...</li> </ul>

Subdivision in accordance with data flow allows modules A and B to be verified separately.

- A will use variables 1, 2 and 3 initialized respectively to [0;10], [1;100] and [10;10]
- B will use variables 4, 5 and 6 initialized respectively to [-60;110], [0;12] and [10;10]

#### The consequences:

- (-) A slight loss of precision on the B module verification, because now all combinations for variables 4, 5 and 6 are considered:
  - It includes all of the possible combinations.
  - It also includes those that would have been restricted by the A module verification.
- For instance. If the B module included the test
- “If var2 is equal to 0, than  $\text{var4} > \text{var5} > 5$ ”
- then the dead code on any subsequent “*else*” clause would not be detected.
- (+) An in depth investigation of the code is not necessary to isolate a meaningful subset. It means that a logical split is possible for any application, in accordance with the logic of the data
- (+) The results remain valid (because there no need to remove (say) a thread that will change shared data)

- (+) The complexity of the code is reduced by a significant factor
- (+) The maximum precision level can be retained.

### **Typical 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 which are accessed through an API. The removal of the API code with the retention of the prototype will result in the PolySpace verification producing a stub which returns  $[-2^{31}, 2^{31}-1]$ . This clearly includes 1 and 0 (yes and no). The procedure `has_an_error_already_occurred` will therefore return all possible answers, just like the code would 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 3 small arrays of data using a very complicated algorithm. This data is then given to a main module for treatment. For the PolySpace Server verification, the buffer can be removed and the 3 arrays initialized with random values.
- Display modules.

### **Subdivide According to Real-Time Characteristics**

Another way of splitting an application is to isolate files which contain only a subset of tasks, and to verify each subset separately.

If a verification is initiated using only a few tasks, PolySpace Server will lose information regarding the interaction between variables.

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

If T1 modifies x and T2 is scheduled to read it at a particular moment, subsequent operations in T2 will be 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 could be declared as volatile in order to take into account all possible executions. Otherwise x will take only its initial value or x variable



will remain constant, and T2's verification will be a subset of possible execution paths. You might have precise results, but it will only include one *scenario* among all possible states for the variable *x*.

- *x* could be initialized to the whole possible range [10;15], and then the T2 entry-point called. This is accurate if *x* is calibration data.

### **Subdivide According to Files**

Simply extract a subset of files and perform a verification either:

- using entry-points, or
- by creating a “*main*” that calls randomly all functions that are not called by any other within this subset of code.

This method may look too simple to be efficient but it can produce good results when the aim is to find red errors and bugs in gray code.

### **What are the Benefits of these Methods?**

It may be desirable to split the code

- **To reduce the verification time for a particular precision mode**
- **To reduce the number of oranges** (see next two sections for details)

The problems subdivision may bring are that

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

### **When the Application is Incomplete**

When the code consists of a small subset of a larger project, a lot of procedures will be automatically stubbed. This is done according to the specification or prototype of the missing functions, and 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,  $a*b$  would cause an overflow, because “a” and “b” can be equal to  $2^{31}$ . The number of incidences of these “data set issue” orange check can be reduced by precise stubbing.

Now consider a procedure  $f$  which modifies its input parameters “a” and “b”, both of which are passed by reference. Suppose that “a” might be modified to any value between 0 and 10, and “b” to any 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 can introduce orange checks in a code snippet such as  $1/(a*b - 100)$ , where the division would be orange.

- So - even where precise stubbing is used, verifying a small piece of application might introduce extra orange checks. However, the net effect from reducing the complexity will be to reduce the total number of orange checks.
- When using the default stubbing, the increase in the number of orange checks as the result of this phenomenon tends to be more pronounced.

### Considering the Effects of Application Code Size

PolySpace Server can make approximations when computing the possible values of the variables, at any point in the program. Such an approximation will always use a superset of the actual possible values.

For instance, in a relatively small application, PolySpace Server might retain very detailed information about the data at a particular point in the code, so that for example the variable VAR can take the values  $\{-2 ; 1 ; 2 ; 10 ; 15 ; 16 ; 17 ; 25\}$ . If VAR is used to divide, the division is green (because 0 is not a possible value).

If the program being verified is large, PolySpace Server would simplify the internal data representation by using a less precise approximation, such as  $[-2 ; 2] \cup \{10\} \cup [15 ; 17] \cup \{25\}$ . Here, the same division appears as an orange check.

If the complexity of the internal data becomes even greater later in the verification, PolySpace Server might further simplify the VAR range to (say)  $[-2 ; 20]$ .

This phenomenon leads to the increase or the number of orange warnings when the size of the program becomes large.

---

**Note** The amount of simplification applied to the data representations also depends on the required precision level (O0, O2), PolySpace Server will adjust the level of simplification, for example:

- -O0 — shorter computation time,
  - -O2 — fewer orange warnings.
  - -O3 — fewer orange warnings and longer computation time.
-

## Obtaining Configuration Information

The `polyspace-ver` command allows you to quickly gather information on your system configuration. You should use this information when entering support requests.

Configuration information includes:

- Hardware configuration
- Operating System
- PolySpace Licenses
- Specific version numbers for PolySpace products

To obtain your configuration information, enter the following command:

- **UNIX/Linux** — `<PolySpaceInstallDir>/Verifier/bin/polyspace-ver`
- **Windows** — `<PolySpaceInstallDir>/Verifier/wbin/polyspace-ver.exe`

The configuration information appears.

```

C:\WINNT\system32\cmd.exe
C:\PolySpace\PolySpaceForCandCPP_R2009b\Verifier\wbin>polyspace-ver.exe
-----
Machine Hardware Configuration:
* Number of CPUs      : 1
* CPU frequency      : 2.211GHz
* CPU type            : i686
* Memory              : 1023MB
* Swap                : 2.40GB
* /tmp free space    : 169.44GB
-----

Machine Software Configuration:
Windows XP (Service Pack 3)
-----

PolySpace Licenses:
PolySpace_Client_C_CPP:
  License Number: DEMO
  Expiration date: 20-oct-2009

PolySpace_Server_C_CPP:
  License Number: DEMO
  Expiration date: 20-oct-2009

PolySpace_Model_Link_SL:
  License Number: DEMO
  Expiration date: 20-oct-2009
-----

PolySpace Versions:
PolySpace Version R2009b
* Kernel                CC-7.1.0.U1
* Viewer                IHME-R2009b-U9
* Launcher              IHML-R2009b-U9
* Remote Launcher      RL-R2009b-U6
* Visual Plugin        PUP6_0_1_5
* PolySpace In One Click POC-R2009b-U4
* MBD Plugin           MBD-R2009b-U4
* Automatic Orange Tester AOT-R2009b-U4

Remote Launcher configuration
* Compatibility version 3_12_2

Server :
PolySpace_Server_C_CPP.mathworks.com
-----

C:\PolySpace\PolySpaceForCandCPP_R2009b\Verifier\wbin>

```

---

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

---

## Removing Preliminary Results Files

By default, the software automatically deletes preliminary results files when they are no longer needed by the verification. However, if you run a client verification using the option `keep-all-files`, preliminary results files are retained in the results directory. This allows you to restart the verification from any stage, but can leave unnecessary files in your results directory.

If you later decide that you no longer need these files, you can remove them.

To remove preliminary results files:

- 1** Open the project containing the results you want to delete In the Launcher.
- 2** Select **Tools > Clean Results**.

The preliminary results files are deleted.

---

**Note** To remove **all** verification results from your results directory (including the final results), select **Tools > Delete Results**.

---

# Reviewing Verification Results

---

- “Before You Review PolySpace Results” on page 8-2
- “Opening Verification Results” on page 8-8
- “Reviewing Results in Assistant Mode” on page 8-21
- “Reviewing Results in Expert Mode” on page 8-29
- “Importing and Exporting Review Comments” on page 8-42
- “Generating Reports of Verification Results” on page 8-45
- “Using PolySpace Results” on page 8-52

## Before You Review PolySpace Results

### In this section...

“Overview: Understanding PolySpace Results” on page 8-2

“Why Gray Follows Red and Green Follows Orange” on page 8-3

“What is the Message and What does it Mean?” on page 8-4

“What is the Ada Explanation?” on page 8-5

### Overview: Understanding PolySpace Results

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

- **Red** – Indicates code that always has an error (errors occur every time the code is executed).
- **Gray** – Indicates unreachable code (dead code).
- **Orange** – Indicates unproven code (code might have a runtime error).
- **Green** – Indicates code that never has a runtime error (safe code).

This section explains how to analyze these colors. There are four rules to remember:

- An instruction is verified only if no runtime error was detected in the previous instruction.
- The verification assumes that each runtime error causes a “core dump.” The corresponding instruction is considered to have stopped, even if the actual run time execution of the code might not stop. This means that red checks are always followed by gray checks, and orange checks only propagate the green parts through to subsequent checks.
- Always focus on the message given 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 problem.
- Always determine an explanation by examining the actual code. Do not focus on what the code is supposed to do.



## Why Gray Follows Red and Green Follows Orange

This section explains why **gray** checks follow **red** checks, and how **green** checks are propagated out of **orange** ones.

In the example below, consider why:

- the **gray** checks follow the **red** in the red function.
- there are **green** checks relating to the array.

```

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

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; -- [array index within bounds]
  Y(X):= 0;
end main;

```

Let's detail each line of code for:

The red function:

- When PolySpace divides by X, X has not been initialized. Therefore the corresponding check (Non Initialized Variable) on X is red;
- As a result all possible execution paths are stopped, because they all produce an RTE.

The propagate function:

- X is assigned the value of Read\_An\_Input. After this assignment,  $X \sim [-2^{31}, 2^{31}-1]$ ;
- At the first array access, an “out of bounds” error is possible since X can be equal to (say) -3 as well as 3;
- All conditions leading to an RTE are assumed to have been truncated - they are no longer considered in the verification. So on the following line, the executions for which  $X \sim [-2^{31}, -1]$  and  $[100, 2^{31}-1]$  are stopped;

- Consequently at the next instructions  $X \sim [0, 99]$ ;
- Hence at the second array access, the check is green because  $X \sim [0, 99]$ .

### Summary

Green checks can be propagated out of orange checks.

---

**Note** When doing 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 4-8.

---

### What is the Message and What does it Mean?

PolySpace numbers the results in the same order in which an execution would have performed the associated operations.

Consider the instruction: `x := x + 1;`

In each case, PolySpace first checks for a potential NIV (Non Initialized Variable) for `x`, then checks the potential OVFL (overflow). An awareness of such sequences will help to understand the message which PolySpace is presenting before going on to assess what that means for the code.

In the example below, the orange NIV on `X` in the test:

```
if ( x > 101)
```

does not mean PolySpace does not know the value of `X`, which might be the conclusion of a hasty analysis.

So - what does it mean?

```
function Read_An_Input return integer;  
procedure Main is  
  X: Integer;  
begin  
  if (Read_An_input) then
```

```

    X := 100;
  end if;
  if (X > 101) then -- [orange on NIV : non initialised variable ]
    X := X + 1; -- gray code
  end if;
end Main;

```

## Explanation

When you click on the check under the Viewer, you see the category of the check. Here, the category is NIV (Non Initialized Variable). However, PolySpace may well verify subsequent lines of code, and continue with an understanding of the possible values as if initialization has taken place.

The correct analysis of this result might be that if X has been initialized, the only possible value for X is { 100 }, which is not greater than 101, so the rest of the code is gray. Hence we can conclude that PolySpace did know the values.

## Summary

- **FALSE:** if "( x > 101 )" means: PolySpace does not know anything.
- **TRUE:** if "( x > 101 )" means: PolySpace does not know if X has been initialized.

The first rules of reviewing results are: focus on the message given by PolySpace and do not focus on a speedy interpretation.

## What is the Ada Explanation?

Try to explain results based on the code and not on:

- A physical action,
- A particular configuration, data calibration,
- Or any other reason than the code itself.

Concentrate on the source code only - remember, PolySpace knows nothing of the environment in which the code will be executed.

In the example below what is the explanation of the dead code (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;

```

This is a method you can use to understand any color:

**1 First Step:** The line containing the access to the Y array is unreachable (this line is unreachable)

- So - the test to assess whether x is equal to 0 is always false
- Now, it would be easy to jump to the conclusion that this results from input data which is always different from 0. However, `Read_An_Input` can be any value in the full integer range, so this is not the right explanation.
- X has been assigned to its full range, but the test assumes that X is never equal to 0 at this line. Why?

**2 Second Step:** "Why is the test always false?"

- After the variable definitions, it can be seen that the first array access is orange: before this line  $X \in [-2^{31}, 2^{31} - 1]$  because of the `Read_An_Input` function, and afterwards,  $X \in [0, 99]$  (see Examples "*Example D*" and "*Example E*")
- So  $X \in [0, 99]$  just after the first array access.

- The next operation to be checked by PolySpace Server is the addition “+ X” which is **green**
- The next operation checked after that will be the division by X which is **orange** because  $X \in [0,99]$ . So after the division,  $X \in [1,99]$ . The orange will truncate all execution paths that lead to a runtime error, so that in our example, all instances where X is equal to 0 are stopped.

**3 Third Step:** The second array index is green and therefore explains why the test is always false.

When the assignment sign is reached,  $X \in [1,99]$  and hence the array access is green.

**4 Conclusion:** The user has found a **bug!** The dead code has shown that the test should be performed before the division.

---

**Note** You must explain a color step by step, until you find the root cause, and focus on explanations within the code only. Try to exclude the knowledge about what the code actually does in its execution environment.

---

---

**Note** In this example, all results are located in the same procedure. The same approach is valid if a check is to be verified involving a procedure called by others. Use the "called by" call tree to help in the analysis of the results.

---

## Opening Verification Results

### In this section...

“Downloading Results from Server to Client” on page 8-8

“Downloading Server Results to UNIX or Linux Clients” on page 8-11

“Downloading Results from Unit-by-Unit Verifications” on page 8-12

“Opening Verification Results” on page 8-12

“Exploring the Viewer Window” on page 8-13

“Selecting Viewer Mode” on page 8-17

“Setting Character Encoding Preferences” on page 8-17

## Downloading Results from Server to Client

When you run a verification on a PolySpace server, the results are stored on the server. Before you can view your results, you must download the results file from the server to the client.

---

**Note** If you download results before the verification completes, you get partial results and the verification continues.

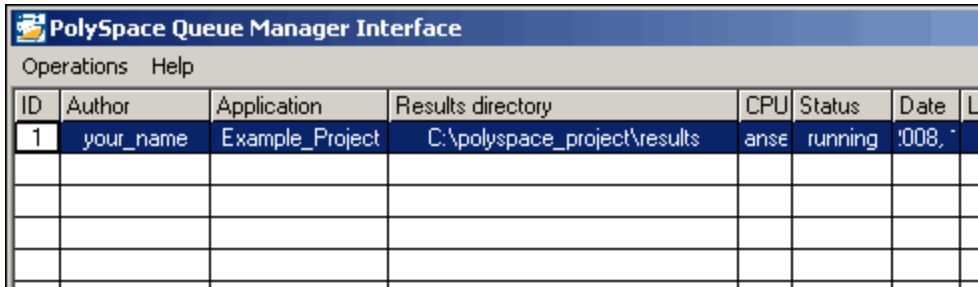
---

To download verification results to your client system:

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



The **PolySpace Queue Manager Interface** opens.



The screenshot shows the PolySpace Queue Manager Interface. At the top, there is a title bar with the PolySpace logo and the text "PolySpace Queue Manager Interface". Below the title bar is a menu bar with "Operations" and "Help". The main area contains a table with the following columns: ID, Author, Application, Results directory, CPU, Status, Date, and Location. The first row of the table is highlighted in blue and contains the following data: ID: 1, Author: your\_name, Application: Example\_Project, Results directory: C:\polyspace\_project\results, CPU: anse, Status: running, Date: '008, and Location: .

ID	Author	Application	Results directory	CPU	Status	Date	Location
1	your_name	Example_Project	C:\polyspace_project\results	anse	running	'008,	.

---

**Note** The PolySpace Queue Manager is not available on UNIX or Linux systems. If you are using the PolySpace Client for C/C++ on a UNIX or Linux system, you must use the `psqueue-download` command to download your results. For information, see “Downloading Server Results to UNIX or Linux Clients” on page 8-11.

---

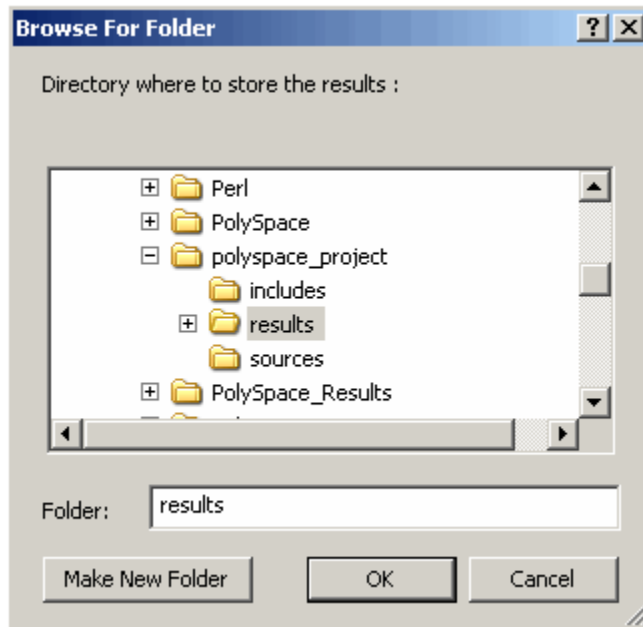
- 2 Right-click the job you want to view, then select **Download Results** from the context menu.

---

**Note** To remove the job from the queue after downloading your results, select **Download Results And Remove From Queue** from the context menu.

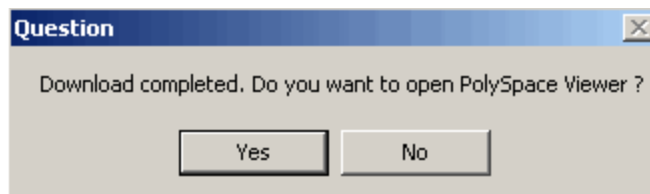
---

The **Browse For Folder** dialog box appears.



- 3 Select the folder into which you want to download results.
- 4 Click **OK** to download the results and close the dialog box.

When the download completes, a dialog box appears asking if you want to open the PolySpace Viewer.



- 5 Click **Yes** to open the results.

Once you have downloaded results, they remain on the client, and you can review them at any time using the PolySpace Viewer.



---

## Downloading Server Results to UNIX or Linux Clients

If you are using PolySpace Client for on a UNIX or Linux system, the Queue Manager interface is not available. To download results from the PolySpace Server, you must use the `psqueue-download` command to download your results.

To download your results, enter the following command:

```
<PolySpaceCommonDir>/RemoteLauncher/bin/psqueue-download <id>  
<results dir>
```

The verification `<id>` is downloaded into the results directory `<results dir>`.

---

**Note** If you download results before the verification is complete, you get partial results and the verification continues.

---

Once you download results, they remain on the client, and you can review them at any time using the PolySpace Viewer.

The `psqueue-download` command has the following options:

- `[-f]` force download (without interactivity)
- `-admin -p <password>` allows administrator to download results.
- `[-server <name>[:port]]` selects a specific Queue Manager.
- `[-v|version]` gives release number.

---

**Note** When downloading a unit-by-unit verification group, all the unit results are downloaded and a summary of the download status for each unit is displayed.

---

For more information on managing verification jobs from the command line, see “Managing Verifications in Batch” on page 6-26.

### Downloading Results from Unit-by-Unit Verifications

If you run a unit-by-unit verification, each source file is sent to PolySpace Server individually. The queue manager displays a job for the full verification group, as well as jobs for each unit (using a tree structure).

You can download and view verification results for the entire project, or for individual units.

To download the results from unit-by-unit verifications:

- To download results for an individual unit, right-click the job for that unit, then select **Download Results**.

The individual results are downloaded and can be viewed as any other verification results.

- To download results for a verification group, right-click the group job, then select **Download Results**.

The results for all unit verifications are downloaded, as well as an HTML summary of results for the entire verification group.

### Opening Verification Results

You use the PolySpace Viewer to review the results of your verification.

---

**Note** You can also open the Viewer from the Launcher by clicking the Viewer icon in the Launcher toolbar with or without an open project.

---

To open the verification results:

- 1 Double-click the PolySpace Viewer icon:



- 2 Select **File > Open**.

**3** In the **Please select a file dialog box**, select the results file you want to view.

**4** Click the **Open** button.

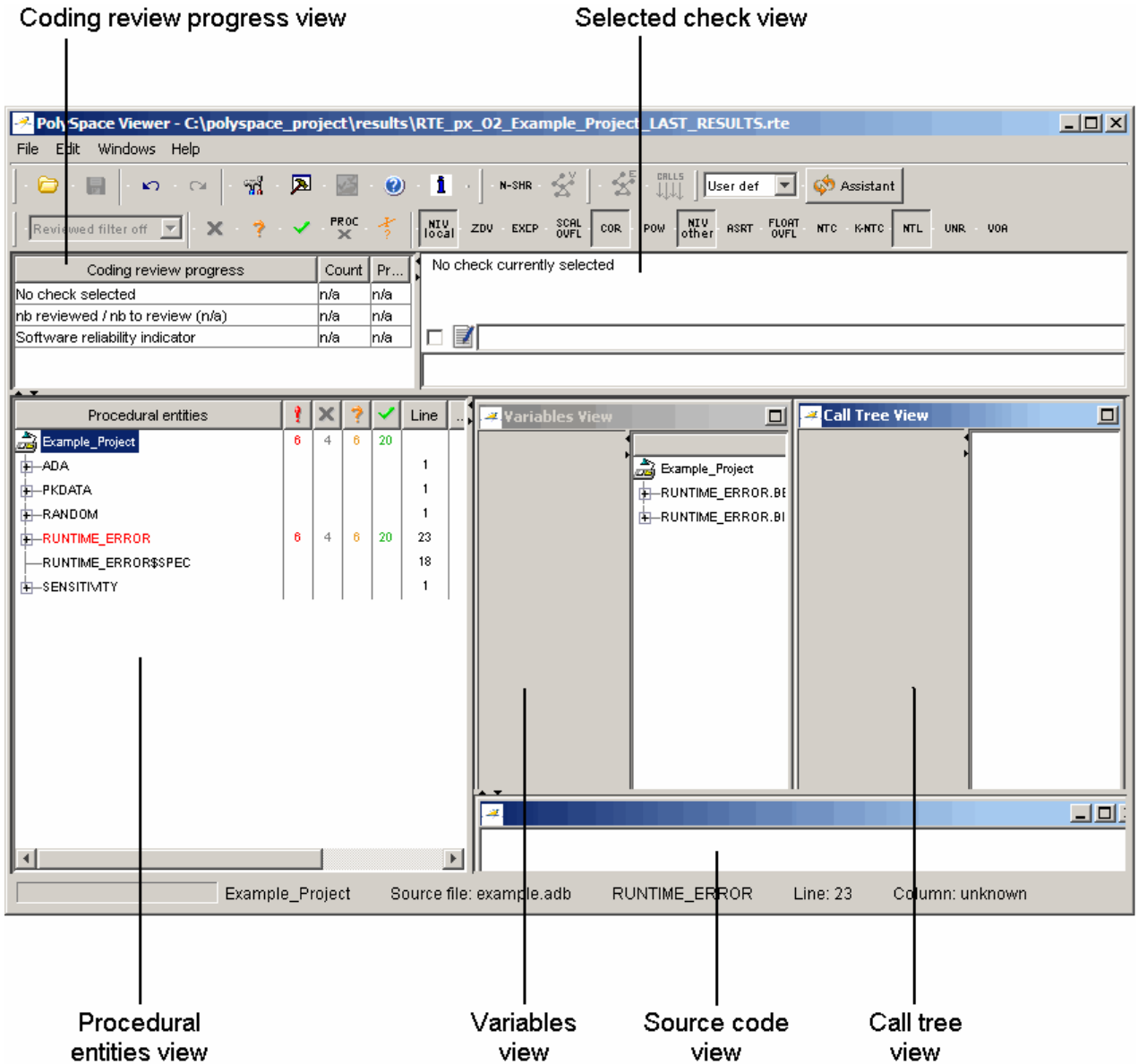
The results appear in the Viewer window.

## **Exploring the Viewer Window**

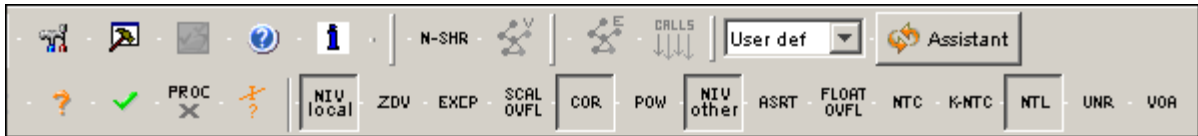
- “Overview” on page 8-13
- “Procedural Entities View” on page 8-15

### **Overview**

The PolySpace Viewer looks like:



The appearance of the Viewer toolbar depends on the Viewer mode. By default, the expert mode toolbar displays.



In both expert mode and assistant mode, the Viewer window has six sections below the toolbar. Each section provides a different view of the results. The following table describes these views.

This view...	Displays...
Procedural entities view (lower left)	List of the diagnostics (checks) for each file and function in the project
Source code view (lower right)	Source code for a selected check in the procedural entities view
Coding review progress view (upper left)	Statistics about the review progress for checks with the same type and category as the selected check
Selected check view (upper right)	Details about the selected check
Variables view	Information about the global variables declared in the source code  <b>Note</b> The file that you use in this tutorial does not have global variables.
Call tree view	Tree structure of function calls

You can resize or hide any of these sections.

## Procedural Entities View

The procedural entities view, in the lower-left part of the Viewer window, displays a table with information about the diagnostics for each file in the project. The procedural entities view is also called the RTE (run-time error) view. The procedural entities view looks like:

Procedural entities					Line	...	%	Data
Example_Project	6	4	6	20			83	
└─ADA					1		0	
└─PKDATA					1		0	
└─RANDOM					1		0	
└─ <b>RUNTIME_ERROR</b>	6	4	6	20	23		83	exampl
└─RUNTIME_ERROR\$SPEC					18		0	exampl
└─SENSITIVITY					1		0	

The package RUNTIME\_ERROR is red because it has a run-time error. PolySpace software assigns a file the color of the most severe error found in that file. The first column of the table is the procedural entity (the file or function). The following table describes some of the other columns in the procedural entities view.

Column Heading	Indicates
	Number of red checks (operations where an error always occurs)
	Number of gray checks (unreachable code)
	Number of orange checks (warnings for operations where an error might occur)
	Number of green checks (operations where an error never occurs)
	Selectivity of the verification (percentage of checks that are not orange) This is an indication of the level of proof.

**Tip** If you see three dots in place of a heading, , resize the column until you see the heading. Resize the procedural entities view to see additional columns.

---

**Note** You can select which columns appear in the procedural entities view by editing the preferences. To learn how to add a **Reviewed** column, see “Making the Reviewed Column Visible” on page 8-35.

---

What you select in the procedural entities view determines what displays in the other views. In the examples in this chapter, you learn how to use the views and how they interact.

## Selecting Viewer Mode

You can review verification results in *expert* mode or *assistant* mode:

- In expert mode, you decide how you review the results.
- In assistant mode, PolySpace software guides you through the results.

You switch from one mode to the other by clicking the appropriate button in the Viewer toolbar:



## Setting Character Encoding Preferences

If the source files you want to verify were created on an operating system that uses different character encoding than your current system (for example, when viewing files containing Japanese characters), you will 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:

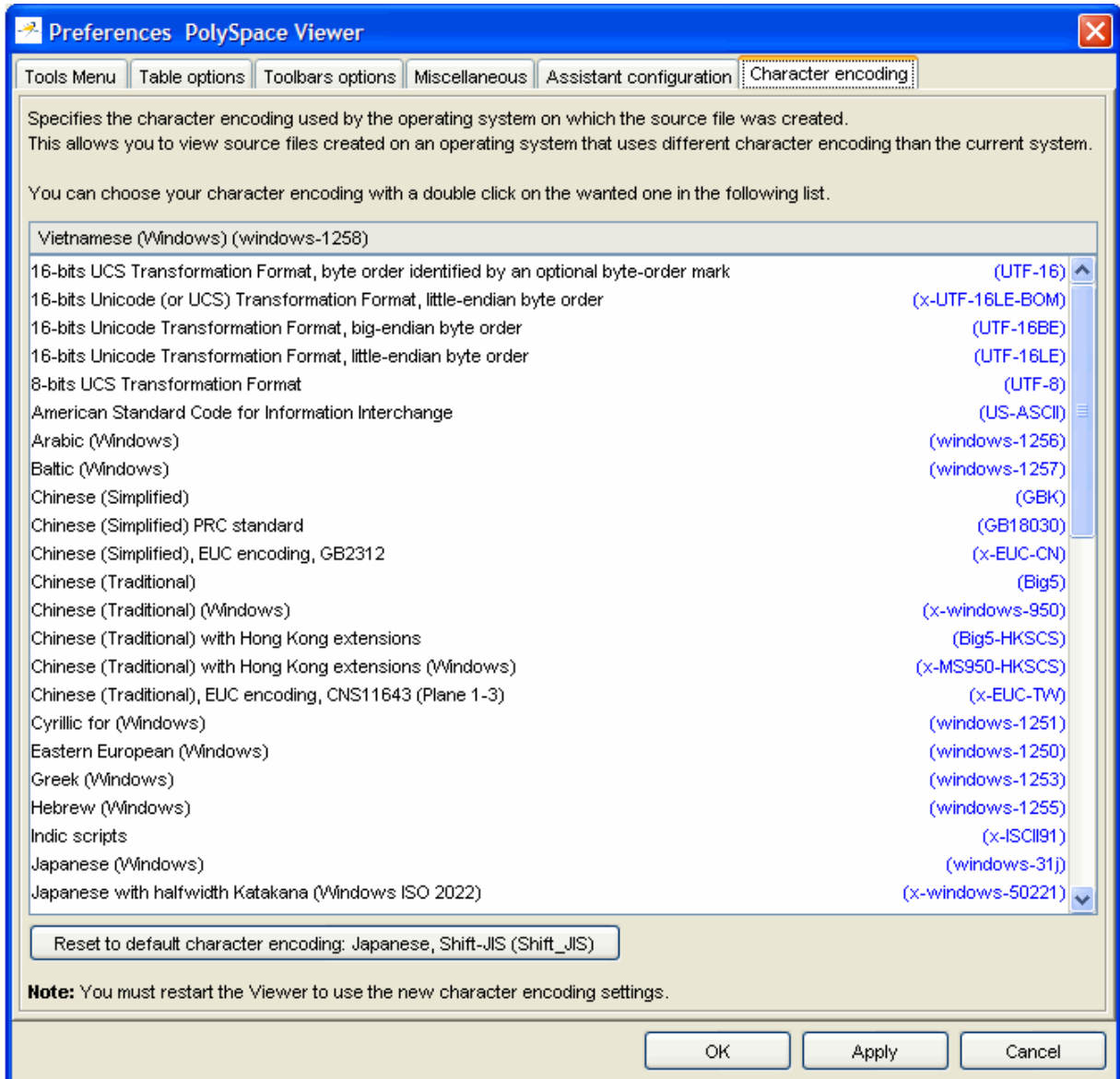
**1** Select **Edit > Preferences** in the Viewer.

The **Preferences PolySpace Viewer** dialog box appears.

**2** Select the **Character encoding** tab.

The Character encoding tab appears.





- 3 Select the character encoding used by the operating system on which the source file was created.

**4** Click **OK**.

---

**Note** You must close and restart the viewer to use the new character encoding settings.

---

**5** Close and restart the Viewer.

## Reviewing Results in Assistant Mode

### In this section...

- “What Is Assistant Mode?” on page 8-21
- “Switching to Assistant Mode” on page 8-21
- “Selecting the Methodology and Criterion Level” on page 8-22
- “Exploring Methodology for Ada” on page 8-23
- “Defining a Custom Methodology” on page 8-25
- “Reviewing Checks” on page 8-26
- “Saving Review Comments” on page 8-28

### What Is Assistant Mode?

In assistant mode, PolySpace software chooses the checks for you to review and the order in which you review them. PolySpace software presents checks to you in this order:

- 1 All red checks
- 2 All blocks of gray checks (the first check in each unreachable function)
- 3 Orange checks according to the selected methodology and criterion level

For more information about methodologies and criterion levels, see “Selecting the Methodology and Criterion Level” on page 8-22.

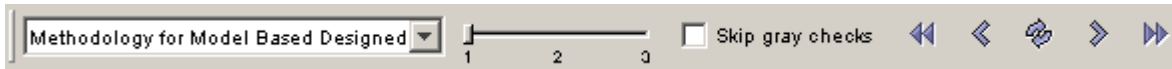
### Switching to Assistant Mode

If the Viewer is in assistant mode, the mode toggle button displays **Expert**. If the Viewer is in expert mode, the mode toggle button displays **Assistant**. To switch from expert mode to assistant mode:

- Click the Viewer mode button



The Viewer window toolbar displays controls specific to assistant mode.



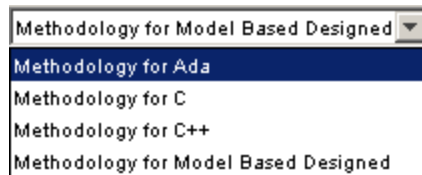
The controls for assistant mode include:

- A menu for selecting the review methodology for orange checks
- A slider for selecting the criterion level within that methodology
- A check box for skipping gray checks
- Arrows for navigating through the reviews

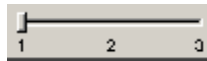
## Selecting the Methodology and Criterion Level

A methodology is a named configuration set that defines the number of orange checks, by category, that you review in assistant mode. Each methodology has three criterion levels. Each level specifies the number of orange checks for a given category. The levels correspond to different development phases that have different review requirements. To select a methodology and level:

- 1 Select **Methodology for Ada** from the methodology menu.



- 2 Select the appropriate level on the level slider.



For the configuration Methodology for Ada, the three levels are:

Level	Description
1	Fresh code
2	Unit tested code
3	Code Review

These three levels correspond to phases of the development process.

## Exploring Methodology for Ada

A methodology defines the number of orange checks that you review in assistant mode. Each methodology has three criterion levels that specify increasing levels of review. These levels correspond to different development phases that have different review requirements.

---

**Note** You cannot change the parameters defined in the Methodology for Ada, but you can create your own custom methodologies.

---

To examine the configuration for **Methodology for Ada**:

- 1 Select **Edit > Preferences**.

The **Preferences PolySpace Viewer** dialog box appears.

- 2 Select the **Assistant configuration** tab.

The configuration for Methodology for Ada appears.

On the right side of the dialog box, a table shows the number of orange checks that you review for a given criterion and check category.

aneous Assistant configuration

Number of checks to review

	Criterion 1	Criterion 2	Criterion 3
Common			
ZDV	10	20	ALL
NIVL	AUTO	50	ALL
S-OVFL	AUTO	50	ALL
COR	AUTO	10	10
POW	AUTO	10	ALL
NIV	AUTO	5	10
F-OVFL	5	10	20
ASRT	AUTO	5	20

For example, the table specifies that you review ten orange ZDV checks when you select criterion 1. The number of checks increases as you move from criterion 1 to criterion 3, reflecting the changing review requirements as you move through the development process.

In the lower-left part of the dialog box, the section **Review threshold criterion** contains text that appears in the tooltip for the criterion slider on the Viewer toolbar (in assistant mode).

Configuration set

Methodology for Ada

Review threshold criterion

Criterion 1	Fresh code
Criterion 2	Unit tested
Criterion 3	Code review

For the configuration Methodology for Ada, the criterion names are:

Criterion	Name in the Tooltip
1	Fresh code
2	Unit tested
3	Code Review

These names correspond to phases of the development process.

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

## Defining a Custom Methodology

A methodology defines the number of orange checks that you review in assistant mode. You cannot change the predefined methodologies, such as Methodology for Ada, but you can define your own methodology.

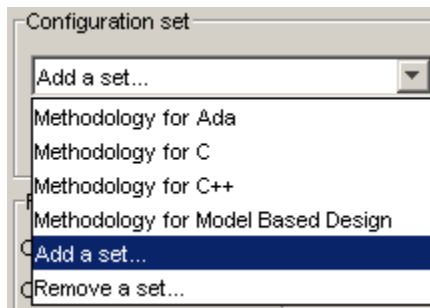
To define a custom methodology:

- 1 Select **Edit > Preferences**.

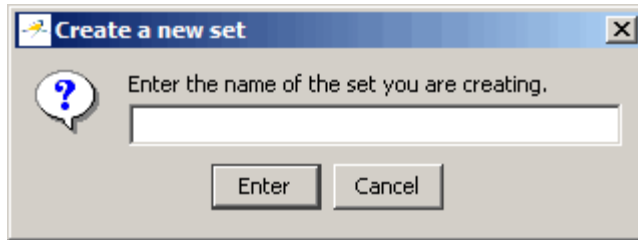
The **Preferences PolySpace Viewer** dialog box appears.

- 2 Select the **Assistant configuration** tab.

- 3 In the **Configuration set** drop-down menu, select **Add a set**.



The Create a new set dialog box appears.



- 4 Enter a name for the new configuration set, then click **Enter**.
- 5 Enter the number checks to review for each type, and each criterion level.
- 6 Click **OK** to save the methodology and close the dialog box.

### Reviewing Checks

In assistant mode, you review checks in the order in which PolySpace software presents them:

- 1 All reds
- 2 All blocks of gray checks (the first check in each unreachable function)


---

**Note** You can skip gray checks by selecting the **Skip gray checks** check box in the toolbar.

---

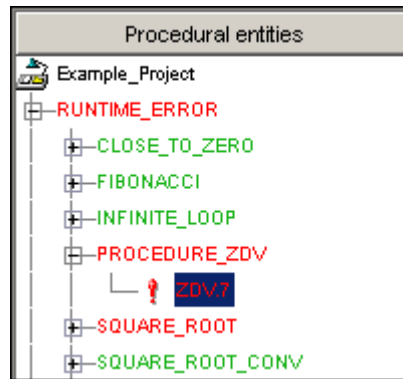
- 3 Orange checks according to the selected methodology and criterion level

To navigate through these checks:

- 1 Click the forward arrow .

  - The procedural entities view (lower left), expands to show the current check.





- The source code view (lower right) displays the source code for this check.
- The current check view (upper right) displays information about this check.

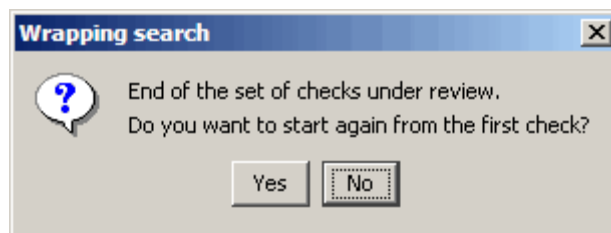
---

**Note** You can display the calling sequence and track review progress. See “Reviewing Results in Expert Mode” on page 8-29.

---

- 2 Review the current check.
- 3 Continue to click the forward arrow until you have gone through all of the checks.

After the last check, a dialog box appears asking if you want to start again from the first check.



- 4 Click No.

### **Saving Review Comments**

After you have reviewed your results, you can save your comments with the verification results. Saving your comments makes them available the next time you open the results file, allowing you to avoid reviewing the same check twice.

To save your review comments:

- 1** Select **File > Save Checks and Comments**.

Your comments are saved with the verification results.

---

**Note** Saving review comments also allows you to import those comments into subsequent verifications of the same module, allowing you to avoid reviewing the same check twice.

---

## Reviewing Results in Expert Mode

### In this section...

- “What Is Expert Mode?” on page 8-29
- “Switching to Expert Mode” on page 8-29
- “Selecting a Check to Review” on page 8-30
- “Displaying the Calling Sequence” on page 8-31
- “Displaying the Access Sequence for Variables” on page 8-32
- “Tracking Review Progress” on page 8-33
- “Making the Reviewed Column Visible” on page 8-35
- “Filtering Checks” on page 8-38
- “Types of Filters” on page 8-38
- “Creating a Custom Filter” on page 8-40
- “Saving Review Comments” on page 8-41

### What Is Expert Mode?

In expert mode, you can see all checks from the verification in the PolySpace Viewer. You decide which checks to review and in what order to review them.

### Switching to Expert Mode

If the Viewer is in expert mode, the mode toggle button displays **Assistant**. If the Viewer is in assistant mode, the mode toggle button displays **Expert**. To switch from assistant to expert mode:

- Click the Viewer mode button:



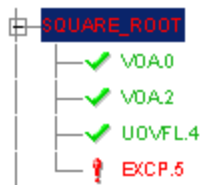
The Viewer window toolbar displays buttons and menus specific to expert mode.

## Selecting a Check to Review

To review a check in expert mode:

- 1 In the procedural entities section of the window, expand any file containing checks.
- 2 Expand the procedure containing the check you want to review.

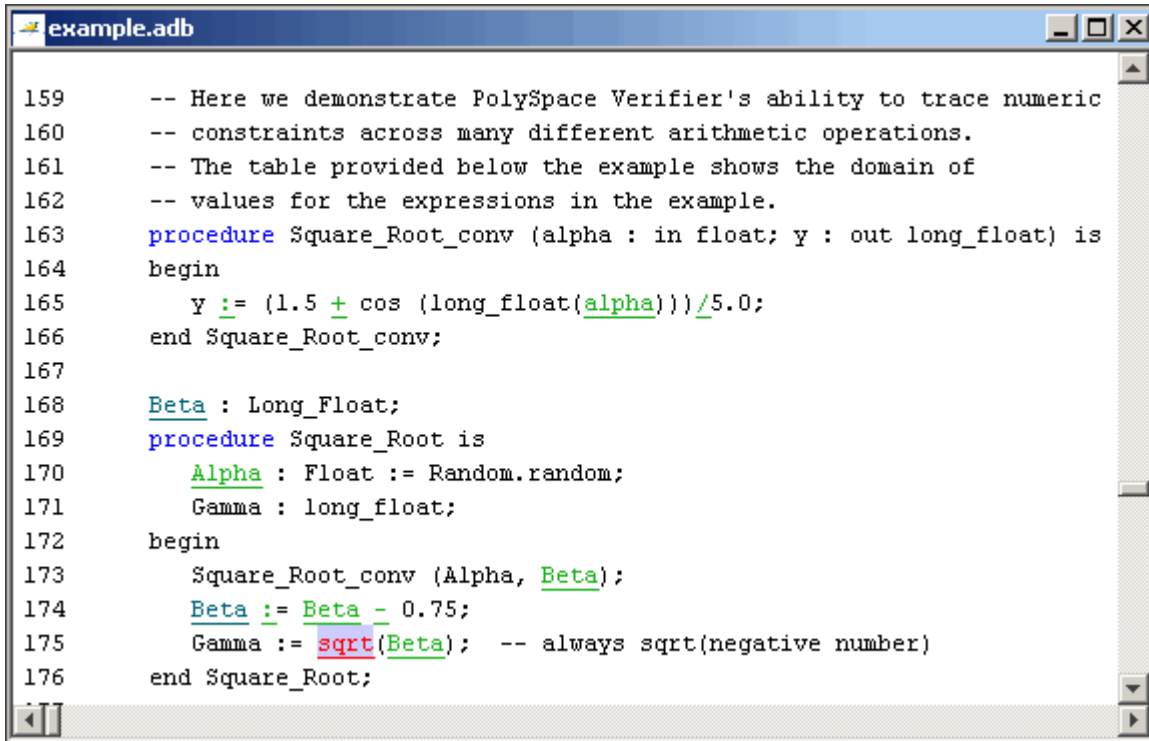
A color-coded list of the checks performed on the procedure appears:



Each item in the list of checks has an acronym that identifies the type of check and a number. For example, in `EXCP.5`, `EXCP` stands for Arithmetic Exception. For more information about different types of checks, see “Check Descriptions” in the *PolySpace Client/Server for Ada Reference*.

- 3 Click the check you want to review.

The source code view displays the section of source code where this error occurs.



```

159  -- Here we demonstrate PolySpace Verifier's ability to trace numeric
160  -- constraints across many different arithmetic operations.
161  -- The table provided below the example shows the domain of
162  -- values for the expressions in the example.
163  procedure Square_Root_conv (alpha : in float; y : out long_float) is
164  begin
165      y := (1.5 + cos (long_float(alpha)))/5.0;
166  end Square_Root_conv;
167
168  Beta : Long_Float;
169  procedure Square_Root is
170      Alpha : Float := Random.random;
171      Gamma : long_float;
172  begin
173      Square_Root_conv (Alpha, Beta);
174      Beta := Beta - 0.75;
175      Gamma := sqrt(Beta); -- always sqrt(negative number)
176  end Square_Root;

```

- 4 Mouse over any colored check in the code.

A tooltip provides ranges for variables, operands, function parameters, and return values.

- 5 Click the colored check in the code.

An message box appears describing the error.

## Displaying the Calling Sequence

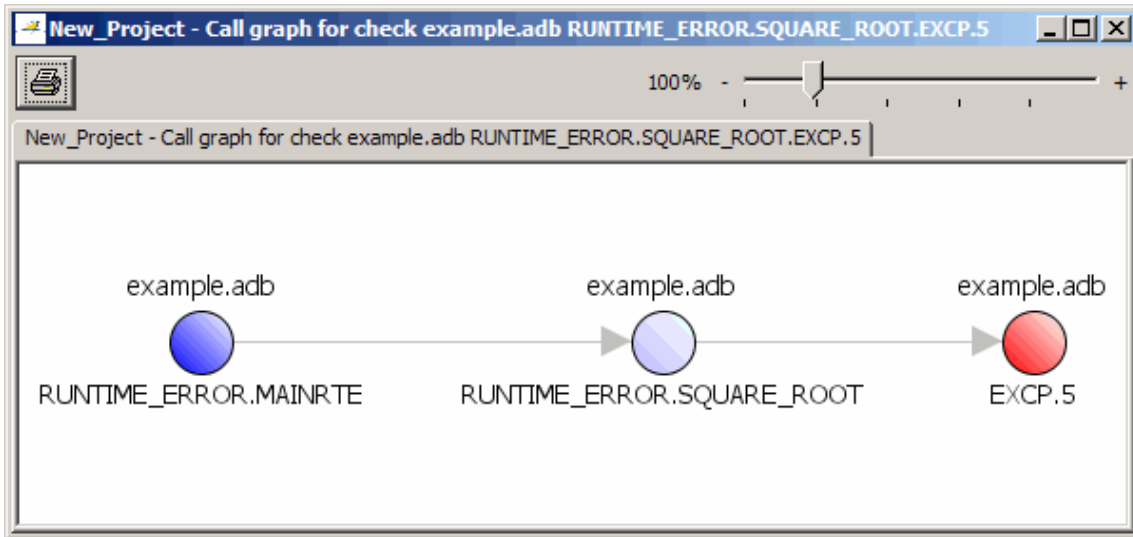
You can display the calling sequence that leads to the code associated with a check. To see the calling sequence for a check:

- 1 Expand the package containing the check you want to review.
- 2 Click the check you want to review.

- 3 Click the call graph button in the toolbar.



A window displays the call graph.



The call graph displays the code associated with the check.

## Displaying the Access Sequence for Variables

You can display the access sequence for any variable that is read or written in the code.

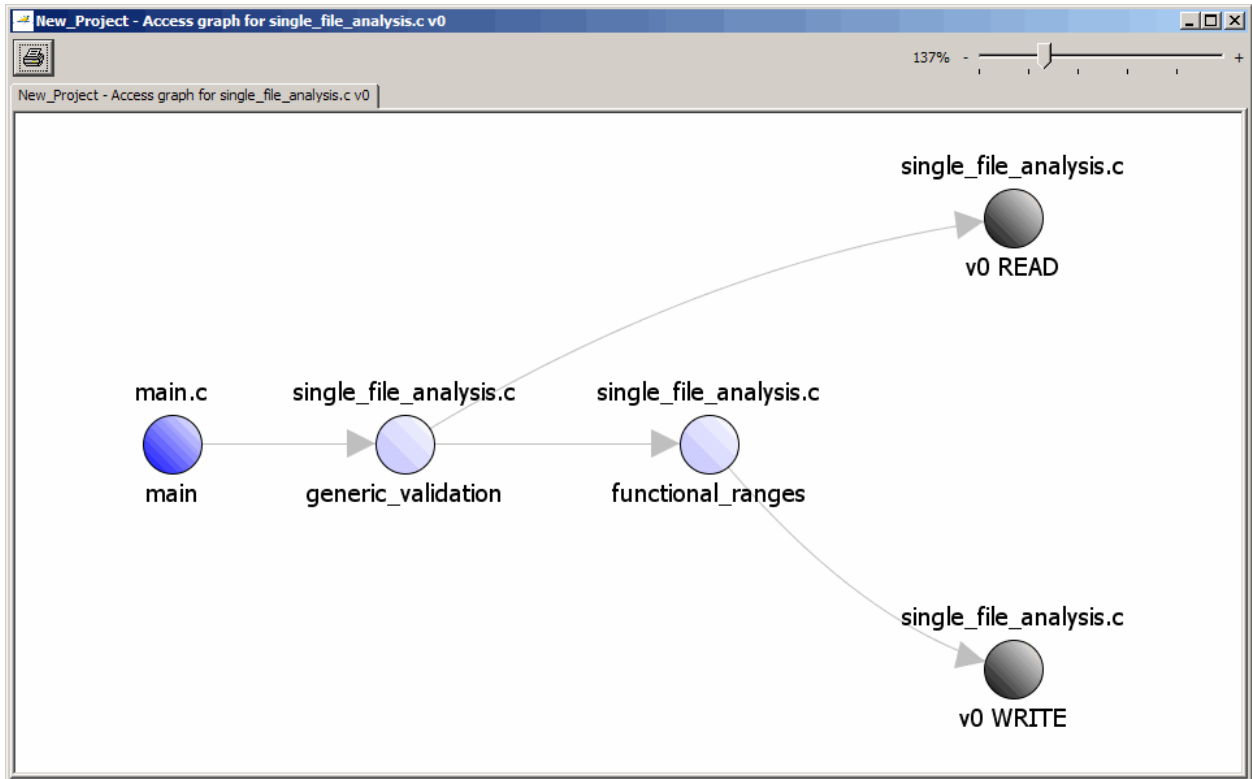
To see the access graph:

- 1 Select the Variables view.
- 2 Select the variable you want to view.

- 3 Click the call graph button in the toolbar.



A window displays the access graph.



The access graph displays the read and write access for the variable.

## Tracking Review Progress

You can keep track of the checks that you have reviewed by marking them. To mark that you have reviewed a check:

- 1 Expand the procedure containing the check you want to review.
- 2 Click the check you want to review.

A table with statistics about the review progress for that category and severity of error appear in the upper-left part of the window.


Coding review progress	Count	Progress
nb EXCP reviewed / nb EXCP to review (Red)	0/1	0
nb reviewed / nb to review (Red)	0/7	0
Software reliability indicator	37/55	67

The **Count** column displays a ratio and the **Progress** column displays the equivalent percentage. The first row displays the ratio of reviewed checks to the total number of checks that have the same color and category as the current check. In this example, it displays the ratio of reviewed red EXCP checks to total red EXCP errors in the project.

The second row displays the ratio of reviewed checks to total checks that have the same color as the current check. In this example, this is the ratio of red errors reviewed to total red errors in the project. The third row displays the ratio of the number of green checks to the total number of checks, providing an indicator of the reliability of the software.

Information about the current check appears in the upper-right part of the Viewer window.

```
example.adb / SQUARE_ROOT / line 175 / column 15
Gamma := sqrt(Beta); -- always sqrt(negative number)
```



```
certain float certain float failure of correctness condition [argument of SQRT must be positive]
```

- 3 Enter a comment in the comment box.
- 4 Select the check box to indicate that you have reviewed this check.

The **Coding review progress** part of the window updates the ratios of errors reviewed to total errors.



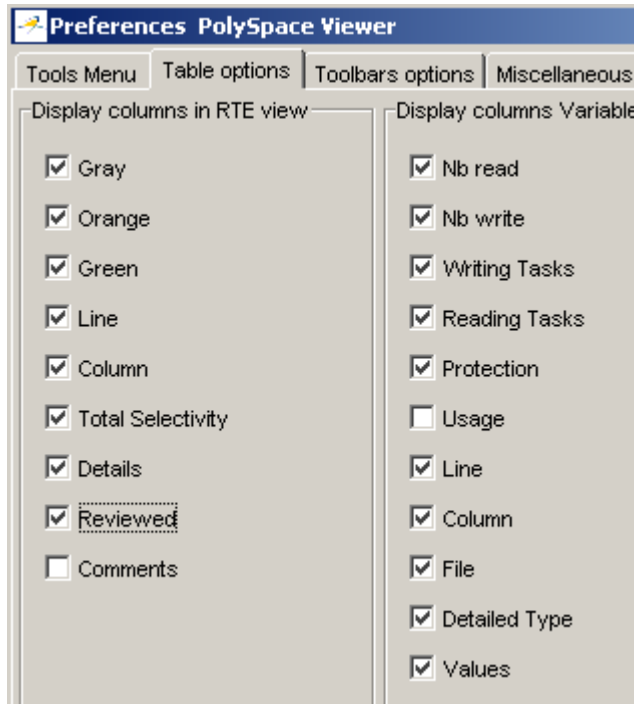
Coding review progress	Count	Progress
nb EXCP reviewed / nb EXCP to review (Red)	1/1	100
nb reviewed / nb to review (Red)	1/7	14
Software reliability indicator	37/55	67

## Making the Reviewed Column Visible

You can change the PolySpace Viewer preferences so that the procedural entities part of the window displays a **Reviewed** column.

- 1 Select **Edit > Preferences**.
- 2 Select the **Table options** tab.
- 3 Under **Display columns in RTE view**, select the **Reviewed** check box.

Now the **Table options** tab looks like:



**4** Click **OK** to apply the preference and close the dialog.

A column of check boxes appears in the **Procedural entities** view.

Procedural entities	!	X	?	✓	Line	...	%	Details	Reviewed
Example_Project	6	4	6	20			83		<input type="checkbox"/>
└ ADA					1		0		<input type="checkbox"/>
└ PKDATA					1		0		<input type="checkbox"/>
└ RANDOM					1		0		<input type="checkbox"/>
└ <b>RUNTIME_ERROR</b>	6	4	6	20	23		83	exampl...	<input type="checkbox"/>
└ CLOSE_TO_ZERO			4	1	91	3	20	exampl...	<input type="checkbox"/>
└ FIBONACCI				3	143	3	100	exampl...	<input type="checkbox"/>
└ INFINITE_LOOP				3	127	3	100	exampl...	<input type="checkbox"/>
└ MAINRTE	3				198	3	100	exampl...	<input type="checkbox"/>
└ MYABS		2			28	3	100	exampl...	<input type="checkbox"/>
└ NON_INFINITE_LOOP				3	114	3	100	exampl...	<input type="checkbox"/>
└ PROCEDURE_STUB					26	3	0	exampl...	<input type="checkbox"/>
└ <b>PROCEDURE_ZDV</b>	1	1		1	38	3	100	exampl...	<input type="checkbox"/>
└ RECURSION			1	4	66	3	80	exampl...	<input type="checkbox"/>
└ RECURSION_CALLER	1				77	3	100	exampl...	<input type="checkbox"/>
└ RECURSIVE_2					60	3	0	exampl...	<input type="checkbox"/>
└ <b>SQUARE_ROOT</b>	1			1	169	3	100	exampl...	<input type="checkbox"/>
└ ✓ VDA.0					170	6			<input type="checkbox"/>
└ ✓ VDA.2					174	11			<input type="checkbox"/>
└ ✓ UOVFL.4				1	174	19		[conve...	<input type="checkbox"/>
└ <b>EXCP.5</b>	1				175	15		argume...	<input checked="" type="checkbox"/>
└ <b>SQUARE_ROOT_CONV</b>				3	163	3	100	exampl...	<input type="checkbox"/>
└ UNREACHABLE_CODE		1	1	1	182	3	67	exampl...	<input type="checkbox"/>
└ ✓ VDA.0					113	3			<input type="checkbox"/>

**Tip** If you do not see this column, resize **Procedural entities** so that you see the column. Resize the column to see the **Reviewed** label.

---

**Note** Selecting a check box in the **Reviewed** column automatically:

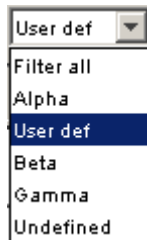
- Selects the check box for that check in the current check view (upper-right part of the window).
  - Updates the counts in the coding review progress view (upper-left part of the window).
- 

### Filtering Checks

You can filter the checks that you see in the Viewer so that you can focus on certain types of checks. PolySpace software provides three predefined composite filters, a custom composite filter, and several individual filters.

The default filter is `User def`.

To filter checks, select a filter from the filter menu.



### Types of Filters

There are three types of filters:

- “Individual Filters” on page 8-39
- “Composite Filters” on page 8-39
- “Custom Filters” on page 8-39

### Individual Filters

You can use an individual filter to display or hide a given check category, such as VOA. When a filter is enabled, that check category does not display. For example, when the VOA filter is enabled, VOA checks do not display. When the filter is disabled, that check category displays. For example, when the VOA filter is disabled, VOA checks display. You can also filter by check color. To enable or disable an individual filter, click the toggle button for that filter on the toolbar.

---

**Tip** The tooltip for a filter button tells you what filter the button is for and whether the filter is enabled or disabled.

---

---

**Note** When you filter a check category, some red checks with that category will still display.

---

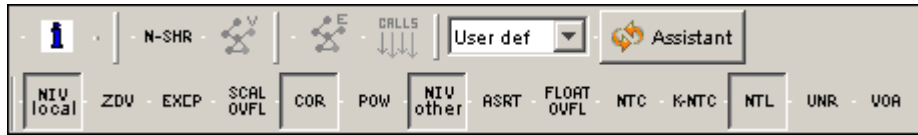
### Composite Filters

Composite filters combine individual filters, allowing you to display or hide groups of checks.

Use this filter...	To...
Alpha	Display all checks
Beta	Hide NIV, NIVL, NIP, Scalar OVFL, and Float OVFL checks
Gamma	Display red and gray checks
User def	Hide checks as defined in a custom filter that you can modify

### Custom Filters

The custom filter is a composite filter that you define. It appears on the composite filter menu as `User def` and is the default composite filter. By default, the custom filter hides the NIV local, COR, NIV other, and NTL checks as shown in the following figure.



To modify the custom filter, see “Creating a Custom Filter” on page 8-40.

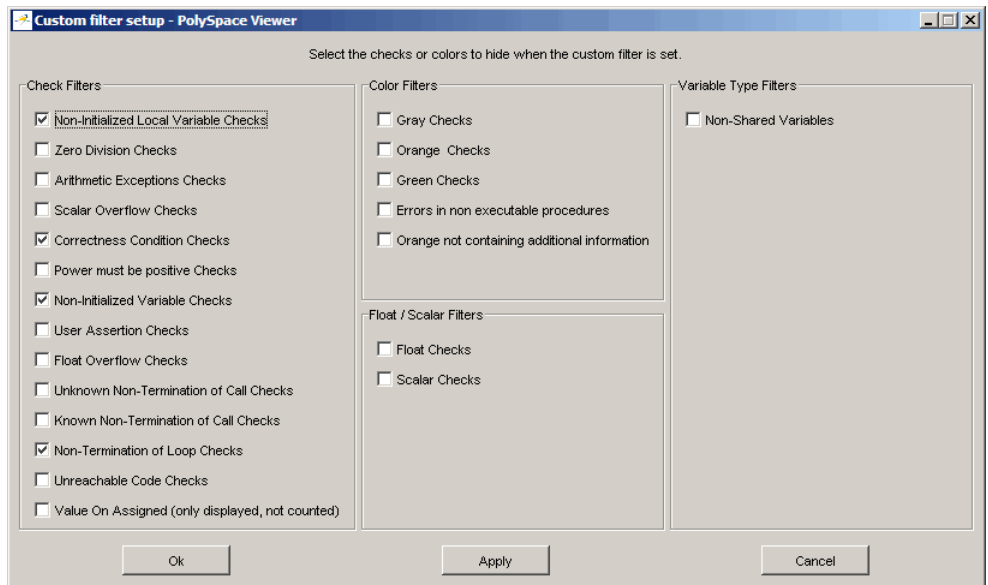
## Creating a Custom Filter

The custom filter is a composite filter that you define. It appears on the composite filter menu as `User def`.

To modify the custom filter:

- 1 Select `User def` from the composite filters menu.
- 2 Select **Edit > Custom filters**.

The **Custom filter setup** dialog box appears.



- 3 Clear the filters for the checks that you want to display. For example, if you clear the **Out of Bound Array Index Checks** box, these checks display.

---

**Note** You do not have to change any of the selections for this tutorial.

---

- 4 Select the filters for the checks that you do not want to display.
- 5 Click **OK** to apply the changes and close the dialog box.

PolySpace software saves the custom filter definition in the Viewer preferences.

## **Saving Review Comments**

After you have reviewed your results, you can save your comments with the verification results. Saving your comments makes them available the next time you open the results file, allowing you to avoid reviewing the same check twice.

To save your review comments:

- 1 Select **File > Save Checks and Comments**.

Your comments are saved with the verification results.

---

**Note** Saving review comments also allows you to import those comments into subsequent verifications of the same module, allowing you to avoid reviewing the same check twice.

---

## Importing and Exporting Review Comments

In this section...
“Reusing Review Comments” on page 8-42
“Exporting Review Comments to Other Verification Results” on page 8-42
“Importing Review Comments from Previous Verifications” on page 8-43

### Reusing Review Comments

After you have reviewed verification results on a module, you can reuse your review comments with subsequent verifications of the same module. This allows you to avoid reviewing the same check twice, or to compare results over time.

The PolySpace Viewer allows you to either:

- Export review comments from the current results to another set of results.
- Import review comments from another set of results into the current results.

---

**Note** If the code has changed since the previous verification, the imported comments may not be applicable to your current results. For example, the justification for an orange check may no longer be relevant to the current code.

---

### Exporting Review Comments to Other Verification Results

After you have reviewed verification results, you can export your review comments for use with other verifications of the same module, allowing you to avoid reviewing the same check twice.

---

**Caution** The comments you export replace any existing comments in the selected results.

---



To export review comments to other verification results:

- 1 Select **File > Export checks and comments**.
- 2 Navigate to the folder containing the other results file.
- 3 Select the results (.RTE) file, then click **Open**.

The review comments from the current results are exported into the selected results.

---

**Note** If the code has changed between the two verifications, the exported comments may not be applicable to the other results. For example, the justification for an orange check may no longer be relevant to the current code.

---

## Importing Review Comments from Previous Verifications

If you have previously reviewed verification results for a module and saved your comments, you can import those comments into the current verification, allowing you to avoid reviewing the same check twice.

---


**Caution** The comments you import replace any existing comments in the current results.

---

To import review comments from a previous verification:

- 1 Open your most recent verification results in the Viewer.
- 2 Select **File > Import checks and comments**.
- 3 Navigate to the folder containing your previous results.
- 4 Select the results (.RTE) file, then click **Open**.

The review comments from the previous results are imported into the current results.

Once you import checks and comments, the **go to next check**  icon in assistant mode will skip any reviewed checks, allowing you to review only checks that you have not reviewed previously. If you want to view reviewed checks, click the **go to next reviewed check** icon.

---

**Note** If the code has changed since the previous verification, the imported comments may not be applicable to your current results. For example, the justification for an orange check may no longer be relevant to the current code.

---

## Generating Reports of Verification Results

In this section...
“PolySpace Report Generator Overview” on page 8-45
“Generating Verification Reports” on page 8-46
“Automatically Generating Verification Reports” on page 8-47
“Generating Excel Reports” on page 8-48

### PolySpace Report Generator Overview

The PolySpace Report Generator allows you to generate reports about your verification results, using pre-defined report templates.

The PolySpace Report Generator provides the following report templates:

- **Coding Rules Report** – Provides information about compliance with MISRA-C Coding Rules, as well as PolySpace configuration settings used for the verification.
- **Developer Report** – Provides information useful to developers, including summary results, detailed lists of red, orange, and gray checks, and PolySpace configuration settings used for the verification.
- **Developer with Green Checks Report** – Provides the same content as the Developer Report, but also includes a detailed list of green checks.
- **Quality Report** – Provides information useful to quality engineers, including summary results, statistics about the code, graphs showing distributions of checks per file, and PolySpace configuration settings used for the verification.

The PolySpace Report Generator allows you to generate verification reports in the following formats:

- HTML
- PDF
- RTF

- WORD
- XML

---

**Note** WORD format is not available on UNIX platforms, RTF format is used instead.

---

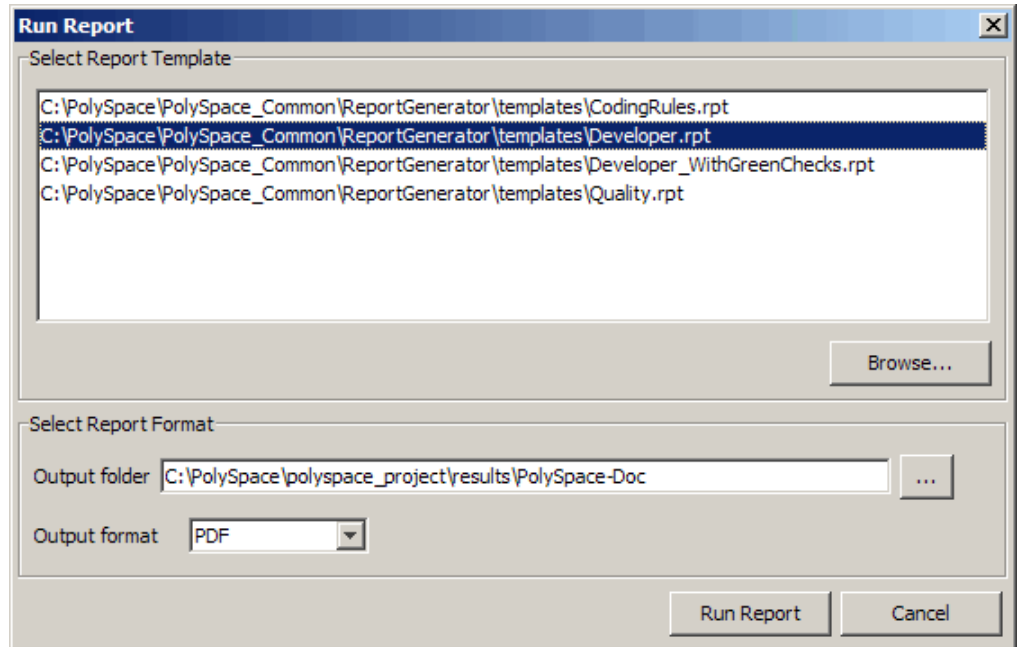
### **Generating Verification Reports**

You can generate reports for any verification results using the PolySpace Report Generator.

To generate a verification report:

- 1** Open your verification results in the Viewer.
- 2** Select **Reports > Run Report**.

The Run Report dialog box opens.



- 3** Select the type of report you want to run in the Select Report Template section.
- 4** Select the Output folder in which to save the report.
- 5** Select the Output format for the report.
- 6** Click **Run Report**.

The software creates the specified report.

## Automatically Generating Verification Reports

You can specify that PolySpace software automatically generate reports for each verification using an option in the Launcher .

To automatically generate reports for each verification:

- 1** Open your project in the Launcher.

**2** In the Analysis options section of the Launcher window, expand **General**.

The General options appear.

**3** Select **Report Generation**.

**4** Select the **Report template name**.

**5** Select the **Output format** for the report.

**6** Save your project.

### Generating Excel Reports

You can also generate a Microsoft® Excel® report of the verification results.

---

**Note** Excel reports do not use the PolySpace Report Generator.

---

To generate an Excel report of your verification results:

**1** Navigate to the PolySpace-Doc folder in your results directory. For example: `polyspace_project\results\PolySpace-Doc`.

The directory should have the following files:

```
Example_Project_Call_Tree.txt
Example_Project_RTE_View.txt
Example_Project_Variable_View.txt
Example_Project-NON-SCALAR-TABLE-APPENDIX.ps
PolySpace_Macros.xls
```

The first three files correspond to the call tree, RTE, and variable views in the PolySpace Viewer window.

**2** Open the macros file `PolySpace_Macros.xls`.

A security warning dialog appears.

**3** Click **Enable Macros**.

A spreadsheet appears. The top part of the spreadsheet looks like:

Apply filters?  No filters  Beta filters

Generate checks by file?  yes  no

Help Use this button to create the complete synthesis in one file. Select the RTE export view and a file in which to save results. If the other views are in the same directory as the RTE view then they will automatically be incorporated into the same file. Help

**Generate PolySpace Results Synthesis**

- 4** Specify the report options you want, then click **Generate PolySpace Results Synthesis**.

The synthesis report combines the RTE, call tree, and variables views into one report.

The **Where is the PolySpace RTE View text file** dialog box appears.

- 5** In **Look in**, navigate to the PolySpace -Doc folder in your results directory. For example: `polyspace_project\results\PolySpace-Doc`.
- 6** Select `Project_RTE_View.txt`.
- 7** Click **Open** to close the dialog box.

The **Where should I save the analysis file?** dialog box appears.

- 8** Keep the default file name and file type.
- 9** Click **Save** to close the dialog box and start the report generation.

Microsoft Excel opens with the spreadsheet that you generated. This spreadsheet has several worksheets:

Microsoft Excel - Example_Project-Synthesis.xls					
File Edit View Insert Format Tools Data Window Help					
	A	B	C	D	E
1	<b>Procedural entities</b>	<b>R</b>	<b>O</b>	<b>Gy</b>	<b>Gn</b>
2	Example_Project	7	6	5	37
3	-ADA				
4	-NUMERICS				
5	-AUX				
6	COS				
7	Sqrt				
8	-PKDATA				
9	ARRAY_OVERFLOW_INIT				
10	NON_INTRUSIVE_INFORMATION				
11	-RANDOM				
12	RANDOM				
13	RANDOM\$2				
14	RANDOM\$3				
15	-RUNTIME_ERROR	7	6	5	37
16	-CLOSE_TO_ZERO		4		1
17	V VOA . 0				
18	V VOA . 1				
19	? UOVFL . 2		1		
20	V VOA . 3				
21	V VOA . 4				
22	? UOVFL . 5		1		
23	V ZDV . 6				1
24	? UOVFL . 7		1		
25	? OVFL . 8		1		
26	-FIBONACCI				3
27	V VOA . 0				
28	V VOA . 1				
29	V VOA . 2				
30	V VOA . 3				

RTE Checks Sheet 1 Launching Options Check Synthesis



- 10 Select the **Check Synthesis** tab to view the worksheet showing statistics by check category:

Microsoft Excel - Example_Project-Synthesis.xls						
File Edit View Insert Format Tools Data Window Help						
	A	B	C	D	E	F
1	<b>RTE Statistics</b>					
2	<b>Check category</b>	<b>Check detail</b>	<b>R</b>	<b>O</b>	<b>Gy</b>	<b>Gr</b>
3	OBAI	Out of Bounds Array Index	0	0	0	0
4	NIVL	Uninitialized Local Variable	0	0	1	15
5	IDP	Illegal Dereference of Pointer	0	0	0	0
6	NIP	Uninitialized Pointer	0	0	0	0
7	NIV	Uninitialized Variable	0	0	0	2
8	IRV	Initialized Value Returned	0	0	0	0
9	COR	Other Correctness Conditions	0	0	0	0
10	ASRT	User Assertion Failure	0	0	0	0
11	POW	Power Must Be Positive	0	0	0	0
12	ZDV	Division by Zero	1	1	1	5
13	SHF	Shift Amount Within Bounds	0	0	0	0
14	OVFL	Overflow	0	2	0	0
15	UNFL	Underflow	0	0	0	0
16	UOVFL	Underflow or Overflow	0	3	2	15
17	EXCP	Arithmetic Exceptions	1	0	0	0
18	NTC	Non Termination of Call	4	0	0	0
19	k-NTC	Known Non Termination of Call	0	0	0	0
20	NTL	Non Termination of Loop	1	0	0	0

RTE Checks Sheet 1 Launching Options **Check Synthesis**

## Using PolySpace Results

### In this section...

“Review Runtime Errors: Fix Red Errors” on page 8-52

“Review Dead Code Checks: Why Gray Code is Interesting” on page 8-53

“Reviewing Orange: Automatic Methodology” on page 8-55

“Reviewing Orange Checks” on page 8-57

“Integration Bug Tracking” on page 8-57

“How to Find Bugs in Unprotected Shared Data” on page 8-58

“Dataflow Verification” on page 8-59

“Potential Side Effect of a Red Error” on page 8-59

“Checks on Procedure Calls with Default Parameters” on page 8-60

“\_INIT\_PROC Procedures” on page 8-62

### Review Runtime Errors: Fix Red Errors

All Runtime Errors highlighted by PolySpace verification are determined by reference to the language standard, and are sometimes implementation dependant — that is, they may be acceptable for a particular compiler but unacceptable according to the language standard.

Consider an overflow on a type restricted from -128 to 127. The computation of  $127+1$  cannot be 128, but depending on the environment a “wrap around” might be performed with a resulting value of -128.

This result is of course mathematically incorrect. If the value represents the altitude of a plane, this could result in a disaster.

By default, PolySpace verification doesn’t make assumptions about the way a variable is used. Any deviation from the recommendations of the language standard is treated as a red error, and must therefore be corrected.

PolySpace verification identifies two kinds of red checks

- Red errors which are compiler-dependant in a specific way. On some occasions a PolySpace option may be used to allow particular compiler specific behavior, and on others the code must be corrected in order to comply. An example of a PolySpace option to permit compiler specific behavior would be the option to force “IN/OUT” ADA function parameters to be initialized. Examples in C include options to deal with constant overflows, shift operation on negative values, etc.
- All other red errors must be fixed. They are bugs.

Most of the bugs you’ll find are easy to correct once they are identified. PolySpace verification identifies bugs regardless of their consequence, or of the ease with which they can be corrected.

## **Review Dead Code Checks: Why Gray Code is Interesting**

- “Functional Bugs Can Be Found in Gray Code” on page 8-53
- “Structural Coverage” on page 8-54

### **Functional Bugs Can Be Found in Gray Code**

PolySpace verification finds different types of dead code. Common examples include:

- Defensive code which is never reached
- Dead code due to a particular configuration
- Libraries which are not used to their full extent in a particular context
- Dead code resulting from bugs in the source code.

The causes of dead code listed in the examples below are taken from critical applications of embedded software by PolySpace verification.

- A lack of parenthesis and operand priorities in the testing clause can change the meaning significantly.
- Consider a line of code such as  
IF NOT a AND b OR c AND d

Now consider how misplaced parentheses might influence how that line behaves

```
IF NOT (a AND b OR c AND d)
```

```
IF (NOT (a) AND b) OR (c AND d))
```

```
IF NOT (a AND (b OR c) AND d)
```

- The test of variable inside a branch where the conditions are never met;
- An unreachable “else” clause where the wrong variable is tested in the “if” statement
- A variable that 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 (say)

As is the case for red errors, the consequence of dead code and the effort needed to deal with it is unpredictable. It can vary

- From 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 3 minutes code review discovering the bug.

Again, as for red errors, PolySpace doesn’t measure the impact of dead code.

The tool provides a list of dead code. A short code review will enable you to place each entry from that list into one of the five categories from the beginning of this chapter. Doing will identify known dead code and uncover real bugs.

**PolySpace experience is that at least 30% of gray code reveals real bugs.**

### **Structural Coverage**

PolySpace software always performs upper approximations of all 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, but it could conclude that no runtime error could be found.

PolySpace verification will find around 80% of dead code that the developer would find by doing structural coverage.

PolySpace verification is intended to be used as a productivity aid in dead code detection. It detects dead code which might take days of effort to find by any other means.

## Reviewing Orange: Automatic Methodology

During a verification, PolySpace is able to automatically highlight some orange checks considered as potential robustness issues in the code.

The automatic methodology separates a sub part of orange NIVL and orange OVFL from all oranges checks:

- All NIVL scalar local **oranges**. These NIV do not concern float, record (and component) and arrays.
- All OVFL/UNFL scalar **oranges** between subtypes: conversion of a subtype in a smaller subtype.

From a Methodology point of view, these checks need to be addressed first. As PolySpace is very precise on them, we can always deduce that an orange of this kind is most of the time synonymous of a robustness issue.

### Example

```

1 Package body Test is
2   ATab : array(0..9) of Integer := (Others => 0);
3   function Assign_array(X : integer) return Integer is
4     Y : Integer;
5     begin
6       y := ATab( X - 12); -- Warning UOVFL on operator - given by
7         -- the Automatic methodology
8     return y;
9   end Assign_Array;
10
11 function read_bus_status return boolean; -- function stubbed

```

```
12 procedure partial_init( New_Alt : in out Integer ) is
13   Y : boolean;
14   begin
15     if read_bus_status then
16       New_Alt := 12;
17       Y := True;
18     else
19       New_Alt := 120;
20     end if;
21     if Y then -- Warning NIVL on Y given by
22       -- the automatic methodology
23       New_Alt := New_Alt * 10;
24     end if;
25   end partial_init;
26 end Test;
```

In the example above, the automatic methodology filters all orange except:

- The orange **UOVFL** at line 6. The associated message associated to this orange says “Scalar variable may underflow/overflow on [conversion from -2\*\*31.. 2\*\*31-1 to 0..9]”. In this case we have a typical conversion in a smaller subtype and nothing around shows a defensive code against this robustness issue.
- The orange **NIVL** at line 21. The associated message associated to this orange says “Local variable may be not initialized”. In this case we have a typical example which leads to a robustness issue if the right branch is not executed.

#### **Activation and filter location:**

In both mode of review (expert or assistant) the automatic methodology is always active.

Opening the Viewer on results, chose expert mode, select “Alpha” filter and then, clicking on “**I/ ?**” button associated to tool tip “**Click to hide orange not associated to additional information**”, allows to show all oranges and only coming from the automatic methodology.

## Reviewing Orange Checks

Orange checks indicate *unproven code*. This means that the code can neither be proven safe, nor can it be proven to contain a runtime error.

The number of orange checks you review is determined by several factors, including:

- The stage of the development process
- Your quality objectives

There are also actions you can take to reduce the number of orange checks in your results.

For information on managing orange checks in your results, see Chapter 9, “Managing Orange Checks”.

## Integration Bug Tracking

By default, integration bug tracking can be achieved by applying the selective orange methodology to integrated code. Each error category will be more likely to reveal integration bugs, depending on the chosen coding rules for the project.

For instance, consider a function receives two unbounded integers. The presence of an overflow can only be checked at integration phase, since at unit phase the first mathematical operation will reveal an orange check.

Consider these two circumstances:

- When integration bug tracking is performed in isolation, a selective orange review will highlight most integration bugs. In this case, a PolySpace verification has been performed integrating tasks.
- When integration bug tracking is performed together with an exhaustive orange review at unit phase, a PolySpace verification has been performed on one or more files.

In this second case, an exhaustive orange review will already have been performed file by file. Therefore, at integration phase **only checks that have turned from green to another color** are worth assessing.

For instance, if a function takes a structure as an input parameter, the standard hypothesis made at unit level is that the structure is well initialized. This will consequentially display a green NIV check at the first read access to a field. But this might not be true at integration time, where this check can turn orange if any context does not initialize these fields.

These orange checks will reveal integration bugs.

### **How to Find Bugs in Unprotected Shared Data**

Based on the list of entry points in a multi-task application, PolySpace verification identifies a list of shared data and provides several pieces of information about each entry:

- The data type;
- A list of reading and writing accesses to the data through functions and entry points;
- The type of any implemented protection against concurrent access.

A shared data item is a global data item that is read from or written to by two or more tasks. It is unprotected from concurrent accesses when one task can access it whilst another task is in the process of doing so. All the possible situations are considered below.

- If there is a possible scenario which would lead to such conflict for a particular variable, then a bug exists and protection is required.
- If there are no such scenarios, then one of the following explanations may apply:
  - The compilation environment guarantees an atomic read/write access on variable of type less than 1, 2 bytes, and therefore all conflicts concerning a particular variable type still guarantee the integrity of the variables content. But beware when porting the code!
  - The variable is protected by a critical section or a mutual temporal exclusion. You may wish to include this information in the PolySpace launching parameters and reverify.



It is also worth checking whether variables are modified which are supposed to be constant. Use the variables dictionary.

## Dataflow Verification

Data flow verification is often performed within certification processes — typically in the avionic, aerospace or transport markets.

This activity makes heavy use of two features of PolySpace results, which are available any time after the Control and Data Flow verification phase.

- Call tree computation
- Dictionary containing read/write access to global variables. (This can also be used to build a database listing for each procedure, for its parameters, and for its variables.)

PolySpace software can help you to build these results by extracting information from both the call tree and the dictionary.

## Potential Side Effect of a Red Error

This section explains why when a red error has been found the verification continues but some cautions need to be taken. Consider this piece of code:

```
7 package body Main is
8   procedure Main is
9     X: array (1..5) of Integer;
10    Tmp: Integer;
11    Zero: Integer:= 0;
12    begin
13      X:= (1,2,3,4,5);
14      if (X(4) > X(5))
15        then
16          Tmp:= 1 / Zero;
17        end if;
18    end;
19
20 end;
```

PolySpace works by propagating data sets representing ranges of possible values throughout the call tree, and throughout the functions in that call tree. Sometimes, PolySpace internally subdivides the functions for verification, and the propagation of the data ranges need several iterations (or integration levels) to complete. That effect can be observed by examining the color of the checks on completion of each of those levels. It can sometimes happen that:

- PolySpace will detect gray code which exists due to a terminal RTE which will not be flagged in red until a subsequent integration level.
- PolySpace flags a **NTC** in red with the content in gray. This red NTC is the result of an imprecision, and should be gray.

Suppose that an NTC is hard to understand at given integration level (level 4):

- If other **red checks** exist at level 4, fix them and restart the verification
- Otherwise, look back through the results from each previous level to see whether other red errors can be located. If so, fix them and restart the verification

### Checks on Procedure Calls with Default Parameters

Some checks may be located on procedure calls. They correspond to default values assigned to parameters of a procedure.

#### Example

```
1 package DCHECK is
2   type Pixel is
3     record
4       X : Integer;
5       Y : Integer;
6     end record;
7   procedure MAIN;
8
9   NError : Integer;
10  procedure Failure (Val : Integer := Nerror);
11  procedure MessageFailure (str : String := "");
12  end DCHECK;
13
```

```

14 package body DCHECK is
15   type TwentyFloat is array (Integer range 1.. 20) of Float;
16
17   procedure AddPixelValue(Vpixel : Pixel) is
18   begin
19     if (Vpixel.X < 3) then
20       Failure; -- NIV Verified: Variable is initialized
(Nerror)
21       MessageFailure; -- COR Verified: Value is in range (string)
22     end if;
23   end AddPixelValue;
24
25   procedure MAIN is
26     B : Twentyfloat;
27     Vpixel : Pixel;
28   begin
29     NError := 12;
30     Vpixel.X := 1;
31     AddPixelValue(Vpixel);
32     NError := -1;
33     for I in 2 .. Twentyfloat'Last loop
34       if ((I mod 2) = 0) then
35         B(I) := 0.0;
36         if (I mod 2) /= 0 then
37           Failure; -- NIV Unreachable: Variable is not
initialized
38           MessageFailure; -- COR Unreachable: Value is not in range
39         end if;
40       end if;
41     end loop;
42     MessageFailure("end of Main");
43   end MAIN;
44 end DCHECK;

```

## Explanation

In the previous example, at line 20 and 37, checks on the procedure calls Failure represent the check NIV made on the default parameter N error (a global parameter).

In the same way, COR checks at line 21 and 38 on `MessageFailure` represent verification made by PolySpace on the default assignment of a null string value on the input parameter.

---

**Note** Not all the checks have been moved to procedure calls. Checks remain on the procedure definition except for the following basic types and values:

- A numerical value (example: 1, 1.4)
  - A string (example: “end of main”)
  - A character (example: A)
  - A variable (example: `Nerror`).
- 

### **\_INIT\_PROC Procedures**

In the PolySpace viewer, it could be possible to find nodes `_INIT_PROC$` in the “Procedural entities” view. As your compiler, PolySpace generates a function `_INIT_PROC` for each record where initialization occurs. When a package defines many records, each `_INIT_PROC` is differentiated by `$I` (I in 1.n).

### **Example**

```
1 package test is
2   procedure main;
3 end test;
4
5 package body test is
6
7   subtype range_0_3 is integer range 0..3;
8   Vg : Integer := 1;
9   Pragma Volatile( Vg );
10
11   function random return integer;
12   type my_rec1 is
13     record
14       a : integer := 2 + random; -- Unproven OVFL coming from
```

```
_INIT_PROC procedure (initialization of V1)
15     b : float := 0.2;
16     end record;
17     V1 : my_rec1;
18     V2 : my_rec1 := (10, 10.10);
19
20     procedure main is
21         Function Random return Boolean;
22     begin
23         null;
24     end;
25 end test;
```

### Explanation

In the previous example, an unproven OVFL on the field a of record my\_rec1 has been detected when initializing the global variable V1. It initializes record of global variable V1 at line 17. Indeed, random procedure could return any value in the integer type and so, leads to an overflow by adding to 2. Check is located in the \_INIT\_PROC node into “Procedural entities” view.



# Managing Orange Checks

---

- “Understanding Orange Checks” on page 9-2
- “Too Many Orange Checks?” on page 9-9
- “Reducing Orange Checks in Your Results” on page 9-11
- “Reviewing Orange Checks” on page 9-23

## Understanding Orange Checks

In this section...
“What is an Orange Check?” on page 9-2
“Sources of Orange Checks” on page 9-6

### What is an Orange Check?

Orange checks indicate *unproven code*. This means that the code can neither be proven safe, nor can it be proven to contain a runtime error.

PolySpace verification does not try to find bugs, it attempts to prove the absence or existence of run time errors. Therefore, all code starts out as unproven prior to verification. The verification then attempts to prove that the code is either correct (green), is certain to fail (red), or is unreachable (gray). Any remaining code stays unproven (orange).

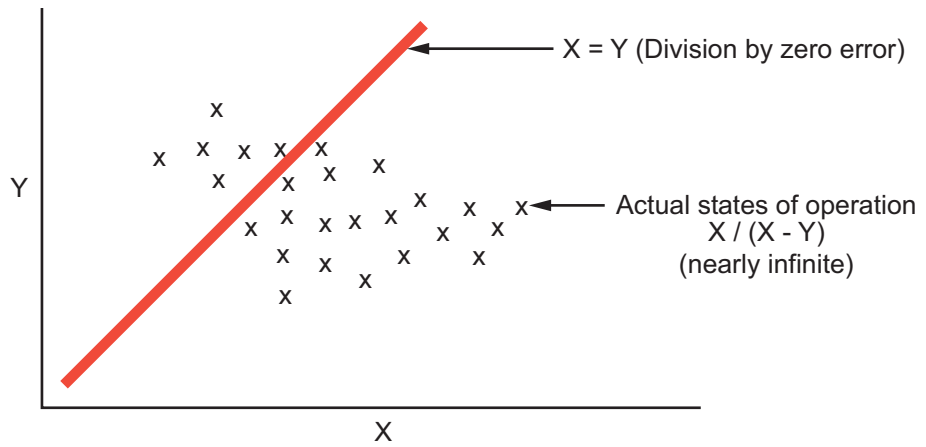
Code often remains unproven in situations where some paths fail while others succeed. For example, consider the following instruction:

$$X = 1 / (X - Y);$$

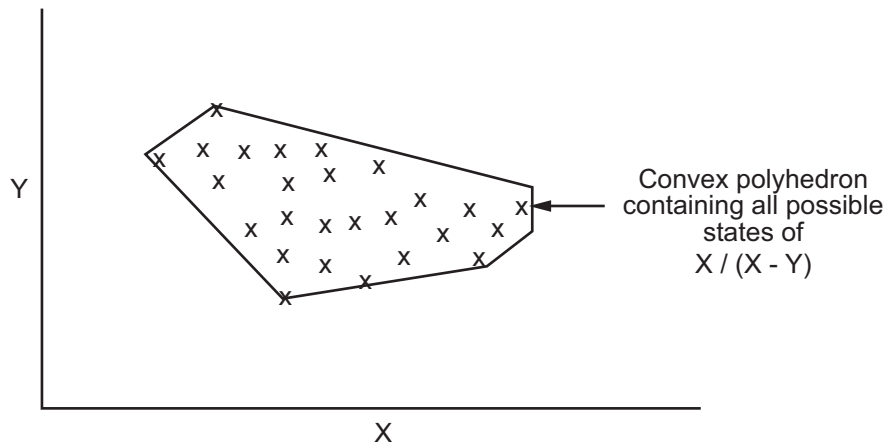
Does a division-by-zero error occur?

The answer clearly depends on the values of  $X$  and  $Y$ . However, there are an almost infinite number of possible values. Creating test cases for all possible values is not practical.

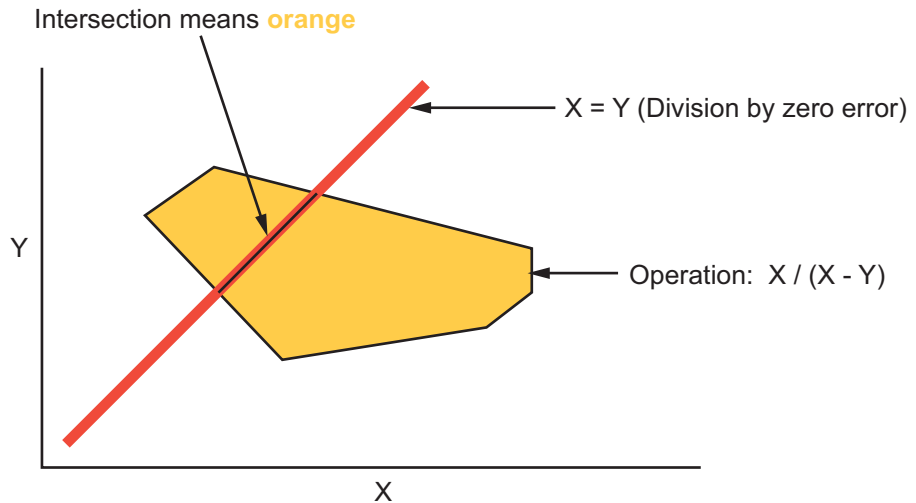




Although it is not possible to test every value for each variable, the target computer and programming language provide limits on the possible values of the variables. PolySpace verification uses these limits to compute a *cloud of points* (upper-bounded convex polyhedron) that contains all possible states for the variables.

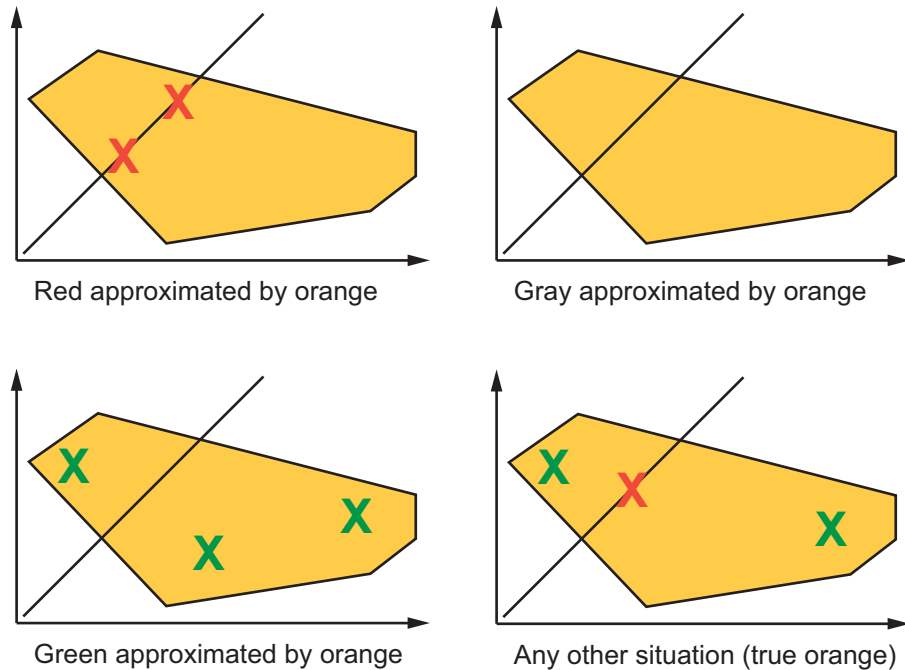


PolySpace verification then compares the data set represented by this polyhedron to the error zone. If the two data sets intersect, the check is orange.



**Graphical Representation of an Orange Check**

A true orange check represents a situation where some paths fail while others succeed. However, because the data set used in the verification is an approximation of actual values, an orange check may actually represent a check of any other color, as shown below.



PolySpace reports an orange check any time the two data sets intersect, regardless of the actual values. Therefore, you may find orange checks that represent bugs, while other orange checks represent code that is safe.

You can resolve some of these orange checks by increasing the precision of your verification, or by adding execution context, but often you must review the results to determine the source of an orange check.

### Sources of Orange Checks

Orange checks can be caused by any of the following:

- Potential bug
- Inconclusive check
- Data set issue
- Basic imprecision

Bugs can be revealed by any of these categories except for basic imprecision.

### 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 ZDV.

### 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.

The verification cannot determine if an error will occur unless you define the call sequence.

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. You may want to recode to ensure there is no risk, depending on the criticality of the function and the required speed of execution.

### **Data Set Issue**

An orange check can result from a theoretical set of data that cannot actually occur.

PolySpace verification uses an *upper approximation* of the data set, meaning that it considers all combinations of input data rather than any particular combination. Therefore, an orange check may result from a combination of input values that is not possible at execution time.

For example, consider three variables  $X$ ,  $Y$ , and  $Z$ :

- Each of these variables is defined as being between 1 and 1,000.
- The code computes  $X*Y*Z$  on a 16-bit data type.
- The result can potentially overflow, so it causes an orange OVFL.

When developing the code, you may know that the three variables cannot all take the value 1,000 at the same time, but this information is not available to the verification. Therefore, the multiplication is orange.

When an orange check is caused by a data set issue, it is usually possible to identify the cause quickly. After identifying a data set issue, you may want to comment the code to flag the warning, or modify the code to take the constraints into account.

### 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 any value in range  $[10 \dots 20]$ . This means that 0 has been excluded from the set of possible values for  $X$ .
- However, due to optimization at low precision levels (-00), the verification approximates  $X$  in the range  $[-5 \dots 20]$ , instead of the previous set of values.
- Therefore, calling the function  $x = 1/x$  causes an orange ZDV.

PolySpace verification is unable to prove the absence of a run-time error in this case.

In cases of basic imprecision, you may be able to resolve orange checks by increasing the precision level. If this does not resolve the orange check, verification cannot help directly. You need to review the code to determine if there is an actual problem.

For more information, see and “Approximations Used During Verification” in the *PolySpace Products for Ada Reference*.

## Too Many Orange Checks?

In this section...
“Do I Have Too Many Orange Checks?” on page 9-9
“How to Manage Orange Checks” on page 9-10

### Do I Have Too Many Orange Checks?

If the goal of code verification is to prove the absence of run time errors, you may be concerned by the number of orange checks (unproven code) in your results.

In reality, asking “Do I have too many orange checks?” is not the right question. There is not an ideal number of orange checks that applies for all applications, not even zero. Whether you have too many orange checks depends on:

- **Development Stage** – Early in the development cycle, when verifying the first version of a software component, a developer may want to focus exclusively on finding red errors, and not consider orange checks. As development of the same component progresses, however, the developer may want to focus more on orange checks.
- **Application Requirements** – There are actions you can take during coding to produce more provable code. However, writing provable code often involves compromises with code size, code speed, and portability. Depending on the requirements of your application, you may decide to optimize code size, for example, at the expense of more orange checks.
- **Quality Goals** – PolySpace software can help you meet quality goals, but it cannot define those goals for you. Before you verify code, you must define quality goals for your application. These goals should be based on the criticality of the application, as well as time and cost constraints.

It is these factors that ultimately determine how many orange checks are acceptable in your results, and what you should do with the orange checks that remain.

Thus, a more appropriate question is “How do I manage orange checks?”

This question leads to two main activities:

- Reducing the number of orange checks
- Working with orange checks

### **How to Manage Orange Checks**

PolySpace verification cannot magically produce quality code at the end of the development process. Verification is a tool that helps you measure the quality of your code, identify issues, and ultimately achieve the quality goals you define. To do this, however, you must integrate PolySpace verification into your development process.

Similarly, you cannot successfully manage orange checks simply by using PolySpace options. To manage orange checks effectively, you must take actions while coding, when setting up your verification project, and while reviewing verification results.

To successfully manage orange checks, perform each of the following steps:

- 1** Define your quality objectives to set overall goals for application quality. See “Defining Quality Objectives” on page 2-5.
- 2** Set PolySpace analysis options to match your quality objectives. See “Specifying Options to Match Your Quality Objectives” on page 3-19.
- 3** Define a process to reduce orange checks. See “Reducing Orange Checks in Your Results” on page 9-11.
- 4** Apply the process to work with remaining orange checks. See “Reviewing Orange Checks” on page 9-23.



## Reducing Orange Checks in Your Results

### In this section...

“Overview: Reducing Orange Checks” on page 9-11

“Applying Coding Rules to Reduce Orange Checks” on page 9-12

“Improving Verification Precision” on page 9-12

“Stubbing Parts of the Code Manually” on page 9-16

“Describing Multitasking Behavior Properly” on page 9-21

### Overview: Reducing Orange Checks

There are several actions you can take to reduce the number of orange checks in your results.

However, it is important to understand that while some actions increase the quality of your code, others simply change the number of orange checks reported by the verification, without improving code quality.

Actions that reduce orange checks and improve the quality of your code:

- **Apply coding rules** – Coding rules are the most efficient means to reduce oranges, and can also improve the quality of your code.

Actions that reduce orange checks through increased verification precision:

- **Set precision options** – There are several PolySpace options that can increase the precision of your verification, at the cost of increased verification time.
- **Implement manual stubbing** – Manual stubs that accurately emulate the behavior of missing functions can increase the precision of the verification.
- **Specify multitasking behavior** – Accurately defining call sequences and other multitasking behavior can increase the precision of the verification.

Options that reduce orange checks but do not improve code quality or the precision of the verification:

- **Create empty stubs** – Providing empty stubs for missing functions can reduce the number of orange checks in your results, but does not improve the quality of the code.

Each of these actions have trade-offs, either in development time, verification time, or the risk of errors. Therefore, before taking any of these actions, it is important to define your quality objectives, as described in “Defining Quality Objectives” on page 2-5.

It is your quality objectives that determine how many orange checks are acceptable in your results, what actions you should take to reduce the number of orange checks, and what you should do with any orange checks that remain.

### Applying Coding Rules to Reduce Orange Checks

The number of **orange checks** per file strongly depends on the coding style used in the project.

Here is a list of simple rules that allow PolySpace to be more precise and will higher the selectivity of any Ada verification:

- Use constrained types. Use subtype and not standard type
- Do not use "use at" clause
- Do not use `unchecked_conversion`
- 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 all assembly code in a procedure/function which can be automatically stubbed.

### Improving Verification Precision

Improving the precision of a verification can reduce the number of orange checks in your results, although it does not affect the quality of the code itself.

There are a number of PolySpace options that affect the precision of the verification. The trade off for this improved precision is increased verification time.

The following sections describe how to improve the precision of your verification:

- “Balancing Precision and Verification Time” on page 9-13
- “Setting the Analysis Precision Level” on page 9-14
- “Setting Software Safety Analysis Level” on page 9-16

### **Balancing Precision and Verification Time**

When performing code verification, you must find the right balance between precision and verification time. Consider the two following extremes:

- If a verification runs in one minute but contains only orange checks, the verification is not useful because each check must be reviewed manually.
- If a verification contains no orange checks (only gray, red, and green), but takes six months to run, the verification is not useful because of the time spent waiting for the results.

Higher precision yields more proven code (red, green, and gray), but takes longer to complete. The goal is therefore to get the most precise results in the time available. Factors that influence this compromise include the time available for verification, the time available to review results, and the stage in the development cycle.

For example, consider the following scenarios:

- **Unit testing** – Before going to lunch, a developer starts a verification. After returning from lunch the developer will review verification results for one hour.
- **Integration testing** – Before going home, a developer starts a verification. The developer will spend the next morning reviewing verification results.
- **Validation testing** – Before leaving the office on Friday evening, a developer starts a verification. The developer will spend the following week reviewing verification results.

Each of these scenarios require the developer to use PolySpace software in different ways. Generally, the first verification should use the lowest precision mode, while subsequent verifications increase the precision level.

---

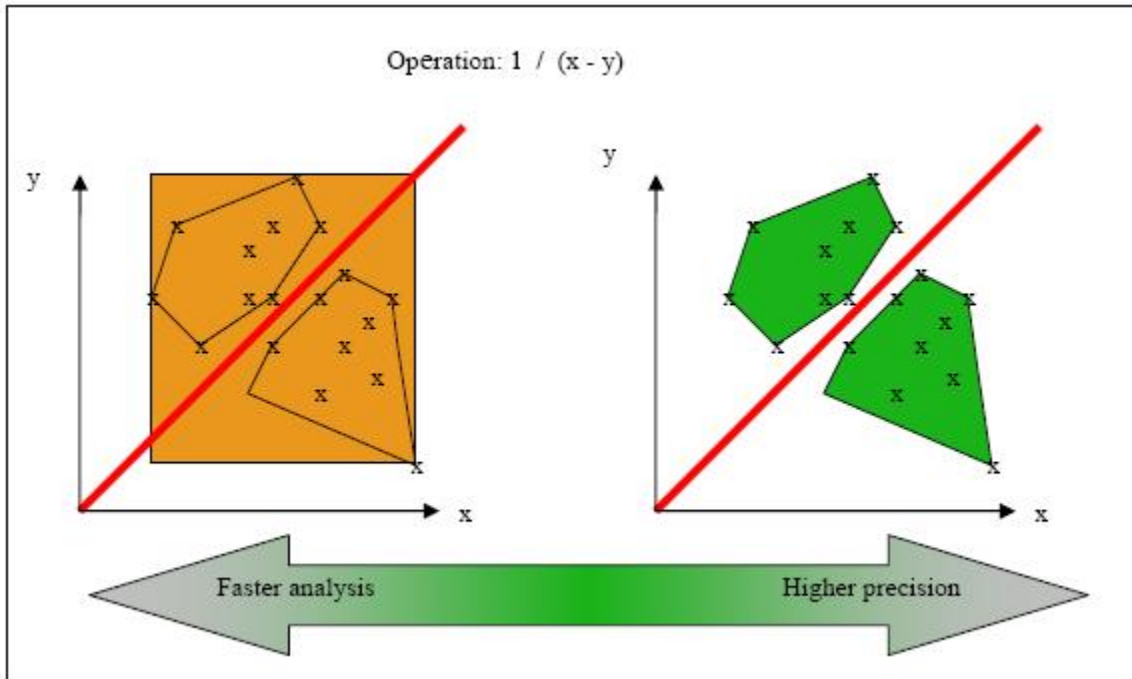
**Note** It is possible that a verification never ends. In this case, you may need to split the application.

---

### **Setting the Analysis Precision Level**

The analysis **Precision Level** specifies the mathematical algorithm used to compute the cloud of points (polyhedron) containing all possible states for the variables.

Although changing the precision level does not affect the quality of your code, orange checks caused by low precision become green when verified with higher precision.



### Affect of Precision Rate on Orange Checks

To set the precision level:

- 1** In the Analysis options section of the Launcher window, select **Precision/Scaling > Precision**.
- 2** Select the -00, -01, -02 or -03 precision level the Precision Level drop-down list.

For more information, see “Precision Level” in the *PolySpace Products for Ada Reference*.

---

**Note** You can select specific precision levels for individual modules in the verification.

---

### Setting Software Safety Analysis Level

The Software Safety Analysis level of your verification specifies how many times the abstract interpretation algorithm passes through your code. The deeper the verification goes, the more precise it is.

There are 5 Software Safety Analysis levels (pass0 to pass4). By default, verification proceeds to pass4, although it can go further if required. Each iteration results in a deeper level of propagation of calling and called context.

To set the Software Safety Analysis level:

- 1 In the Analysis options section of the Launcher window, select **Precision/Scaling > Precision**.
- 2 Select the appropriate level in the **To end of** drop-down list.

For more information, see “To end of” in the *PolySpace Products for Ada Reference*.

---

**Note** The Software Safety Analysis level applies to the entire application. You cannot select specific levels for individual modules in the verification.

---

### Stubbing Parts of the Code Manually

Manually stubbing parts of your code can reduce the number of orange checks in your results. However, manual stubbing generally does not improve the quality of your code, it only changes the results.

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

If a function is supposed to return an integer, the default automatic stubbing will stub it on the assumption that it can potentially take any value from the full type of an integer.

The following sections describe how to reduce orange checks using manual stubbing:

- “Manual vs. Automatic Stubbing” on page 9-17
- “Emulating Function Behavior with Manual Stubs” on page 9-18
- “Reducing Orange Checks with Empty Stubs” on page 9-19
- “Applying Constraints to Variables Using Stubs” on page 9-20

## Manual vs. Automatic Stubbing

There are two types of stubs in PolySpace verification:

- **Automatic stubs** – The software automatically creates stubs for unknown functions based on the function’s prototype (the function declaration). Automatic stubs do not provide insight into the behavior of the function, but are very conservative, ensuring that the function does not cause any runtime errors.
- **Manual stubs** – You create these stub functions to 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.

## Stubbing Example

The following example shows the effect of automatic stubbing.

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

Due to automatic stubbing, the verification assumes that  $a$  can be any integer, including 0. This produces an orange check on the division.

If you provide an empty manual stub for the function, the division would be green. This reduces the number of orange checks in the result, but does not improve the quality of the code itself. The function could still potentially cause an error.

However, if you provide a detailed manual stub that accurately emulates the behavior of the function, the division could be any color, including red.

### **Emulating Function Behavior with Manual Stubs**

You can improve both the speed and selectivity of your verification by providing manual stubs that accurately 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 the effect that the code might have on the remainder of the system.

### **Example**

This example shows a header for a missing function (which may occur when the verified code is an incomplete subset of a project).

```
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 will propagate more precision throughout the application, which will result in a higher selectivity rate (more proven colors, i.e. more red+ green + gray). For this function, you could just add a simple body:

```
procedure a_missing_function
  (dest: in out integer,
   src  : in integer)
begin
  dest := src;
end;
```



In this case, it is obvious that instead of considering the full range for the `dest` parameter, PolySpace will consider the relation between input parameter `src` and the output parameter, propagating more precision throughout the application. See the same example in the section of this guide titled “Manual vs. Automatic Stubbing” on page 5-2.

## Reducing Orange Checks with Empty Stubs

Providing empty manual stubs can reduce the number of orange checks in your results, but it does not make your code more reliable.

For example, consider the following code:

```
void write_or_not1(int *x);

void write_or_not2(int *x);
{ //empty manual stub
}

void orange(void)
{
    int x = 12;
    int y;

    write_or_not1(&x);
    y = y / x;    //Orange ZDV due to automatic stub
}

void green(void)
{
    int x = 12;
    int y;

    write_or_not2(&x);
    y = y / x;    // Green due to empty stub
}
```

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 the actual function does not result in a runtime error.

### **Applying Constraints to Variables Using Stubs**

Another way to increase the selectivity is to indicate to the PolySpace software that some variables (detailed below) might vary between some functional ranges instead of the full range of the considered type.

This 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 will stub it on the assumption that it can potentially take any value from the full type of an integer.

Given that PolySpace models data ranges throughout the code it verifies, it will obviously produce 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 three possible approaches are presented here.

with volatile and assert	with assert and without volatile	without assert, without volatile, without "if"
<pre>function stub return INTEGER is tmp: INTEGER; random: INTEGER; pragma volatile (random); begin tmp:= random; pragma assert (tmp&gt;=1); pragma assert (tmp&lt;=10); return tmp; end;</pre>	<pre>function random return INTEGER; pragma Interface (C, random); function stub return INTEGER is tmp: INTEGER; begin tmp:= random; pragma assert (tmp&gt;=1); pragma assert (tmp&lt;=10); return tmp; end;</pre>	<pre>function random return INTEGER; pragma Interface (C, random); function stub return INTEGER is tmp: INTEGER; begin tmp:= random; while (tmp&lt;1 or tmp&gt;10) loop tmp:=random; end loop; return tmp; end;</pre>

There is no particular advantage in using one approach or another (except, perhaps, that the assertions in the first two will usually generate orange checks) – it is largely down to personal preference.

## Describing Multitasking Behavior Properly

The asynchronous characteristics of your application can have a direct impact on the number of orange checks. Properly describing characteristics such as implicit task declarations, mutual exclusion, and critical sections can reduce the number of orange checks in your results.

For example, consider a variable  $X$ , and two concurrent tasks T1 and T2.

- $X$  is initialized to 0.
- T1 assigns the value 12 to  $X$ .
- T2 divides a local variable by  $X$ .
- A division by zero error is possible because T1 can be started before or after T2, so the division causes an orange ZDV.

The verification cannot determine if an error will occur without knowing the call sequence. Modelling the task differently could turn this orange check green or red.

Refer to “*Preparing Multitasking Code*” for information on tasking facilities, including:

- Shared variable protection:
  - Critical sections,
  - Mutual exclusion,
  - Tasks synchronization,
- Tasking:
  - Threads, interruptions,
  - Synchronous/asynchronous events,
  - Real-time OS.

## Reviewing Orange Checks

In this section...
“Overview: Reviewing Orange Checks” on page 9-23
“Defining Your Review Methodology” on page 9-23
“Performing Selective Orange Review” on page 9-24
“Importing Review Comments from Previous Verifications” on page 9-27
“Performing an Exhaustive Orange Review” on page 9-28

### Overview: Reviewing Orange Checks

After you define a process that matches your quality objectives, you do not have too many orange checks. You have the correct number of orange checks for your quality model.

At this point, the goal is not to eliminate orange checks, it is to work efficiently with them.

Working efficiently with orange checks involves:

- Defining a review methodology to work consistently with orange checks
- Reviewing orange checks efficiently
- Importing comments to avoid duplicating review effort

### Defining Your Review Methodology

Before reviewing verification results, you should configure a methodology for your project. The methodology defines both the type and number of orange checks you need to review to meet three criteria levels.

The screenshot shows a dialog box titled "Assistant configuration" with a sub-header "Number of checks to review". Below this, there is a table with three columns: "Criterion 1", "Criterion 2", and "Criterion 3". The rows represent different methodology criteria: ZDV, NIVL, S-OVFL, COR, POW, NIV, F-OVFL, and ASRT. Each row contains values for the three criteria.

	Criterion 1	Criterion 2	Criterion 3
Common			
ZDV	10	20	ALL
NIVL	AUTO	50	ALL
S-OVFL	AUTO	50	ALL
COR	AUTO	10	10
POW	AUTO	10	ALL
NIV	AUTO	5	10
F-OVFL	5	10	20
ASRT	AUTO	5	20

### Sample Review Methodology

The criteria levels displayed in the methodology represent quality levels you defined as part of the quality objectives for your project.

---

**Note** For information on setting the quality levels for your project, see Chapter 2.

---

After you configure a methodology, each developer uses it to review verification results. This ensures that all users apply the same standards when reviewing orange checks in each stage of the development cycle.

For more information on defining a methodology, see “Selecting the Methodology and Criterion Level” on page 8-22.

### Performing Selective Orange Review

Once you have defined a methodology for your project, you can use assistant mode to perform a *selective orange review*.

The number and type of orange checks you review is determined by your methodology and the quality level you are trying to achieve. As a project

progresses, the quality level (and number of orange checks to review) generally increases.

For example, you may perform a level 1 review in the early stages of development, when trying to improve the quality of freshly written code. Later, you may perform a level 2 review as part of unit testing.

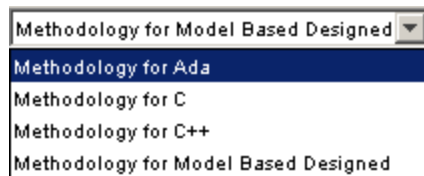
In general, the goal of a selective orange review is to find the maximum number of bugs in a short period of time. Many orange checks take only a few seconds to understand. Therefore, to maximize the number of bugs you can identify, you should focus on those checks you can understand quickly, spending no more than 5 minutes on each check. Checks that take longer to understand are left for later analysis.

To perform a selective orange review:

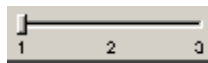
- 1 Click the **Assistant** button in the Viewer to select assistant mode.


The Viewer window toolbar displays the assistant mode controls.

- 2 Select the methodology for your project from the methodology menu.



- 3 Select the appropriate quality level for your review using the level slider.



- 4 Navigate through the checks by clicking the forward arrow .

- 5 Perform a quick code review on each orange check, spending no more than 5 minutes on each.

Your goal is to quickly identify whether the orange check is a:

- **potential bug** – code which will fail under some circumstances.
- **inconclusive check** – a check that requires additional information to resolve, such as the call sequence.
- **data set issue** – a theoretical set of data that cannot actually occur.

See “Sources of Orange Checks” on page 9-6 for more information on each of these causes.

---

**Note** If an orange check is too complicated to explain quickly, it may be an inconclusive check caused by complex code structure, or the result of basic imprecision (approximation of the data set used for verification). These types of checks often take a substantial amount of time to understand.

---

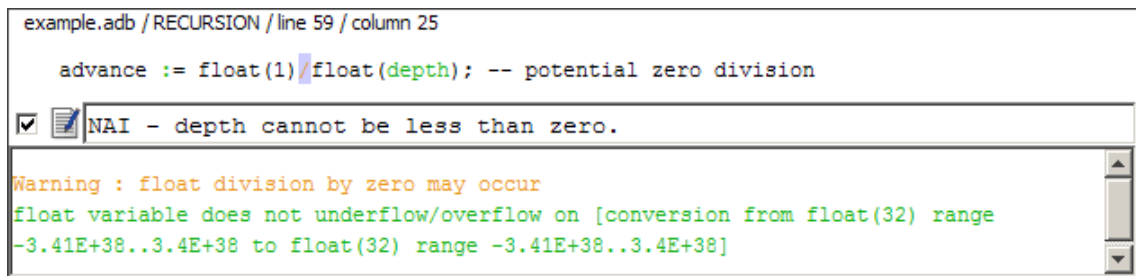
- 6 If you cannot identify a cause within 5 minutes, move on to the next check.

---

**Note** Your goal is to find the maximum number of bugs in a short period of time. Therefore, you want to identify the source of as many orange checks as possible, while leaving more complex situations for future analysis.

---

- 7 Once you understand the cause of an orange check, select the check box to indicate that you have reviewed the check.



The screenshot shows a code editor window with the following content:

```
example.adb / RECURSION / line 59 / column 25
advance := float(1)/float(depth); -- potential zero division
```

Below the code, there is a checkbox that is checked, followed by the text: `NAI - depth cannot be less than zero.`

Below the checkbox, there is a warning message in orange text: `Warning : float division by zero may occur`. Below the warning, there is a green text message: `float variable does not underflow/overflow on [conversion from float(32) range -3.41E+38..3.4E+38 to float(32) range -3.41E+38..3.4E+38]`.

- 8 Enter a comment for the reviewed check in the text box, indicating the results of your review.

For example, you can use acronyms to classify the checks being reviewed:



- **FNO** – Bug to be Fixed NOW
- **FNR** – Bug to be Fixed in Next Release
- **MQI** – Minor Quality Issue.
- **RBI** – RoBustness Issue
- **DFC** – DeFensive Code
- **NAI** – Not An Issue

**9** Continue to click the forward arrow until you have reviewed all of the checks identified by the assistant.

**10** Select **File > Save checks and comments** to save your review comments.


## Importing Review Comments from Previous Verifications

Once you have reviewed verification results for a module and saved your comments, you can import those comments into subsequent verifications of the same module, allowing you to avoid reviewing the same check twice.

To import review comments from a previous verification:

- 1** Open your most recent verification results in the Viewer.
- 2** Select **File > Import checks and comments**.
- 3** Navigate to the folder containing your previous results.
- 4** Select the results (.RTE) file, then click **Open**.

The review comments from the previous results are imported into the current results.

Once you import checks and comments, the **go to next check**  icon in assistant mode will skip any reviewed checks, allowing you to review only checks that you have not reviewed previously. If you want to view reviewed checks, click the **go to next reviewed check** icon.

---

**Note** If the code has changed since the previous verification, the imported comments may not be applicable to your current results. For example, the justification for an orange check may no longer be relevant to the current code.

---

### **Performing an Exhaustive Orange Review**

Up to 80% of orange checks can be resolved using multiple iterations of the process described in “Performing Selective Orange Review” on page 9-24. However, for extremely critical applications, you may want to resolve all orange checks. Exhaustive orange review is the process for resolving the remaining orange checks.

An exhaustive orange review is generally conducted later in the development process, during the unit testing or integration testing phase. The purpose of an exhaustive orange review is to analyze any orange checks that were not resolved during previous selective orange reviews, to identify potential bugs in those orange checks.

You must balance the time and cost of performing an exhaustive orange review against the potential cost of leaving a bug in the code. Depending on your quality objectives, you may or may not want to perform an exhaustive orange review.

### **Cost of Exhaustive Orange Review**

During an exhaustive orange review, each orange check takes an average of 5 minutes to review. This means that 400 orange checks require about four days of code review, and 3,000 orange checks require about 25 days.

However, if you have already completed several iterations of selective orange review, the remaining orange checks are likely to be more complex than average, increasing the average time required to resolve them.

### **Exhaustive Orange Review Methodology**

Performing an exhaustive orange review involves reviewing each orange check individually. As with selective orange review, your goal is to identify whether the orange check is a:

- **potential bug** – code which will fail under some circumstances.
- **inconclusive check** – a check that requires additional information to resolve, such as the call sequence.
- **data set issue** – a theoretical set of data that cannot actually occur.
- **Basic imprecision** – checks caused by imprecise approximation of the data set used for verification.

---

**Note** See “Sources of Orange Checks” on page 9-6 for more information on each of these causes.

---

Although you must review each check individually, there are some general guidelines to follow.

- 1** Start your review with the modules that have the highest selectivity in your application.

If the verification finds only one or two orange checks in a module or function, these checks are probably not caused by either inconclusive verification or basic imprecision. Therefore, it is more likely that these orange checks contain actual bugs. In general, these types of orange checks can also be resolved more quickly.

- 2** Next, examine files that contain a large percentage of orange checks compared to the rest of the application. These files may highlight design issues.

Often, when you examine modules containing the most orange checks, those checks will prove inconclusive. If the verification is unable to draw a conclusion, it often means the code is very complex, which can mean low robustness and quality. See “Inconclusive Verification” on page 9-6.

- 3** For all files you review, spend the first 10 minutes identifying checks that you can quickly categorize (such as potential bugs and data set issues), similar to what you do in a selective orange review.

Even after performing a selective orange review, a significant number of checks can be resolved quickly. These checks are more likely than average to reflect actual bugs.

#### 4 Spend the next 40 minutes of each hour tracking more complex bugs.

If an orange check is too complicated to explain quickly, it may be an inconclusive check caused by complex code structure, or the result of basic imprecision (approximation of the data set used for verification). These types of checks often take a substantial amount of time to understand. See “Sources of Orange Checks” on page 9-6.

#### 5 Depending on the results of your review, correct the code or comment it to identify the source of the orange check.

### Inconclusive Verification and Code Complexity

The most interesting type of inconclusive check occurs when verification reveals that the code is too complicated. In these cases, most orange checks in a file are related, and careful analysis identifies a single cause — perhaps a function or a variable modified many times. These situations often focus on functions or variables that have caused problems earlier in the development cycle.

For example, consider a variable *Computed\_Speed*.

- *Computed\_Speed* is first copied into a signed integer (between  $-2^{31}$  and  $2^{31}-1$ ).
- *Computed\_Speed* is then copied into an unsigned integer (between 0 and  $2^{31}-1$ ).
- *Computed\_Speed* is next copied into a signed integer again.
- Finally, *Computed\_Speed* is added to another variable.

The verification reports 20 orange overflows (OVFL).

This scenario does not cause a real bug, but the development team may know that this variable caused trouble during development and earlier testing phases. PolySpace verification also identified a problem, suggesting that the code is poorly designed.

## **Resolving Orange Checks Caused by Basic Imprecision**

On rare occasions, a module may contain many orange checks caused by imprecise approximation of the data set used for verification. These checks are usually local to functions, so their impact on the project as a whole is limited.

In cases of basic imprecision, you may be able to resolve orange checks by increasing the precision level. If this does not resolve the orange check, however, verification cannot help directly.

In these cases, PolySpace software can only assist you through the call tree and dictionary. The code needs to be reviewed using alternate means. These alternate means may include:

- Additional unit tests
- Code review with the developer
- Checking an interpolation algorithm in a function
- Checking calibration data

For more information on basic imprecision, see “Basic Imprecision” on page 9-8.



# Day to Day Use

---

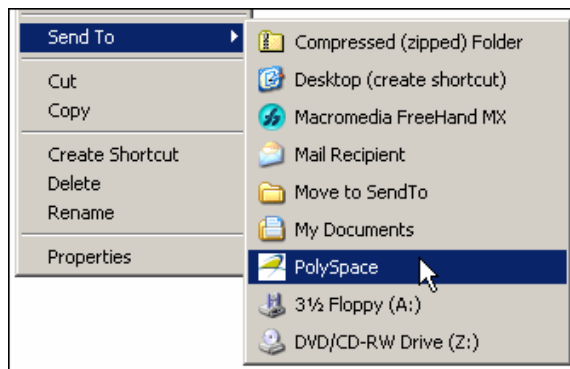
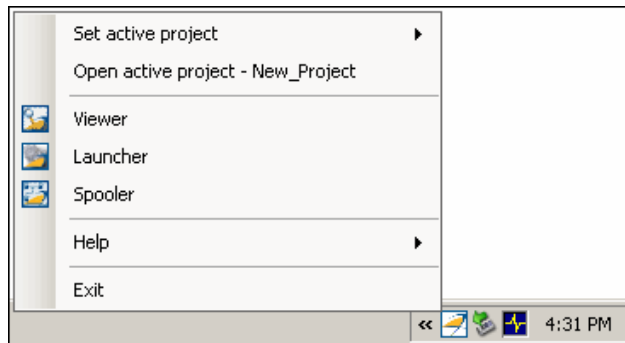
- “PolySpace In One Click Overview” on page 10-2
- “Using PolySpace In One Click” on page 10-3

## PolySpace In One Click Overview

Most developers verify the same files multiple times (writing new code, unit testing, integration), and usually need to run verifications on multiple project files using the same set of options. In a Microsoft Windows environment, PolySpace In One Click provides a convenient way to streamline your work when verifying several files using the same set of options.

Once you have set up a project file with the options you want, you designate that project as the *active project*, and then send the source files to PolySpace software for verification. You do not have to update the project with source file information.

On a Windows systems, the plug-in provides a PolySpace Toolbar in the Windows Taskbar, and a **Send To** option on the desktop pop-up menu:





## Using PolySpace In One Click

### In this section...

“PolySpace In One Click Workflow” on page 10-3

“Setting the Active Project” on page 10-3

“Launching Verification” on page 10-5

“Using the Taskbar Icon” on page 10-9

## PolySpace In One Click Workflow

Using PolySpace In One Click involves two steps:

- 1 Setting the active project.
- 2 Sending files to PolySpace software for verification.

## Setting the Active Project

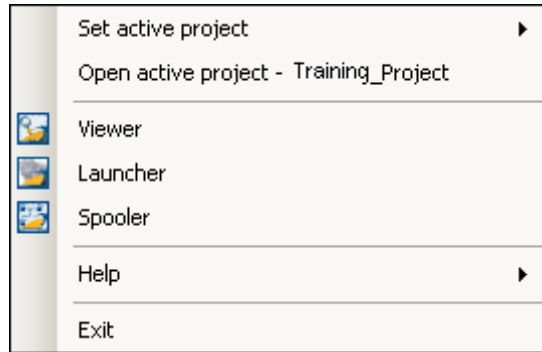
The active project is the project that PolySpace In One Click uses to verify the files that you select. Once you have set an active project, it remains active until you change the active project. PolySpace software uses the analysis options from the project; it does not use the source files or results directory from the project.

To set the active project:

- 1 Right-click the PolySpace In One Click icon in the taskbar area of your Windows desktop:

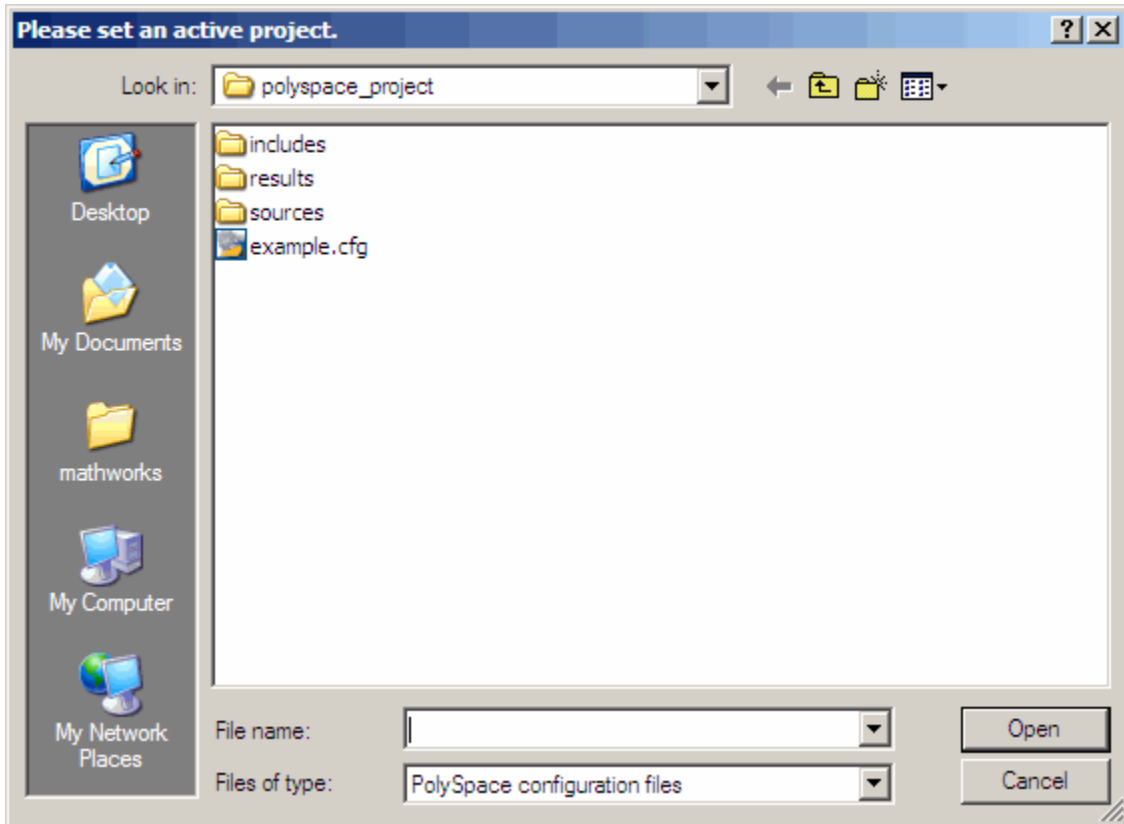


The context menu appears.



**2** Select **Set active project > Browse** from the menu.

The **Please set an active project** dialog box appears:



**3** Select the project you want to use as the active project.

**4** Click **Open** to apply the changes and close the dialog box.

## Launching Verification

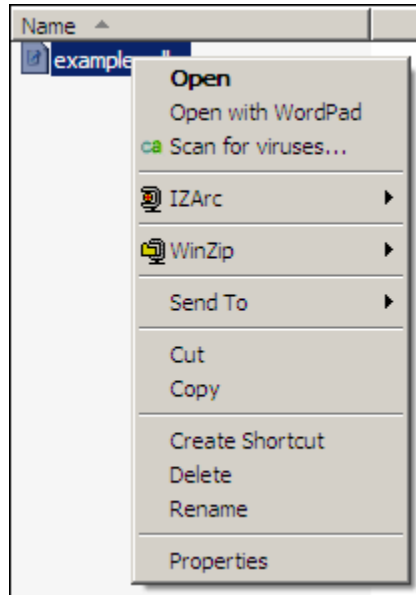
PolySpace in One Click allows you to send multiple files to PolySpace software for verification.

To send a file to PolySpace software for verification:

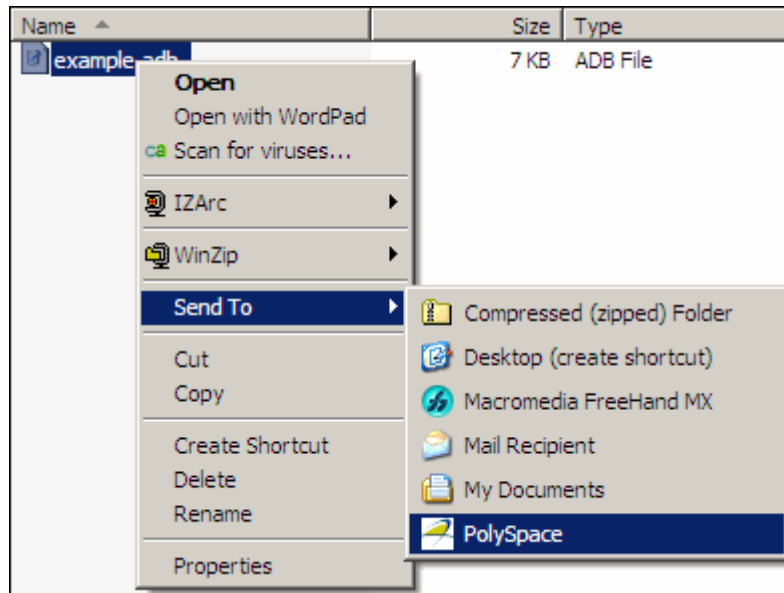
**1** Navigate to the directory containing the source files you want to verify.

**2** Right-click the file you want to verify.

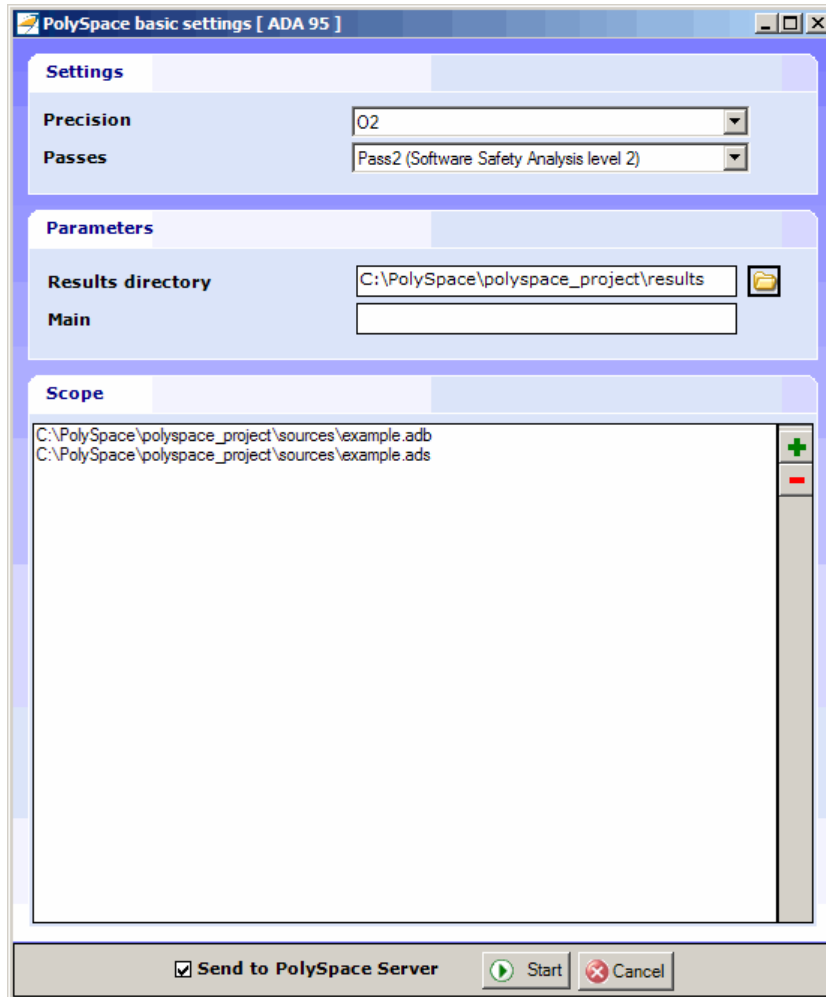
The context menu appears.



**3** Select **Send To > PolySpace**.



The **PolySpace basic settings** dialog box appears.



---

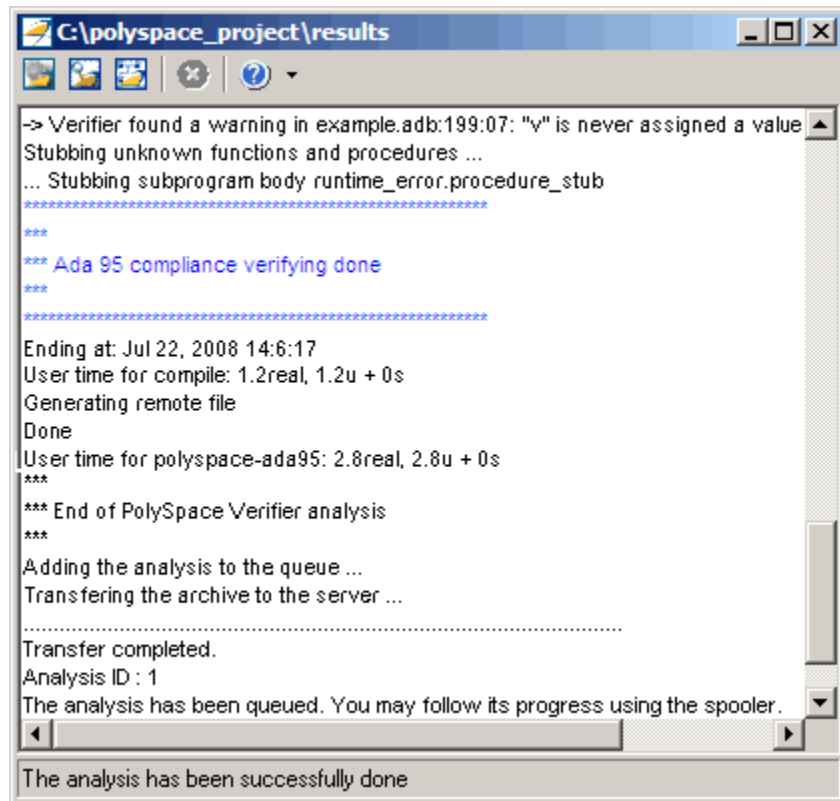
**Note** The options you specify the basic settings dialog box override any options set in the configuration file. These options are also preserved between verifications.

---

- 4 Enter the appropriate parameters for your verification.

- 5 Leave the default values for the other parameters.
- 6 Click **Start**.

The verification starts and the verification log appears.



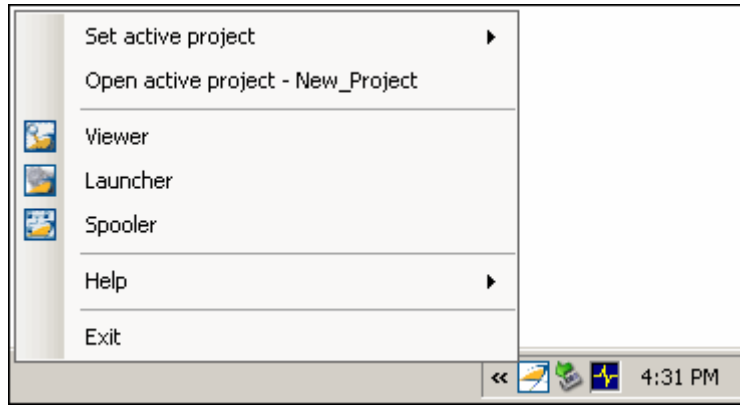
```

C:\polyspace_project\results
-> Verifier found a warning in example.adb:199:07: "v" is never assigned a value
Stubbing unknown functions and procedures ...
... Stubbing subprogram body runtime_error.procedure_stub
*****
***
*** Ada 95 compliance verifying done
***
*****
Ending at: Jul 22, 2008 14:6:17
User time for compile: 1.2real, 1.2u + 0s
Generating remote file
Done
User time for polyspace-ada95: 2.8real, 2.8u + 0s
***
*** End of PolySpace Verifier analysis
***
Adding the analysis to the queue ...
Transferring the archive to the server ...
.....
Transfer completed.
Analysis ID : 1
The analysis has been queued. You may follow its progress using the spooler.
The analysis has been successfully done

```

## Using the Taskbar Icon

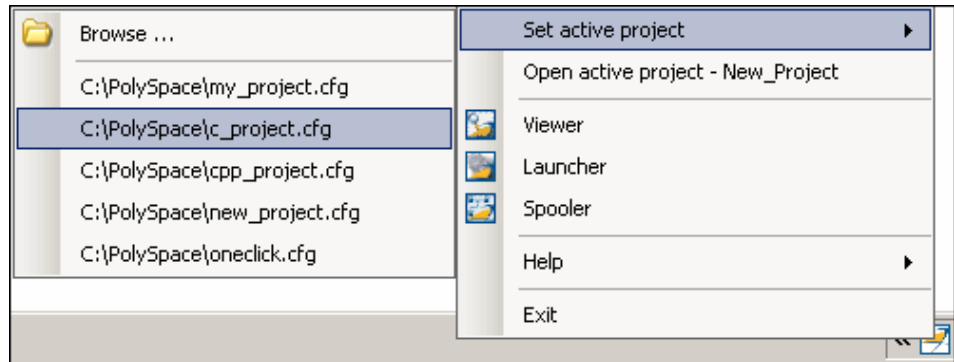
The PolySpace in One Click Taskbar icon allows you to access various software features.



Click the PolySpace Taskbar Icon, then select one of the following options:

- **Set active project** — Allows you to set the active configuration file. Before you start, you have to choose a PolySpace configuration file which contains the common options. You can choose a template of a previous project and move it to your working directory.

A standard file browser allows you to choose the configuration file. If you have multiple configuration files, you can quickly switch between them using the browse history.





---

**Note** No configuration file is selected by default. You can create an empty file with a .cfg extension.

---

- **Open active project** — Opens the active configuration file. This allows you to update the project using the standard PolySpace Launcher graphical interface. It allows you to specify all PolySpace common options, including directives of compilation, options, and paths of standard and specific headers. It does not affect the precision of a verification or the results directory.
- **Viewer** — Opens the PolySpace viewer. This allows you to review verification results in the standard graphical interface. In order to load results into the viewer, you must choose a verification to review in the Verification Log window.
- **Launcher** — Opens the PolySpace Launcher. This allows you to launch a verification using the standard PolySpace graphical interface.
- **Spooler** — Opens the PolySpace Spooler. If you selected a server verification in the “PolySpace Preferences” dialog box, the spooler allows you to follow the status of the verification.



**Atomic**

In computer programming, the adjective *atomic* describes a unitary action or object that is essentially indivisible, unchangeable, whole, and irreducible.

**Atomicity**

In a transaction involving two or more discrete pieces of information, either all of the pieces are committed or none are.

**Batch mode**

Execution of PolySpace from the command line rather than via the Launcher GUI.

**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 viewer.

**Code Verification**

The PolySpace process through which code is tested to reveal definite and potential runtime errors and a set of results is generated for review.

**Dead Code**

Code which is inaccessible at execution time under all circumstances due to the logic of the software executed prior to it.

**Development Process**

The process used within a company to progress through the software development lifecycle.

**Green check**

Code has been proven to be free of runtime errors.

**Gray check**

Unreachable code; dead code.

**Imprecision**

Approximations are made during a 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.



## A

- access sequence graph 8-32
- active project
  - definition 10-3
  - setting 10-3
- analysis options 3-16 3-19
- assistant mode
  - criterion 8-22
  - custom methodology 8-25
  - methodology 8-22
  - methodology for Ada 8-22 to 8-23
  - overview 8-21
  - reviewing checks 8-26
  - selection 8-21
  - use 8-21 8-26

## C

- call graph 8-31
- call tree view 8-13
- calling sequence 8-31
- cfg. *See* configuration file
- client 1-5 6-2
  - installation 1-6
  - verification on 6-21
- Client
  - overview 1-6
- coding review progress view 8-13 8-33
- color-coding of verification results 1-3 8-15
- compile
  - log 7-6
- compile log
  - Launcher 6-23
  - Spooler 6-6
- compile phase 6-3
- composite filters 8-39
- configuration file
  - definition 3-2
- contextual verification 2-5
- criteria

- quality 2-7
- custom methodology
  - definition 8-25

## D

- default directory
  - changing in preferences 3-6
- desktop file
  - definition 3-2
- directories
  - includes 3-10 3-13 3-15
  - results 3-10 3-13 3-15
  - sources 3-10 3-13 3-15
- downloading
  - results 8-8
  - results to UNIX or Linux clients 8-11
  - unit-by-unit verification results 8-12
- dsk. *See* desktop file

## E

- expert mode
  - filters 8-38
    - composite 8-39
    - individual 8-39
  - overview 8-29
  - selection 8-29
  - use 8-29

## F

- files
  - includes 3-10 3-13 3-15
  - results 3-10 3-13 3-15
  - source 3-10 3-13 3-15
- filters 8-38
  - alpha 8-39
  - beta 8-39
  - custom
    - modification 8-39 to 8-40

- use 8-39 to 8-40
- gamma 8-39
- individual 8-39
- user def 8-39

## G

- global variable graph 8-32

## H

- hardware requirements 7-2
- help
  - accessing 1-8

## I

- installation
  - PolySpace Client for Ada 1-6
  - PolySpace products 1-6
  - PolySpace Server for Ada 1-6

## L

- Launcher 1-5
  - monitoring verification progress 6-23
  - opening 3-3
  - starting verification on client 6-21
  - starting verification on server 6-3
  - viewing logs 6-23
  - window 3-3
    - overview 3-3
    - progress bar 6-23
- level
  - quality 2-7
- logs
  - compile
    - Launcher 6-23
    - Spooler 6-6
  - full
    - Launcher 6-23

- Spooler 6-6
- stats
  - Launcher 6-23
  - Spooler 6-6
- viewing
  - Launcher 6-23
  - Spooler 6-6

## M

- methodology for Ada 8-22 to 8-23

## O

- objectives
  - quality 2-5

## P

- PolySpace Client
  - overview 1-6
- PolySpace Client for Ada
  - installation 1-6
- PolySpace In One Click
  - active project 10-3
  - overview 10-2
  - sending files to PolySpace software 10-5
  - starting verification 10-5
  - use 10-2
- PolySpace products for Ada
  - components 1-5
  - installation 1-6
  - overview 1-2
  - user interface 1-5
- PolySpace products for C/C++
  - related products 1-6
- PolySpace Queue Manager Interface. *See* Spooler
- PolySpace Server
  - overview 1-6
- PolySpace Server for Ada
  - installation 1-6



- preferences
    - Launcher
      - default directory 3-6
      - default server mode 6-3
      - server detection 7-3
    - Viewer
      - assistant configuration 8-23
      - display columns in RTE view 8-35
  - procedural entities view 8-13
    - reviewed column 8-35
  - product overview 1-2
  - progress bar
    - Launcher window 6-23
  - project
    - creation 3-2
    - definition 3-2
    - directories
      - includes 3-3
      - results 3-3
      - sources 3-3
    - file types
      - configuration file 3-2
      - desktop file 3-2
    - saving 3-18
- Q**
- quality level 2-7
  - quality objectives 2-5 3-19
- R**
- related products 1-6
    - PolySpace products for linking to Models 1-7
    - PolySpace products for verifying C code 1-6
    - PolySpace products for verifying C++
      - code 1-6
  - reports
    - generation 8-45
  - results
    - directory 3-10 3-13 3-15
    - downloading from server 8-8
    - downloading to UNIX or Linux clients 8-11
    - opening 8-12
    - report generation 8-45
    - unit-by-unit 8-12
  - reviewed column 8-35
  - robustness verification 2-5
  - rte view. *See* procedural entities view
- S**
- selected check view 8-13
  - server 1-5 6-2
    - detection 7-3
    - information in preferences 7-3
    - installation 1-6 7-3
    - verification on 6-3
  - Server
    - overview 1-6
  - source code view 8-13
  - Spooler 1-5
    - monitoring verification progress 6-6
    - removing verification from queue 8-8
    - use 6-6
    - viewing log 6-6
- T**
- troubleshooting failed verification 7-2
- V**
- variables view 8-13
  - verification
    - Ada code 1-2
    - C code 1-6
    - C++ code 1-6
    - client 6-2
    - compile phase 6-3
    - contextual 2-5

- failed 7-2
- monitoring progress
  - Launcher 6-23
  - Spooler 6-6
- phases 6-3
- results
  - color-coding 1-3
  - opening 8-12
  - report generation 8-45
  - reviewing 8-8
- robustness 2-5
- running 6-2
- running on client 6-21
- running on server 6-3
- starting
  - from Launcher 6-2 to 6-3 6-21
  - from PolySpace In One Click 6-2 10-5
- stopping 6-24

- troubleshooting 7-2
- Viewer 1-5
  - modes
    - selection 8-17
  - opening 8-12
  - window
    - call tree view 8-13
    - coding review progress view 8-13
    - overview 8-13
    - procedural entities view 8-13
    - selected check view 8-13
    - source code view 8-13
    - variables view 8-13

## **W**

- workflow
  - setting quality objectives 2-5