Polyspace® Products for Ada

User's Guide

R2013**b**
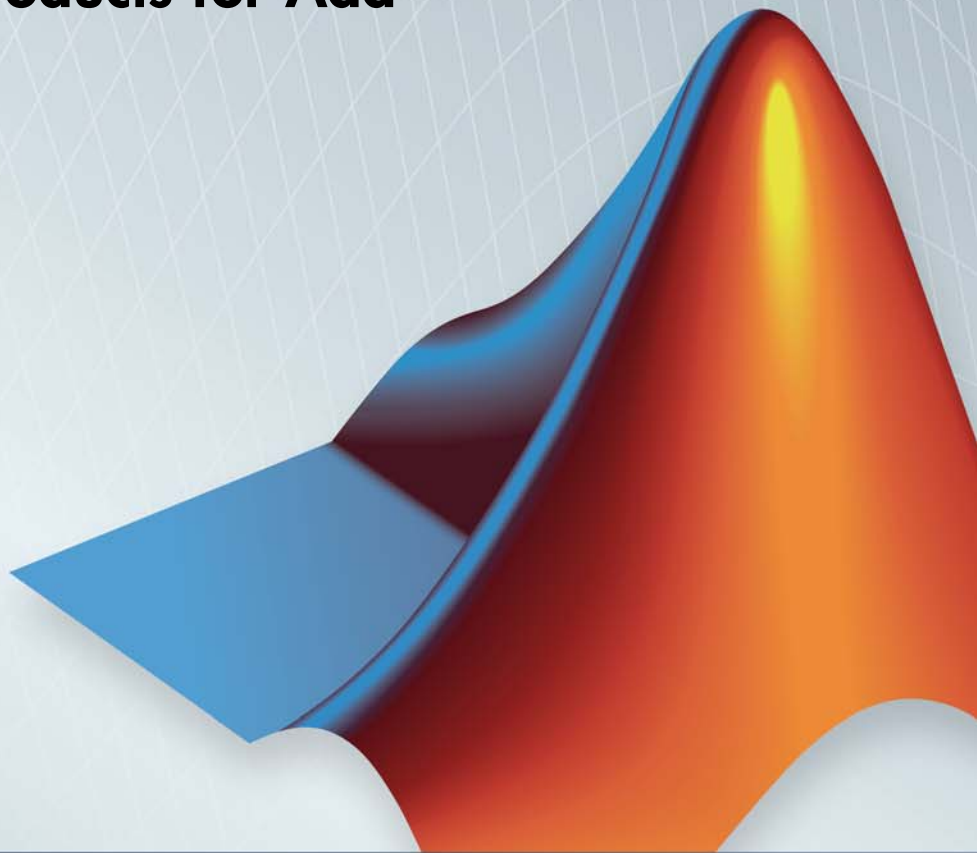
How to Contact MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Polyspace® Products for Ada User's Guide

© COPYRIGHT 1999–2013 by The MathWorks, Inc.

Trademarks

Patents

Revision History

Contents

Setting Up a Verification Project

3

Emulating Your Run-Time Environment

4

Preparing Source Code for Verification

5

Running a Verification

6

Troubleshooting Verification

7

Reviewing Verification Results

8

9

Software Quality with Polyspace Metrics

10

Verifying Code in the Eclipse IDE

11

Introduction to Polyspace Products

Product Description

In this section...
"Polyspace® Client™ for Ada" on page 1-2
"Polyspace® Server™ for Ada" on page 1-2

Polyspace Client for Ada

Prove the absence of run-time errors in source code

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

Key Features

- Verification of individual packages and package sets

- Formal method based abstract interpretation

- Display of run-time errors directly in code

- Eclipse™ IDE integration

Polyspace Server for Ada

Perform code verification on computer clusters and publish metrics

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

Key Features

- Web-based dashboard providing code metrics and quality status
- Automated job scheduling and e-mail notification
- Multi-server job queue manager
- Verification report generation
- Mixed operating system environment support

Overview of Polyspace Verification

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

To verify the source code, you set up verification parameters in a project, run the verification, and review the results. A graphical user interface helps you to efficiently review verification results. Results are color-coded:

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

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

The Value of Polyspace Verification

In this section...

Enhance Software Reliability

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

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

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

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

Decrease Development Time

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

You use Polyspace software to verify Ada source code before compile time. To verify the source code, you set up verification parameters in a project, run the verification, and review the results. This process takes significantly less

time than using manual methods or using tools that require you to modify code or run test cases.

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

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

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

Improve the Development Process

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

Polyspace verification software supports code verification throughout the development process:

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

How Polyspace Verification Works

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

What is Static Verification

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

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

For example, consider the following code:

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

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

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

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 domain variation of i is smaller than the range of tab. Only one check is required to establish that — and hence the gain in efficiency compared to traditional approaches.

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

Exhaustiveness

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

Product Components

The Polyspace products for verifying Ada code are:

In this section...
"Polyspace Verification Environment" on page 1-9
"Other Polyspace Components" on page 1-13

Polyspace Verification Environment

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

For Ada verification, you use two perspectives of the Polyspace verification environment:

- "Project Manager Perspective" on page 1-9
- "Results Manager Perspective" on page 1-11

Project Manager Perspective

The Project Manager perspective allows you to create projects, set verification parameters, and start verifications.
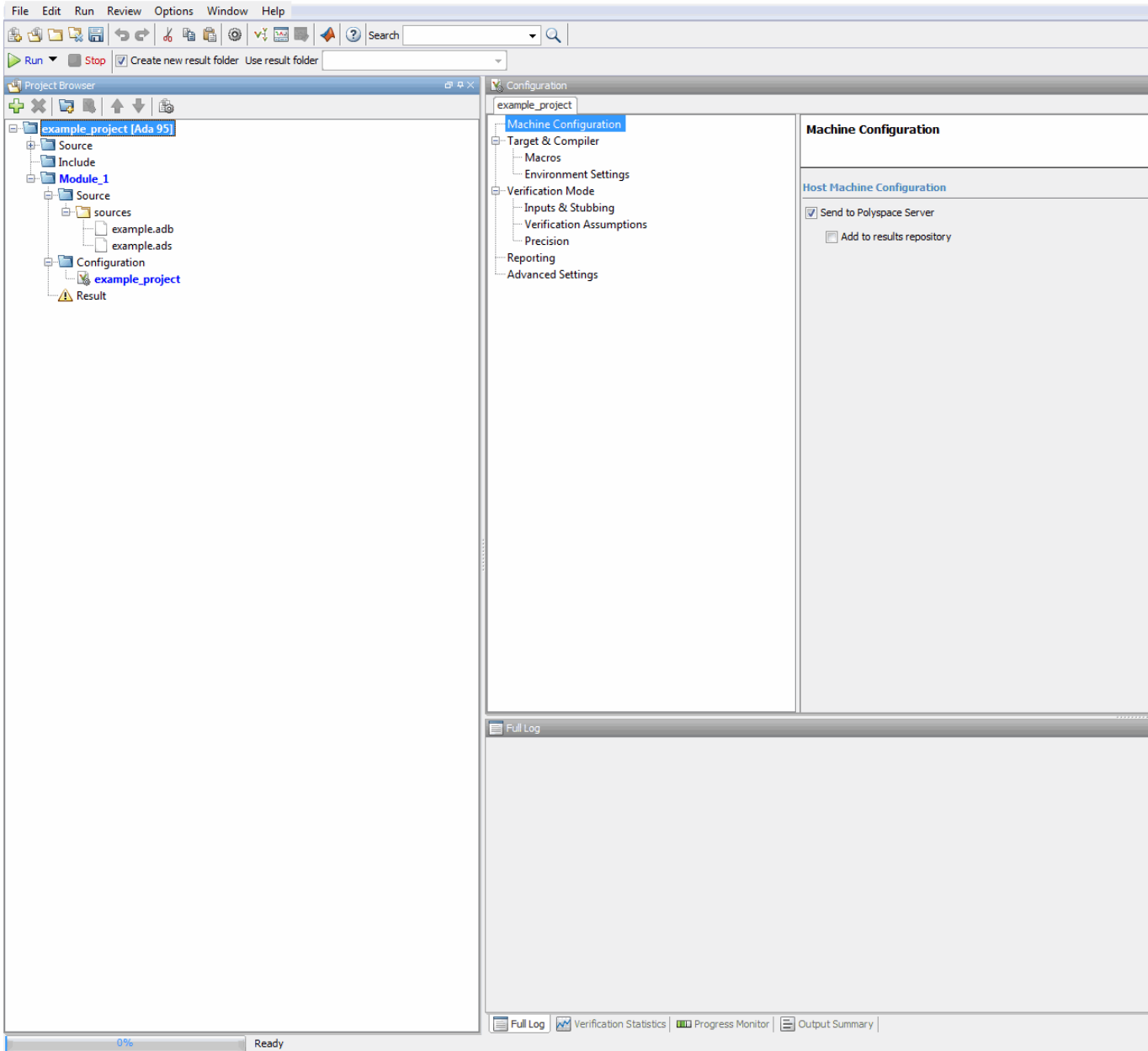
For information on using the Project Manager perspective, see "Project Configuration".

Results Manager Perspective

The Results Manager perspective allows you to review verification results, comment individual checks, and track review progress.

For information on using the Results Manager perspective, see "Run-Time Error Review".

Other Polyspace Components

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

- Polyspace Queue Manager Interface (Spooler)
- Polyspace Metrics Web Interface

Polyspace Queue Manager Interface (Polyspace Spooler)

The Polyspace Queue Manager (also called the Polyspace Spooler) is the graphical user interface of the Polyspace Server for Ada software. You use the Polyspace Queue Manager Interface to move jobs within the queue, remove jobs, monitor the progress of individual verifications, and download results.
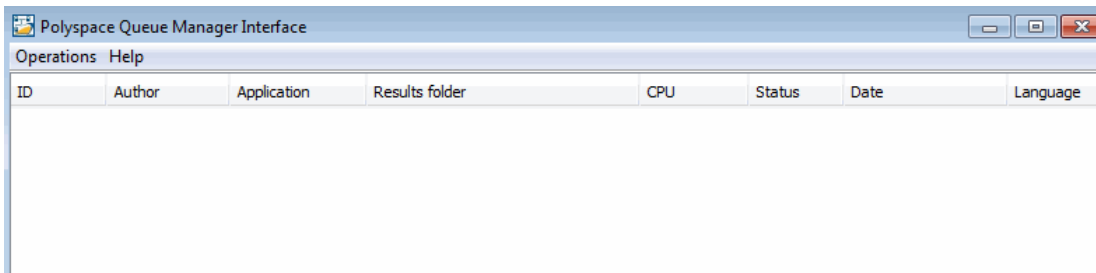
For information on using the Polyspace Queue Manager Interface, see "Verification Management".

Polyspace Metrics Web Interface

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

For information on using Polyspace Metrics, see "Quality Metrics".

Polyspace Documentation

In this section...
"About this Guide" on page 1-14
"MathWorks Online" on page 1-14

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.

MathWorks Online

For additional information and support, see:

`www.mathworks.com/products/polyspace`

How to Use Polyspace Software

Polyspace Verification and the Software Development Cycle

In this section...

Software Quality and Productivity

The goal of most software development teams is to maximize both quality and productivity. However, when developing software, there are always three related variables: cost, quality, and time.
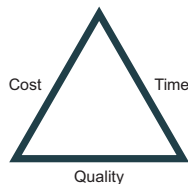
Changing the requirements for one of these variables always impacts the other two.

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

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

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

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

Best Practices for Verification Workflow

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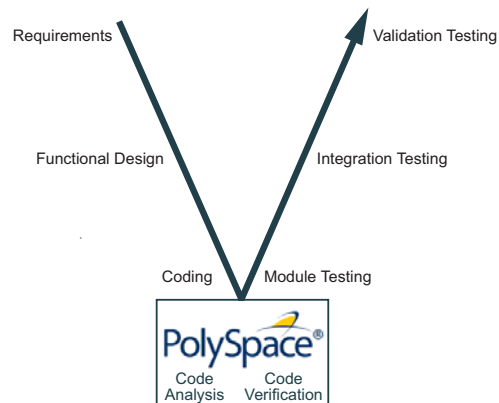
Polyspace® Verification in the Development Cycle

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

Using verification at this stage of the development cycle improves both quality and productivity, because it allows you to find and manage defects soon after the code is written. This saves time because each developer is familiar with their own code, and can quickly determine why the code contains defects. In addition, defects are cheaper to fix at this stage as they can be addressed before the code is integrated into a larger system.

Implementing a Process for Polyspace Verification

In this section...

Overview of the Polyspace Process

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

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

1 Define your quality goals.

2 Define a process to match your quality goals.

3 Apply the process to assess the quality of your code.

4 Improve the process.

Defining Quality Goals

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

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

Choosing Robustness or Contextual Verification

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

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

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

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

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

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

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

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

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

- Use Data Range Specifications (DRS) to specify the ranges for your variables, thereby limiting the verification to these cases. For more information, see "Specifying Data Ranges for Variables, Functions, and Procedures (Contextual Verification)" on page 4-8.

- Create a detailed main program to model the call sequence, instead of using the default main generator. For more information, see "Verifying an Application Without a Main" on page 4-5.

- Provide manual stubs that emulate the behavior of missing functions, instead of using the default automatic stubs. For more information, see "Stubbing" on page 5-2.

Choosing Strict or Permissive Verification

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

Options that make verification more strict include:

- **-strict**

Options that make verification more permissive include:

- **-permissive**

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

Defining Software Quality Levels

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

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

Software Quality Levels

Criteria	Software Quality Levels			
	QL1	QL2	QL3	QL4
Document static information	X	X	X	X
Review all red checks	X	X	X	X
Review all gray checks	X	X	X	X
Review first criteria level for orange checks		X	X	X
Review second criteria level for orange checks			X	X
Perform dataflow analysis			X	X
Review third criteria level for orange checks				X

In the example above, the quality criteria include:

- **Static Information** – Includes information about the application architecture, the structure of each module, and all files. Full verification of your application requires the documentation of static information.

- **Red checks** – Represent errors that occur every time the code is executed.

- **Gray checks** – Represent unreachable code.

- **Orange checks** – Indicate unproven code, meaning a run-time error may occur. .

- **Dataflow analysis** – Identifies errors such as non-initialized variables and variables that are written but never read. This can include inspection of:

  - Application call tree

  - Read/write accesses to global variables

  - Shared variables and their associated concurrent access protection

Defining a Verification Process to Meet Your Goals

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

These actions include:

- Setting standards for code development, such as coding rules.
- Setting Polyspace Analysis options to match your quality goals. See "Creating a Project" on page 3-2.
- Setting review criteria in the Polyspace Results Manager perspective so that results are reviewed consistently. See "Run-Time Error Review".

Applying Your Verification Process to Assess Code Quality

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

This process includes:

1 Running a Polyspace verification for each software component as it is written.

2 Reviewing verification results consistently. See "Review Checks Using Predefined Methodologies" on page 8-51.

3 Saving review comments for each component, so they are available for future review. See "Import and Export Review Comments" on page 8-81.

4 Performing additional verifications on each component, as defined by your quality goals.

Improving Your Verification Process

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

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

- Reassessing your quality goals.

- Changing your development process to produce code that is easier to verify.

- Changing Polyspace analysis options to improve the precision of the verification.

- Changing Polyspace options to change how verification results are reported.

For more information, see "Orange Check Management".

Sample Workflows for Polyspace Verification

Overview of Verification Workflows

Polyspace verification supports two goals at the same time:

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

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

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

Software Developers – Standard Development Process

User Description

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

- In Ada, no unit test tools or coverage tools are used – functional tests are performed just after coding.

- In C, either no coding rules are used, or rules are not followed consistently.

Quality

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

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

Verification Workflow

This process involves file-by-file verification immediately after coding, and again just before functional testing.
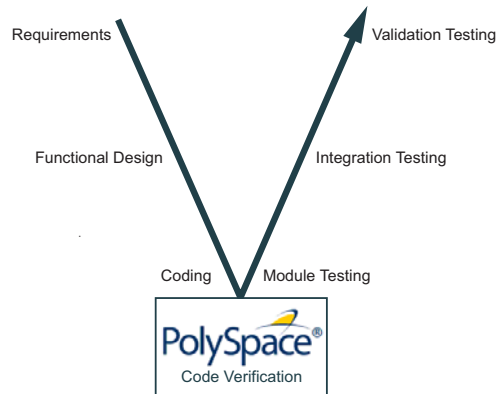
The verification workflow consists of the following steps:

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

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

2 Each developer performs file-by-file verification as they write code, and reviews verification results.

3 The developer fixes all **red** errors and examines **gray** code identified by the verification.

4 Until coding is complete, the developer repeats steps 2 and 3 as required.

5 Once a developer considers a file complete, they perform a final verification.

6 The developer fixes any **red** errors, examines **gray** code, and performs a selective orange review.

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

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

Costs and Benefits

When using verification to detect bugs:

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

- **Orange checks** — The time required to find one bug varies from 5 minutes to 1 hour, and is typically around 30 minutes. This represents an average of two minutes per orange check review, and a total of 20 orange checks per package in Ada and 60 orange checks per file in C.

Disadvantages to this approach:

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

Software Developers – Rigorous Development Process

User Description

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

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

Quality

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

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

Verification Workflow

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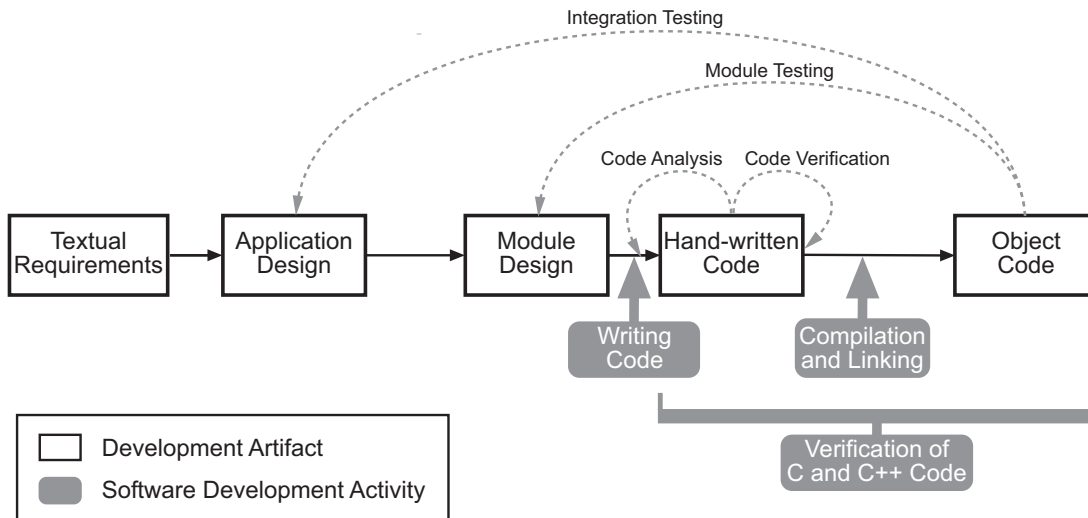
Workflow for Code Verification

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

The verification workflow consists of the following steps:

1 The project leader configures a Polyspace project to perform contextual
verification. This involves:

- Creates a "main" program to model call sequence, instead of using the
automatically generated main.

- Sets options to check the properties of some output variables. For
example, if a variable "y" is returned by a function in the file and should
always be returned with a value in the range 1 to 100, then Polyspace
can flag instances where that range of values might be breached.

2 The project leader configures the project to check the required coding rules.

3 Each developer performs file-by-file verification as they write code, and
reviews both coding rule violations and verification results.

4 The developer fixes any coding rule violations, fixes all **red** errors, examines **gray** code, and performs a selective orange review.

5 Until coding is complete, the developer repeats steps 2 and 3 as required.

6 Once a developer considers a file complete, they perform a final verification.

7 The developer performs an exhaustive orange review on the remaining orange checks.

Note The goal of the exhaustive orange review is to examine all orange checks that were not reviewed as part of previous reviews. This is possible when using coding rules because the total number of orange checks is reduced, and the remaining orange checks are likely to reveal problems with the code.

Optionally, an additional verification can be performed during the integration phase. The purpose of this additional verification is to track integration bugs, and review:

- Red and gray integration checks;
- The remaining orange checks with a selective review: *Integration bug tracking*.

Costs and Benefits

With this approach, Polyspace verification typically provides the following benefits:

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

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

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

- A selective orange review may reveal **one integration bug per hour**
  of code review.

- Selective orange review takes about 6 hours to complete. This is long
  enough to review orange checks throughout the whole application and
  represents a step towards an exhaustive orange check review. Spending
  more time is unlikely to be efficient.

- An exhaustive orange review takes between 4 and 6 days, assuming that
  50,000 lines of code contains approximately 400–800 orange checks.

Quality Engineers – Code Acceptance Criteria

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

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

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

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

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

Verification Workflow
This process usually involves both code analysis and code verification before validation phase, and thorough review of verification results based on defined quality goals.
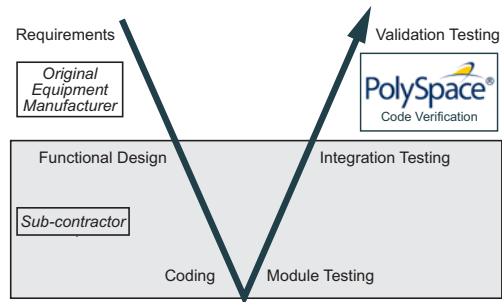
> **Note** Verification is often performed multiple times, as multiple versions of the software are delivered.

The verification workflow consists of the following steps:

1 Quality engineering group defines clear quality goals for the code to be written, including specific quality levels for each version of the code to be delivered (first, intermediate, or final delivery) For more information, see "Defining Quality Goals" on page 2-5.

2 Development group writes code according to established standards.

3 Development group delivers software to the quality engineering group.

4 The project leader configures the Polyspace project to meet the defined quality goals, as described in "Defining a Verification Process to Meet Your Goals" on page 2-9.

5 Quality engineers perform verification on the code.

6 Quality engineers review all **red** errors, **gray** code, and the number of orange checks defined in the process.

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

7 Quality engineers create reports documenting the results of the verification, and communicate those results to the supplier.

8 Quality engineers repeat steps 5–7 for each version of the code delivered.

Costs and Benefits

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

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

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

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

Project Managers — Integrating Polyspace Verification with Configuration Management Tools

User Description

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

Quality

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

Verification Workflow

The verification workflow consists of the following steps:

1 Project manager defines quality goals, including individual quality levels for each stage of the development cycle.

2 Project leader configures a Polyspace project to meet quality goals.

3 Developers run verification at the following stages:

- **Daily check-in** — On the files currently under development. Compilation must complete without the permissive option.

- **Pre-unit test check-in** — On the files currently under development.

- **Pre-integration test check-in** — On the whole project, ensuring that compilation can complete without the permissive option. This stage differs from daily check-in because link errors are highlighted.

- **Pre-build for integration test check-in** — On the whole project, with all multitasking aspects accounted for as required.

- **Pre-peer review check-in** — On the whole project, with all multitasking aspects accounted for as required.

4 Developers review verification results for each check-in activity to confirm that the code meets the required quality level. For example, the transition criterion could be: "No bug found within 20 minutes of selective orange review"

3

Setting Up a Verification Project

Creating a Project

What Is a Project?

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

- Source files

- Include folders

- One or more configurations, specifying a set of analysis options

- One or more modules, each of which include:

- Source (specific versions of source files used in the verification)
- Configuration (specific set of analysis options used for the verification)
- Verification results

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

What Is a Project Template?

A Project Template (Compilation Environment Template), is a predefined set of analysis options configured for a specific compilation environment.

When you create a new project, you can select a project template to automatically set analysis options for your compiler, and help locate required include folders.

Polyspace software provides predefined templates for common compilers such as Aonix, Gnat, Greenhills, and Rational.

You can also create custom templates from existing project configurations. For more information, see "Creating Custom Project Templates" on page 3-17.

Project Folders

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 folder, and then in that folder, create separate folders for the source files, include files, and results. For example:

polyspace_project/

- sources
- includes
- results

Opening Polyspace Verification Environment

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

To open the Polyspace verification environment:

1 Double-click the Polyspace icon.

2 If you have both Polyspace for C/C++ and Polyspace for Ada products on your system, the **Polyspace Language Selection** dialog box opens.
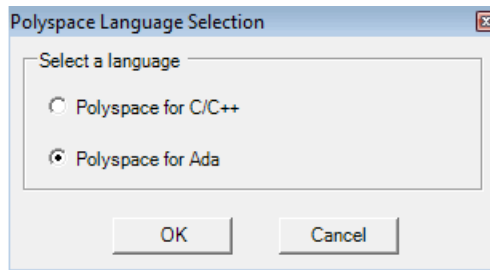
Select **Polyspace for Ada**. Then click **OK**.

The Polyspace verification environment opens:
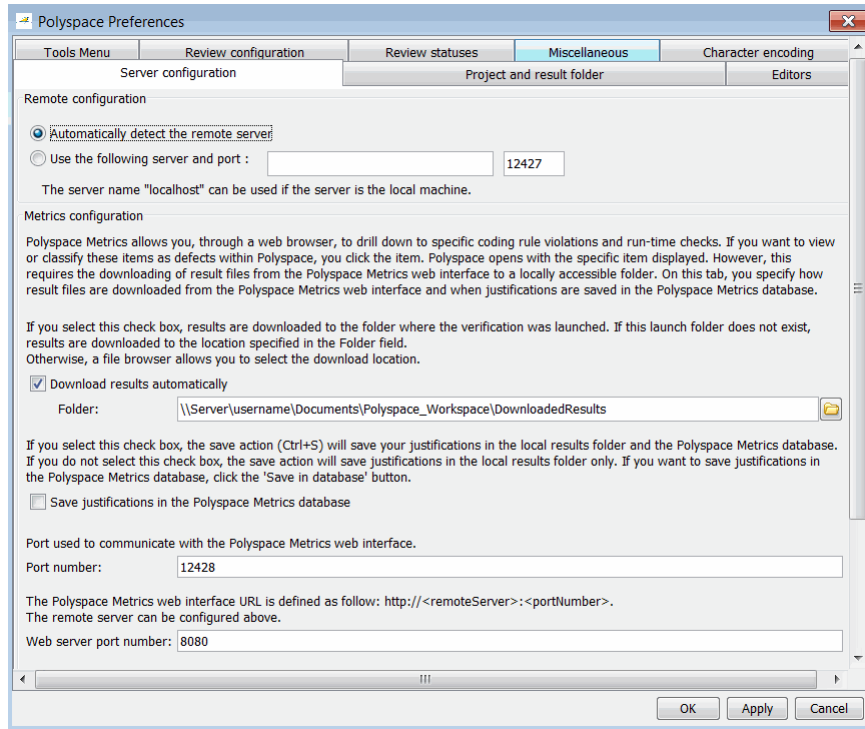
By default, the Polyspace verification environment displays the Project Manager perspective. The Project Manager perspective has three main sections.

Use this section...	For...
Project Browser (upper-left)	Specifying: • Source files • Include folders

Use this section...	For...
	• Results folder
Configuration (upper-right)	Specifying verification options
Output (lower-right)	Monitoring the progress of a verification, and viewing status, log messages, and general verification statistics.

You can resize or hide any of these sections.

Creating New Projects

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

To create a new project:

1 Select **File > New Project**. The Project - Properties dialog box opens.

2 In the **Project name** field, enter a name for your project.

3 If you want to specify a location for your project, clear the **Use default location** check box, and enter a **Location** for your project.

Note You can update the default project location. Select **Options > Preferences**, which opens the Polyspace Preferences dialog box. On the **Project and result folder** tab, in the **Define default project location** field, specify the new default location.

---

4 In the Project language section, select either **Ada 83** or **Ada 95**.
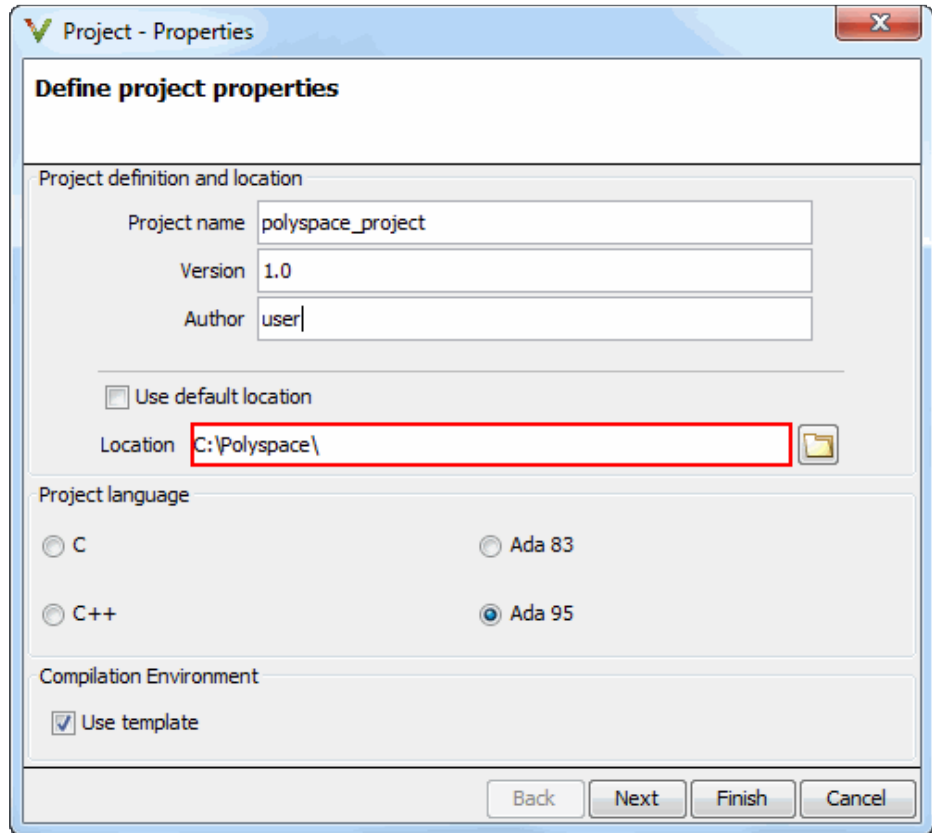
5 Click **Next**.

The Project – Browse for a Project Template dialog box opens.
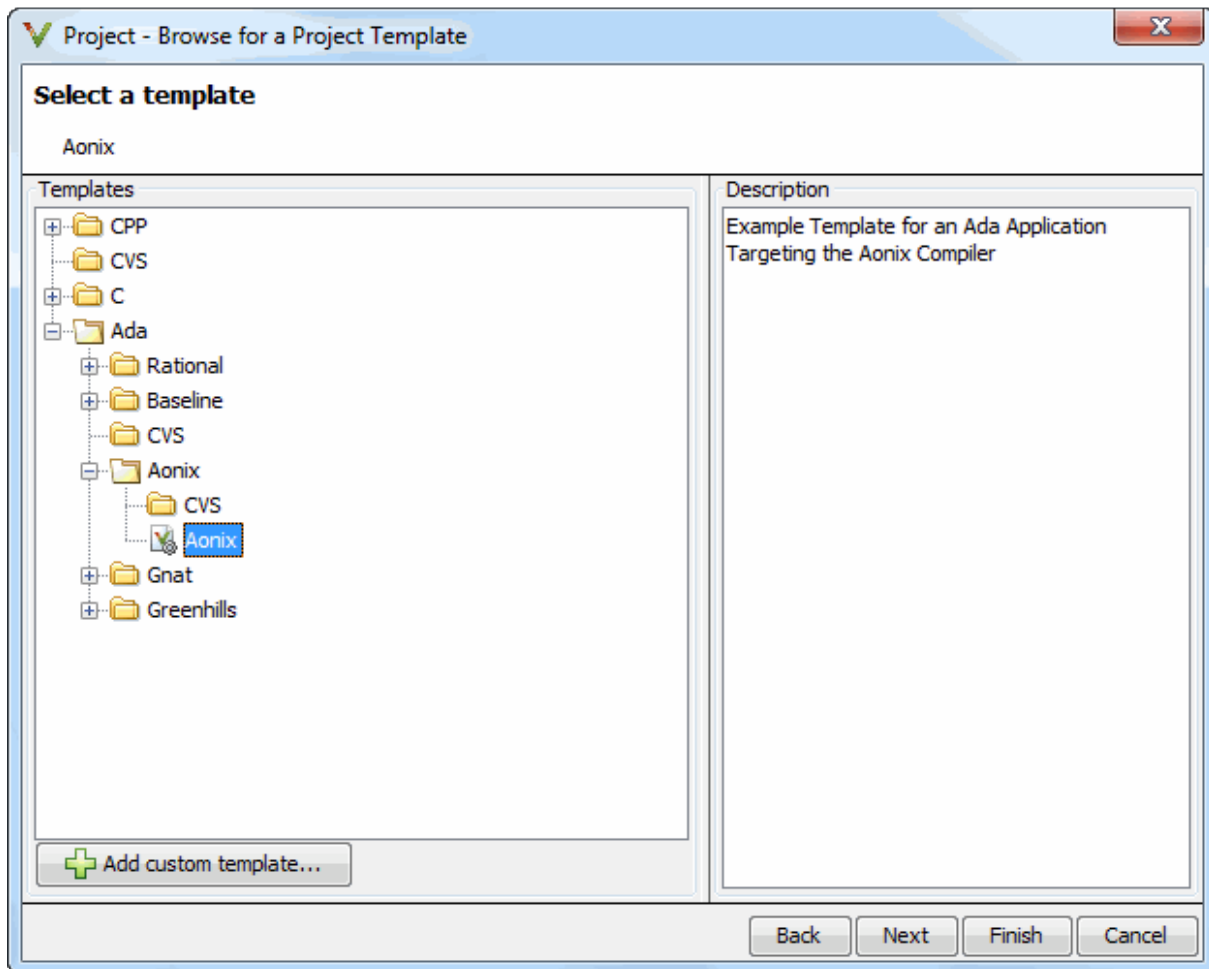
6 Select the Template for your compiler (for more information, see "What Is a Project Template?" on page 3-3).

If your compiler does not appear in the list of predefined templates, select **Baseline** to start with a generic Ada95 template.

Note If you do not want to use a template, click **Back**, then clear the **Use template** check box.

7 Click **Next**. The Project – Add Source Files and Include Folders dialog box opens.
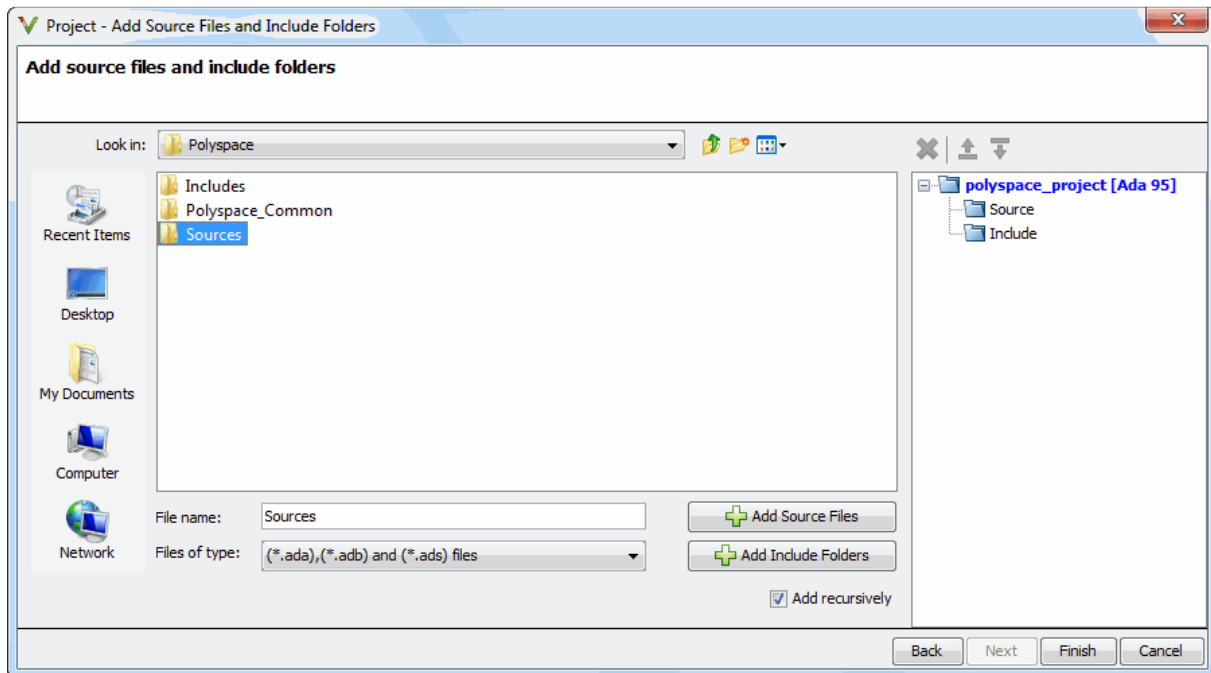
8 The project folder location that you specified in step 3 should appear in the **Look in** field. Otherwise, navigate to that folder.

9 Select the source files that you want to include in the project. Then click **Add Source Files**. The source files appear in the Source tree for your project.

10 Select the Include folders that you want to include in the project. Then click **Add Include Folders**. The include folders appear in the Include tree for your project.

11 Click **Finish**. The new project opens in the Polyspace verification environment, with default options based on the project template you selected.
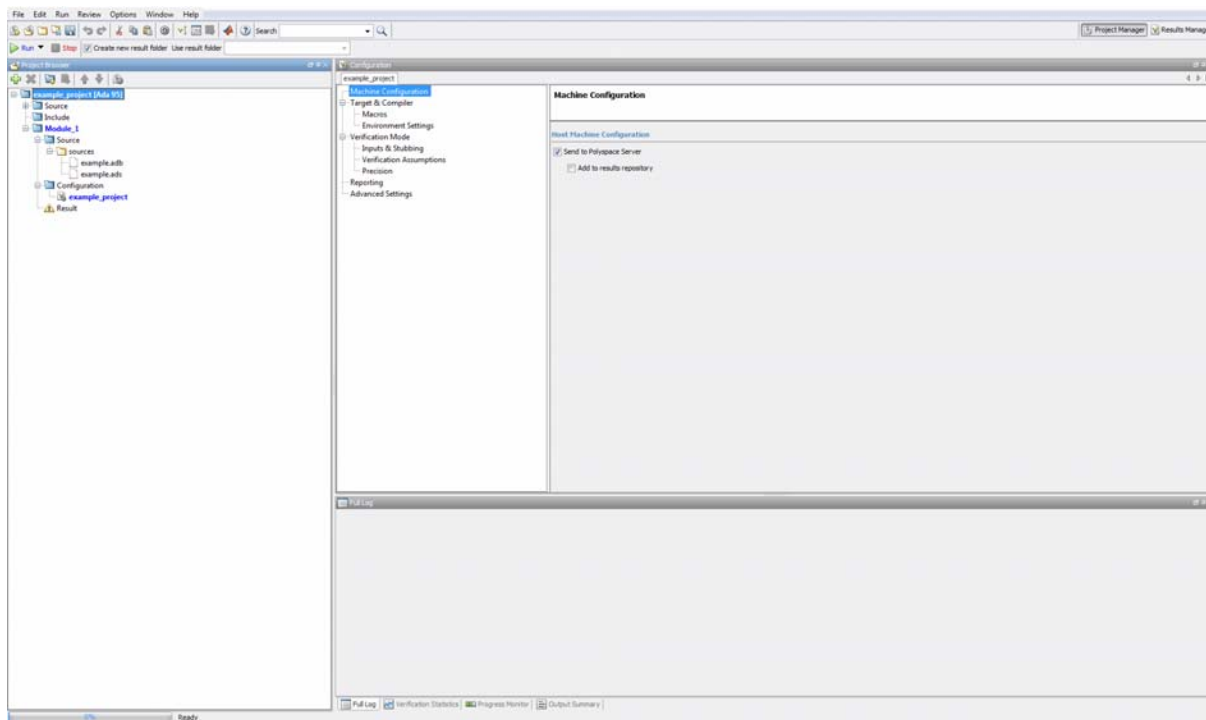
12 To save your project, click the Save button or enter **Ctrl+S**.

Opening Existing Projects

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

To open an existing project:

1 Select **File > Open Project**.

The **Open Polyspace Project** dialog box opens.

2 Select the project you want to open. Then click **Open**.

The selected project opens in the Project Manager perspective.
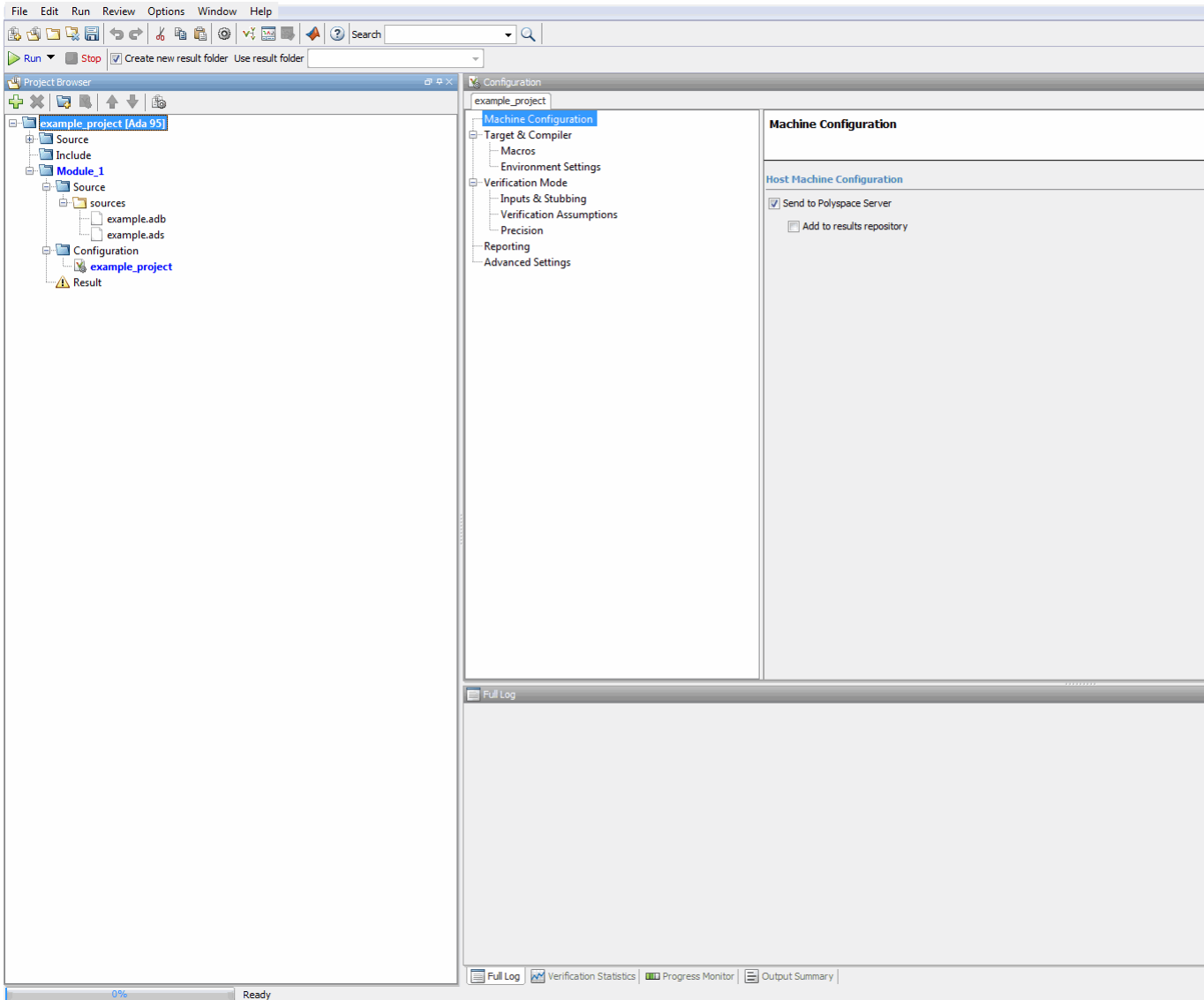
Closing Existing Projects

In the Polyspace verification environment, you can manage multiple projects simultaneously. When you create a new project or open an existing project, the project is added to the Project Browser tree. To remove a project from the Project Browser tree, you must close the project.

To close a project:

1 In the Project Browser, select the project you want to close.

2 Right-click the project. From the context menu, select **Close Active Project**.

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

Specifying Source Files

To specify the source files for your project:

1 In **Project Browser**, select the **Source** folder.

2 On the **Project Browser** toolbar, click the icon The Project - Add Source Files and Include Folders dialog box opens.

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

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

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

5 Click **Finish** to apply the changes and close the dialog box. The source files that you selected appear in the Project Browser.

6 Save your project (**Ctrl+S**).

Specifying Include Folders

To specify the include folders for the project:

1 In **Project Browser**, select the **Include** folder.

2 On the **Project Browser** toolbar, click the icon .

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

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

4 Select the include folders that you want to include in the project. Then click **Add Include Folders**. The Include folders appear in the Include tree for your project.

5 Click **Finish** to apply the changes and close the dialog box. The Include folders that you selected appear in **Project Browser**.

6 Save your project (**Ctrl+S**).

Managing Include File Sequence

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

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

1 In the Project Browser, expand the **Include** folder.

2 Select the include folder you want to move.

3 On the Project Browser toolbar, click the **Move up** or **Move down** icons to move the include folder. The Include folders are reordered in the Project Browser.

Creating Multiple Modules

Each Polyspace project can contain multiple modules. You can use each of these modules to verify a specific set of source files with a specific set of analysis options.

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

To create a new module in your project:

1 In the Project Browser, select any project.

2 On the Project Browser toolbar, click the Create a new module icon . A
   second module, Module_2, appears in the Project Browser tree.

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

   The source files appear in the Source tree of Module_2.
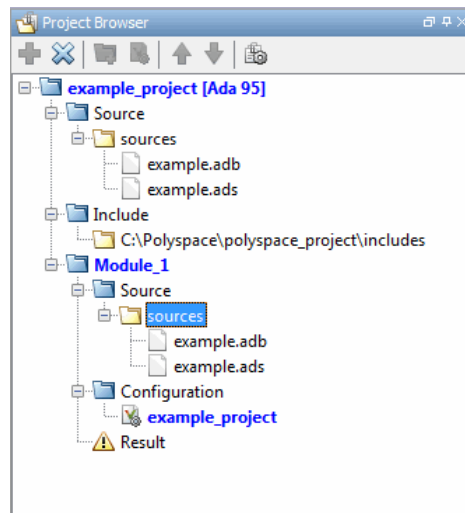
Creating Multiple Analysis Option Configurations

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

To create a new configuration in your project:

1 In the Project Browser, select any module.

2 Right-click the Configuration folder in the module, and select **Create New Configuration**. The new configuration appears in the Project Browser.
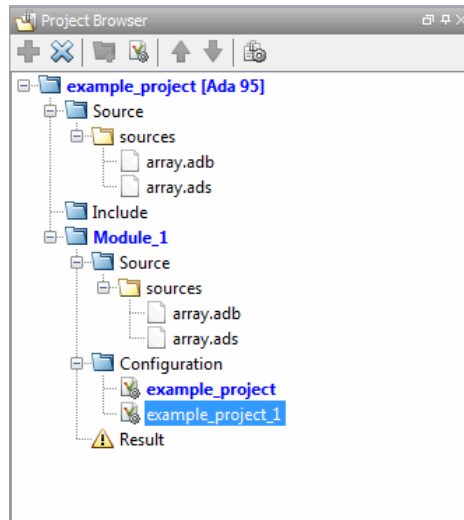
3 In the **Configuration** pane, specify the analysis options for the configuration.

4 To save your project with the new settings, select **File > Save**.

For detailed information about specific analysis options, see "Analysis Options".

Specify Parent Folder for Results

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

1 From the Polyspace verification environment toolbar, select **Options > Preferences**.

2 On the **Project and result folder** tab, select the **Create new result folder check box**.

3 In the **Define parent result folder location** field, specify the location that you want.

4 If you require a subfolder, select the **Add a sub folder using the project name** check box. This subfolder takes the name of the project.

5 If required, specify additional formatting options for the folder name . The options allow you to incorporate the following information into the name of the results folder:

- **Result folder prefix** — A string that you define. Default is `Result`.

- **Project Variable** — Project, module, and configuration.

- **Date Format** — Date of verification

- **Time Format** — Time of verification

- **Counter** — Count value that automatically increments by one with each verification

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

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

Specifying Analysis Options

The **Configuration** pane in the Project Manager perspective allows you to set **Analysis options** that Polyspace software uses during the verification process.

To specify analysis options for your project:

1 In the **Configuration** tree view, select the required node. For example, for precision options, select **Configuration > Precision**. You see the **Precision** pane.
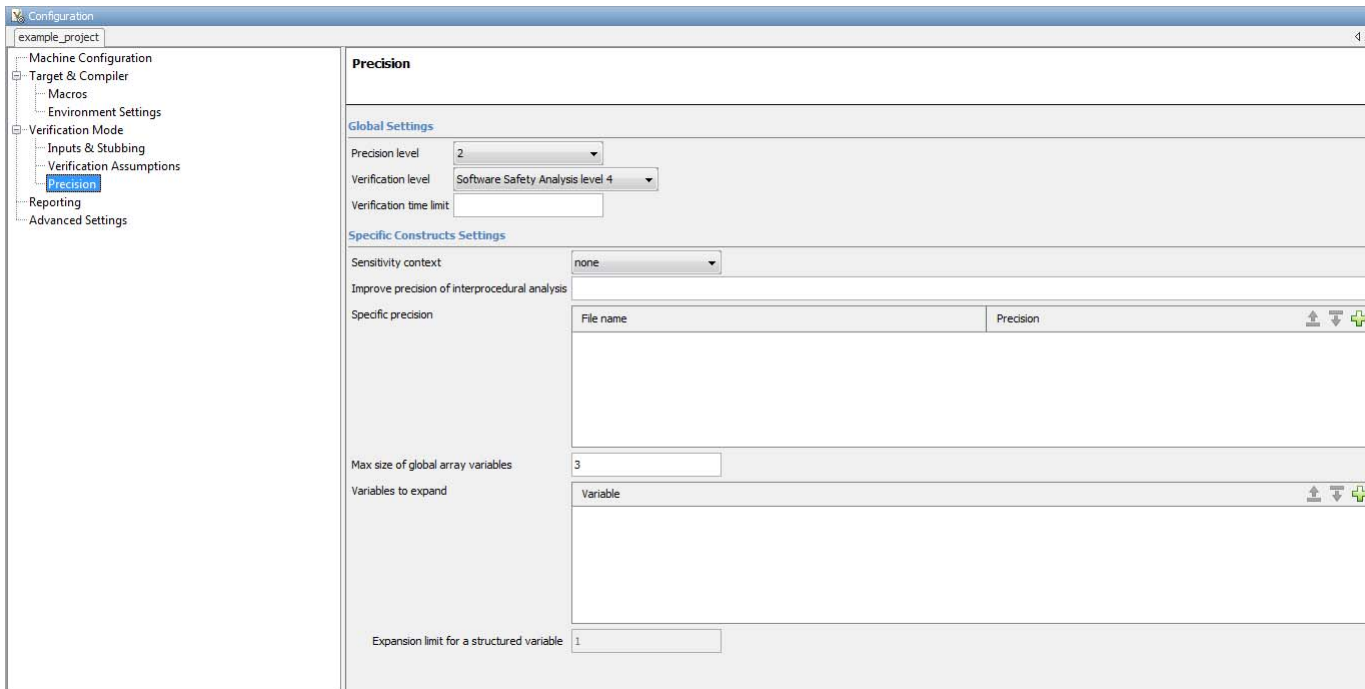
2 On the pane, specify the required options for your project.

3 To save your project with the new settings, select **File > Save**.

For detailed information about specific analysis options, see "Analysis Options".

Creating Custom Project Templates

Once you have configured a project, you can save the Configuration as a custom Project Template, and use it to configure future projects. Using custom templates allows you to automatically set up the compilation environment, include folders, and other analysis options for your projects.

You can include any Analysis option in the template, but the following options should typically be included:

- Target processor type (-target)

- Operating system target for Standard Libraries (`-OS-target`)

- Defined Preporcessor Macros (`-D`)

To create a custom project template:

1 Open the project you want to use a template.

2 Right-click the configuration you want to use, and select **Export As Template**.
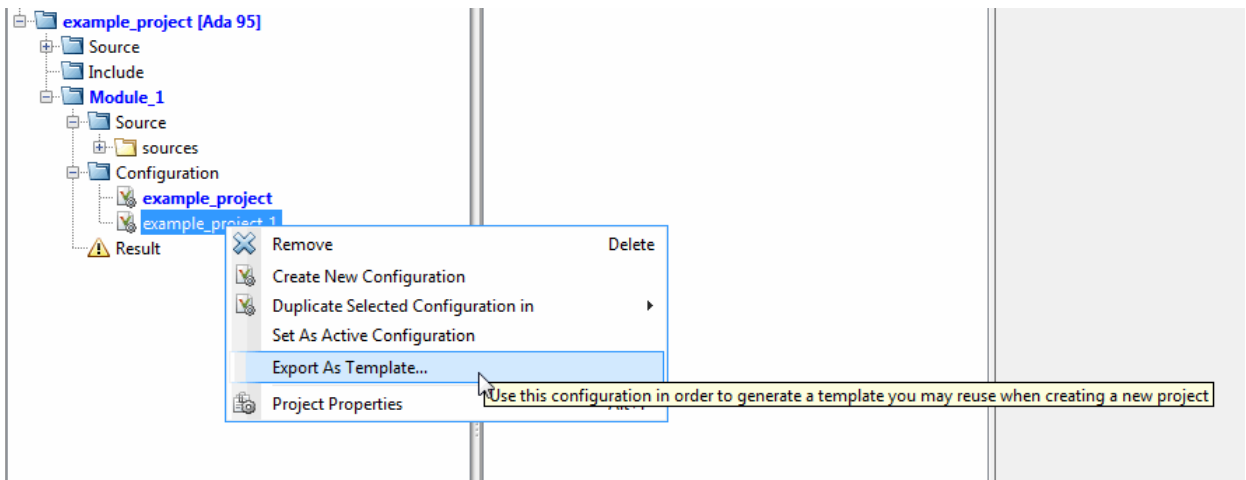
3 Enter a description for the template, then click **Proceed**.
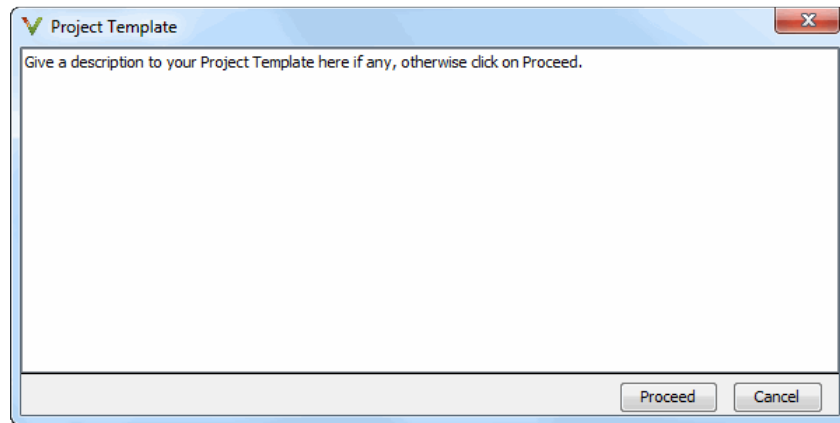
4 Save your Compilation Environment Template file with the name you want to appear in the Templates browser.

5 Create a new project.

6 In the Project – Browse for a Project Template dialog box, select **Add custom template**.
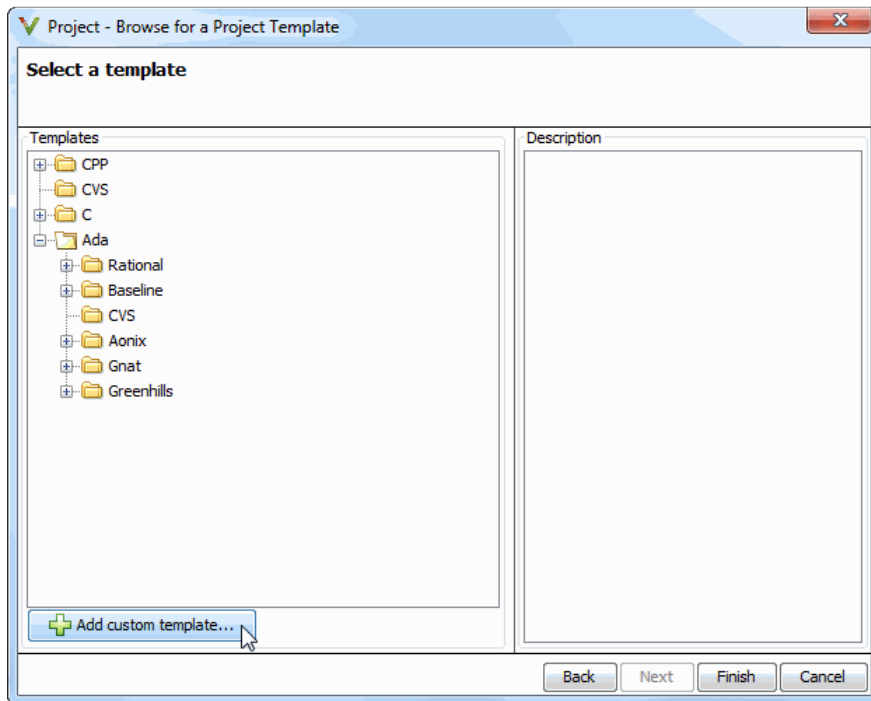
7 Select the template that you exported, then click **Open**.

8 The new template appears in the **Custom templates** folder of the Templates browser, and can be used for future projects.
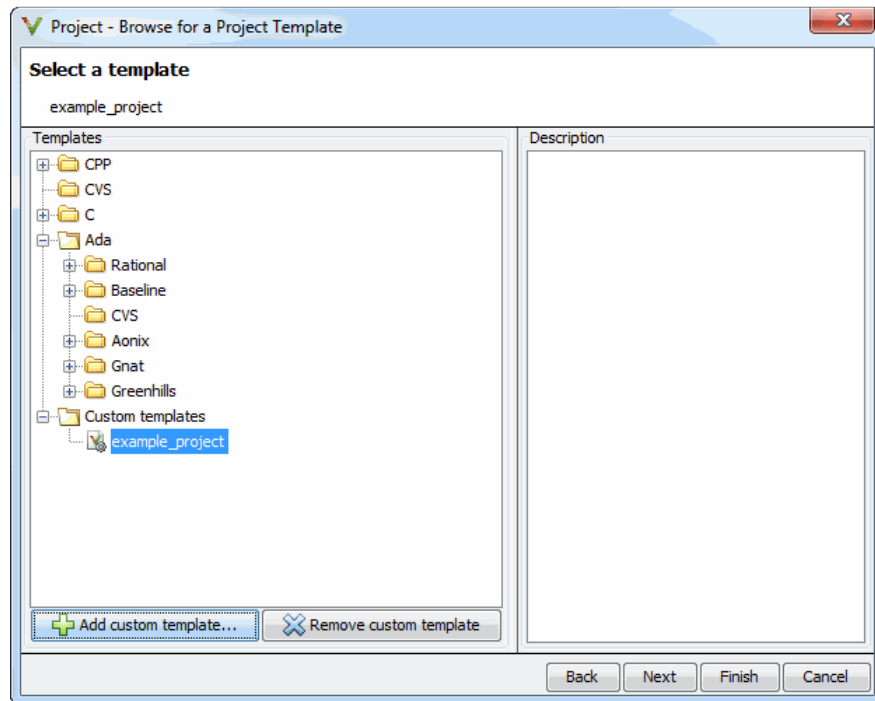
Configuring Text Editor

Before you run a verification, you should configure preferences for your text editor. With your text editor, you can view source files and reports directly from the Polyspace environment.

To configure your text editor:

1 Select **Options > Preferences**.

2 From the Polyspace Preferences dialog box, click the **Editors** tab.

3 In the **Text Editor** field, specify your text editor. For example:

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

4 To automatically specify the command line arguments for your editor, from the **Arguments** drop-down list, select your text editor.

3-21

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

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

5 Click **OK**.

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

1 In the **Text Editor** field, enter /usr/bin/xterm.

2 From the **Arguments** drop-down list, select Custom.

3 In the field to the right, enter -e /usr/bin/vi $FILE.

Saving the Project

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

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

Specifying Options to Match Your Quality Goals

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

This includes:

In this section...
"Quality Goals Overview" on page 3-23
"Choosing Contextual Verification Options" on page 3-23
"Choosing Strict or Permissive Verification Options" on page 3-24

Quality Goals Overview

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

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

Note For information on defining the quality goals for your project, see "Defining Quality Goals" on page 2-5.

Choosing Contextual Verification Options

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

For more information on robustness and contextual verification, see "Defining Quality Goals" on page 2-5.

Note If you are aware of run-time errors in your code but still want to run a verification, you can annotate your code so that these known errors are highlighted in the Results Manager perspective. For more information, see "Highlighting Known Run-Time Errors" on page 5-28.

To specify contextual verification for your project:

1 In the Project Manager perspective, on the **ConfigurationVerification Mode** pane, select **Verify module**.

2 In the **Verification Mode > Inputs & Stubbing > Variable/function range setup** field, specify a data range specification (DRS) file.

3 On the **Verification Mode > Inputs & Stubbing** pane, control stubbing behavior with the following options:

- **No automatic stubbing** — If you select this option, the software does not automatically stub functions. The software lists the functions to be stubbed and stops the verification.

- **Initialization of uninitialized global variables** — Specify how uninitialized global variables are initialized.

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

Choosing Strict or Permissive Verification Options

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

Note If you are aware of run-time errors in your code but still want to run a verification, you can annotate your code so that these known errors are highlighted in the Results Manager perspective. For more information, see "Highlighting Known Run-Time Errors" on page 5-28.

For strict verification, on the **Verification Mode > Inputs & Stubbing** pane, select the **No automatic stubbing** check box.

For permissive verification:

1 On the **Verification Mode > Inputs & Stubbing** pane, clear the **No automatic stubbing** check box

2 On the **Verification Assumptions** pane, select the **Continue with non-initialized in/out parameters** check box.

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

Setting Up Project to Generate Metrics

About Polyspace Metrics

Polyspace Metrics is a Web-based tool for software development managers, quality assurance engineers, and software developers. In software projects, this tool enables you to do the following :

- Evaluate software quality metrics

- Monitor the variation of code metrics and run-time checks over the lifecycle of a project

- View defect numbers, run-time reliability of the software, review progress, and the status of the code with respect to software quality objectives

For information on using Polyspace Metrics, see "Quality Metrics".

Enabling Polyspace Metrics

1 On the **Configuration > Machine Configuration** pane:

    a Select the **Send to Polyspace Server** check box.

    b Select the **Add to results repository** check box.

The software generates Polyspace Metrics for the next verification.

Specifying Automatic Verification

You can configure verifications to start automatically and periodically, for example, at a specific time every night. At the end of each verification, the software stores results in the repository and updates the project metrics.

You can also configure the software to send you an email at the end of the verification.

For more information, see "Specifying Automatic Verification" on page 10-3.

4

Emulating Your Run-Time Environment

Setting Up a Target

In this section...
"Target & Compiler Overview" on page 4-2
"Specifying Target & Compiler Parameters" on page 4-2
"Predefined Target Processor Specifications" on page 4-3

Target & Compiler Overview

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

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

For detailed information on each **Target & Compiler** option, see "Target & Compiler".

Specifying Target & Compiler Parameters

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

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

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

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

For detailed specifications of each predefined target processor, see "Predefined Target Processor Specifications" on page 4-3.

For information about target and compiler options, see "Target & Compiler".

Predefined Target Processor Specifications

Polyspace software supports many processors. To specify a predefined processor, from the **Configuration > Target & Compiler > Target processor type** drop-down list, select the processor.

If the your processor is not specified in the drop-down list, use the following table to select a processor that shares the same characteristics as your processor.

Target	sparc	m68k ColdFire	1750a	powerpc 32bit	powerpc 64bit	I386
Character	8	8	16	8	8	8
short_integer	16	16	16	16	16	16
Integer	32	32	16	32	32	32
long_integer	32	32	32	32	64	32
long_long_integer	64	64	64	64	64	64
short_float	32	32	32	32	32	32
Float	32	32	32	32	32	32
long_float	64	64	48	64	64	64
long_long_float	64	64	48	64	64	64

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

- `powerpc32bits` — 64.

- `powerpc64bits` — 64.

- `i386` — 32.

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

```
with TEXT_IO;
procedure TEMP is
```

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

Verifying an Application Without a Main

In this section...
"Main Generator Overview" on page 4-5
"Automatically Generating a Main" on page 4-5
"Manually Generating a Main" on page 4-6
"How Polyspace Verifies Generic Packages" on page 4-7

Main Generator Overview

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

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

Automatically Generating a Main

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

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

1 On the **Configuration > Verification Mode** pane, click **Verify whole application**.

2 In the **Main entry point** field, specify your file that defines the main, for example, INIT.MAIN.

With Polyspace Server for Ada, the software sets the -main option by default. You can choose to automatically generate a main using the -main-generator option. On the **Configuration > Verification Mode** pane, click **Verify module**.

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

Manually Generating a Main

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

To manually define the main:

1 Identify the API functions and extract their declaration.

2 Create a main containing declarations of a volatile variable for each type that is listed in the function prototypes.

3 Create a loop with a volatile end condition.

4 Inside this loop, create a switch block with a volatile condition.

5 For each API function, create a case branch that calls the function using the volatile variable parameters that you created.

The following code shows the five steps:

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

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

  --Step 3: Create loop
```

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

How Polyspace Verifies Generic Packages

Consider the following code, which instantiates a generic package.

```
with Ada.Numerics.Generic_Elementary_Functions;

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

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

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

end Test;
```

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

Specifying Data Ranges for Variables, Functions, and Procedures (Contextual Verification)

In this section...
"Overview of Data Range Specifications (DRS)" on page 4-8
"Specifying Data Ranges Using Text Files" on page 4-8
"Performing Efficient Module Testing with DRS" on page 4-12
"Reducing Orange Checks with DRS" on page 4-13

Overview of Data Range Specifications (DRS)

By default, Polyspace software performs *robustness verification*, proving that the software works under all conditions. Robustness verification assumes that all data inputs are set to their full range. Therefore, nearly any operation on these inputs could produce an overflow.

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

You can use DRS to set constraints on:

- Global variables
- Stubbed functions and procedures
- Function call input values

Note Data ranges are applied during verification level 2 (`pass2`).

Specifying Data Ranges Using Text Files

To use the DRS feature, you must specify a file that constrains the range of values for global variables, values returned by stubbed functions, out

or `in/out` parameters of stubbed procedures, or input parameters of user subprograms called by the main generator during verification. See "DRS Text File Format" on page 4-9.

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

1 In Project Manager perspective, select the **Configuration > Verification Mode > Inputs & Stubbing** pane.

2 To the right of the **Variable/function range setup** row, click 📁. The Load a DRS file dialog box opens.

3 Use this dialog box to navigate to the folder that contains your DRS text file.

4 In the **File name** field, specify your DRS file.

5 Click **Open**. You see the file path in the **Variable/function range setup** field.

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

DRS Text File Format

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

Each line of the DRS file must have the following format:

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

- *var_func_param* — A variable name, the name of a function that returns a value, or a subprogram parameter name.

- *min_val*, *max_val* — Constants that specify minimum and maximum range values. Data type of these values can be character, enumerator, integer, or float. The integer or float values may be binary, octal, decimal, or hexadecimal.

- reinit — Sets global variables to the specified range at the entry point for each subprogram called by the main generator, or the entry point for the user-defined main subprogram.

- init — Initializes subprogram input parameters to a specified range when the subprogram is called by the main generator.

- permanent — Sets the return, out, or in/out parameters to the specified range of a stubbed subprogram each time the subprogram is called.

Tips for Creating DRS Text Files

- You can replace *min_val* and *max_val* by the words "min" or "max". In this case, the software uses the corresponding minimum and maximum value for the declared data subtype (true even for an enumeration type that has enumerated values min and max). For example, with a SPARC® processor, the minimum value for the integer data type is -2^31 and the maximum value is 2^31-1.

- You can use tab, comma, space, or semicolon as column separators.

- You can apply data range specification to variables and subprograms declared within a package specification or body, or subprograms outside a package. For subprograms outside a package, use the subprogram name as package name.

- You cannot apply data range specification to:
  - Local subprograms or task entries
  - Constant qualified variables, record discriminants, variables of access type, or variables defined in a protected type or task type

Example DRS Text File

The following lines:

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

```
P.proc.i -1.0 1.0 init  # main generator calls the user
            # procedure proc with the parameter
            # i initialized to [-1.0;1.0]
dummy_f.dummy_f -10 10 permanent # stubbed free function dummy_f
               # returns [-10;10].
```

are data range specifications for a scenario where:

- x and y are two global variables declared in the package P

- s1 is a variable of record type that has a character type component c

- bar is the name of a stubbed function

- bar1 is a stubbed procedure with outp as out parameter

- proc is a procedure defined with a parameter named i

- dummy_f is a function declared with no parent package

DRS Warning Messages

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

- When a data range constraint is applied:

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

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

  -

    ```
    <DRS_file>, line <line#>: Warning: data range specification with
    incorrect min that is greater than max
    ```

  -

    ```
    <DRS_file>, line <line#>: Warning: data range specification with
    incorrect min or max type. <[Integer|Float|Enum]> value is expected
    ```

  -

4-11

> ```
> <DRS_file>, line <line#>: Warning: data range specification with
> incorrect mode
> ```

- ```
  <DRS_file>, line <line#>: Warning: data range specification with
  incorrect [max|min] value
  ```

- ```
  <DRS_file>, line <line#>: Warning: data range specification
  with [min|min] out of range of ada type
  ```

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

  - ```
    <DRS_file>, line <line#>: Warning: data range specification with
    unsupported object type | DRS cannot be applied to constant variable,
    record discriminant or variant, access type, protected type, task
    entry and local subprogram
    ```

  - ```
    <DRS_file>, line <line#>: Warning: data range specification with
    unsupported variable scope | Variable must be defined within a
    package specification or body
    ```

Performing Efficient Module Testing with DRS

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

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

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

Using the DRS feature, you can define:

- The nominal range for input arguments as preconditions of subprogram calls

- The generic range for input global variables at the start point of each subprogram test

- The generic range for return parameters of stubbed functions, and out or in/out parameters of procedures

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

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

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

Reducing Orange Checks with DRS

When performing robustness (worst case) verification, data inputs are always set to their full range. Therefore, every operation on these inputs, even a

simple "one_input + 10" can produce an overflow, as the range of one_input varies between the minimum value and the maximum value of the type.

If you use DRS to restrict the range of "one-input" to the real functional constraints found in a specification, design document, or models, you can reduce the number of orange checks reported on the variable. For example, if you specify that "one-input" can vary between 0 and 10, Polyspace software recognizes that:

- one_input + 100 never overflows
- the results of this operation are always between 100 and 110

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

The red circle indicates the orange checks that are removed by using DRS.
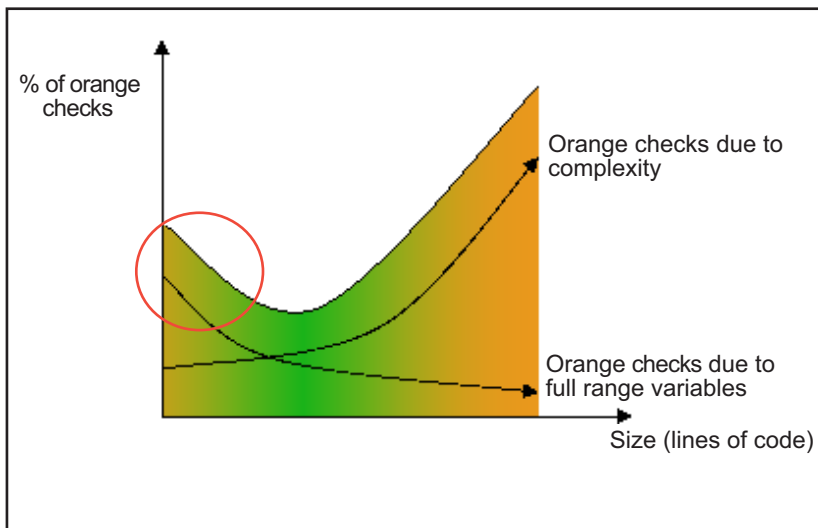
Removing orange checks caused by full-range (worst-case) data can significantly reduce the total number of orange checks, especially in the verification of small files or modules. However, the number of orange checks caused by code complexity is not affected by DRS.

Using Pragma Assert to Set Data Ranges

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

```
pragma assert(<integer expression>);
```

If *<integer expression>* evaluates to zero, then the program is assumed to be terminated, therefore there is a "real" run-time error. This condition is why Polyspace produces checks for the assertions.. The behavior matches the one exhibited during execution, because **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.

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

- The maximum and minimum speed limit (a car never goes faster than 200 miles per hour or slower than 0 miles per hour),

- The maximum duration of software exploitation (five years for a satellite and one hour for its launcher)

Example

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

```
end main;
```

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

Supported Ada Pragmas

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

Pragma	How Polyspace Software Processes Pragma
Import, Import_Function, and Import_Procedure	Stubs function or procedure
Interface and Interface_Name	Stubs function or procedure
Inspection_Point	Provides information about possible values for the variable. May display a range.
Volatile	Variable becomes full-range
Volatile_Components	If you specify Polyspace for Ada95, you get the same results as with the pragma Volatile. However, in this case, the pragma applies to arrays. If you specify Polyspace for Ada83, there is no effect.
Assert	Produces a user assertion check, ASRT. See "User Assertion: ASRT".
Restrictions	Ignored for standard Ada or GNAT compiler restrictions. Other restriction pragmas produce a warning.
Ada_83 and Ada_95	Polyspace option -lang overwrites this pragma (option set by default when you use polyspace-ada or polyspace-ada95).

Pragma	**How Polyspace Software Processes Pragma**

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

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

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

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

How Polyspace Evaluates Function and Procedure Parameters

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

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

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

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

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

```
1     procedure Test2
2     is
3
4      type Rec is
```

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

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

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

5

Preparing Source Code for Verification

Stubbing

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;
  -- 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 run-time error will be ever introduced by automatic stubbing; to minimize preparation time.

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

• External.

• Written in another programming language, for example, C.

• Part of the system libraries.

By default, Polyspace automatically stubs these functions. For information about how Polyspace automatically stubs functions, see "No automatic stubbing".

Preparing Code for Variables

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 for Ada 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 for Ada:

- 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 5-25

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 \
-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 Ada 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 for Ada 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 folder. 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 for Ada does not handle atomicity, so other task synchronization mechanisms (including the use of critical sections) are not recognized by Polyspace Server for Ada.

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
   -- call functions
   X:= 12;
  end; -- end accept
``
	``
	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
	 begin;
	  null;
	 end;
	``

package_first_task	other tasks
``` -- return to main execution end task_1; end first_task; ```	``` end other_tasks; ```

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 for Ada. Nevertheless, Critical sections may be used to model the behavior.

Preparing Multitasking Code

Polyspace Software Assumptions

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

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

- The main procedure must terminate in order for entry-points (or tasks) to start.

- All tasks or entry-points start after the execution of the main has completed. They 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 using the `-main` option is required.

- **The tasks declared in `-entry-points` may not take parameters and may not have return values:** procedure MyTask is end MyTask;

5-15

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

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

Solution 1

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

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

- is more precise,

- but you need to know the exact sequence of events.

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

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

Solution 2

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

- is less precise,

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

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

```
  if (random = 1) than tsk_50ms; end if;
 end_loop
end upper_approx_Ada_function;

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

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

Interruptions and Asynchronous Events/Tasks

Problem
Interrupt service routines appear gray (dead code) in the Results Manager perspective.

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 for Ada.

The execution model is such that the main is executed initially. Only if the main terminates and returns control (i.e. if it is not an infinite loop) will the task entry points be started, 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 for Ada does not distinguish between interrupt service routines and tasks. If you specify an interrupt to be an -entry-point, it will have the same priority level as any other procedures that are also declared as tasks via the -entry-point option. Therefore, as Polyspace Server for Ada 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 that is required to embed the task with body
my_real_package;

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

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

Command Line to Open Polyspace Results Manager Perspective

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

Mailboxes

Problem
My application has several tasks:

- some that post messages in a mailbox;

- others that read these messages asynchronously.

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

Explanation

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

- for send(char *buffer, int length): the content of the buffer will only be written when the procedure is called;

- for receive(char *buffer, int *length): each element of the buffer will contain the full range of values for the corresponding data type.

Solution

You can provide similar mechanisms with different levels of precision.

Mechanism	Description
Let **Polyspace Server for Ada stub automatically**	- Quick and easy to code **Imprecise** because there is no direct connection between a mailbox sender and receiver. It means that even if the sender is only submitting data within a small range, the full data range for the type(s) will be used for the receiver data.
Provide a **real mailbox** mechanism	- Can be very costly (time consuming) to implement - Can introduce errors in the stubs - Is too much effort compared with the solution below

Mechanism	Description
	• 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 for Ada 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 for Ada 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 for Ada considers write/read instructions atomic, so **task 2 can only read 0xFF55**, even if X is not protected (refer to "Shared Variables" on page 5-9).

Critical Sections

In terms of critical sections, Polyspace Server for Ada 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 for Ada verification of run-time errors supposes that there is no conflict when writing the shared variables. Hence even if a shared variable is not protected, the run-time error verification is complete and correct.

More information is available in "Critical Sections" on page 5-9.

Priorities

Priorities are not taken into account by Polyspace. Timing implications of software execution — the primary reason for implementing software task prioritization — are not relevant to the verification performed by Polyspace Server for Ada . In addition, because of priority inversion issues, Polyspace does not make the assumption that priorities can protect shared variables.

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 for Ada 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, regardless of the sequence of instructions — all possible interruptions will be accounted for.

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 that (ignoring priority inversion) never occurs during execution. Polyspace Server for Ada has made an **upper approximation of all scheduling**

and all interleaving. More paths than what is encountered during execution are modelled, but this has no adverse effect on the results obtained.

Highlighting Known Run-Time Errors

Annotating Code to Indicate Known Run-Time Errors

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

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

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

Annotate your code before running a verification:

1 Open your source file using a text editor.

2 Locate the code that produces a run-time error.

3 Insert the required comment.

```
if (Random.random) then
-- polyspace<RTE: NTC: Low: No action planned > A known run-time error

 Square_Root;
end if;
```

See also "Syntax for Run-Time Errors" on page 5-29.

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

4 Save your file.

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

When the verification is complete, open the Results Manager perspective. You see run-time errors in the procedural entities view.

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

Syntax for Run-Time Errors

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

```
-- polyspace<RTE: RunTimeError1[,RunTimeError2[,…]] :
[Classification] :  [Status] >[Additional text]
```

where

- Square brackets *[ ]* indicates optional information.

- *RunTimeError1, RunTimeError2, …* are formal Polyspace checks, for example, COR, OVFL, and NIV. You can also specify ALL, which covers every check.

- *Classification*, for example, High and Low, allows you to categorize the severity of the run-time error with a predefined classification. To see the complete list of predefined classifications, in the PolyspacePreferences dialog box, click the **Review statuses** tab.

- *Status* allows you to categorize the run-time error with a either a predefined status or a status that you define in the Polyspace Preferences dialog box, through the **Review statuses** tab.
- *Additional text* appears in the **Comment** column of the procedural entities view of the Results Manager perspective. Use this text to provide information about the run-time errors.

The following is an example of a comment:

```
-- polyspace<RTE: NTC: Low: No action planned > Known issue
```

The software applies the comments, in a case-insensitive way, to the first non-commented line of Ada code that follows the annotation.

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

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

```
-- polyspace:begin<RTE: RunTimeError1[,RunTimeError2[,…]] :
[Classification] :  [Status] > [Additional text]

... Code section ...

-- polyspace:end<RTE: RunTimeError1[,RunTimeError2[,…]] :
[Classification] :  [Status] > [Additional text]
```

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)
Client	• When the server is busy • Small files **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

Specifying Source Files to Verify

Each Polyspace project can contain multiple modules. Each of these modules can verify a specific set of source files using a specific set of analysis options. Therefore, before you launch a verification, you must specify which files in your project that you want to verify.

To copy a source file to a module:

1 Open the project containing the file you want to verify.

2 In the **Project Browser** Source tree, right-click the source file that you want to verify.

3 From the context menu, select **Copy Source File to > Module_#.**

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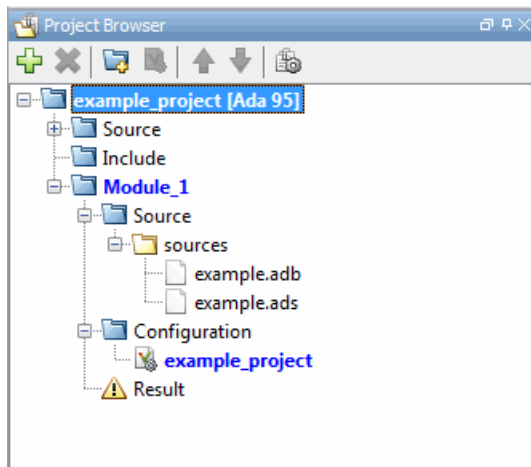
If you have twenty or more modules in your project, when you select **Copy Source File to**, the Select Modules dialog box opens. From the module list, choose the required modules. Then click **Select**.

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

Specifying Results Folder

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

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

folder before launching verification. For example, you may want to reuse a result folder if you stopped a verification before it completed, and are restarting the same verification.

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

To specify the result folder for a verification:

1 In the Project Browser select the module you want to verify.

2 On the Project Manager toolbar, clear the **Create a new result folder** check box.

3 In the **Use result folder** drop-down menu, select the folder you want to use.
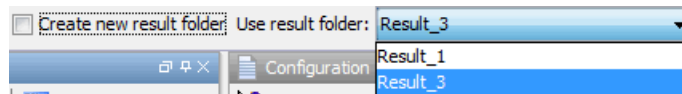
When you run a verification, the software saves verification results in the selected result folder.

Specifying Analysis Options Configuration

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

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

To specify the configuration to use for a verification:

1 In the Project Browser, select the module you want to run.

2 In the Configuration folder of the module, right click the configuration you want to use, and select **Set As Active Configuration**.
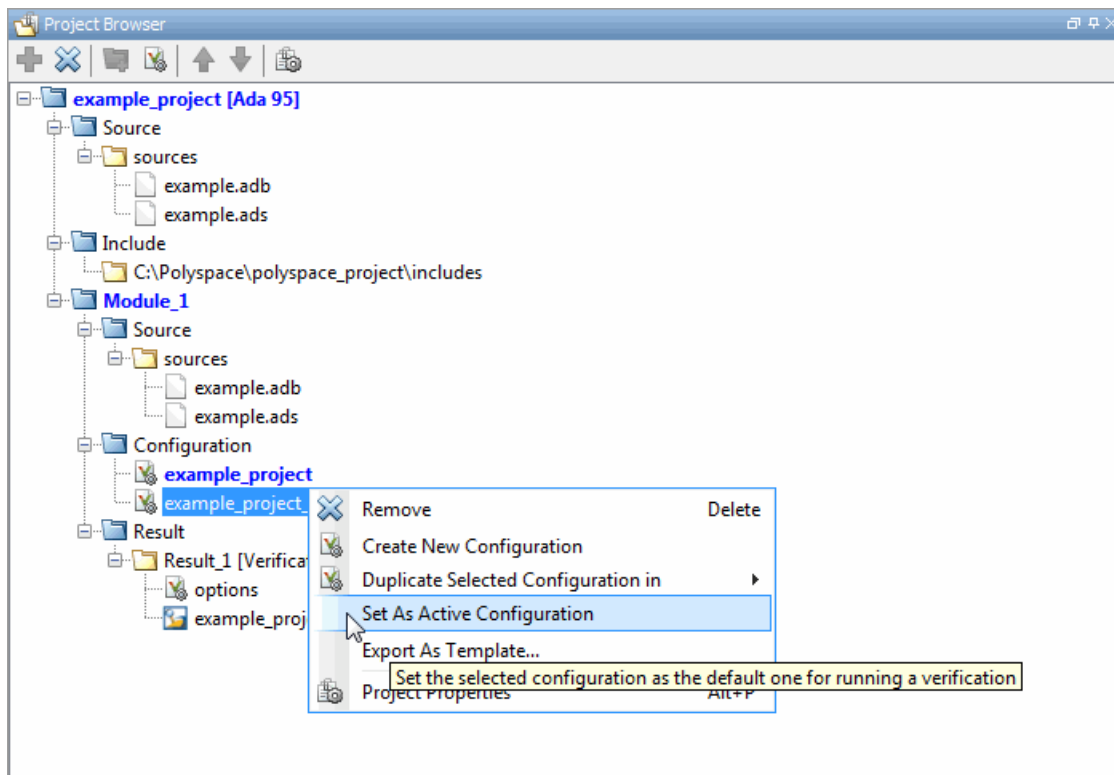
When you run a verification, the software uses the specified analysis options configuration.

Starting Server Verification

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

To start a verification that runs on a server:

1 In the Project Manager perspective, from the **Project Browser** pane, select the module you want to verify.

2 Select the **Configuration > Machine Configuration** pane.

3 Select the **Send to Polyspace ServerGeneral >** check box.

4 On the Project Manager toolbar, click the **Run** button .

The verification starts. For information on the verification process, see "What Happens When You Run Verification" on page 6-7.

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

What Happens When You Run Verification

The verification has three main phases:

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

2 Generating a main if the verification does not find a main and you selected the **Verify module** option. For more information, see "Generate a main".

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

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

- You see the message `queued on server` at the bottom of the Project Manager perspective. This message indicates that the part of the verification that takes place on the client is complete. The rest of the verification runs on the server.

- A message in the **Output Summary** view gives you the identification number (Analysis ID) for the verification.

Running Verification Unit-by-Unit

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

To run a unit-by-unit verification:

1 In the Project Manager perspective, select the **Configuration > Machine Configuration** pane.

2 Select the **Send to Polyspace Server** check box.

3 Select the **Configuration > Verification Mode** pane.

4 Select the **Run unit by unit verification** check box. The **Unit by unit common source** files view is now visible.

5 You can create a list of common files to include with each unit verification:

    a Click ![plus icon] The software creates a new row.

    b In the new row, enter the full path to a common file. For example, C:\Polyspace\polyspace_project\includes\include.h.

    Repeat steps a and b until you have created your list of common files. These files are compiled once, and then linked to each unit before verification.

6 Save your project.

7 On the Project Manager toolbar, click **Run**.

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

Verify All Modules in Project

You can have many modules within a project, each module containing a set of source files and an active configuration.

To verify all modules in a project:

1 In the Project Manager perspective, from the Project Browser, select the project for which you want to run verifications.

2 Select **Run > Run All**.

The software verifies each module as an individual job.

Managing Verification Jobs Using 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 On the Polyspace toolbar, double-click the icon .

The Polyspace Queue Manager opens.

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

Follow Progress...
View Log File...
Download Results...
Download Results in New Polyspace Window...
Download Results and Remove From Queue...

Move Down in Queue

Stop...
Stop and Download Results...
Stop and Remove From Queue...

Remove From Queue...
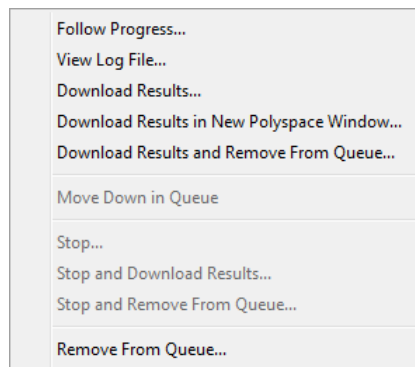
3 Select the required option from the context menu.

Monitoring Progress of Server Verification

Monitoring Progress Using Project Manager

You can monitor the progress of your verification by viewing the progress
monitor and logs at the bottom of the Project Manager perspective.
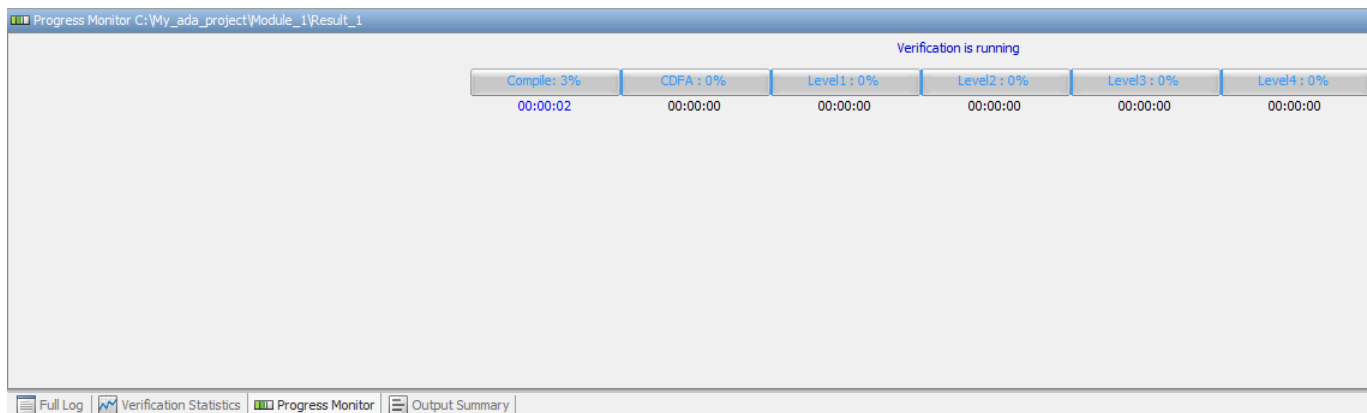
The progress monitor highlights the current phase in blue and displays the
amount of time and completion percentage for that phase.

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

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

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

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

> **Note** You can search the logs. In the **Search** field, enter a search term and click the left arrow to search backward or the right arrow to search forward.

Monitoring Progress Using Queue Manager

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

To monitor a verification:

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

The Polyspace Queue Manager Interface opens.
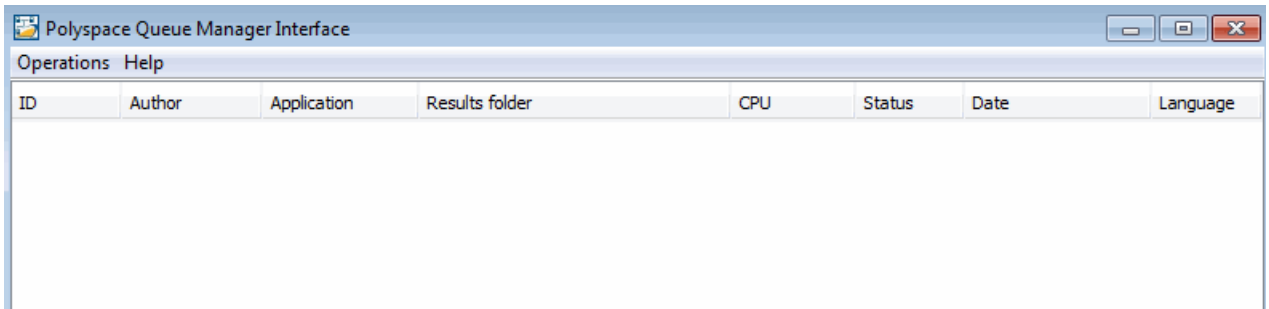
> **Tip** You can also open the Polyspace Queue Manager Interface from the Run-Time Checks perspective toolbar. Click the Polyspace Queue Manager icon .

2 Place the cursor anywhere in the row containing your job.

3 Right-click, and select **Follow Progress** from the context menu.

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

The **Progress Monitor** opens.

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

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

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

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

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

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

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

5 Wait for the verification to finish.

When the verification is complete, 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.

Polyspace Queue Manager Interface							
Operations Help							
ID	Author	Application	Results folder	CPU	Status	Date	Language
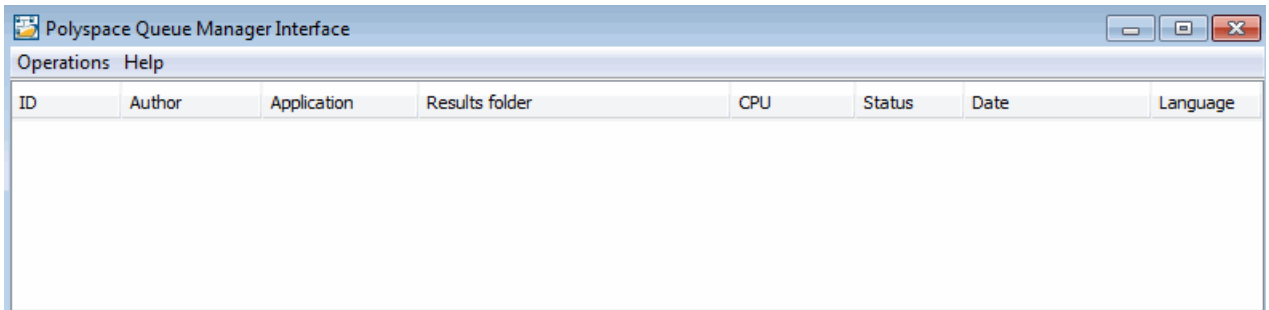
2 Right-click the job you want to monitor, and select **View log file**.

A window opens displaying the last one-hundred lines of the verification.
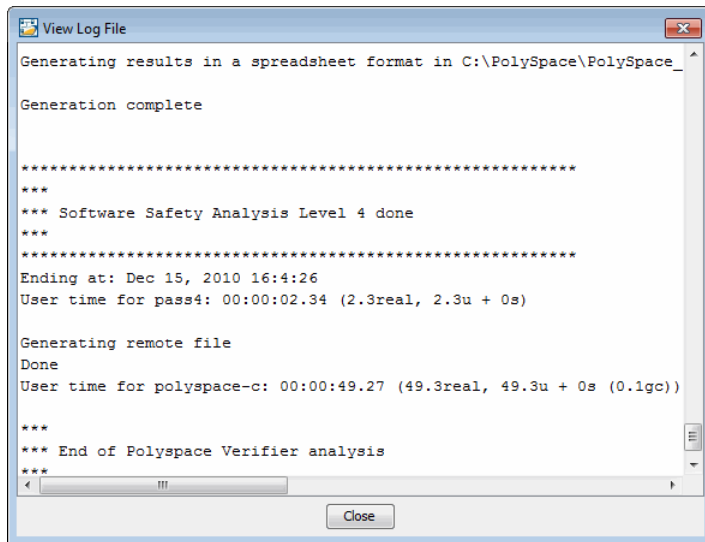
3 Click **Close** to close the window.

Stopping Server Verification Before It Completes

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

To stop a server verification:

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

The Polyspace Queue Manager Interface opens.
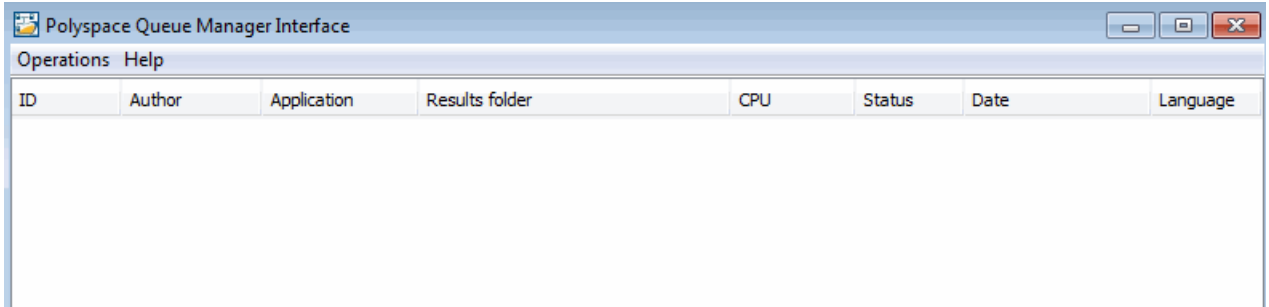
2 Right-click the job you want to monitor, and select one of the following
options:

- **Stop** — Stops a unit-by-unit verification job without removing it. The
  status of the job changes from "running" to "aborted". All jobs in the
  unit-by-unit verification group remain in the queue, and other jobs in
  the group will continue to run.

- **Stop and download results** — Stops the verification job immediately
  and downloads any preliminary results. The status of the verification
  changes from "running" to "aborted". The verification remains in the
  queue.

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

Removing Verification Jobs from Server Queue

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 6-14). 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.

ID	Author	Application	Results folder	CPU	Status	Date	Language

Polyspace Queue Manager Interface

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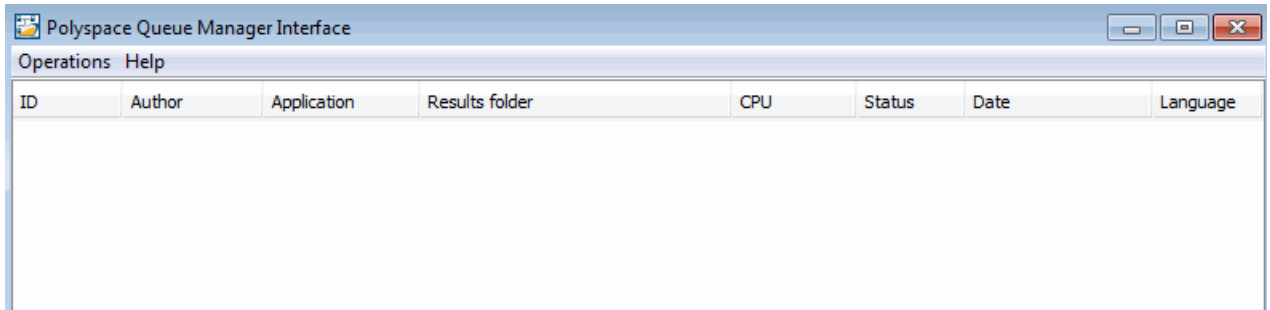
2 Right-click the job you want to remove, and select **Remove from queue**.

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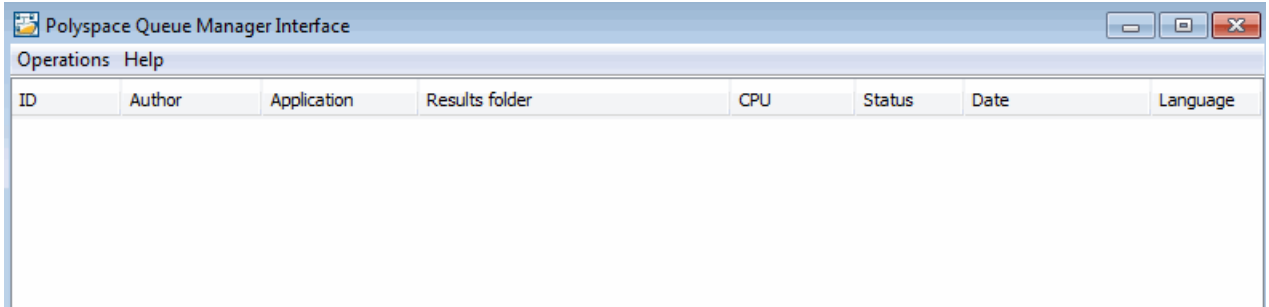
2 Right-click the job you want to remove, and select **Move down in queue**.

The job is moved down in the queue.

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

Purging Server Queue

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

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

To purge the server queue:

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

The Polyspace Queue Manager Interface opens.
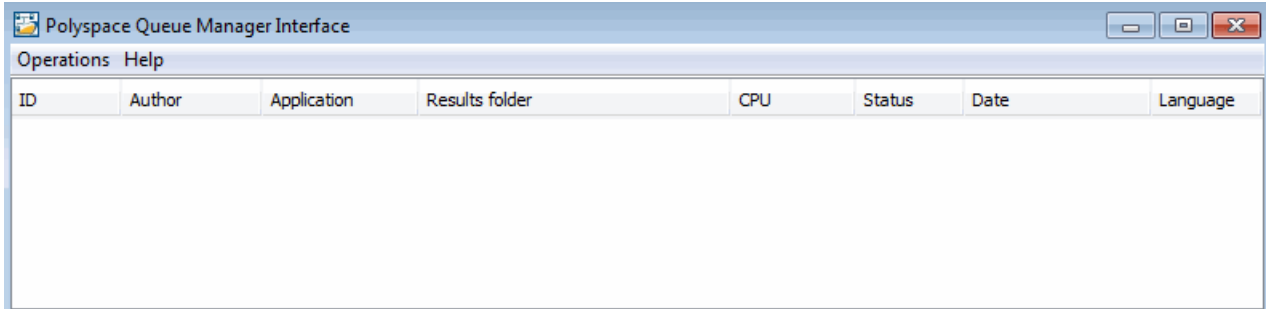
2 Select **Operations > Purge queue**. The Purge queue dialog box opens.
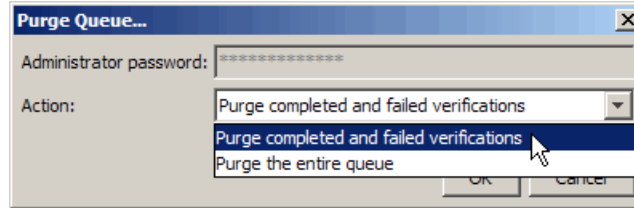
3 Select one of the following options:

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

- **Purge the entire queue** — Removes all jobs from the server queue.

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

4 Enter the Queue Manager **Password**.

5 Click **OK**.

The server queue is purged.

Changing Queue Manager Password

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

Note The default password is `admin`.

To set the Queue Manager password:

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

The Polyspace Queue Manager Interface opens.

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

The Change Administrator Password dialog box opens.

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

The password is changed.

Note Passwords are limited to 8 characters.

Sharing Server Verifications Between Users

Security of Jobs in Server Queue

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

When you manage jobs in the server queue (for example, download, kill, and remove), 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` which is associated to a user account. This file is located in `%APPDATA%\Polyspace`:

- **UNIX®** — `/home/<username>/.Polyspace`

- **Windows** — `C:\Users\<username>\AppData\Roaming\Polyspace`

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

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

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

The fields in the file are tab-separated.

The file cannot contain blank lines.

Example:

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

Sharing Verifications Between Accounts

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

To share a verification with another user:

1 Find the line in your `analysis-keys.txt` file containing the `<ID>` for the job you want to share.

2 Add this line to the `analysis-keys.txt` file of the person who wants to share the file.

The second user can then download or manage the verification.

Magic Key to Share Verifications

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

The format for a magic key is as follows:

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

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

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

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

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

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

To use administrator mode:

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

The Polyspace Queue Manager Interface opens.
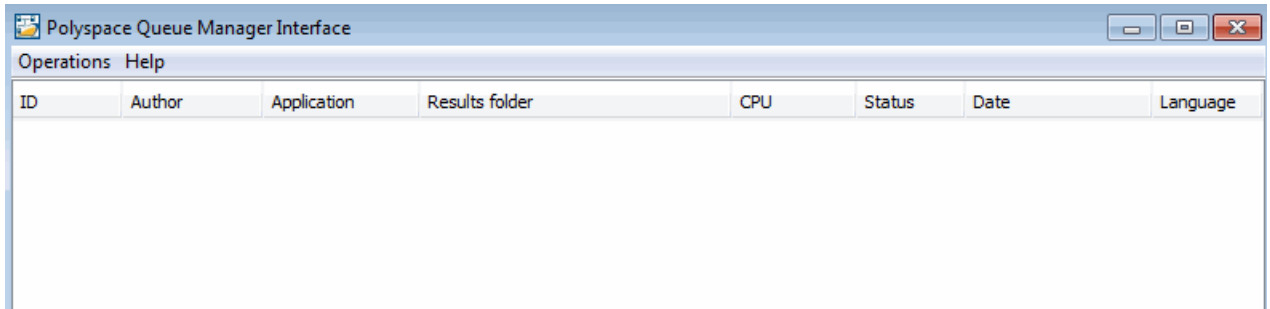
2 Select **Operations > Enter Administrator Mode**.

3 Enter the Queue Manager **Password**.

4 Click **OK**.

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

Running Verifications on Polyspace Client

In this section...
"Specifying Source Files to Verify" on page 6-23
"Starting Verification on Client" on page 6-24
"What Happens When You Run Verification" on page 6-25
"Monitoring the Progress of the Verification" on page 6-25
"Stopping a Client Verification" on page 6-26

Specifying Source Files to Verify

Each Polyspace project can contain multiple modules. Each of these modules can verify a specific set of source files using a specific set of analysis options. Therefore, before you launch a verification, you must specify which files in your project that you want to verify.

To copy a source file to a module:

1 Open the project containing the file you want to verify.

2 In the **Project Browser** Source tree, select the source file that you want to verify.

3 Right-click the file. From the context menu, select **Copy Source File to > Module_#.**

The selected source file appears in the Source tree of the module.
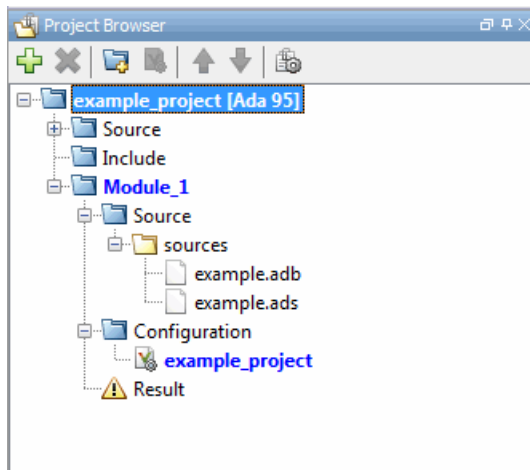
Note You can also drag source files from a project into the Source folder of a module.

Starting Verification on Client

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

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

If you start a verification of Ada code containing more than 2,000 assignments and calls, the verification stops and you see an error message.

To start a verification that runs on a client:

1 In the Project Manager perspective, from the **Project Browser** view, select the module you want to verify.

2 Select the **Configuration > Machine Configuration** pane.

3 Clear the **Send to Polyspace Server** check box.

4 On the Project Manager toolbar, click .

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

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

What Happens When You Run Verification

The verification has three main phases:

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

2 Generating a main if the verification does not find a main and you selected the **Verify module** option. For more information, see "Verify module".

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

Monitoring the Progress of the Verification

You can monitor the progress of the verification by viewing the progress monitor and logs at the bottom of the Project Manager perspective.
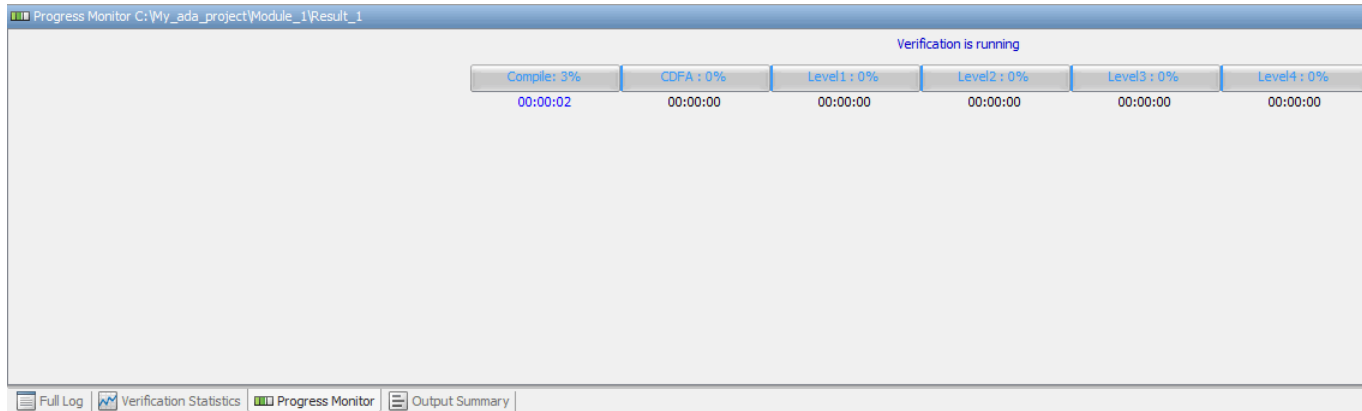
The other tabs report additional information about the progress of the verification:

- Select the **Output Summary** tab to display compilation messages and errors.

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

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

Note You can search the **Output Summary** and **Full Log**. Select the corresponding tab, and in the **Search** field, enter a search term. Click the left arrow to search backward or the right arrow to search forward.

Stopping a Client Verification

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

To stop a verification:

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

A warning dialog box appears.
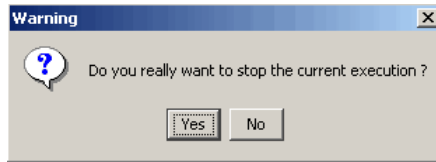
2 Click **Yes**. The verification stops.

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

Running Verifications from Command Line

In this section...
"Starting Batch Verifications" on page 6-28
"Managing Batch Verifications" on page 6-28

Starting Batch Verifications

A set of commands allow you to run batch verifications.

These commands begin with the following prefixes:

- **Server verification** —
  Polyspace_Install/polyspace/bin/polyspace-remote-ada95

- **Client verification** —
  Polyspace_Install/polyspace/bin/polyspace-remote-desktop-ada95

For example, polyspace-remote-desktop-ada95 -server
[<*hostname*>:[<*port*>] | auto] connects the client to the specified server.
This connection allows you to run verifications remotely on the server.

These commands are equivalent to commands with the prefix
PolyspaceInstall/polyspace/bin/polyspace-.

Managing Batch Verifications

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

- psqueue-download <*id*> <*results dir*> — download an identified
  verification into a results folder. When downloading a unit-by-unit
  verification group, all the unit results are downloaded and a summary of
  the download status for each unit is displayed.

  - [-f] force download (without interactivity)
  - -admin -p <*password*> allows administrator to download results.

- ▪ `[-server <name>[:port]]` selects a specific Queue Manager.

- ▪ `[-v|version]` gives release number.

- `psqueue-kill <id>` — kill an identified verification. For unit-by-unit verification groups, you can stop the entire group, or individual jobs within the group. Stopping an individual job does not kill the entire group.

- `psqueue-purge all|ended` — remove all completed verifications from the queue. For unit-by-unit verification jobs, no jobs are removed until the entire group has been verified.

- `psqueue-dump` — gives the list of all verifications in the queue associated with the default Queue Manager. Unit-by-unit verification groups are shown using a tree structure.

- `psqueue-move-down <id>` — move down an identified verification in the Queue. Individual jobs can be moved within a unit-by-unit verification group, but not outside of the group.

- `psqueue-remove <id>` — remove an identified verification in the queue. You cannot remove a single job that is part of a unit-by-unit verification group, you can only remove the entire group.

- `psqueue-get-qm-server` — give the name of the default Queue Manager.

- `psqueue-progress <id>:` give progression of the currently identified and running verification. This command does not apply to unit-by-unit verification groups, only the individual jobs within a group.

  - ▪ `[-open-launcher]` display the log in the Project Manager perspective.

  - ▪ `[-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 "Software Installation".

  - ▪ `[-list-versions]` give the list of available release to upgrade.

  - ▪ `[-install-version <version number> [-install-dir <folder>]] [-silent]` allow to install an upgrade in a given folder and in silent.

Note *Polyspace_Install*/polyspace/bin/psqueue- *<command>* -h
provides information about all available options for each command.

7

Troubleshooting Verification

Verification Process Failed Errors

In this section...
"Verification Failed Messages" on page 7-2
"Hardware Does Not Meet Requirements" on page 7-2
"Location of Included Files Not Specified" on page 7-2
"Polyspace Software Cannot Find the Server" on page 7-3
"Limit on Assignments and Function Calls" on page 7-5

Verification Failed Messages

If you see a message stating that `Verification process failed`, Polyspace software did not perform the verification. The following sections present some possible reasons for a failed verification.

Hardware Does Not Meet Requirements

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

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

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

Location of Included Files Not Specified

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

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

For information on how to specify the location of include files, see "Project Creation".

Polyspace Software Cannot Find the Server

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

```
Error:  Unknown host :
```

To find the server information:

 1 Select **Options > Preferences**.

 2 Select the **Server configuration** tab.

How you handle this error depends on the selected remote configuration option.

Remote Configuration Option	Solution
Automatically detect the remote server	Specify the server by selecting **Use the following server and port** and providing the server name and port.
Use the following server and port	Check the server name and port number.

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

Limit on Assignments and Function Calls

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

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

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

Failure at: Dec 21, 2009 18:21:42
User time for polyspace-desktop-ada95: 1773real, 1097.1u + 101s (6.1gc)
Exiting because of previous error

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

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

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

Compilation Errors

In this section...
"Compilation Error Overview" on page 7-7
"Configuring a Text Editor" on page 7-7
"Examining the Compile Log" on page 7-7
"Common Compile Errors" on page 7-8

Compilation Error Overview

You can use Polyspace software instead of your compiler to make syntactical, semantic, and other static checks. The standard compliance-checking stage takes about the same amount of time as a compiler. If you use Polyspace software early in development, you see objective, automatic, and early control of development work. This control allows you to avoid errors prior to checking code into a configuration management system.

Some compile errors can arise because of implementation-related differences between your environment and Polyspace software.

Configuring a Text Editor

Before you can open source files, you must configure a text editor. For more information, see "Configuring Text Editor" on page 3-21.
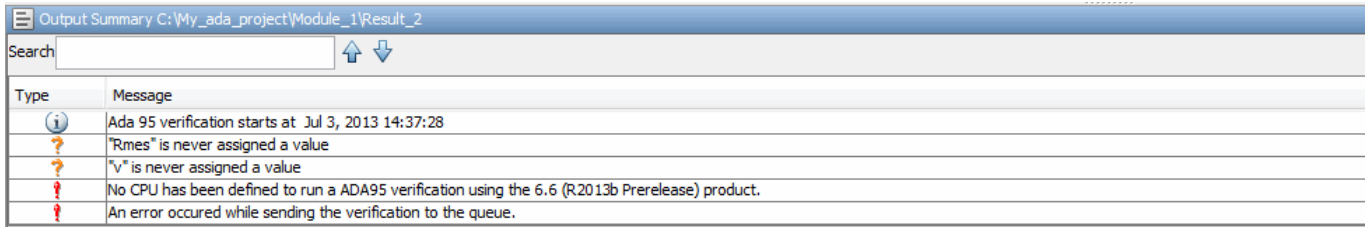
Examining the Compile Log

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

To examine errors in the Compile log:

1 In the log area of the Project Manager perspective, click **Compile**.

A list of compile-phase messages appear in the log part of the window.
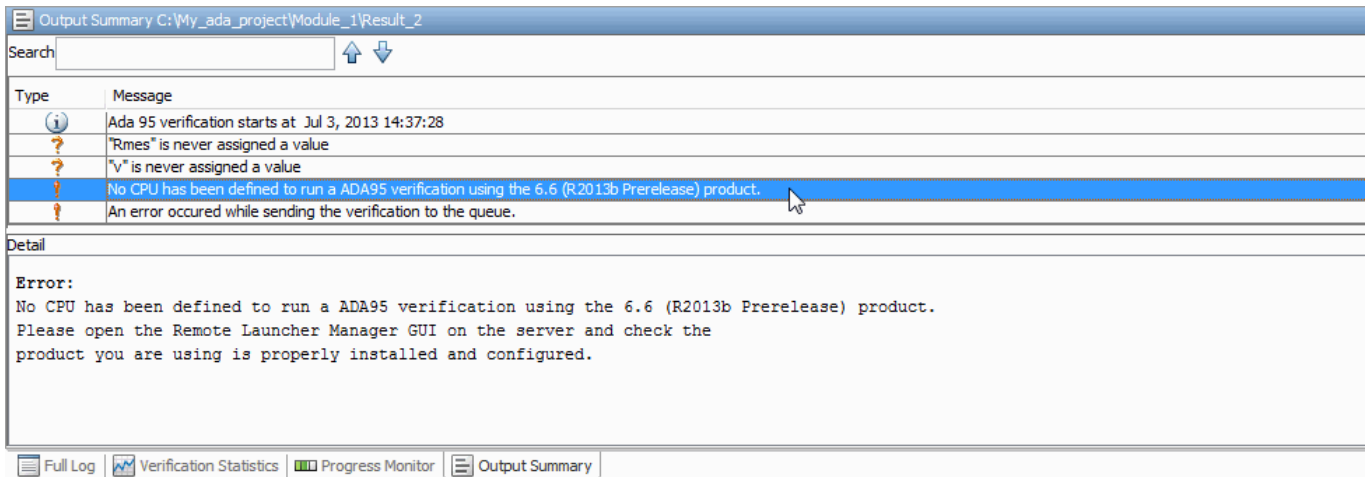
2 Click any of the messages to see message details, as well as the full path of the file containing the error.

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

The file opens in your text editor.

4 Fix the error and run the verification again.

Common Compile Errors

- "Missing specification for unit" on page 7-9
- "Calendar not found" on page 7-10
- "Not a predefined library unit" on page 7-10

Missing specification for unit

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

```
Verifying _pst_main

Verifying my_package

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

Solution/Workaround. Include the specifications of the package body in the list of supplied sources.

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

Calendar not found

Problem. The compiler did not find the package calendar.

Solution/Workaround. In the sources folder, create a file with:

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

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

Not a predefined library unit

Problem. You see the error message:

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

Solution/Workaround. In the sources folder, create a file with the following lines:

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

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

representation clause appears too late

Problem. The compilation phase stops and displays the warning:

```
representation clause appears too late
```

Solution/Workaround. Change:

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

to:

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

Explanation. If you use the `type` keyword between the declaration clause and the representation clause, the Polyspace software displays this warning. The representation clause must apply either all the time or only after the representation clause.

Package system and standard include

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

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

Solution/Workaround. Copy the `system.ads` file from *<product_dir>*\adainclude into your `sources` folder and insert the line:

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

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

Unsigned type

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

Solution/Workaround. Define unsigned types as follows:

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

Function not declared in package

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

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

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

Explanation. Add any missing specifications to the package.

preelaborated unit

Problem. This package has a pragma preelaborate construct.

Solution/Workaround. Comment out the pragma preelaborate construct.

actual must be a definite subtype

Problem. The compile error message is:

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

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

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

The example code is:

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

function UNCHECKED_CONVERSION (S : SOURCE) return TARGET;
```

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

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

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

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

     procedure Main;

end Test;
```

Solution/Workaround. Change the lines:

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

to:

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

to match the Polyspace definitions.

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

'ref attribute

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

Two examples that cause a compile error are:

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

a_var'ref
```

Solution/Workaround. In the preceding examples, use the following code instead:

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

var'address
```

Explanation. This attribute is dependent on the compiler.

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

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

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

Solution/Workaround. Comment out every line that uses the AST_entry or Type_class attribute.

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

Green Hills standard include

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

- The compatibility between the Polyspace and Green Hills include files

- A limitation the Polyspace Verifier encounters when compiling a Green Hills include file

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

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

Package Analysis Limitation

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

Solution/Workaround.

1 Copy package specifications that have unsupported construction from the `includes` folder to the `include-modified` folder.

2 In these files, comment out every unsupported construction.

3 Use the `-ada-include-dir` option to incorporate the modified files in the analysis.

   For example:

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

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

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

Use this workaround for any Ada compiler standard include file.

Ambiguous Bounds in Discrete Range

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

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

Solution/Workaround. Rerun the verification with the following options:

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

Reducing Verification Time

In this section...

Factors Affecting Verification Time

The following factors affect the duration of a verification:

- Size of the code

- Number of global variables

- Nesting depth of the variables (the more nested the variables are, the longer the verification takes)

- Depth of the application call tree

- "Intrinsic complexity" of the code, particularly the arithmetic manipulation

Because many factors impact verification time, there is no precise formula for calculating verification duration. Instead, Polyspace software provides graphical and textual output to indicate how the verification is progressing.

Displaying Verification Status Information

For *client* verifications, monitor the progress of your verification using the **Progress Monitor** and **Verification Statistics** tabs in the Project Manager. For more information, see "Monitoring the Progress of the Verification" on page 6-25.

For *server* verifications, use the Polyspace Queue Manager to follow the
progress of your verification. For more information, see "Monitoring Progress
of Server Verification" on page 6-10.

The progress bar highlights each completed phase and displays the amount
of time for that phase. You can estimate the remaining verification time by
extrapolating from this data, and considering the number of files and passes
remaining.

Ideal Application Size

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

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

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

% of oranges

Oranges due to complexity

Oranges due to missing parts of the software

Size (lines of code)

Best usage: 100 KB - 120 KB lines of code
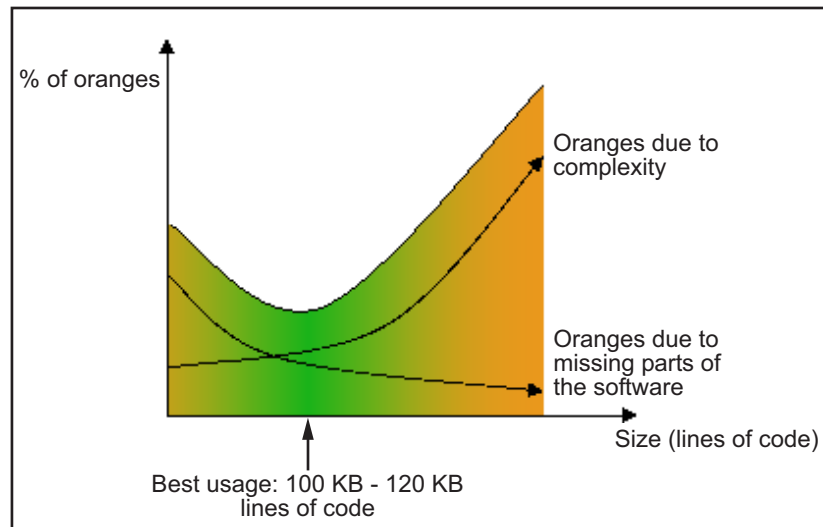
A compromise between selectivity and size

Optimum Size

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

- Makes broader approximations, producing more orange checks.

- Can take much more time to verify the application.

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

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

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

- Keep using Polyspace only upstream in the process.

- Verify subsets of the code.

- Use the `-unit-by-unit` option, as described in "Subdivide According to Files" on page 7-25.

Selecting a Subset of Code

If a project is subdivided into logical sections by considering data flow, the total verification time is shorter than for the project considered in one pass. (See also "Volatile Variables" on page 5-7 and "Automatic Stubbing" on page 5-5.)

In such an application, consider the following:

- Function entry points — Refer to the Polyspace execution model because the function entry points are started concurrently, without any assumption regarding sequence or priority. They represent the beginning of your call tree.

- Data entry points — Examine the lines in the code where data is acquired as "data entry points"

Consider the following examples.

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

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

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

Results

- (–) A slight loss of precision on y. Polyspace considers all possible values for y, including the formatted values present at the first verification.

- (+) A huge investigation of the code is not required to isolate a meaningful subset. Any application can be split logically with this approach.

- (+) No functional modules are lost.

- (+) The results are still valid, 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 (for example, O2) can be maintained.

Examples of Removable Components

- **Error management modules**. Contain a large array of structures that are accessed through an API, but return only a Boolean value. By removing the API code and retaining the prototype, the automatically generated stub is assumed to return a value in the range [-2^31, 2^31-1], which includes 1 and 0. The procedure is considered to return all possible results

- **Buffer management for mailboxes coming from missing code**. Suppose an application reads a huge buffer of 1024 char, and then uses it to populate three small arrays of data, using a very complicated algorithm before passing it to the main module. If the buffer is excluded from the verification and the arrays are initialized with random values instead, the verification of the remaining code is not affected.

Subdivide According to Data Flow

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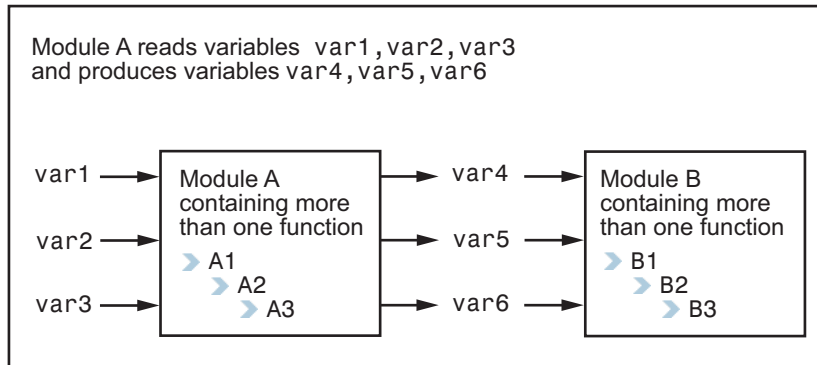
In this application, var1, var2, and var3 can vary between the following ranges.

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

Specification of Module A:

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

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

Ranges	`var4`
	`var5`
	`var6`
		Between -60 and 110
		Between 0 and 12
		Between 0 and 100

Properties	A set of properties between variables	For example: • If `var2` is equal to 0, then `var4` > `var5` > 5. • If `var3` is greater than 4, then `var4` < `var5` < 12

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

• `A` uses `var1`, `var2`, and `var3`, initialized respectively to `[0;10]`, `[1;100]` and `[ 10;10]`.

• `B` uses `var4`, `var5`, and `var6`, initialized respectively to `[-60;110]`, `[0;12]`, and `[ 10;10]`.

Results:

• (–) A slight loss of precision on the `B` module verification, because now all combinations for `var4`, `var5`, and `var6` are:

  ▪ All of the possible combinations.

  ▪ Combinations restricted by the `A` module verification.

• For instance, if the `B` module included the test:

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

  then the dead code on any subsequent `else` clause is undetected.

- (+) An in-depth investigation of the code is not required to isolate a meaningful subset. A logical split is possible for any application, in accordance with the logic of the data.

- (+) The results remain valid, because there is no need to remove a thread that changes shared data.

- (+) The complexity of the code is reduced by a significant factor.

- (+) The maximum precision level can be retained.

Examples of removable components:

- Error management modules. A function has_an_error_already_occurred might return TRUE or FALSE. Such a module may contain a big array of structures that are accessed through an API. The removal of the API code with the retention of the prototype results in the Polyspace verification producing a stub which returns [-2^31, 2^31-1], including 1 and 0 (yes and no). Therefore, the procedure has_an_error_already_occurred returns all possible answers, as the code does at execution time.

- Buffer management for mailboxes coming from missing code. Suppose a large buffer of 1024 char is read, and the data is then collated into three small arrays of data using a complicated algorithm. This data is then given to a main module for processing. For the Polyspace Server verification, the buffer can be removed and the three arrays initialized with random values.

- Display modules.

Subdivide According to Real-Time Characteristics

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

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

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

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

As an example, consider that `T1` can write either 10 or 12 into `x` and that `T2` can both write 15 into `x` and read the value of `x`. There are two ways to achieve a sound standalone verification of `T2`:

- `x` can be declared as volatile to take into account all possible executions. Otherwise, `x` takes only its initial value or `x` remains constant, and `T2` verification is a subset of possible execution paths. You might have precise results, but for only one *scenario* among all possible states for the variable `x`.

- `x` can be initialized to the whole possible range `[10;15]`, and then the `T2` entry point called.

Subdivide According to Files

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

- Use entry points.
- Create a `main` that calls randomly all functions that are not called by any other function within this subset of code.
- Relaunch your verification using the `-unit-by-unit` option. (For more information, see "Run unit by unit verification".)

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

Benefits of Methods

You might want to split the code:

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

The problems that subdivision may create are:

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

When the Application is Incomplete

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

Consider two 32-bit integers a and b, which are initialized with their full range due to missing functions. Here, a*b causes an overflow, because a and b can be equal to 2^{31}. The number of incidences of these "data set issue" orange checks can be reduced by precise stubbing.

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

- Even where precise stubbing is used, verifying a small part of an application might introduce extra orange checks. However, the net effect from reducing the complexity is to reduce the total number of orange checks.

- When using the default stubbing, the increase in the number of orange checks is more pronounced.

Effects of Application Code Size

The Polyspace Server can make approximations when computing the possible values of the variables at any point in the program. Such an approximation always uses a superset of the actual possible values.

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

If the program being verified is large, the Polyspace Server simplifies the internal data representation using a less precise approximation, such as

[-2 ; 2] U {10} U [15 ; 17] U {25} . Here, the same division appears as an orange check.

If the complexity of the internal data becomes even greater later on in the verification, the Polyspace Server might further simplify the VAR range to [-2 ; 25].

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

Note The amount of simplification applied to the data representations also depends on the required precision level (00, 02). The Polyspace Server adjusts the level of simplification, for example:

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

Obtaining Configuration Information

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

Configuration information includes:

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

To obtain configuration information, use the following command:

Polyspace_Install/polyspace/bin/polyspace-ver

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

Storage of Temporary Files

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

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

Disk Defragmentation and Antivirus Software

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

```
Some stats on aliases use:
  Number of alias writes:      22968
  Number of must-alias writes: 3090
  Number of alias reads:       0
  Number of invisibles:        949
Stats about alias writes:
  biggest sets of alias writes: foo1:a (733), foo2:x (728), foo1:b (728)
  procedures that write the biggest sets of aliases: foo1 (2679), foo2 (226
**** C to intermediate language translation - 17 (P_PT) took 44real, 44u +
exception SysErr(OS.SysErr(name="Directory not empty", syserror=notempty))
unhandled exception: SysErr: No such file or directory [noent]


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

On your machine, you must do the following:

• Stop the disk defragmentation tool.

• Deactivate the antivirus software, or configure exception rules for the antivirus software that allow Polyspace to run without failure.

Out-of-Memory Errors During Report Generation

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

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

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

8

Reviewing Verification Results

- "Run Report Generator from Command Line" on page 8-88
- "Automatically Generate Verification Reports" on page 8-90
- "Customizing Verification Reports" on page 8-91
- "Open Verification Report" on page 8-97
- "Using Polyspace Results" on page 8-99

Before You Review Polyspace Results

In this section...
"Overview: Understanding Polyspace Results" on page 8-3
"What Follows Red and Why Green Follows Orange" on page 8-4
"Understanding Polyspace Code Analysis" on page 8-5
"The Code Explanation" on page 8-6

Overview: Understanding Polyspace Results

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

- **Red** – Indicates code that always has an error (errors occur every time the code is executed).

- **Gray** – Indicates unreachable code (dead code).

- **Orange** – Indicates unproven code (code might have a run-time error).

- **Green** – Indicates code proved not to have a run-time error

When reviewing verification results, remember these rules:

- An instruction is verified only if no run-time error is detected in the previous instruction.

- The verification assumes that each run-time error causes a "core dump". The corresponding instruction is considered to have stopped, even if the actual run-time execution of the code might not stop. With orange checks, only the green parts propagate through to subsequent checks.

- Focus on the message produced by the verification, and do not jump to false conclusions. You must understand the color of a check step by step, until you find the root cause of the issue.

- Determine the cause by examining the actual code. Do not focus on what the code is supposed to do.

What Follows Red and Why Green Follows Orange

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

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

When Polyspace divides by X, X is not initialized. Therefore, the software generates a **red** NIV check for the non-initialized variable X. All possible execution paths are stopped because they all produce a run-time error. No checks are generated for the statement X:= X + 1;

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

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

For the propagate procedure:

- X is assigned the value of read_an_input. After this assignment, X = [-2^31, 2^31-1].

- At the first array access, an "out of bounds" error is possible because X can be equal to, for example, -3 as well as 3.

- All conditions leading to a run-time error are truncated. They are no longer considered in the verification. On the following line, the executions for which X = [-2^31, -1] and [100, 2^31-1] are stopped.

- At the next instruction, X = [0, 99].

- At the second array access, the check is green because X = [0, 99].

Summary

Green checks can propagate out of orange checks.

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

See "Using Pragma Assert to Set Data Ranges" on page 4-15.

Understanding Polyspace Code Analysis

Polyspace software numbers checks to correspond to the code execution order.

Consider the instruction x := x + 1; .

The verification first checks for a potential NIV (Non Initialized Variable) for x, and then checks the potential overflow (OVFL). This action mimics the actual execution sequence.

Understanding these sequences can help you to understand the message presented by the verification and the implications for the code.

Consider an orange NIV on x in the test:

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

You might conclude that the verification does not keep track of the value of x. However, consider the context in which the check is made:

```
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 the check in the Results Manager perspective, you see the category of the check. Here, the category is NIV. However, Polyspace might verify subsequent lines of code, and continue the verification as if initialization has taken place.

The 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.

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 Code Explanation

Verification results depend entirely on the code that you are verifying. When interpreting the results, do not consider:r:

- Any physical action from the environment in which the code operates.

- Any configuration that is not part of the verification.

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

```
function Read_An_Input return integer;
procedure Main is
X: Integer;
Y: array (0..99) of Integer;
begin
X := Read_An_input;
Y(X) := 0; -- [array index may be without its bounds] [x is
```

```
initialized]
Y(X-1):= (1 / X) + X ; [array index is within its bounds]
if (X = 0) then
Y(X) := 1; -- this line is unreachable
end if;
end Main;
```

You can see that:

- The line containing the access to the Y array is unreachable.

- Therefore, the test to assess whether x = 0 is always false.

- The initial conclusion is that *the test is always false*. You might conclude that this results from input data that is not equal to 0. However, Read_An_Input can be any value in the full integer range, so this is not the correct explanation.

Instead, consider the execution path leading to the gray code:

- The orange check on the array access (y[x]) truncates any execution path leading to a run-time error, meaning that subsequent lines deal with only x = [0, 99].

- The orange check on the division also truncates all execution paths that lead to a run-time error, so all instances where x = 0 are also stopped. Therefore, for the code execution path after the orange division sign, x = [1; 99].

- x is never equal to 0 at this line. The array access is green (y (x − 1).

Summary

In this example, all the results are located in the same procedure. However, by using the call tree, you can follow the same process even if an orange check results from a procedure at the end of a long call sequence. Follow the "called by" call tree, and concentrate on explaining the issues by reference to the code alone.

Opening Verification Results

Downloading Results from Server to Client

When you run a verification on a Polyspace server, the Polyspace software automatically downloads the results to the client system that launched the verification. In addition, the results are stored on the Polyspace server. You can then download the results from the server to other client systems.

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

To download verification results to your client system:

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

The Queue Manager Interface opens.
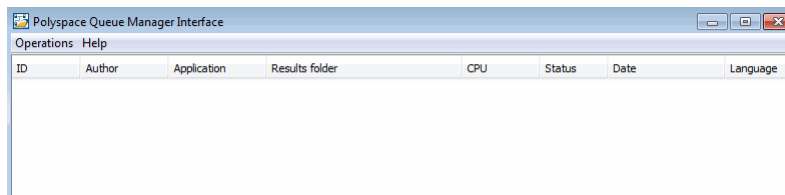
2 Right-click the job that you want to view. From the context menu, select **Download Results** .

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

The Save dialog box opens.

3 Select the folder into which you want to download results.

4 Click **OK** to download the results and close the dialog box.

When the download is complete, the following dialog box opens.
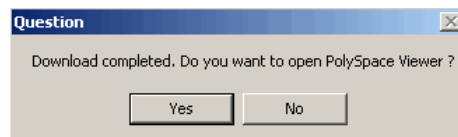
5 Click **Yes** to open the results.

Once you download results, they remain on the client, and you can review them at any time using the Polyspace verification environment.

Downloading Server Results Using the Command Line

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

To download your results, enter the following command:

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

The verification *<id>* is downloaded into the results folder *<results dir>*.

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

Once you download results, they remain on the client, and you can review them at any time using the Polyspace Results Manager perspective.

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

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

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

For more information on managing verification jobs from the command line, see "Managing Batch Verifications" on page 6-28.

Downloading Results from Unit-by-Unit Verifications

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

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

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

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

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

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

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

PolySpace Unit By Unit Results Synthesis

	Green	Orange	Inputs Orange	Dark Orange	Red	Grey	Total	Selectivity	Results	Log file
Source compliance phase results										Open log file
Unit single_file_analysis	97	8	8		2	4	111	93%	Open results	Open log file
Unit main	12	5	3				17	75%	Open results	Open log file
Unit example	99	10	10		5	77	191	95%	Open results	Open log file
Unit tasks2	30	2	2				32	94%	Open results	Open log file
Unit initialisations	52	6				3	61	90%	Open results	Open log file
Unit tasks1	33	5	1				38	87%	Open results	Open log file
	323	36	24	0	7	84	450	92%		

Opening Verification Results from Project Manager Perspective

You can open verification results directly from the Project Browser in the Project Manager perspective. Because each Polyspace project can contain multiple verifications, the Project Browser allows you to quickly identify and open the results that you want to review.

To open verification results from the Project Manager:

1 Open the project containing the results that you want to review.

2 In the Project Browser tree, navigate to the results that you want to review.
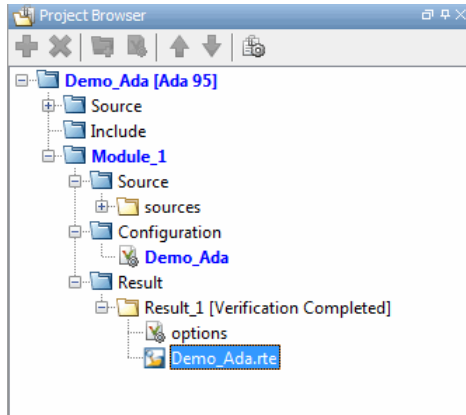
3 Double-click the results file.
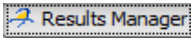
The results open in the Results Manager perspective.

Opening Verification Results from Results Manager Perspective

Use the Results Manager perspective to review verification results. If you know the location of the results file that you want to review, you can open the file directly from the Results Manager perspective.

Note You can also browse and open results from the Project Browser in the Project Manager perspective.

To open verification results from the Results Manager perspective:

1 On the Polyspace verification environment toolbar, select the **Results Manager** button ⬛ Results Manager .

2 Select **File > Open Result**

The Open results dialog box opens.

3 Select the results file that you want to view.

4 Click **Open**.

The results open in the Results Manager perspective.

Search Results in Results Manager

This example shows how to search for all occurrences of a variable or function name in the Results Manager perspective. Search for the variable or function name in the following situations:

- A read/write operation on a variable causes a check. However, the check might be related to an instruction prior to this read/write operation.

  Selecting a check in the **Results Summary** pane displays the read/write operation only. On the **Source** pane, you can look in the source code for prior instructions containing the variable name. Instead, searching for the occurrences is an easier way to find and quickly navigate to them.

  For instance, consider the check, `Out of bounds array index`. Though an access operation on the array causes the check, it is useful to quickly navigate to the array declaration.

- A function call causes a check. However, the check might be related to an instruction in the function definition. Therefore, it is useful to quickly navigate to the function definition.

Search Variable Name

1 Enter the variable name in the Search box in the Results Manager perspective toolbar. Alternatively, on the **Source** pane, right-click the variable name and select **Search "*variable_name*" In All Source Files**.
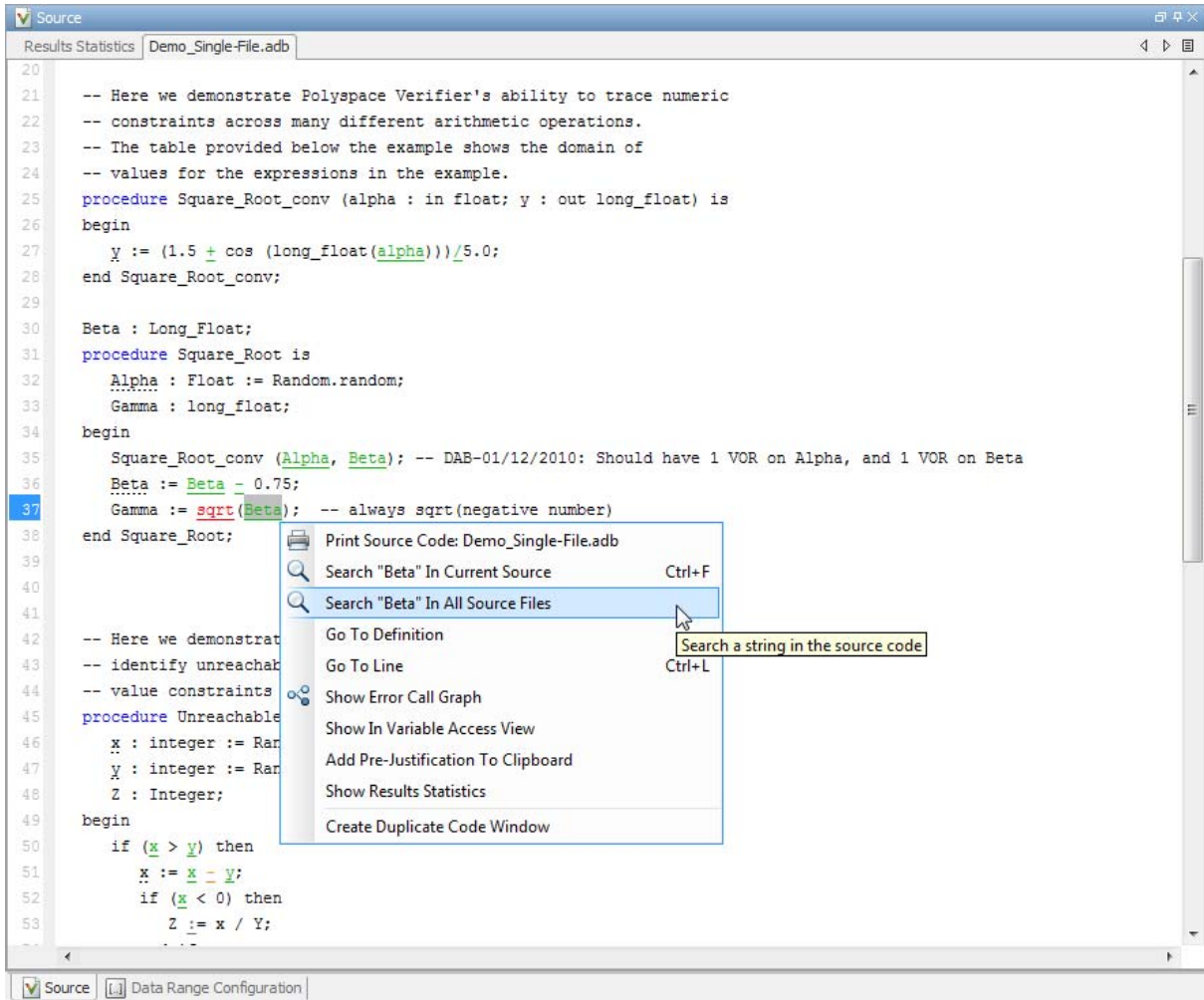
The **Search** tab displays all occurrences of the variable name under four categories.

2 To see all occurrences of the variable name in the source code, expand the node **Source Code View**.

This node lists each occurrence of the variable name along with the file name and the line number. Use the file name and line number to identify all occurrences of the variable name before a check occurs.
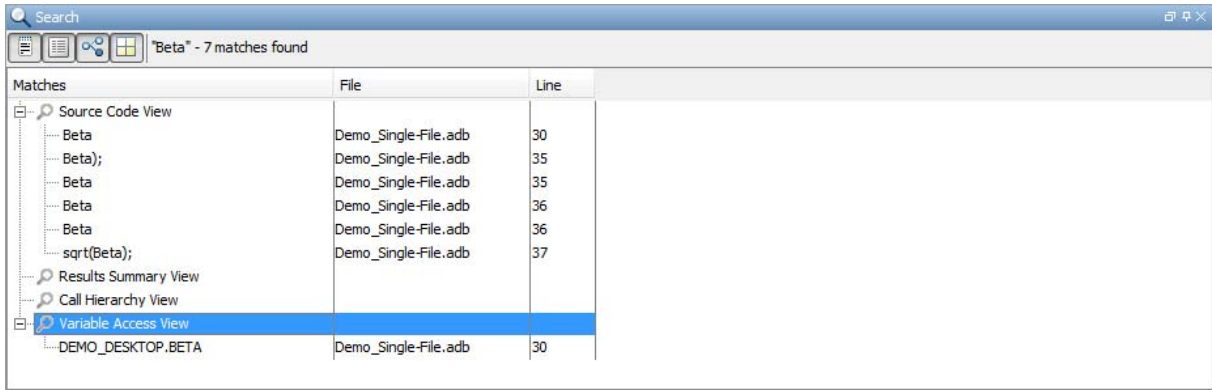
3 To navigate to a particular occurrence of the variable name in the source code, use the up and down arrow keys.

The **Source** pane displays the corresponding line of code.

4 If the variable is a global variable, to navigate to its declaration quickly, expand the node **Variable Access View**. Select the only search result under this node.

Search Procedure Name

1 Enter the procedure name in the Search box in the Results Manager perspective toolbar. Alternatively, on the **Source** pane, right-click the procedure name and select **Search "*procedure_name*" In All Source Files**.

The **Search** tab displays all occurrences of the procedure name under four categories.

2 To see all occurrences of the procedure name in the source code, expand the node **Source Code View**.

This node lists each occurrence of the procedure name along with the file name and the line number.

3 To navigate to a particular occurrence of the procedure name in the source code, use the up and down arrow keys.

The **Source** pane displays the corresponding line of code.

4 To navigate to the procedure definition quickly:

a On the **Results Summary** pane, select `Checks by File/Function` from the drop-down list.

The **Results Summary** pane displays all file names in alphabetical order. Under each file name, the pane displays the procedure names in alphabetical order.

b Select the name of the procedure.

The **Source** pane displays the procedure definition.

Set Character Encoding Preferences

If the source files that you want to verify are created on an operating system that uses different character encoding than your current system (for example, when viewing files containing Japanese characters), you receive an error message when you view the source file or run certain macros.

The **Character encoding** option allows you to view source files created on an operating system that uses different character encoding than your current system.

To set the character encoding for a source file:

1 Select **Options > Preferences**.

2 In the **Polyspace Preferences** dialog box, select the **Character encoding** tab.

Polyspace Preferences

Server Configuration		Project and Results Folder		Editors
Tools Menu	Review Configuration	Review Statuses	Miscellaneous	Character Encoding

Specifies the character encoding used by the operating system on which the source file was created.
This allows you to view source files created on an operating system that uses different character encoding than the current system.

Select the character encoding that you want to use.

Latin/Western European (ISO) (ISO-8859-1)

16-bits UCS Transformation Format, byte order identified by an optional byte-order mark	(UTF-16)
16-bits Unicode (or UCS) Transformation Format, little-endian byte order with byte-order mark	(x-UTF-16LE-BOM)
16-bits Unicode Transformation Format, big-endian byte order	(UTF-16BE)
16-bits Unicode Transformation Format, little-endian byte order	(UTF-16LE)
32-bits UCS Transformation Format, byte order identified by an optional byte-order mark	(UTF-32)
32-bits Unicode (or UCS) Transformation Format, big-endian byte order with byte-order mark	(X-UTF-32BE-BOM)
32-bits Unicode (or UCS) Transformation Format, little-endian byte order with byte-order mark	(X-UTF-32LE-BOM)
32-bits Unicode Transformation Format, big-endian byte order	(UTF-32BE)
32-bits Unicode Transformation Format, little-endian byte order	(UTF-32LE)
8-bits UCS Transformation Format	(UTF-8)
American Standard Code for Information Interchange	(US-ASCII)
Arabic	(IBM420)
Arabic	(IBM864)
Arabic (Macintosh)	(x-MacArabic)
Arabic (Windows)	(windows-1256)
Arabic - Windows	(x-IBM1046)
Austria, Germany	(IBM273)
Baltic	(IBM775)
Baltic (Windows)	(windows-1257)
Canadian French (MS-DOS)	(IBM863)
Catalan/Spain, Spanish Latin America	(IBM284)
Chinese (AIX)	(x-IBM1383)
Chinese (AIX)	(x-IBM834)
Chinese (AIX)	(x-IBM964)
Chinese (Hong Kong, Taiwan)	(x-IBM950)
Chinese (OS/2)	(x-IBM1381)
Chinese (Simplified)	(GBK)
Chinese (Simplified) Host mixed with 1880 UDC, superset of 5031	(x-IBM935)
Chinese (Simplified) PRC standard	(GB18030)
Chinese (Simplified), GB2312 in ISO 2022 CN form	(x-ISO-2022-CN-GB)
Chinese (Traditional)	(Big5)

Select current operating system character encoding: Western European (Windows) (windows-1252)

Note: You must restart Polyspace to use the new character encoding settings.

OK   Apply   Cancel

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

4 Click **OK**.

5 Close and restart the Polyspace verification environment to use the new character encoding settings.

Review Checks Progressively

This example shows how to review checks progressively using the Results Manager perspective.

1 Select the **Results Summary** view.
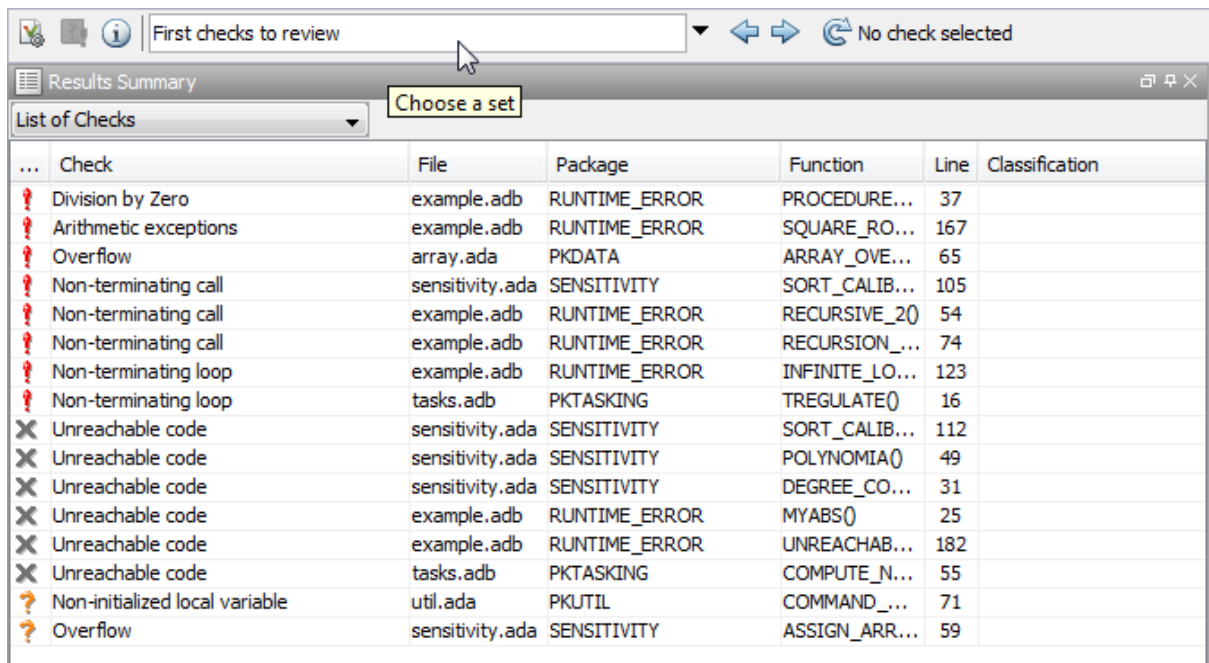
2 Click the forward arrow to go to the first check in the set:

- The **Source** pane displays the source code for this check.

- The **Check Details** pane displays information about this check.

3 Review the current check.

After you review a check, you can classify the check and enter comments to describe the results of your review. You can also mark the check as Justified to help track your review progress.

4 Continue to click the forward arrow until you have gone through all of the checks.

After the last check, a dialog box opens asking if you want to start again from the first check.
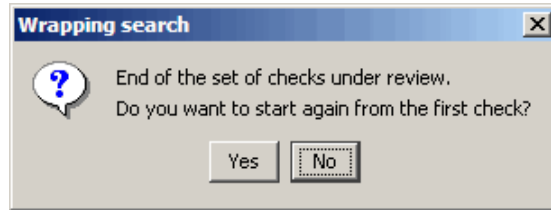
5 Click **No**.

Note If you want to navigate through justified checks, use the justified check forward arrow and back arrow .

Review and Comment Checks

This example shows how to review and comment checks using the Results Manager perspective. When reviewing checks, you can assign a status to checks, and enter comments to describe the results of your review. These actions help you to track the progress of your review and avoid reviewing the same check twice.
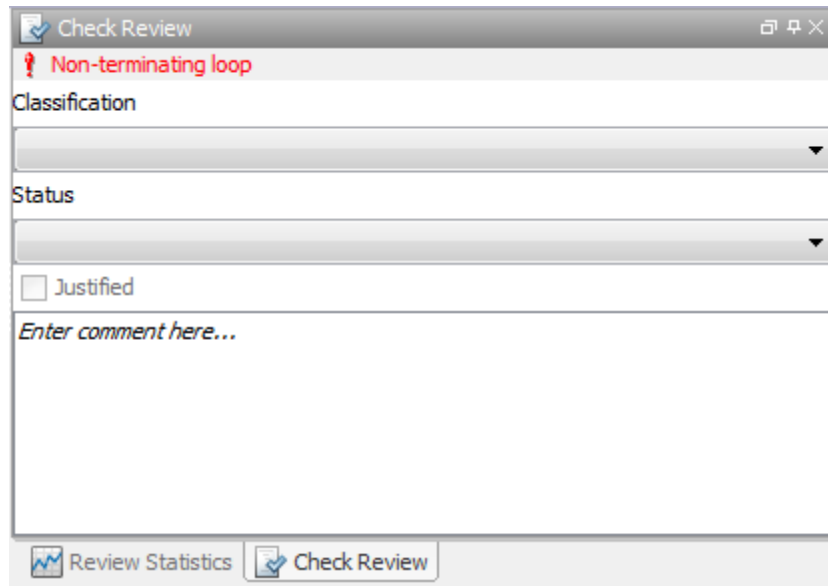
Review and Comment Individual Check

1 On the **Results Summary** pane, select the check that you want to review.

The **Check Details** pane displays information about the current check.

The **Check Review** tab displays fields where you can enter review information.

2 Select a **Classification** to describe the severity of the issue:

- Unset
- High
- Medium
- Low
- Not a defect
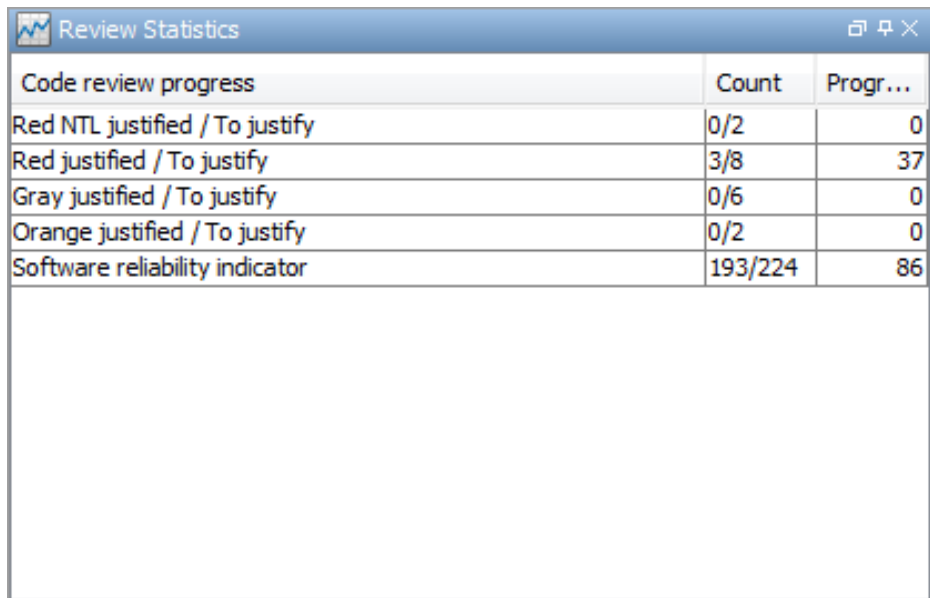
3 Select a **Status** to describe how you intend to address the issue:

- Fix
- Improve
- Investigate
- Justify with annotations
- No action planned
- Other
- Restart with different options

- Undecided

4 To justify the check, select one of the **Status** options, `Justify with annotations` or `No action planned`.

On the **Review Statistics** pane, the software updates the ratios of errors justified to total errors.

Code review progress	Count	Progr...
Red NTL justified / To justify	0/2	0
Red justified / To justify	3/8	37
Gray justified / To justify	0/6	0
Orange justified / To justify	0/2	0
Software reliability indicator	193/224	86

5 In the **Comment** field, enter remarks, for example, defect or justification information.

Note You can also enter the review information through the **Classification**, **Status**, and **Comment** fields on the **Results Summary** pane.

Review and Comment Group of Checks

1 On the **Results Summary** pane, select a group of checks using one of the following methods:

- For contiguous checks, left-click the first check. Then **Shift**-left click the last check.

...	Check	File	Package	Function
!	Division by Zero	example.adb	RUNTIME_ERROR	PROCEDURE_ZDV()
!	Arithmetic exceptions	example.adb	RUNTIME_ERROR	SQUARE_ROOT()
!	Overflow	array.ada	PKDATA	ARRAY_OVERFLOW_INIT()
!	Non-terminating call	sensitivity.ada	SENSITIVITY	SORT_CALIBRATION()
!	Non-terminating call	example.adb	RUNTIME_ERROR	RECURSIVE_2()
!	Non-terminating call	example.adb	RUNTIME_ERROR	RECURSION_CALLER()
!	Non-terminating loop	example.adb	RUNTIME_ERROR	INFINITE_LOOP()
!	Non-terminating loop	tasks.adb	PKTASKING	TREGULATE()
✘	Unreachable code	sensitivity.ada	SENSITIVITY	SORT_CALIBRATION()

To group together all checks belonging to a certain category, click the **Check** column header on the **Results Summary** pane.

- For non-contiguous checks, **Ctrl**-left click each check.

...	Check	File	Package ▲ 1	Function
!	Division by Zero	example.adb	RUNTIME_ERROR	PROCEDURE_ZDV()
!	Arithmetic exceptions	example.adb	RUNTIME_ERROR	SQUARE_ROOT()
!	Non-terminating call	example.adb	RUNTIME_ERROR	RECURSIVE_2()
!	Non-terminating call	example.adb	RUNTIME_ERROR	RECURSION_CALLER()
!	Non-terminating loop	example.adb	RUNTIME_ERROR	INFINITE_LOOP()
✘	Unreachable code	example.adb	RUNTIME_ERROR	MYABS()
✘	Unreachable code	example.adb	RUNTIME_ERROR	UNREACHABLE_CODE()
?	Division by Zero	example.adb	RUNTIME_ERROR	RECURSION()
?	Overflow	example.adb	RUNTIME_ERROR	UNREACHABLE_CODE()

- For all checks of a similar color and category, right-click one check. From the context menu, select **Select All *Color Type*Checks**, for instance, **Select All Red "Non-terminating call" Checks**.

...	Check	File	Package	Function
🥣	Division by Zero	example.adb	RUNTIME_ERROR	PROCEDURE_ZDV()
🥣	Arithmetic exceptions	example.adb	RUNTIME_ERROR	SQUARE_ROOT()
🥣	Overflow	array.ada	PKDATA	ARRAY_OVERFLOW_INIT()
🥣	Non-terminating call	sensitivity.ada	SENSITIVITY	SORT_CALIBRATION()
🥣	Non-terminating call			RECURSIVE_2()
🥣	Non-terminating call			RECURSION_CALLER()
🥣	Non-terminating loop			INFINITE_LOOP()
🥣	Non-terminating loop			TREGULATE()
✖	Unreachable code			SORT_CALIBRATION()
✖	Unreachable code			POLYNOMIA()
✖	Unreachable code	sensitivity.ada	SENSITIVITY	DEGREE_COMPUTATION()
✖	Unreachable code	example.adb	RUNTIME_ERROR	MYABS()

Context menu overlay:
- Show Error Call Graph
- Add Pre-Justification To Clipboard
- Show Results Statistics
- Select All Red "Non-terminating call" Checks

2 On the **Check Review** tab, enter the required information. The software applies this information to the selected checks.

Save Review Comments

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

To save your review comments, select **File > Save**. Your comments are saved with the verification results.

Organize Check Review Using Filters and Groups

This example shows how to filter and group checks on the **Results Summary** pane. To organize your review of checks, use filters and groups when you want to:

- Review certain categories of checks in preference to others. For instance, you first want to address all checks resulting from `Non-terminating loop`.

- Not address the full set of coding rule violations detected by the coding rules checker.

- Not review checks you have already justified.

  Typically, in your second or later rounds of review, you would have some checks already justified.

- Review only those checks that you have already assigned a certain status. For instance, you want to review only those checks to which you have assigned the status, `Investigate`.

- Review all checks in the body of a particular file or function. Because of continuity of code, reviewing all these checks together can help you organize your review process.

  You can also review all checks in a file if you have written the code for that file only and not the entire set of source files used for verification.

- Not review the checks in automatically generated functions.

Review Checks in a Given Category

To review all checks resulting from `Non-terminating loop`:

1 Open the results file, with extension, `.rte`.

2 On the **Results Summary** pane, from the drop-down list, select `Checks by Family`.

  The checks are grouped by type of check.

Family	File	Package	Function	Line	%	Classification	Status
⊟ 1 Red Check					100		
⊞ Control flow		6			100		
⊞ Numerical		2			100		
⊟ 2 Gray Check		6			100		
⊞ Data flow		6			100		
⊟ 3 Orange Check		17			0		
⊞ Data flow		3			0		
⊞ Numerical		13			0		
⊞ Other		1			0		
⊟ 4 Green Check		193			100		
⊞ Data flow		135			100		
⊞ Numerical		58			100		

3 Under the category **1 Red Check**, expand the subcategory **Control flow**.

You see the subcategory **Non-terminating loop**.

Family	File	Package	Function	Line	%	Classification	Status
⊟ 1 Red Check		8			100		
⊟ Control flow		6			100		
⊞ Arithmetic exceptions		1			100		
⊞ Non-terminating call		3			100		
⊟ Non-terminating loop		2			100		
❗ example.adb		RUNTIME_ERR...	INFINITE_LO...	123			
❗ tasks.adb		PKTASKING	TREGULATE()	16			
⊞ Numerical		2			100		

Expand **Non-terminating loop** to view all red checks resulting from this error.

To see further information about a check, select it. The information appears on the **Check Details** pane.

4 To view all orange checks resulting from this error, repeat step 3 for the subcategory **Control flow** under the category **3 Orange Check**.

5 To view only the checks resulting from the error, `Non-terminating loop`, on the **Results Summary** pane, from the drop-down list, select `List of Checks`.

6 Place your cursor on the **Check** column head.
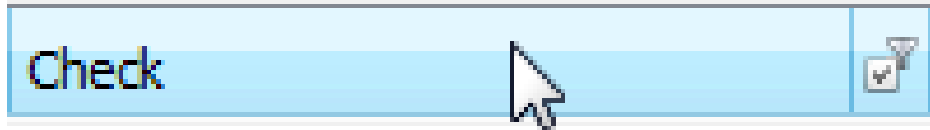
7 Click the filter icon.

A context menu lists all the filter options available.
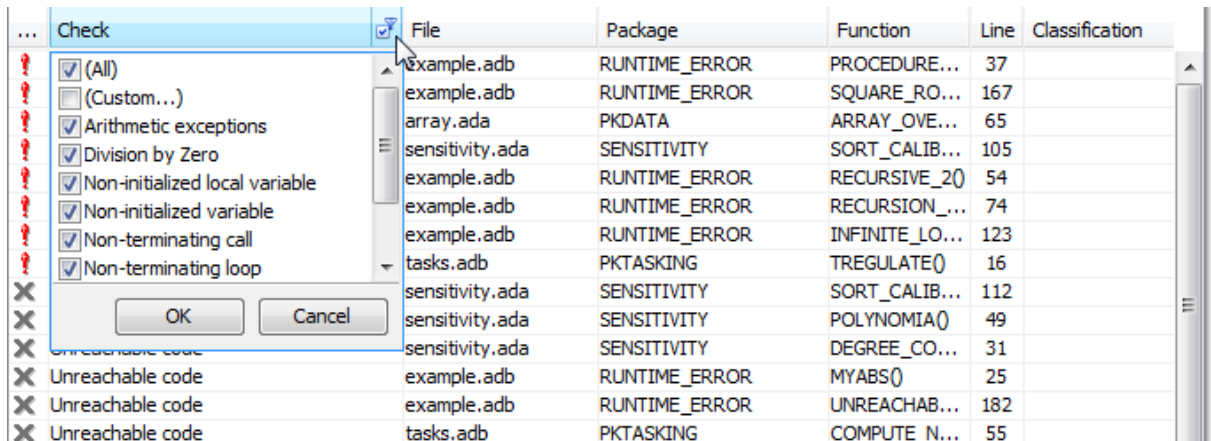
8 Clear the **All** check box.

9 Scroll down to the **Non-terminating loop** check box and select it. Click **OK**.

The **Results Summary** pane displays only the checks resulting from the `Non-terminating loop` error.

Review Checks Not Justified

To review only the checks that you have not justified:

1 Open the results file, with extension, `.rte`.

2 On the **Results Summary** pane, place your cursor on the **Justified** column head.

3 Click the filter icon.

A context menu lists all the filter options available.

4 Clear the **True** check box. Click **OK**.

The **Results Summary** pane displays only the checks that you have not justified.

Review Checks with Given Status

To review only the checks with `Investigate` status:

1 Open the results file, with extension, `.rte`.

2 On the **Results Summary** pane, place your cursor on the **Status** column head.

3 Click the filter icon.

A context menu lists all the filter options available.
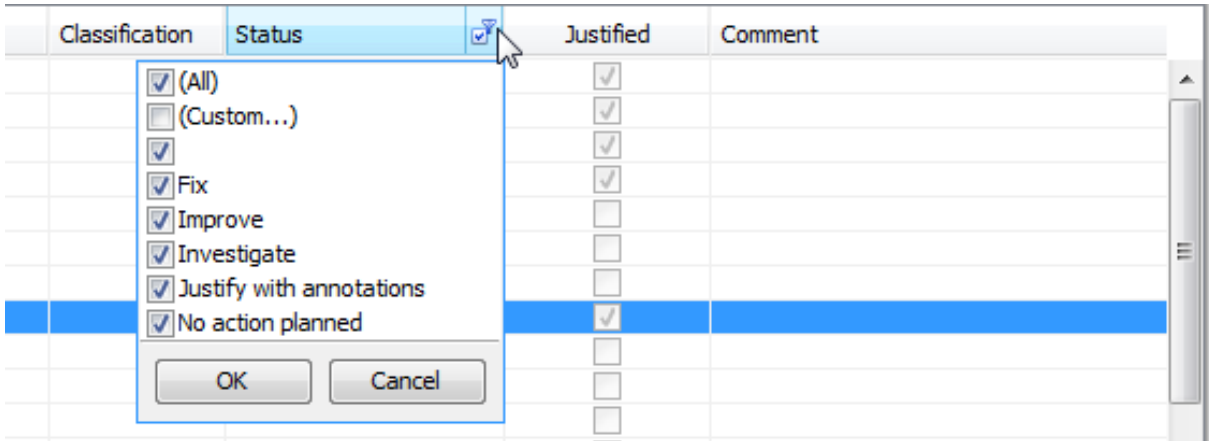
4 Clear the **All** check box.

5 Select the **Investigate** check box. Click **OK**.

The **Results Summary** pane displays only the checks with the
`Investigate` status.

Review All Checks in a File

To review the checks in the file, `tasks.adb`:

1 On the **Results Summary** pane, from the drop-down list, select `Checks`
`by File/Function`.

The checks displayed are grouped by files. The file names are sorted
alphabetically. Within each file name, the checks are grouped by functions,
sorted alphabetically. Each file or function is colored by the most severe
check that occurs. The severity decreases in this order:

- Red

- Gray

- Orange

- Green

Results Summary

Checks by File/Function ▼

Family	Check	Package	Line	%	Classification	Status
⊟ array.ada		1 49		100		
⊞ ARRAY_OVERFLOW_INIT()		1 23		100		
⊞ NON_INTRUSIVE_INFORMATIONS()		25		100		
⊞ SEND_MESSAGE()		1		100		
⊟ example.adb		5 2 6 65		92		
⊞ CLOSE_TO_ZERO()		4 7		64		
⊞ FIBONACCI()		10		100		
⊞ INFINITE_LOOP()		1 7		100		
⊞ MAINRTE()		1		100		
⊞ MYABS()		1 2		100		
⊞ NON_INFINITE_LOOP()		10		100		
⊞ PROCEDURE_ZDV()		1 5		100		
⊞ RECURSION()		1 6		86		
⊞ RECURSION_CALLER()		1 2		100		
⊞ RECURSIVE_2()		1 1		100		
⊞ SQUARE_ROOT()		1 5		100		
⊞ SQUARE_ROOT_CONV()		4		100		
⊞ UNREACHABLE_CODE()		1 1 5		86		
⊟ random.adb		3		100		
⊞ RANDOM$2()		1		100		
⊞ RANDOM$3()		1		100		
⊞ RANDOM()		1		100		
⊟ sensitivity.ada		1 3 3 40		94		
⊞ ASSIGN_ARRAY()		1 4		80		
⊞ DEGREE_COMPUTATION()		1 16		100		
⊞ DIVIDE()		1 2		67		
⊞ POLYNOMIA()		1 5		100		
⊞ SORT_ALGORITHM()		1 11		92		
⊞ SORT_CALIBRATION()		1 1 2		100		
⊟ tasks.adb		1 1 3 25		90		
⊞ COMPUTE_NEW_COORDONATES()		1 3		100		
⊞ ONE_INTERRUPT1()		2		100		
⊞ ONE_INTERRUPT2()		6		100		
⊞ TREGULATE()		1 1 6		88		
⊞ TREGULATE.ORDER()		2 2		50		
⊞ TSERVER()		6		100		
⊟ util.ada		5 11		69		
⊞ COMMAND_ORDERING()		1		0		
⊞ COMPUTE_INJECTION()		2 4		67		
⊞ EXEC_ONE_CYCLE()		2		100		
⊞ GET_POWERLEVEL()		1		100		
⊞ INCREASE_POWERLEVEL()		1 1		50		
⊞ PARTIAL_INIT()		1 1		50		
⊞ PILOT_BALANCE()		1		100		
⊞ SEQUENCER()		1		100		

2 To view the checks in `tasks.adb`, expand any procedure name under the category, **tasks.adb**.

⊟ tasks.adb	1 1 3 25		90		
⊞ COMPUTE_NEW_COORDONATES()	1 3		100		
⊞ ONE_INTERRUPT1()	2		100		
⊞ ONE_INTERRUPT2()	6		100		
⊟ TREGULATE()	1 1 6		88		
⦗!⦘ Non-terminating loop	PKTASKING	16			
⦗?⦘ User assertion	PKTASKING	28			
✔ Non-initialized variable	PKTASKING	24			
✔ Overflow	PKTASKING	24			
✔ Non-initialized variable	PKTASKING	24			
✔ Overflow	PKTASKING	24			
✔ Non-initialized variable	PKTASKING	24			
✔ Non-initialized variable	PKTASKING	28			
⊞ TREGULATE.ORDER()	2 2		50		
⊞ TSERVER()	6		100		

To view further information on a check, select the check. The information on the check appears on the **Check Details** pane.

3 To view only the checks in `tasks.adb`, on the Results Summary pane, from the drop-down list, select `List of Checks`.

The **Results Summary** pane displays all checks without any grouping.

4 Place your cursor on the **File** column head.

5 Click the filter icon.

A context menu lists all the filter options available.
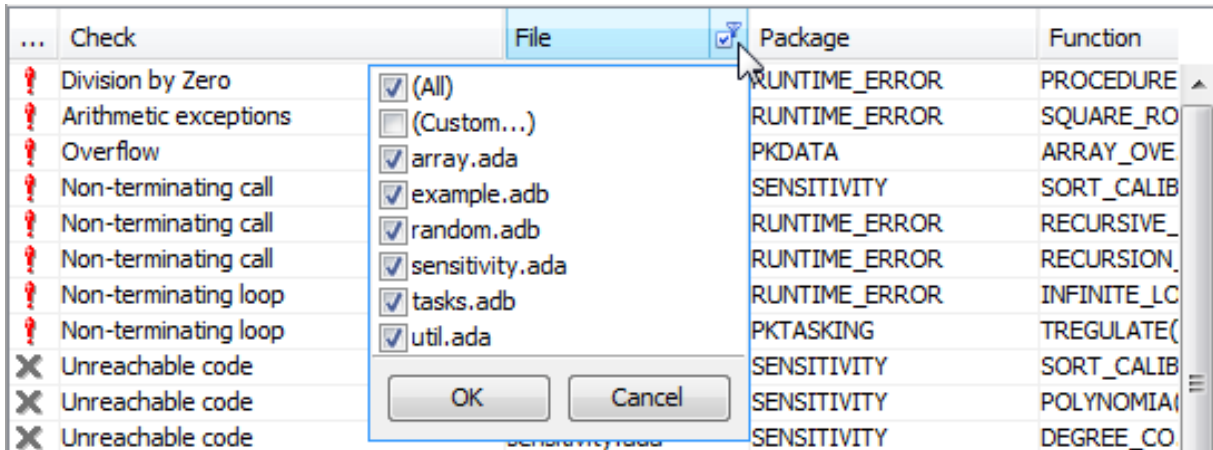
6 Clear the **All** check box.

7 Select the **tasks.adb** check box. Click **OK**.

The **Results Summary** pane displays only the checks in `tasks.adb`.

Tip If you apply a filter on a column on the **Results Summary** pane, the column header displays the number of check boxes selected in the filter menu. Use this information to keep track of any filters that you have applied.

Display Call Sequence for a Check

This example shows how to display the call sequence that leads to the code line associated with a check.

1 On the **Results Summary** pane, select the check that you want to review.

   On the **Check Details** pane, the **Check Details** tab displays further information about the check.

   The **Source** pane displays the code line associated with the check.

2 On the **Check Details** tab toolbar, click the Show error call graph button,

   On the **Check Details** pane, the **Graph** tab displays the call graph.
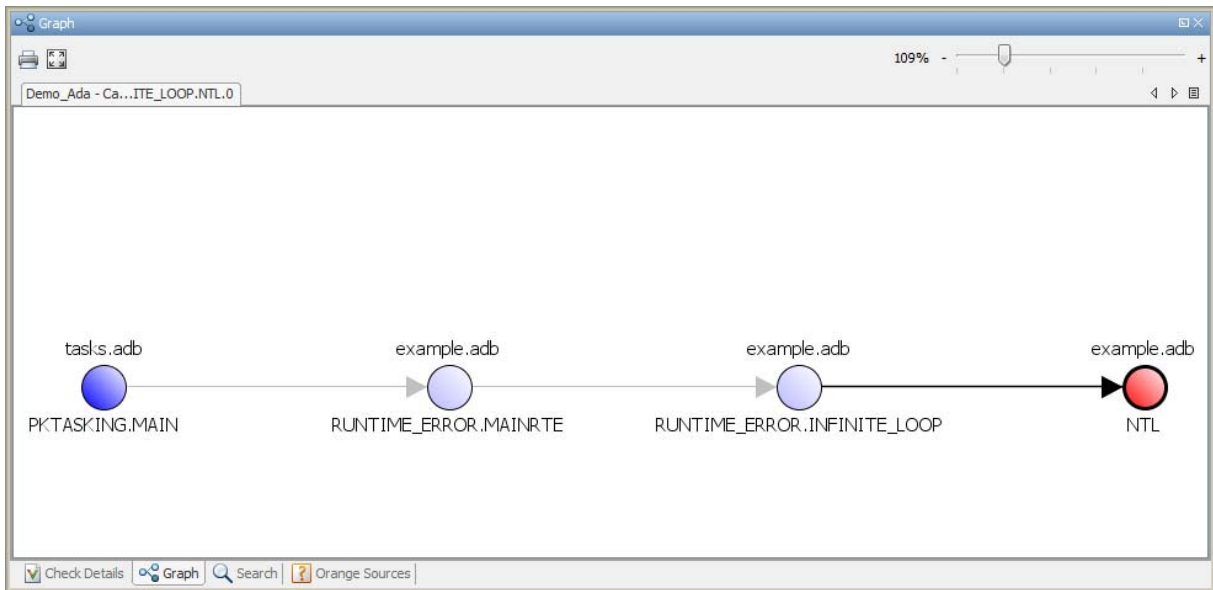
The call graph displays the call sequence leading to the code associated
with the check. Each node on the graph, except for the terminal node,
represents a procedure. The procedure name is below the node. The name
of the file containing the procedure is above the node.

3 Select a node to navigate to the procedure definition in the source code.

The **Source** pane displays the procedure definition.

4 Select the terminal node to navigate back to the code line associated with
the check.

View Call Tree

In this section...
"View Callers and Callees of a Procedure" on page 8-38
"Navigate Call Tree" on page 8-41

The call tree (or call graph) shows the calling relationship between procedures (and tasks) in a program. From the call tree, for each procedure or task, `foo`, you can see its:

- Callers: Procedures and tasks calling `foo`.
- Callees: Procedures and tasks called by `foo`.

Sometimes, an error in a procedure might be related to an instruction in its callers or callees. Therefore, to review errors quickly, it is useful to:

- View all callers and callees of a procedure without navigating in the source code.
- Navigate quickly between a procedure, and its callers and callees.
- Verify dataflow for certification purposes.

You can perform all these tasks from the **Call Hierarchy Pane** in the Results Manager perspective.

For a complete description of the **Call Hierarchy** pane, see "Call Hierarchy" on page 8-78.

Note If you do not see the **Call Hierarchy** pane in the Results Manager perspective, select **Window > Show/Hide View > Call Hierarchy**.

View Callers and Callees of a Procedure

You can view all callers and callees of a procedure from the **Call Hierarchy** pane.

1 View the procedure in the **Call Hierarchy** pane.

Begin from procedure name.

To view a procedure starting from its name, group the checks in the **Results Summary** pane by function names first. Select **Checks by File/Function** from the dropdown menu at the top of this pane.

Then, list the checks under a procedure by double-clicking the procedure name. Select any check. The procedure appears on the **Call Hierarchy** pane. This pane also lists the names of callers and callees.

Begin from check in procedure.

To view a procedure starting from a check inside the procedure, select the check on the **Results Summary** pane (in the Results Manager perspective).

The procedure containing the check appears on the **Call Hierarchy** pane. This pane also lists the names of callers and callees.

2 Select the procedure name.

In the **Source** pane, the current line shows the beginning of the procedure definition.

3 Select a callee name. These are listed below the procedure name and marked by ▶ (procedures) or ‖▶ (tasks).

On the **Source** pane, the current line shows where the callee is called.
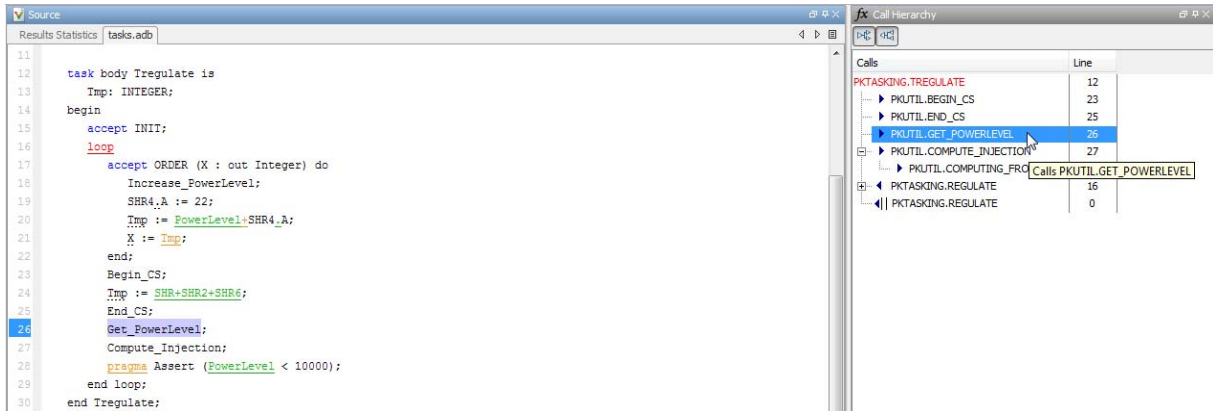
4 Select a caller name. These are listed below the function name and marked by ◀ (functions) or ◀|| (tasks).

In the **Source** pane, the current line shows where the caller calls the function.
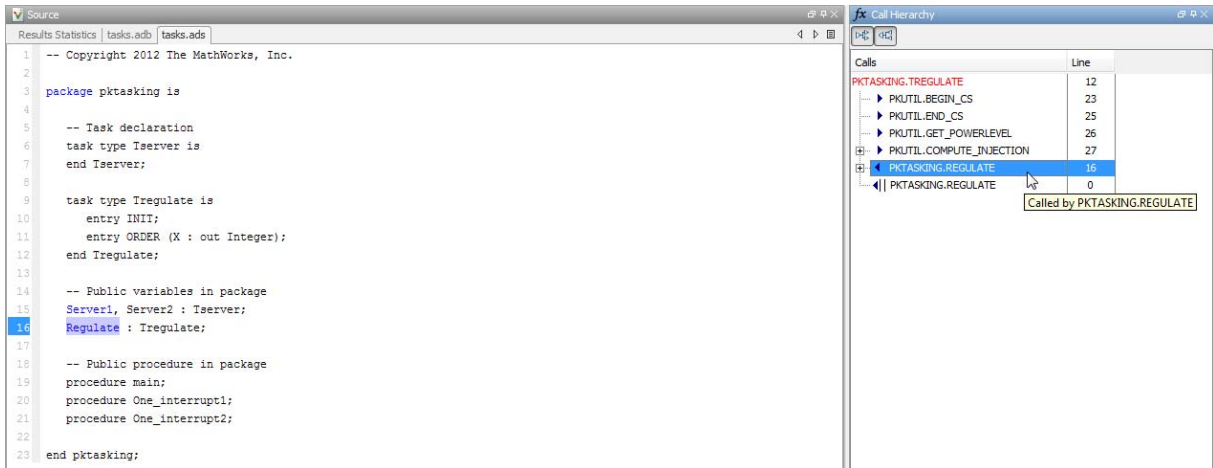
5 View all branches of a callee by progressively clicking ⊞ next to the callee name.

6 View all branches ending with the caller by progressively clicking ⊞ next to the caller name.

Tip Instead of progressively viewing the branches by clicking ⊞ , you can expand all caller/callee names at once. Right-click anywhere in the **Call Hierarchy** pane. From the context menu, select **Expand All Nodes**. Likewise, you can collapse all caller/callee names by right-clicking anywhere in the **Call Hierarchy** pane and selecting **Collapse All Nodes**.

Navigate Call Tree

To navigate between a procedure and its callers and callees in the source code:

1 Select a check contained in the procedure from the **Results Summary** pane. The **Call Hierarchy** pane shows the procedure.

2 To navigate to a callee in the source code, double-click the callee name. These names are listed below the procedure name and marked by ▶ (procedures) or ‖▶ (tasks). Alternatively, right-click the callee name and from the context menu, select **Go To Definition**.

The **Call Hierarchy** pane now shows the callee. In the **Source** pane, the current line shows the beginning of the callee procedure definition.

3 To navigate to a caller, double-click the caller name. These names are listed below the procedure name and marked by ◀ (procedures) or ◀‖ (tasks). Alternatively, right-click the caller name and from the context menu, select **Go To Definition**.

The **Call Hierarchy** pane now shows the caller. In the **Source** pane, the current line shows the beginning of the caller procedure definition.

Display Access Graph for Variables

This example shows how to display the access sequence for any global variable that is read or written in the code.

1 On the **Variable Access** pane, select the variable that you want to view.

2 On the **Variable Access** pane toolbar, click the Show Access Graph button

.

A window displays the access graph.
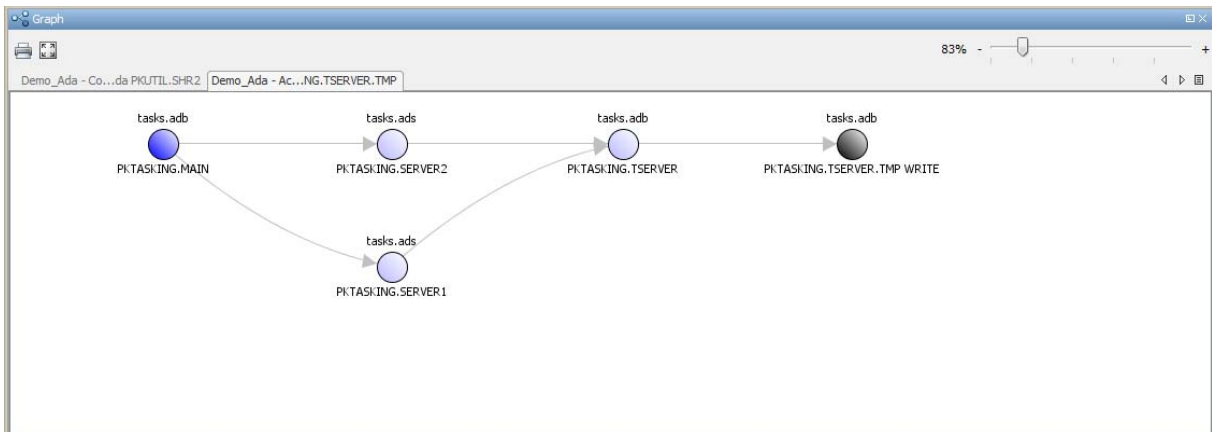
The access graph displays the read and write accesses for the variable. Each node represents a function.

3 On the graph, click any node to navigate to the corresponding function on the **Source** pane. The **Call Hierarchy** pane displays the call tree of the function.

Customize Review Status

This example shows how to customize the statuses you assign on the **Check Review** pane.

Define Custom Status

1 Select **Options > Preferences**.

2 Select the **Review Statuses** tab.

3 Enter your new status at the bottom of the dialog box, then click **Add**.

Polyspace Preferences				✕

Server Configuration		Project and Results Folder		Editors
Tools Menu	Review Configuration	**Review Statuses**	Miscellaneous	Character Encoding

Statuses	Justify
Fix	☐
Improve	☑
Investigate	☐
Justify with annotations	☑
No action planned	☑
Other	☐
Restart with different options	☐
Undecided	☐

User Statuses	Justify

Remove

Add a new status: | Not an issue | ☑ | **Add**

Add a new status:

OK   Apply   Cancel

The new status appears in the **Status** list.

4 Click **OK** to save your changes and close the dialog box.

When reviewing checks, you can select the new status from the **Check Review > Status** drop-down list.

Add Justification to Existing Status

By default, a check is automatically justified if you assign the status, `Justify with annotations` or `No action planned`. However, you can change this default setting so that a check is justified when you assign one of the other existing statuses.

To add justification to existing status `Improve`:

1 Select **Options > Preferences**.

2 Select the **Review Statuses** tab. For the `Improve` status, select the check box in the **Justify** column. Click **OK**.

Statuses	Justify
Fix	☐
Improve	☑
Investigate	☐
Justify with annotations	☑
No action planned	☑
Other	☐
Restart with different options	☐
Undecided	☐

User Statuses	Justify

Remove

Add a new status: ☑ Add

OK Apply Cancel
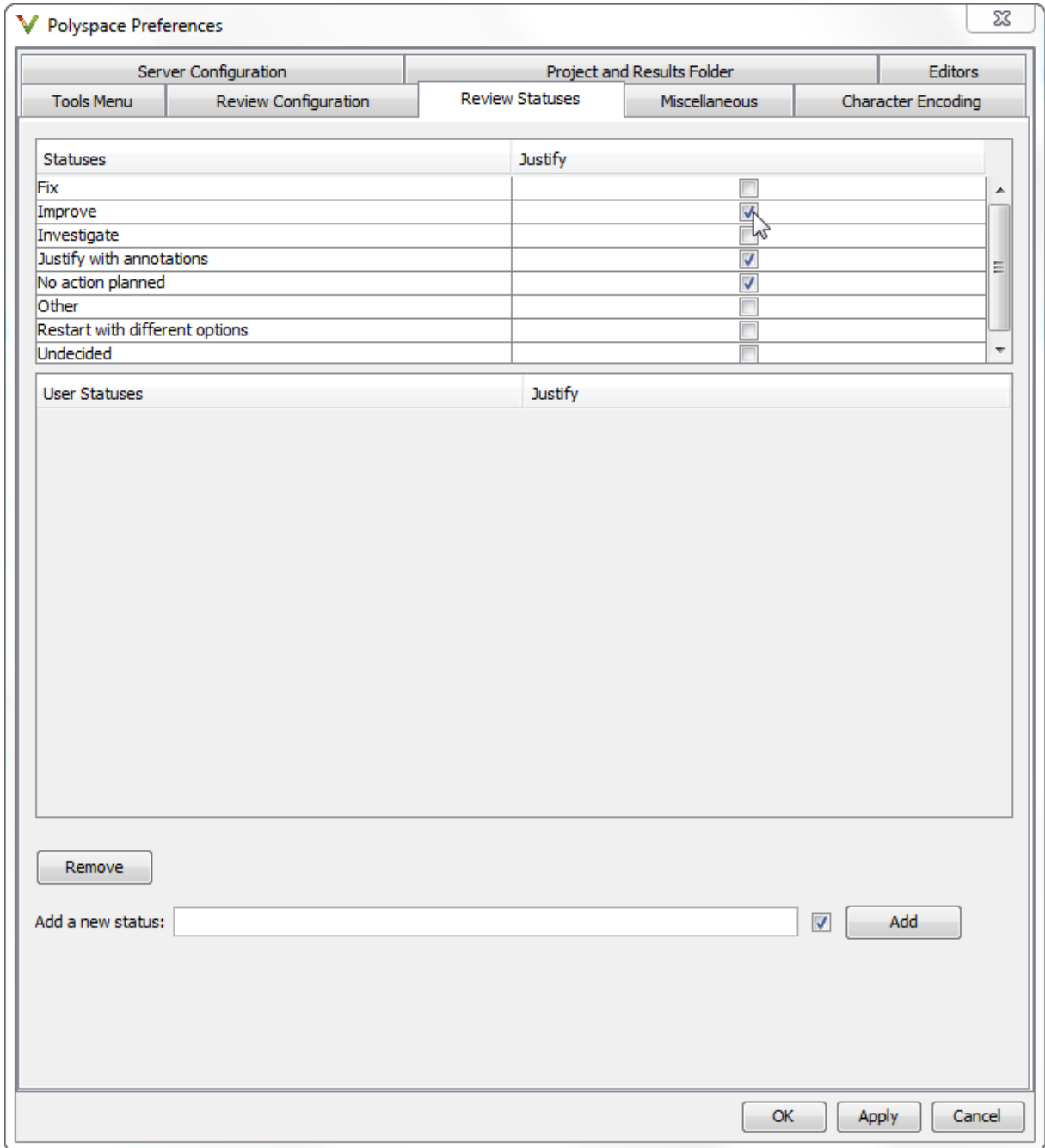
If you assign the `Improve` status to a check on the **Check Review** pane, the check gets automatically justified.

Track Review Progress

This example shows how to track review progress using the **Review Statistics** pane.

1 To see how many checks of a certain type you have reviewed, select a check of that type on the **Results Summary** pane.

The **Count** column on the **Review Statistics** pane displays the number of unreviewed checks as a ratio of total number of checks. The first row on the **Review Statistics** pane displays the ratio for checks of the same color and type as the selected check.

2 To see how many checks of a certain color you have reviewed, select any check on the **Results Summary** pane.

The **Count** column on the **Review Statistics** pane displays the ratio of unreviewed checks to total number of checks for all check colors.

The **Progress (%)** column displays the same ratio as a percentage.

What Are Review Methodologies?

To facilitate your review of verification results, you can specify the number and type of checks displayed on the **Results Summary** pane of the Results Manager perspective. These specifications are known as review methodologies.

By choosing a methodology, you can review a subset of checks. The code verification software provides four predefined review methodologies.

Select the predefined review methodologies in the following order to incrementally view the checks displayed. For more information, see "Review Checks Using Predefined Methodologies" on page 8-51.

1 **First checks to review** — The software displays all red and gray checks. It also displays a few orange checks that are most likely to be run-time errors. When reviewing code for the first time, choose this methodology.

2 **Methodology for Ada > Light** — The software displays all red, gray, and purple checks. The software also displays a subset of orange checks based on specifications in **Polyspace Preferences > Review Configuration**.

3 **Methodology for Ada > Moderate** — The software displays all red, gray, and purple checks. The software also displays a larger subset of orange checks based on specifications in **Polyspace Preferences > Review Configuration**.

4 **All** — The software displays all red, gray, green, and orange checks. When you want to carry out an exhaustive review of your verification results, use this methodology.

You can also define review methodologies customized to your priorities. For more information, see "Review Checks Using Custom Methodologies" on page 8-56.

In the Results Manager perspective, the following toolbar provides controls related to review methodologies.

The controls include:

- A menu for selecting the review methodology.

- Arrows for navigating through checks.

Review Checks Using Predefined Methodologies

This example shows how to incrementally view the checks using predefined review methodologies provided by the code verification software.

Review methodologies specify the number and type of checks displayed on the **Results Summary** pane.

Review Checks Incrementally

1 In the Results manager perspective, from the drop-down list above the **Results Summary** pane, select the methodology, **First checks to review**.
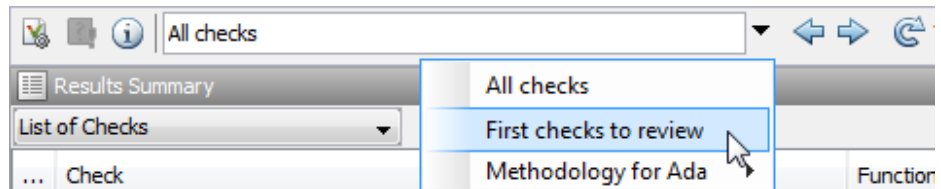
The **Results Summary** pane displays all red and gray checks, as well as orange checks most likely to be run-time errors. Investigate and fix the errors, set the status of the checks, and justify them.

2 Once all the checks have been justified, select the methodology, **Methodology for Ada > Light**.

In the **Results Summary** pane, you can view all red and gray checks, as well as a subset of orange checks. To filter the unjustified checks, from the drop-down list beside the **Justified** column header, clear all boxes except **False** and select **OK**.

...	Check	File	Package	Function	Justified
!	Division by Zero	example.adb	RUNTIME_ERROR	PRO	
!	Arithmetic exceptions	example.adb	RUNTIME_ERROR	SQU	
!	Overflow	array.ada	PKDATA	ARR	
!	Non-terminating call	sensitivity.ada	SENSITIVITY	SOR	
!	Non-terminating call	example.adb	RUNTIME_ERROR	REC	
!	Non-terminating call	example.adb	RUNTIME_ERROR	REC	
!	Non-terminating loop	example.adb	RUNTIME_ERROR	INFI	
!	Non-terminating loop	tasks.adb	PKTASKING	TREGULATE()	
✖	Unreachable code	sensitivity.ada	SENSITIVITY	SORT_CALIB...	

Drop-down overlay on Justified column:
- ☑ (All)
- ☐ (Custom...)
- ☑ False
- ☑ True
- [OK] [Cancel]

Investigate and fix the errors, set the status of the checks, and justify them.

3 Once all the checks have been justified, to see a larger subset of orange checks, select **Methodology for Ada > Moderate**. The number of orange checks on the **Results Summary** pane increases.

To refresh the list to show unjustified checks only, reopen the drop-down list beside the **Justified** column header and select **OK**.

Investigate and fix the errors, set the status of the checks and justify them.

4 To exhaustively review all of the checks, select **All checks**. In addition to all orange checks, this methodology also reveals all the green checks on the **Results Summary** pane.

View Light and Moderate Methodology Requirements

You can view the requirements of the methodologies, **Methodology for Ada > Light** and **Methodology for Ada > Moderate** through the Polyspace Preferences dialog box.

Note You cannot change the parameters specified in any of the predefined methodologies. However, you can create your own custom methodologies.

1 In the Polyspace verification environment, select **Options > Preferences**.

2 In the Polyspace Preferences dialog box, select the **Review configuration** tab.

3 From the drop-down list on this tab, select `Methodology for Ada-Light`.

The table shows the number of orange checks of each type you review when you select this methodology in the Results Manager perspective.

Polyspace Preferences

| Server Configuration | Project and Results Folder | Editors |

| Tools Menu | Review Configuration | Review Statuses | Miscellaneous | Character Encoding |

Methodology

Select a methodology that defines the orange checks you review. With predefined methodologies, such as "Methodology for C", you cannot customize the number of checks to review. However, you can add custom methodologies. To define a methodology, select 'Add a methodology...' from the drop-down list.

The number of checks to review can be either a specific number or a percentage. If you want to use a specific number as your criterion, the value must be:

- A number between 0 and 9999
- The word ALL - all checks
- (Ada only) The word AUTO - automatic check selection

If you want to use a percentage as your criterion, select the check box. The value must be a number between 0 and 100. This value specifies the required percentage of green and justified orange checks. The percentage is calculated as:

$$(Green + Orange\ justified) / (Green + Orange)$$

Methodology for Ada - Light

☐ Specify percentage of green and justified orange checks

Number of checks to review

Common

ZDV	10
NIVL	10
Scalar OVFL	10
COR	10
NIV	5
Float OVFL	5
ASRT	5

C & C++ only

OBAI	
SHF	
IDP	
NIP	
STD_LIB	
ABS_ADDR	

C only

IRV	

C++ only

NNT	
CPP	
FRV	
OOP	
EXC	

Ada only

EXCP	
POW	10

OK   Apply   Cancel

For example, the table specifies that you review ten orange ZDV checks when you select the methodology, `Methodology for Ada-Light`. The number of checks of each type increases as you move from `Methodology for Ada-Light` to `Methodology for Ada-Moderate`.

Related Examples

- "Review Checks Using Custom Methodologies" on page 8-56

Concepts

- "What Are Review Methodologies?" on page 8-49

Review Checks Using Custom Methodologies

This example shows how to define and use a custom review methodology to specify the number and type of checks displayed on the **Results Summary** pane. Define a custom methodology to:

- Prioritize the checks that you review.

- Set standards that all developers of this software must meet.

Define a Custom Methodology

1 In the Polyspace verification environment, select **Options > Preferences**.

2 In the Polyspace Preferences dialog box, select the **Review configuration** tab.

3 From the drop-down list on this tab, select Add a methodology....
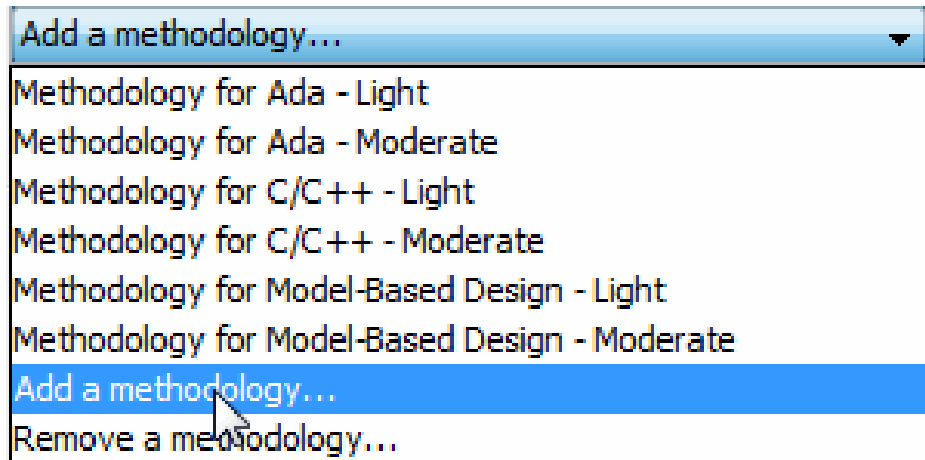
4 Enter a name for your methodology in the Create a new methodology dialog box. For this example, enter the name, My_Methodology. Then, click **Enter**.
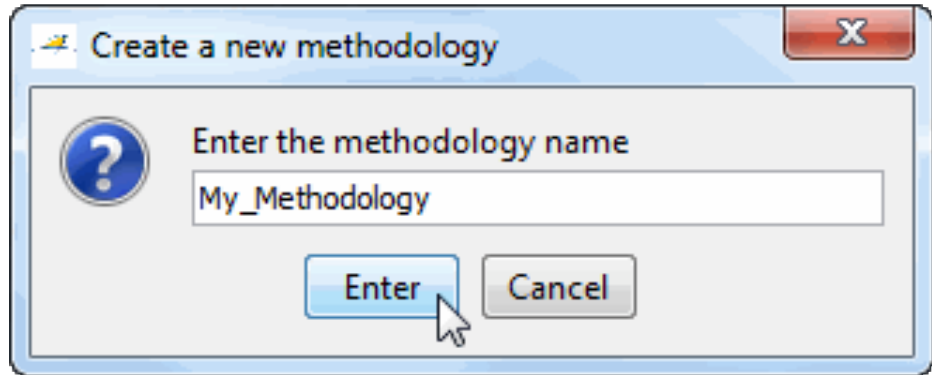
5 If you want to review orange checks by percentage, select the **Specify percentage of green and justified orange checks** check box.

With custom methodologies, you can specify either a specific number of orange checks to review or a minimum percentage of orange checks that must be reviewed. The percentage is calculated by:

```
(green checks + justified orange checks) x 100/(green checks + total orange checks)
```

6 Enter the total number of checks (or percentage of checks) to review for each type of check for your methodology.

My_Methodology

☐ Specify percentage of green and justified orange checks

Number of checks to review

Common

ZDV	5
NIVL	5
Scalar OVFL	5
COR	5
NIV	5
Float OVFL	5
ASRT	5

C & C++ only

OBAI	
SHF	
IDP	
NIP	
STD_LIB	
ABS_ADDR	

C only

IRV	

C++ only

NNT	
CPP	
FRV	
OOP	
EXC	

Ada only

EXCP	
POW	

OK   Apply   Cancel

In this example, 5 was entered for ZDV indicating that 5 Division by Zero orange checks must be displayed when you choose **Custom methodology > My_Methodology** from the Results Manager perspective.

7 Click **OK** to save the methodology and close the dialog box.

Use Your Custom Methodology

In the Results manager perspective, from the drop-down list above the **Results Summary** pane, select the methodology, **Custom methodology > My_Methodology**.
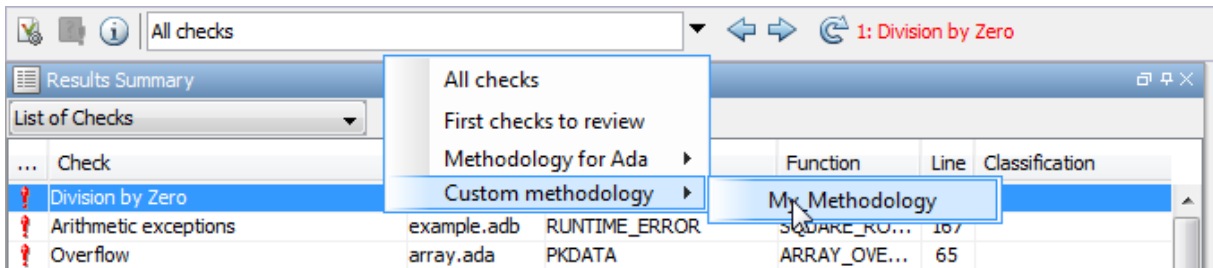
The **Results Summary** pane displays orange checks according to the definition specified for **My_Methodology**. For instance, it displays 5 Division by Zero orange checks.

Related Examples

- "Review Checks Using Predefined Methodologies" on page 8-51

Concepts

- "What Are Review Methodologies?" on page 8-49

Exploring the Results Manager Perspective

Overview

The Results Manager perspective looks like this.
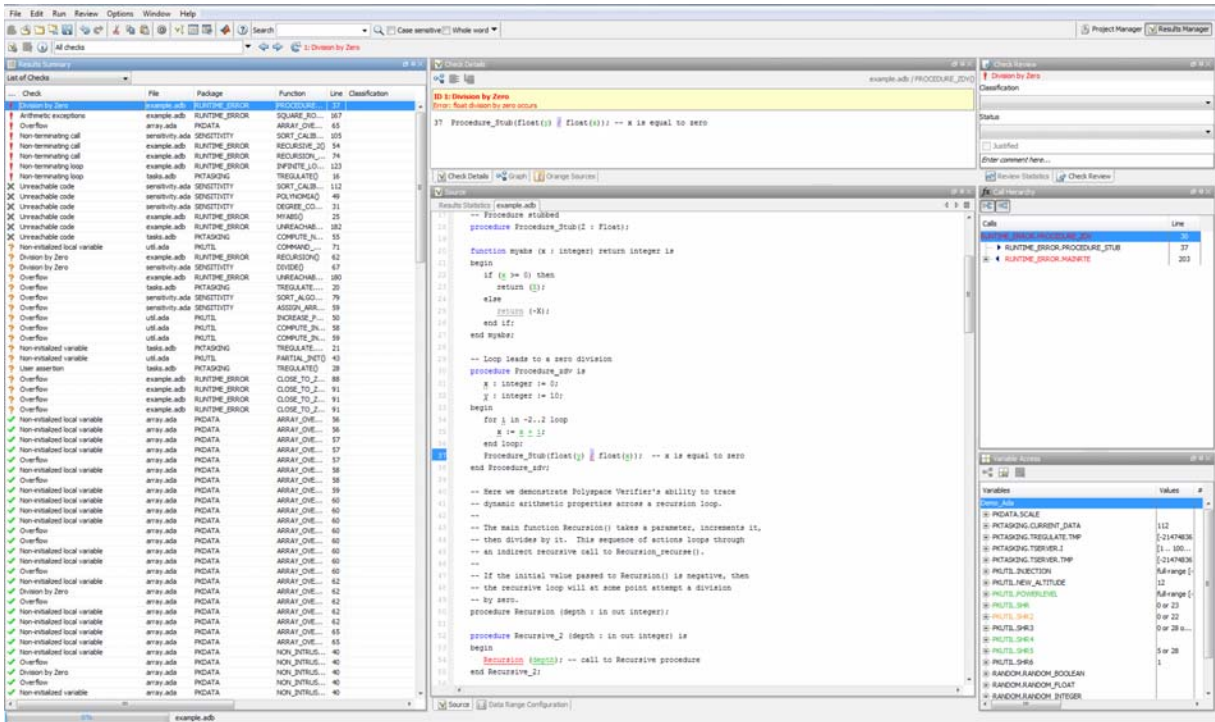
The Results Manager perspective has the following sections below the toolbar:

This pane or view ...	Displays ...
Results Summary	List of checks (diagnostics) for each file and function in the project
Results Statistics	• Graphical view of code coverage and check distribution • Top five orange checks (likely errors in unproven code) and purple checks (coding rule violations)
Source	Source code for a selected check in the **Results Summary** view

This pane or view ...	Displays ...
Check Details	Details about the selected check
Review Statistics	Statistics about the review progress for checks with the same type and category as the selected check
Check Review	Review information about selected check
Variable Access	Information about global variables declared in the source code
Call Hierarchy	Tree structure of function calls

You can resize or hide any of these sections.

Results Summary

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

- `List of Checks`: Lists all checks without any grouping. The checks are sorted in the following order:

  1 **Red**: Indicates code that is proven to contain an error. The check indicates that the code will fail every time it is executed.

  2 **Gray** — Indicates unreachable code.

  3 **Orange** — Indicates unproven code that might contain an error.

  4 **Green** — Indicates code that is proven to not contain an error.

- `Checks by Family`: Lists all checks grouped by color. Within each color, the checks are grouped by category. For more information on the checks covered by a category, see the check reference pages.

- `Checks by File/Function`: Lists all checks grouped by file. Within each file, the checks are grouped by procedure.

- `Checks by Package`: Lists all checks grouped by package. Within each package, the checks are grouped by procedure.

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

Attribute	Description
Family	Group to which the check belongs. For instance, if you choose the grouping `Checks by File/Function`, this column contains the name of the file and function containing the check.
ID	Unique identification number of the check. In the default view on the **Results Summary** pane, the checks appear sorted by this number.
Type	Check color
Category	Category of the check. For more information on the checks covered by a category, see the check reference pages.
Check	Description of the error
Information	For run-time errors, this attribute indicates whether the check is related to path or bounded input values. For coding rule violations, this attribute indicates whether the rule is `Required`.
File	File containing the instruction where the check occurs
Package	Package containing the instruction where the check occurs
Function	Function containing the instruction where the check occurs.

Attribute	Description
Line	Line number of the instruction where the check occurs.
Col	Column number of the instruction where the check occurs. The column number is the number of characters from the beginning of the line.
%	Percentage of checks that are not orange. This column is most useful when you choose the grouping Checks by File/Function. The entry in this column against a file or function indicates the percentage of checks in the file or function that are not orange.
Classification	Level of severity you have assigned to the check. The possible levels are: • Unset • High • Medium • Low • Not a defect
Status	Review status you have assigned to the check. The possible statuses are: • Fix • Improve • Investigate • Justify with annotations • No action planned • Other • Restart with different options

Attribute	Description
Justified	Check boxes showing whether you have justified the checks
Comments	Comments you have entered about the check
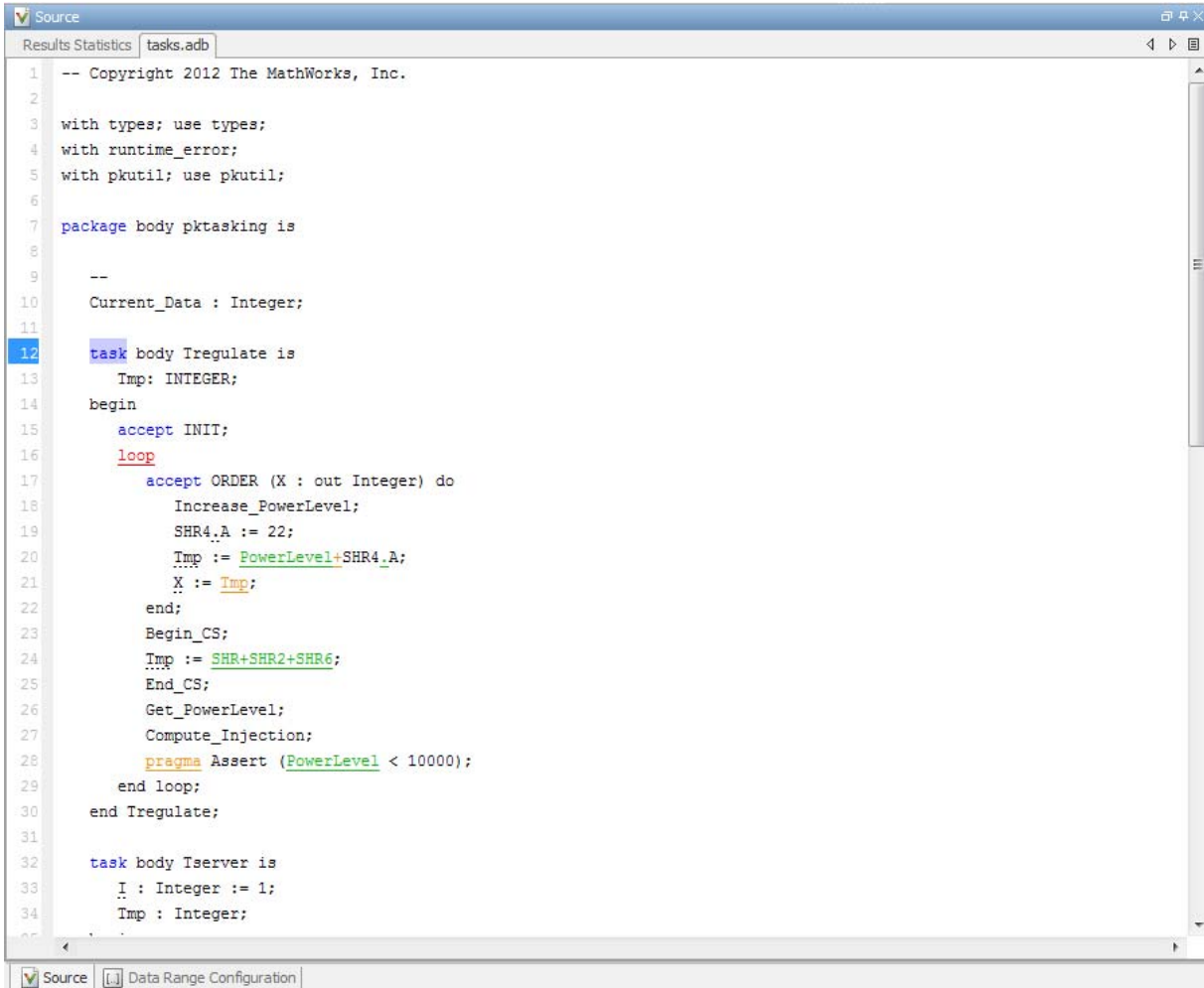
To show or hide any of the columns, right-click anywhere on the column titles. From the context menu, select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through all the checks. For more information, see "Review and Comment Checks" on page 8-23.

- Organize your check review using filters on the appropriate columns. For more information, see "Organize Check Review Using Filters and Groups" on page 8-28.

Source

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

Tooltips

Placing your cursor over a check displays a tooltip that provides range information for variables, operands, function parameters, and return values. For more information on tooltips, see "Using Range Information in Results Manager Perspective" on page 8-100.

Examine Source Code

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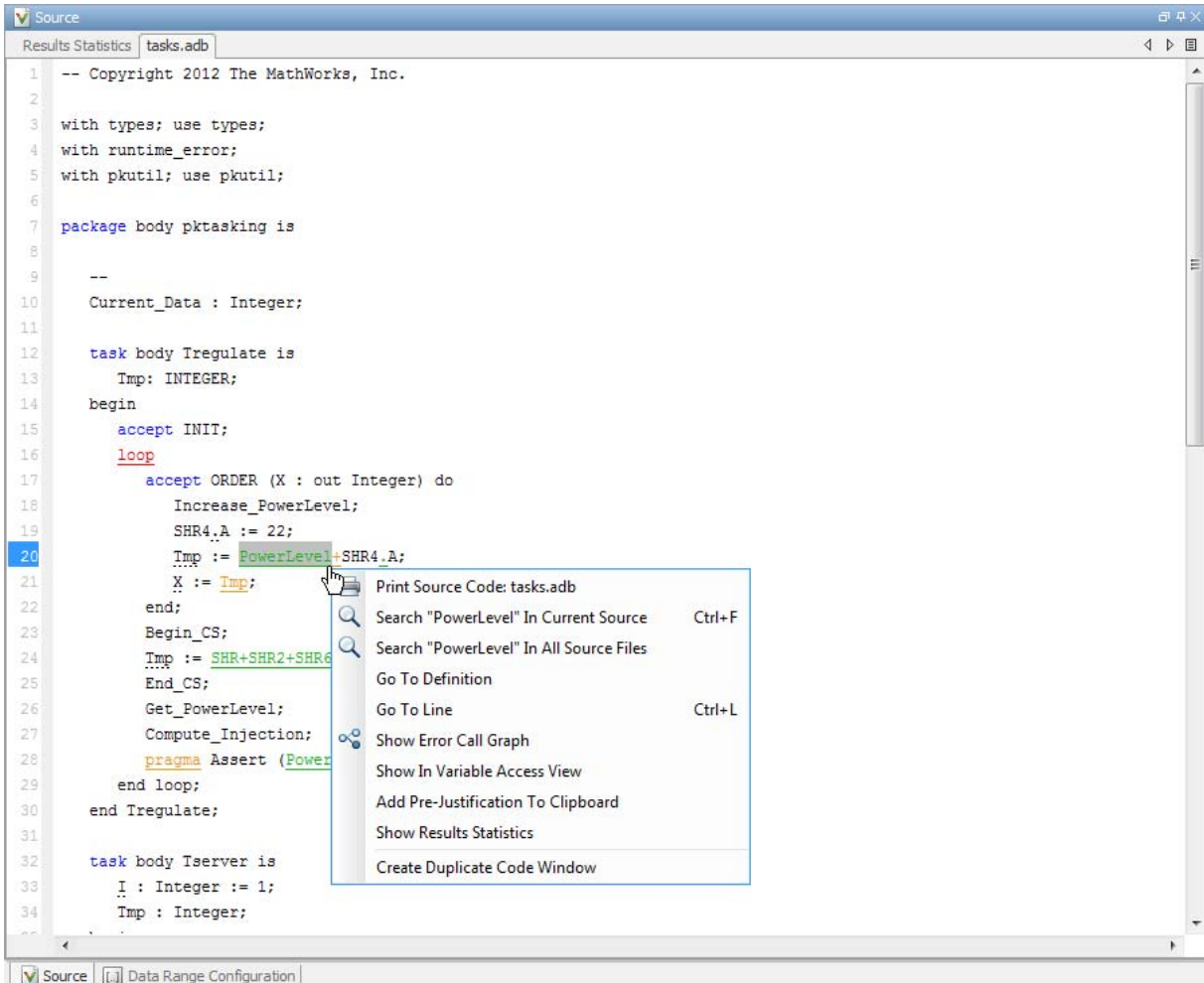
Use the following options to examine and navigate through your code:

- **Search "PowerLevel" in Current Source** — List occurrences of the string within the current source file in the **Search** pane.

- **Search "PowerLevel" in All Source Files** — List occurrences of the string within all source files in the **Search** pane.

- **Go to Definition** — Go to the line of code that contains the declaration of PowerLevel. The software supports this feature for global and local variables, functions and types.

- **Go To Line** — Open the Go to line dialog box. If you specify a line number and click **Enter**, the software displays the specified line of code.

- **Expand All Macros** or **Collapse All Macros** — Display or hide the content of all macros in current source file.

Manage Multiple Files in Source Pane

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

On the **Source** pane toolbar, right-click any view to manage source files.
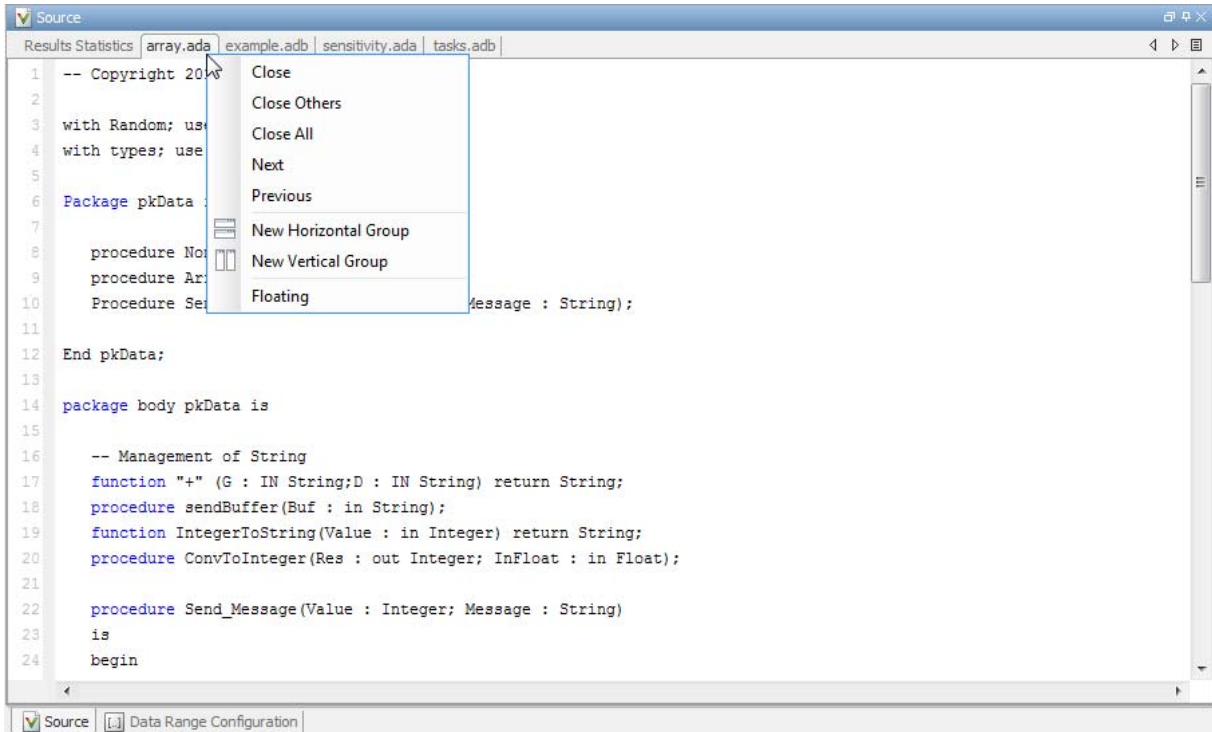
From the **Source** pane context menu, you can:

- **Close** – Close the currently selected source file.

- **Close Others** – Close all source files except the currently selected file.

- **Close All** – Close all source files.

- **Next** – Display the next view.

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

- **New Horizontal Group** – Split the Source window horizontally to display the selected source file below another file.

- **New Vertical Group** – Split the Source window vertically to display the selected source file side-by-side with another file.

- **Floating** – Display the current source file in a new window, outside the Source pane.

Check Details

On the **Results Summary** pane, if you click a check, you see additional information on the **Check Details** pane.
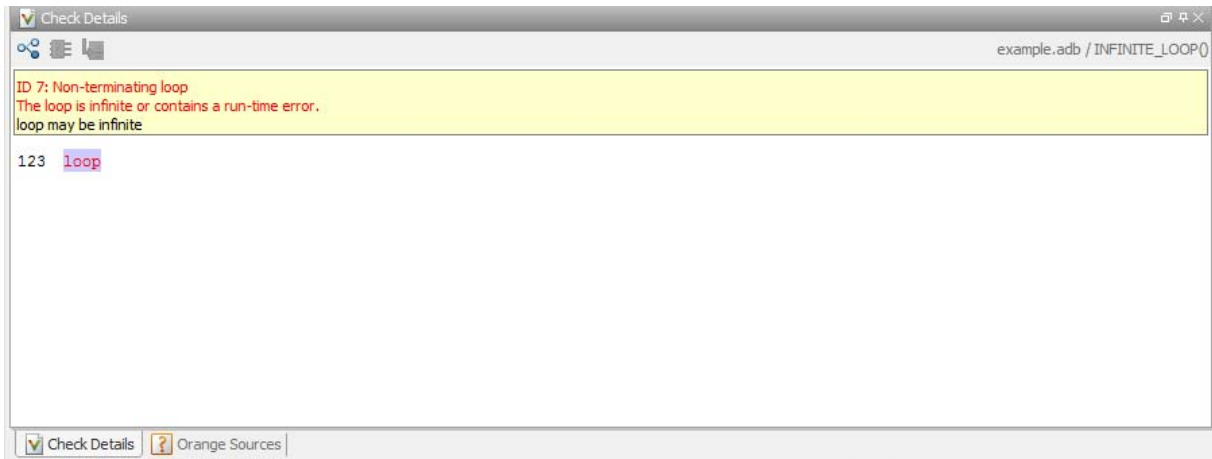
Check Review
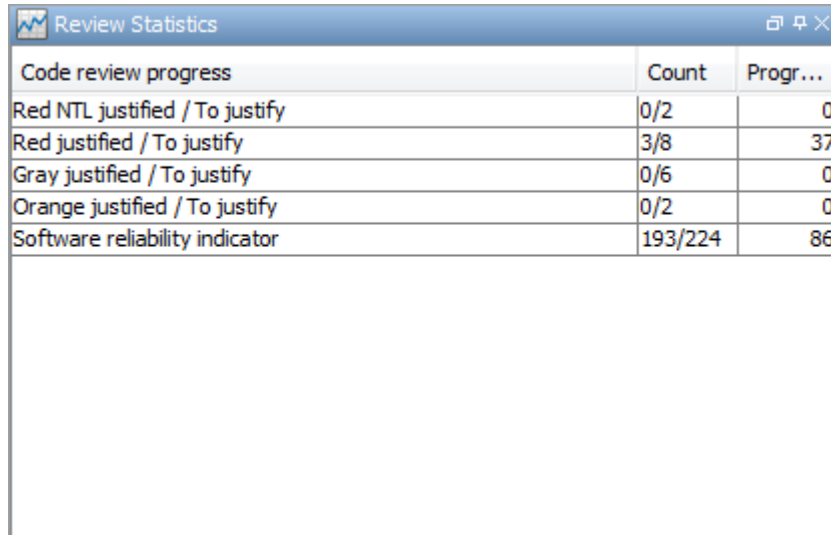
When reviewing checks, use the **Check Review** tab to assign a **Classification** and **Status** to each check. You can also enter comments to describe the results of your review. This action helps you track the progress of your review and avoid reviewing the same check twice.

For more information, see "Review and Comment Checks" on page 8-23.

Review Statistics

The **Review Statistics** view allows you to keep track of the checks that you have reviewed.

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

The second, third, and fourth rows display the ratio of justified checks to total checks for red, gray, and orange checks respectively. The fifth row displays the ratio of the number of green checks to the total number of checks, providing an indicator of the software reliability.

Variable Access

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

Variable Access		

Variables	Values	# Reads
Demo_Ada		
⊞ PKDATA.SCALE		4
⊞ PKTASKING.CURRENT_DATA	112	0
⊞ PKTASKING.TREGULATE.TMP	$[-2147483626 .. 2^{31}-1]$	1
⊞ PKTASKING.TSERVER.I	$[1 .. 10000]$	2
⊞ PKTASKING.TSERVER.TMP	$[-2147483626 .. 2^{31}-1]$	0
⊞ PKUTIL.INJECTION	full-range $[-2^{31} .. 2^{31}-1]$	0
⊞ PKUTIL.NEW_ALTITUDE	12	3
⊞ PKUTIL.POWERLEVEL	full-range $[-2^{31} .. 2^{31}-1]$	7
⊞ PKUTIL.SHR	0 or 23	1
⊞ PKUTIL.SHR2	0 or 22	1
⊞ PKUTIL.SHR3	0 or 28 or 51	1
⊞ PKUTIL.SHR4		2
⊞ PKUTIL.SHR5	5 or 28	2
⊞ PKUTIL.SHR6	1	2
⊞ RANDOM.RANDOM_BOOLEAN		1
⊞ RANDOM.RANDOM_FLOAT		1
⊞ RANDOM.RANDOM_INTEGER		1
⊞ RUNTIME_ERROR.BETA	$[-0.6501 .. -0.2499]$ or $[0.09999 .. 0.5001]$	2
⊞ RUNTIME_ERROR.BIG		1
⊞ SENSITIVITY.ATAB		1

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

Attribute	Description
Variables	Name of Variable, *Package_Name. Variable_NamePackage_Name*: Name of package where variable is declared
Values	Value (or range of values) of variable
# Reads	Number of times the variable is read
# Writes	Number of times the variable is written
Written by task	Name of all tasks writing on variable using aliases, `t1,t2,t3` **Tip** To see the full names for aliases, right-click anywhere on the **Variable Access** pane and select **Show Legend**.
Read by task	Name of all tasks reading variable using aliases, `t1,t2,t3`
Protection	Whether shared variable is protected from concurrent access (Filled only when `Usage` column has entry, `Shared`) The possible entries in this column are: • `Critical Section`: If variable is accessed in critical section of code • `Temporal Exclusion`: If variable is accessed in mutually exclusive tasks For more details on these entries, see "Shared Variables".

Attribute	Description
Usage	Shared, if variable is shared between tasks; otherwise, blank
Line	Line number of variable declaration
Col	Column number (number of characters from beginning of line) of variable declaration
File	Source file containing variable declaration
Detailed Type	Data type of variable

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

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

Attribute	Description
Variables	Names of procedure (or task) performing read/write access on the variable, *Package_Name. Procedure_NamePackage_Name*: Name of package containing procedure (or task) definition
Values	Value or range of values of variable in the procedure or task performing read/write access
Written by task	*Only for tasks*: Name of task performing write access on variable
Read by task	*Only for tasks*: Name of task performing read access on variable

8-75

Attribute	Description
Line	Line number where procedure or task accesses variable
Col	Column number where procedure or task accesses variable
File	Source file containing access operation on variable

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

-

  View Access Graph

  View the access operations on a global variable in graphical format using

  the **Variable Access** pane. Select the global variable and click . For more information, see "Display Access Graph for Variables".

  Here is an example of an access graph:

-
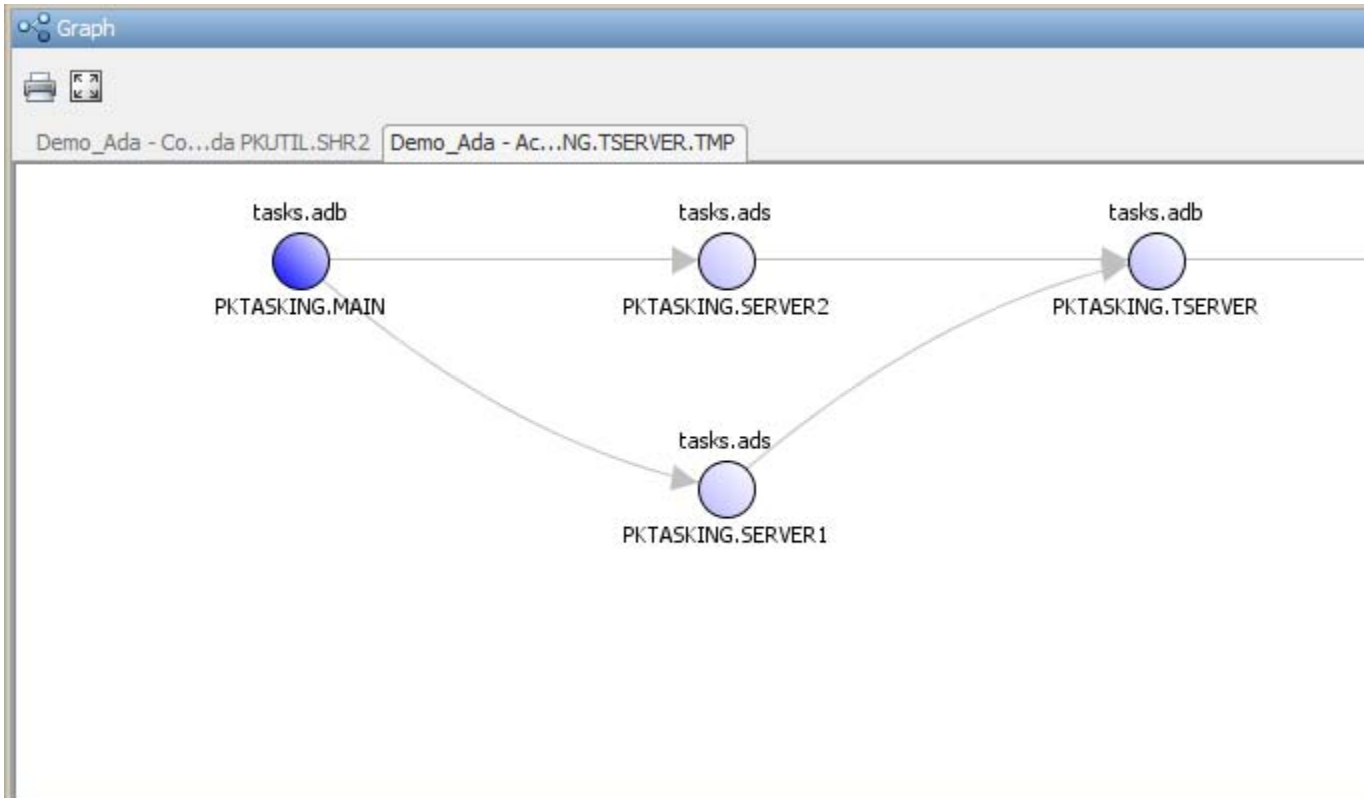
Show/Hide Callers and Callees

Customize the **Variable Access** pane to show only the shared variables.
On the Variable Access pane toolbar, click the Non-Shared Variables

button ![ns] to show or hide non-shared variables.

-

Hide Access in Unreachable Code

Hide read/write access occurring in dead code by clicking the filter button
![filter].

Call Hierarchy

The **Call Hierarchy** pane displays the call tree of procedures in the source code.

For each procedure,foo, the **Call Hierarchy** pane lists the procedures and tasks that call foo (callers) and those called by foo (callees). The callers are indicated by ◀ (procedures), or ◀|| (tasks). The callees are indicated by ▶ (procedures) or ||▶ (tasks).

In the following example, the **Call Hierarchy** pane displays the procedure, SORT_CALIBRATION, in the package, SENSITIVITY. It also displays all the callers and the callees of SORT_CALIBRATION.

fx Call Hierarchy		
Calls	**Line**	
SENSITIVITY.SORT_CALIBRATION	97	
⊞ ▸ SENSITIVITY.SORT_ALGORITHM	101	
⊞ ▸ SENSITIVITY.SORT_ALGORITHM	102	
▸ PKUTIL.READ_ON_BUS	102	
⊞ ▸ SENSITIVITY.SORT_ALGORITHM	103	
▸ PKUTIL.ERROR_ON_BUS	104	
▸ SENSITIVITY.DIVIDE	105	
▸ SENSITIVITY.DIVIDE	107	
⊟ ▸ SENSITIVITY.POLYNOMIA	108	
⊟ ▸ SENSITIVITY.DEGREE_COMPUTATION	43	
▸ SENSITIVITY.RETURN_CODE	31	
▸ SENSITIVITY.RETURN_CODE	49	
⊞ ▸ PKDATA.SEND_MESSAGE	110	
⊞ ▸ PKDATA.SEND_MESSAGE	112	
⊞ ◂ RUNTIME_ERROR.MAINRTE	222	

Depending on the name, the corresponding line number in the **Call Hierarchy** pane refers to a different line in the source code:

- For the function name, the line number refers to the beginning of the procedure definition. In the preceding example, the definition of SENSITIVITY.SORT_CALIBRATION begins on line 97.

- For a callee name, the number refers to the line where the callee is called. In the preceding example, callee, SENSITIVITY.POLYNOMIA, is called by SENSITIVITY.SORT_CALIBRATION on line 108.

- For a caller name, the number refers to the line where the caller calls the procedure. In the preceding example, caller, RUNTIME_ERROR.MAINRTE, calls SENSITIVITY.SORT_CALIBRATION on line 222.

Tip Select a caller or callee name to navigate to the line of code, referred to by the line number, in the **Source** pane (not possible if the caller or callee is a task).

You can perform the following actions from the **Call Hierarchy** pane:

-

Show/Hide Callers and Callees

Customize the view to display callers only or callees only. Show or hide callers and callees by clicking this button

-

Go to Caller/Callee Definition

Go directly to the definition of a caller or callee in the source code. Right-click the name of the caller or callee and select **Go to definition**. .

Import and Export Review Comments

In this section...
"Reusing Review Comments" on page 8-81
"Importing Review Comments from Previous Verifications" on page 8-81
"Viewing Checks and Comments Report" on page 8-82

Reusing Review Comments

After you have reviewed verification results on a module, you can reuse your review comments with subsequent verifications of the same module. You avoid reviewing the same check twice and you can compare results over time.

The Results Manager perspective allows you to either:

- Import review comments from another set of results into the current results.

- Export review comments from the current results to a spreadsheet.

You can also generate a report that compares the source code and verification results from two verifications, and highlights differences in the results.

Note If the code has changed since the previous verification, the imported comments might not be applicable to your current results. For example, the justification for an orange check might no longer be relevant to the current code. Open the Import/Export Report to see changes that affect your review comments.

Importing Review Comments from Previous Verifications

If you have previously reviewed verification results for a module and saved your comments, you can import those comments into the current verification, allowing you to avoid reviewing the same check twice.

To import review comments from a previous verification:

1 Open your most recent verification results in the Results Manager perspective.

2 Select **Review > Import > Import Comments**.

3 Navigate to the folder containing your previous results.

4 Select the results file with extension `.rte`, then click **Open**.

   The review comments from the previous results are imported into the current results. The Import checks and comments report opens. For more information, see "Viewing Checks and Comments Report" on page 8-82.

To automatically import comments from the last verification, before running your verification:

1 Select **Options > Preferences**, which opens the Polyspace Preferences dialog box.

2 Select the **Project and result folder** tab.

3 Under **Import Comments**, select the **Automatically import comments from last verification** check box.

4 Click **OK**.

Once you import checks and comments, the ▶ button ignores any reviewed checks. This behavior allows you to review only previously unreviewed checks.

Note If the code has changed since the previous verification, the imported comments might not be applicable to your current results. For example, the justification for an orange check might no longer be relevant to the current code.

Viewing Checks and Comments Report

Importing review comments from a previous verification can be extremely useful, since it allows you to avoid reviewing checks twice, and to compare verification results over time.

However, if your code has changed since the previous verification, or if you have upgraded to a new version of the software, the imported comments might not be applicable to your current results. For example, the color of a check may have changed, or the justification for an orange check may no longer be relevant to the current code.

The Import/Export checks and comments report allows you to compare the source code and verification results from a previous verification to the current verification, and highlights differences in the results.

To view the Import/Export checks and comments report:

1 Select **Review > Import > Open Import Report**.

The Import/Export checks and comments report opens, highlighting differences in the two results, such as unmatched lines and changes to the color of checks.

If the color of a check changes, the previous review comments are imported, but the check is not marked as reviewed.

If a check no longer appears in the code, the report highlights the change, but the software does not import any comments on the check.

Polyspace Report Generator Overview

The Polyspace Report Generator allows you to generate reports about your verification results, using predefined report templates.

The Polyspace Report Generator provides the following report templates:

- **Coding Rules Report** — Provides information about compliance with Coding Rules, as well as Polyspace configuration settings for the verification.

- **Developer Report** — Provides information useful to developers, including summary results, detailed lists of red, orange, and gray checks, and Polyspace configuration settings for the verification. Detailed results are sorted by type of check (Proven Run-Time Violations, Proven Unreachable Code Branches, Unreachable Functions, and Unproven Run-Time Checks).

- **Developer Review Report** – Provides the same information as the Developer Report, but reviewed results are sorted by review acronym (NOW, NXT, ROB, DEF, MIN, OTH) and untagged checks are sorted by file location.

- **Developer with Green Checks Report** — Provides the same content as the Developer Report, but also includes a detailed list of green checks.

- **Quality Report** — Provides information useful to quality engineers, including summary results, statistics about the code, graphs showing distributions of checks per file, and Polyspace configuration settings for the verification.

- **Software Quality Objectives Report** — Provides comprehensive information on software quality objectives (SQO), including code metrics, code verification (run-time checks), and the configuration settings for the verification. The code metrics section provides the same information as the **Code Metrics** view of Polyspace Metrics.

Using the Polyspace Report Generator, you can generate verification reports in the following formats:

- HTML
- PDF (version 1.4)

- RTF
- DOC (Microsoft® Word version 2003 or later)
- XML (DocBook 4.2)

Note Microsoft Word format is not available on UNIX platforms. Use the RTF format instead.

Generate Verification Reports

You can generate reports for any verification results using the Polyspace Report Generator.

To generate a verification report:

1 In the Results Manager perspective, open your verification results.

2 Select **Run > Run Report > Run Report**. The Run Report dialog box opens.

3 In the **Select Reports** section, select the types of reports that you want to generate. For example, you can select **Developer** and **Quality**.
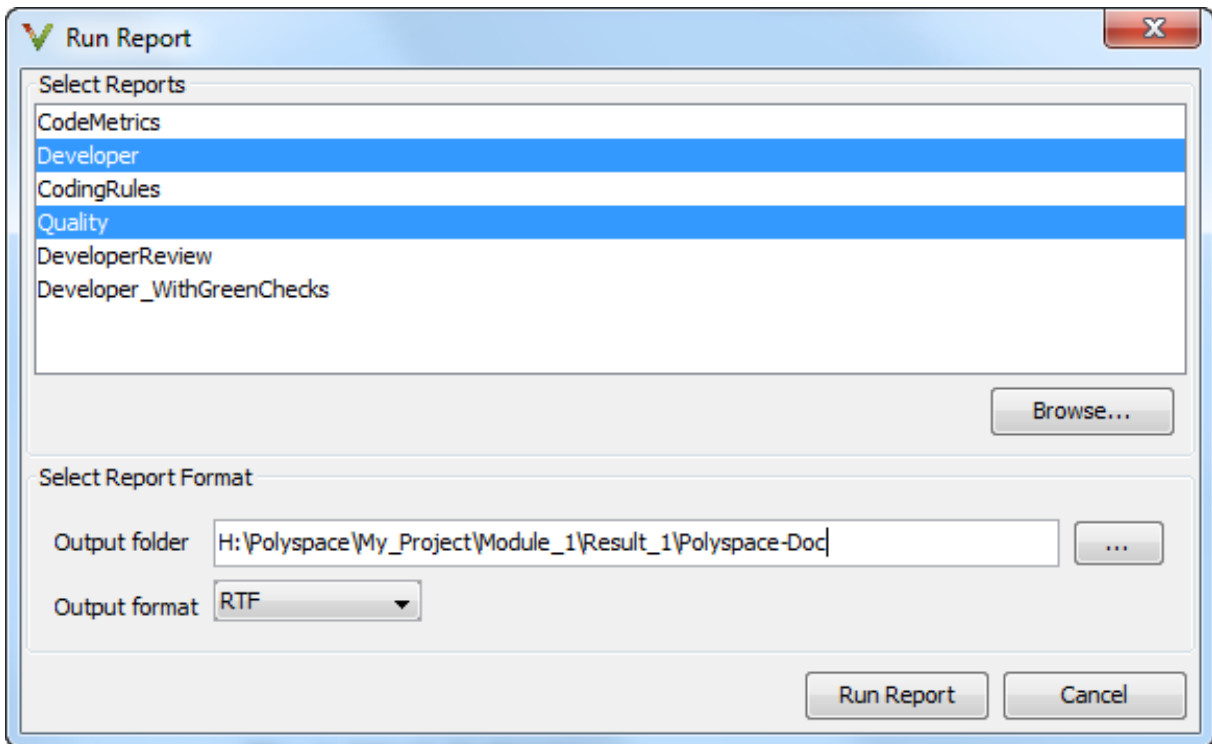
4 Select the Output folder in which to save the reports.

5 Select the Output format for the reports.

6 Click **Run Report**.

The software creates the specified reports.

Run Report Generator from Command Line

You can also run the Report Generator, with options, from the command line. For example:

Polyspace_Install\polyspace\bin\report-generator -template *path*
-format *type* -results-dir *folder_paths*

For information about the available options, see the following sections.

-template *path*

Specify the *path* to a valid Report Generator template file. For example:

Polyspace_Install\polyspace\toolbox\psrptgen\templates\Developer.rpt

Other supplied templates are CodingRules.rpt,
Developer_WithGreenChecks.rpt, DeveloperReview.rpt, and Quality.rpt.

-format *type*

Specify the format *type* of the report. Use HTML, PDF, RTF, WORD, or XML. The default is RTF.

-help or -h

Displays help information.

-noview

The software does not open the report at the end of the generation process.

Note If you use a Linux® system and want to run the Report Generator from the command line with the -format RTF option, you must also specify the -noview option.

-output-name *filename*

Specify the *filename* for the report generated.

-results-dir *folder_paths*

Specify the paths to the folders that contain your verification results.

You can generate a single report for multiple verifications by specifying *folder_paths* as follows:

```
"folder1, folder2, folder3,..., folderN"
```

folder1, folder2, ... are the file paths to the folders that contain the results of your verifications (normal or unit-by-unit). For example:

```
"C:\Results1,C:\Recent\results,C:\Old"
```

If you do not specify a folder path, the software uses verification results from the current folder.

Automatically Generate Verification Reports

You can specify that Polyspace software automatically generate reports for each verification using an option in the Project Manager perspective.

Note You cannot generate reports of software quality objectives automatically.

To automatically generate reports for each verification:

1 In the Project Manager perspective, open your project.

2 Select the **Configuration > Reporting** pane.

3 Select the **Generate report** check box.

4 From the **Report template** drop-down list, select a template.

5 From the **Output format** drop-down list, select a format for the report.

6 Save your project.

Customizing Verification Reports

If you have MATLAB® Report Generator™ software installed on your system, you can customize the Polyspace report templates or create your own reports. You can then generate these custom reports using the Polyspace Report Generator.

Before you can customize Polyspace reports, you must configure the MATLAB Report Generator software to access the following folders:

- **Custom components** –
  Polyspace_Install/polyspace/toolbox/psrptgen/psrptgen

- **Report templates** –
  Polyspace_Install/polyspace/toolbox/psrptgen/templates

To customize a Polyspace report:

1 Open MATLAB.

2 Add the Polyspace reports custom components folder to the MATLAB search path, using the following command:

addpath(`*Polyspace_Install*/polyspace/toolbox/psrptgen/psrptgen')

3 Set the current folder in MATLAB to the Polyspace reports template folder, using the following command:

cd('*Polyspace_Install*/polyspace/toolbox/psrptgen/templates')

4 Open the Report Explorer using the following command:

report
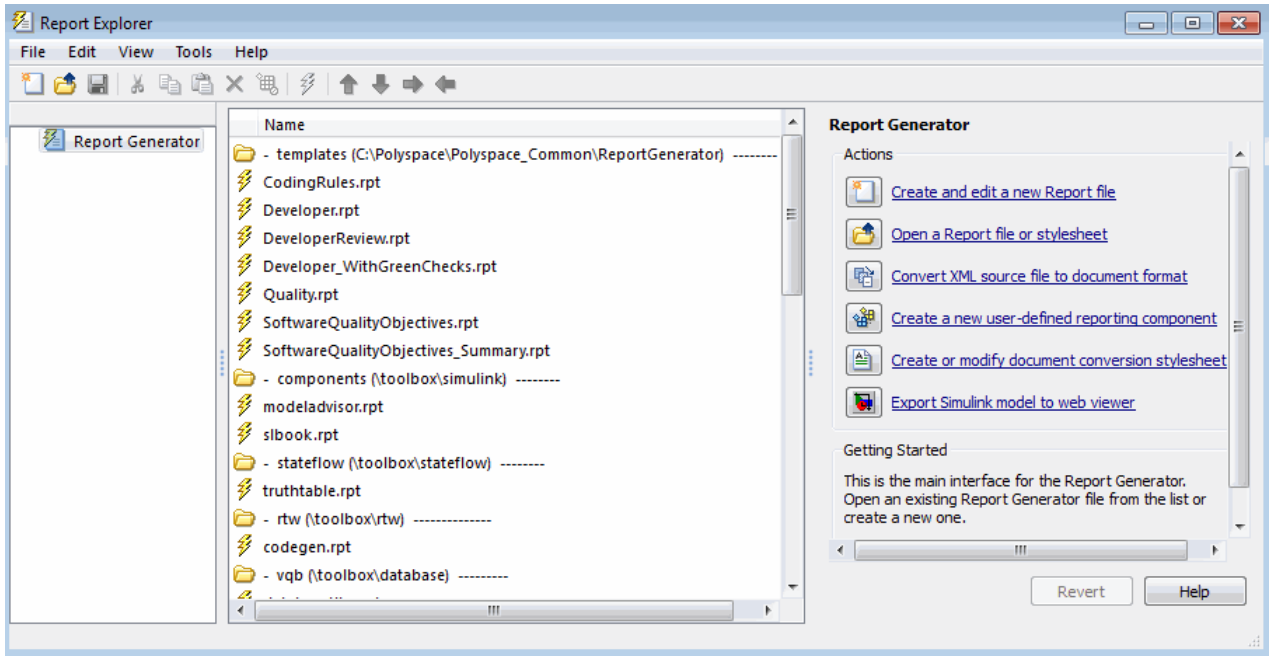
5 Open the template that you want to customize. For example, double-click
Developer.rpt. You can add, delete, or alter components of this template.

6 To save the modified template, select **File > Save As**. The
Save Report File as dialog box opens. If you are not in the
Polyspace_Install/polyspace/toolbox/psrptgen/templates folder,
navigate to this folder.

7 In the **File name** field, specify the name of the modified template, for
example Developer_modified.rpt. Then, click **Save**.

When you next generate a report, Developer_modified.rpt will be available
as an option in the Run Report dialog box.

Note To produce custom reports with the Polyspace
Report Generator, you must save the report template in:
Polyspace_Install/polyspace/toolbox/psrptgen/templates.

You can use the **Report Customization (Filtering) component** to apply
filters to your report. This component provides a number of filters, for
example, for code metrics, coding rules, and run-time checks.

1 From the **Name** list, under **Polyspace**, select the **Report Customization
(Filtering)** component. Then drag this component to the required point in
the template. For example, if you want to apply global filters to the report,
place the component above the **Title Page** component.
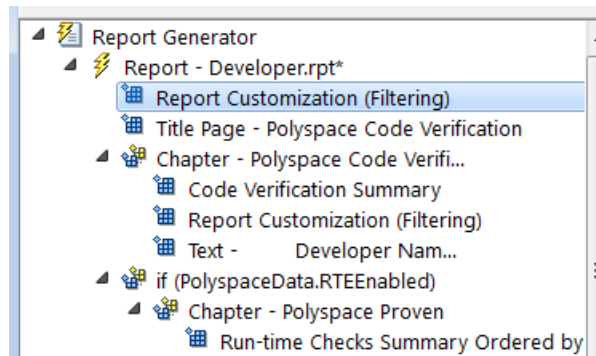
2 Select the **Report Customization (filtering)** component.

3 On the right of the dialog box, specify your filters. For example, you might
want your report to contain only specific run-time checks. In addition,
you might want to see these checks for only certain functions. Under
Advanced Filters:

- In the **Check types to include** field, enter the run-time checks, for
  example, ZDV and OVFL.

- In the **Function names to include** field, enter the functions, for
  example, function1 and functionX.

Note You can also exclude items. For example, to exclude all comments containing the string GEN, in the **Comments to include** field, enter the expression `^(?:(?!GEN).)*$`. For more information about regular expressions, refer to *Regular Expressions* in the MATLAB documentation.

4 Save the modified template.

For the **Run-time Check Details Ordered by Color/File** component, you can override the global filters and specify different filters for this component. For example, you might want a report chapter that contains NIV checks filtered by the global filter.

To override global filters:

1 Select the **Run-time Check Details Ordered by Color/File** component.
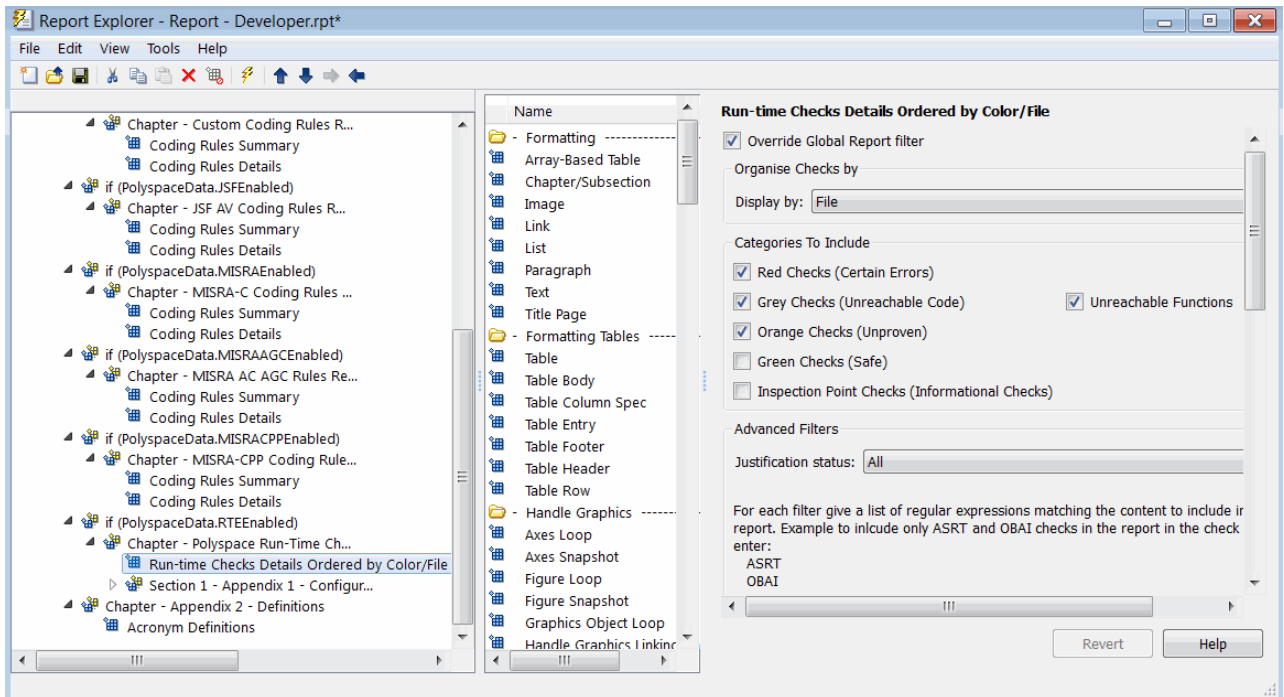
2 On the right of the dialog box, select the **Override Global Report filter** check box.

3 Specify your filters for this component. For example, in the **Check types to include** field, enter NIV.

4 Save the template.

For more information, refer to the MATLAB Report Generator documentation.

Open Verification Report

To view a verification results report:

1 From the Results Manager toolbar, select **Run > Run Report > Open Report**, which opens the Open Report dialog box.

2 Navigate to the folder that contains your report. Then select the report.
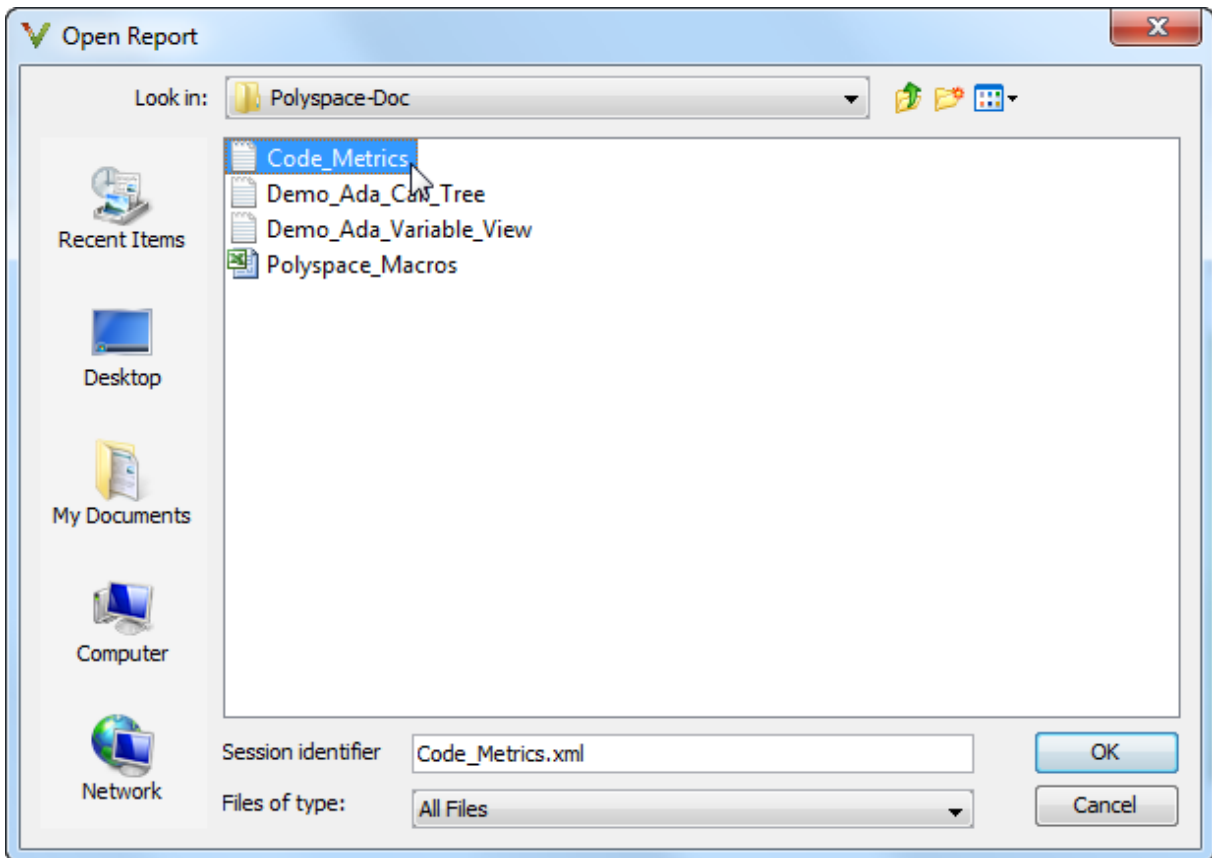
3 Click **OK**.

Note Before you can open a verification report, you must specify a text editor. See "Configuring Text Editor" on page 3-21.

Using Polyspace Results

In this section...

Review Run-Time Errors: Fix Red Errors

All run-time 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 mathematically incorrect. If the value represents the altitude of a plane, this result could be disastrous.

By default, Polyspace verification does not 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. A Polyspace option might be used to allow particular compiler specific behavior. In other situations, the code must be corrected. An example of a Polyspace option to permit compiler specific behavior is the option to force "IN/OUT" ADA function parameters to be initialized. Examples in C include options to deal with constant overflows, shift operation on negative values, etc.

- All other red errors must be fixed. They are bugs.

Most of the bugs that you find are easy to rectify once they are identified. Polyspace verification identifies bugs regardless of their consequence or the ease with which they can be corrected.

Using Range Information in Results Manager Perspective

Viewing Range Information

Within the **Source** pane, you can see range information associated with variables. Place your cursor over a variable. A tooltip displays the range information, if it is available. The software displays tooltips only for variables with a scalar, enumeration, or float data type.

The displayed range information represents a superset of dynamic values, which the software computes using static methods.

Note Computing range information for read operations may take a long time. You can reduce verification time by limiting the amount of range information displayed in verification results. See"`-less-range-information`".

If you click a check, the software provides information about the check in the **Check Details** pane.

Interpreting Range Information

To display range information of variables, the software uses the following syntax:

variable_name (type: *data_type* [*data_type_range*]) :
[*min1 .. max1*] or [*min2 .. max2*] or [*min3 .. max3*] or *exact_value*

In the following example:

```
80          if   Y then
81              New_Alt := New_Alt * 10;
82          end if
83      end;
```
assignment of variable 'new_alt' (type: integer [$-2^{31}..2^{31}-1$]): 120

the tooltip indicates that the variable New_Alt is a 32-bit integer with the value 120. The range of the data type is -2^{31} to $2^{31} - 1$.

In the next example, the 32-bit integer depth lies either between –3 and 0 or is equal to 11.

```
69          depth := depth + 1;
70          advance := float(1)/float(depth);  -- potential zero division
71          if depth < 5 then
72              Recursive_2 (depth);
```
variable 'depth' (type: integer [$-2^{31}..2^{31}-1$]): [-3 .. 0] or 11

With the next case:

```
43          Square_Root_conv (Alpha, Beta);
44          Beta := Beta - 0.75;
45          Gamma :=
46      end Square_Ro
```
variable 'beta' (type: long_float [float64 $-1.8E^{+308}..1.79E^{+308}$]): [$9.9999E^{-2}$ .. $5.0001E^{-1}$]

the tooltip indicates that the variable Beta is a 64-bit float that lies between 9.9999E-2 and 5.0001E-1. The range of the data type is –1.8E+308 to 1.79E+308.

The tooltip can also reveal whether the variable occupies the full range:

```
43      Square_Root_conv (Alpha, Beta);
44      Beta := Beta - 0.7
45      Gamma := sqrt(Beta
```
variable 'alpha' (type: float [float32 $-3.41E^{+38}..3.4E^{+38}$]): full-range [$-3.4029E^{+38}$ .. $3.4029E^{+38}$]

The message shows that the value of the variable `Alpha` is a 32-bit `float` that occupies the full range of the data type, −3.4029E+38 to 3.4029E+38.

Consider a procedure with an `in out` parameter:

```
127  procedure proc(X : in out Integer) is
128  begin
129     X :=
130     X
131     +
132     100;
133  end;
```

The procedure is called with a parameter `X`:

```
180      procedure main is
181      begin
182
183          R.F1 := 12;
184          Proc
185            (
186               X
187            )
188      Proc assignment of variable 'x' (type: integer [-2^31..2^31-1]): 104
189        (
190          variable 'x' (type: integer [-2^31..2^31-1]): 4
```

The first line of the tooltip provides information about the `out` value of `X`, a 32-bit integer with the value 104. The second line gives information about the `in` value of `X`, a 32-bit integer with the value 4.

If a procedure has only an `out` parameter, for example:

```
148  procedure Proc_Init_Out(X : out Integer) is
149  begin
150     X := 100;
151     null;
152  end;
```

the tooltip is a single line with information about the `out` parameter:

```
214          Niv: Integer;
215        begin
216          Proc_Init_Out(Niv);
217          pragma  Assert()
218        end;                      assignment of variable 'niv' (type: integer [-2^31..2^31-1]): 100
```

Why Review Dead Code Checks

- "Functional Bugs in Gray Code" on page 8-103
- "Structural Coverage" on page 8-104

Functional Bugs in Gray Code

Polyspace verification finds different types of dead code. Common examples include:

- Defensive code which is never reached.

- Dead code due to a particular configuration.

- Libraries which are not used to their full extent in a particular context.

- Dead code resulting from bugs in the source code.

The causes of dead code listed in the following examples are taken from critical applications of embedded software by Polyspace verification.

- A lack of parenthesis and operand priorities in the testing clause can change the meaning significantly.

- Consider a line of code such as

  IF NOT a AND b OR c AND d

  Now consider how misplaced parentheses might influence how that line behaves

  IF NOT (a AND b OR c AND d)

  IF (NOT (a) AND b) OR (c AND d))

  IF NOT (a AND (b OR c) AND d)

- The test of variable inside a branch where the conditions are never met.

- An unreachable "else" clause where the wrong variable is tested in the "if" statement.
- A variable that is supposed to be local to the file but instead is local to the function.
- Wrong variable prototyping leading to a comparison which is always false.

The consequences of dead code and the effort to deal with it is unpredictable. From a one-week effort of functional testing on target, trying to build a scenario going into that branch, and wondering why the functional behavior is altered to a three-minute code review discovering the bug.

Polyspace does not measure the impact of dead code.

The tool provides a list of dead code. A short code review enables you to place each entry from that list into one of the five categories. Doing so identifies known dead code and uncovers real bugs.

The Polyspace experience is that at least 30% of gray code reveals real bugs.

Structural Coverage

Polyspace software 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. It could conclude that no run-time error could be found.

Polyspace verification finds around 80% of dead code that the developer would find by doing structural coverage.

Polyspace verification is intended to be a productivity aid in dead code detection. It detects dead code which might take days of effort to find by any other means.

Reviewing Orange: Automatic Methodology

During a verification, Polyspace can automatically highlight orange checks associated with code that might not be robust.

The automatic methodology separates the following orange NIVL and OVFL checks from other orange checks:

- All NIVL scalar local orange checks. These checks do not concern floats, records (and components), and arrays.

- All OVFL scalar orange checks within subtypes: conversion of a subtype into a smaller subtype.

You must address these checks first. Polyspace is very precise with these checks. These checks indicate issues of robustness.

Example

```
1 Package body Test is
2  ATab : array(0..9) of Integer := (Others => 0);
3  function Assign_array(X : integer) return Integer is
4    Y : Integer;
5  begin
6    y := ATab(X - 12);  -- Warning OVFL 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;
```

8-105

In this example, the automatic methodology filters all orange checks except:

- OVFL at line 6. The associated message is:

```
Unproven : operation [-] on scalar may overflow
(on MIN or MAX bounds of integer [from -2^63 .. 2^63-1 to 0..9])
```

Line 6 is an example of a conversion within a smaller subtype. There is no neighboring code that ensures robustness.

- NIVL at line 21. The associated message is:

```
Warning : local variable may be non-initialized
variable 'y' (type: boolean [false..true]): 1
```

Line 21 is an example where there is a robustness issue if the right branch is not executed.

Reviewing Orange Checks

Orange checks indicate *unproven code*. The code cannot be proved to either have or not have a run-time error.

The number of orange checks that you review is determined by several factors, including:

- The stage of the development process
- Your quality goals

There are also actions you can take to reduce the number of orange checks in your results.

For more information, see "Orange Check Management".

Integration Bug Tracking

By default, integration bug tracking can be achieved by applying the selective orange methodology to integrated code. Each error category is more likely to reveal integration bugs, depending on the chosen coding rules for the project.

For instance, consider a function that receives two unbounded integers. The presence of an overflow can be checked only at integration phase, because at unit phase the first mathematical operation reveals an orange check.

Consider the following cases:

- When integration bug tracking is performed in isolation, a selective orange review highlights 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.

With the second case, as an exhaustive orange review has been performed file by file, only checks that have turned from green to another color at integration phase 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. The verification displays a green NIV check at the first read access to a field. But this hypothesis might not be true at integration time, where the check can turn orange if the field is not initialized.

This type of orange checks reveals integration bugs.

Finding Bugs in Unprotected Shared Data

Based on the list of entry points in a multitask 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 while another task is in the process of doing so. The following is a possible scenario:

- As there is conflict with a particular variable, a bug exists and protection is required.

If this is not a possible scenario, then one of the following explanations might apply:

- The compilation environment guarantees an atomic read or write access for variables of a type that is less than one or two bytes in size. Therefore, the integrity of the variables' content is maintained even if there are conflicts associated with these variables.

- The variable is protected by a critical section or a mutual temporal exclusion. You may want to include this information in the Polyspace project configuration parameters and rerun the verification.

It is also worth checking whether variables that are supposed to be constant are modified . 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 and write access operations on global variables. (This dictionary 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.

Checks on Procedure Calls with Default Parameters

Some checks may be located on procedure calls. They correspond to default values assigned to parameters of a procedure.

Example

```
1  package DCHECK is
2   type Pixel is
3    record
4     X : Integer;
5     Y : Integer;
6    end record;
7   procedure MAIN;
8
9   NError : Integer;
10   procedure Failure (Val : Integer := Nerror);
11   procedure MessageFailure (str : String := "");
12  end DCHECK;
13
14  package body DCHECK is
15   type TwentyFloat is array (Integer range 1.. 20) of Float;
16
17   procedure AddPixelValue(Vpixel : Pixel) is
18   begin
19    if (Vpixel.X < 3) then
20     Failure; -- NIV Verified: Variable is initialized
(Nerror)
21     MessageFailure; --COR Verified: Value is in range (string)
22    end if;
23   end AddPixelValue;
24
25   procedure MAIN is
26    B : Twentyfloat;
27    Vpixel : Pixel;
28   begin
29    NError := 12;
30    Vpixel.X := 1;
31    AddPixelValue(Vpixel);
32    NError := -1;
```

```
33    for I in 2 .. Twentyfloat'Last loop
34     if ((I mod 2) = 0) then
35      B(I) := 0.0;
36      if (I mod 2) /= 0 then
37       Failure; -- NIV Unreachable: Variable is not
initialized
38        MessageFailure; -- COR Unreachable: Value is not in range
39      end if;
40     end if;
41    end loop;
42    MessageFailure("end of Main");
43   end MAIN;
44  end DCHECK;
```

Explanation

In the previous example, at line 20 and 37, checks on the procedure called Failure represent the check NIV made on the default parameter N error (a global parameter).

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 Results Manager perspective, you might find nodes _INIT_PROC$ in the "Procedural entities" view. As your compiler, Polyspace

generates a function _INIT_PROC for each record where initialization occurs. When a package defines many records, each _INIT_PROC is differentiated by $I (I in 1.n).

Example

```
1  package test is
2   procedure main;
3  end test;
4
5  package body test is
6
7    subtype range_0_3 is integer range 0..3;
8   Vg : Integer := 1;
9   Pragma Volatile( Vg );
10
11   function random return integer;
12   type my_rec1 is
13    record
14     a : integer := 2 + random; -- Unproven OVFL coming from
_INIT_PROC procedure (initialization of V1)
15     b : float := 0.2;
16    end record;
17   V1 : my_rec1;
18   V2 : my_rec1 := (10, 10.10);
19
20   procedure main is
21    Function Random return Boolean;
22   begin
23    null;
24   end;
25  end test;
```

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. A random procedure could return any value in the integer type and lead to an overflow by adding to 2. The check is located in the _INIT_PROC node in the **Results Summary** view.

Pointers to Explicit Tasks

If a task type is used through a pointer, then Polyspace automatically adds two instances of this task type to the Polyspace execution model of your application. All task pointer objects that are used in your application are represented by these two instances. Polyspace uses these instances to simulate all tasks associated with the execution of your application.

Consider the following example.

```
package Test is
   task type Ressource_T is
      entry Get (X : out integer);
      entry Set (X : in integer);
   end Ressource_T;
   type Ressource_Ptr_T is access Ressource_T;
   Ressource_Ptr : Ressource_Ptr_T;
   V : Integer := 0;
   function Alloc_Ressource return Ressource_Ptr_T;
   procedure Test_Ressource;
private
end Test;

package body Test is
   task body Ressource_T is
      Random : Boolean;
      pragma Volatile (Random);
   begin
      while Random loop
         select
            accept Get (X : out Integer) do
               X := V + 2;
            end Get;
         or
            accept Set(X : in Integer) do
               V := X - 2;
            end Set;
         end select;
      end loop;
   end Ressource_T;
```

```
     function Alloc_Ressource return Ressource_Ptr_T is
     begin
        return new Ressource_T;
     end Alloc_Ressource;

     procedure Test_Ressource is
       X : Integer;
       Tp : Ressource_Ptr_T;
     begin
        Tp := Alloc_Ressource;
        Tp.Get(X);
     end Test_Ressource;
end Test;
```

At the end of verification, in the **Results Summary** view, you see two
instances of the task type Ressource_T, that is, PST_Ressource_T_1 and
PST_Ressource_T_2.

9

Managing Orange Checks

What Is an Orange Check?

Orange checks indicate *unproven code*, which means the software cannot prove that the code:

- Produces a run-time error

- Does not produce a run-time error

Polyspace verification attempts to prove the absence or existence of run-time errors. Therefore, the software considers all code unproven before a verification. During a verification, the software attempts to prove that the code is:

- Without run-time errors (green)

- Certain to fail (red)

- Unreachable (gray)

Any code that is not assigned one of these categories (colors) stays unproven (orange).

Code often remains unproven in situations where some paths fail while others succeed. For example, consider the following instruction:

```
X = 1 / (X - Y);
```

Does a division-by-zero error occur?

The answer depends on the values of X and Y. However, there are an almost infinite number of possible values. Creating test cases for all possible values is not practical.
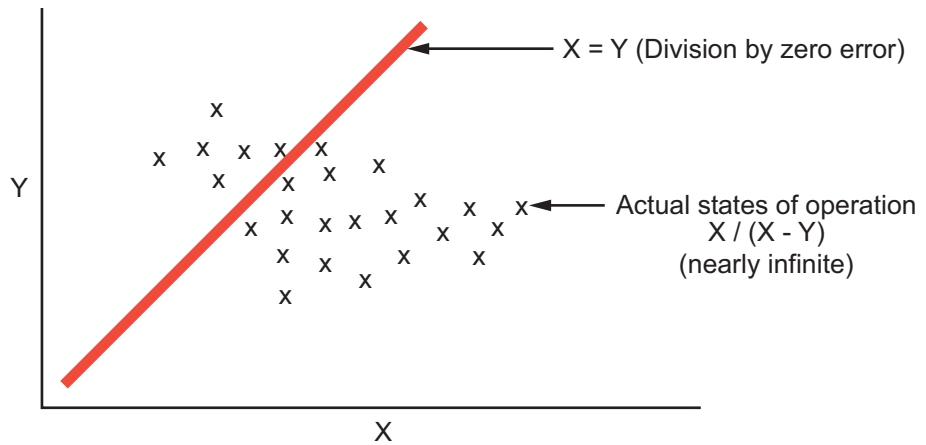
Because it is not possible to test every value for each variable, the target computer and programming language provide limits on the possible values of the variables. Polyspace verification uses these limits to compute a *cloud of points* (upper-bounded convex polyhedron) that contains all possible states for the variables.
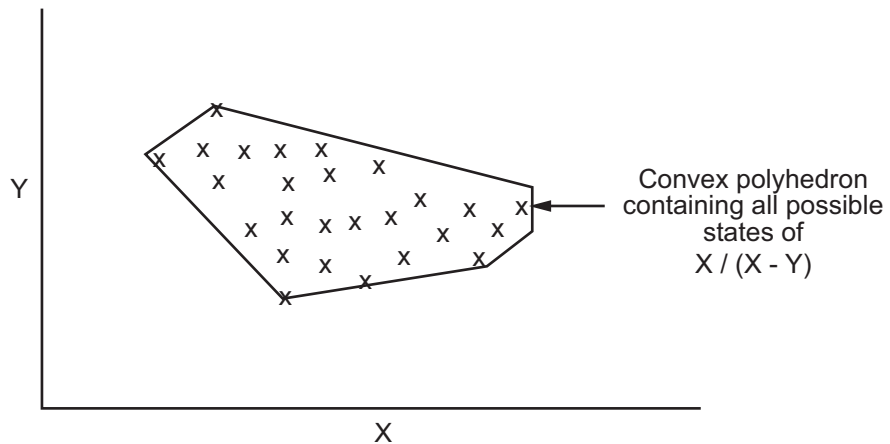
Polyspace verification then compares the data set represented by this polyhedron to the error zone. If the two data sets intersect, the check is orange.
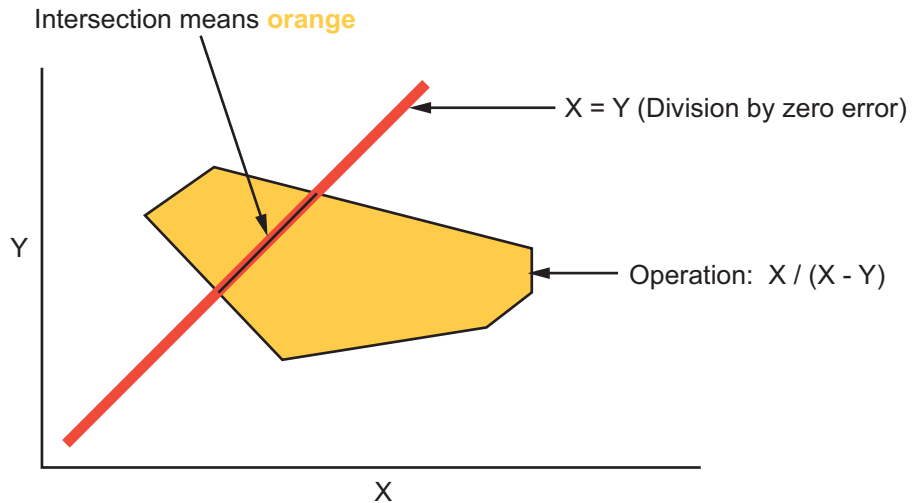
Graphical Representation of an Orange Check

A true orange check represents a situation where some paths fail while others succeed. However, because the data set in the verification is an approximation of actual values, an orange check may actually represent a check of any other color.
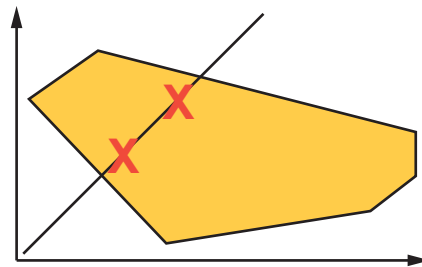
Red approximated by orange   Gray approximated by orange
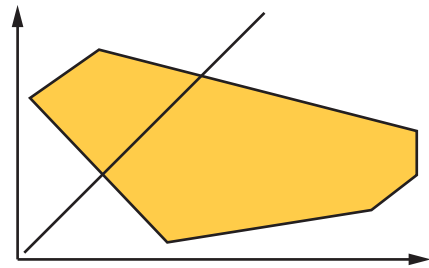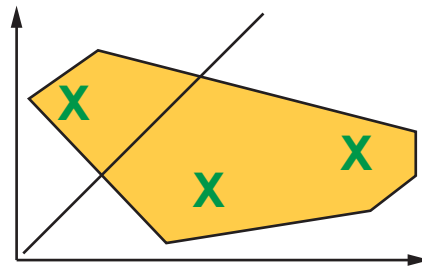
Green approximated by orange   Any other situation (true orange)
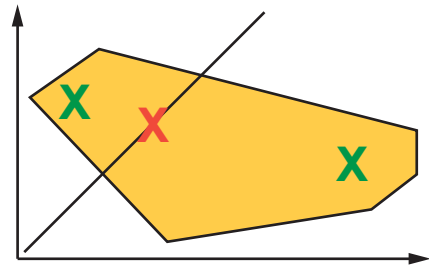
Polyspace reports an orange check any time the two data sets intersect, regardless of the actual values. Therefore, you may find orange checks that represent bugs, while other orange checks represent code that does not have run-time errors.

You can resolve some of these orange checks by increasing the precision of your verification, or by adding execution context, but often you must review the results to determine the source of an orange check.

Sources of Orange Checks

Orange checks can be separated into two categories:

In this section...
"Orange Checks from Code" on page 9-6
"Orange Checks from Verification Limitations" on page 9-7

Orange Checks from Code

Potential Bug

An orange check can reveal code which will fail under some circumstances. These types of orange checks often represent real bugs.

For example, consider a function Recursion():

- Recursion() takes a parameter, increments it, then divides by it.

- This sequence of actions loops through an indirect recursive call to Recursion_recurse().

If the initial value passed to Recursion() is negative, then the recursive loop will at some point attempt a division by zero. Therefore, the division operation causes an orange Division by Zero.

Data Set Issue

An orange check can result from a theoretical set of data that cannot actually occur.

Polyspace verification uses an *upper approximation* of the data set, meaning that it considers all combinations of input data rather than any particular combination. Therefore, an orange check may result from a combination of input values that is not possible at execution time.

For example, consider three variables *X*, *Y*, and *Z*:

- Each of these variables is defined as being between 1 and 1,000.

- The code computes `X*Y*Z` on a 16-bit data type.

- The result can potentially overflow, so it causes an orange OVFL.

When developing the code, you might know that the three variables cannot all take the value 1,000 at the same time, but this information is not available to the verification. Therefore, the multiplication is orange.

When an orange check is caused by a data set issue, you can usually identify the cause quickly. After identifying a data set issue, you might want to comment the code to flag the warning, or modify the code to take the constraints into account.

Orange Checks from Verification Limitations

Inconclusive Verification

An orange check can be caused by situations in which the verification is unable to conclude whether a problem exists.

In some code, it is impossible to conclude whether an error exists without additional information.

For example, consider a variable X, and two concurrent tasks `T1` and `T2`.

- X is initialized to `0`.

- `T1` assigns the value `12` to X.

- `T2` divides a local variable by X.

- A division by zero error is possible because `T1` can be started before or after `T2`, so the division causes an orange ZDV.

Unless you define the call sequence, the verification cannot determine if an error will occur.

Most inconclusive orange checks take some time to investigate. An inconclusive orange check often results from complex code structure. Sometimes, such situations take an hour or more to understand. Depending

on the criticality of the function and the required speed of execution, you may might want to rewrite the code to remove risk of failure.

Basic Imprecision

An orange check can be caused by imprecise approximation of the data set used for verification.

For example, consider a variable *X*:

- Before the function call, *X* is defined as having the following values:
  -5, -3, 8, or any value in range [10...20].
  0 has been excluded from the set of possible values for *X*.

- However, due to optimization at low precision levels (-OO), the verification approximates *X* in the range [-5...20], instead of the previous set of values.

- Therefore, calling the function x = 1/x causes an orange ZDV.

Polyspace verification is unable to prove the absence of a run-time error in this case.

In cases of basic imprecision, you might be able to resolve orange checks by increasing the precision level. If increasing the precision level does not resolve the orange check, verification cannot help directly. You must review the code to determine the problem.

For more information, see "Verification Approximations".

Do I Have Too Many Orange Checks?

If the goal of code verification is to prove the absence of run-time errors, you might be concerned by the number of orange checks in your results.

However, the presence of multiple orange checks need not be a cause for concern. There is no ideal minimum number of orange checks for all applications. The minimum number that you want depends on several factors:

- **Development Stage** – When verifying the first version of a software component, focus exclusively on resolving red checks. As development progresses, start considering the orange checks more and more.

- **Application Requirements** – Sometimes, to write provable code, you can compromise with properties such as code size, speed, and portability. Depending on the requirements of your application, you might optimize one or more of these properties at the expense of more orange checks.

- **Quality Goals** – Using Polyspace software, you can meet your quality goals. Therefore, before you verify code, you must define quality goals for your application. These goals should be based on the criticality of the application, as well as time and cost constraints. Based on your quality goals, you can choose to retain a specific minimum number of orange checks in your application.

It is these factors that ultimately determine how many orange checks are acceptable in your results, and what you must do with the orange checks that remain.

Therefore, it is essential to understand how to manage the remaining orange checks. For more information, see "Manage Orange Checks" on page 9-10.

Manage Orange Checks

Polyspace verification by itself cannot produce quality code at the end of the development process. Verification helps you measure the quality of your code, identify issues, and achieve your quality goals. To do this, you must effectively integrate Polyspace verification into your development process.

To manage orange checks effectively, perform each of the following steps:

1 Define your quality goals.

2 Set Polyspace analysis options to match your quality goals.

3 Define a process to reduce orange checks. See "Overview: Reducing Orange Checks" on page 9-11.

4 Apply the process to work with remaining orange checks.

Overview: Reducing Orange Checks

Actions that reduce orange checks and improve the quality of your code:

- "Apply Coding Guidelines" on page 9-12.

Actions that reduce orange checks through increased verification precision:

- "Improve Verification Precision" on page 9-13
- "Stub Parts of the Code Manually" on page 9-15.
- "Specify Multitasking Behavior" on page 9-21.

Each of these actions have trade-offs, either in development time, verification time, or the risk of errors. Therefore, before taking any of these actions, it is important to define your quality goals.

Your quality goals determine how many orange checks are acceptable, what actions you should take to reduce orange checks, and what you should do with the remaining orange checks.

Apply Coding Guidelines

The number of orange checks per file depends on the coding style used in the project.

The following coding guidelines improve Polyspace precision and selectivity in Ada code verification:

- Use constrained types. Use subtype and not standard type.

- Do not use "use at" clause.

- Minimize the use of big and complex types (record of record, array of record, etc.).

- Minimize the use of volatile variables.

- Minimize the use of assembler code.

- Do not mix assembly code and Ada. Gather all assembly code in a procedure or function which can be automatically stubbed.

Improve Verification Precision

Improving the precision of a verification can reduce the number of orange checks in your results.

There are a number of Polyspace options that affect the precision of the verification. The compromise for this improved precision is increased verification time.

The following sections describe how to improve the precision of your verification:

In this section...
"Set the Analysis Precision Level" on page 9-13
"Set Software Safety Analysis Level" on page 9-13

Set the Analysis Precision Level

The precision level specifies the mathematical algorithm used to compute the cloud of points (polyhedron) containing all possible states for the variables. Changing the precision level does not affect the quality of your code. However, orange checks caused by low precision can become green when verified with higher precision. The default precision level is 2. To set the precision level:

1 In the Project Manager perspective, on the **Configuration** pane, select **Precision**.

2 From the **Precision Level** drop-down list, select 0, 1, 2, or 3.

For more information, see "Precision level".

Set Software Safety Analysis Level

The verification level specifies how many times the abstract interpretation algorithm passes through your code. Each pass results in a deeper level of propagation of calling and called context. The deeper the verification goes, the more precise it is. By default, verification proceeds to `Software Safety Analysis Level 4`. To set the verification level:

1 In the Project Manager perspective, on the **Configuration** pane, select **Precision**.

2 From the **Verification level** drop-down list, select the appropriate level.

For more information, see "Verification level".

Stub Parts of the Code Manually

Manually stubbing parts of your code can reduce the number of orange checks in your results. Manual stubbing does not improve the quality of your code, but only changes the results.

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

If a function is supposed to return an integer, the default automatic stubbing will stub it on the assumption that it can potentially take any value from the full type of an integer.

The following sections describe how to reduce orange checks using manual stubbing:

In this section...
"Manual vs. Automatic Stubbing" on page 9-15
"Emulating Function Behavior with Manual Stubs" on page 9-16
"Reducing Orange Checks with Empty Stubs" on page 9-17
"Applying Constraints to Variables Using Stubs" on page 9-19

Manual vs. Automatic Stubbing

There are two types of stubs in Polyspace verification:

- **Automatic stubs** – The software automatically creates stubs for unknown functions based on the function prototype (the function declaration). Automatic stubs do not provide insight into the behavior of the function, but are very conservative, ensuring that the function does not cause any run-time errors.

- **Manual stubs** – You create these stub functions to emulate the behavior of the missing functions, and manually include them in the verification with the rest of the source code. Manual stubs can better emulate missing functions, or they can be empty.

By default, Polyspace software automatically stubs functions. However, because automatic stubs are conservative, they can lead to more orange checks in your results.

Stubbing Example

The following example shows the effect of automatic stubbing.

```
procedure a_missing_function
(dest: in out integer,
src : in integer);
procedure test is
a: integer;
b: integer;
begin
a: = 1;
b: = 0;
a_missing_function(a,b);
b:= 1/a;
end;
```

Due to automatic stubbing, the verification assumes that *a* can be any integer, including 0. This assumption produces an orange check on the division.

If you provide an empty manual stub for the function, the division is green. This action reduces the number of orange checks in the result, but does not improve the quality of the code itself. The function could still potentially cause an error.

However, if you provide a detailed manual stub that emulates the behavior of the function, the division can be any color, including red.

Emulating Function Behavior with Manual Stubs

You can improve both the speed and selectivity of your verification by providing manual stubs that emulate the behavior of missing functions. The trade-off is time spent writing the stubs.

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

Example

This example shows a header for a missing function (which may occur when the verified code is an incomplete subset of a project).

```
procedure a_missing_function
  (dest: in out integer,
  src  : in integer);
```

Applying fine-level modeling of constraints in primitives and outside functions at the application periphery propagates more precision throughout the application, which results in a higher selectivity rate (more proven colors, i.e. more red+ green + gray). For this function, you could add a simple body:

```
procedure a_missing_function
  (dest: in out integer,
  src : in integer)
begin
  dest := src;
end;
```

In this case, instead of considering the full range for the `dest` parameter, Polyspace considers the relation between input parameter `src` and the output parameter, propagating more precision throughout the application.

Reducing Orange Checks with Empty Stubs

Providing empty manual stubs can reduce the number of orange checks in your results.

For example, consider the following code:

```
package automatic_vs_manual_stub is

    procedure write_or_not1(x : in out Integer);
    procedure write_or_not2(x : in out Integer);
    procedure green;
    procedure orange;

end;
```

```
package body automatic_vs_manual_stub is

    procedure write_or_not2(x : in out Integer) is
    begin
        null;
    end;


    procedure orange is
        x : Integer;
        y : Integer;
    begin
        x := 12;
        y := 1;
        write_or_not1(x);
        y := y/x;   -- Orange ZDV due to automatic stub
    end;

    procedure green is
        x : Integer;
        y : Integer;
    begin
        x := 12;
        y := 1;
        write_or_not2(x);
        y := y/x;    -- Green due to empty stub
    end;

end;
```

The code for the two functions is identical, but the automatic stub produces an orange check, while the empty stub produces a green.

While the empty stub reduces the number of orange checks in your results, you must take additional steps to ensure that the actual function does not result in a run-time error.

Applying Constraints to Variables Using Stubs

Another way to increase the selectivity is to indicate to the Polyspace software that some variables may lie within smaller functional ranges instead of the full range of the considered type.

This smaller function range primarily concerns two items from the language:

- Parameters passed to functions.
- Variables' content, mostly globals, which might change from one execution to another. Typically, these might include things like calibration data or mission specific data. These variables might be read directly within the code, or read through an API of functions.

Reduce the cloud of points

If a function is supposed to return an integer, the default automatic stubbing stubs it on the assumption that it can potentially take any value from the full type of an integer.

Polyspace models data ranges throughout the code it verifies. It produces more precise, informative results provided that the data it considers from the "outside world" is representative of the data that can be expected when the code is implemented. There is a certain number of mechanisms available to model such a data range within the code itself, and there are three possible approaches.

with volatile and assert	with assert and without volatile	without assert, without volatile, without "if"
| ```
function stub return INTEGER is
tmp: INTEGER;
random: INTEGER;
pragma volatile (random);
begin
tmp:= random;
pragma assert (tmp>=1);
pragma assert (tmp<=10);
``` | ```
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);
``` | ```
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
``` |

`return tmp;`
`end;`
	`return tmp;`
	`end;`
		`tmp:=random;`
		`end loop;`
		`return tmp;`
		`end;`

There is no particular advantage in using one approach instead of another (except, perhaps, that the assertions in the first two usually generate orange checks).

Specify Multitasking Behavior

The asynchronous characteristics of your application can have a direct impact on the number of orange checks. Properly describing characteristics such as implicit task declarations, mutual exclusion, and critical sections can reduce the number of orange checks in your results.

For example, consider a variable X, and two concurrent tasks T1 and T2.

- X is initialized to 0.
- T1 assigns the value T2 to X.
- T2 divides a local variable by X.
- A division by zero error is possible because T1 can be started before or after T2, so the division causes an orange Division by Zero error.

The verification cannot determine if an error will occur without knowing the call sequence. Modeling the task differently could turn this orange check green or red.

For more information, see "Preparing Multitasking Code" on page 5-14.

Overview: Reviewing Orange Checks

After you define a process that matches your quality goals, you end up with a certain number of orange checks for your quality model.

At this point, the goal is not to eliminate orange checks, but to work efficiently with them.

To work efficiently with orange checks:

- "Define Your Review Methodology" on page 9-23
- "Perform Selective Review" on page 9-24
- "Import and Export Review Comments" on page 8-81

Define Your Review Methodology

Before reviewing verification results, configure a methodology for your project. The methodology specifies both the type and number of orange checks you need to review. For more information, see "What Are Review Methodologies?" on page 8-49.

As part of the process for defining quality goals for your project:

- Polyspace Client for Ada provides four predefined review methodologies. Choose one of the predefined review methodologies to specify the number and type of orange checks to review. For more information, see "Review Checks Using Predefined Methodologies" on page 8-51.

- To control the number and type of orange checks to review, define a custom methodology. For more information, see "Review Checks Using Custom Methodologies" on page 8-56.

  After you define a methodology, each developer can use the methodology to review verification results. This approach ensures that all users apply the same standards when reviewing orange checks at each stage of the development cycle.

Perform Selective Review

This example shows how to perform a selective orange check review using a predefined methodology. The number and type of orange checks that you review is determined by your review methodology. As a project progresses, change your review methodology to progressively review greater number of orange checks. For example, you can choose the review methodologies, **First checks to review** and **Methodology for Ada > Light** in the early stages of development. Later, you can choose the methodology, **Methodology for Ada > Moderate**.

To perform a selective orange review using the methodology **Methodology for Ada > Light**:

1 In the Results Manager perspective, from the drop-down list above the **Results Summary** pane, select the methodology **Methodology for Ada > Light**.

2 Select the first check on the **Results Summary** pane.

The **Source** pane displays the source code for this check.

...	Check	File	Package	Function
🥄	Division by Zero	example.adb	RUNTIME_ERROR	PROCEDURE_ZDV()
🥄	Arithmetic exceptions	example.adb	RUNTIME_ERROR	SQUARE_ROOT()
🥄	Overflow	array.ada	PKDATA	ARRAY_OVERFLOW_INIT()
🥄	Non-terminating call	sensitivity.ada	SENSITIVITY	SORT_CALIBRATION()
🥄	Non-terminating call	example.adb	RUNTIME_ERROR	RECURSIVE_2()
🥄	Non-terminating call	example.adb	RUNTIME_ERROR	RECURSION_CALLER()
🥄	Non-terminating loop	example.adb	RUNTIME_ERROR	INFINITE_LOOP()
🥄	Non-terminating loop	tasks.adb	PKTASKING	TREGULATE()
✗	Unreachable code	sensitivity.ada	SENSITIVITY	SORT_CALIBRATION()
✗	Unreachable code	sensitivity.ada	SENSITIVITY	POLYNOMIA()
✗	Unreachable code	sensitivity.ada	SENSITIVITY	DEGREE_COMPUTATION()
✗	Unreachable code	example.adb	RUNTIME_ERROR	MYABS()
✗	Unreachable code	example.adb	RUNTIME_ERROR	UNREACHABLE_CODE()
✗	Unreachable code	tasks.adb	PKTASKING	COMPUTE_NEW_COORDONA
?	Non-initialized local variable	util.ada	PKUTIL	COMMAND_ORDERING()
?	Division by Zero	example.adb	RUNTIME_ERROR	RECURSION()
?	Division by Zero	sensitivity.ada	SENSITIVITY	DIVIDE()
?	Overflow	example.adb	RUNTIME_ERROR	UNREACHABLE_CODE()
?	Overflow	tasks.adb	PKTASKING	TREGULATE.ORDER()
?	Overflow	sensitivity.ada	SENSITIVITY	SORT_ALGORITHM()
?	Overflow	sensitivity.ada	SENSITIVITY	ASSIGN_ARRAY()
?	Overflow	util.ada	PKUTIL	INCREASE_POWERLEVEL()
?	Overflow	util.ada	PKUTIL	COMPUTE_INJECTION()
?	Overflow	util.ada	PKUTIL	COMPUTE_INJECTION()
?	Non-initialized variable	tasks.adb	PKTASKING	TREGULATE.ORDER()
?	Non-initialized variable	util.ada	PKUTIL	PARTIAL_INIT()
?	User assertion	tasks.adb	PKTASKING	TREGULATE()
?	Overflow	example.adb	RUNTIME_ERROR	CLOSE_TO_ZERO()
?	Overflow	example.adb	RUNTIME_ERROR	CLOSE_TO_ZERO()
?	Overflow	example.adb	RUNTIME_ERROR	CLOSE_TO_ZERO()
?	Overflow	example.adb	RUNTIME_ERROR	CLOSE_TO_ZERO()

3 Perform a quick code review on each orange check. Your goal is to quickly identify whether the orange check is a:

- Potential bug — code that will fail under some circumstances.

- Inconclusive check — check that requires additional information to resolve, such as the call sequence.

- Data set issue — check originating from a set of data that cannot actually occur.

Note If you cannot understand an orange check quickly, it can be caused by complex code structure or approximate data set used for verification. These checks can often take a substantial amount of time to understand.

---

4 If you cannot identify a cause for the check, proceed to the next check.

5 Once you understand the cause of an orange check, record your review on the **Check Review** pane.

   a Select a **Classification** to describe the severity of the issue:

- `High`

- `Medium`

- `Low`

- `Not a defect`

   b Select a **Status** to describe how you intend to address the issue:

- `Fix`

- `Improve`

- `Investigate`

- `Justify with annotations`

- `No Action Planned`

- `Other`

- `Restart with different options`

- `Undecided`

You can also define your own statuses, which then appear in the user-defined acronym menu.

   c In the text box, enter a comment for the reviewed check.

6 Click to navigate to the next check, and repeat steps 3, 4, 5 and 6.

7 Continue to click until you have reviewed all of the checks identified on the **Results Summary** pane.

8 Select **File > Save** to save your review comments.

Tip The goal of a selective orange review is to find the maximum number of bugs in a short period of time. Many orange checks take only a few seconds to understand. To maximize the number of defects you can identify, focus on those checks you can understand quickly. Leave the checks that take longer to understand for later analysis.

Perform Exhaustive Review

Most orange checks can be resolved using multiple selective reviews. For more information, see "Perform Selective Review" on page 9-24. However, for extremely critical applications, you might want to resolve all orange checks.

To display all orange checks, in the Results Manager perspective, select **All checks** from the drop-down list above the **Results summary** pane.

An exhaustive orange review is conducted later in the development process, during the unit testing or integration testing phase. The purpose of an exhaustive orange review is to analyze any orange checks that were not resolved during previous selective orange reviews, to identify potential defects in those orange checks.

Before performing an exhaustive orange review, you must balance the time and cost of performing an exhaustive orange review against the potential cost of leaving a defect in the code.

Exhaustive Orange Review Methodology

Performing an exhaustive orange review involves reviewing each orange check individually. As with selective orange review, your goal is to identify whether the orange check is a:

- Potential bug – code which will fail under some circumstances.

- Inconclusive check – a check that requires additional information to resolve, such as the call sequence.

- Data set issue – a theoretical set of data that cannot actually occur.

- Basic imprecision – checks caused by imprecise approximation of the data set used for verification.

Note For more information on each of these causes, see "Sources of Orange Checks" on page 9-6.

Although you must review each check individually, there are some general guidelines to follow.

- Start your review with the files that have the highest selectivity in your application.

  If the verification finds only one or two orange checks in a file or function, these checks are probably not caused by either inconclusive verification or basic imprecision. Therefore, it is more likely that these orange checks contain actual defects. These types of orange checks can also be resolved more quickly.

- Next, examine files that contain a large percentage of orange checks compared to the rest of the application. These files can highlight design issues.

  If the verification is unable to draw a conclusion, it often means the code is very complex, which can mean low robustness and quality.

- For all files that you review, first identify checks that you can quickly categorize (such as potential bugs and data set issues).

- Depending on the results of your review, update the code or insert comments to identify the source of the orange check.

Inconclusive Verification and Code Complexity

Sometimes an inconclusive check occurs because the code is too complicated. In these cases, most orange checks in a file are related, and careful analysis identifies a single cause — perhaps a function or a variable modified many times. These situations often focus on functions or variables that have caused problems earlier in the development cycle.

For example, consider a variable `Computed_Speed`.

- `Computed_Speed` is first copied into a signed integer (between -2^31 and 2^31-1).

- `Computed_Speed` is then copied into an unsigned integer (between 0 and 2^31-1).

- `Computed_Speed` is next copied into a signed integer again.

- Finally, `Computed_Speed` is added to another variable.

The verification can report orange `Overflow`-s.

This report does not indicate a real defect, but informs the development team that the variable `Computed_Speed` caused trouble during development and testing phases. The report also suggests that the code is poorly designed.

Resolving Orange Checks Caused by Imprecise Approximation

On rare occasions, a module may contain many orange checks caused by imprecise approximation of the data set used for verification. These checks are usually local to functions, so their impact on the project as a whole is limited.

In cases of imprecise approximation, you can resolve orange checks by increasing the precision level. If this does not resolve the orange check, however, verification cannot help directly.

In these cases, Polyspace software can assist you only through the call tree and dictionary. For more information, see "Dataflow Verification" on page 8-108. The code must be reviewed using alternate means. These alternate means may include:

- Additional unit tests
- Code review with the developer
- Checking an interpolation algorithm in a function
- Checking calibration data

For more information on basic imprecision, see "Sources of Orange Checks" on page 9-6.

Software Quality with Polyspace Metrics

- "Software Quality with Polyspace Metrics" on page 10-2
- "Setting Up Verification to Generate Metrics" on page 10-3
- "Accessing Polyspace Metrics" on page 10-10
- "What You Can Do with Polyspace Metrics" on page 10-17
- "Customizing Software Quality Objectives" on page 10-28
- "Tips for Administering Results Repository" on page 10-35

Software Quality with Polyspace Metrics

Polyspace Metrics is a Web-based tool for software development managers, quality assurance engineers, and software developers, to do the following in software projects:

- Evaluate software quality metrics
- Monitor the variation of code metrics and run-time checks through the lifecycle of a project
- View defect numbers, run-time reliability of the software, review progress, and the status of the code with respect to software quality objectives.

If you are a development manager or a quality assurance engineer, through a Web browser, you can:

- View software quality information about your project. See "View Polyspace Metrics Project Index" on page 10-10.
- Observe trends over time, by project or module. See "Review Overall Progress" on page 10-17.
- Compare metrics of one project version with those of another. See "Compare Project Versions" on page 10-22.

If you have the Polyspace product installed on your computer, you can drill down to run-time checks in the Polyspace verification environment. This feature allows you to review run-time checks and, if required, classify these checks as defects. In addition, if you think that run-time checks can be justified, you can mark them as justified and enter relevant comments. See "Review Run-Time Checks" on page 10-23.

If you are a software developer, Polyspace Metrics allows you to focus on the latest version of the project that you are working on. You can use the view filters and drill-down functionality to go to code defects, which you can then fix. See "Fix Defects" on page 10-26.

Polyspace calculates metrics that are used to determine whether your code fulfills predefined software quality objectives. You can redefine these software quality objectives. See "Customizing Software Quality Objectives" on page 10-28.

Setting Up Verification to Generate Metrics

You can run, either manually or automatically, verifications that generate metrics. In each case, the Polyspace product uses a metrics computation engine to evaluate metrics for your code, and stores these metrics in a results repository.

Before you run a verification manually, in the Project Manager perspective:

1 Select the **Configuration > Machine Configuration** pane:

2 Select the **Send to Polyspace Server** check box.

3 Select the **Add to results repository** check box.

To set up scheduled, automatic verification runs, see "Specifying Automatic Verification" on page 10-3.

The software saves generated metrics in the following XML file:

Results_Folder/Polyspace-Doc/Code_Metrics.xml

Specifying Automatic Verification

You can configure verifications to start automatically and periodically, for example, at a specific time every night. At the end of each verification, the software stores results in the repository and updates the project metrics. You can also configure the software to send you an email at the end of the verification. This email will contain:

- Links to results
- An attached log file if the verification produces compilation errors
- A summary of new findings, for example, new potential and actual run-time errors

To configure automatic verification, you must create an XML file Projects.psproj that has the following elements:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Polyspace Metrics Automatic Verification Project File -->
```

```
<Configuration>
  <Project>
    <Options>
    </Options>
    <LaunchingPeriod>
    </LaunchingPeriod>
    <Commands>
    </Commands>
    <Users>
      <User>
      </User>
    </Users>
  </Project>
  <SmtpConfiguration>
  </SmtpConfiguration>
</Configuration>
```

Configure the verification by providing data for the elements (and their attributes) within `Configuration`. See "Element and Attribute Data for Projects.psproj" on page 10-5.

After creating `Projects.psproj`, place the file in the following folder on the Polyspace Queue Manager server:

`/var/Polyspace/results-repository`

Note If the flag `process_automation` in your configuration file `polyspace.conf` is set to `yes`, then, when you start your Polyspace Queue Manager server, Polyspace generates two template files in the results repository folder:

- `ProcessAutomationWindowsTemplate.psproj` for Windows

- `ProcessAutomationLinuxTemplate.psproj` for Linux

Use the relevant template to create your `Projects.psproj` file.

For more information about the configuration file `polyspace.conf`, see "Manual Configuration of the Polyspace Server".

Element and Attribute Data for Projects.psproj

Project. Specify three attributes:

- `name` — Your project name.

- `language` — `ADA` or `ADA95`.

- `verificationKind` — Mode, which is either `INTEGRATION` or `UNIT-BY-UNIT`.

For example,

```
<Project name="Demo_Ada" language="Ada" verificationKind="INTEGRATION">
```

The Project element also contains the following elements:

- "Options" on page 10-5
- "LaunchingPeriod" on page 10-5
- "Commands" on page 10-6
- "Users" on page 10-7

Options. Specify a list of all Polyspace options required for your verification, with the exception of `unit-by-unit`, `results-dir`, `prog` and `verif-version`. If these four options are present, they are ignored.

The following is an example.

```
<Options>
  -O2
  -to pass2
  -target sparc
  -temporal-exclusions-file sources/temporal_exclusions.txt
  -entry-points tregulate,proc1,proc2,server1,server2
  -critical-section-begin Begin_CS:CS1
  -critical-section-end End_CS:CS1
</Options>
```

LaunchingPeriod. For the starting time of the verification, specify five attributes:

- `hour`. Any integer in the range 0–23.

- `minute`. Any integer in the range 0–59.

- `month`. Any integer in the range 1–12.

- `day`. Any integer in the range 1–31.

- `weekDay`. Any integer in the range 1–7, where 1 specifies Monday.

Use `*` to specify all values in range, for example, `month="*"` specifies a verification every month.

Use `-` to specify a range, for example, `weekDay="1-5"` specifies Monday to Friday.

You can also specify a list for each attribute. For example, `day="1,15"` specifies the first and the fifteenth day of the month.

Default: If you do not specify attribute data for `LaunchingPeriod`, then a verification is started each week day at midnight.

The following is an example.

```
<LaunchingPeriod hour="12" minute="20" month="*" weekDay="1-5">
```

Commands. You can provide a list of optional commands. There must be only one command per line, and these commands must be executable on the computer that starts the verification.

- `GetSource`. A command to retrieve source files from the configuration management system, or the file system of the user. Executed in a temporary folder on the client computer, which is also used to store results from the compilation phase of the verification. This temporary folder is removed after the verification is spooled to the Polyspace server.

  For example:

  ```
  <GetSource>
    cvs co  r 1.4.6.4 myProject
    mkdir sources
    cp  fvr myProject/*.adb sources
  </GetSource>
  ```

You can also use:

```
<GetSource>
  find /  /myProject  name  *.adb  | tee sources_list.txt
</GetSource>
```

and add `-sources-list-file` *sources_list*.txt to the `options` list.

- `GetVersion`. A command to retrieve the version identifier of your project. Polyspace uses the version identifier as a parameter for `-verif-version`.

  For example:

```
<GetVersion>
  cd / ../myProject ; cvs status Makefile 2>/dev/null | grep 'Sticky Tag:'
  | sed 's/Sticky Tag://'  | awk '{print $1"-"$3}'| sed 's/).*$//'
</GetVersion>
```

Users. A list of users, where each user is defined using the element "User" on page 10-7.

User. Define a user using three elements:

- `FirstName`. First name of user.
- `LastName`. Last name of user.
- `Mail`. Use the attributes `resultsMail` and `compilationFailureMail` to specify conditions for sending an email at the end of verification. Specify the email address in the element.

  - `resultsMail`. You can use any of the following values:

    · `ALWAYS`. Default. Email sent at the end of each automatic verification (even if the verification does not produce new run-time checks).

    · `NEW-CERTAIN-FINDINGS`. Email sent only if verification produces new red, gray, NTC, or NTL checks.

    · `NEW-POTENTIAL-FINDINGS`. Email sent only if verification produces new orange check.

    · `ALL-NEW-FINDINGS`. Email sent if verification produces a new run-time check.

- `compilationFailureMail`. Either Yes (default) or No. If Yes, email sent when automatic verification fails because of a compilation failure.

The following is an example of `Mail`.

```
<Mail resultsMail="NEW-POTENTIAL-FINDINGS"
compilationFailureMail="yes">
 user_id@yourcompany.com
</Mail>
```

SmtpConfiguration. This element is mandatory for sending email, and you must specify the following attributes:

- `server`. Your Simple Mail Transport Protocol (SMTP) server.

- `port`. SMTP server port. Optional, default is 25.

For example:

```
<SmtpConfiguration server="smtp.yourcompany.com" port="25">
```

Example of Projects.psproj
The following is an example of `Projects.psproj`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Polyspace Metrics Automatic Verification Project File -->
<Configuration>
<Project name="Demo_Ada" language="ADA" verificationKind="INTEGRATION">
  <Options>
    -O2
    -to pass2
    -target sparc
    -temporal-exclusions-file sources/temporal_exclusions.txt
    -entry-points tregulate,proc1,proc2,server1,server2
    -critical-section-begin Begin_CS:CS1
    -critical-section-end End_CS:CS1
  </Options>
  <LaunchingPeriod hour="12" minute="20" month="*" weekDay="1-5">
  </LaunchingPeriod>
  <Commands>
    <GetSource>
```

```
      /bin/cp -vr /yourcompany/home/auser/tempfolder/Demo_Ada_Studio/sources/ .
    </GetSource>
    <GetVersion>
    </GetVersion>
  </Commands>
  <Users>
    <User>
      <FirstName>Polyspace</FirstName>
      <LastName>User</LastName>
      <Mail resultsMail="ALWAYS"
            compilationFailureMail="yes">userid@yourcompany.com
      </Mail>
    </User>
  </Users>
</Project>
<SmtpConfiguration server="smtp.yourcompany.com" port="25">
</SmtpConfiguration>
</Configuration>
```

Accessing Polyspace Metrics

In this section...
"View Polyspace Metrics Project Index" on page 10-10
"Organize Polyspace Metrics Projects" on page 10-11
"Protect Access to Project Metrics" on page 10-13
"Monitor Verification Progress" on page 10-15
"Web Browser Support" on page 10-15

View Polyspace Metrics Project Index

1 In the address bar of your Web browser, enter the following URL:

protocol:// *ServerName*: *PortNumber*

- *protocol* is either http (default) or https.
- *ServerName* is the name or IP address of the server that is your Polyspace Queue Manager.
- *PortNumber* is the Web server port number (default 8080)

To use HTTPS, you must set up the configuration file and the **Metrics configuration** preferences. For more information, see "Configure Web Server for HTTPS".

2 Select the **Projects** tab.

You can save the project index page as a bookmark for future use. You can also save as bookmarks any Polyspace Metrics pages that you subsequently navigate to.

To refresh the page at any point, click .

At the top of each column, use the filters to shorten the list of displayed projects. For example:

- In the **Project** filter, if you enter demo_, the browser displays a list of projects with names that begin with demo_.

- From the drop-down list for the **Language** filter, if you select Ada, the browser displays only Ada projects.

If a new verification has been carried out for a project since your last visit to the project index page, then the icon appears before the name of the project.

If you place your cursor anywhere on a project row, in a box on the left of the window, you see the following project information:

- **Language** — For example, Ada, C, C++.

- **Mode** — Either Integration or Unit by Unit.

- **Last Run Name** — Identifier for last verification performed.

- **Number of Runs** — Number of verifications performed in project.

In a project row, click anywhere to go to the **Summary** view for that project.

Organize Polyspace Metrics Projects

The Polyspace Metrics project index allows you to display projects as categories, a useful feature when you have a large number of projects to manage. You can:

- Create multiple-level project categories.

- Move projects between categories by dragging and dropping projects.

- Rename and remove categories. When you remove a category, the software does not delete the projects within the category but moves the projects back to the parent or root level.

To create a root-level project category:

1 On the Polyspace Metrics project index, right-click a project.

2 From the context menu, select **Create Project Category**. The Add To Category dialog box opens.

3 In **Enter the name of the project category** field, enter the required name, for example, MyNewCategory. Then click **OK**.

4 To add projects to this new category, drag and drop the required projects into this category.

To create a subroot-level category:

1 Right-click a project category.

2 From the context menu, select **Create Project Category**. The Add To Category dialog box opens.

3 In **Enter the name of the project category** field, enter the required name, for example, SubCategory1. If you decide that you do not want a subroot category, but want a new root category instead, select the **Create a root project category** check box. Then click **OK**.

4 To add projects to this new category, drag and drop the required projects into this category.

To rename a project category:

1 Right-click the project category.

2 From the context menu, select **Rename Project Category**. The category name becomes editable.

3 Enter the new name for your category. Press **Return**.

4 A message dialog box opens requesting confirmation. Click **OK**. The software updates the category name.

To remove a project category:

1 Right-click the project category.

2 From the context menu, select **Delete Project Category**. If the project category is a:

- Root-level project category, the software moves all projects to the root level and removes the project category and all associated subroot categories.

- Subroot-level category, the software moves all projects within the subroot category to the parent level and removes the subroot category.

Note The software does not delete projects when removing project categories.

You can move projects back to the root level from project categories without removing the project categories:

1 From within project categories, select the projects that you want to move to the root level.

2 Right-click the selected projects. From the context menu, select **Move to Root**. The software moves the projects back to the root level.

Protect Access to Project Metrics

You can restrict access to the metrics for a project by specifying a password:

- When you run a verification with Polyspace Metrics enabled or upload results to Polyspace Metrics:

  1 The Authentication Required dialog box opens.

  2 In the **Project password** and **Confirm password** fields, enter your password.

  3 Click **OK**.

- After the creation of a project:

  1 From the Polyspace Metrics project index, right-click the project.

  2 From the context menu, select **Change/Set Password**. The Change Project Password dialog box opens.

  3 In the **New password** and **Confirm new password** fields, enter your password.

  4 Click **OK**. The software displays the password-restricted icon next to the project.

From the command line, you can use the -password option. For example:

```
polyspace-results-repository.exe -prog psdemo_model_link_sl -password my_passwd
```

Note The password for a Polyspace Metrics project is encrypted. The Web data transfer is not encrypted. The password feature minimizes unintentional data corruption, but it does not provide data security. However, data transfers between the Polyspace Client and Polyspace Server are always encrypted. To use a secure Web data transfer with HTTPS, see "Configure Web Server for HTTPS".

After you enter your password, the project pages are accessible for a session that lasts 30 minutes. Access is available for this period of time, even if you close your Web browser. If you return to the Polyspace Metrics project index, the session ends. If you click during a session, the project pages are accessible for another 30 minutes.

Monitor Verification Progress

In the **Summary > Verification Status** column, Polyspace Metrics provides status information for each verification in the project. The status can be `queued`, `running`, or `completed`.

If the verification mode is `Unit By Unit`, the software provides status information in each unit row. If the verification mode is `Integration`, the software provides status information in the parent row only.

If the verification status is `running` (and you have installed the Polyspace product on your computer), you can monitor progress of the verification with the Polyspace Queue Manager.

To open the Progress Monitor of the Polyspace Queue Manager:

1 In the **Summary > Verification Status** column, right-click the parent or unit cell with the status `running`.

Verification	Verification Status	Code Metrics		Run-Time Errors						Review Progress
		Files	Lines of Code	Confirmed Defects	Run-Time Selectivity	Green	Red	Orange	Gray	
1.0 (8)	queued (ID=39)									
1.0 (7)	queued (ID=38)									
1.0 (6)	running (ID=13)									
1.0 (5)	completed (PA	Follow Progress…			83.8%	176	3	40	19	0.0%
1.0 (4)	completed (PA	Rename Run			83.8%	176	3	40	19	0.0%
1.0 (3)	completed (PA	Delete Run from Repository			83.8%	176	3	40	19	0.0%

2 From the context menu, select **Follow Progress**.

The Progress Monitor opens in the Polyspace verification environment.

For more information, see "Monitoring Progress Using Queue Manager" on page 6-11.

Web Browser Support

Polyspace Metrics supports the following Web browsers:

• Internet Explorer® 7.0, or later

- Firefox® version 3.6, or later

- Google® Chrome version 12.0, or later

To use Polyspace Metrics, you must install on your computer Java, version 1.4 or later.

For the Firefox Web browser, you must manually install the required Java plug-in. For example, if your computer uses the Linux operating system:

1 Create a Firefox folder for plug-ins:

```
mkdir ~/.mozilla/plugins
```

2 Go to this folder:

```
cd ~/.mozilla/plugins
```

3 Create a symbolic link to the Java plug-in, which is available in the Java Runtime Environment folder of your Polyspace installation:

```
ln -s Polyspace_Install/jre/lib/amd64/libnpjp2.so
```

What You Can Do with Polyspace Metrics

In this section...
"Review Overall Progress" on page 10-17
"Displaying Metrics for Single Project Version" on page 10-21
"Creating File Module and Specifying Quality Level" on page 10-21
"Compare Project Versions" on page 10-22
"Review New Findings" on page 10-23
"Review Run-Time Checks" on page 10-23
"Fix Defects" on page 10-26
"Review Code Metrics" on page 10-27

Review Overall Progress

For a development manager or quality assurance engineer, the Polyspace Metrics **Summary** view provides useful high-level information, including quality trends, over the course of a project.

To obtain the **Summary** view for a project:

1 Open the Polyspace Metrics project index. See "View Polyspace Metrics Project Index" on page 10-10.

2 Click anywhere in the row that contains your project. You see the **Summary** view.

At the top of the **Summary** view, use the **From** and **To** filters to specify the project versions that you want to examine. By default, the **From** and **To** fields specify the earliest and latest project versions respectively.

In addition, by default, the **Quality Objectives** filter is OFF, and the **Display Mode** is Review/Justification Progress (%).

Below the filters, you see:

- Plots that reveal how the number of verified files, lines of code, defects, and run-time selectivity vary over the different versions of your project

- A table containing summary information about your project versions

  If you have projects with two or more file modules in the Polyspace verification environment, by default, Polyspace Metrics displays verification results using the same module structure. However, Polyspace Metrics also allows you to create or delete file modules. See "Creating File Module and Specifying Quality Level" on page 10-21.

With the default filter settings, you can monitor progress in terms of run-time checks that quality assurance engineers or developers have reviewed.

You can also monitor progress in terms of software quality objectives. You may, for example, want to find out whether the latest version fulfills quality objectives.

To display software quality information, from the **Quality Objectives** drop-down list, select ON .

Under **Software Quality Objectives**, look at **Review Progress** for the latest version (CC-R2011bMain-S10 (2)), which, in this example, indicates that the review of verification results is incomplete (3.4 % reviewed). You also see that the Overall Status is FAIL. This status indicates that, although the review is incomplete, the project code fails to meet software quality objectives for the quality level MW-QO-3. With this information, you may conclude that you cannot release version CC-R2011bMain-S10 (2) to your customers.

Verification	Verification Status	Code Metrics		Run-Time Errors		Software Quality Objectives			
		Files	Lines of Code	Confirmed Defects	Run-Time Reliability	Overall Status	Level	Review Progress	Justified Run-Time Errors
CC-R2011bMain-S10 (2)	failed (----)	289	150473	7	87.6%	FAIL ⚠	MW-QO-3	3.4% ⚠	3.1% ⚠

When Polyspace Metrics adds the results for a new project version to the repository, Polyspace Metrics also imports comments from the previous version. For this reason, you rarely see the review progress metric at 0% after verification of the first project version.

Note You may want to find out whether your code fulfills software quality objectives at another quality level, for example, MW-QO-1. **Under Software Quality Objectives**, in the **Level** cell, select MW-QO-1 from the drop-down list.

There are seven quality levels, which are based on predefined software quality objectives. You can customize these software quality objectives and modify the way quality is evaluated. See "Customizing Software Quality Objectives" on page 10-28.

To investigate further, under **Run-Time Errors**, in the **Run-Time Reliability** cell, you click the link 87.6%. This action takes you to the **Run-Time Checks** view, where you see an expanded view of check information for each file in the project.

Verification	Confirmed Defects	Run-Time Reliability	Green Code	Systematic Run-Time Errors (Red Checks)		Unreachable Branches (Gray Checks)		Other Run-Time Errors (Orange Checks)		Non-terminating Constructs		Software Quality Objectives		
			Checks	Justified	Checks	Justified	Checks	Justified	Checks	Justified	Checks	Quality Status	Level	Review Progress
⊟ CC-R2011bMain-S10 (2)	7	87.6%	73537	94.4% ⚠	72	0.0% ⚠	1272	0.0% ⚠	9126	0.0% ⚠	205	FAIL ⚠	MW-QO-3	3.4% ⚠
⊞ xref_tab		46.0%	523	100.0%	1	0.0%	5	100.0%	611			FAIL	MW-QO-3	16.7%
⊞ xref		60.3%	450	100.0%	1			100.0%	297			PASS	MW-QO-3	100.0%
⊞ widechar		85.4%	216					100.0%	37			PASS	MW-QO-3	100.0%
⊞ usage		75.0%	3					100.0%	1			PASS	MW-QO-3	100.0%
⊞ urealp		76.6%	669			0.0%	1	0.0%	203			FAIL	MW-QO-3	0.0%
⊞ uname		81.5%	285	100.0%	2			0.0%	65			FAIL	MW-QO-3	40.0%
⊞ uintp		83.3%	1520	100.0%	2	0.0%	18	0.0%	287			FAIL	MW-QO-3	7.7%
⊞ types		98.2%	55					100.0%	1			PASS	MW-QO-3	100.0%
⊞ treepr		89.9%	587	100.0%	3	0.0%	8	0.0%	58			FAIL	MW-QO-3	25.0%
⊞ tree_io		84.8%	189					100.0%	34			PASS	MW-QO-3	100.0%
⊞ tree_gen		100.0%	2									PASS	MW-QO-3	100.0%
⊞ tbuild		81.5%	154					0.0%	35			FAIL	MW-QO-3	0.0%

To view any of these checks in the Polyspace verification environment, in the relevant cell, click the numeric value for the check. The Polyspace verification environment opens with the Results Manager perspective displaying verification information for this check.

Note If you update any check information through the Polyspace verification environment (see "Review Run-Time Checks" on page 10-23), when you return to Polyspace Metrics, click **Refresh** to incorporate this updated information.

If you want to view check information with reference to check type, from the **Group by** drop-down list, select Run-Time Categories .

Verification	Confirmed Defects	Run-Time Reliability	Green Code	Systematic Run-Time Errors (Red Checks)		Unreachable Branches (Gray Checks)		Other Run-Time Errors (Orange Checks)		Non-terminating Constructs		Software Quality Objectives		
			Checks	Justified	Checks	Justified	Checks	Justified	Checks	Justified	Checks	Quality Status	Level	Review Progress
⊟ CC-R2011bMain-S10 (2)	7	87.6%	73537	94.4% ⚠	72	0.0% ⚠	1272	0.0% ⚠	9126	0.0% ⚠	205	FAIL ⚠	MW-QO-3	3.4% ⚠
⊞ ZDV (Scalar) - Division by Zero		96.7%	260	⚠				0.0% ⚠	9			FAIL ⚠		0.0% ⚠
⊞ ZDV (Float) - Division by Zero				⚠				⚠				⚠		⚠
⊞ UOVFL (Scalar)				⚠								⚠		⚠
⊞ UOVFL (Float)				⚠								⚠		⚠
⊞ UNR - Unreachable Code		0.0%				0.0% ⚠	1272					FAIL ⚠		0.0% ⚠
⊞ UNFL (Scalar)				⚠								⚠		⚠
⊞ UNFL (Float)				⚠								⚠		⚠
⊞ STD_LIB - Argument of standard				⚠				⚠				⚠		⚠
⊞ SHF - Shift amount is outside its				⚠				⚠				⚠		⚠
⊞ POW - Power must be positive				⚠								⚠		⚠
⊞ OVFL (Scalar) - Overflow	1	61.2%	3709	⚠					2348			⚠		⚠
⊞ OVFL (Float) - Overflow		50.0%	1	⚠					1			⚠		⚠

Displaying Metrics for Single Project Version

To display metrics for a single project version:

1 In the **From** field, from the drop-down list, select the required project version.

2 In the **To** field, from the drop-down list, select the same project version.

3 In **# items** field, enter the maximum number of files for which you want information displayed.

The software displays:

- Bar charts with file defect information, ordering the files according to the number of defects in each file
- A table with information about the selected project version

Creating File Module and Specifying Quality Level

You can group files into a module and specify a quality level for the module, which applies to all files within the module. By grouping your files in different modules, you can specify different quality levels for your files.

To create a module of files:

1 Select a tab, for example, **Summary**.

2 In the **Verification** column, expand the node corresponding to the verification that you are interested. You see the verified files.

3 Select the files that you want to place in a module.

4 Right-click the selected files, and, from the context menu, select **Add To Module**. The Add to Module dialog box opens.

5 In the text field, enter the name for your new module, for example, Example_module. Click **OK**. You see a new node.

To specify a quality level for the module:

1 Select the row containing the module.

2 Under **Software Quality Objectives**, click the **Level** cell.

3 From the drop-down list, select the quality level for your module.

To remove files from a module:

1 Expand the node corresponding to the module.

2 Select the files that you want to remove from the module.

3 Right-click your selection, and from the context menu, select **Remove From Module**. The software removes the files from the module. If you remove all files from the module, the software also removes the module from the tree.

Note You can drag and drop files into and out of folders. For example, you can select back_end and drag it to Example_module.

Compare Project Versions

You can compare metrics of two versions of a project.

1 In the **From** drop-down list, select one version of your project.

2 In the **To** drop-down list, select a newer version of your project.

3 Select the **Compare** check box.

In each view, for example, **Summary** and **Run-Time Checks**, you see metric differences and tooltip messages that indicate whether the newer version is an improvement over the older version.

Review New Findings

You can specify a project version and focus on the differences between the verification results of your specified version and the previous verification. For example, consider a project with versions `1.0`, `1.1`, `1.2`, `2.0`, and `2.1`.

1 In the **To** field, specify a version of your project, for example, `2.0`.

2 Select the **New Findings Only** check box. In the **From** field, you see `1.2` in dimmed lettering, which is the previous verification. The charts and tables now show the changes in results with respect to the previous verification.

To manage the content of the bar charts, in the **# items** field, enter the maximum number of files for which you want information displayed. The software displays file defect information, ordering the files according to the number of defects in each file.

Review Run-Time Checks

If you have installed Polyspace on your computer, you can use Polyspace Metrics to review and add information about run-time checks produced by a verification.

You may use the **Review Progress** metric in the **Summary** view to decide when your team of developers should start work on the next version of the software. For example, you may wait until the review is complete (**Review Progress** cell displays 100%), before informing your development team.

Consider an example, where you see the following in the **Summary** view.

Verification	Verification Status	Code Metrics		Run-Time Errors		Software Quality Objectives			
		Files	Lines of Code	Confirmed Defects	Run-Time Reliability	Overall Status	Level	Review Progress	Justified Run-Time Errors
CC-R2011bMain-S10 (2)	failed (----)	289	150473	7	87.6%	FAIL ⚠	MW-QO-3	3.4% ⚠	3.1% ⚠

Under **Run-Time Errors**, click any cell value. This action takes you to the **Run-Time Checks** view.

Verification	Confirmed Defects	Run-Time Reliability	Green Code	Systematic Run-Time Errors (Red Checks)		Unreachable Branches (Gray Checks)		Other Run-Time Errors (Orange Checks)		Non-terminating Constructs		Software Quality Objectives		
			Checks	Justified	Checks	Justified	Checks	Justified	Checks	Justified	Checks	Quality Status	Level	Review Progress
CC-R2011bMain-S10 (2)	7	87.6%	73537	94.4% ⚠	72	0.0% ⚠	1272	0.0% ⚠	9126	0.0% ⚠	205	FAIL ⚠	MW-QO-3	3.4% ⚠
⊞ xref_tab		46.0%	523	100.0%	1	0.0%	5	100.0%	611			FAIL	MW-QO-3	16.7%
⊞ xref		60.3%	450	100.0%	1			100.0%	297			PASS	MW-QO-3	100.0%
⊞ widechar		85.4%	216					100.0%	37			PASS	MW-QO-3	100.0%
⊞ usage		75.0%	3					100.0%	1			PASS	MW-QO-3	100.0%
⊞ urealp		76.6%	669			0.0%	1	0.0%	203			FAIL	MW-QO-3	0.0%
⊞ uname		81.5%	285	100.0%	2			0.0%	65			FAIL	MW-QO-3	40.0%
⊞ uintp		83.3%	1520	100.0%	2	0.0%	18	0.0%	287			FAIL	MW-QO-3	7.7%
⊞ types		98.2%	55					100.0%	1			PASS	MW-QO-3	100.0%
⊞ treepr		89.9%	587	100.0%	3	0.0%	8	0.0%	58			FAIL	MW-QO-3	25.0%
⊞ tree_io		84.8%	189					100.0%	34			PASS	MW-QO-3	100.0%
⊞ tree_gen		100.0%	2									PASS	MW-QO-3	100.0%
⊞ tbuild		81.5%	154					0.0%	35			FAIL	MW-QO-3	0.0%

The **Review Progress** column reveals the progress level for each file. Expand sem_ch12.

Verification	Confirmed Defects	Run-Time Reliability	Green Code	Systematic Run-Time Errors (Red Checks)		Unreachable Branches (Gray Checks)		Other Run-Time Errors (Orange Checks)		Non-terminating Constructs		Software Quality Objectives		
			Checks	Justified	Checks	Justified	Checks	Justified	Checks	Justified	Checks	Quality Status	Level	Review Progress
⊟ sem_ch12	1	94.0%	2843	80.0%	5	0.0%	107	0.0%	73		47	FAIL	MW-QO-3	4.4%
⊞ g-htable.adb		88.2%	15						2					
⊟ sem_ch12.adb	1	94.3%	2699		5		104		58		47			
ASRT - User Assertion		80.0%			4				1					
K_NTC - Known Non-Termination of C											47			
NIV - Non-Initialized Variable		94.4%	611						36					
NIVL - Non-Initialized Local Variable	1	99.7%	2065		1				6					
OVFL (Scalar) - Overflow		59.5%	22						15					
UNR - Unreachable Code		0.0%					104							
ZDV (Scalar) - Division by Zero		100.0%	1											

In the row containing the NIVL (Non-Initialized Local Variable) check, click the value in the red **Checks** cell. This action opens the Polyspace verification environment with the Results Manager perspective. You see the NIVL check on the **Results Summary** tab.

To view details in the **Check Details** pane, double-click the NIVL check.

On the **Check Review** tab, using the drop-down list for the **Classification** field, you can classify the check as a defect (High, Medium, or Low) or specify that the check is Not a defect.

Using the drop-down list for the **Status** field, you can assign a status for the check, for example, `Fix` or `Investigate`. When you assign a status, the software considers the check to be *reviewed*.

If you think that the presence of the check in your code can be justified, select the check box **Justified**. In the **Comment** field, enter remarks that justify this check.

Save the review. See "Saving Review Comments and Justifications" on page 10-25.

Note Classifying a run-time check as a defect or assigning a status for an unreviewed check in the Polyspace verification environment increases the corresponding metric values (**Confirmed Defects** and **Review Progress**) in the **Summary** and **Run-Time Checks** views of Polyspace Metrics.

Specifying Download Folder for Polyspace Metrics

When you click a coding rule violation or run-time check, Polyspace downloads result files from the Polyspace Metrics web interface to a local folder. You can specify this folder as follows:

1 Select **Options > Preferences > Server configuration**.

2 If you want to download result files to the folder from which the verification is launched, select the check box **Download results automatically**.

3 If this launch folder does not exist, specify another path in the **Folder** field.

If you do not specify a folder using step 2 or 3, when you click a violation or check, the software opens a file browser. Use this browser to specify the download location.

Saving Review Comments and Justifications

By default, when you save your project (**Ctrl+S**), the software saves your comments and justifications to a local folder. See "Specifying Download Folder for Polyspace Metrics" on page 10-25.

If you want to save your comments and justifications to a local folder *and* the Polyspace Metrics repository, on the Results Manager toolbar, click the button [icon].

This default behavior allows you to upload your review comments and justifications only when you are satisfied that your review is, for example, correct and complete.

If you want the software to save your comments and justifications to the local folder *and* the Polyspace Metrics repository whenever you save your project (**Ctrl+S**):

1 Select **Options > Preferences > Server configuration**.

2 Select the check box **Save justifications in the Polyspace Metrics repository**.

Note In Polyspace Metrics, click [Refresh icon] to view updated information.

Fix Defects

If you are a software developer, you can begin to fix defects in code when, for example:

- In the **Summary** view, **Review Progress** shows 100%

- Your quality assurance engineer informs you

You can use Polyspace Metrics to access defects that you must fix.

Within the **Summary** view, under **Run-Time Errors**, click any cell value. This action takes you to the **Run-Time Checks** view.

You want to fix defects that are classified as defects. In the **Confirmed Defects** column, click a non-zero cell value. Polyspace Code Prover™ opens with the checks visible in **Results Summary** tab.

Double-click the row containing a check. In the **Check Details** pane, you see information about this check. You can now go to the source code and fix the defect.

Review Code Metrics

Polyspace Metrics generates metrics about your Ada code. These metrics provide the number of:

- Files

- Lines of code

- Packages

- Packages that appear in `with` statements

- Subprograms that appear in `with` statements

- Protected shared variables

- Unprotected shared variables

To review code metrics for your project, in the **Summary** view, click any value in a **Code Metrics** cell. The **Code Metrics** view opens.

Verification	Project Metrics						
	Files	Lines of Code	Packages	Packages in With Statements	Subprograms in With Statements	Protected Shared Variables	Non-Protected Shared Variables
CC-R2011bMain-S10 (2)	289	150473					
CC-R2011bMain-S10 (1)	289	150456					
CC-R2011bMain-S9 (2)	289	150456					
CC-R2011bMain-S9 (1)	289	150433					

Customizing Software Quality Objectives

About Customizing Software Quality Objectives

When you run your first verification to produce metrics, Polyspace software uses *predefined* software quality objectives (SQO) to evaluate quality. In addition, when you use Polyspace Metrics for the first time, Polyspace creates the following XML file that contains definitions of these software quality objectives:

RemoteDataFolder/Custom-SQO-Definitions.xml

RemoteDataFolder is the folder where Polyspace stores data generated by remote verifications. See "Modify Polyspace Server Configuration" in the *Polyspace Installation Guide*.

If you want to customize SQOs and modify the way quality is evaluated, you must change Custom-SQO-Definitions.xml. This XML file has the following form:

```
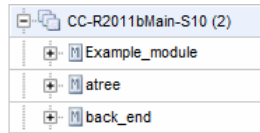<?xml version="1.0" encoding="utf-8"?>
<MetricsDefinitions>
```

```
    SQO Level 2
    SQO Level 3
    SQO Level 4
    SQO Level 5
    SQO Level 6
    SQO Exhaustive
    Run-Time Checks Set 1
    Run-Time Checks Set 2
    Run-Time Checks Set 3
    Status Acronym 1
    Status Acronym 2
</MetricsDefinitions>
```

The following topics provide information about MetricsDefinitions elements and how SQO levels are calculated. Use this information when you modify or create elements.

SQO Level 2

The default SQO Level 2 element is:

```
<SQO ID="SQO-2" ParentID="SQO-1">
  <Num_Unjustified_Red>O</Num_Unjustified_Red>
  <Num_Unjustified_NT_Constructs>O</Num_Unjustified_NT_Constructs>
</SQO>
```

To fulfill requirements of SQO Level 2, the code must meet the requirements of SQO Level 1 **and** the following:

- Number of unjustified red checks Num_Unjustified_Red must not be greater than the threshold (default is zero)

- Number of unjustified NTC and NTL checks Num_Unjustified_NT_Constructs must not be greater than the threshold (default is zero)

SQO Level 3

The default SQO Level 3 element is:

```
<SQO ID="SQO-3" ParentID="SQO-2">
  <Num_Unjustified_UNR>O</Num_Unjustified_UNR>
```

```
</SQO>
```

To fulfill requirements of SQO Level 3, the code must meet the requirements of SQO Level 2 **and** the number of unjustified UNR checks must not exceed the threshold (default is zero).

SQO Level 4

The default SQO Level 4 element is:

```
<SQO ID="SQO-4" ParentID="SQO-3">
  <Percentage_Proven_Or_Justified>
   Runtime_Checks_Set_1
  </Percentage_Proven_Or_Justified>
</SQO>
```

To fulfill requirements of SQO Level 4, the code must meet the requirements of SQO Level 3 **and** the following ratio as a percentage

```
(green checks + justified orange checks) / (green checks + all orange checks)
```

must not be less than the thresholds specified by "Run-Time Checks Set 1" on page 10-31.

SQO Level 5

The default SQO Level 5 element is:

```
<SQO ID="SQO-5" ParentID="SQO-4">
  <Percentage_Proven_Or_Justified>
   Runtime_Checks_Set_2
  </Percentage_Proven_Or_Justified>
</SQO>
```

To fulfill requirements of SQO Level 5, the code must meet the requirements of SQO Level 4 **and** the percentage of green and justified checks must not be less than the thresholds specified by "Run-Time Checks Set 2" on page 10-32.

SQO Level 6

The default SQO Level 6 element is:

```
<SQO ID="SQO-6" ParentID="SQO-5">
  <Percentage_Proven_Or_Justified>
   Runtime_Checks_Set_3
  </Percentage_Proven_Or_Justified>
</SQO>
```

To fulfill requirements of SQO Level 6, the code must meet the requirements of SQO Level 5 **and** the percentage of green and justified checks must not be less than the thresholds specified by "Run-Time Checks Set 3" on page 10-33.

SQO Exhaustive

The default Exhaustive element is:

```
<SQO ID="Exhaustive" ParentID="SQO-1">
  <Num_Unjustified_Red>O</Num_Unjustified_Red>
  <Num_Unjustified_NT_Constructs>O</Num_Unjustified_NT_Constructs>
  <Num_Unjustified_UNR>O</Num_Unjustified_UNR>
  <Percentage_Proven_Or_Justified>100</Percentage_Proven_Or_Justified>
</SQO>
```

Run-Time Checks Set 1

The Run-Time Checks Set 1 is composed of Check elements with data that specify thresholds. The Name and Type attribute in each Check element defines a run-time check, while the element data specifies a threshold in percentage. The default structure of Run-Time Checks Set 1 is:

```
<RuntimeChecksSet ID="Runtime_Checks_Set_1">
      <Check Name="OBAI">80</Check>
      <Check Name="ZDV" Type="Scalar">80</Check>
      <Check Name="ZDV" Type="Float">80</Check>
      <Check Name="NIVL">80</Check>
      <Check Name="NIV">60</Check>
      <Check Name="IRV">80</Check>
      <Check Name="FRIV">80</Check>
      <Check Name="FRV">80</Check>
      <Check Name="OVFL" Type="Scalar">60</Check>
      <Check Name="OVFL" Type="Float">60</Check>
      <Check Name="IDP">60</Check>
      <Check Name="NIP">60</Check>
```

```
                         <Check Name="POW">80</Check>
                         <Check Name="SHF">80</Check>
                         <Check Name="COR">60</Check>
                         <Check Name="NNR">50</Check>
                         <Check Name="EXCP">50</Check>
                         <Check Name="EXC">50</Check>
                         <Check Name="NNT">50</Check>
                         <Check Name="CPP">50</Check>
                         <Check Name="OOP">50</Check>
                         <Check Name="ASRT">60</Check>
</RuntimeChecksSet>
```

When you use Run-Time Checks Set 1 in evaluating code quality, the software calculates the following ratio as a percentage for each run-time check in the set:

```
(green checks + justified orange checks)/(green checks + all orange checks)
```

If the percentage values do not exceed the thresholds in the set, the code meets the quality level.

To modify the default set, you can change the check threshold values.

Run-Time Checks Set 2

This set is similar to "Run-Time Checks Set 1" on page 10-31, but has more stringent threshold values.

```
 <RuntimeChecksSet ID="Runtime_Checks_Set_2">
      <Check Name="OBAI">90</Check>
      <Check Name="ZDV" Type="Scalar">90</Check>
      <Check Name="ZDV" Type="Float">90</Check>
      <Check Name="NIVL">90</Check>
      <Check Name="NIV">70</Check>
      <Check Name="IRV">90</Check>
      <Check Name="FRIV">90</Check>
      <Check Name="FRV">90</Check>
      <Check Name="OVFL" Type="Scalar">80</Check>
      <Check Name="OVFL" Type="Float">80</Check>
      <Check Name="IDP">70</Check>
      <Check Name="NIP">70</Check>
```

```
        <Check Name="POW">90</Check>
        <Check Name="SHF">90</Check>
        <Check Name="COR">80</Check>
        <Check Name="NNR">70</Check>
        <Check Name="EXCP">70</Check>
        <Check Name="EXC">70</Check>
        <Check Name="NNT">70</Check>
        <Check Name="CPP">70</Check>
        <Check Name="OOP">70</Check>
        <Check Name="ASRT">80</Check>
</RuntimeChecksSet>
```

Run-Time Checks Set 3

This set is similar to "Run-Time Checks Set 1" on page 10-31, but has more stringent threshold values.

```
<RuntimeChecksSet ID="Runtime_Checks_Set_3">
        <Check Name="OBAI">100</Check>
        <Check Name="ZDV" Type="Scalar">100</Check>
        <Check Name="ZDV" Type="Float">100</Check>
        <Check Name="NIVL">100</Check>
        <Check Name="NIV">80</Check>
        <Check Name="IRV">100</Check>
        <Check Name="FRIV">100</Check>
        <Check Name="FRV">100</Check>
        <Check Name="OVFL" Type="Scalar">100</Check>
        <Check Name="OVFL" Type="Float">100</Check>
        <Check Name="IDP">80</Check>
        <Check Name="NIP">80</Check>
        <Check Name="POW">100</Check>
        <Check Name="SHF">100</Check>
        <Check Name="COR">100</Check>
        <Check Name="NNR">90</Check>
        <Check Name="EXCP">90</Check>
        <Check Name="EXC">90</Check>
        <Check Name="NNT">90</Check>
        <Check Name="CPP">90</Check>
        <Check Name="OOP">90</Check>
        <Check Name="ASRT">100</Check>
    </RuntimeChecksSet>
```

Status Acronyms

When you click a link, StatusAcronym elements are passed to the Polyspace verification environment. This feature allows you to define, through your Polyspace server, additional items for the drop-down list of the **Status** field in **Check Review**. See "Review Run-Time Checks" on page 10-23.

Polyspace Metrics provides the following default elements:

```
<StatusAcronym Justified="yes" Name="Justify with code/model annotations"/>
<StatusAcronym Justified="yes" Name="No action planned"/>
```

The **Name** attribute specifies the name that appears on the **Status** field drop-down list. If you specify the Justify attribute to be yes, then when you select the item, for example, No action planned, the software automatically selects the **Justified** check box. If you do not specify the Justify attribute, then the **Justified** check box is not selected automatically.

You can remove the default elements and create new StatusAcronym elements, which are available to all users of your Polyspace server.

Tips for Administering Results Repository

In this section...
"Through the Polyspace Metrics Web Interface" on page 10-35
"Through the Command Line" on page 10-36
"Backup of Results Repository" on page 10-37

Through the Polyspace Metrics Web Interface

You can rename or delete projects and verifications.

Project Renaming

To rename a project:

1 In your Polyspace Metrics project index, right-click the row with the project that you want to rename.

2 From the context menu, select **Rename Project**.

3 In the **Project** field, enter the new name.

Project Deletion

To delete a project:

1 In your Polyspace Metrics project index, right-click the row with the project that you want to delete.

2 From the context menu, select **Delete Project from Repository**.

Verification Renaming

To rename a verification:

1 Select the **Summary** view for your project.

2 In the **Verification** column, right-click the verification that you want to rename.

3 From the context menu, select **Rename Run**.

4 In the **Project** field, edit the text to rename the verification.

Verification Deletion

To delete a verification:

1 Select the **Summary** view for your project.

2 In the **Verification** column, right-click the verification that you want to delete.

3 From the context menu, select **Delete Run from Repository**.

Through the Command Line

You can run the following batch command with various options.

```
Polyspace_Install/polyspace/bin/polyspace-results-repository[.exe]
```

- To rename a project or version, use the following options:

  ```
  [-f] [-server hostname] -rename [-prog old_prog -new-prog new_prog]
  [-verif-version old_version -new-verif-version new_version]
  ```

  - *hostname* — Polyspace server. localhost if you run the command directly on the server. Can be omitted if, in the Polyspace Preferences dialog box, on the **Server configuration** tab, you have specified a server name. See "Modify Polyspace Client Configuration".

  - *old_prog* — Current project name

  - *new_prog* — New project name

  - *old_version* — Old version name

  - *new_version* — New version name

  - -f — Specifies that no confirmation is requested

- To delete a project or version, use the following options:

  ```
  [-f] [ server hostname] -delete -prog prog [-verif-version version]
  [-unit-by-unit|-integration]
  ```

- *hostname* — Polyspace server. `localhost` if you run the command directly on the server. Can be omitted if, in the Polyspace Preferences dialog box, on the **Server configuration** tab, you have specified a server name. See "Modify Polyspace Client Configuration".

- *prog* — Project name

- *version* — Version name. If omitted, all versions are deleted

- `unit-by-unit|-integration` — Delete only unit-by-unit or integration verifications

- `-f` — Specifies that no confirmation is requested

- To get *information* about other commands, for example, retrieve a list of projects or versions, and download and upload results, use the `-h` option.

Renaming and Deletion Examples

To change the name of the project `psdemo_model_link_sl` to `Track_Quality`:

```
polyspace-results-repository.exe -prog psdemo_model_link_sl
-new-prog Track_Quality -rename
```

To delete the fifth verification run with version `1.0` of the project `Track_Quality`:

```
polyspace-results-repository.exe -prog Track_Quality -verif-version 1.0
-run-number 5 -delete
```

To rename verification `1.2` as `1.0`:

```
polyspace-results-repository.exe -prog Track_Quality -verif-version 1.2
-new-verif-version 1.0 -rename
```

To rename the fourth verification run with version `1.0` as version `0.4`:

```
polyspace-results-repository.exe -prog Track_Quality -verif-version 1.0
-run-number 4 -new-verif-version 0.4 -rename
```

Backup of Results Repository

To preserve your Polyspace Metrics data, create a backup copy of the results repository *Polyspace_RLDatas*/results-repository — *Polyspace_RLDatas*

is the path to the folder where Polyspace stores data generated by remote verifications. See "Configure the Polyspace Server".

For example, on a Linux system, do the following:

1 $cd *Polyspace_RLDatas*

2 $zip -r *Path_to_backup_folder*/results-repository.zip
   results-repository

If you want to restore data from the backup copy:

1 $cd *Polyspace_RLDatas*

2 $unzip *Path_to_backup_folder*/results-repository.zip

11

Verifying Code in the Eclipse IDE

Verifying Code in the Eclipse IDE

You can use Polyspace software to verify Ada code that you develop within the Eclipse Integrated Development Environment (IDE).

The workflow is:

- Use the editor to create an Eclipse project and develop code within your project.
- Set up the Polyspace verification by configuring analysis options and settings.
- Start the verification and monitor the process.
- Review the verification results.

Before you verify code, install the Polyspace plug-in for Eclipse IDE. See "Polyspace Plug-In Requirements" and "Install Polyspace Plug-In for Eclipse IDE" on page 11-3.

Install Polyspace Plug-In for Eclipse IDE

You can install the Polyspace plug-in only if you have already set up the Eclipse Integrated Development Environment (IDE). For information about downloading and installing the Eclipse IDE, go to www.eclipse.org.

In addition to the Eclipse IDE, you must have:

- The GNATbench 2.5.1 plug-in. For more information, go to www.adacore.com.

- A GNAT compiler, a free compiler for Ada95 that is integrated into the GCC compiler system. For more information, go to www.gnu.org/software/gnat/.

Note On a Windows system, the GNATbench plug-in supports only the 32-bit version of the Eclipse IDE. Therefore, on a 64-bit Windows machine, you must install the 32-bit version of the Polyspace product. From a DOS command window, run the following command:

DVD\Installer32bits\Windows\Disk1\InstData\VM\Polyspace.exe

To install the Polyspace plug-in:

1 From the Eclipse editor, select **Help > Install New Software**. The Install wizard opens, displaying the Available Software page.

2 Click **Add**, which opens the Add Repository dialog box.

3 In the **Name** field, specify a name for your Polyspace site, for example, Polyspace_13a.

4 Click **Local**, which opens the Browse for Folder dialog box.

5 Navigate to the *Polyspace_Install*\polyspace\plugin\eclipse folder. Then click **OK**.

6 Click **OK**, which closes the Add Repository dialog box.

7 On the Available Software page, select Polyspace Plugin for Eclipse.

8 Click **Next**.

9 On the Install Details page, click **Next**.

10 On the Review Licenses page, review and accept the licence agreement. Then click **Finish**.

Once you install the Polyspace plug-in, in the Eclipse editor, you have access to:

• A **Polyspace** menu

• **Polyspace Log** and **Polyspace Setting** views

Creating an Eclipse Ada Project

In this section...

Creating a New Project

If your source files do not belong to an Eclipse project, then create a project using the Polyspace editor:

1 Select **File > New > Project** to open the New Project dialog box.

2 Under **Wizards**, select **Ada > Ada Managed Project**.

Note The software supports only Ada Managed Project and Ada Standard Project. Although Eclipse allows you to create a General Project for any type of source file, if you use General Project for Ada source files, you may encounter problems.

---

3 Click **Next** to start the New Ada Project wizard.

4 On the Create an Ada Project page, specify the name and location of the Ada project.

Either:

a Select the **Use default location** check box.

b In the **Project name** field, enter a name, for example, Demo_Ada.

Or:

a Clear the **Use default location** check box.

b Click **Browse**, and select a folder, for example, C:\Test\Source_ada.

c In the **Project name** field, enter a name, for example, Demo_Ada.

Click **Next**.

5 On the Select Ada Toolchain and Build Type page, specify the default build configuration for your project:

    a In the **Configuration Name** field, specify the library name, for example, lib.

    b In the **Toolchain Type** field, from the drop-down list, select GNAT for Windows (or GNAT Linux x86).

    c In the **Build Type** field, from the drop-down list, select either Executable or Static Library.

    d Under **Toolchain Path**, select **Use Installed Toolchain**.

6 Click **Manage** to open the Preferences dialog box.

7 Click **Add** to start the Install a new Ada toolchain wizard.

8 On the New Ada toolchain page, specify:

    a In the **Type** field, from the drop-down list, select GNAT for Windows (or GNAT Linux x86).

    b In the **Name** field, your toolchain, for example, GNAT GPL.

    c In the **Path** field, the path to your Ada installation folder, for example, C:\GNAT\2009.

    Click **Next**.

9 On the Select source directories page, select the folder that contains the Ada include files, and click **Finish**.

10 Click **OK** to close the Preferences dialog box.

11 On the Select Ada Toolchain and Build Type page, under **Initial Setting Values**, select **Use Default Settings**.

12 If you want to specify options for the Ada build on the Select Build Options page, click **Next**. Otherwise, click **Finish**.

You have created an Polyspace project.

Adding Source Files

To add Ada source files to a project:

1 In the **Ada Navigator** view, right-click the project, for example, `Demo_Ada`.

2 Select **Import** to start the Import wizard.

3 On the Select page, select **General > File System**, and click **Next**.

4 On the File system page:

    a In the **From directory** field, specify your source folder.

    b In the folder or file view, select the check boxes for the folder or files that you want to import.

    c In the **Into folder** field, specify the project, for example, `Demo_Ada`.

    d Under **Options**, select **Create selected folders** only.

    e Click **Finish**.

For information on developing code within Eclipse IDE, go to www.eclipse.org.

Setting Up Polyspace Verification with Eclipse Editor

In this section...
"Analysis Options" on page 11-9
"Other Settings" on page 11-9

Analysis Options

You can specify analysis options for your verification using **Configure Project** from the Polyspace drop-down list:

1 Using **Ada Navigator**, select the project or files that you want to verify.

2 Select **Polyspace > Configure Project**, which opens the Polyspace Launcher for Ada95 dialog box.

3 Select your analysis options.

4 Save your options, and close the dialog box.

For information on how to choose your options, see "Analysis Options".

Other Settings

In the Polyspace Settings view, specify:

- In the **Results folder** field, the location of your results folder, for example, `C:\eclipse_workspace_ada\Demo_Ada\results`.
- The required **Verification level**, for example, `Level4`.

In the **Polyspace Settings** view, you can also do one of the following:

- Select the **Generate a main** check box to automatically generate a main procedure.
- In the **Name of the main subprogram** field, specify the path to the main procedure that you manually created.

For more information on generating a main procedure either automatically or
manually, see "Verifying an Application Without a Main" on page 4-5.

Starting Verification from Eclipse Editor

To start a Polyspace verification from the Eclipse editor:

1 Select the files that you want to verify.

2 Either right-click and select **Start Polyspace Verification**, or select **Polyspace > Start Polyspace Verification**.

In the **Polyspace Log** view, you can follow the progress of the verification. If you see an error or warning, you can double-click it to go to the corresponding location in the source code.

For more information, see "Monitoring the Progress of the Verification" on page 6-25 and "Troubleshooting in Polyspace Products for Ada".

Reviewing Verification Results from Eclipse Editor

Use the Results Manager perspective of the Polyspace verification environment to examine results of the verification:

1 Select **Polyspace > Open Verification Results** to open the Results Manager perspective.

2 Select **File > Open**. The Please select a file dialog box opens.

3 Select the results file that you want to view, and click **Open**. You see the results in the Results Manager perspective.

For information on reviewing and understanding Polyspace verification results, see "Run-Time Error Review".

Using Polyspace Spooler

Use the Polyspace spooler to manage jobs running on remote servers. To open the spooler, select **Polyspace > Open Spooler** .

For more information, see "Managing Verification Jobs Using Polyspace Queue Manager" on page 6-9.

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 through the Project Manager perspective.

Category

One of four types of orange check: *potential bug, inconclusive check, data set issue* and *basic imprecision.*

Certain error

See "red check."

Check

A test performed by Polyspace during a verification and subsequently colored red, orange, green or gray in the Run-Time Checks perspective.

Code Verification

The Polyspace process through which code is tested to reveal definite and potential runtime errors and a set of results is generated for review.

Dead Code

Code which is inaccessible at execution time 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