PolySpace® Products for Ada 5

User's Guide

The MathWorks™

Accelerating the pace of engineering and science

How to Contact The MathWorks

www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support
suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.com	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

PolySpace® Products for Ada User's Guide

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)

Contents

1 Introduction to PolySpace Products

2 Choosing How to Use PolySpace Software

3 Setting Up a Verification Project

4 Emulating Your Runtime Environment

5 Preparing Source Code for Verification

6 Running a Verification

7 Troubleshooting Verification Problems

8 Reviewing Verification Results

9 Managing Orange Checks

10 Day to Day Use

Glossary

Index

1

Introduction to PolySpace Products

Introduction to PolySpace Products

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

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/polyspacemodelsl/`

`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 or 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`

2

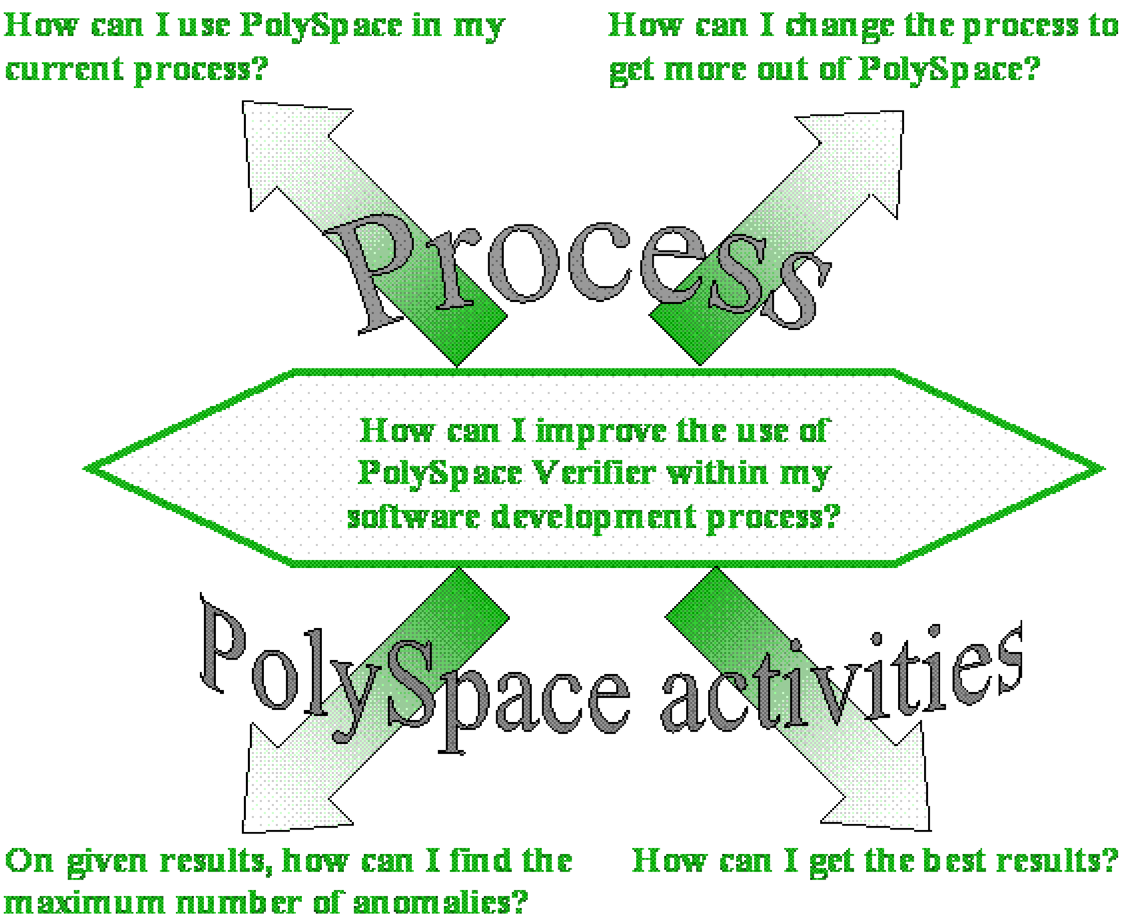
Choosing How to Use PolySpace Software

How to Use This Chapter

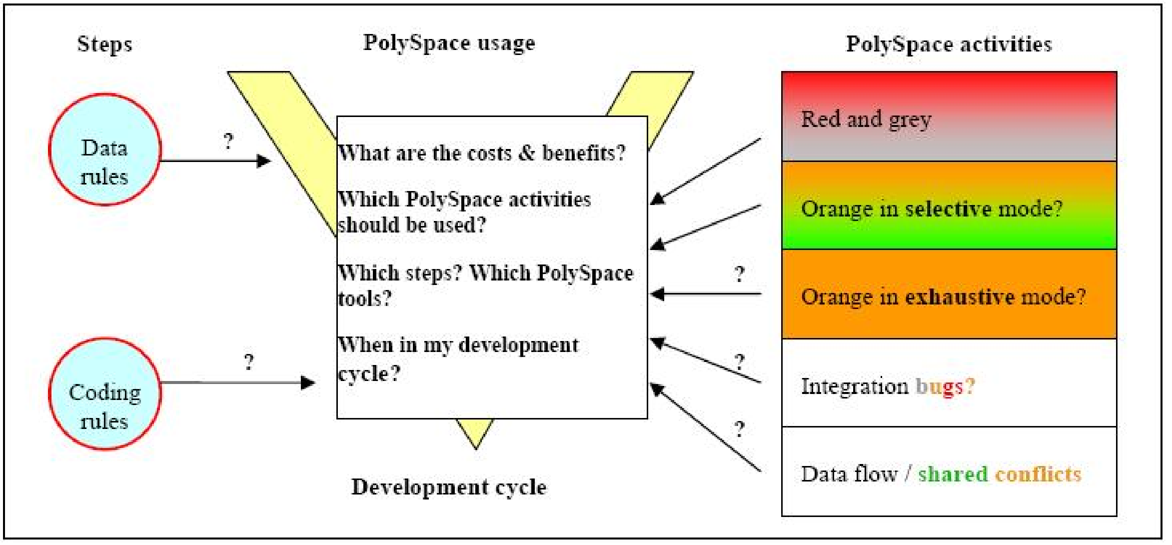
This chapter is designed for **Project managers, quality managers, and developers** who want to integrate PolySpace verification into their project development cycle. It explains how to apply PolySpace verification to each phase of the typical project lifecycle.

PolySpace verification supports both productivity and quality, but there is always a balance between these two goals. Generally, the criticality of your application determines your quality model — the balance between them.

This chapter assumes that your primary goal is to achieve maximum productivity with no quality defects. The document describes how to use PolySpace verification to achieve this goal at each phase of the development cycle. You must asses the costs of implementing each recommendation yourself, given your own quality model.



This guide suggests answers to the following questions:


Steps
PolySpace usage
PolySpace activities
Data rules
?
Coding rules
?
What are the costs & benefits?
Which PolySpace activities should be used?
Which steps? Which PolySpace tools?
When in my development cycle?
Development cycle
Red and grey
Orange in selective mode?
?
Orange in exhaustive mode?
?
Integration bugs?
?
Data flow / shared conflicts

Applying PolySpace Verification to Your Development Process

In this section...

Overview of the PolySpace Approach

PolySpace verification supports two objectives at the same time:

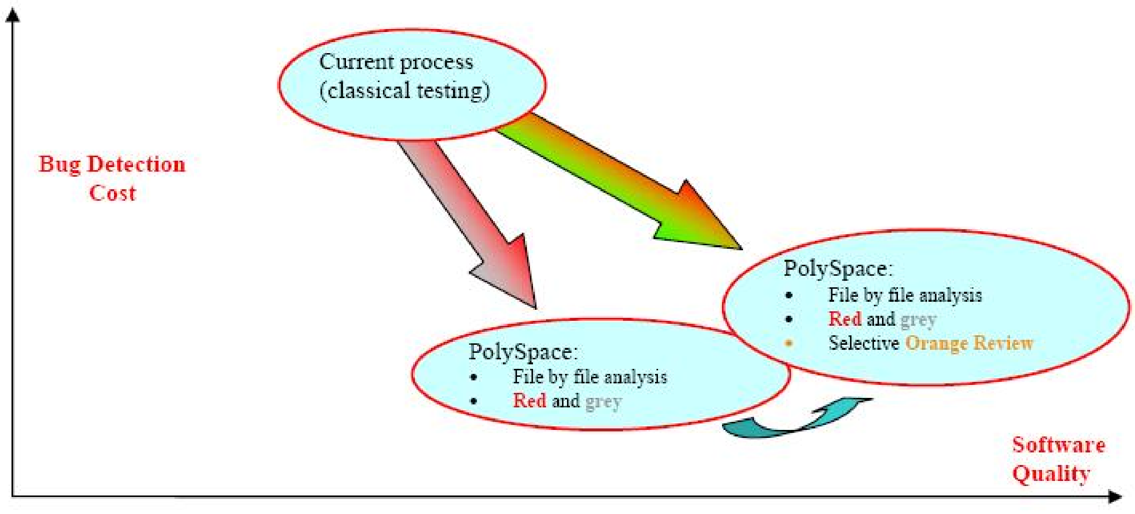
- Reducing the cost of testing and validation
- Improving software quality

You can use PolySpace verification in different ways depending on the your development context. The primary difference being how you exploit the results. The following diagrams summarize the different approaches.

Note This section does not attempt to compare the cost of certification processes, or of development processes with or without coding rules. The graphs compare the costs of typical processes with and without PolySpace software.

When No Coding Rules Are Adopted

During coding, there are two recommended approaches:

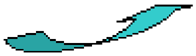


Note The sentence in previous figure about "file by file analysis" needs to be understood as a "package by package verification". Indeed, most of the time each package is developed in a file.

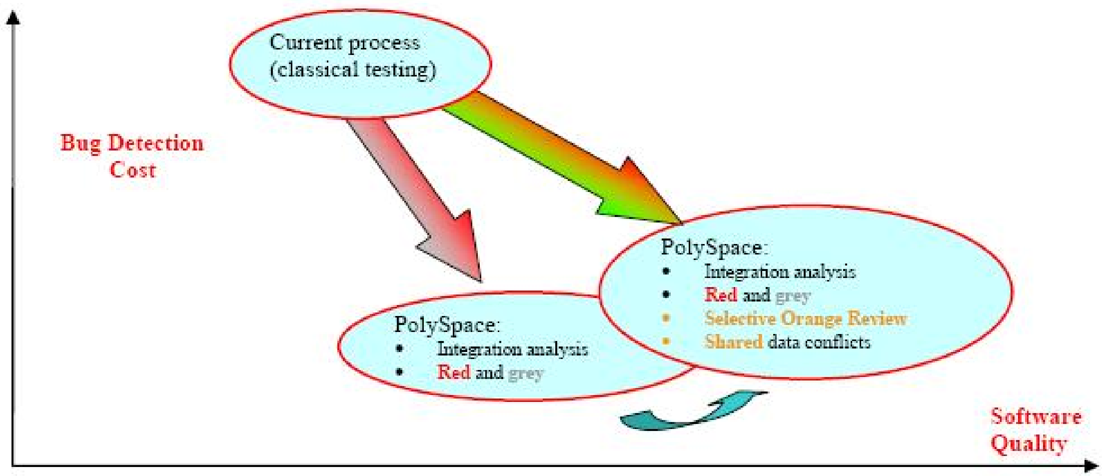
The first approach is to focus on **red** and **gray** results only — fix the red bugs, and check the dead code for abnormalities.

The second approach performs these activities, and adds a partial review of the orange warnings. The goal is to find as many bugs as possible in a limited amount of time. This approach finds more bugs, and therefore improves the overall quality of the software. It does involve more effort, but the amount of time spent to find each bug remains very small.

Note Using PolySpace verification on a single package is efficient. Even though the verification has no knowledge of the file context, experience shows that 50% of bugs detected by PolySpace verification can be found locally.

 This symbol is used to indicate that when a team has successfully implemented one approach, they can migrate to a more demanding (and more fruitful) one. This migration may not be desirable — it depends on the context of the project.

Then, **after coding**, before testing activity:



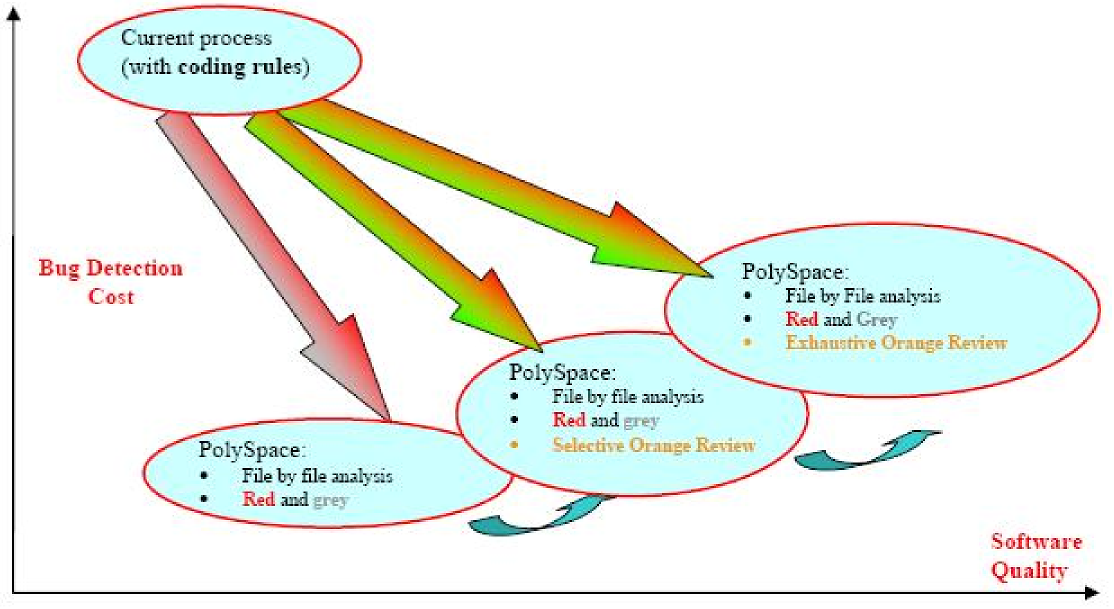
Again, the first approach is to use only the red and gray results — fix the red bugs, and check the dead code.

The second approach performs these activities, and adds a partial review of the orange warnings and of the orange shared data.

When Coding Rules Have Been Adopted

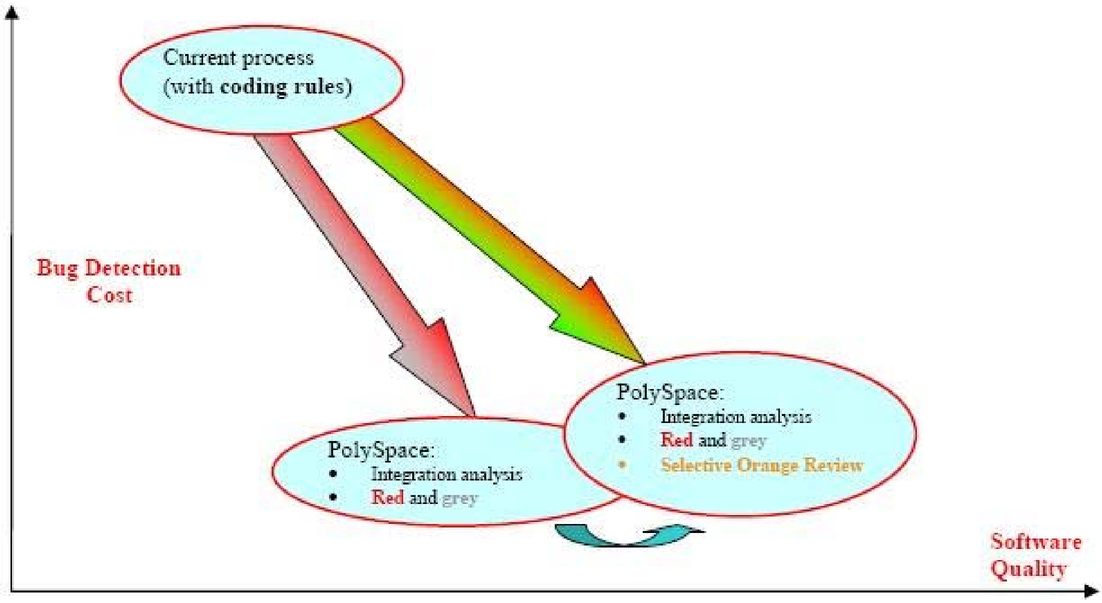
The main difference between this process and the previous process is in respect to the cost of bug detection. When PolySpace verification is used in conjunction with coding rules, the costs of bug detection are much lower.

During coding, there are three ways to use PolySpace verification:



Compared to the previous situation (without coding rules), there is an additional possibility. Instead of reviewing only certain orange warnings in a file, you can check all of them systematically. This is possible because when the **right coding rules** are respected, there are very few orange checks in a file. Therefore, checking all orange warnings can be very fruitful. A large proportion of those anomalies require some correction to the code, with some users reporting up to 50%.

Then, **after coding**, before the testing activity:



Note It is also possible to migrate from a selective to an exhaustive orange review when performing an integration verification, but this activity is very costly.

In a Certification Context

In a certification context“A Quality/Qualification Approach” on page 2-16, a “quality/qualification” approach where PolySpace verification replaces an existing activity. In this case quality is already high and maybe at a “zero defects” level, but PolySpace verification will reduce the cost of achieving such quality. In this context, PolySpace verification can replace the traditional time consuming control and data flow verification, as well as shared data conflict detection.

As an Acceptance Tool

The fourth and last approach implies the use of PolySpace verification as an acceptance tool, or as a method of meeting an acceptance criterion.

Standard Development Process

Overview

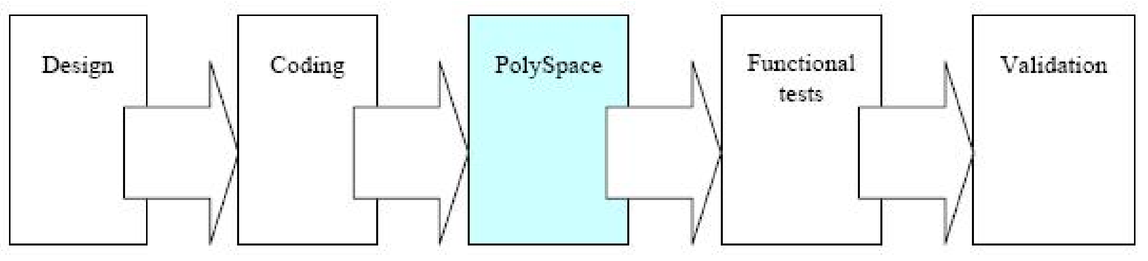
This approach is mainly for consideration by a project manager rather than a quality manager. It aims to improve productivity rather than to prove the quality of the application being analyzed.

The Software Development Process

This section describes how to introduce PolySpace verification to a standard software development process. For instance,

- In Ada, no unit test tools or coverage tools are used: functional tests are performed just after coding
- In C and C++, either no coding rules are present or they are not always followed.

The figure below illustrates the revised process, with PolySpace verification introduced in the tool chain. It will be used just before functional testing.



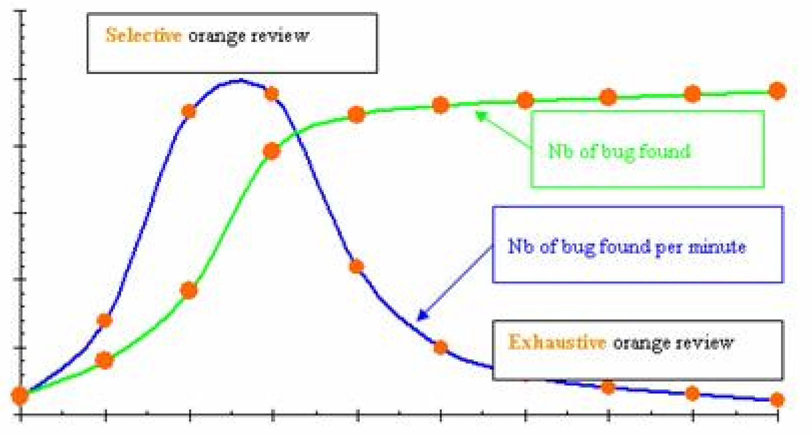
The Objective of Using PolySpace Verification

PolySpace verification will be used to improve the software quality and productivity. It will help the developer to find and fix bugs much quicker than the existing process. It will also improve the software quality by finding bugs which would otherwise be likely to remain in the software after delivery.

It does not prove the robustness of the code because the prime objective is to deliver code of at least similar quality to before, but to ensure that code is produced in a predictable time frame with controlled and minimized delay and costs. Another approach for this purpose is described in the next section.

The PolySpace Approach

The way forward here is for PolySpace products to be applied by developers or testers on a file-by-file (package-by-package) verification basis. The users will use the **default PolySpace options**, the most prominent feature of which is the automatically generated "main" function. This main will call all unused procedures and functions with full range parameters. The users will be required to fix **red** errors and examine **gray** code, and they will also do a selective orange review.



Cost/Benefits of a Selective Orange Review

This selective orange review can be applied on specific Runtime Error categories, such as "Out of Bound Array Index", or on all error categories. This depends on each individual developers coding style.

It is true that with this approach some bugs might remain in the unchecked oranges, but it represents a significant move forward from the initial position. Coding rules would help further if more improvement is sought.

A Complementary Approach

A second approach is also possible which, unlike the first, focuses only on an increase in quality. If coding rules are applied, this second approach will turn into a cheap and productive one as described by the second arrow on the illustration.

Integration tests are also possible at this stage. This verification will be performed by PolySpace software on larger modules, and the orange review will be focused on orange Runtime errors **which were not examined** after the file-by-file verification.

For instance, if the project construction is such that scalar overflows can only be reviewed at integration phase, then

- The user will ignore orange overflows with PolySpace Client when performing package-by-package verification.
- He will examine them with PolySpace Server.

Integration with Configuration Management Tools

PolySpace verification can also be used by project managers to establish and test for transition criteria to proceed to file check-in

- **Daily check-in** — PolySpace verification is applied to the file(s) currently under development. Compilation must complete without the permissive option.
- **Pre-unit test check-in** — PolySpace verification is applied to the file(s) currently under development.
- **Pre-integration test check-in** — PolySpace verification is applied to the whole project until compilation can complete without the permissive option. This stage will differ from the daily check-in activity because link errors will be highlighted here.
- **Pre-build for integration test check-in** — PolySpace verification is applied to the whole project, with all multitasking aspects accounted for as appropriate.
- **Pre-peer review check-in** — PolySpace verification is applied to the whole project, with all multitasking aspects accounted for as appropriate.

For each check-in activity mentioned above, the transition criterion could be: “No bug found within the allocated time defined by the process”. For instance, if the process defines that 20 minutes should be dedicated to a selective review, the criterion could be: “no bug found during these 20 minutes”.

Costs and Benefits

Using PolySpace to find **unit/local bugs** in this way will both reduce the cost of the software and improve the quality:

- Red checks and bugs in gray checks. The number of bugs found thanks to these colors can vary from one user to another, but experience shows that on average, around 40 percent of verifications will reveal one or more red errors and/or will reveal bugs in gray code.
- Orange checks. Experience suggests that the time needed to find one bug per file 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 or C++.

With this approach, using PolySpace verification to find **integration bugs** will increase the quality, but at a higher usage cost:

- **75% of bugs are local in this type of code** — the selective orange review at integration phase reveals a of integration bugs, and the rest () of local bugs. Finding real integration bugs might require another process which requires coding rules to be efficient.
- **Setup time —** the time needed to setup the verification can be higher due to a lack of coding rules. Code modifications might be needed. Most of these modifications cannot be automatic without changes in the process.
- **Anomalies and complexity —** In this configuration, any particular file will receive more orange checks (about twice as many). These oranges are likely to be anomalies, and will make the orange check review more time consuming.
- **An exhaustive orange review can take 25 man-days for a 50,000 line project —** This would represent the effort where the aspiration is for bug free software, assuming that a 50,000 line application contains about 3,000 orange checks

Rigorous Development Process: Introducing Tools and Coding Rules

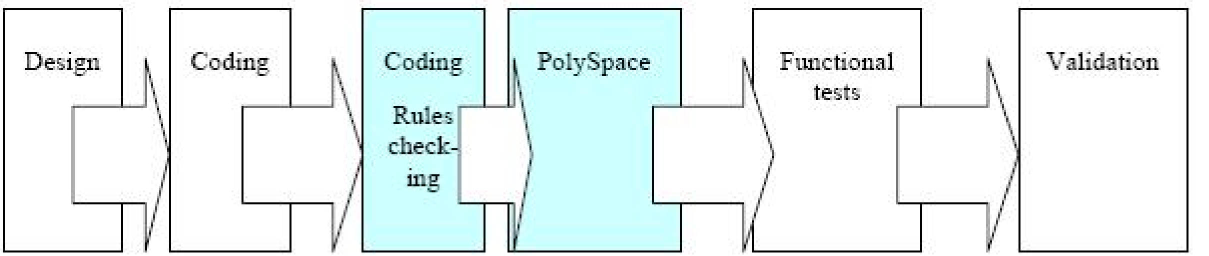
Overview

This is of interest for both project and quality managers, who are likely to be interested in this approach.

The Software Development Process

This section describes how to use PolySpace verification within a development process. In Ada, no unit test tools or coverage tools are used; instead, functional tests are performed just after coding.

The picture below describes the new process, with PolySpace verification introduced into the tool chain. It will be used just before functional testing.



PolySpace verification will be used to increase both the software quality and its productivity.

The PolySpace Approach

Use PolySpace on a file by file verification basis.

- The “main” used to analyze each file is very often **automatically generated by the project**, and not by PolySpace (unlike the standard approach).
- **Initialization ranges** should be applied to input data. For instance, 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 as part of the verification.

- [**Optional**] Some properties of output variables might be checked. For instance, 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.
- Red errors will be fixed and gray code examined, and an exhaustive orange review will be completed.
- The usage of permissive options is not advisable at this stage.

Note The distinguishing feature for this approach as compared with the standard approach is that the orange check review **is exhaustive here**.

A Complementary Approach

A second approach is also possible. Use PolySpace at integration phase 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, using PolySpace to find bugs will typically provide the following benefits

- 3-5 orange checks per file, 3 gray checks per file yielding an average of 1 bug per file. Typically, 2 of these oranges might represent the same bug, and another might represent an anomaly.
- An average of 2 verifications by PolySpace per file is typical before the file 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 which determine how the data flow are designed, the benefits might even be higher. The data rules would implicitly reduce the potential for PolySpace to find integration bugs.

With this approach, using PolySpace verification to find integration bugs might bring the following results. On a typical 50,000 line project:

- A selective orange check review might reveal **one integration bug per hour of orange** code review and takes about after 6 hours, which long enough to review the main orange points throughout the whole application. This represents a step towards an exhaustive orange check review. Spending more time is unlikely to be efficient, and wont guarantee that no bugs remain.
- An exhaustive orange review takes between 4 and 6 days, given that a 50,000 lines of code application might contain about 400-800 orange checks.

A Quality/Qualification Approach

Overview

Quality managers are likely to be interested in this approach.

The Software Development Process

This section describes how to use PolySpace verification within a process which includes coding and data rules. Such a process is typical of a *qualification* environment, with existing activities which must be performed. Before the introduction of PolySpace verification, they will have been performed by hand, with classical testing methods, or using previous generation tools. PolySpace verification will **replace these activities**, and reduce the cost of the process.

PolySpace verification is not intended to improve the quality which is already at the desired level. It will complete the same tasks more efficiently, bringing improved productivity.

The Objective of Using PolySpace Verification

PolySpace verification will be used to increase the productivity on existing activities, such as

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

The PolySpace Approach

- For data and control flow verification and shared data detection, PolySpace verification can be used on the whole application or on a subsection of the application.
- For robustness unit tests (as opposed to functional unit tests), PolySpace verification might be used in the same way as the method applied to the Rigorous development process.

Costs and Benefits

The replacement of these activities can lead to a significant cost reduction. For instance, the time spent on data and control flow verification can drop from 3 months to 2 weeks.

Quality will also become much more consistent since a much greater part of the process will be automated. PolySpace tools are equally efficient on a Friday afternoon and on a Tuesday morning!

Code Acceptance Criterion

Overview

This is likely to be of interest for a quality manager in a company which is outsourcing software development, and who wishes to impose acceptance criteria for the code.

The Software Development Process

This section describes how to define transition criteria for intermediate or final deliveries.

The Objective of Using PolySpace Verification

The objective is to control and evaluate the safety of an application. The means for doing so could vary from no red errors to exhaustive oranges review.

The PolySpace Approach

The example list of acceptance criteria below shows increasingly stringent tests, any or all of which may be adopted.

- No compilation errors
- No compilation warning errors
- No red code sections
- No unjustified gray code section
- A selective/exhaustive orange review according to the development process
  - 20% orange code sections reviewed or a time base threshold (described in the previous sections)
  - 100% orange code sections reviewed
- 20% concurrent access graph reviewed
- 100% concurrent access graph reviewed

Choosing the Type of Verification You Want to Perform

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

- **Robustness Verification** – Prove that the software works under all conditions, including "abnormal" conditions. This can be thought of as "worst case" analysis.

- **Contextual Verification** – Prove that the software works under normal working conditions. This can limit the amount of analysis that needs to be done by providing the software with the ranges of various parameters, so that the code only needs to be verified within these ranges.

By default, PolySpace software assumes you want to perform robustness verification (full range). However, this approach can lead to many orange checks in your results.

When performing contextual verification, you can use several PolySpace options to reduce the number of orange checks. You can use `pragma assert` in your code to limit verification to the data ranges imposed by the environment in which the software will run. You also can create a very detailed main generator.

3

Setting Up a Verification Project

Creating a Project

In this section...

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`

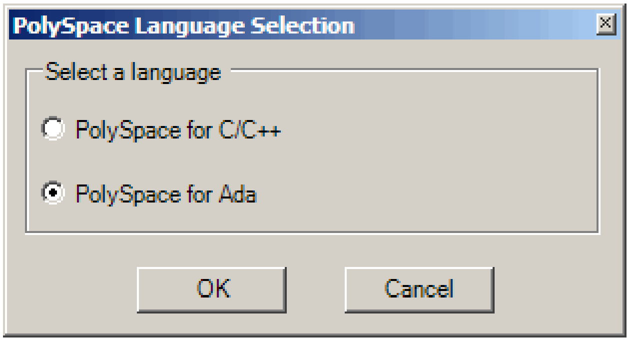
Opening PolySpace Launcher

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

To open the PolySpace Launcher:

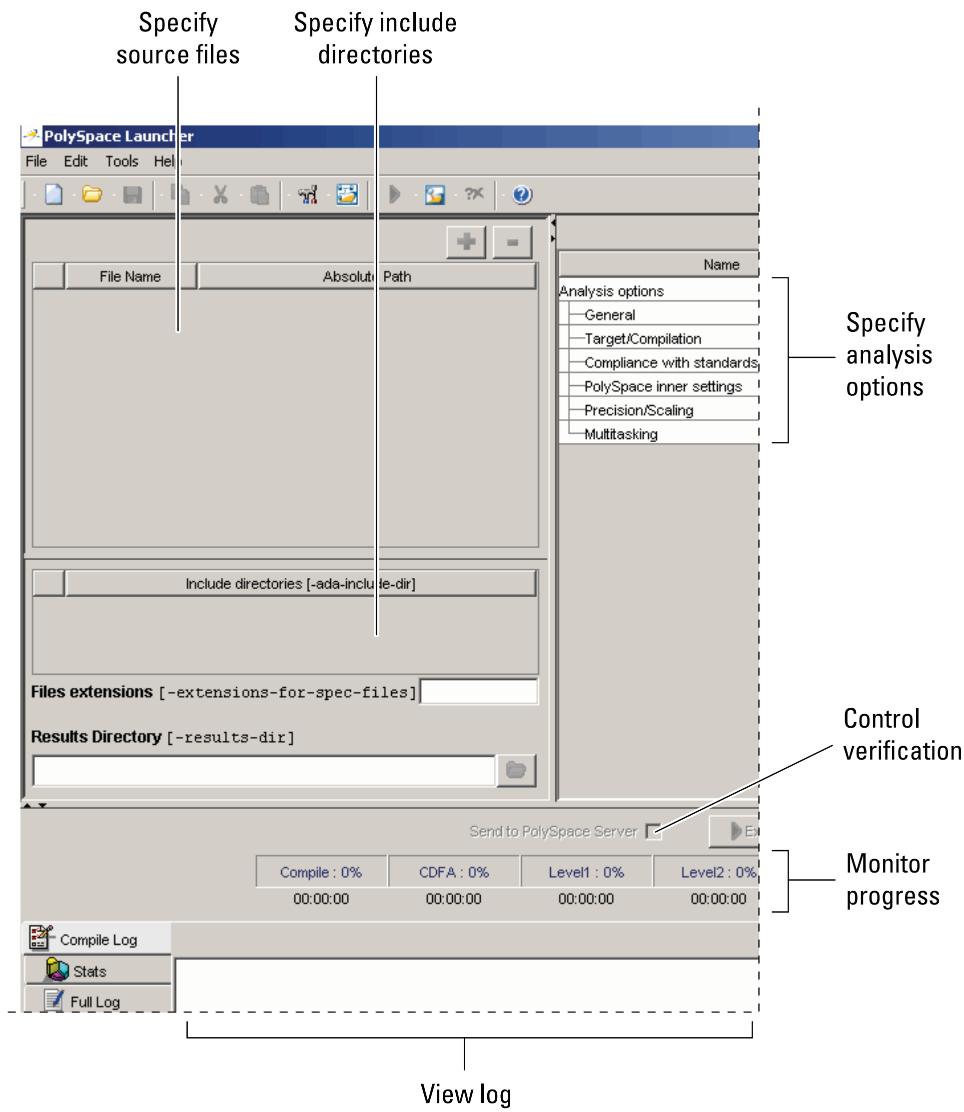
1 Double-click the PolySpace Launcher icon.

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 lick **OK**.

The PolySpace Launcher window appears:



The Launcher window has three main sections.

Use this section...	For...
Upper-left	Specifying: • Source files • Include directories • Results directory
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.

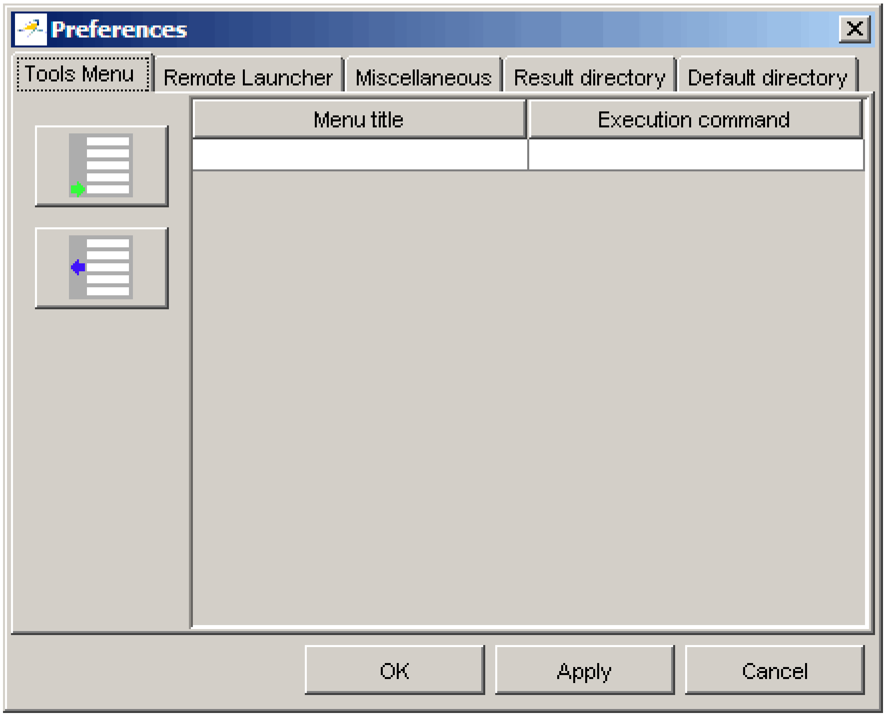
Specifying Default Directory

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. hanging the default directory to the project directory makes it easier for you to locate and specify source files and include directories in dialog boxes.

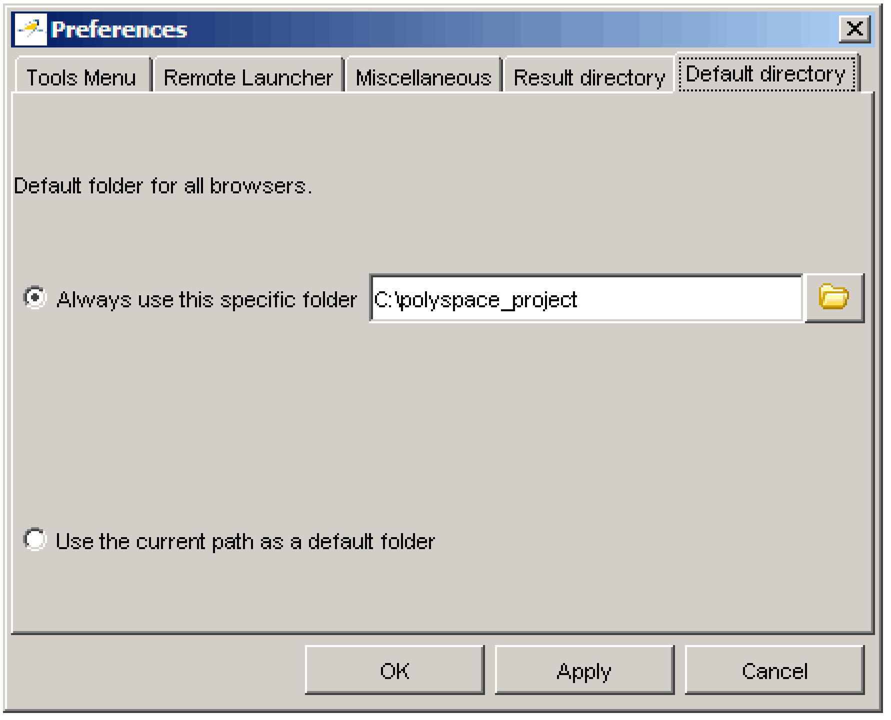
To change the default directory to the project directory:

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

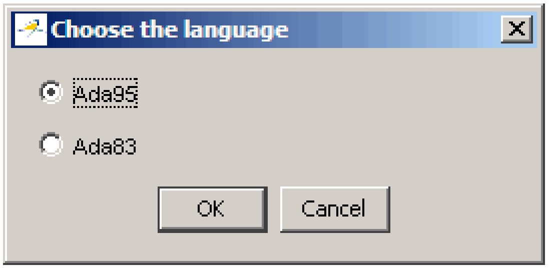
Creating New Projects

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

To create a new project:

1 Select **File > New Project**.

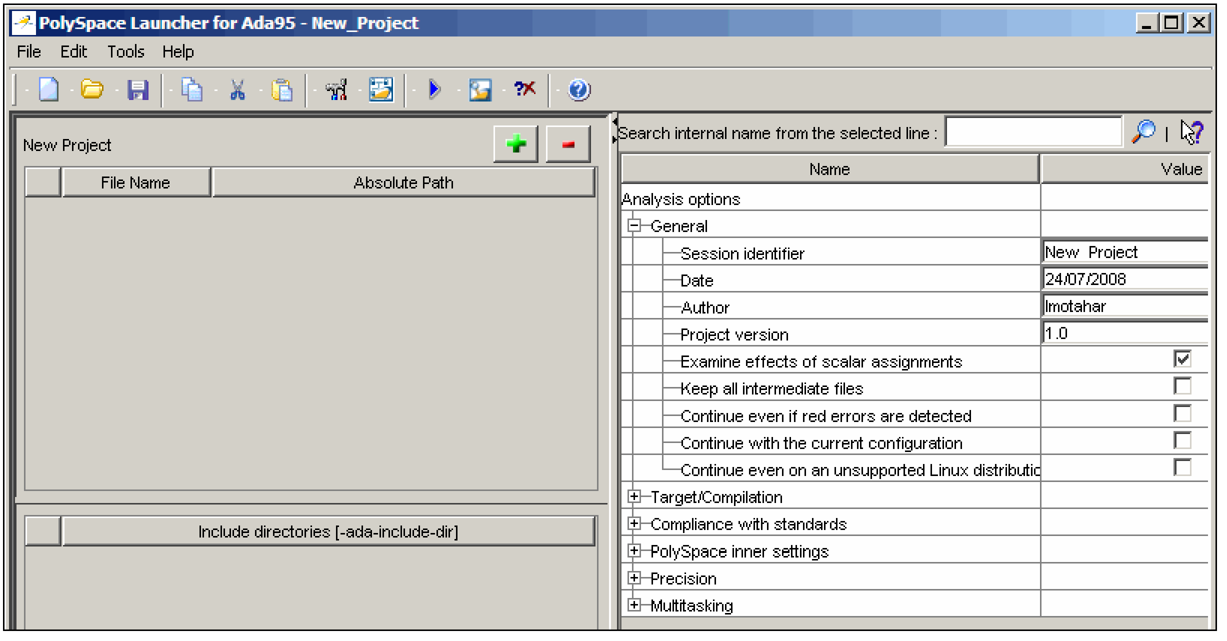
The **Choose the language** dialog box appears:



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

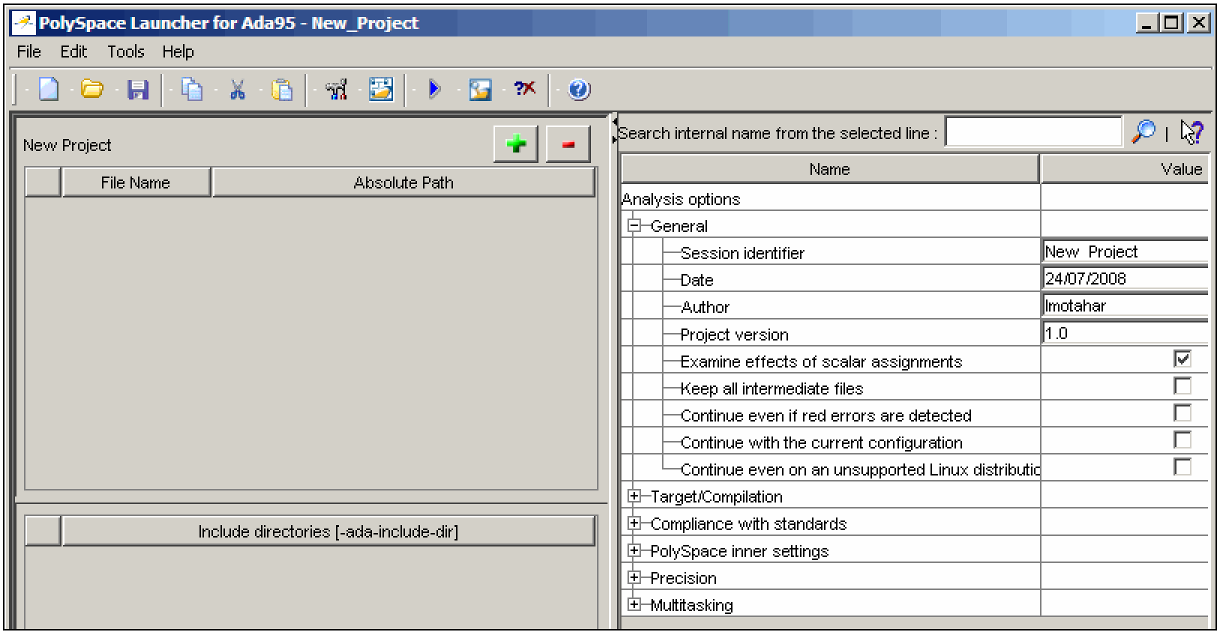
To open an existing project:

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.

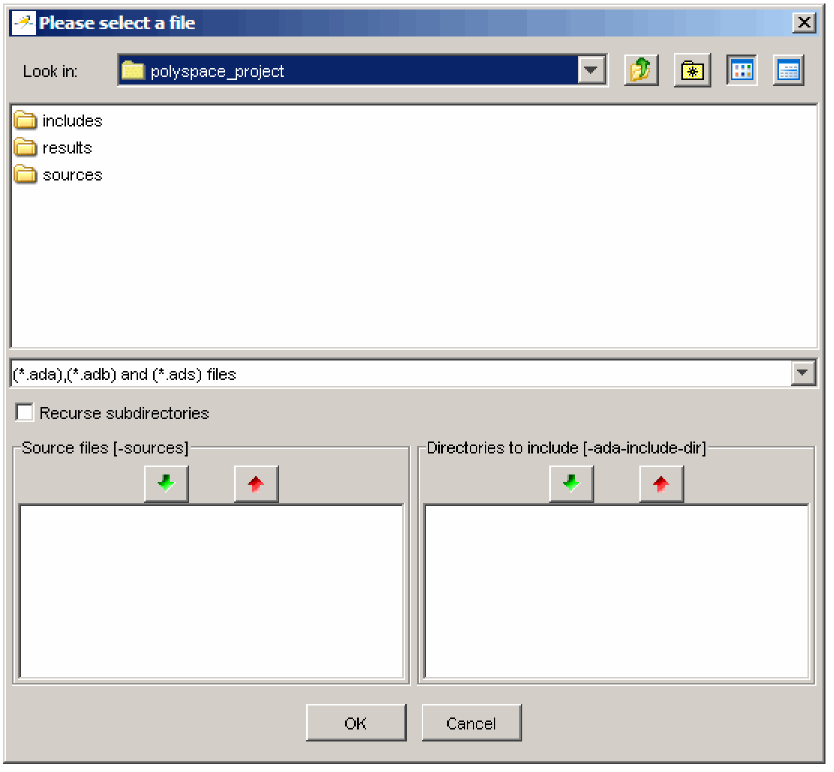


Specifying Source Files

To specify the source files for your project:

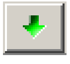
1 lick the green plus sign button in the upper right of the files section of the Launcher window.

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



2 In the **Look in** field, navigate to your project directory containing your source files.

3 Select the files you want to verify, then click the green down arrow button in the **Source files** section.



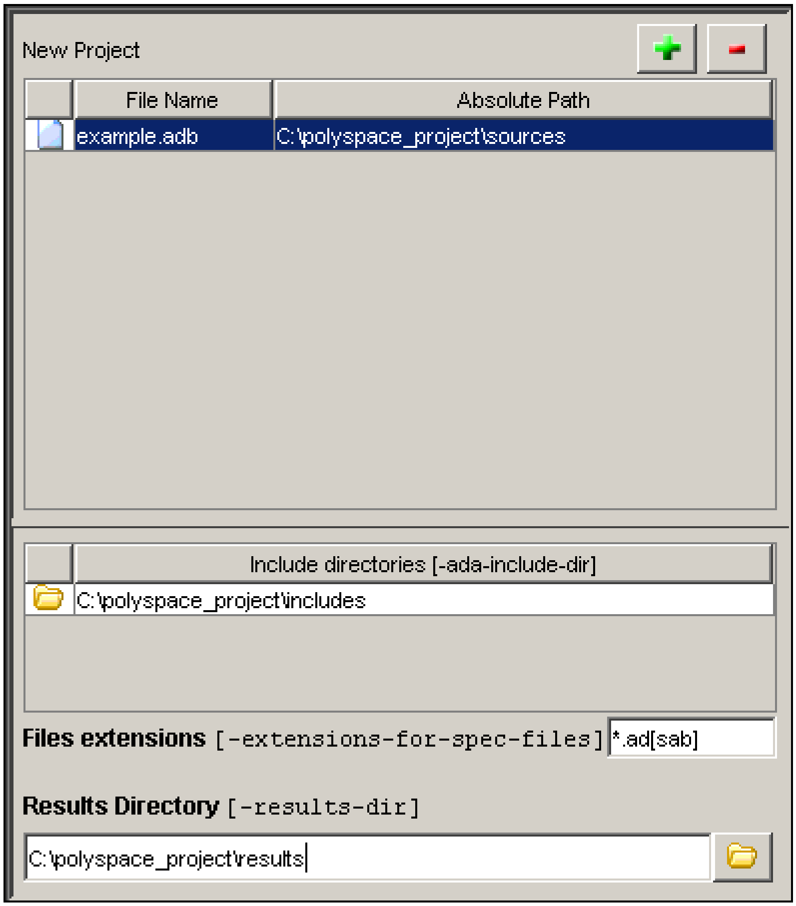
The path of each source files appear in the source files list.

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.

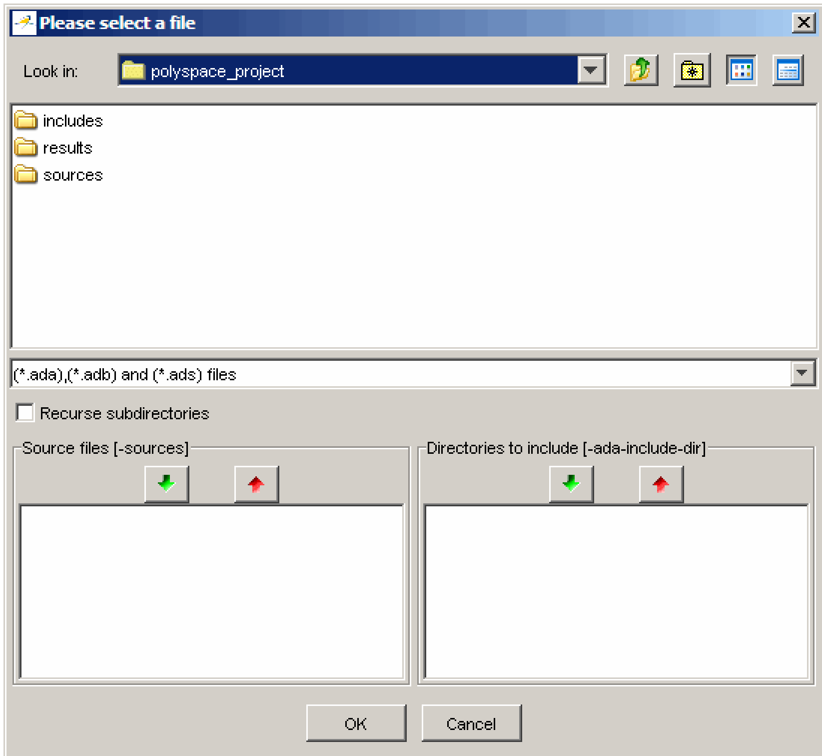


Specifying Include Directories

To specify the include directories for the project:

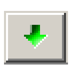
1 lick the green plus sign button in the upper right of the files section of the Launcher window.

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



2 In the **Look in** field, navigate to your project directory.

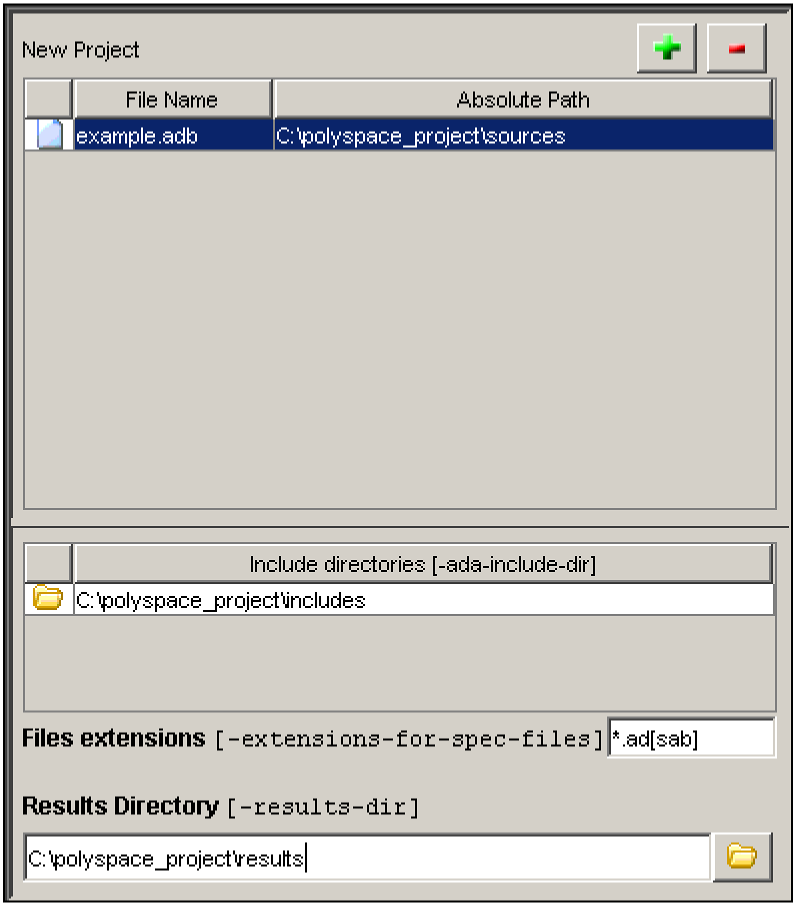
3 Select the directory containing the include files for your project, then click the green down arrow button in the **Directories to include** section.



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.

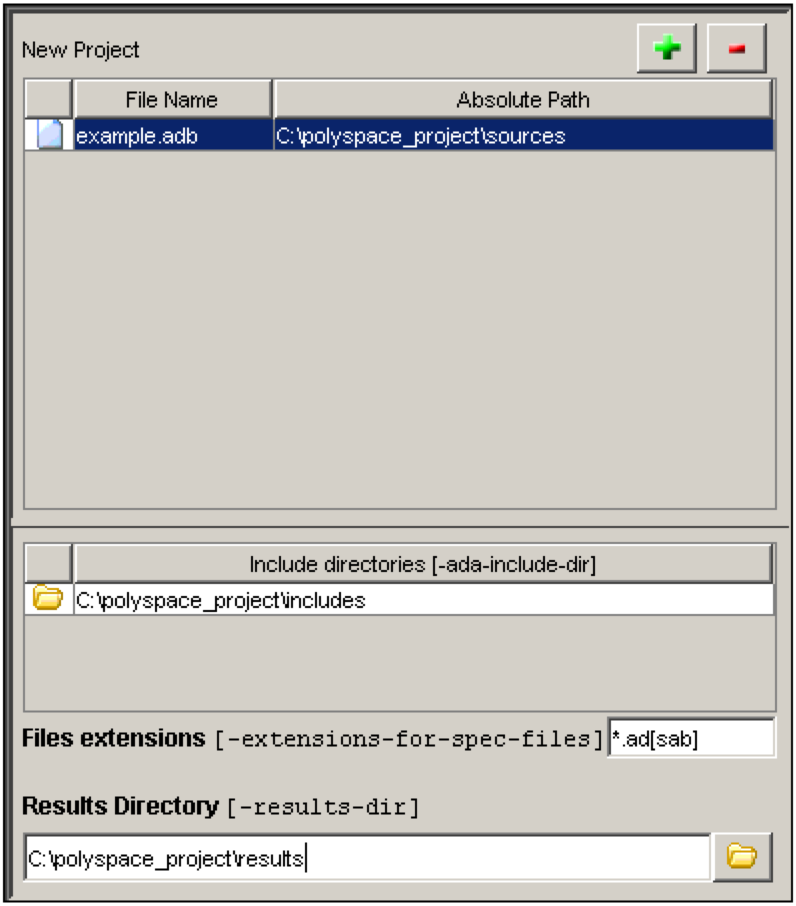


Specifying Results Directory

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:



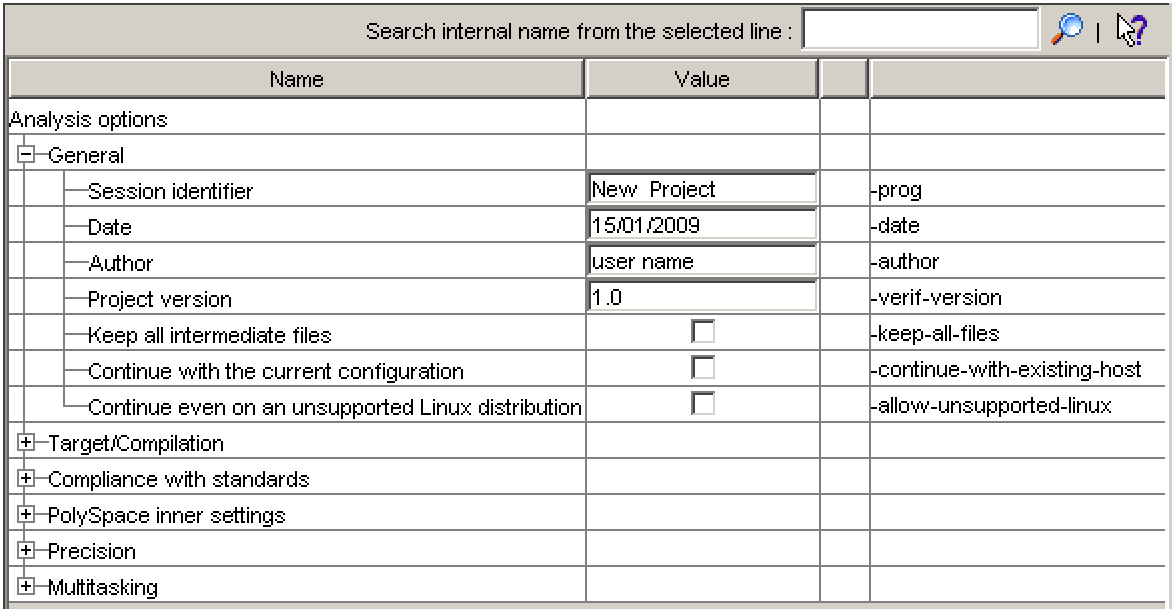
Specifying Analysis Options

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.



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 Viewer. Configuring text and XML editors in the Viewer allows you to view source files directly from the Viewer logs.

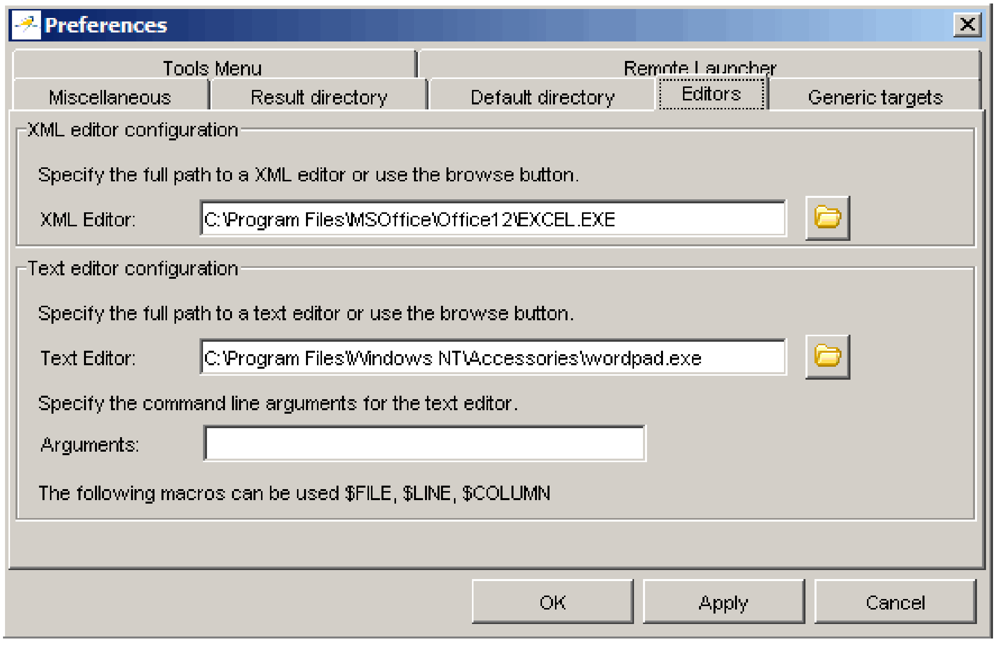
To configure your text and .XML editors:

1 Select **Edit > Preferences**.

The Preferences dialog box opens.

2 Select the **Editors** tab.

The Editors tab opens.



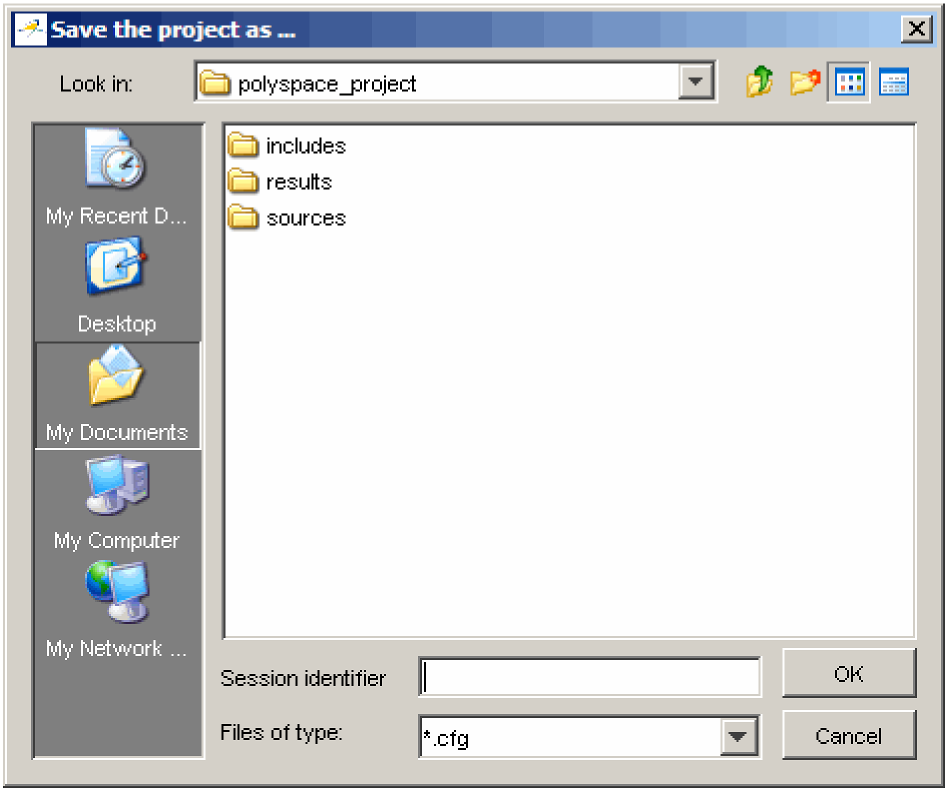
3 Specify a Text editor to use to view source files from the Viewer logs.

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.

Emulating Your Runtime Environment

Setting Up a Target

In this section...

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*.

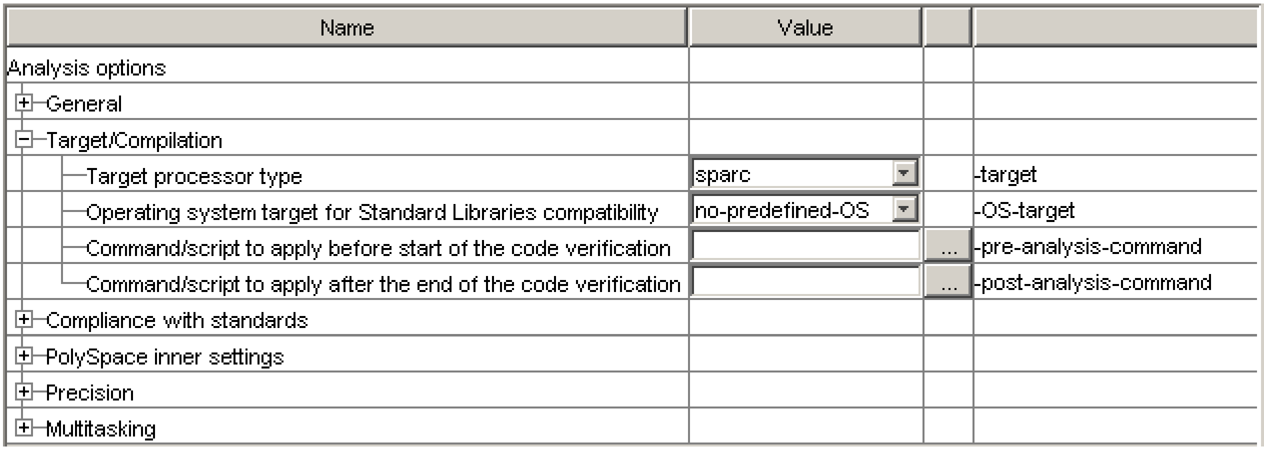
Specifying Target/Compilation Parameters

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.



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

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

In this section...

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

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 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

`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`
	`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`

package_first_task	other tasks

<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

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 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;
```

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

Mechanism	Description
Provide a **real mailbox** mechanism	• Can be very costly (time consuming) to implement • Can introduce errors in the stubs • Is too much effort compared with the solution below • Precise, but does not provide a much better precision than the upper approximation
Provide an **upper approximation of the mailbox**	in which each new read to the mailbox reads **one** of the recently posted messages, but not necessarily the last one. • Quick and easy to code • Gives precise results • See detailed implementation below

package mailboxes

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

package body mailboxes

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

procedure receive

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

task body task_1

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

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

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

Atomicity

Definitions

- *Atomic* — In computer programming, atomic describes a unitary action or object that is essentially indivisible, unchangeable, whole, and irreducible
- *Atomicity* — In a transaction involving two or more discrete pieces of information, either all 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.

6

Running a Verification

Types of Verification

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

Use...	For...
Server	• Best performance • Large files (more than 800 lines of code including comments) • Multitasking
Client	• An alternative to the server when the server is busy • Small files with no multitasking **Note** Verification on a client takes more time. You might not be able to use your client computer when a verification is running on it.

Running Verifications on PolySpace Server

In this section...

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 **Execute** 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 **Execute**.

   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:

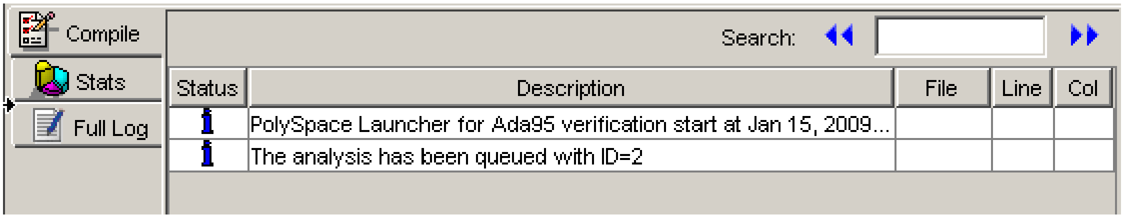
1 Checking syntax and semantics (the compile phase). Because PolySpace software is independent of any particular Ada compiler, it ensures that your code is portable, maintainable, and complies with ANSI® standards.

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 "-main-generator" in the "Options Description" chapter of the *PolySpace Client/Server for Ada User's Guide*.

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.



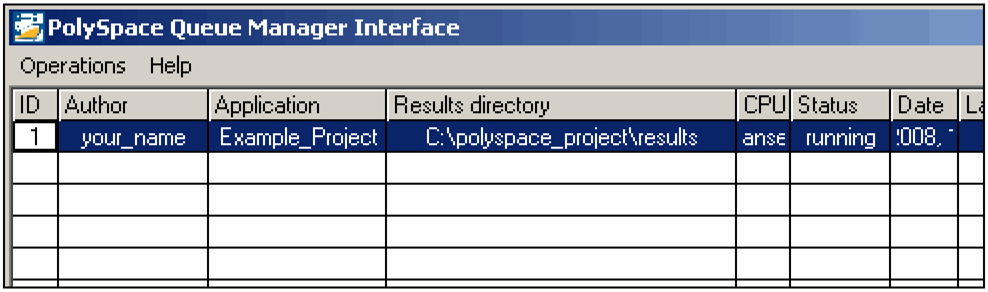
Managing Verification Jobs Using the PolySpace Queue Manager

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

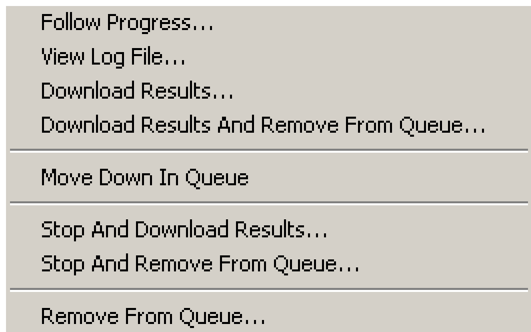
To manage verification jobs on the PolySpace Server:

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

The **PolySpace Queue Manager Interface** opens.



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



3 Select the appropriate option from the context menu.

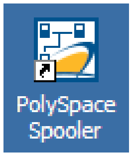
Tip You can also open the Polyspace Queue Manager Interface by clicking the PolySpace Queue Manager icon  in the PolySpace Launcher toolbar.

Monitoring Progress of Server Verification

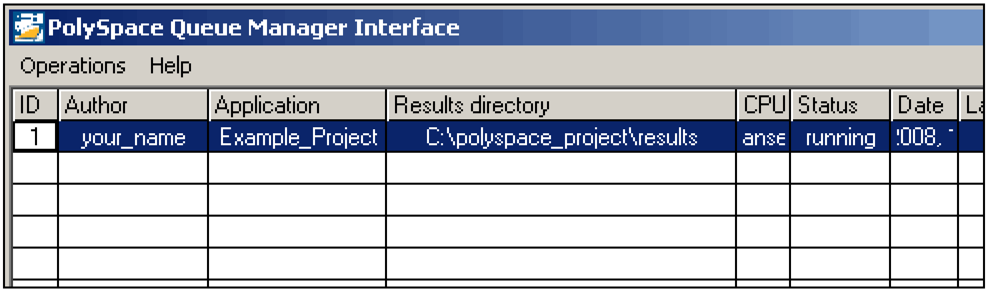
You can view the log file of a server verification using the PolySpace Queue Manager.

To view a log file on the server:

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

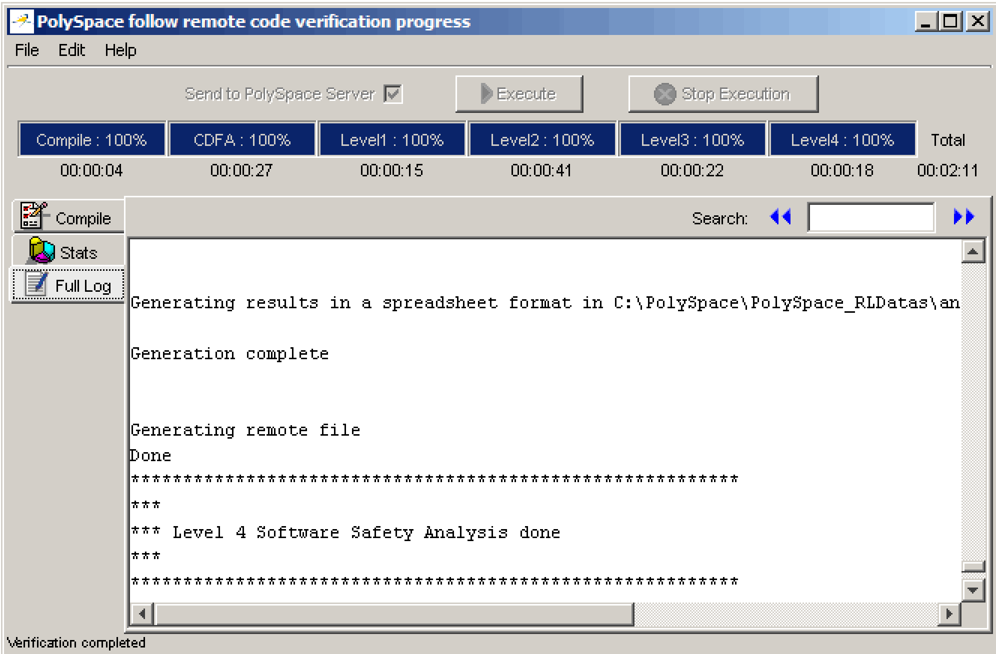


The **PolySpace Queue Manager Interface** opens.



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

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 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.

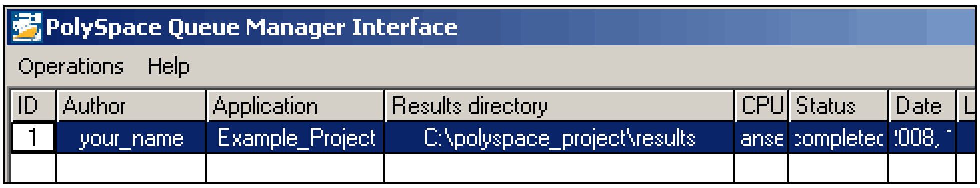
3 Click the **Compile Log** button to display compile phase messages and errors. You can search the log by entering search terms in the **Search in the log** box and clicking the left arrows to search backward or the right arrows to search forward. Click on any message in the log to get details about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



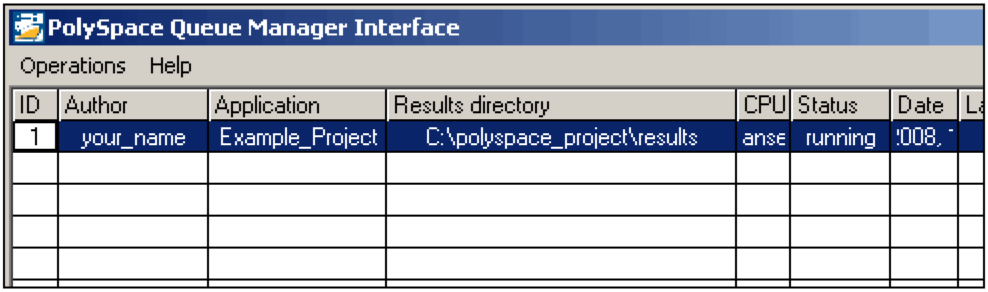
Viewing Verification Log File on Server

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

To view a log file on the server:

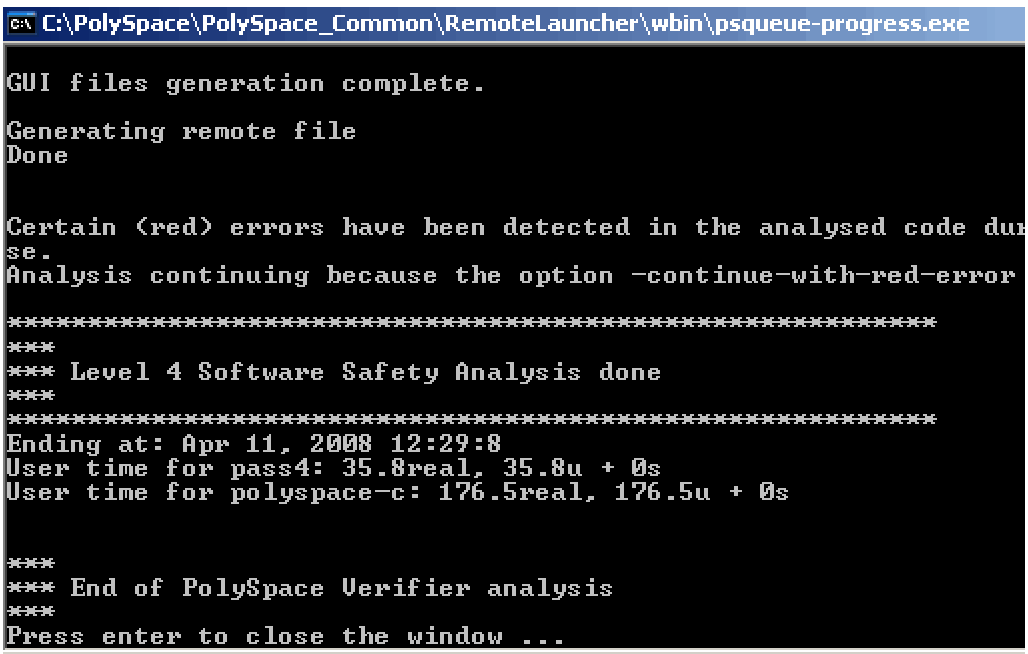
1 Double-click the **PolySpace Spooler** icon:

The **PolySpace Queue Manager Interface** opens.



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

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



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

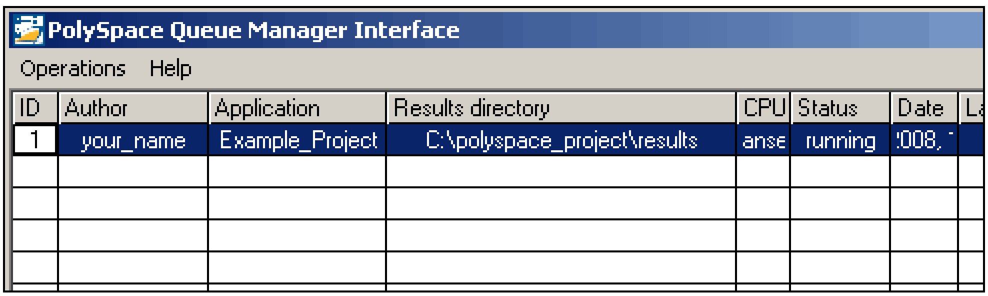
Stopping Server Verification Before It Completes

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.



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

- **Kill and download results** — Stops the verification immediately and downloads any preliminary results. The status of the verification changes from “running” to “aborted”. The verification remains in the queue.
- **Kill and remove from queue** — Stops the verification immediately and removes it from the queue.

Removing Verification Jobs from Server Before They Run

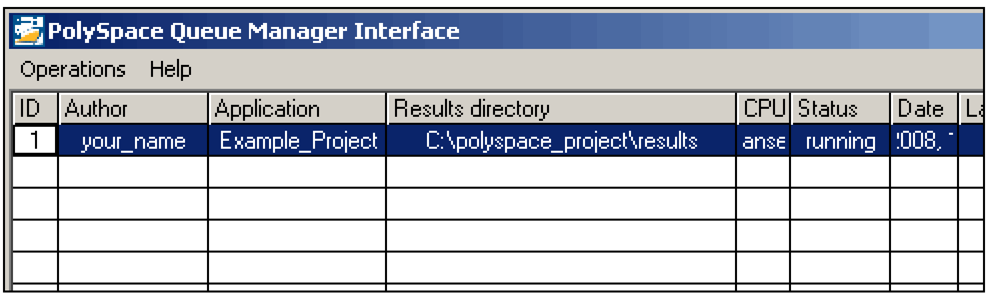
If your job is in the server queue, but has not yet started running, you can remove it from the queue using the PolySpace Queue Manager.

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 6-11). Once you have aborted a verification, you can remove it from the queue.

To remove a job from the server queue:

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

The **PolySpace Queue Manager Interface** opens.



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

The job is removed from the queue.

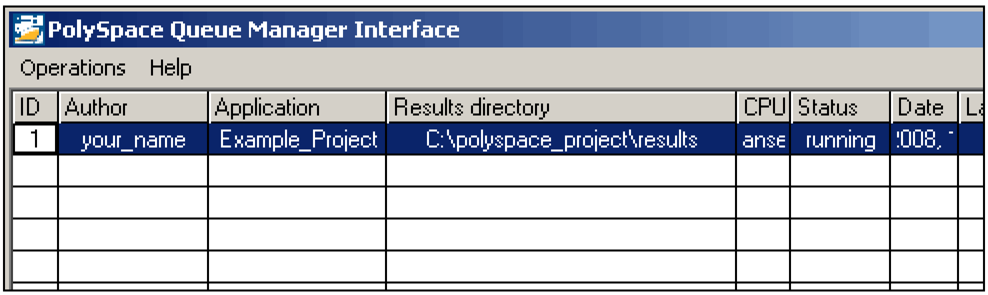
Changing Order of Verification Jobs in Server Queue

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

To move a job within the server queue:

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

The **PolySpace Queue Manager Interface** opens.



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.

Purging Server Queue

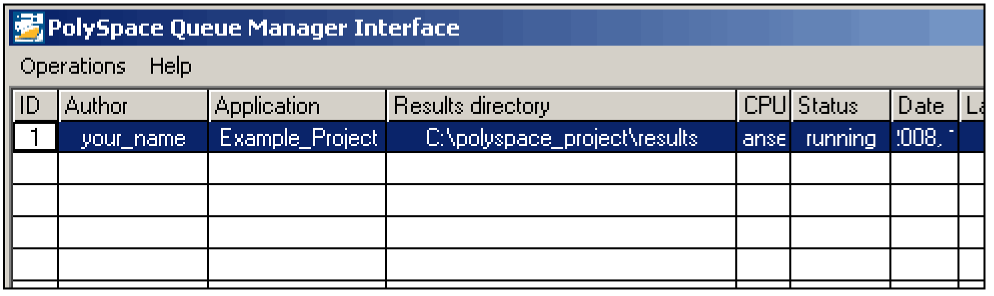
You can purge the server queue of all jobs, or completed and aborted jobs using the using the PolySpace Queue Manager.

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

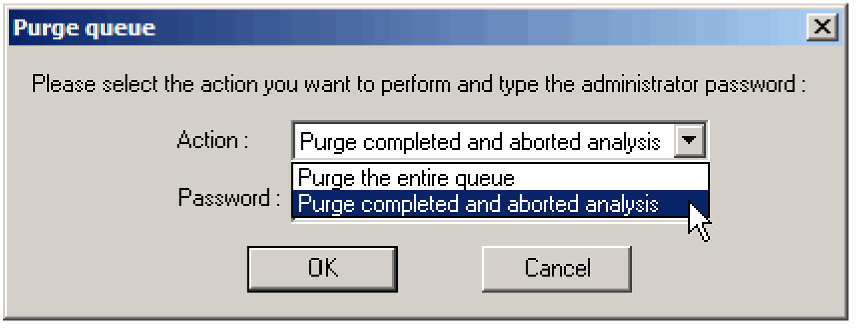
To purge the server queue:

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

The **PolySpace Queue Manager Interface** opens.



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



3 Select one of the following options:

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

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.

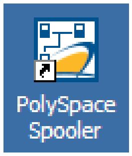
If analysis-keys.txt File is Lost or Corrupted

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.

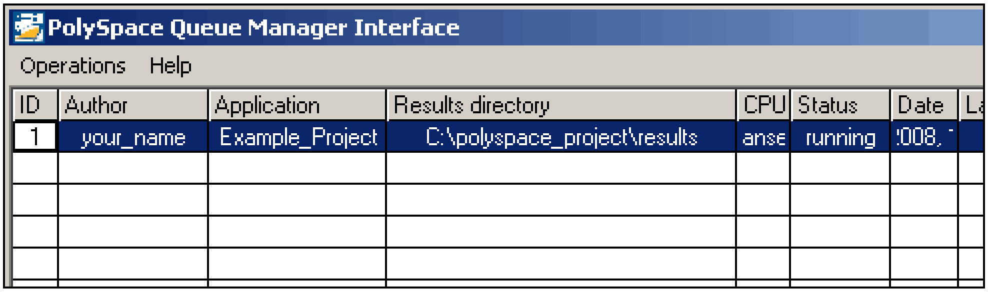
Note You must have the queue manager password to use Administrator Mode.

To use administrator mode:

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



The **PolySpace Queue Manager Interface** opens.



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

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 **Execute** button.

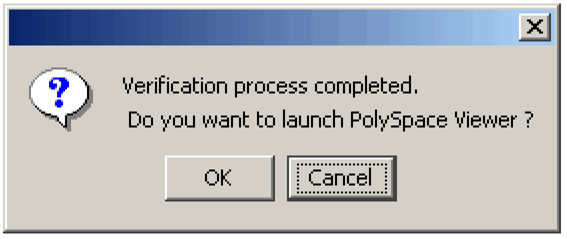
6 If you see a caution that PolySpace software will remove existing results from the results directory, click **Yes** to continue and close the message dialog box.

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.

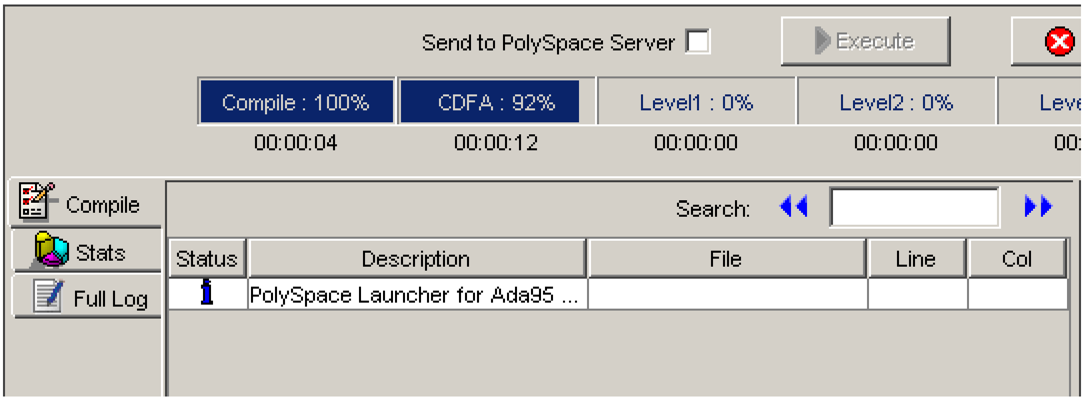
2 Generating a main if it does not find a main and the **Generate a Main** option is selected. For more information about generating a main, see

the section "-main-generator" in the "Options Description" chapter of the *PolySpace Client/Server for Ada User's Guide*.

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.

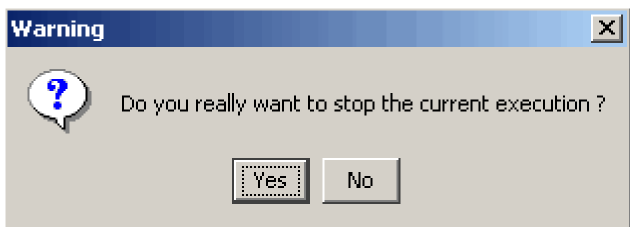
Stopping Client Verification Before It Completes

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

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.
  - `[-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.
- `psqueue-purge all|ended` — remove all completed verifications from the queue.
- `psqueue-dump` — gives the list of all verifications in the queue associated with the default Queue Manager.
- `psqueue-move-down <id>` — move down an identified verification in the Queue.
- `psqueue-remove <id>` — remove an identified verification in the queue.
- `psqueue-get-qm-server` — give the name of the default Queue Manager.
- `psqueue-progress <id>:` give progression of the currently identified and running verification.
  - `[-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.

7

Troubleshooting Verification Problems

Verification Process Failed Errors

In this section...

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`.

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

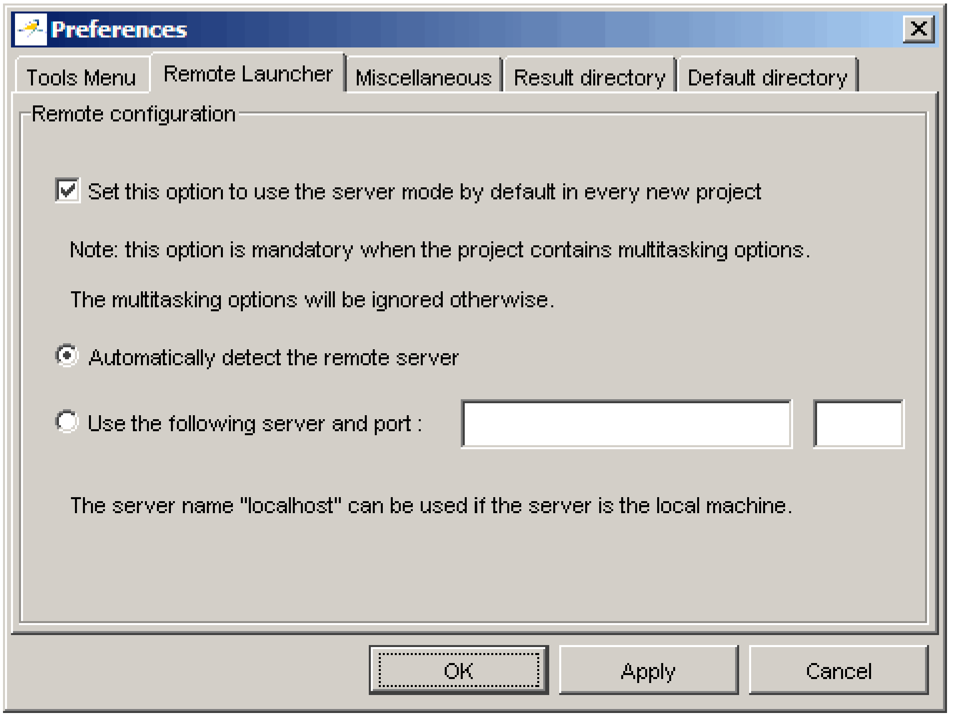
If you see the following message in the log, PolySpace software cannot find the server.

```
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

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).

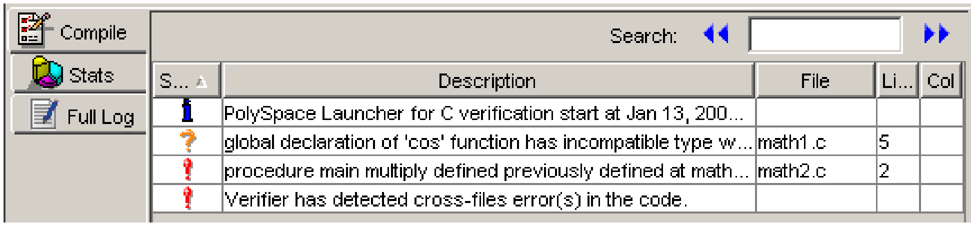
Examining the Compile Log

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.

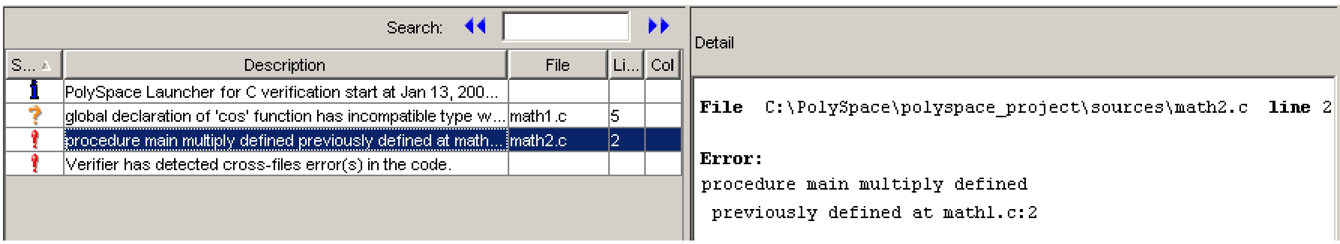
To examine errors in the Compile log:

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

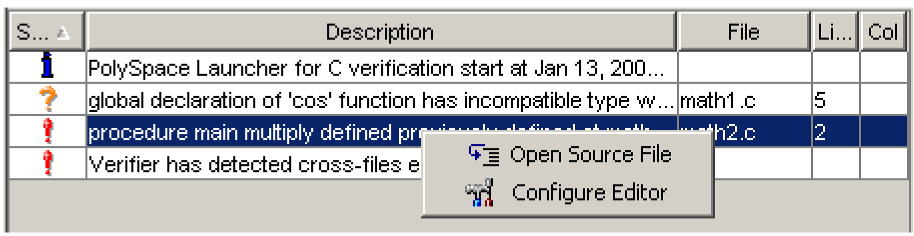
   A list of compile phase messages appear in the log part of the window.



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 3-16.

---

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

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.

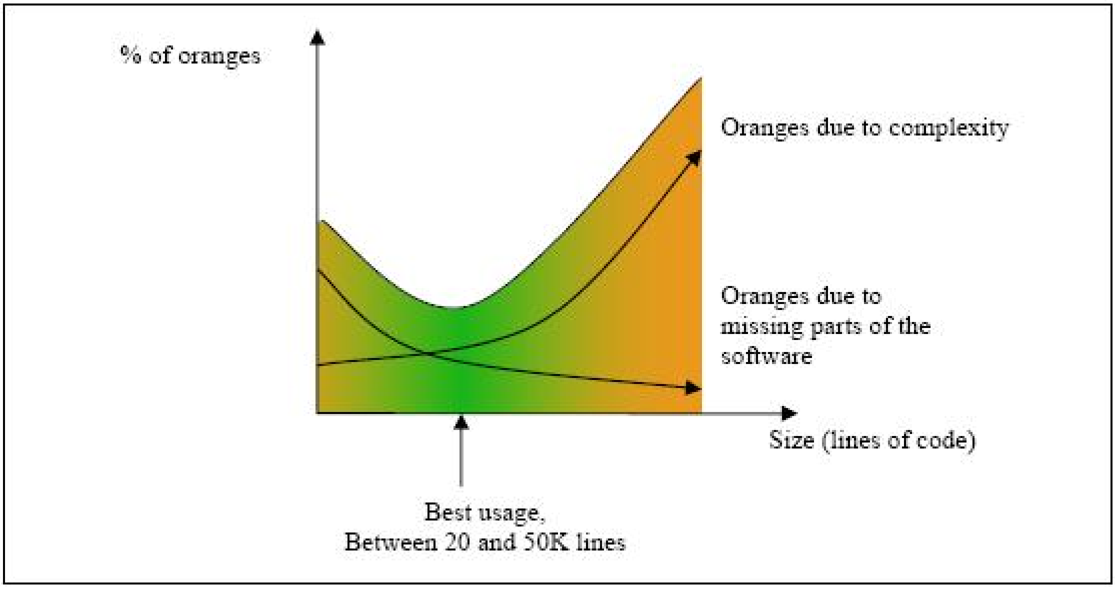
The fact that so many factors are involved makes it impossible to derive a precise formula to calculate verification duration. Following sub section try to give some hints to reduce time of a verification.

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

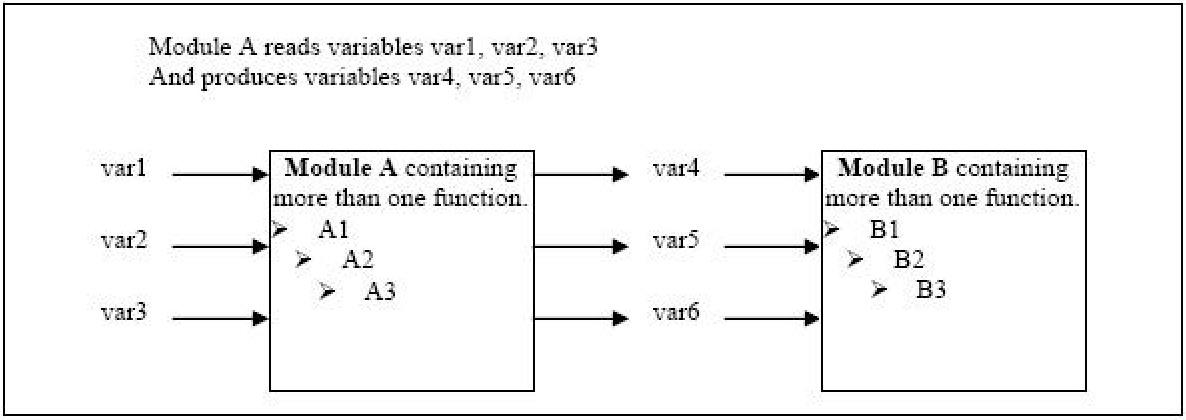
- (-) A slight loss of precision on y. PolySpace will now consider all possible values for y, including the formatted ones that were present at the first verification.
- (+) A huge investigation of the code is not necessary to isolate a meaningful subset. Any application can be split logically in this 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	• If var2 is equal to 0, than var4>var5>5. • If var3 is greater than 4, than var4<var5<12 • ...

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 var4>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 T2entry-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] U {10} U [15 ; 17] U {25} . Here, the same division appears as an orange check.

If the complexity of the internal data becomes even greater later in the verification, PolySpace 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.

8

Reviewing Verification Results

Before You Review PolySpace Results

In this section...

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 ~ [-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 ~ [-2^31, -1] and [100, 2^31-1] are stopped;

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

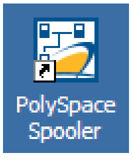
Downloading Results from Server to Client

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.

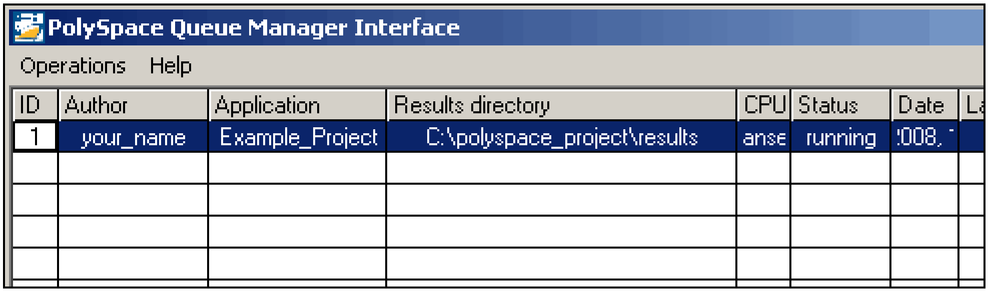
Note If you download results before the verification completes, you get partial results and the verification continues.

To download verification results to your client system:

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



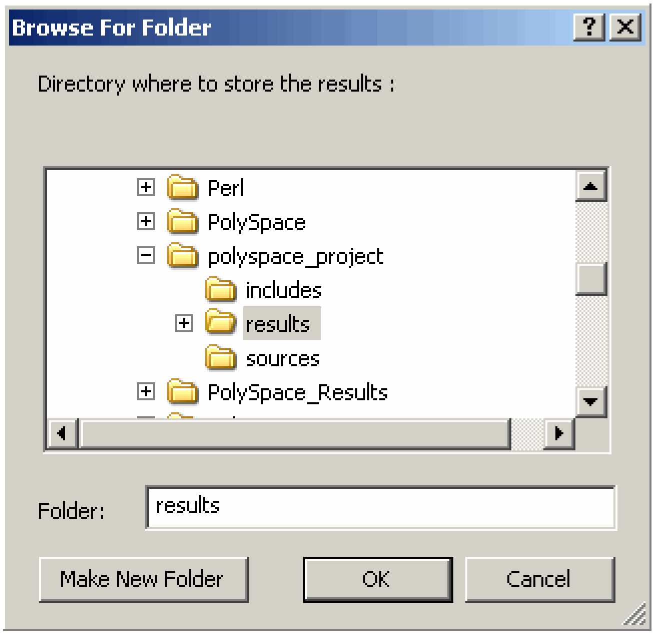
The **PolySpace Queue Manager Interface** opens.



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.

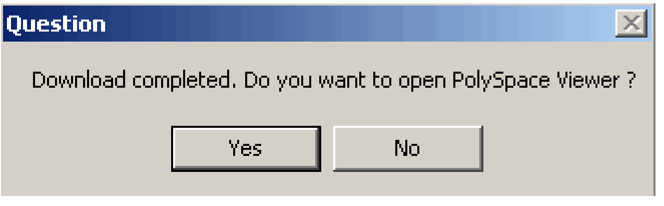
The **Browse For Folder** dialog box appears.



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.

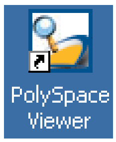
Opening Verification Results

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.

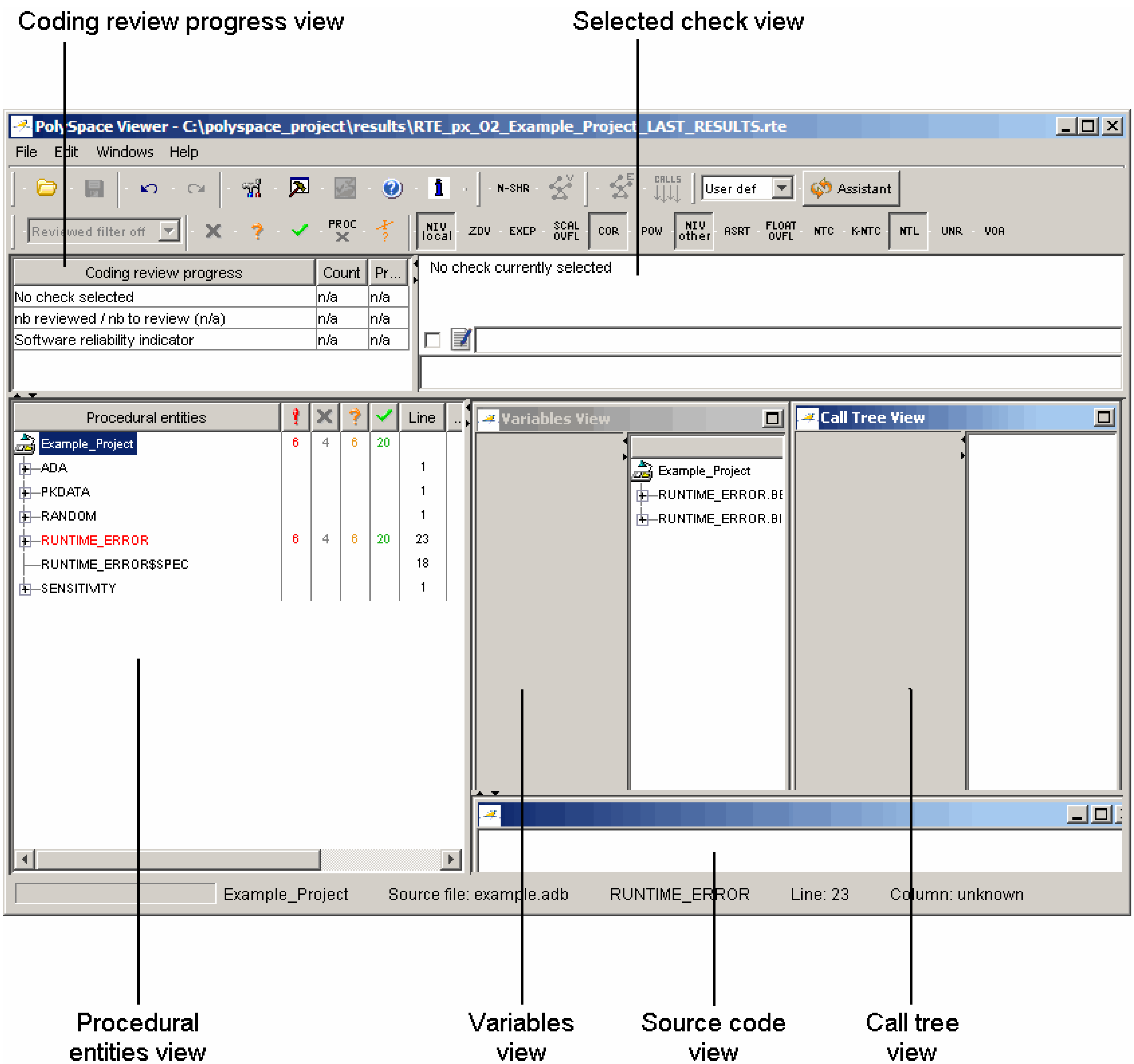
The results appear in the Viewer window.

Exploring the Viewer Window

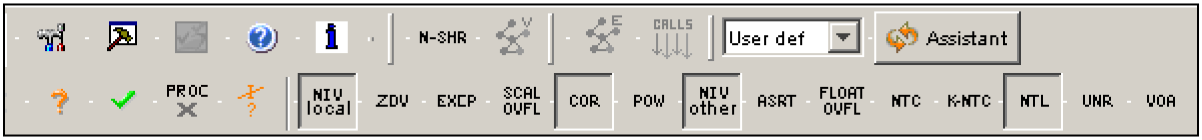
- "Overview" on page 8-11
- "Procedural Entities View" on page 8-13

Overview

The PolySpace Viewer looks like:



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



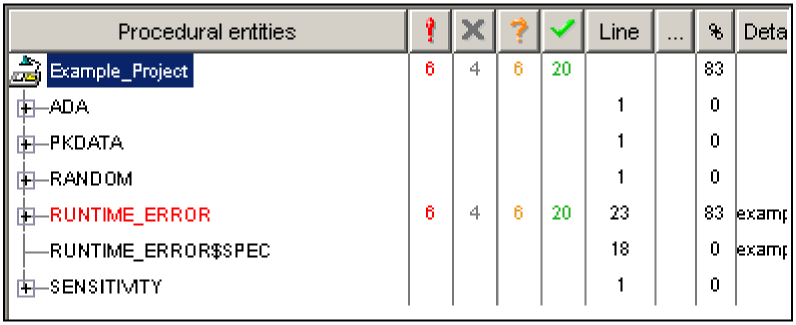
In both expert mode and assistant mode, the Viewer window has six sections below the toolbar. Each section provides a different view of the results. The following table describes these views.

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 **Note** The file that you use in this tutorial does not have global variables.
Call tree view	Tree structure of function calls

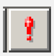
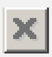
You can resize or hide any of these sections.

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:



The package `RUNTIME_ERROR` is red because its 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 (for operations where an error always occurs)
	Number of gray checks (for unreachable code)
	Number of orange checks (warnings for operations where an error might occur)
	Number of green checks (for operations where an error never occurs)
	Total number of red, green, and gray checks (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 8-30.

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.

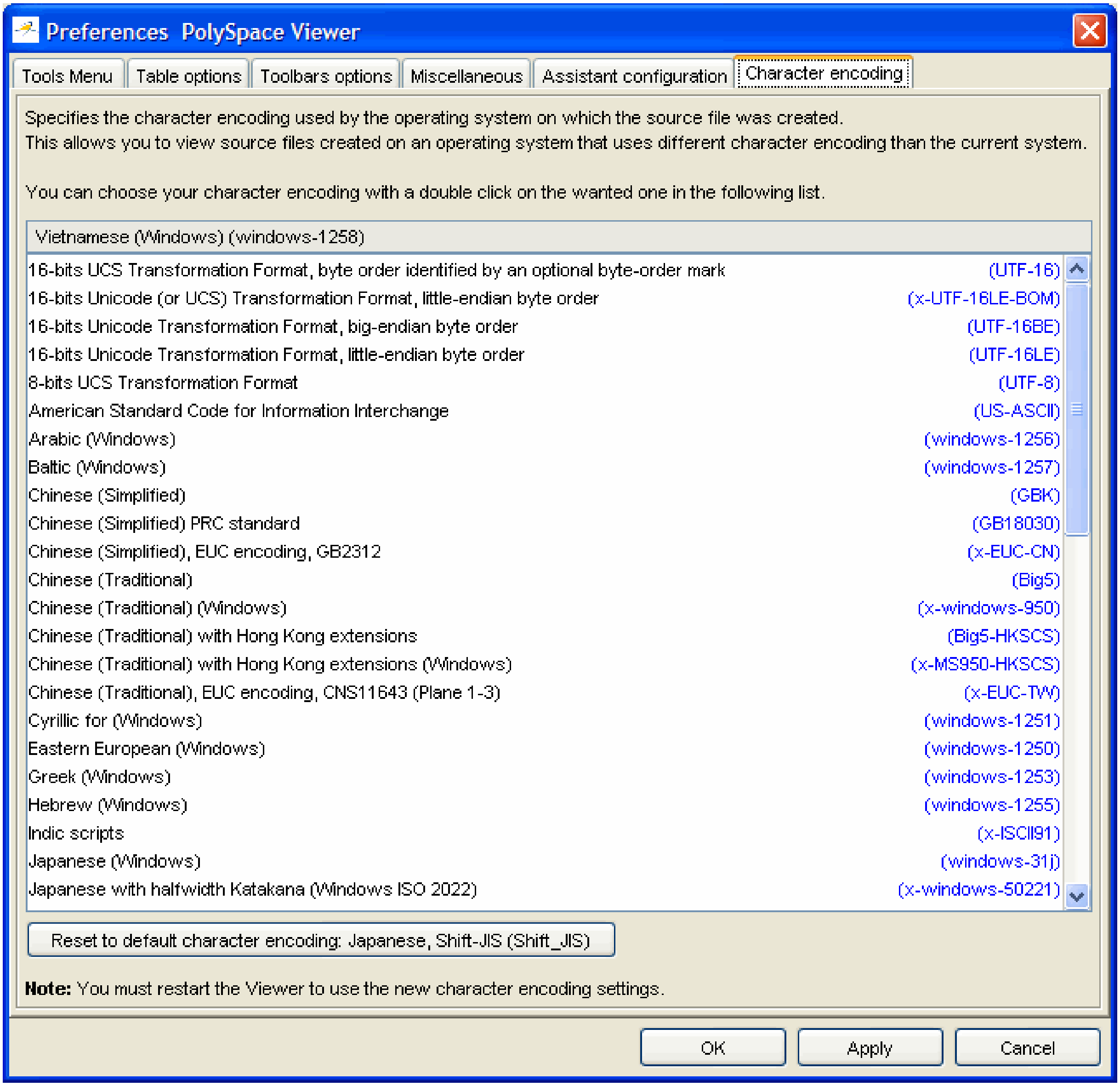
To set the character encoding for a source file:

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?

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 8-20.

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.

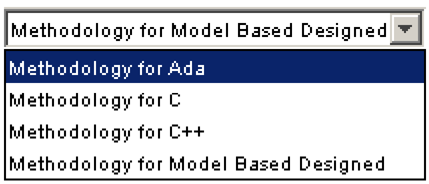
The controls for assistant mode include:

- 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:**

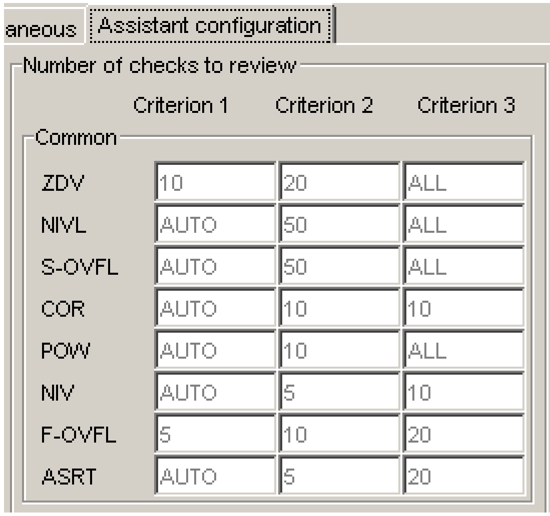
1. Select **Edit > Preferences**.

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

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

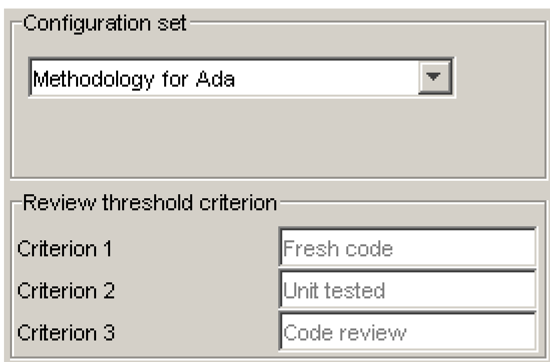
   The configuration for Methodology for Ada appears.

   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.



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).



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.

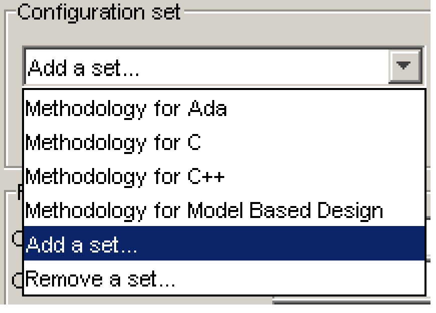
To define a custom methodology:

1 Select **Edit > Preferences**.

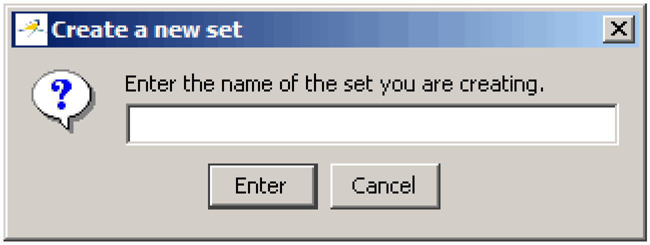
The **Preferences PolySpace Viewer** dialog box appears.

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.

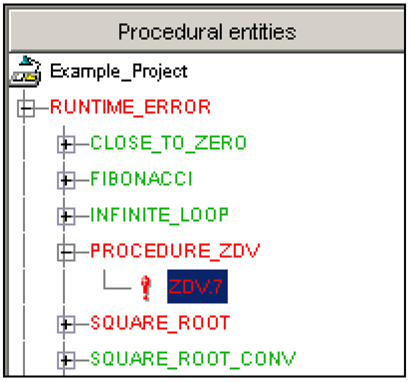
---

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 8-26.

---

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**.

Reviewing Results in Expert Mode

In this section...

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.

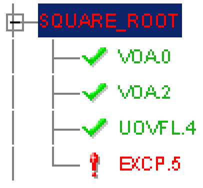
Selecting a Check to Review

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.

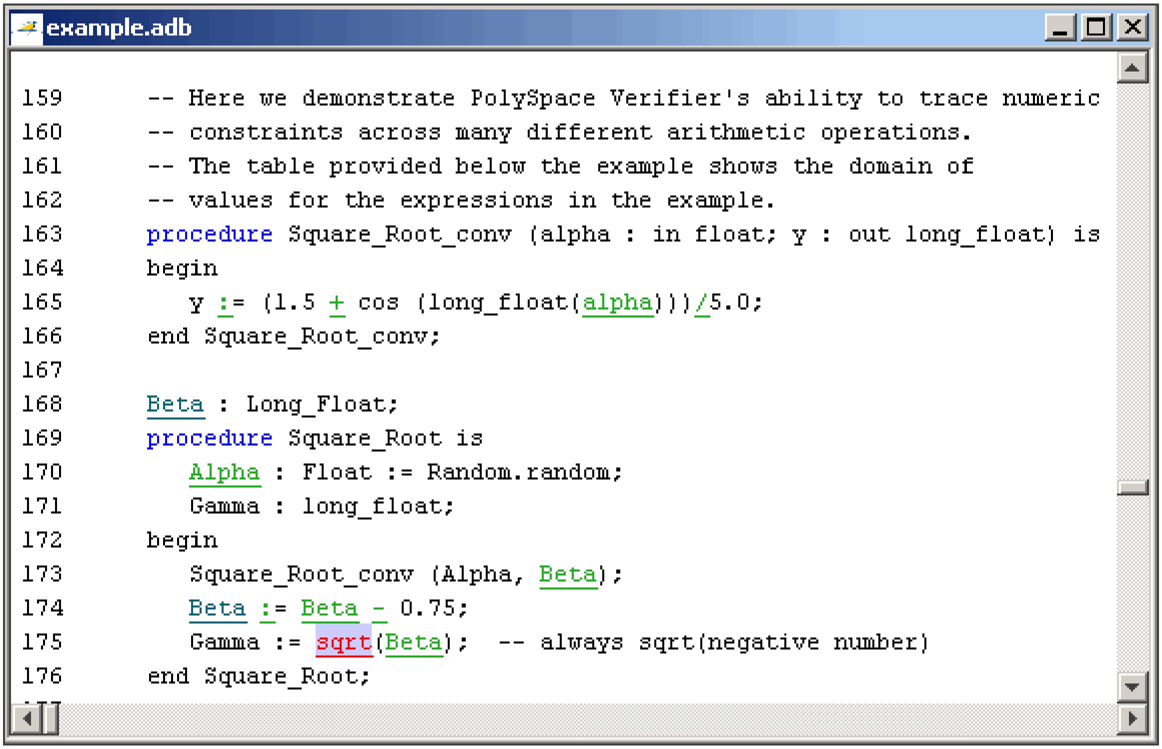
A color-coded list of the checks performed on the procedure appears:



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.




```
-- Here we demonstrate PolySpace Verifier's ability to trace numeric
-- constraints across many different arithmetic operations.
-- The table provided below the example shows the domain of
-- values for the expressions in the example.
procedure Square_Root_conv (alpha : in float; y : out long_float) is
begin
   y := (1.5 + cos (long_float(alpha)))/5.0;
end Square_Root_conv;

Beta : Long_Float;
procedure Square_Root is
   Alpha : Float := Random.random;
   Gamma : long_float;
begin
   Square_Root_conv (Alpha, Beta);
   Beta := Beta - 0.75;
   Gamma := sqrt(Beta);  -- always sqrt(negative number)
end Square_Root;
```

4 Click the colored check in the code.

An message box appears describing the error.

Displaying the Calling Sequence

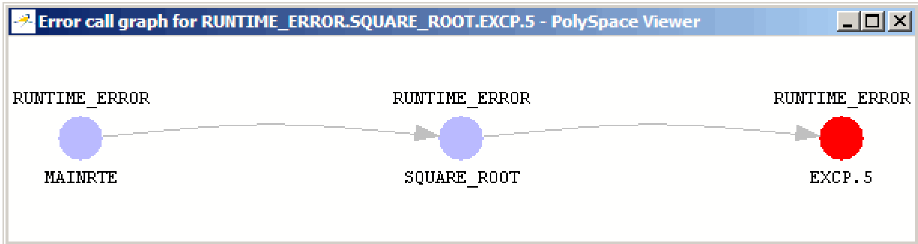
You can display the calling sequence that leads to the code associated with a check. To see the calling sequence for a check:

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.

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

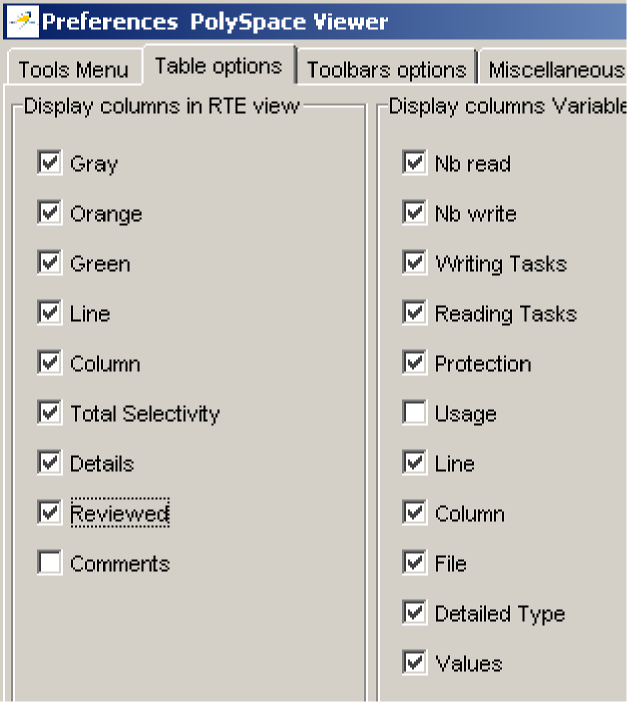
You can change the PolySpace Viewer preferences so that the procedural entities part of the window displays a **Reviewed** column.

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	!	✗	?	✓	Line	...	%	Details	Reviewed
Example_Project	6	4	6	20			83		☐
ADA					1		0		☐
PKDATA					1		0		☐
RANDOM					1		0		☐
RUNTIME_ERROR	6	4	6	20	23		83	exampl...	☐
CLOSE_TO_ZERO			4	1	91	3	20	exampl...	☐
FIBONACCI				3	143	3	100	exampl...	☐
INFINITE_LOOP				3	127	3	100	exampl...	☐
MAINRTE	3				198	3	100	exampl...	☐
MYABS		2			28	3	100	exampl...	☐
NON_INFINITE_LOOP				3	114	3	100	exampl...	☐
PROCEDURE_STUB					26	3	0	exampl...	☐
PROCEDURE_ZDV	1	1		1	38	3	100	exampl...	☐
RECURSION			1	4	66	3	80	exampl...	☐
RECURSION_CALLER	1				77	3	100	exampl...	☐
RECURSIVE_2					60	3	0	exampl...	☐
SQUARE_ROOT	1			1	169	3	100	exampl...	☐
VOA.0					170	6			☐
VOA.2					174	11			☐
UOVFL.4				1	174	19		[conve...	☐
EXCP.6	1				175	15		argume...	☑
SQUARE_ROOT_CONV				3	163	3	100	exampl...	☐
UNREACHABLE_CODE		1	1	1	182	3	67	exampl...	☐
VOA.0					113	3			☐

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:

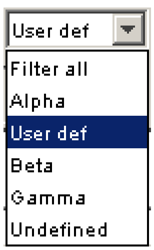
- Selects the check box for that check in the current check view (upper-right part of the window).
- Updates the counts in the coding review progress view (upper-left part of the window).

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 8-34
- "Composite Filters" on page 8-34
- "Custom Filters" on page 8-34

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.

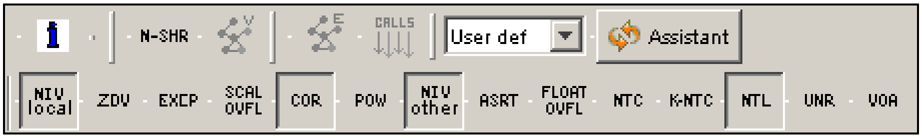
Composite Filters

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 8-35.

Creating a Custom Filter

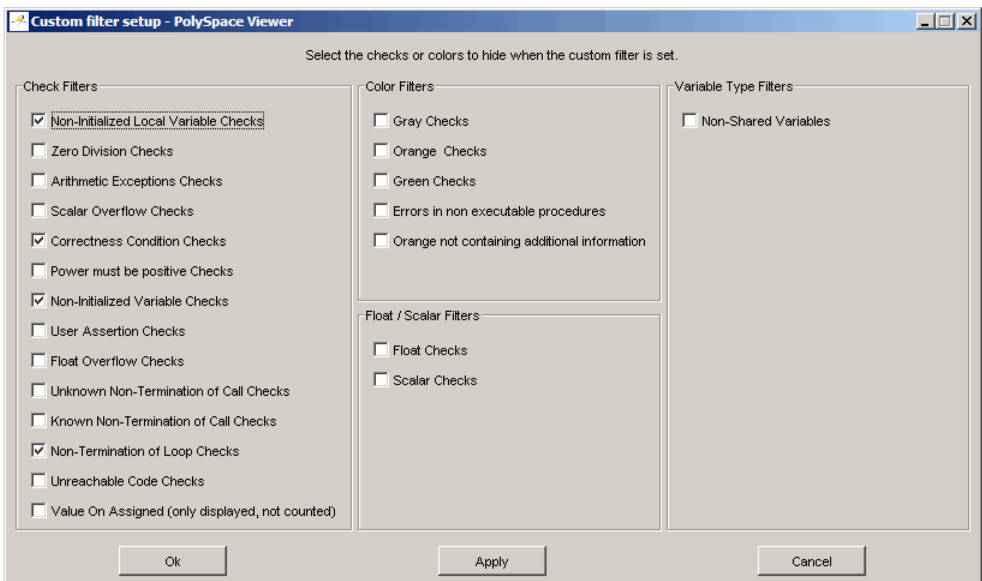
The custom filter is a composite filter that you define. It appears on the composite filter menu as `User def`.

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.

Generating Reports of Verification Results

You can generate a Microsoft® Excel® report of the verification results.

To generate an Excel report of your verification results:

1. Navigate to the `PolySpace-Doc` folder in your results directory. For example:`polypace_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
   ```

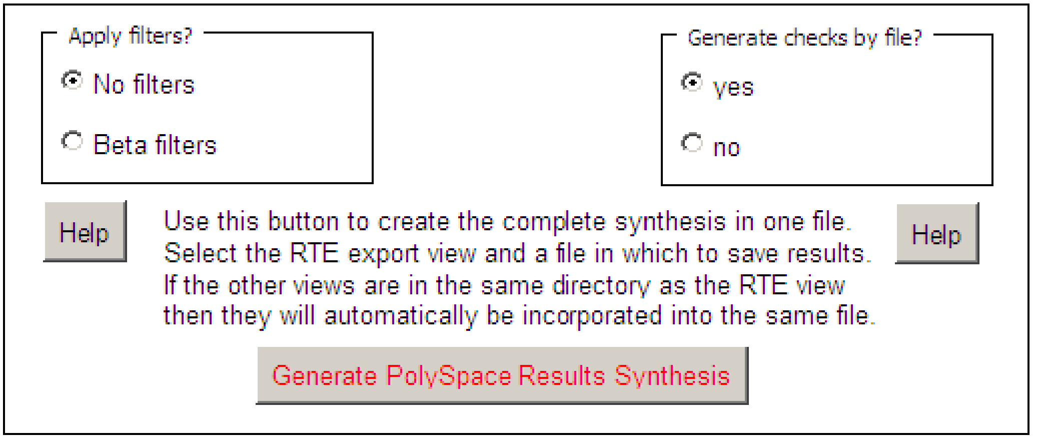
   The first three files correspond to the call tree, RTE, and variable views in the PolySpace Viewer window.

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:



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:`polypace_project\results\PolySpace-Doc`.

6 Select *`Project`*`_RTE_View.txt`.

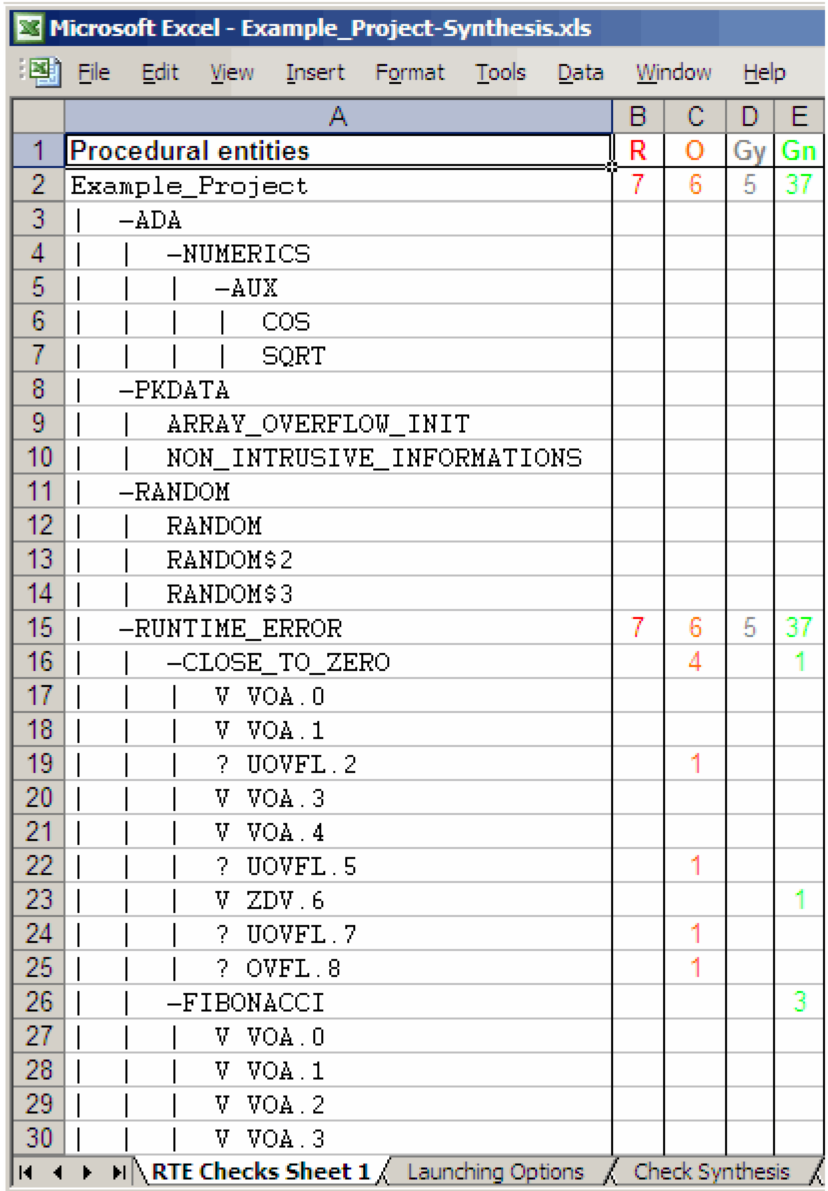
7 Click **Open** to close the dialog box.

   The **Where should I save the analysis file?** dialog box appears.

8 Keep the default file name and file type.

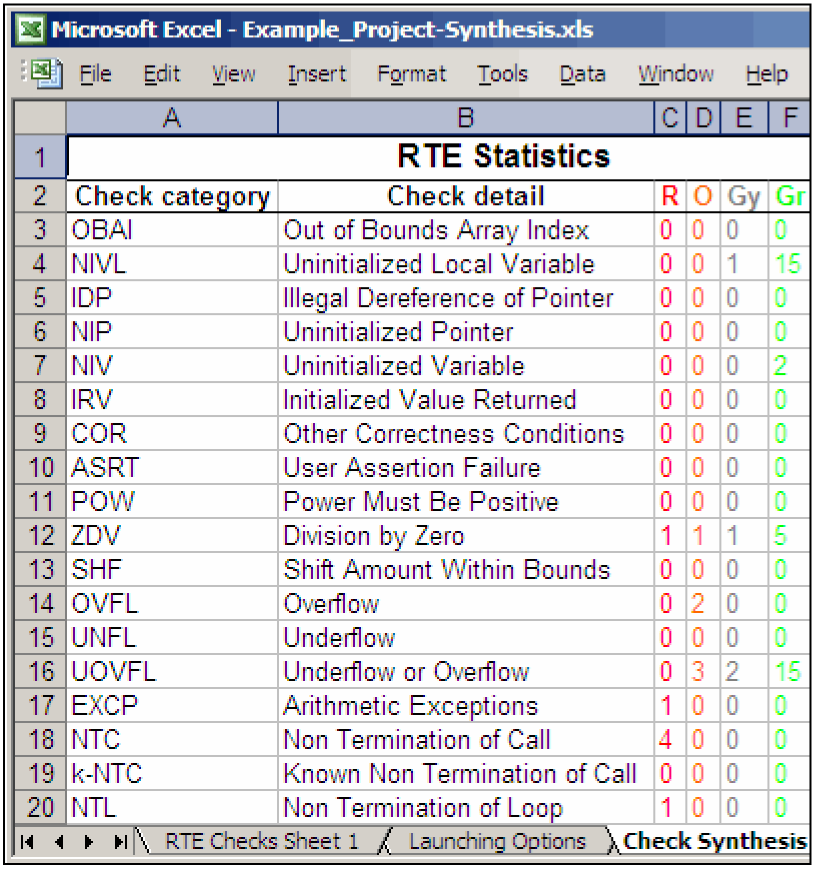
9 Click **Save** to close the dialog box and start the report generation.

   Microsoft Excel opens with the spreadsheet that you generated. This spreadsheet has several worksheets:



<table>
<tr><th></th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th></tr>
<tr><td>1</td><td>Procedural entities</td><td>R</td><td>O</td><td>Gy</td><td>Gn</td></tr>
<tr><td>2</td><td>Example_Project</td><td>7</td><td>6</td><td>5</td><td>37</td></tr>
<tr><td>3</td><td>| -ADA</td><td></td><td></td><td></td><td></td></tr>
<tr><td>4</td><td>| | -NUMERICS</td><td></td><td></td><td></td><td></td></tr>
<tr><td>5</td><td>| | | -AUX</td><td></td><td></td><td></td><td></td></tr>
<tr><td>6</td><td>| | | | COS</td><td></td><td></td><td></td><td></td></tr>
<tr><td>7</td><td>| | | | SQRT</td><td></td><td></td><td></td><td></td></tr>
<tr><td>8</td><td>| -PKDATA</td><td></td><td></td><td></td><td></td></tr>
<tr><td>9</td><td>| | ARRAY_OVERFLOW_INIT</td><td></td><td></td><td></td><td></td></tr>
<tr><td>10</td><td>| | NON_INTRUSIVE_INFORMATIONS</td><td></td><td></td><td></td><td></td></tr>
<tr><td>11</td><td>| -RANDOM</td><td></td><td></td><td></td><td></td></tr>
<tr><td>12</td><td>| | RANDOM</td><td></td><td></td><td></td><td></td></tr>
<tr><td>13</td><td>| | RANDOM$2</td><td></td><td></td><td></td><td></td></tr>
<tr><td>14</td><td>| | RANDOM$3</td><td></td><td></td><td></td><td></td></tr>
<tr><td>15</td><td>| -RUNTIME_ERROR</td><td>7</td><td>6</td><td>5</td><td>37</td></tr>
<tr><td>16</td><td>| | -CLOSE_TO_ZERO</td><td></td><td>4</td><td></td><td>1</td></tr>
<tr><td>17</td><td>| | | V VOA.0</td><td></td><td></td><td></td><td></td></tr>
<tr><td>18</td><td>| | | V VOA.1</td><td></td><td></td><td></td><td></td></tr>
<tr><td>19</td><td>| | | ? UOVFL.2</td><td></td><td>1</td><td></td><td></td></tr>
<tr><td>20</td><td>| | | V VOA.3</td><td></td><td></td><td></td><td></td></tr>
<tr><td>21</td><td>| | | V VOA.4</td><td></td><td></td><td></td><td></td></tr>
<tr><td>22</td><td>| | | ? UOVFL.5</td><td></td><td>1</td><td></td><td></td></tr>
<tr><td>23</td><td>| | | V ZDV.6</td><td></td><td></td><td></td><td>1</td></tr>
<tr><td>24</td><td>| | | ? UOVFL.7</td><td></td><td>1</td><td></td><td></td></tr>
<tr><td>25</td><td>| | | ? OVFL.8</td><td></td><td>1</td><td></td><td></td></tr>
<tr><td>26</td><td>| | -FIBONACCI</td><td></td><td></td><td></td><td>3</td></tr>
<tr><td>27</td><td>| | | V VOA.0</td><td></td><td></td><td></td><td></td></tr>
<tr><td>28</td><td>| | | V VOA.1</td><td></td><td></td><td></td><td></td></tr>
<tr><td>29</td><td>| | | V VOA.2</td><td></td><td></td><td></td><td></td></tr>
<tr><td>30</td><td>| | | V VOA.3</td><td></td><td></td><td></td><td></td></tr>
</table>

10 Select the **Check Synthesis** tab to view the worksheet showing statistics by check category:



	A	B	C	D	E	F
1	**RTE Statistics**					
2	**Check category**	**Check detail**	R	O	Gy	Gr
3	OBAI	Out of Bounds Array Index	0	0	0	0
4	NIVL	Uninitialized Local Variable	0	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

Using PolySpace Results

In this section...

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 8-42
- "Structural Coverage" on page 8-44

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

```
Package body Test is
 ATab : array(0..9) of Integer := (Others => 0);
 function Assign_array(X : integer) return Integer is
   Y : Integer;
  begin
   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.

Selective Orange Review: Finding the Maximum Number of Bugs in One Hour

A selective orange review is appropriate for the early stages of development, when you want to improve the quality of your code while it is being developed. Performing a selective orange review allows you to find the maximum number of bugs in a short period of time. For example, if you want to spend the first hour of the day reviewing a verification that was performed overnight. This type of review is generally supported by more extensive verification as the project nears completion.

A selective orange review can generally find about 5 bugs (in orange checks) during an hour of review.

Choosing What to Review

When performing a selective orange review, focus on the modules that have the highest selectivity in your application, meaning the highest ratio of (green + gray + red) / (total number of checks).

If PolySpace verification finds only one or two orange checks in a module or function, these checks are probably not caused by “basic imprecision.” Therefore, it is more likely that you will find bugs in these orange checks than in those found elsewhere in the code.

Note For each function, PolySpace verification may be better at detecting some kinds of Runtime Errors than others. For example, one function may yield precise results for OVFL, but imprecise results for NIV, while a second function may have the opposite results.

Therefor, you must apply the “high selectivity focus” to each type of error **separately**.

Reviewing Oranges Quickly

While performing a selective orange review:

- Spend no more than 5 minutes per orange check.
- Review at least 50 checks an hour.

If you find a check that takes more than a few minutes to understand, it may be the result of inconclusive PolySpace verification. To maximize the number of bugs you can find in a limited time, you should move on to another check. Generally, you should spend no more than 5 minutes on each check, remembering that your goal is to review at least 50 checks per hour to maximize the number of bugs found.

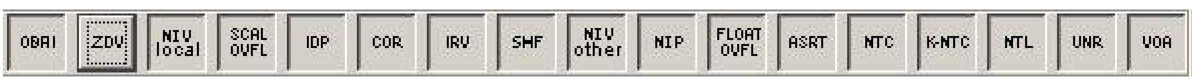
Performing a Selective Orange Review

The goal of a selective orange review is to identify the maximum number of bugs within a short period of time.

To perform a selective orange review:

1. Select one type of RTE, such as Zero Division (ZDV).
2. Click **Filter all** 

.
3. Click the type of check you want to review (ZDV in this example).



4. Identify files containing only 1 or 2 orange checks of the selected type.
5. Using the call tree and dictionary, perform a quick code review on each orange check, spending no more than 5 minutes on each.

   Your goal is to identify whether the orange check is a *potential bug, inconclusive check* or *data set issue*.

   If the check proves too complicated to explain quickly, it may well be the result of *basic imprecision*.

6 Once you identify the source of the orange check, select the **Verified** checkbox in the PolySpace Viewer, and enter an explanation in the comment field. For example, “inconclusive,” or “data set issue when calibration of <x> is set greater than 100.”

7 Select another type of RTE and repeat the procedure.

Note You can use the **Beta** filter to highlight the types of check most likely to include critical Runtime Errors.

Exhaustive Orange Review at Unit Phase

An exhaustive orange review during the unit testing phase can identify bugs not found during the selective orange review. However, the cost of performing an exhaustive orange review needs to be balanced with the cost leaving a bug in the code.

An exhaustive orange review typically progresses at a rate of about **50 orange checks per hour.** However, an hour spent on an exhaustive check review is different to an hour spent on a selective orange review in several significant ways:

- The first 10 minutes of the exhaustive check will be dedicated to the classification of 2/3 of the orange as false anomalies.
- The last 40 minutes will be used to track more complex bugs.

80% of the **orange checks** will require only a few seconds of effort before a conclusion can be reached. These are not integration bugs, so tracking the cause of an orange check is often much faster than the same activity in a larger piece of code. The typical time spent reviewing each check is about 1 minute.

Note If you apply coding rules to your project, reviewing PolySpace results generated by a unit verification normally takes no more than 15 minutes.

Exhaustive Orange Review at Integration Phase

An exhaustive orange during the integration testing phase can identify bugs not found by a selective orange review. However, the time/cost of performing an exhaustive orange review needs to be balanced with the cost leaving a bug in the code.

Cost

Reviewing each orange check will typically take approximately **4-5 minutes** . 400 orange checks will therefore require about four days of code review, and 3,000 orange checks will require 25 days.

However, if you review the checks as described in the Selective Orange Review section, the first 80% of checks will take a much smaller amount of time to review. You can then decide how far you want to pursue reviewing the remaining checks.

Method

There are sometimes situations where files contain a particularly high number of orange checks compared with the rest of the application. This may well highlight design issues.

Consider the three possible reasons for an orange check:

- **Potential bug and Data set issues**
- **Inconclusive verification**
- **Basic imprecision**

The method described in the following chapter explains how to focus on finding potential bugs in the orange code. We will focus here on the first and second types. We are assuming that in the modules containing the most orange checks, those checks will prove inconclusive. If PolySpace is unable to draw a conclusion, the implication is often that the code itself is very complex - which in turn can identify sections of code of low robustness and quality.

Real Bugs and Data Sets. If the data set verified reveals real bugs, they should be corrected. If it highlights potential input bugs (depending on the input data which might eventually be used), then the source code should be commented.

Inconclusive Check. The most interesting type of inconclusive check is identified when PolySpace states that the code is too complicated. In such a case it is usually true that most orange checks in the problem file are related, and that patient navigation will always draw the user back to a same cause - perhaps a function or a variable modified many times. Experience suggests that such situations often focus on functions or variables which have also caused trouble earlier in the development cycle.

Consider an example below. Suppose that

- a *signed* is an integer between -2^31 and 2^31-1
- an *unsigned* is an integer between 0 and 2^32-1
- The variable "Computed_Speed" is copied into a signed, and afterward into an unsigned, than signed, than added to another variable, and finally produces 20 orange overflows (OVFL).

There is no scenario identified which leads to a real bug, but perhaps the development team knows that there was trouble with this variable during development and the earlier testing phases. PolySpace has also found this to be a problem, providing supporting evidence that the code is poorly designed.

Basic Imprecision. On some rare occasions, a module will contain a lot of similar occurrences of a “basic imprecision”. This is most likely to be caused by a function close to the edge of an application, or in the stub routines.

In this case, PolySpace can only assist by means of the call tree and dictionary. This code needs to be reviewed by an alternative activity - perhaps through additional unit tests or code review with the developer. These checks are usually local to functions, so their impact on the project as a whole is limited.

Examples of extra activities might be

- Checking an interpolation algorithm in a function

- Checking calibration data consisting of huge constant arrays, which are manipulated mathematically

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:

```
package body Main is
 procedure Main is
  X: array (1..5) of Integer;
   Tmp: Integer;
   Zero: Integer:= 0;
  begin
   X:= (1,2,3,4,5);
   if (X(4) > X(5))
   then
    Tmp:= 1 / Zero;
   end if;
  end;

 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

```
package DCHECK is
 type Pixel is
  record
   X : Integer;
   Y : Integer;
  end record;
 procedure MAIN;

 NError : Integer;
  procedure Failure (Val : Integer := Nerror);
  procedure MessageFailure (str : String := "");
 end DCHECK;

 package body DCHECK is
  type TwentyFloat is array (Integer range 1.. 20) of Float;

  procedure AddPixelValue(Vpixel : Pixel) is
  begin
   if (Vpixel.X < 3) then
    Failure; -- NIV Verified: Variable is initialized (Nerror)
    MessageFailure; -- COR Verified: Value is in range (string)
   end if;
  end AddPixelValue;

  procedure MAIN is
   B : Twentyfloat;
   Vpixel : Pixel;
  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

```
package test is
 procedure main;
end test;

package body test is

 subtype range_0_3 is integer range 0..3;
 Vg : Integer := 1;
 Pragma Volatile( Vg );

  function random return integer;
  type my_rec1 is
   record
    a : integer := 2 + random; -- Unproven OVFL coming from _INIT_PROC procedure (initialization of V1)
    b : float := 0.2;
   end record;
  V1 : my_rec1;
  V2 : my_rec1 := (10, 10.10);

  procedure main is
   Function Random return Boolean;
  begin
   null;
  end;
 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.

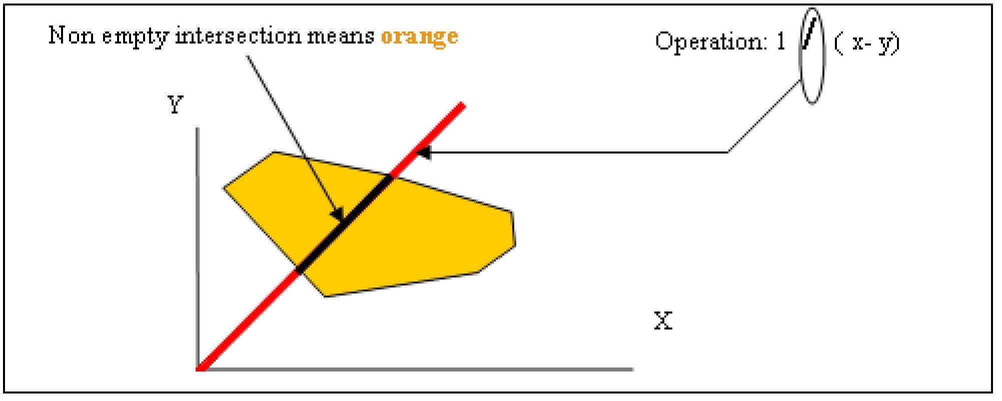
Managing Orange Checks

Understanding Orange Checks

In this section...

What is an Orange Check?

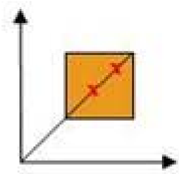
If a check is orange, it means that the approximate data set assumed by the verification to represent a variable intersects with the error zone.



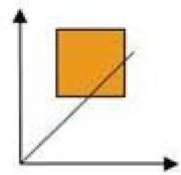
Graphical Representation of an Orange Check

Behind this picture, the orange color can reveal any of the situations below.

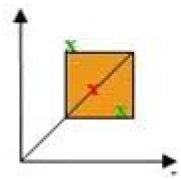
Note Any an orange check can approximate a check of any other color.



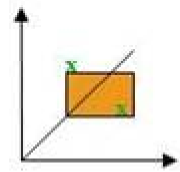
Red approximated by orange



Gray approximated by orange



Any other situation: real orange



Green approximated by orange

If PolySpace software attempted to manipulate every possible discrete value for all variables, the overheads for the verification would be so large that the problem would become incomputable. PolySpace verification manipulates polyhedrons representing data sets, and therefore cannot distinguish the category of an orange. That task is left to you, and is detailed in the following chapters.

(As a consequence, sometimes you may find an orange check which represents something which seems an obvious bug, and at other times you may find such a check which is obviously safe. As far as the mechanism within PolySpace software is concerned, it simply represents the intersection of two data sets – which is why you are left to perform the results review to draw these distinctions.)

Sources of Orange Checks

There are a number of possible causes of orange checks to be considered.

- **Potential bug** — an orange check can represent a real bug.

  Example - loop with division by zero

- **Inconclusive check** — an orange check can represent a situation where PolySpace verification is unable to conclude whether a problem exists. It is sometimes in the nature of software code that it cannot be concluded whether there is a potential error. In the example below, the task T1 can be started before or after T2, so PolySpace verification cannot conclude without the calling sequence being defined.

- Consider a variable X initialized to 0, and two concurrent tasks T1 and T2.
- Suppose that T1 assigns a value of 12 to variable X
- Now suppose that T2 divides a local variable by X. The division is shown as an orange check because T1 can be started before or after T2 (so a division by zero is possible).

• **Data set issue** — an orange check resulting from a theoretical set of data. PolySpace verification considers all combinations of input data rather than *one* particular combination (that is, it uses an upper approximation of the data set). Therefore a check may be colored orange as the result of a combination of input values which is analyzed by PolySpace, but which will not be possible at execution time.

- Consider three variables X, Y and Z which can vary between 1 and 1000
- Now suppose that the code computes a value of X*Y*Z on a type 16 bits. The result can potentially overflow. It may be known when the code is developed that the variables cant all take the value 1000 at the same time, but this information is not available to PolySpace software. The code will be colored orange, accordingly.

• **Basic imprecision** — an orange check can be due to an imprecise approximation.

- Consider that *X*, before the function call, can have the following values: `-5`, `-3`, `8`, or any value in range `[10...20]`.
- This means that `0` has been excluded from the set of possible values for *X*. Therefore, PolySpace software will approximate *X* in the range `[-5...20]`, instead of the previous unions of values, because of imprecision and optimization.
- In this case, calling the function `x = 1/x` leads to an orange ZDV. PolySpace is not able to prove the absence of a run-time error.

Determining Cause of Orange Checks

Consider each of the four categories in turn. Bugs may be revealed by any category of orange check other than the “Basic imprecision” category.

- **Potential bug** — An orange check can reveal code which will fail under some circumstances. The following section describes how to find them.
- **Inconclusive verification** — Most inconclusive orange checks will take some time to investigate. An inconclusive orange check may well result from a very complex situation such that it may take an hour or more to understand the cause. You may decide to recode in order to be certain that there is no risk, bearing in mind the criticality of the function and the required speed of execution.
- **Data set issue** — It is normally possible to conclude that an orange check is the result of data set problem in a couple of minutes. You may wish to comment the code to flag this warning, or alternatively modify the code in order to take constraints into account.
- **Basic imprecision** — PolySpace verification cannot help to debug this code. You may or may not have a problem here, but you will need a supplementary activity to be sure. Most of the time, a quick code review is a suitable path to take, perhaps using the Viewers navigation facilities.

Reducing Orange Checks in Your Results

In this section...

Options to Reduce Orange Checks

Although PolySpace verification is effective and straightforward to launch with the minimum of effort, you may find that some applications would benefit from some code preparation in order to streamline the job of working through the resulting orange checks. There are four primary approaches which may be adopted in isolation or in combination.

- Apply coding rules. This is **the most efficient means to reduce oranges.**
- Implement manual stubbing of previously missing (and therefore automatically stubbed) functions.
- Specify call sequences with care.
- Constrain some "Applying Function Constraints to Variables Via Stubs" on page 9-10. Conventional testing verifies a single set of data, whereas PolySpace software can analyze your module for problems by taking into account all possible data values. If the range of possible values is specified more precisely than the default "full range" approach, then there will be less "noise" in the form of orange checks resulting from "impossible" values.

Generic Objectives: A Balance Between Precision and Verification Time

The methodology objective is quite simple: “To get the most precise results in the time available”.

PolySpace verification needs to be fast and precise.

- If a verification takes an eternity and the results contain the maximum possible number of gray, red and green checks, this verification is not useful because of the time spent waiting for the results.
- If a verification is very quick but contains only orange checks, the verification wont be very useful because of the large number of manual checks to be performed.

Using PolySpace verification is a compromise between verification time and precision. Factors such as the amount of time the developer has to assign to using PolySpace software, and the stage in the V cycle also influence the compromise. Consider for example the following scenarios that require the PolySpace software to be used in different ways:

- Unit testing phase: before going to lunch, a developer starts a verification. After returning from lunch the developer will analyze PolySpace results for a maximum of **one hour**.
- Integration/module testing: before going home, a developer starts a verification and will spend **the next morning** analyzing the results.
- Validation/acceptance testing: the developer leaves the office on Friday evening and starts a verification. The developer will spend the following **week** analyzing the results.

Note So verification time and precision depends on how long the developer wants to wait for the results and the amount of time available to review the results. It can happen that a verification never ends. The user might need to split his application.

Note With knowledge of the tool, users will choose one of the four precision options, (-O0, -O1, -O2, or -O3) before applying it to their process. It is implicit that a higher precision will require a longer verification time - but will yield more red, green and gray code and fewer oranges.

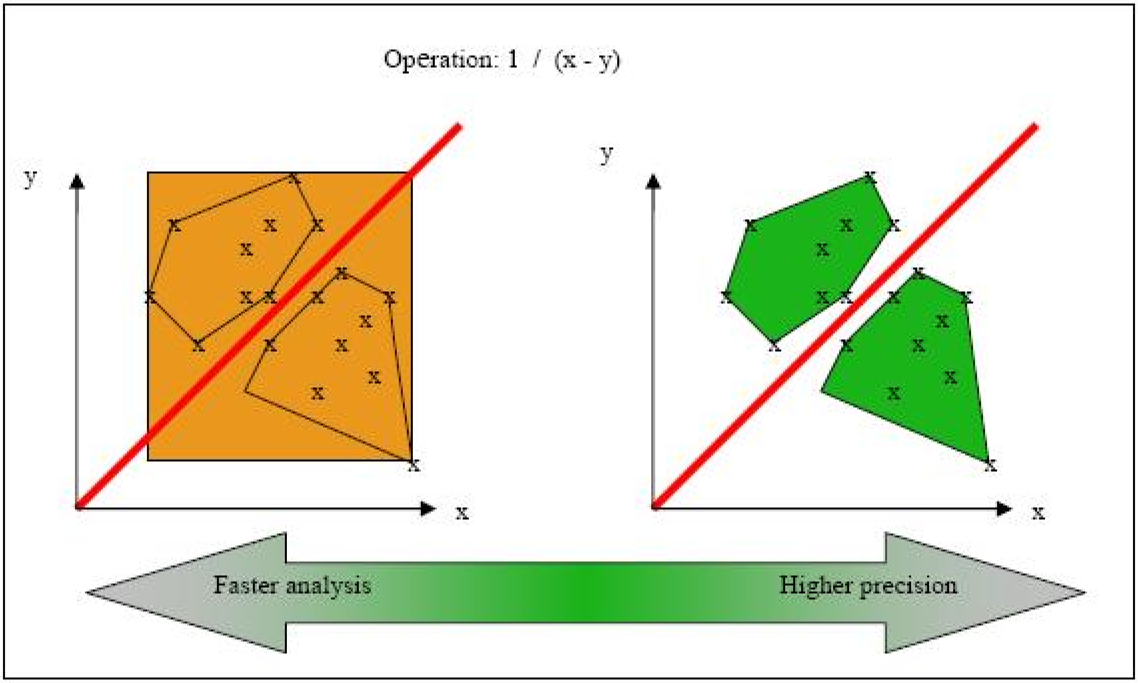
Most of the time, the first verification should use the lowest precision mode.

Note All activities and methods relating to results verification remain unchanged regardless of the precision selected (-O0, -O1, -O2 or -O3).

Varying the Precision Level

One way to affect precision is to select the algorithm that will be used to model the cloud of points. The exact method of modelling is managed internally, but you can influence it by selecting the -O0, -O1, -O2 or -O3 precision level. You can also select a particular precision for a specific file.

The methods used by PolySpace to represent the data internally are reflected in the level of precision to be seen in the results. As illustrated below, the same orange check which results from a low precision verification will become green when analyzed at a higher precision.



Vary the Precision Rate

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.

Increase the Number of Red and Green Checks

This example shows a header for a missing function (which might occur, for example, if the code is an incomplete subset of a project). The missing function copies the value of the src parameter to dest:

```
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.

Applying Function Constraints to Variables Via 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

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.

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"

function stub return INTEGER is
tmp: INTEGER;
random: INTEGER;
pragma volatile (random);
begin
tmp:= random;
pragma assert (tmp>=1);
pragma assert (tmp<=10);
return tmp;
end;
	function random return INTEGER;
	pragma Interface (C, random);
	function stub return INTEGER is
	tmp: INTEGER;
	begin
	tmp:= random;
	pragma assert (tmp>=1);
	pragma assert (tmp<=10);
	return tmp;
	end;
		function random return INTEGER;
		pragma Interface (C, random);
		function stub return INTEGER is
		tmp: INTEGER;
		begin
		tmp:= random;
		while (tmp<1 or tmp>10)
		loop
		tmp:=random;
		end loop;
		return tmp;

		`end;`

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.

Tuning Advanced Parameters

There is a compromise to be made to balance the time required to obtain results, and the precision of those results. Consequently, launching PolySpace with the following options will allow the time taken for verification to be reduced but will compromise the precision of the results. It is suggested that the parameters should be used in the sequence shown - that is, if the first suggestion does not increase the speed of verification sufficiently then introduce the second, and so on.

- switch from -O3 to a lower precision, -O2, -O1 or -O0

Describing Multitasking Behavior Properly

The proper description of the asynchronous characteristics of the application (implicit task declarations, mutual exclusion, critical sections) is necessary if the best results are to be achieved with the PolySpace software.

Consider two tasks T1 and T2 and a shared variable X set to 0 at initialization phase:

- T1 sets X to 12
- T2 divides by X

Because the task T1 can be started *before* or *after* T2, the division is orange. Modelling the task differently could turn this orange check green or red.

Refer to *"Preparing Multitasking Code" on page 5-15* for a complete description of tasking facilities. These include:

- Shared variable protection:
  - Critical sections,

  - Mutual exclusion,
  - Tasks synchronization,
- Tasking:
  - Threads, interruptions,
  - Synchronous/asynchronous events,
  - Real-time OS.

Reviewing Orange Checks

In this section...

Selective Orange Review

A selective orange review is appropriate for the early stages of development, when you want to improve the quality of your code while it is being developed. Performing a selective orange review allows you to find the maximum number of bugs in a short period of time. For example, if you want to spend the first hour of the day reviewing a verification that was performed overnight. This type of review is generally supported by more extensive verification as the project nears completion.

A selective orange review can generally find about 5 bugs (in orange checks) during an hour of review.

Choosing What to Review

When performing a selective orange review, focus on the modules that have the highest selectivity in your application, meaning the highest ratio of (green + gray + red) / (total number of checks).

If PolySpace verification finds only one or two orange checks in a module or function, these checks are probably not caused by "basic imprecision." Therefore, it is more likely that you will find bugs in these orange checks than in those found elsewhere in the code.

Note For each function, PolySpace verification may be better at detecting some kinds of Runtime Errors than others. For example, one function may yield precise results for OVFL, but imprecise results for NIV, while a second function may have the opposite results.

Therefor, you must apply the “high selectivity focus” to each type of error **separately**.

Review Oranges Quickly

While performing a selective orange review:

- Spend no more than 5 minutes per orange check.
- Review at least 50 checks an hour.

80% of **orange checks** require only a few seconds of effort before you can reach a conclusion. These are not integration bugs, so tracking the cause of an **orange check** is often much faster than the same activity in a larger piece of code.

If you find a check that takes more than a few minutes to understand, it may be the result of inconclusive PolySpace verification. To maximize the number of bugs you can find in a limited time, you should move on to another check. Generally, you should spend no more than 5 minutes on each check, remembering that your goal is to review at least 50 checks per hour to maximize the number of bugs found.

Performing a Selective Orange Review

The goal of a selective orange review is to identify the maximum number of bugs within a short period of time.

To perform a selective orange review:

1 Select one type of RTE, such as Zero Division (ZDV).

2 Click **Filter all** 
.

3 Click the type of check you want to review (ZDV in this example).

4 Identify files containing only 1 or 2 orange checks of the selected type.

5 Using the call tree and dictionary, perform a quick code review on each orange check, spending no more than 5 minutes on each.

   Your goal is to identify whether the orange check is a *potential bug, inconclusive check* or *data set issue.*

   If the check proves too complicated to explain quickly, it may well be the result of *basic imprecision.*

6 Once you identify the source of the orange check, select the **Verified** checkbox in the PolySpace Viewer, and enter an explanation in the comment field. For example, “inconclusive,” or “data set issue when calibration of <x> is set greater than 100.”

7 Select another type of RTE and repeat the procedure.

Note You can use the **Beta** filter to highlight the types of check most likely to include critical Runtime Errors.

Exhaustive Orange Review

An exhaustive orange review is generally conducted later in the development process, during the unit testing phase and integration testing phase. The purpose of an exhaustive orange review is to identify bugs not found during the selective orange review. The time/cost of performing an exhaustive orange review needs to be balanced with the cost leaving a bug in the code.

Reviewing each orange check will typically take approximately **4-5 minutes** . 400 orange checks will therefore require about four days of code review, and 3,000 orange checks will require 25 days.

However, if you review the checks as described in the Selective Orange Review section, the first 80% of checks will take a much smaller amount of time to review. You can then decide how far you want to pursue reviewing the remaining checks.

Performing an Exhaustive Orange Review

Performing an exhaustive orange review involves reviewing each orange check individually. However, there are some general guidelines to follow. In any hour performing an exhaustive orange review:

- The first 10 minutes will be dedicated to classifying 2/3 of the orange checks as false anomalies.
- The last 40 minutes will be used to track more complex bugs.

There are sometimes situations where files contain a particularly high number of orange checks compared with the rest of the application. This may well highlight design issues.

Consider the possible reasons for an orange check:

- **Potential bug and Data set issues**
- **Inconclusive verification**
- **Data set issue**
- **Basic imprecision**

Generally, in the modules containing the most orange checks, those checks will prove inconclusive. If PolySpace verification is unable to draw a conclusion, the implication is often that the code itself is very complex — which in turn can identify sections of code of low robustness and quality.

Inconclusive

The most interesting type of inconclusive check is identified when PolySpace verification states that the code is too complicated. In such a case it is usually true that most orange checks in the problem file are related, and that patient navigation will always draw the user back to a same cause — perhaps a function or a variable modified many times. Experience suggests that such situations often focus on functions or variables which have also caused trouble earlier in the development cycle.

Consider an example below. Suppose that

- a *signed* is an integer between -2^31 and 2^31-1
- an *unsigned* is an integer between 0 and 2^32-1
- The variable "Computed_Speed" is copied into a signed, and afterward into an unsigned, than signed, than added to another variable, and finally produces 20 orange overflows (OVFL).

There is no scenario identified which leads to a real bug, but perhaps the development team knows that there was trouble with this variable during development and the earlier testing phases. PolySpace software has also found this to be a problem, providing supporting evidence that the code is poorly designed.

Basic Imprecision

On some rare occasions, a module will contain a lot of *basic imprecision* due to approximations made by PolySpace. (For more information, see "Sources of Orange Checks" on page 9-3 and "Approximations Used During Verification"in the PolySpace Products for Ada Reference).

In this case, PolySpace verification can only assist by means of the call tree and dictionary. This code needs to be reviewed by an alternative activity - perhaps through additional unit tests or code review with the developer. These checks are usually local to functions, so their impact on the project as a whole is limited.

Examples of extra activities might be

- Checking an interpolation algorithm in a function
- Checking calibration data consisting of huge constant arrays, which are manipulated mathematically

Real Bugs and Data Sets

If the data set analyzed reveals real bugs, they should be corrected If it highlights potential input bugs (depending on the input data which might eventually be used) then the source code should be commented.

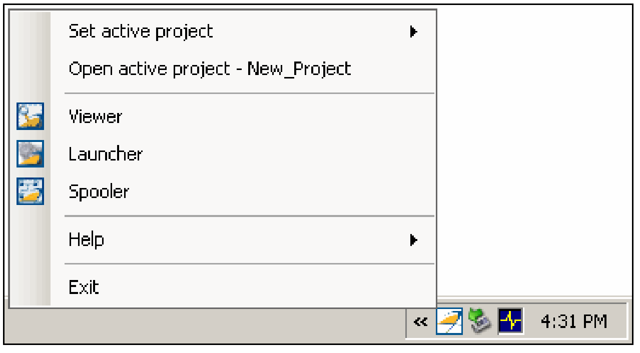
Day to Day Use

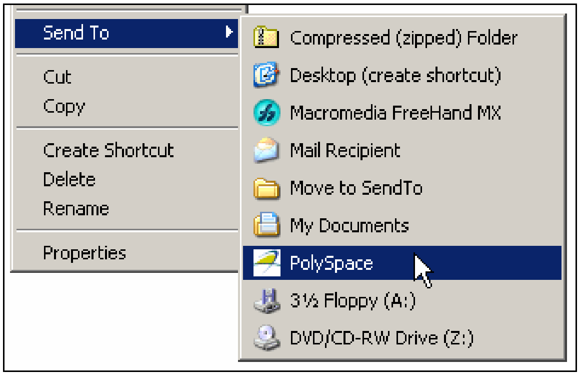
PolySpace In One Click Overview

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

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

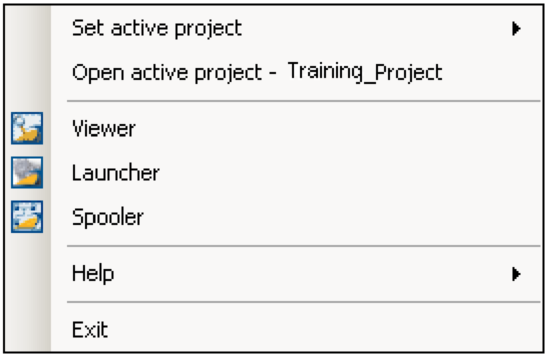
The active project is the project that PolySpace In One Click uses to verify the files that you select. Once you have set an active project, it remains active until you change the active project. PolySpace software uses the analysis options from the project; it does not use the source files or results 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:

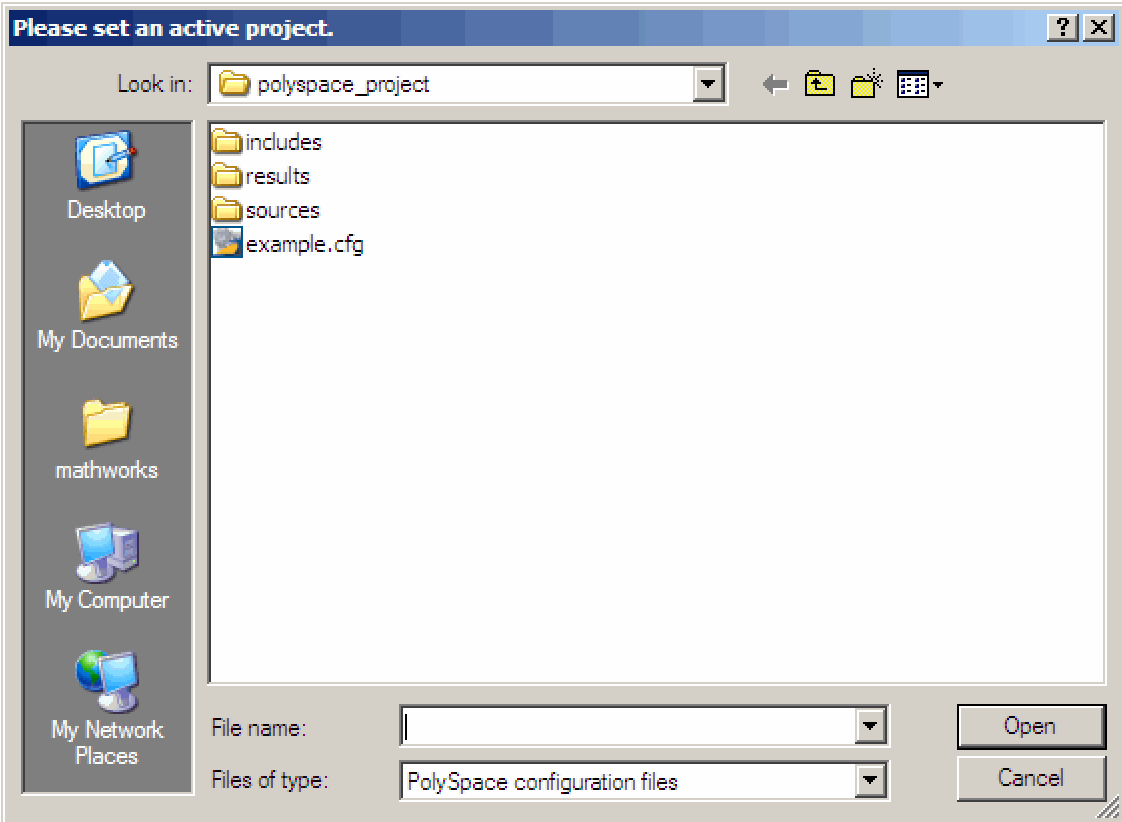
   

   The context menu appears.



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

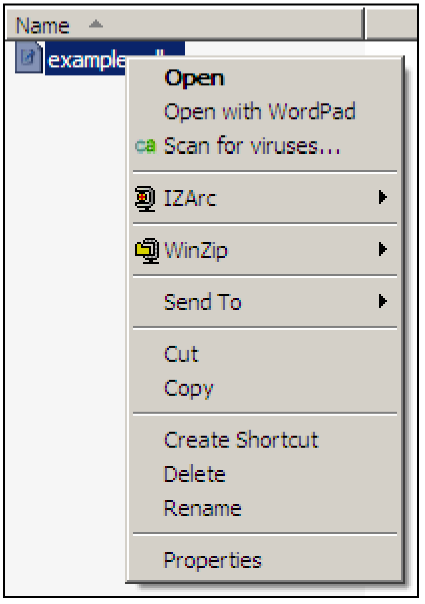
PolySpace in One Click allows you to send multiple files to PolySpace software for verification.

To send a file to PolySpace software for verification:

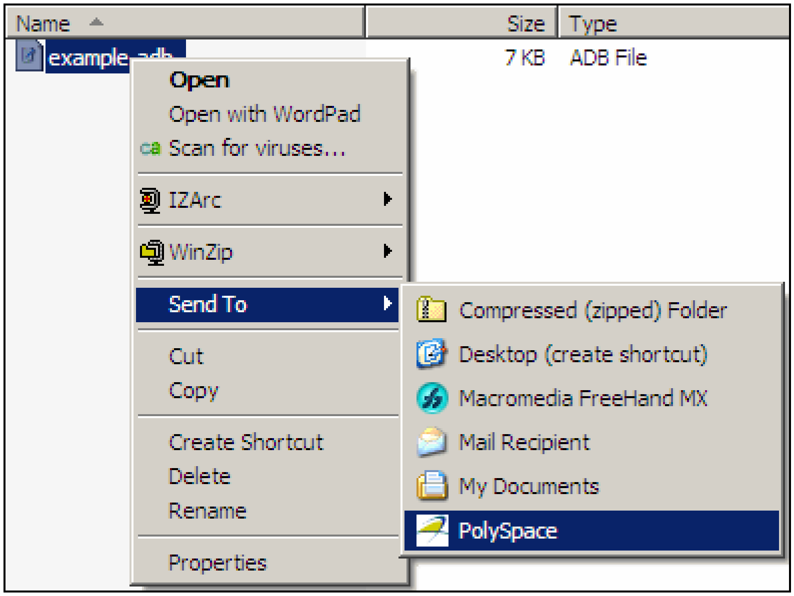
1 Navigate to the directory containing the source files you want to verify.

2 Right-click the file you want to verify.

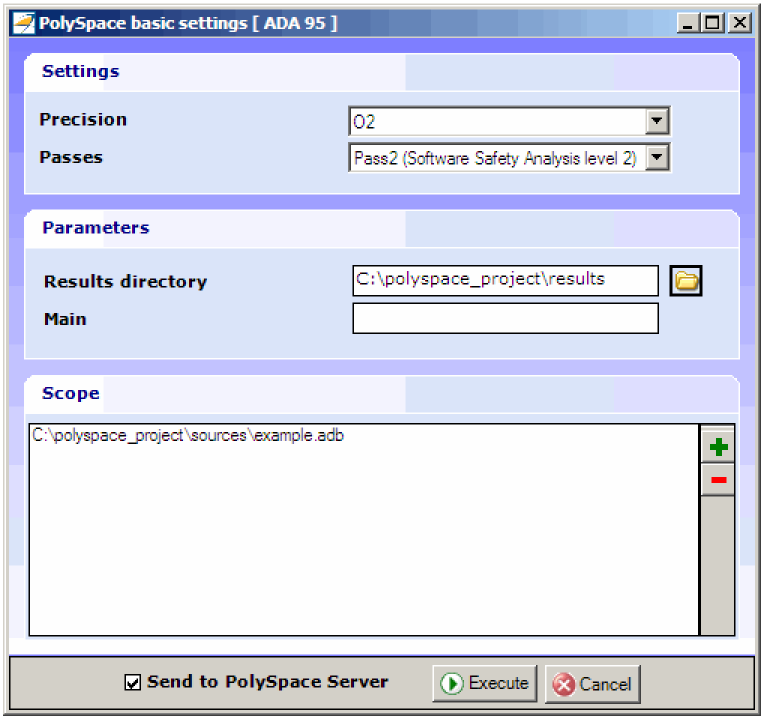
The context menu appears.



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.

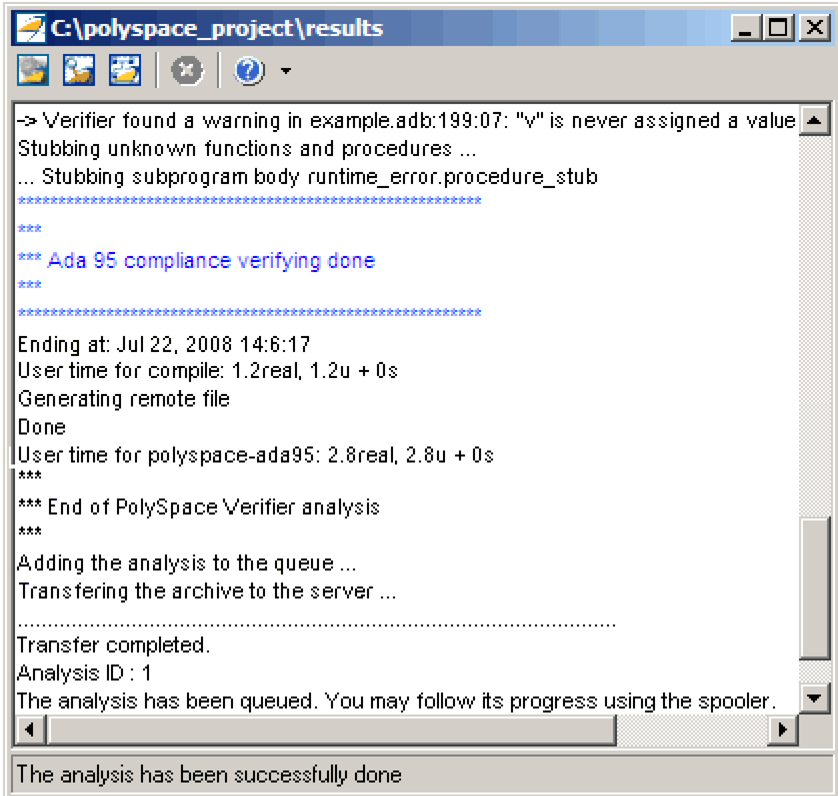
---

4 Enter the appropriate parameters for your verification.

5 Leave the default values for the other parameters.

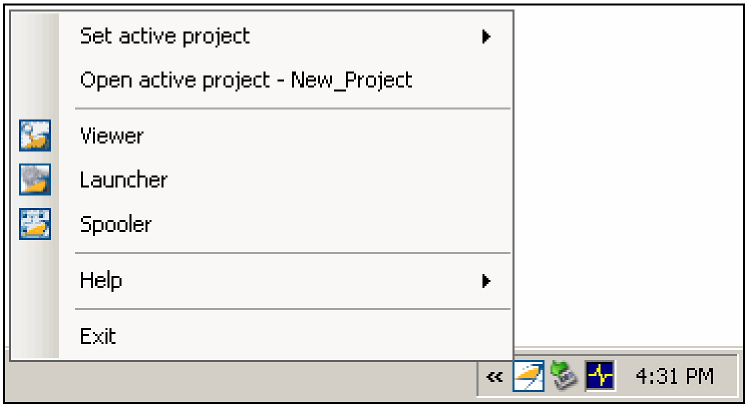
6 Click **Execute**.

The verification starts and the verification log appears.



Using the Taskbar Icon

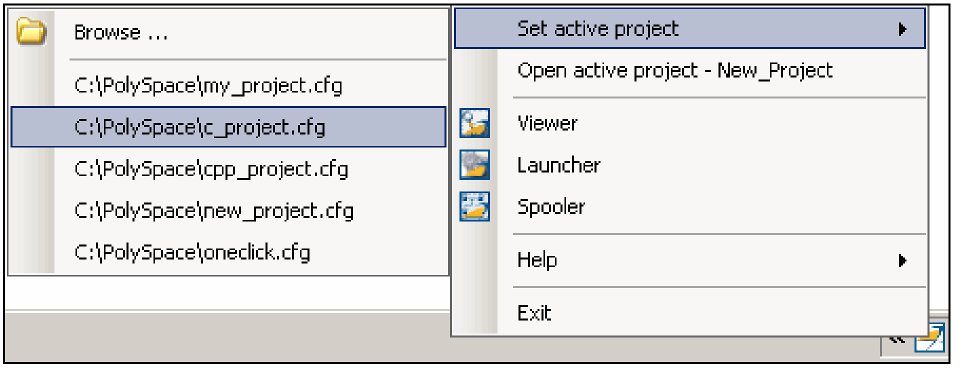
The PolySpace in One Click Taskbar icon allows you to access various software features.



Click the PolySpace Taskbar Icon, then select one of the following options:

- **Set active project** — Allows you to set the active configuration file. Before you start, you have to choose a PolySpace configuration file which contains the common options. You can choose a template of a previous project and move it to your working 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.

Glossary

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.

Index