PolySpace® Products for C++ 7
User's Guide

The MathWorks™

Accelerating the pace of engineering and science
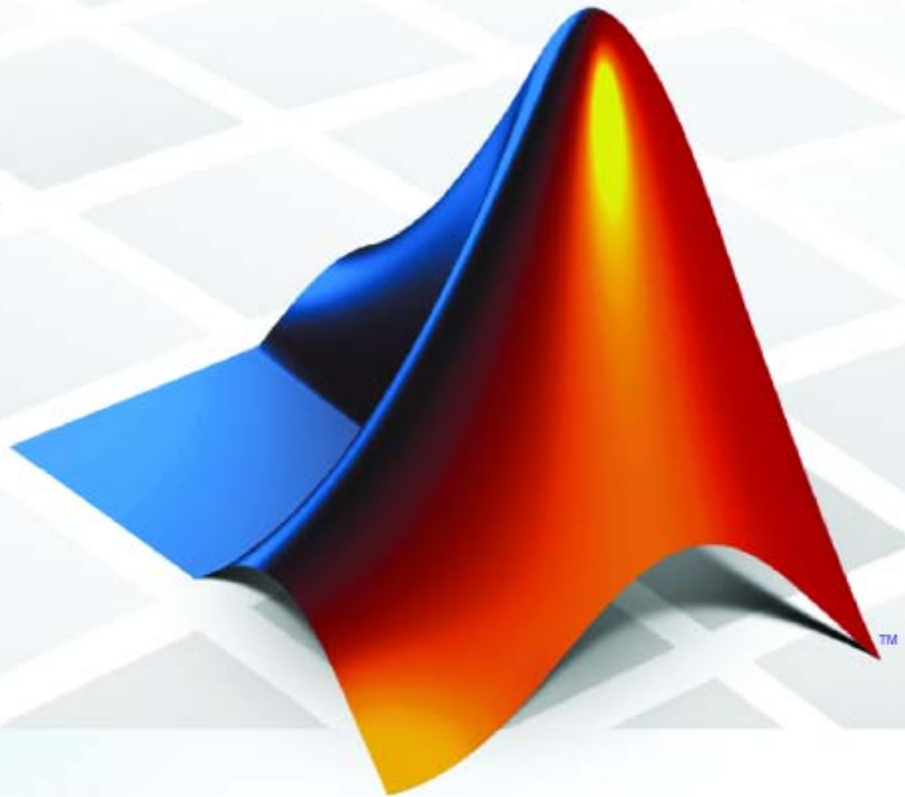
How to Contact The MathWorks

	www.mathworks.com	Web
	comp.soft-sys.matlab	Newsgroup
	www.mathworks.com/contact_TS.html	Technical Support
@	suggest@mathworks.com	Product enhancement suggestions
	bugs@mathworks.com	Bug reports
	doc@mathworks.com	Documentation error reports
	service@mathworks.com	Order status, license renewals, passcodes
	info@mathworks.com	Sales, pricing, and general information
☎	508-647-7000 (Phone)	
⌷	508-647-7001 (Fax)	
✉	The MathWorks, Inc.	
	3 Apple Hill Drive	
	Natick, MA 01760-2098	

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

PolySpace® Products for C++ User's Guide

© COPYRIGHT 1999–2010 by The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

Contents

Introduction to PolySpace Products

1

How to Use PolySpace Software

2

PolySpace Class Analyzer

3

Setting Up a Verification Project

4

Emulating Your Runtime Environment

5

Preparing Source Code for Verification

6

Running a Verification

7

Troubleshooting Verification Problems

8

Reviewing Verification Results

9

Managing Orange Checks

10

Day to Day Use

11

Checking Coding Rules

12

Introduction to PolySpace Products

Introduction to PolySpace Products

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

The Value of PolySpace Verification

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

PolySpace verification can help you to:

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

Ensure Software Reliability

PolySpace software ensures the reliability of your C++ applications by proving code correctness and identifying run-time errors. Using advanced verification techniques, PolySpace software performs an exhaustive verification of your source code.

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

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

- Might have an error

With this information, you can be confident that you know how much of your code is run-time error free, and you can improve the reliability of your code by fixing the errors.

You can also improve the quality of your code by using PolySpace verification software to check that your code complies with JSF C++ coding rules.

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

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

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

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

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

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

Improve the Development Process

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

PolySpace verification software supports code verification throughout the development process:

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

How PolySpace Verification Works

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

What is Static Verification

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

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

For example, consider the following code:

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

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

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

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

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

Exhaustiveness

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

Product Components

The PolySpace products for verifying C++ code are combined with the PolySpace products for verifying C++ code. These products are:

- "PolySpace® Client for C/C++ Software" on page 1-6
- "PolySpace® Server for C/C++ Software" on page 1-6

PolySpace Client for C/C++ Software

PolySpace Client software is the management and visualization tool of PolySpace products. You use it to submit jobs for execution by PolySpace Server, and to review verification results. The PolySpace Client software includes the Viewer, DRS, JSF C++ Checker, and Report Generator features.

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

PolySpace Server for C/C++ Software

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

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

Installing PolySpace Products

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

Related Products

- "PolySpace Products for Verifying C Code" on page 1-7
- "PolySpace Products for Verifying Ada Code" on page 1-7
- "PolySpace Products for Linking to Models" on page 1-7

PolySpace Products for Verifying C Code

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

```
http://www.mathworks.com/products/polyspaceclientc/
```

```
http://www.mathworks.com/products/polyspaceserverc/
```

PolySpace Products for Verifying Ada Code

For information about PolySpace products that verify Ada code, see the following:

```
http://www.mathworks.com/products/polyspaceclientada/
```

```
http://www.mathworks.com/products/polyspaceserverada/
```

PolySpace Products for Linking to Models

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

```
http://www.mathworks.com/products/polyspacemodelsl/
```

```
http://www.mathworks.com/products/polyspaceumlrh/
```

PolySpace Documentation

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

About this Guide

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

This guide is intended for both novice and experienced users.

Related Documentation

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

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

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

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

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

The MathWorks Online

For additional information and support, see:

`www.mathworks.com/products/polyspace`

How to Use PolySpace Software

PolySpace Verification and the Software Development Cycle

Software Quality and Productivity

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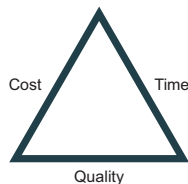
Changing the requirements for one of these variables always impacts the other two.

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

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

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

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

Best Practices for Verification Workflow

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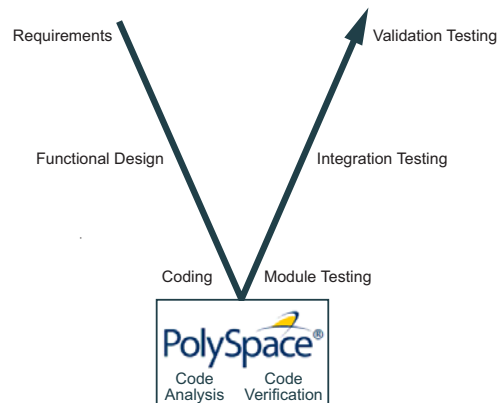
PolySpace Verification in the Development Cycle

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

Using verification early in the development cycle improves both quality and productivity, because it allows you to find and manage defects soon after the code is written. This saves time because each user is familiar with their own code, and can quickly determine why code cannot be proven safe. In addition, defects are cheaper to fix at this stage, since they can be addressed before the code is integrated into a larger system.

Implementing a Process for PolySpace Verification

In this section...
"Overview of the PolySpace Process" on page 2-4
"Defining Quality Objectives" on page 2-5
"Defining a Verification Process to Meet Your Objectives" on page 2-10
"Applying Your Verification Process to Assess Code Quality" on page 2-11
"Improving Your Verification Process" on page 2-11

Overview of the PolySpace Process

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

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

1 Define your quality objectives.

2 Define a process to match your quality objectives.

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

4 Improve the process.

Defining Quality Objectives

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

This process involves:

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

Choosing Robustness or Contextual Verification

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

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

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

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

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

- Assumes function inputs are full range

- Initializes global variables to full range

- Automatically stubs missing functions

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

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

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

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

- 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"".

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

Choosing Coding Rules

Coding rules are one of the most efficient means to improve both the quality of your code, and the quality of your verification results.

If your development team observes certain coding rules, the number of orange checks (unproven code) in your verification results will be reduced substantially. This means that there is less to review, and that the remaining checks are more likely to represent actual bugs. This can make the cost of bug detection much lower.

PolySpace software can check that your code complies with specified coding rules. Before starting code verification, you should consider implementing coding rules, and choose which rules to enforce.

For more information, see "MISRA® Checker".

Choosing Strict or Permissive Verification Objectives

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

Options that make verification more strict include:

- **-detect-unsigned-overflows** – Verification is more strict with overflowing computations on unsigned integers.

- **-wall** – Specifies that all C compliance warnings are written to the log file during compilation.

Options that make verification more permissive include:

- **-allow-negative-operand-in-shift** – Verification allows a shift operation on a negative number.

- **-ignore-constant-overflow** – Verification is permissive with overflowing computations on constants.

- **-allow-undef-variables** – Verification does not stop due to errors caused by undefined global variables.

For more information on these options, see "Option Descriptions" in the *PolySpace Products for C Reference*.

Defining Software Quality Levels

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

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

Software Quality Levels

Criteria	Software Quality Levels			
	QL1	QL2	QL3	QL4
Document static information	X	X	X	X
Enforce coding rules with direct impact on selectivity	X	X	X	X
Review all red checks	X	X	X	X
Review all gray checks	X	X	X	X
Review first criteria level for orange checks		X	X	X
Review second criteria level for orange checks			X	X
Enforce coding rules with indirect impact on selectivity			X	X
Perform dataflow analysis			X	X
Review third criteria level for orange checks				X

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

- **Static Information** – Includes information about the application architecture, the structure of each module, and all files. This information must be documented to ensure that your application is fully verified.

- **Coding rules** – PolySpace software can check that your code complies with specified coding rules. The section "Applying Coding Rules to Reduce Orange Checks"defines two sets of coding rules – a first set with direct impact on the selectivity of the verification, and a second set with indirect impact on selectivity.

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

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

- **Orange checks** – Indicate unproven code, meaning a run-time error may occur. PolySpace software allows you to define three criteria levels for reviewing orange checks in the PolySpace Viewer. For more information, see "Reviewing Results in Assistant Mode".

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

  - Application call tree

  - Read/write accesses to global variables

  - Shared variables and their associated concurrent access protection

Defining a Verification Process to Meet Your Objectives

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

These actions include:

- Setting standards for code development, such as coding rules.

- Setting PolySpace Analysis options to match your quality objectives. See "Creating a Project".

- Setting review criteria in the PolySpace Viewer to ensure results are reviewed consistently. See "Defining a Custom Methodology".

Applying Your Verification Process to Assess Code Quality

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

This process includes:

1 Launching PolySpace verification on each software component as it is written. See "Using PolySpace In One Click".

2 Reviewing verification results consistently. See "Reviewing Results in Assistant Mode".

3 Saving review comments for each component, so they are available for future review. See "Importing Review Comments from Previous Verifications".

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

Improving Your Verification Process

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

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

• Reassessing your quality objectives.

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

• Changing PolySpace analysis options to improve the precision of the verification.

• Changing PolySpace options to change how verification results are reported.

For more information, see "Managing Orange Checks".

Sample Workflows for PolySpace Verification

Overview of Verification Workflows

PolySpace verification supports two objectives at the same time:

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

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

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

Software Developers and Testers – Standard Development Process

User Description

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

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

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

Quality Objectives

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

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

Verification Workflow

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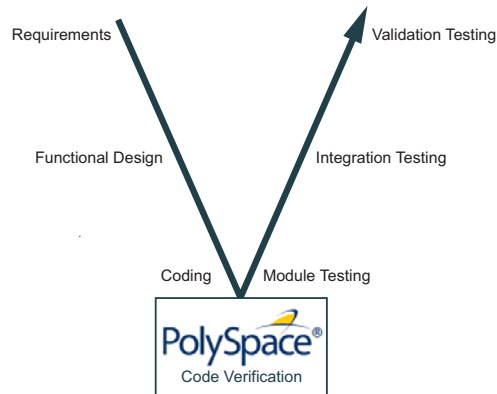
The verification workflow consists of the following steps:

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

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

---

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

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

4 The developer repeats steps 2 and 3 as needed, while completing the code.

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

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

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

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

Costs and Benefits

When using verification to detect bugs:

- **Red and gray checks** – Reviewing red and gray checks provides a quick method to identify real run-time errors in the code.
- **Orange checks** – Selective orange review provides a method to identify potential run-time errors as quickly as possible. The time required to find one bug varies from 5 minutes to 1 hour, and is typically around 30 minutes. This represents an average of two minutes per orange check review, and a total of 20 orange checks per package in Ada and 60 orange checks per file in C.

Disadvantages to this approach:

- **Number of orange checks** – If you do not use coding rules, your verification results will contain more orange checks.
- **Unreviewed orange checks** – Some bugs may remain in unchecked oranges.

Software Developers and Testers – Rigorous Development Process

User Description

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

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

Quality Objectives

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

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

Verification Workflow

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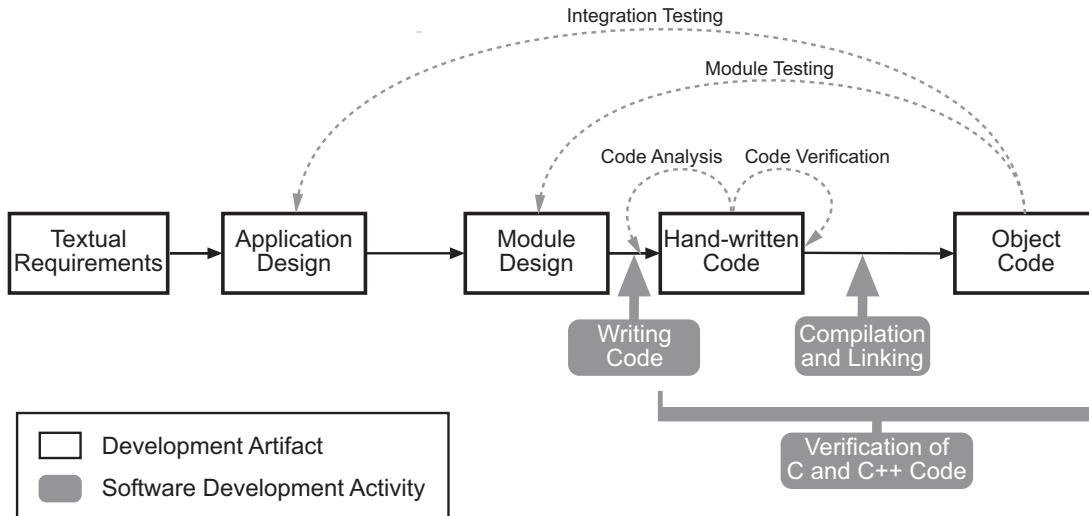
Workflow for Code Verification

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

The verification workflow consists of the following steps:

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

- Using Data Range Specifications (DRS) to define initialization ranges for input data. For example, if a variable "x" is read by functions in the file, and if x can be initialized to any value between 1 and 10, this information should be included in the DRS file.

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

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

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

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

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

5 The developer repeats steps 2 and 3 as needed, while completing the code.

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

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

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

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

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

Costs and Benefits

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

- Fewer orange checks in the verification results (improved selectivity). The number of orange checks is typically reduced to 3–5 per file, yielding an average of 1 bug. Often, several of the orange checks represent the same bug.

- Fewer gray checks in the verification results.

- Typically, each file requires two verifications before it can be checked-in to the configuration management system.

- The average verification time is about 15 minutes.

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

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

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

- Selective orange review takes about 6 hours to complete. This is long enough to review orange checks throughout the whole application. This represents a step towards an exhaustive orange check review. However, spending more time is unlikely to be efficient, and will not guarantee that no bugs remain.

- An exhaustive orange review would take between 4 and 6 days, assuming that 50,000 lines of code contains approximately 400–800 orange checks. Exhaustive orange review is typically recommended only for high-integrity code, where the consequences of a potential error justify the cost of the review.

Quality Engineers – Code Acceptance Criteria

User Description

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

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

Quality Objectives

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

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

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

Verification Workflow

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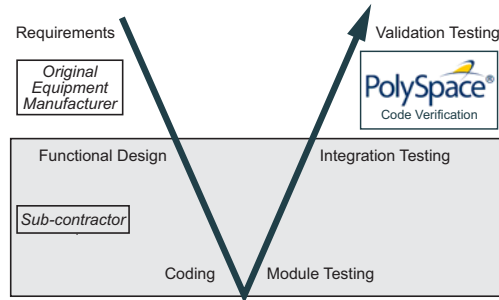
Note Verification is often performed multiple times, as multiple versions of the software are delivered.

The verification workflow consists of the following steps:

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

2 Development group writes code according to established standards.

3 Development group delivers software to the quality engineering group.

4 The project leader configures the PolySpace project to meet the defined quality objectives, as described in "Defining a Verification Process to Meet Your Objectives" on page 2-10.

5 Quality engineers perform verification on the code.

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

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

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

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

Costs and Benefits

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

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

- Developing code using strict coding and data rules.

- Providing accurate manual stubs for all unresolved function calls.

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

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

Quality Engineers – Certification/Qualification

User Description

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

These users must perform a set of activities to meet certification requirements.

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

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

Model-Based Design Users — Verifying Generated Code

User Description

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

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

Quality Objectives

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

PolySpace verification allows you to find run time errors:

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

Verification Workflow

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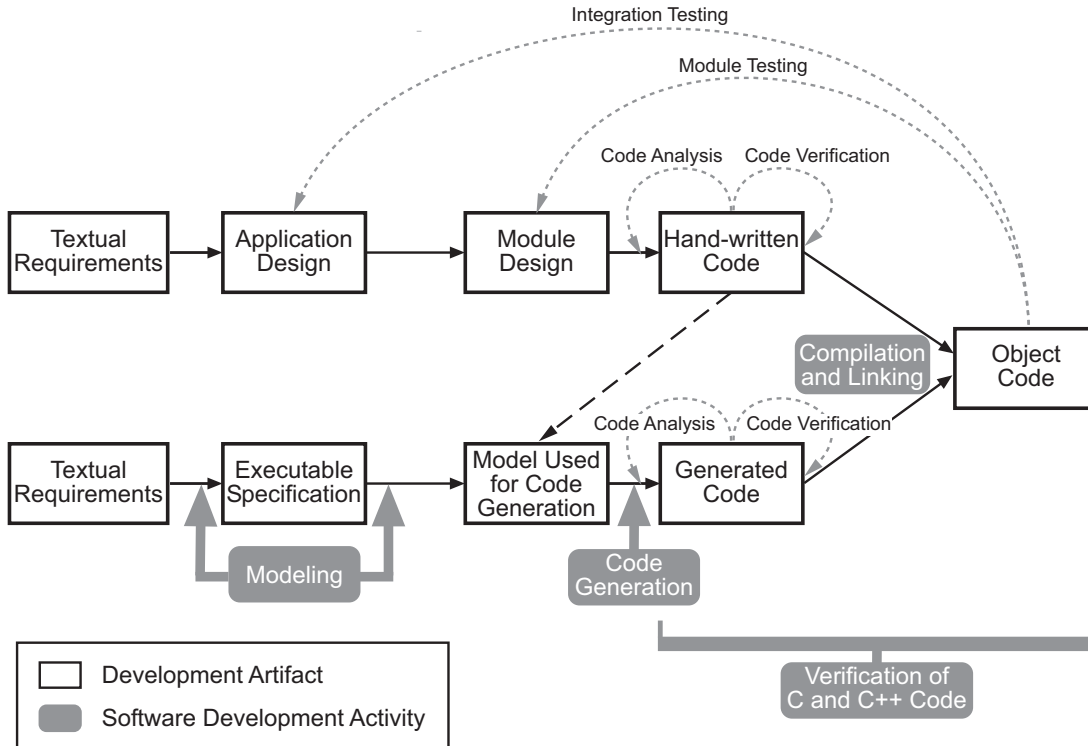
Workflow for Verification of Generated and Mixed Code

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

The verification workflow consists of the following steps:

1 The project leader configures a PolySpace project to meet defined quality objectives.

2 Developers write hand-coded sections of the application.

3 Developers or testers perform **PolySpace verification** on any hand-coded sections within the generated code, and review verification results according to the established quality objectives.

4 Developers create Simulink® model based on requirements.

5 Developers validate model to ensure it is logically correct (using tools such as Simulink Model Advisor, and the Simulink® Verification and Validation™ and Simulink® Design Verifier™ products).

6 Developers generate code from the model.

7 Developers or testers perform **PolySpace verification** on the entire software component, including both hand-written and generated code.

8 Developers or testers review verification results according to the established quality objectives.

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

Costs and Benefits

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

Examples of Common Run-Time Errors

Type of Error	Design or Model Errors	Code Errors
Arithmetic errors	• Incorrect Scaling • Unknown calibrations • Untested data ranges	• Overflows/Underflows • Division by zero • Square root of negative numbers
Memory corruption	• Incorrect array specification in state machines • Incorrect legacy code (look-up tables)	• Out of bound array indexes • Pointer arithmetic
Data truncation	• Unexpected data flow	• Overflows/Underflows • Wrap-around
Logic errors	• Unreachable states • Incorrect Transitions	• Non initialized data • Dead code

Project Managers — Integrating PolySpace Verification with Configuration Management Tools

User Description

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

Quality Objectives

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

Verification Workflow

The verification workflow consists of the following steps:

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

2 Project leader configures a PolySpace project to meet quality objectives.

3 Developers or testers run verification at the following stages:

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

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

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

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

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

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

3

PolySpace Class Analyzer

Analyzing C++ Classes

In this section...
"Overview" on page 3-2
"Why Provide a Class Analyzer" on page 3-2

Overview

This chapter explains how to use PolySpace to verify C++ classes in order to identify, and possibly remove, most of the run-time errors present in a class.

Why Provide a Class Analyzer

One aim of object-oriented languages such as C++ is reusability. A class or a class family is reusable if it is free of bugs for all possible uses of the class. It can be considered free of bugs if run-time errors have been removed and functional tests are successful. The foremost objective when developing code in such a language is to identify and remove as many run-time errors as possible.

PolySpace class analyzer is a tool for removing run-time errors at compilation time. The software will simulate all the possible uses of a class by:

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

2 Calling all methods (public, static, and protected) on previous objects in every order.

3 Calling all methods of the class between time zero and infinity.

4 Calling every destructor on previous objects (if they exist).

How the Class Analyzer Works

In this section...
"Overview" on page 3-3
"Sources to be Verified" on page 3-4
"Architecture of the Generated main" on page 3-4
"Log File" on page 3-5
"Characteristics of a Class and Messages in the Log File" on page 3-6
"Behavior of Global variables and members" on page 3-6
"Methods and Class Specificities" on page 3-9

Overview

The PolySpace™ Class Analyzer verifies applications class by class, even if these classes are only partially developed.

The **benefits** of this process include error detection at a very early stage, even if the class is not fully developed, without any test cases to write. The process is very simple: provide the class name and the software will verify its robustness.

- PolySpace will generate a "pseudo" main.
- It will call each constructor of the class.
- It will then call each public function from the constructors.
- Each parameter will be initialized with full range (i.e., with a random value).
- External variables will also be assigned random values.

Note Only prototypes of objects (classes, methods, variables, etc.) are needed to verify a given class. All missing code will be automatically stubbed.

Sources to be Verified

The sources associated with the verification normally concern public and protected methods of the class. However, sources can also come from inherited classes (fathers) or be the sources of other classes that are used by the class under investigation (friend, etc.).

Architecture of the Generated main

PolySpace generates the call to each constructor and method of the class. Each method will be analyzed with all constructors. Each parameter is initialized to random. Note that even if you can get an idea of the architecture of the generated main in PolySpace Viewer, the main is not real. You cannot reuse or compile it.

Consider the example class MathUtils in training.cpp which is located in *<PolySpaceInstallDir>*\Examples\Demo_Cpp_Long\sources\training.cpp. This class contains one constructor, one destructor and seven public methods. The architecture of the generated main is as follows:

```
Generating call to constructor: MathUtils:: MathUtils ()
While (random) {
 If (random) Generating call to function: MathUtils::Pointer_Arithmetic()
 If (random) Generating call to function: MathUtils::Close_To_Zero()
 If (random) Generating call to function: MathUtils::MathUtils()
 If (random) Generating call to function: MathUtils::Recursion_2(int *)
 If (random) Generating call to function: MathUtils::Recursion(int *)
 If (random) Generating call to function: MathUtils::Non_Infinite_Loop()
 If (random) Generating call to function: MathUtils::Recursion_caller()
}
Generating call to destructor: MathUtils::~MathUtils()
```

Note An ASCII file representing the "pseudo" main can be seen in C:\PolySpace_Results\ALL\SRC_polyspace_main.cpp

If a class contains more than one constructor, they are called before the "while" statement in an "if then else" statement. This architecture ensures that the verification will evaluate each function method with every constructor.

Log File

During a class verification, the list of methods used for the main appears in
the log file during the normalization phase of the C++ verification.

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

```
*****************************************************************
***
*** Beginning C++ source normalization
***
*****************************************************************
Number of files : 1
Number of lines : 202
Number of lines with libraries : 7009
**** C++ source normalization 1 (Loading)
**** C++ source normalization 1 (Loading) took 20.8real, 7.9u + 11.4s
(1gc)
**** C++ source normalization 2 (P_INIT)
* Generating the Main ...
Generating call to function: MathUtils::Pointer_Arithmetic()
Generating call to function: MathUtils::Close_To_Zero()
Generating call to function: MathUtils::MathUtils()
Generating call to function: MathUtils::Recursion_2(int *)
Generating call to function: MathUtils::Recursion(int *)
Generating call to function: MathUtils::Non_Infinite_Loop()
Generating call to function: MathUtils::~MathUtils()
Generating call to function: MathUtils::Recursion_caller()
```

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

```
*** Beginning C++ source normalization

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

Characteristics of a Class and Messages in the Log File

The log file may contain some error messages concerning the class to be analyzed. These messages appear when characteristics of a class are not respected.

- It is not possible to analyze a class that does not exist in the given sources. The verification will halt with the following message:

```
---------------------------------------------------------
@User Program Error: Argument of option -class-analyzer
must be defined : <name>.
Please correct the program and restart the verifier.
---------------------------------------------------------
```

- It is not possible to analyze a class that only contains declarations without code. The verification will halt with the following message:

```
---------------------------------------------------------
@User Program Error: Argument of option -class-analyzer
must contain at least one function : <name>.
Please correct the program and restart the verifier.
---------------------------------------------------------
```

Behavior of Global variables and members

Global Variables

During a class verification, global variables are not considered to be following ANSI Standard anymore if they are defined but not initialized. Remember that ANSI Standard considers, by default, that global variables are initialized to zero.

In a class verification, global variables do not follow standard behaviors:

- Defined variables are initialized to random and then follow the data flow of the code to be analyzed.

- Initialized variables are used with the specified initialized values and then follow the data flow of the code to be analyzed.

- External variables are assigned definitions and initialized to random values.

An example below demonstrates the behaviors of two global variables:

```
1
2 extern int fround(float fx);
3
4 // global variables
5 int globvar1;
6 int globvar2 = 100;
7
8 class Location
9 {
10 private:
11  void calculate_new(void);
12  int x;
13
14 public:
15  // constructor 1
16  Location(int intx = 0) { x = intx; };
17  // constructor 2
18  Location(float fx) { x = fround(fx); };
19
20  void setx(int intx) { x = intx; calculate_new(); };
21  void fsetx(float fx) {
22   int tx = fround(fx);
23   if (tx / globvar1 != 0) // ZDV check is orange
24   {
25    tx = tx / globvar2; // ZDV check is green
26    setx(tx);
27   }
28  };
29 };
```

In the above example, globvar1 is defined but not initialized (see line 5), so the check ZDV is orange at line 23. In the same example, globvar2 is initialized to 100 (see line 6), so the ZDV check is green at line 25.

Data Members of Other Classes

During the verification of a specific class, variable members of other classes, even members of parent classes, are considered to be initialized. They exhibit the following behaviors:

1 They may not be considered to be initialized if the constructor of the class is not defined. They are assigned to full range, and then they follow the data flow of the code to be analyzed.

2 They are considered to be initialized to the value defined in the constructor if the constructor of the class is defined in the class and is provided for the verification. If the `-class-only` option is applied, the software behaves as though the definition of the constructor is missing (see item 1 above).

3 They may be checked as run-time errors if and only if the constructor is defined but does not initialize the member under consideration.

The example below displays the results of a verification of the class `MyClass`. It demonstrates the behavior of a variable member of the class `OtherClass` that was provided without the definition of its constructor. The variable member of `OtherClass` is initialized to random; the check is orange at line 7 and there are possible overflows at line 17 because the range of the return value `wx` is "full range" in the type definition.

```
class OtherClass
{
protected:
 int x;
public:
 OtherClass (int intx);      // code is missing
 int getMember(void) {return x;};  // NIV is warning
};
class MyClass
{
 OtherClass m_loc;
public:
 MyClass(int intx) : m_loc(O) {};
 void show(void) {
  int wx, wl;
  wx = m_loc.getMember();
```

```
   wl = wx*wx + 2;   // Possible overflows because OtherClass
           // member is assigned to full range
  };
 };
```

Methods and Class Specificities

Template

A template class cannot be verified on its own. PolySpace will only consider a specific instance of a template to be a class that can be analyzed.

Consider `template<class T, class Z> class A { }.`

If we want to analyze template class `A` with two class parameters `T` and `Z`, we have to define a typedef to create an instance of the template with specified specializations for `T` and `Z`. In the example below, `T` represents an `int` and `Z` a `double`:

```
template class A<int, double>;   // Explicit specialisation
typedef class A<int, double> my_template;
```

`my_template` is used as a parameter of the `-class-analyzer` option in order to analyze this instance of template `A`.

Abstract Classes

In the real world, an instance of an abstract class cannot be created, so it cannot be analyzed. However, it is easy to establish a verification by removing the pure declarations. For example, this can be accomplished via an abstract class definition change:

`void abstract_func () = 0; by void abstract_func ();`

If an abstract class is provided for verification, the software will make the change automatically and the virtual pure function (`abstract_func` in the example above) will then be ignored during the verification of the abstract class.

This means that no call will be made from the generated main, so the function is completely ignored. Moreover, if the function is called by another one, the pure virtual function will be stubbed and an orange check will be placed on the call with the message "call of virtual function [f] may be pure."

Static Classes

If a class defines a static methods, it is called in the generated main as a classical one.

Inherited Classes

When a function is not defined in a derived class, even if it is visible because it is inherited from a father's class, it is not called in the generated main. In the example below, the class `Point` is derived from the class `Location`:

```
class Location
{
protected:
 int x;
 int y;
 Location (int intx, int inty);
public:
 int getx(void) {return x;};
 int gety(void) {return y;};
};
class Point : public Location
{
protected:
 bool visible;
public :
 Point(int intx, int inty) : Location (intx, inty)
 {
 visible = false;
 };
 void show(void) { visible = true;};
 void hide(void) { visible = false;};
 bool isvisible(void) {return visible;};
};
```

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

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

Types of Classes

In this section...

Simple Class

Consider the following class:

Stack.h

```
#define MAXARRAY 100

class stack
{
  int array[MAXARRAY];
  long toparray;

public:
  int top (void);
  bool isempty (void);
  bool push (int newval);
  void pop (void);
  stack ();
};
```

stack.cpp

```
1 #include "stack.h"
2
3 stack::stack ()
4 {
```

```
5  toparray = -1;
6  for (int i = 0 ; i < MAXARRAY; i++)
7  array[i] = 0;
8  }
9
10 int stack::top (void)
11 {
12  int i = toparray;
13  return (array[i]);
14 }
15
16 bool stack::isempty (void)
17 {
18  if (toparray >= 0)
19   return false;
20  else
21   return true;
22 }
23
24 bool stack::push (int newvalue)
25 {
26  if (toparray < MAXARRAY)
27  {
28   array[++toparray] = newvalue;
29   return true;
30  }
31
32  return false;
33 }
34
35 void stack::pop (void)
36 {
37  if (toparray >= 0)
38   toparray--;
39 }
```

The class analyzer calls the constructor and then all methods in any order many times.

The verification of this class highlights two problems:

- The `stack::push` method may write after the last element of the array, resulting in the OBAI orange check at line 28.
- If called before `push`, the `stack::top` method will access element -1, resulting in the OBAI and NIV checks at line 13.

Fixing these problems will eliminate run-time errors in this class.

Simple Inheritance

Consider the following classes:
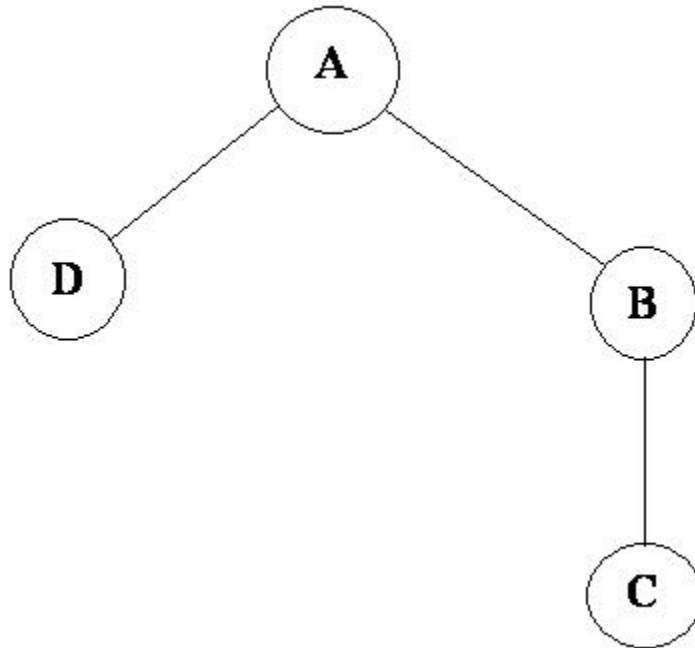
A is the base class of B and D.

B is the base class of C.

In a case such a this, PolySpace allows you to run the following verifications:

1 You can analyze class A just by providing its code to the software. This corresponds to the previous "Simple Class" section in this chapter.

2 You can analyze class B class by providing its code and the class A declaration. In this case, A code will be stubbed automatically by the software.

3 You can analyze class B class by providing B and A codes (declaration and definition). This is a "first level of integration" verification. The class analyzer will not call A methods. In this case, the objective is to find bugs only in the class B code.

4 You can analyze class C by providing the C code, the B class declaration and the A class declaration. In this case, A and B codes will be stubbed automatically.

5 You can analyze class C by providing the A, B and C codes for an integration verification. The class analyzer will call all the C methods but not inherited methods from B and A. The objective is to find bugs only in class C.

In these cases, there is no need to provide D class code for analyzing A, B and C classes as long as they do not use the class (e.g., member type) or need it (e.g., inherit).

Multiple Inheritance

Consider the following classes:
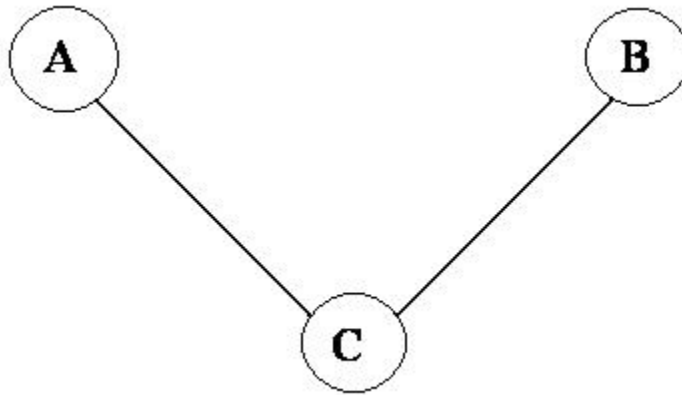
A and B are base classes of C.

In this case, PolySpace allows you to run the following verifications:

1 You can analyze classes A and B separately just by providing their codes to the software. This corresponds to the previous "Simple Class" section in this chapter.

2 You can analyze class C by providing its code with A and B declarations. A and B methods will be stubbed automatically.

3 You can analyze class C by providing A, B and C codes for an integration verification. The class analyzer will call all the C methods but not inherited methods from A and B. The objective is to find bugs only in class C.

Abstract Classes

Consider the following classes:
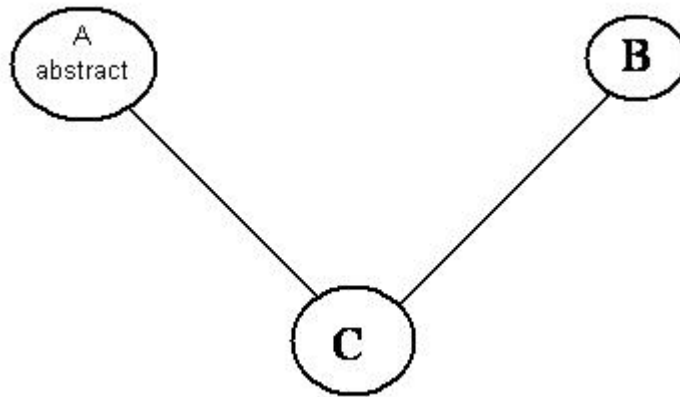
A is an abstract class

B is a simple class.

A and B are base classes of C.

C is not an abstract class.

As it is not possible to create an object of class A, this class cannot be analyzed separately from other classes. Therefore, you are not allowed to specify class A to the PolySpace class analyzer. Of course, class C can be analyzed in the same way as in the previous section "Multiple Inheritance."

Virtual Inheritance

Consider the following classes:
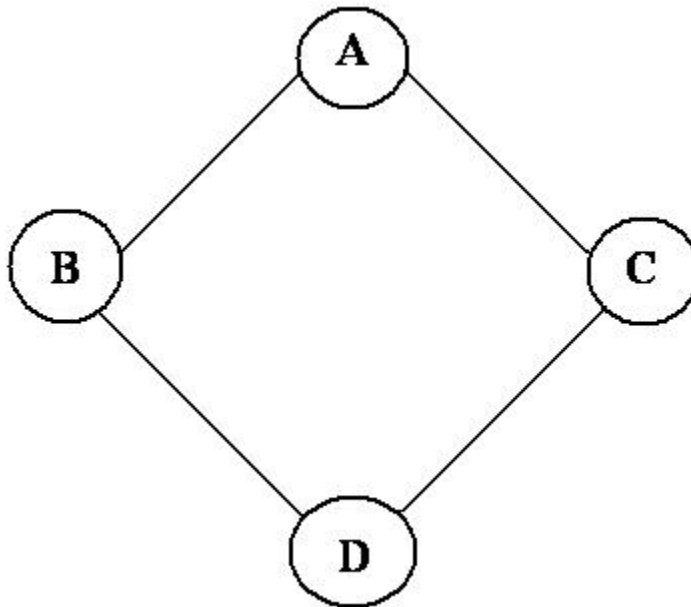
B and C classes virtually inherit the A class

B and C are base classes of D.

A, B, C and D can be analyzed in the same way as described in the previous section "Abstract Classes."

Virtual inheritance has no impact on the way of using the class analyzer.

Other Types of Classes

Template Class
A template class can not be analyzed directly. But a class instantiating a template can be analyzed by PolySpace.

> **Note** If only the template declaration is provided, missing functions'
> definitions will automatically be stubbed.

Example

```
template<class T > class A {
public:
 T i;
 T geti() {return i;}
 A() : i(1) {}
};
```

You have to define a `typedef` to create a specialization of the template:

```
template class A<int>;       // Explicit specialization
typedef class A<int> my_template;  // complete instance of the template
```

and use option `-class-analyzer my_template`.

The software will analyze a single instance of the template.

Class Integration

Consider a `C` class that inherits from `A` and `B` classes and has object members
of `AA` and `BB` classes.

A class integration verification consists of verifying class `C` and providing the
codes for `A`, `B`, `AA` and `BB`. If some definitions are missing, the software will
automatically stub them.

4

Setting Up a Verification Project

Creating a Project

What Is a Project?

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

A project includes:

- The location of source files and include folders
- The location of a folder for verification results
- Analysis options

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

A project file has one of the following file types:

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

Project Folders

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

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

polyspace_project/

- sources

- includes

- results

Opening PolySpace Launcher

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

To open the PolySpace Launcher:

1 Double-click the PolySpace Launcher icon.

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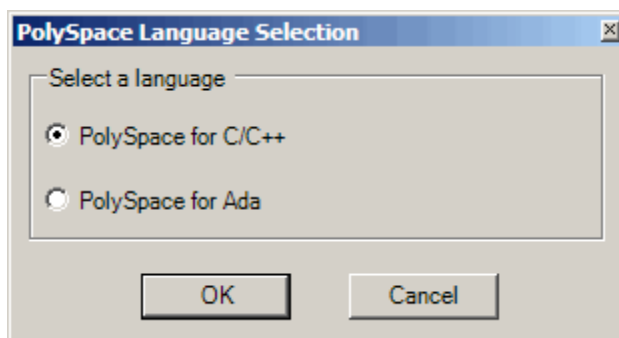
Select **PolySpace for C/C++**, then click **OK**.

The PolySpace Launcher window appears:

Specify
source files

Specify
include folders

Specify
analysis
options

Control
verification

Monitor
progress

View log
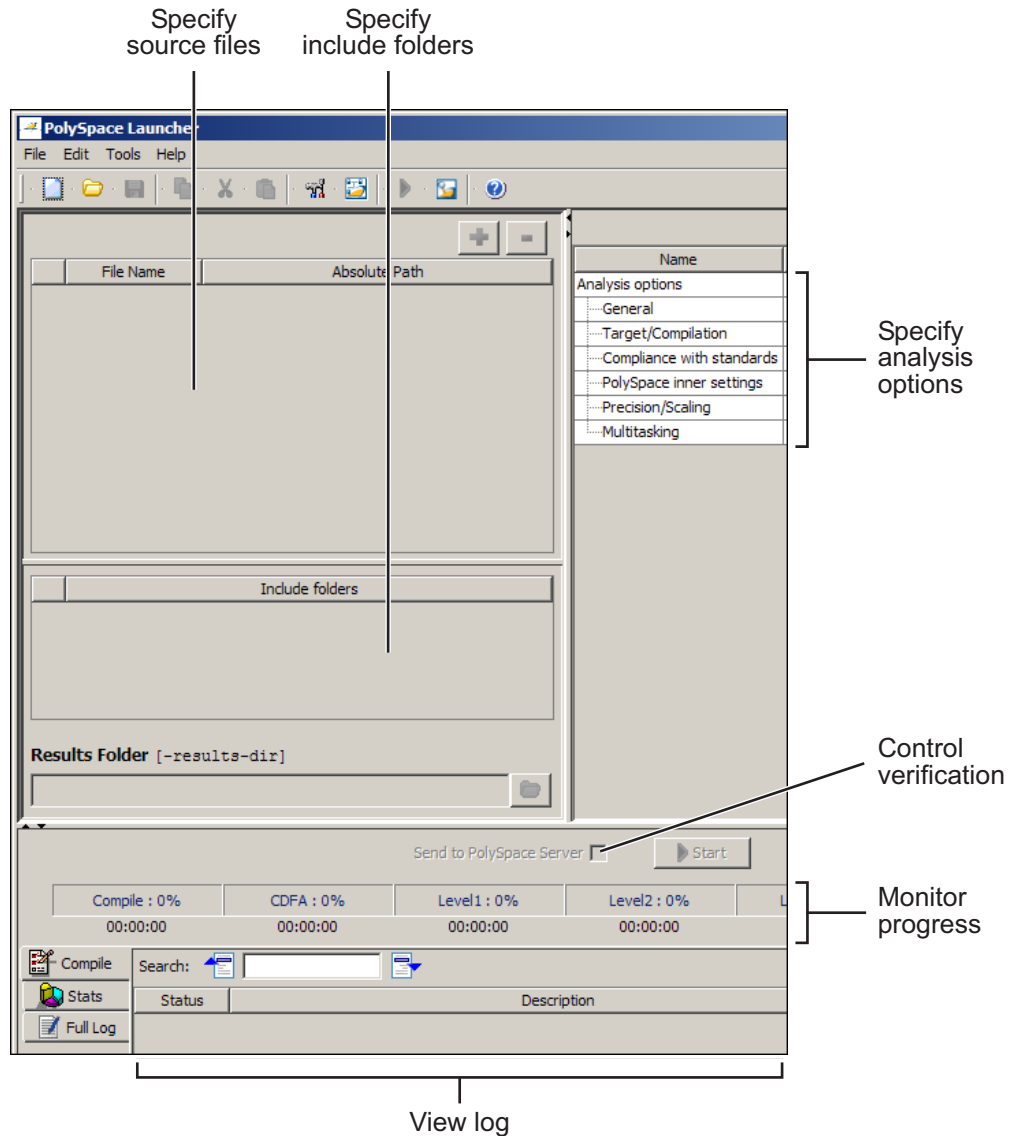
The Launcher window has three main sections.

Use this section...	For...
Upper-left	Specifying: • Source files • Include folders • Results folder
Upper-right	Specifying analysis options
Lower	Controlling and monitoring a verification

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

Specifying Default Folder

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

To change the default folder to the project folder:

1 Select **Edit > Preferences**.

The **Preferences** dialog box appears.

2 Select the **Default folder** tab.
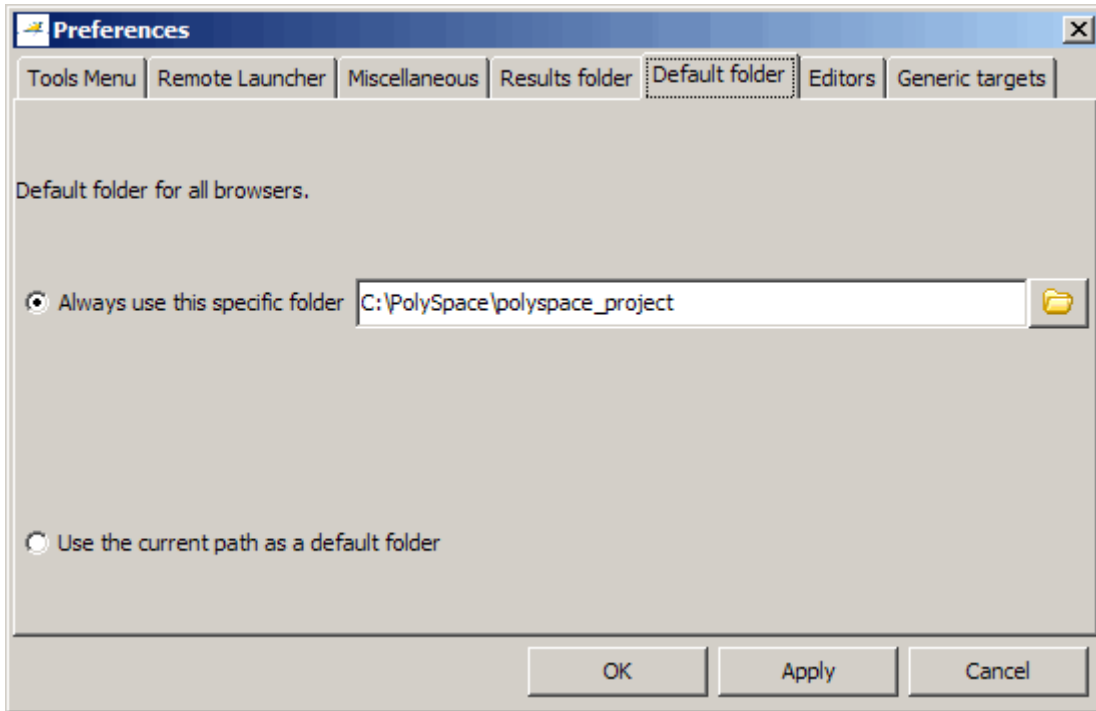
3 Select **Always use this specific folder** if it is not already selected.

4 Enter or navigate to the project folder you want to use.

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

Creating New Projects

To create a new project:

1 Select **File > New Project**.

The **Choose the language** dialog box appears:
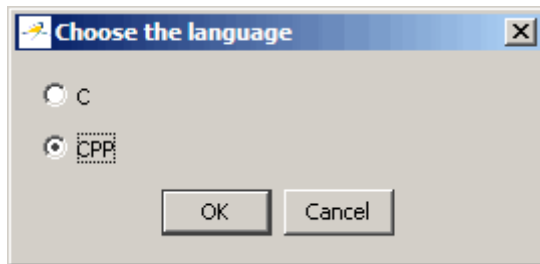
2 Select **CPP,** then click **OK**.

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

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

Opening Existing Projects

To open an existing project:

1 Select **File > Open Project**.

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

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

The selected project opens in the Launcher.
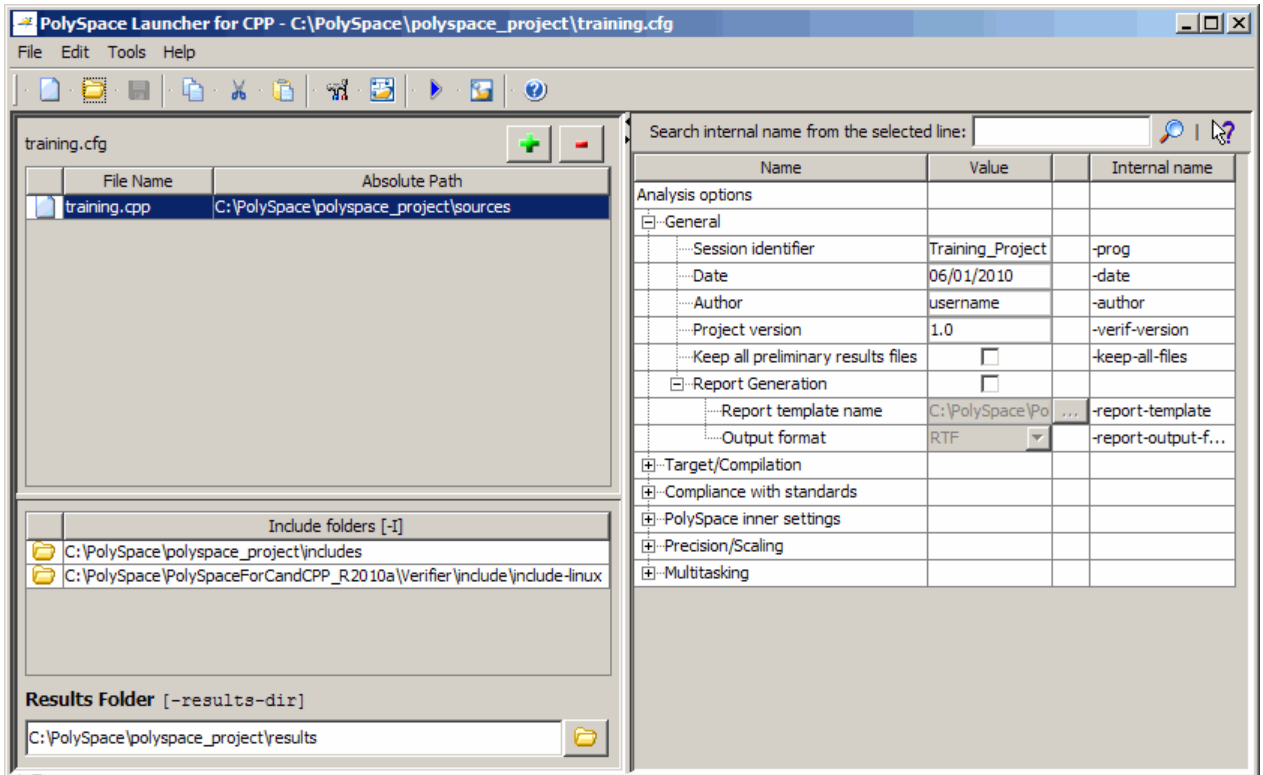
Specifying Source Files

To specify the source files for your project:

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

The **Please select a file** dialog box appears.
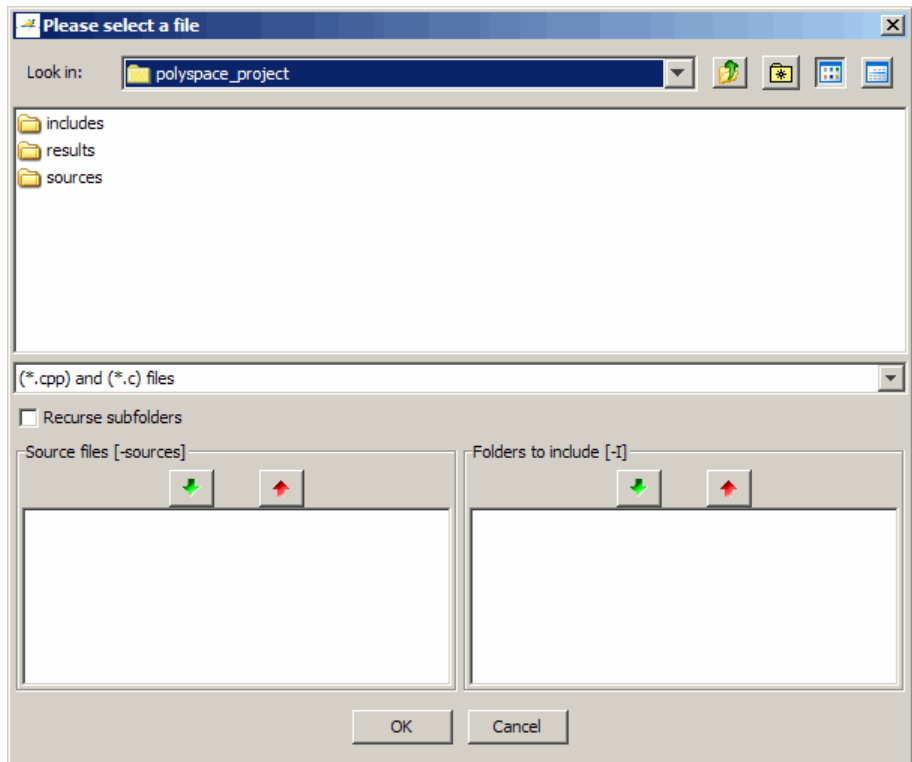
2 In the **Look in** field, navigate to your project folder containing your source files.

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

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

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

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

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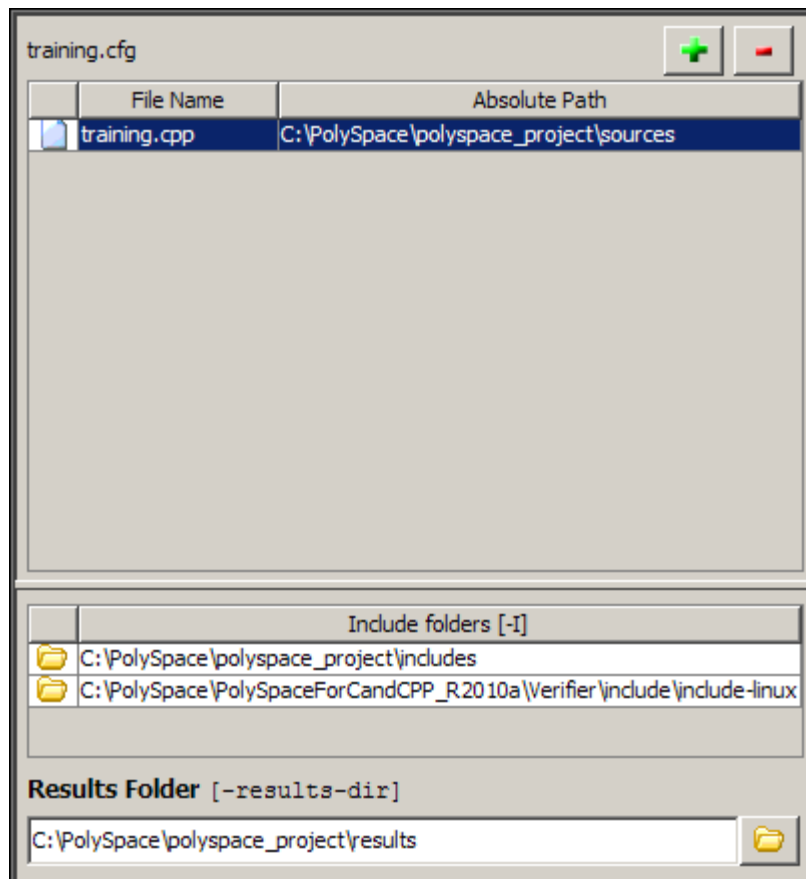
Specifying Include Folders

To specify the include folders for the project:

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

The **Please select a file** dialog box appears.
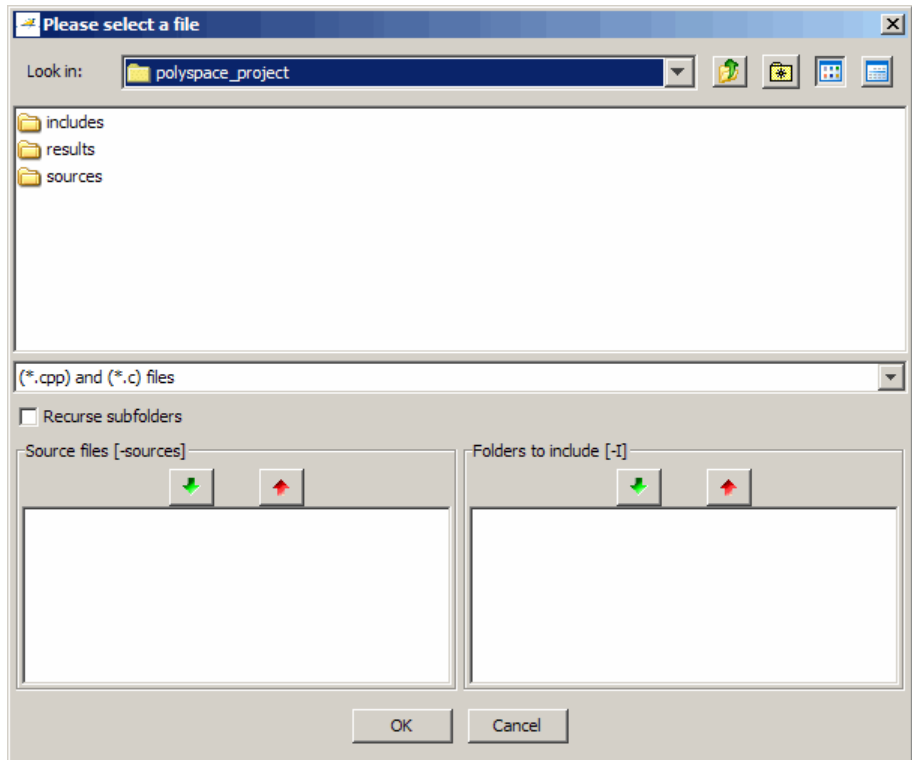
2 In the **Look in** field, navigate to your project folder.

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

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

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

The include folders you selected appear in the Include folders section on the left side of the Launcher window.
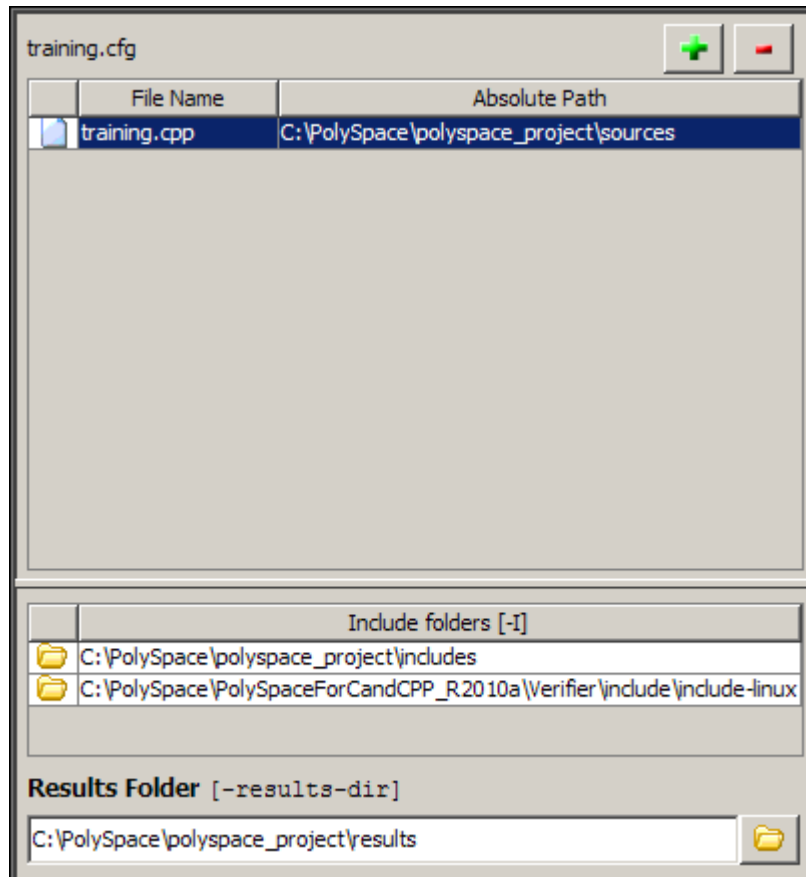
Specifying Results Folder

To specify the results folder for the project:

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

The files section of the Launcher window now looks like:
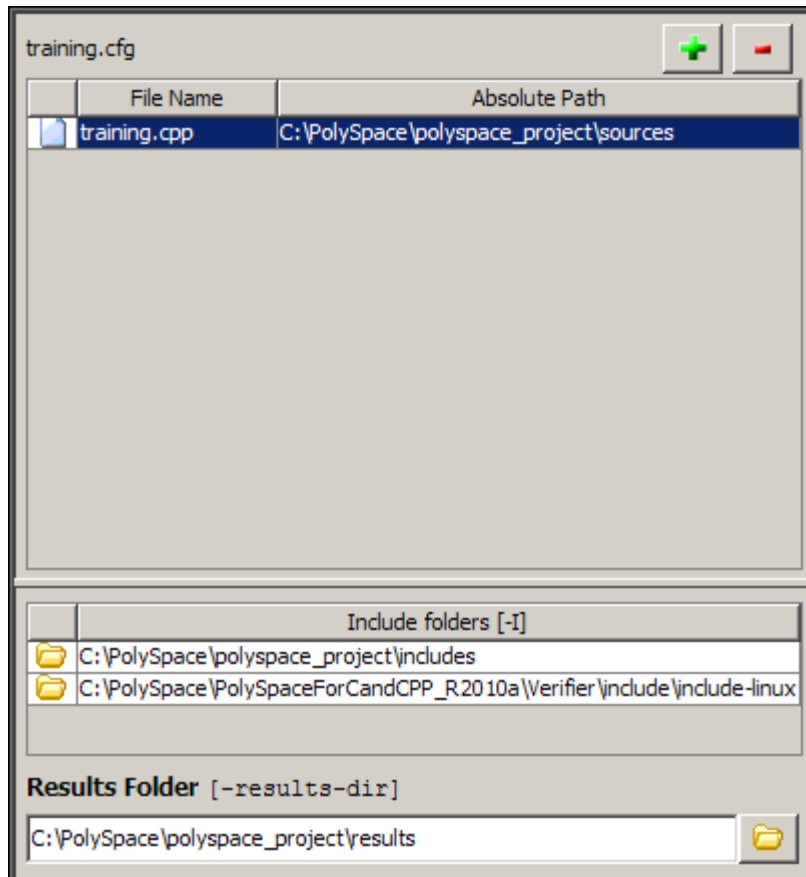
Specifying Analysis Options

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

To specify General parameters for your project:

1 In the Analysis options section of the Launcher window, expand **General**.

2 The General options appear.

Name	Value		Internal name
Analysis options			
⊟··General			
····Session identifier	New_Project		-prog
····Date	06/01/2010		-date
····Author	username		-author
····Project version	1.0		-verif-version
····Keep all preliminary results files	☐		-keep-all-files
⊟··Report Generation	☐		
····Report template name	C:\PolySpace\P	...	-report-template
····Output format	RTF	▼	-report-output-format
⊞··Target/Compilation			
⊞··Compliance with standards			
⊞··PolySpace inner settings			
⊞··Precision/Scaling			
⊞··Multitasking			

Search internal name from the selected line:

3 Specify the appropriate general parameters for your project.

For detailed information about specific analysis options, see "Option Descriptions" in the *PolySpace Products for C++ Reference*.

Configuring Text and XML Editors

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

To configure your text and .XML editors:

1 Select **Edit > Preferences**.

The Preferences dialog box opens.

2 Select the **Editors** tab.

The Editors tab opens.
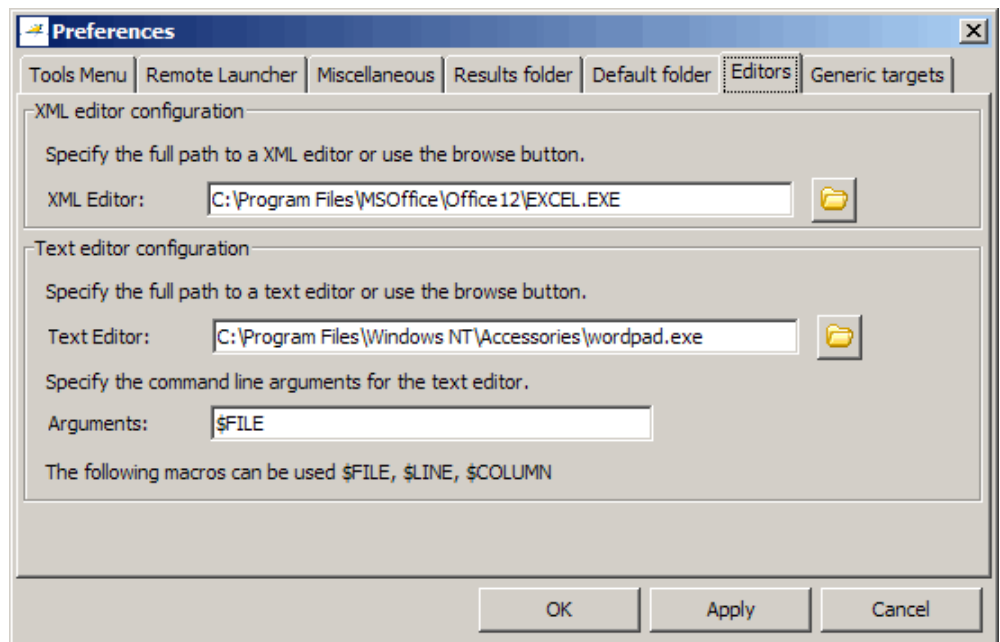
3 Specify an XML editor to use to view JSF++ reports. For example:

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

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

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

5 Specify command line arguments for the text editor. For example:

```
$FILE
```

6 Click **OK**.

Saving the Project

To save the project:

1 Select **File > Save project**. The **Save the project as** dialog box appears.
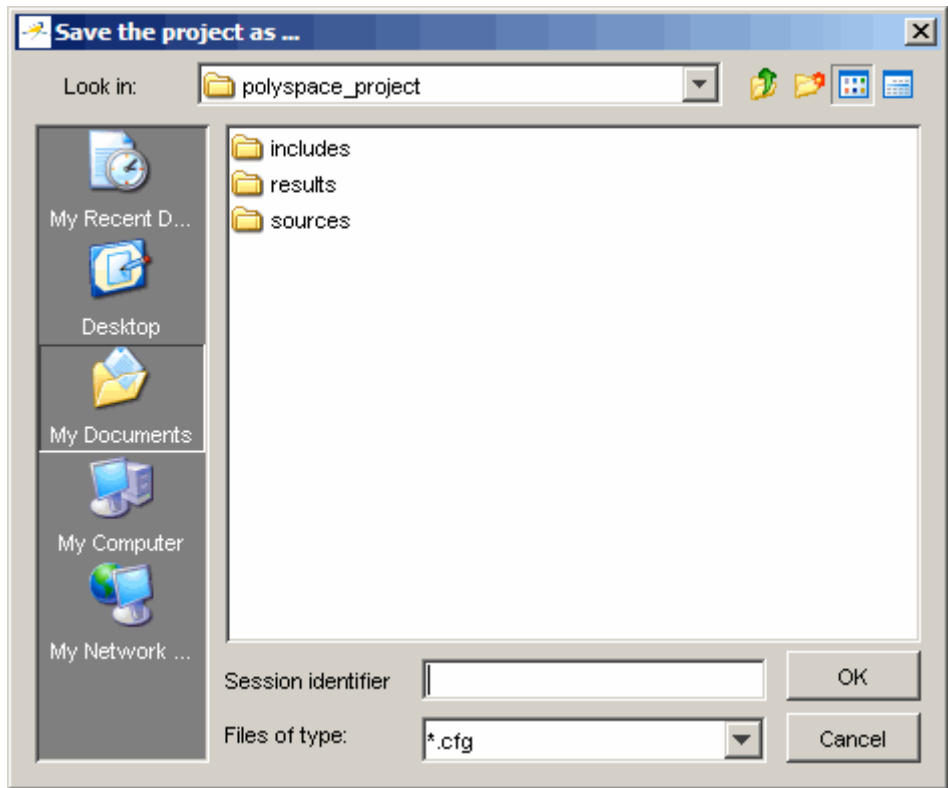
2 In **Look in**, select your project folder.

3 In **Session identifier**, enter a name for your project.

4 Click **OK** to save the project and close the dialog box.

Specifying Options to Match Your Quality Objectives

In this section...
"Quality Objectives Overview" on page 4-19
"Choosing Contextual Verification Options" on page 4-19
"Choosing Strict or Permissive Verification Options" on page 4-21
"Choosing Coding Rules" on page 4-23

Quality Objectives Overview

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

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

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

Choosing Contextual Verification Options

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

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

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 Viewer. For more information, see "Annotating Code to Indicate Known Run-Time Errors" on page 6-26.

To specify contextual verification for your project:

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

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

Name	Value		Internal name
Analysis options			
⊞ General			
⊞ Target/Compilation			
⊞ Compliance with standards			
⊟ PolySpace inner settings			
⊞ Run a verification unit by unit	☐		-unit-by-unit
⊞ Specify a Visual Studio compliant main	☐		
⊞ Generate a main for a given class	☑		
⊟ Generate a main for the given functions	☐		
Function calls	unused ▼	...	-main-generator-calls
⊟ Main generation general options			
First function to call			-function-called-before-main
Write accesses to global variables	uninit ▼	...	-main-generator-writes-variables
⊟ Stubbing			
Variable range setup		...	-data-range-specifications
No automatic stubbing	☐		-no-automatic-stubbing
⊞ Assumptions			

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

- **Variable range setup (`-data-range-specifications`)** – Activates the DRS option, allowing you to set specific data ranges for a list of global variables.

- **Write accesses to global variables (`-main-generator-writes-variables`)** – Specifies how the generated main initializes global variables.

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

- **Function calls (`-main-generator-calls`)** – Specifies how the generated main calls functions.

- **First function to call (`-function-called-before-main`)** – Specifies an initialization function called after initialization of global variables but before the main loop.

5 To control stubbing behavior, use the following option:

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

For more information on these options, see "Options Description" in the *PolySpace Products for C++ Reference*.

Choosing Strict or Permissive Verification Options

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

Note If you are aware of run-time errors in your code but still want to run a verification, you can annotate your code so that these known errors are highlighted in the Viewer. For more information, see "Annotating Code to Indicate Known Run-Time Errors" on page 6-26.

To specify the strictness of your verification:

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

2 In addition, expand **PolySpace Inner Settings > Assumptions**.

Permits overflowing computations on constants	☐		-ignore-constant-overflows
Continue even with undefined global variables	☐		-allow-undef-variables
Do not check the sign of operand in left shifts	☐		-allow-negative-operand-in-shift
Give all warnings	☐		-Wall
⊟ PolySpace inner settings			
⊞ Run a verification unit by unit	☐		-unit-by-unit
⊞ Specify a Visual Studio compliant main	☐		
⊞ Generate a main for a given class	☐		
⊞ Generate a main for the given functions	☐		
⊞ Main generation general options			
⊞ Stubbing			
⊟ Assumptions			
Ignore float rounding	☐		-ignore-float-rounding
Detect overflows on unsigned integers	☐		-detect-unsigned-overflows

3 Use the following options to make verification more strict:

- **Detect overflows on unsigned integers (`-detect-unsigned-overflows`)** – Verification is more strict with overflowing computations on unsigned integers.

- **Give all warnings (`-wall`)** – Specifies that all C compliance warnings are written to the log file during compilation.

4 Use the following options to make verification more permissive:

- **Do not check the sign of operand in left shifts (`-allow-negative-operand-in-shift`)** – Verification allows a shift operation on a negative number.

- **Permits overflowing computations on constants (`-ignore-constant-overflows`)** – Verification is permissive with overflowing computations on constants.

- **Continue even with undefined global variables (`-allow-undef-variables`)** – Verification does not stop due to errors caused by undefined global variables.

For more information on these options, see "Options Description" in the *PolySpace Products for C++ Reference*.

Choosing Coding Rules

PolySpace software can check that your code complies with specified coding rules. Before starting code verification, you should consider implementing coding rules, and choose which rules to enforce.

For more information, see "Setting Up Project to Check Coding Rules" on page 4-24.

Note If you are aware of coding rule violations, but still want to run a verification, you can annotate your code so that these known violations are highlighted in the Launcher. For more information, see "Annotating Code to Indicate Known Coding Rule Violations" on page 6-24.

Setting Up Project to Check Coding Rules

PolySpace Coding Rules Checker Overview

The PolySpace coding rules checker helps you comply with established C++ coding standards. The coding rules checker can check either MISRA C++ or JSF++ coding standards. These coding standards are designed to improve the robustness of C++ code, and improve maintainability.

The PolySpace coding rules checker enables PolySpace software to provide messages when MISRA C++ or JSF++ rules are not respected. Most messages are reported during the compile phase of a verification.

Note The PolySpace MISRA C++ checker is based on MISRA C++:2008 – "Guidelines for the use of the C++ language in critical systems." For more information on these coding standards, see http://www.misra-cpp.com.

The PolySpace JSF C++ checker is based on JSF++:2005. For more information on these coding standards, see http://www.jsf.mil/downloads/documents/JSF_AV_C++_Coding_Standards_Rev_C.doc.

Checking Compliance with Coding Rules

To check coding rules compliance, you set an option in your project before running a verification. PolySpace software finds the violations during the compile phase of a verification. When you have addressed all coding rules violations, you run the verification again.

To set the coding rules checking option:

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

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

⊟···Coding rules checker			
⊟···Check JSF C++ rules	☑		
···JSF C++ rules configuration		...	-jsf-coding-rules
⊟···Check MISRA C++ rules	☐		
···MISRA C++ rules configuration		...	-misra-cpp
···Files and folders to ignore		...	-includes-to-ignore

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

2 Select either the **Check JSF C++ rules** or **Check MISRA C++** rules check box.

3 Specify which rules to check and which, if any, files to exclude from the checking.

Note For more information on using the coding rules checker, see Chapter 12, "Checking Coding Rules".

Creating a Coding Rules File

You must have a rules file to run a verification with MISRA C++ or JSF++ checking. You can use an existing file or create a new one.

To create a new rules file:

1 Click the button ⎵...⎵ to the right of the **Rules configuration** option.

The New File window opens, allowing you to create a new MISRA C++ or JSF++ rules file, or open an existing file.
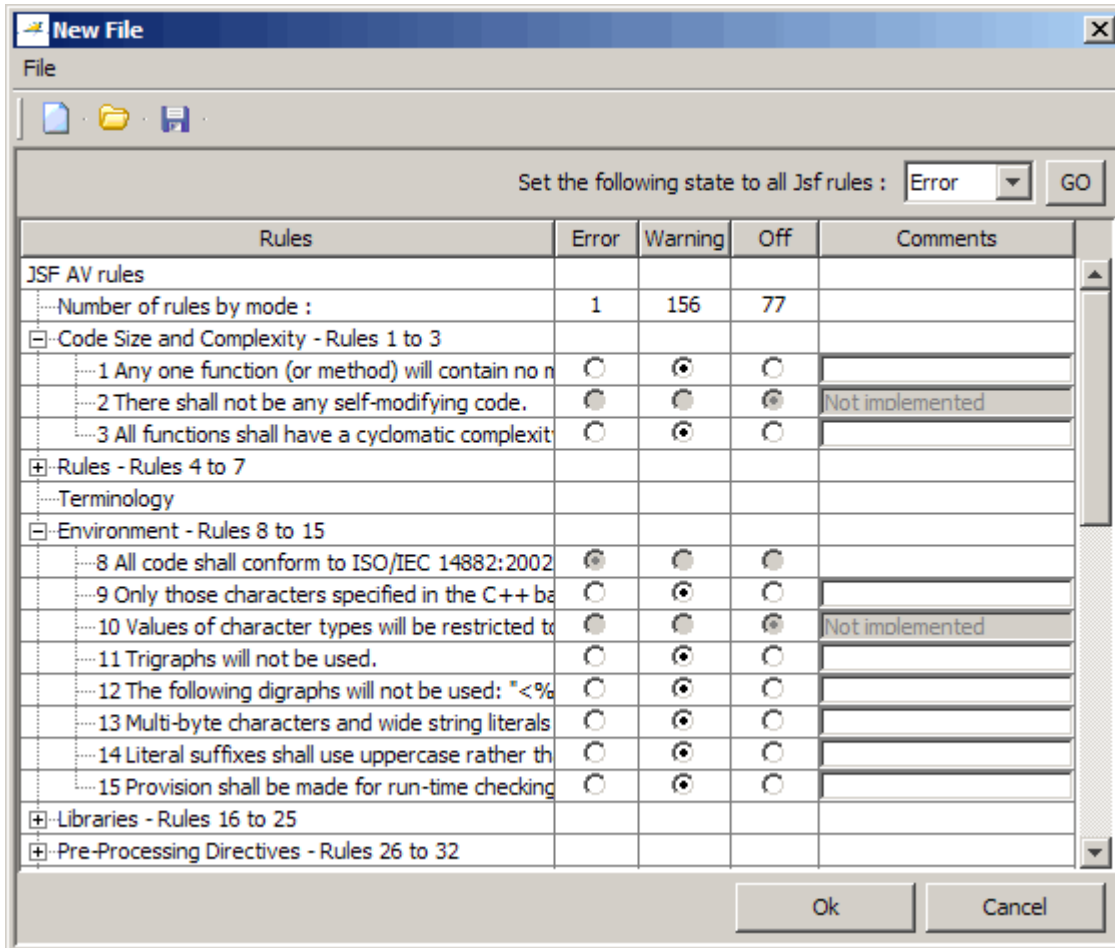
2 For each coding rule, specify one of these states:

State	Causes the verification to...
Error	End after the compile phase when this rule is violated.
Warning	Display warning message and continue verification when this rule is violated.
Off	Skip checking of this rule.

> **Note** The default state for most rules is `Warning`. The state for rules that have not yet been implemented is `Off`. Some rules always have state `Error` (you cannot change the state of these).

3 Click **OK** to save the rules and close the window.

The **Save as** dialog box opens.

4 In **File**, enter a name for your rules file.

5 Click **OK** to save the file and close the dialog box.

> **Note** If your project uses a dialect other than ISO, some JSF++ coding rules may not be completely checked. For example, AV Rule 8: "All code shall conform to ISO/IEC 14882:2002(E) standard C++."

Excluding Files from Coding Rules Checking

You can exclude files from MISRA C++ or JSF++ checking. For example, you may want to exclude some included files.

To exclude files from MISRA C++ or JSF++ checking:

1 Click the button ⬚ to the right of the **Files and folders to ignore** option.

The Files and folders to ignore (includes-to-ignore) dialog box opens.
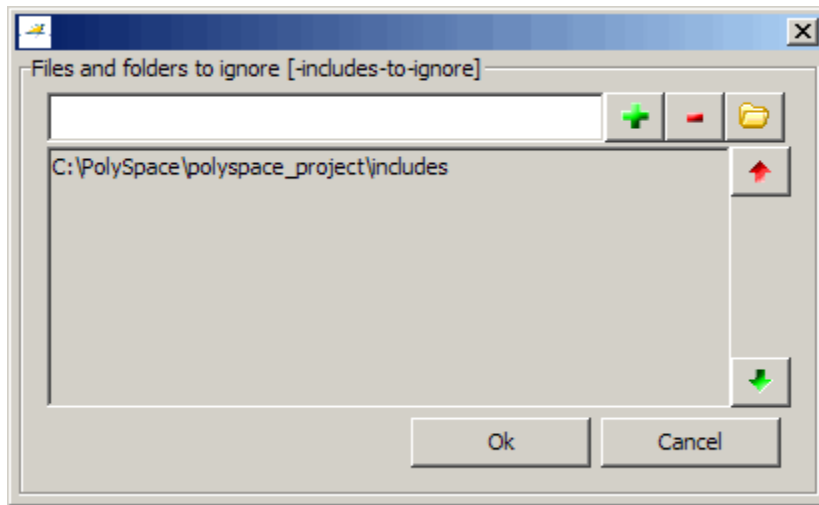
2 Click the folder icon .

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

3 Select the files or folders you want to exclude.

4 Click **OK**.

The select files and folders appear in the list of files to ignore.

5 Click **OK** to close the dialog box.

Setting Up Project for Generic Target Processors

In this section...

Project Model Files

What Is a PolySpace Project Model File?

A PolySpace project model file is a project file that includes generic target processors. You can use this file to share project information with your development team.

Although you can populate a project with information, such as source files and project options, from a project model file, you cannot run a verification with a project model file. You must have a configuration file to run a verification.

Workflow for Using Project Model Files

A PolySpace project model file is a project file that includes generic target processors. A development team uses this file to share project information. The workflow is:

1 A team leader creates a project model file (`.ppm`). This file has the analysis options for the project, including generic targets.

2 The team leader distributes the `.ppm` file to the team.

3 A developer opens the `.ppm` file. From this file, PolySpace software populates the project parameters and the generic targets in the preferences.

4 The developer adds source files, include folders, and a results folder to the project and saves it as a configuration file (`.cfg`).

5 The developer launches a verification with the `.cfg` file.

Creating Project Model Files

You use the PolySpace Launcher to create a PolySpace project model file.

To create a project model file:

1 Select **File > New Project** to create a new project.

2 Define the generic target, as described in the following sections.

3 Select **File > Save project**.

The **Save the project as** dialog box appears.

4 Select ***.ppm** from the **Files of type** menu.

5 In **Session identifier**, enter a name for your project model file.

6 Click **OK** to save the file and close the dialog box.

Viewing Existing Generic Targets

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

To view existing generic targets:

1 Select **Edit > Preferences**.

The **Preferences** dialog box appears.

2 Select the **Generic targets** tab.

Previously defined generic targets appear in the generic targets list.
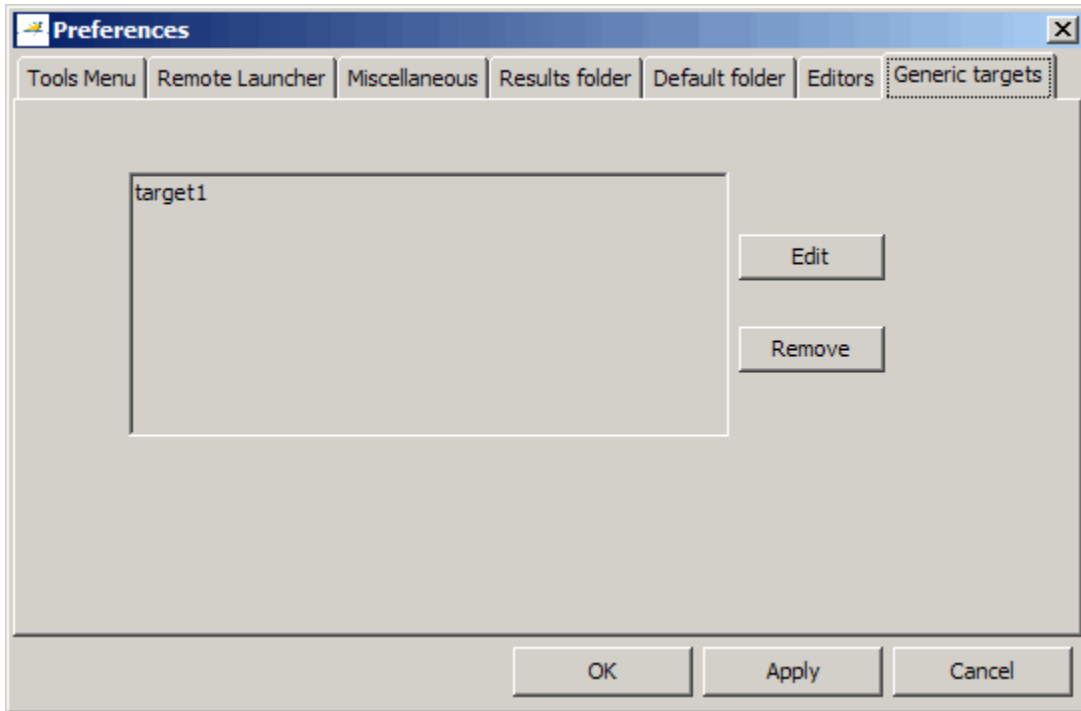
3 Click **Cancel** to close the dialog box.

Defining Generic Targets

If your application is designed for a custom target processor, you can configure many basic characteristics of the target by selecting the PST Generic target, and specifying the characteristics of your processor.

To configure a generic target:

To define a generic target:

1 In **Analysis options**, expand **Target/Compilation**.

2 Click the down arrow to open the **Target processor type** menu.
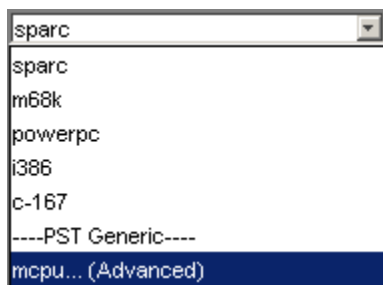
3 Select **mcpu... (Advanced)**.
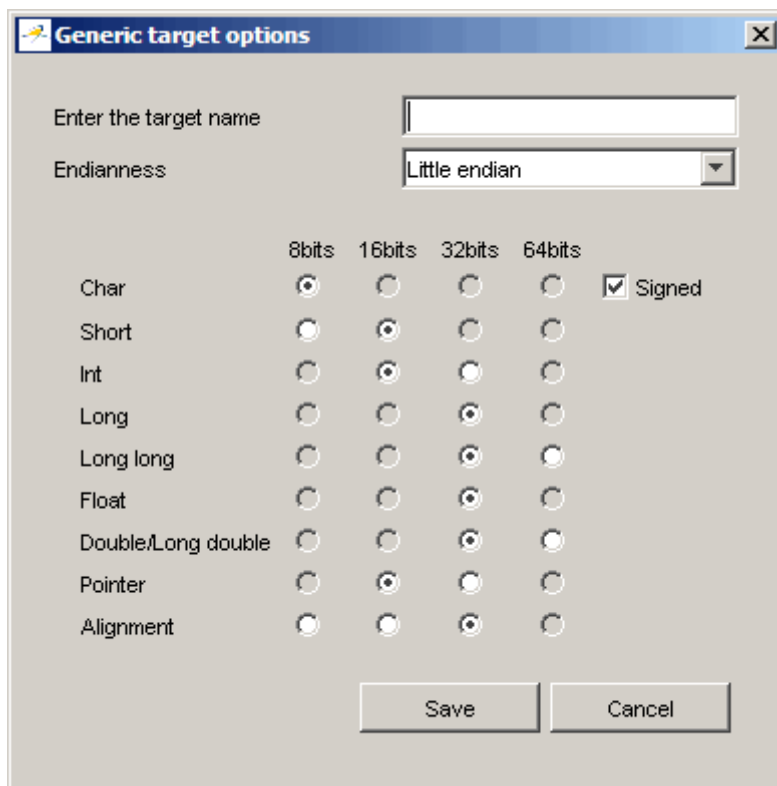
The **Generic target options** dialog box appears.

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

5 Specify the appropriate parameters for your target, such as the size of basic types, and alignment with arrays and structures.

For example, when the alignment of basic types within an array or structure is always 8, it implies that the storage assigned to arrays and structures is strictly determined by the size of the individual data objects (without fields and end padding).

Note For a list of common generic targets, see "Common Generic Targets" on page 5-9.

For detailed information on each target option, see "GENERIC ADVANCED TARGET OPTIONS"in the PolySpace Products for C++ Reference.

6 Click **Save** to save the generic target options and close the dialog box.

Deleting a Generic Target

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

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

To delete a generic target:

1 Select **Edit > Preferences**.

The **Preferences** dialog box appears.

2 Select the **Generic targets** tab.

3 Select the target you want to remove.

4 Click **Remove**.

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

Creating a Configuration File from a PolySpace Project Model File

To run a verification, you must have a configuration file, not just a project model file. However, you can create a configuration file from a project model file.

To create a configuration file from a project model file:

1 Open the project model file.

> **Note** When opening files, you can select **Project Model (*.ppm) files** in the File of type section to view only project model files.

Opening the project model file populates the:

- Generic targets in the preferences
- Analysis options and other project information

2 Enter additional project information, such as the results folder and source files.

> **Note** If you enter the results folder and source files in the project before you save it as a PolySpace project model file, then that information is saved in the file and appears in the project when you open the file.

3 Select **File > Save project**.

The **Save the project as** dialog box appears.

4 Enter a name for your configuration file.

5 Leave the default type as `*.cfg`.

6 Click **OK** to save the project and close the dialog box.

Emulating Your Runtime Environment

- "Setting Up a Target" on page 5-2
- "Specifying Data Ranges for Variables and Functions (Contextual Verification)" on page 5-21

Setting Up a Target

Target/Compiler Overview

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

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

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

Specifying Target/Compilation Parameters

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

To specify target parameters for your project:

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

2 The Target/Compilation options appear.

Name	Value		Internal name
Analysis options			
⊞─General			
⊟─Target/Compilation			
──Target processor type	sparc ▼	...	-target
──Operating system target for PolySpace stubs	Solaris ▼		-OS-target
──Defined Preprocessor Macros		...	-D
──Undefined Preprocessor Macros		...	-U
──Include		...	-include
──Command/script to apply to preprocessed files		...	-post-preprocessing-command
──Command/script to apply after the end of the code verification		...	-post-analysis-command
⊞─Compliance with standards			
⊞─PolySpace inner settings			
⊞─Precision/Scaling			
⊞─Multitasking			

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

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

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

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

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

Note The targets Motorola ST7, ST9, Hitachi H8/300, H8/300L, Hitachi H8/300H, H8S, H8C, H8/Tiny are described in the next section.

Target	char	short	int	long	long long	float	double	long double	ptr	char is	Endian	ptr diff type
sparc	8	16	32	32	64	32	64	128	32	signed	Big	int, long
i386	8	16	32	32	64	32	64	96	32	signed	Little	int, long
c-167	8	16	16	32	32	32	64	64	16	signed	Little	int
m68k / ColdFire[1]	8	16	32	32	64	32	64	96	32	signed	Big	int, long
powerpc	8	16	32	32	64	32	64	128	32	unsigned	Big	int, long

If your target processor does not match the characteristics of any processor described above, contact The MathWorks technical support for advice.

The following table describes target processors that are not fully supported by PolySpace products. Nevertheless, the target processor mentioned in column "Nearest Processor" can be chosen for a Server verification, knowing that information in red is not compatible in both target processors.

---

1. The M68k family (68000, 68020, etc.) includes the "ColdFire" processor

Target	char	short	int	long	long long	float	double	long double	ptr	char is	ptr diff type	Nearest target processor
tms470r1x	8	16	32	32	N/A	32	64	64	32	signed	int, long	i386
mpc555	8	16	32	32	64	32	64	64	32	signed	int, long	i386
hc12	8	16	16	32	32	32	32	32	16	signed	int	c-167

Modifying Predefined Target Processor Attributes

You can modify certain attributes of the predefined target processors. If your specific processor is not listed, you may be able to specify a similar processor and modify its characteristics to match your processor.

To modify target processor attributes:

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

The Target/Compilation options appear.

2 Select the **Target processor type** you want to use.

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

The Advanced target options dialog box opens.
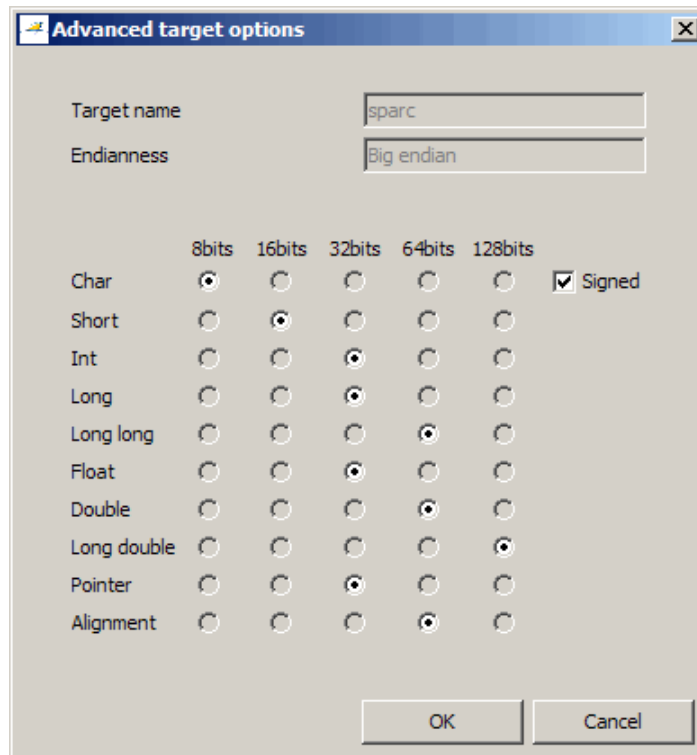
4 Modify the attributes as needed.

For information on each target option, see "GENERIC ADVANCED TARGET OPTIONS"in the *PolySpace Products for C++ Reference*.

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

Defining Generic Targets

If your application is designed for a custom target processor, you can configure many basic characteristics of the target by selecting the PST Generic target, and specifying the characteristics of your processor.

To configure a generic target:

To define a generic target:

1 In **Analysis options**, expand **Target/Compilation**.

2 Click the down arrow to open the **Target processor type** menu.
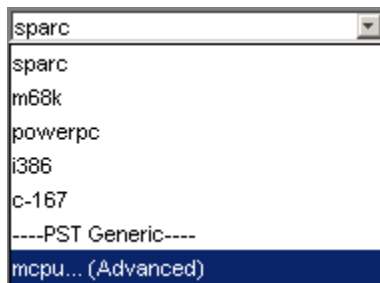
3 In the **Target processor type** menu, select **mcpu... (Advanced)**.
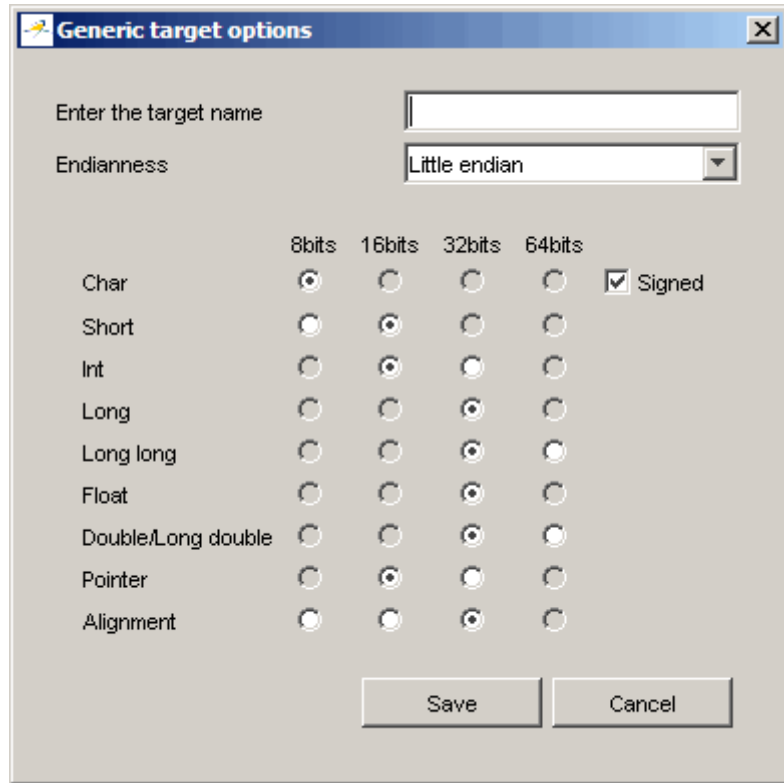
The **Generic target options** dialog box appears.

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

5 Specify the appropriate parameters for your target, such as the size of basic types, and alignment with arrays and structures.

For example, when the alignment of basic types within an array or structure is always 8, it implies that the storage assigned to arrays and structures is strictly determined by the size of the individual data objects (without fields and end padding).

Note For more information, see "GENERIC ADVANCED TARGET OPTIONS" in the PolySpace Products for C++ Reference.

6 Click **Save** to save the generic target options and close the dialog box.

For more information, see "Setting Up Project for Generic Target Processors" on page 4-29.

Common Generic Targets

The following tables describe the characteristics of common generic targets.

ST7 (Hiware C compiler : HiCross for ST7)

ST7	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16	32	32	32	32	32	16/32	unsigned	Big
alignment	8	16/8	16/8	32/16/8	32/16/8	32/16/8	32/16/8	32/16/8	32/16/8	N/A	N/A

ST9 (GNU C compiler : gcc9 for ST9)

ST9	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16	32	32	32	64	64	16/64	unsigned	Big
alignment	8	8	8	8	8	8	8	8	8	N/A	N/A

Hitachi H8/300, H8/300L

Hitachi H8/300, H8/300L	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16/32	32	64	32	654	64	16	unsigned	Big
alignment	8	16	16	16	16	16	16	16	16	N/A	N/A

Hitachi H8/300H, H8S, H8C, H8/Tiny

Hitachi H8/300H, H8S, H8C, H8/Tiny	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16/32	32	64	32	64	64	32	unsigned	Big
alignment	8	16	32/16	32/16	32/16	32/16	32/16	32/16	32/16	N/A	N/A

The size of some basic types is configurable using the `-int-is-32bits` option, compiler memory model option, and near/far pointer syntax.

The alignment of some basic types with arrays and structures is configurable (depending on the compiler implementation or optimization options). For example, when the alignment of basic types within an array or structure is always 8, it implies that the storage assigned to arrays and structures is strictly determined by the size of the individual data objects (without fields and end padding).

The sign of char is configurable using `-default-sign-of-char` [`signed`|`unsigned`].

Viewing Existing Generic Targets

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

To view existing generic targets:

1 Select **Edit > Preferences**.

The **Preferences** dialog box appears.

2 Select the **Generic targets** tab.

Previously defined generic targets appear in the generic targets list.

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

Deleting a Generic Target

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

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

To delete a generic target:

1 Select **Edit > Preferences**.

The **Preferences** dialog box appears.

2 Select the **Generic targets** tab.

3 Select the target you want to remove.

4 Click **Remove**.

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

Compiling Operating System Dependent Code (OS-target issues)

This section describes the options required to compile and verify code designed to run on specific operating systems. It contains the following:

- "List of Predefined Compilation Flags" on page 5-12
- "My Target Application Runs on Linux" on page 5-15
- "My Target Application Runs on Solaris" on page 5-15
- "My Target Application Runs on Vxworks" on page 5-16
- "My Target Application Does Not Run on Linux, vxworks nor Solaris" on page 5-16

List of Predefined Compilation Flags

The following table shown for each OS-target, the list of compilation flags defined by default, including pre-include header file (see also include):

-OS-target	Compilation flags	-include file	Minimum set of options
Linux	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D__STL_CLASS_PARTIAL_ SPECIALIZATION -D__GNU_SOURCE -D__STDC__ -D__ELF__ -Dunix -D__unix -D__unix__ -Dlinux	<product_dir>/ cinclude/ pst-linux.h	polyspace-[desktop-]cpp -OS-target Linux \ -I <product_dir>/include/ include-linux \ -I <product_dir>/include/ include-linux/next Where the PolySpace product has

-OS-target	Compilation flags	-include file	Minimum set of options
	-D__linux -D__linux__ -Di386 -D__i386 -D__i386__ -Di686 -D__i686 -D__i686__ -Dpentiumpro -D__pentiumpro -D__pentiumpro__		been installed in the folder \<product_dir\>
vxWorks	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D__STL_CLASS_PARTIAL_ SPECIALIZATION -DANSI_PROTOTYPES -DSTATIC= -DCONST=const -D__STDC -D__GNU_SOURCE -Dunix -D__unix -D__unux__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__ -D__svr4 -D__SVR4	\<product_dir\>/ cinclude/ pstvxworks. h	polyspace-[desktop-]cpp \ -OS-target vxworks \ -I /your_path_to/ Vxworks_include_folders

-OS-target	Compilation flags	-include file	Minimum set of options

visual /visual6	-D__SIZE_TYPE__=unsigned
	-D__PTRDIFF_TYPE__=int
	-D__STRICT_ANSI__
	-D__inline__=inline
	-D__signed__=signed
	-D__gnuc_va_list=va_list
	-D_POSIX_SOURCE
	-D__STL_CLASS_PARTIAL_
	SPECIALIZATION
		`<product_dir>/`
		`cinclude/`
		`pstvisual. h`	

Solaris	-D__SIZE_TYPE__=unsigned
	-D__PTRDIFF_TYPE__=int
	-D__inline__=inline
	-D__signed__=signed
	-D__gnuc_va_list=va_list
	-D__STL_CLASS_PARTIAL_
	SPECIALIZATION
	-D__GNU_SOURCE
	-D__STDC
	-D__GCC_NEW_VARARGS__
	-Dunix
	-D__unix
	-D__unux__
	-Dsparc
	-D__sparc
	-D__sparc__
	-Dsun
	-D__sun
	-D__sun__
	-D__svr4
	-D__SVR4
			If PolySpace runs on a Linux machine:
			
			`polyspace-[desktop-]cpp \`
			`-OS-target Solaris \`
			`-I`
			`/your_path_to_solaris_include`
			
			If PolySpace runs on a Solaris machine:
			
			`polyspace-cpp \`
			`-OS-target Solaris \`
			`-I /usr/include`

no-predefined-OS	-D__SIZE_TYPE__=unsigned
	-D__PTRDIFF_TYPE__=int
	-D__STRICT_ANSI__
	-D__inline__=inline
	-D__signed__=signed
	-D__gnuc_va_list=va_list
			`polyspace-[desktop-]cpp \`
			`-OS-target no-predefined-OS`
			`\`
			`-I /your_path_to/`
			`MyTarget_include_folders`

-OS-target	Compilation flags	-include file	Minimum set of options
	-D_POSIX_SOURCE -D__STL_CLASS_PARTIAL_ SPECIALIZATION		

Note This list of compiler flags is written in every log file.

My Target Application Runs on Linux

The minimum set of options is as follows:

```
polyspace-cpp \
 -OS-target Linux \
 -I /usr/local/PolySpace/CURRENT-VERSION/include/include-linux \
 -I /usr/local/PolySpace/CURRENT-VERSION/include/include-linux/next \
 ...
```

where the PolySpace product has been installed in the folder
/usr/local/PolySpace/CURRENT-VERSION.

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

```
polyspace-cpp \
 -OS-target Linux \
 -I POLYSPACE_C\Verifier\include\include-linux \
 -I POLYSPACE_C\Verifier\include\include-linux\next \
 ...
```

where the PolySpace product has been installed in the folder POLYSPACE_C.

My Target Application Runs on Solaris

If PolySpace software runs on a Linux machine:

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

5-15

If PolySpace runs on a Solaris™ machine:

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

My Target Application Runs on Vxworks

If PolySpace runs on either a Solaris or a Linux machine:

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

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

If PolySpace runs on either a Solaris or a Linux machine:

```
polyspace-cpp \
 -OS-target no-predefined-OS \
 -I /your_path_to/MyTarget_include_folders
```

Ignoring or Replacing Keywords Before Compilation

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

To ignore keywords:

1 Save the following template in `c:\PolySpace\myTpl.pl`.

2 In the Target/Compilation options, select **Command/script to apply to preprocessed file**s.

3 Select `myTpl.pl` using the browse button.

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

Content of the myTpl.pl file

```perl
#!/usr/bin/perl

################################################################
# Post Processing template script
#
################################################################
# Usage from Launcher GUI:
#
# 1) Linux: /usr/bin/perl PostProcessingTemplate.pl
# 2) Solaris: /usr/local/bin/perl PostProcessingTemplate.pl
# 3) Windows: \Verifier\tools\perl\win32\bin\perl.exe <pathtoscript>\
PostProcessingTemplate.pl
#
################################################################

$version = 0.1;

$INFILE = STDIN;
$OUTFILE = STDOUT;

while (<$INFILE>)
{

 # Remove far keyword
 s/far//;

 # Remove "@ 0xFE1" address constructs
 s/\@\s0x[A-F0-9]*//g;

 # Remove "@0xFE1" address constructs
 # s/\@0x[A-F0-9]*//g;

 # Remove "@ ((unsigned)&LATD*8)+2" type constructs
 s/\@\s\(\(\(unsigned\)\)\&[A-Z0-9]+\*8\)\)\+\d//g;

 # Convert current line to lower case
 # $_ =~ tr/A-Z/a-z/;
```

```
 # Print the current processed line
 print $OUTFILE $_;
}
```

Perl Regular Expression Summary

```
############################################################
# Metacharacter What it matches
############################################################
# Single Characters
# . Any character except newline
# [a-zO-9] Any single character in the set
# [^a-zO-9] Any character not in set
# \d A digit same as
# \D A non digit same as [^O-9]
# \w An Alphanumeric (word) character
# \W Non Alphanumeric (non-word) character
#
# Whitespace Characters
# \s Whitespace character
# \S Non-whitespace character
# \n newline
# \r return
# \t tab
# \f formfeed
# \b backspace
#
# Anchored Characters
# \B word boundary when no inside []
# \B non-word boundary
# ^ Matches to beginning of line
# $ Matches to end of line
#
# Repeated Characters
# x? O or 1 occurence of x
# x* O or more x's
# x+ 1 or more x's
# x{m,n} Matches at least m x's and no more than n x's
# abc All of abc respectively
# to|be|great One of "to", "be" or "great"
```

```
#
# Remembered Characters
# (string) Used for back referencing see below
# \1 or $1 First set of parentheses
# \2 or $2 First second of parentheses
# \3 or $3 First third of parentheses
############################################################
# Back referencing
#
# e.g. swap first two words around on a line
# red cat -> cat red
# s/(\w+) (\w+)/$2 $1/;
#
############################################################
```

How to Gather Compilation Options Efficiently

The code is often tuned for the target. Rather than applying minor changes to the code, create a single polyspace.h file which will contain all target specific functions and options. The -include option can then be used to force the inclusion of the polyspace.h file in all source files under verification.

Where there are missing prototypes or conflicts in variable definition, writing the expected definition or prototype within such a header file will yield several advantages.

Direct benefits:

- The error detection is much faster since it will be detected during compilation rather than in the link or subsequent phases.

- The position of the error will be identified more precisely.

- There will be no need to modify original source files.

Indirect benefits:

- The file is automatically included as the very first file in all original .c files.

- The file can contain much more powerful macro definitions than simple -D options.

• The file is reusable for other projects developed under the same environment.

Example

This is an example of a file that can be used with the include option.

```
// The file may include (say) a standard include file implicitly
// included by the cross compiler
#include <stdio.h>
#include "another_file.h"

// Generic definitions, reusable from one project to another
#define far
#define at(x)

// A prototype may be positioned here to aid in the solution of
a link phase conflict between
// declaration and definition. This will allow detection of the
// same error at compilation time instead of at link time.
// Leads to:
// - earlier detection
// - precise localization of conflict at compilation time
void f(int);

// The same also applies to variables.
extern int x;
// Standard library stubs can be avoided,
// and OS standard prototypes redefined.
#define __polyspace_no_sscanf
#define __polyspace_no_fgetc
void sscanf(int, char, char, char, char, char);
void fgetc(void);
```

Specifying Data Ranges for Variables and Functions (Contextual Verification)

Overview of Data Range Specifications (DRS)

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

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

You can use DRS to set constraints on:

- Global variables
- Input parameters for user-defined functions called by the main generator

• Return values for stub functions

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

Specifying Data Ranges Using DRS Template

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

PolySpace software can analyze the files in your project, and generate a DRS template containing all the global variables, user defined functions, and stub functions for which you can specify data ranges. You can then modify this template to set data ranges.

To use a DRS template to set data ranges:

1 Open the Project for which you want to set data ranges.

2 Ensure that the Project contains all the source files and Include folders you want to verify, and specifies the Analysis options you want to use for the verification. The Compile phase of verification must complete successfully for the software to generate a DRS template.

3 In the Analysis options section of the Launcher window, select **PolySpace inner settings > Stubbing**.

4 In the **Variable range setup** row, select the browse button [...].

The PolySpace DRS configuration dialog box opens.
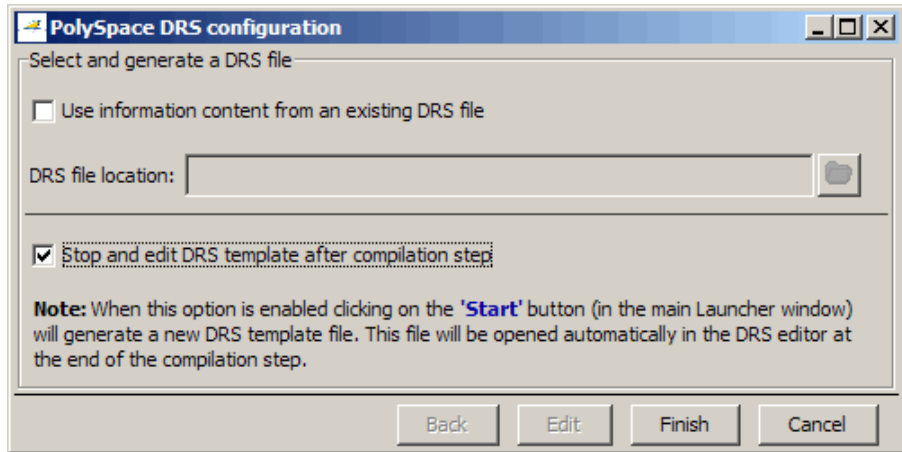
5 Select **Stop and edit DRS template after compilation step**, then click **Finish**.

6 Click the **Start** button ![Start].

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

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

| Back | Edit | Finish | Cancel |

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

7 Specify the data ranges for global variables, user-defined function inputs, and stub-function return values. For more information, see "DRS Configuration Settings" on page 5-25.

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

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

---

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

DRS Configuration Settings

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

Column	Settings
Name	Displays the list of variables and functions in your Project for which you can specify data ranges. This Column displays three expandable menu items: • **Globals** – Displays a list of all global variables in the Project. • **User defined functions** – Displays a list of all user-defined functions in the Project. Expand any function name to see a list of the input arguments for which you can specify a data range. • **Stubbed functions** – Displays a list of all stub functions in the Project. Expand any function name to see a list of the return values for which you can specify a data range.
File	Displays the name of the source file containing the variable or function.
Attributes	Displays information about the variable or function. For example, static variables display `static`.
Type	Displays the variable type.

Column	Settings
Main Generator Called	Applicable only for user-defined functions. Specifies whether the main generator calls the function: • **MAIN GENERATOR** – Main generator may call this function, depending on the value of the -main-generator-calls parameter. • **NO** – Main generator will not call this function. • **YES** – Main generator will call this function.
Init Mode	Specifies how the software assigns a range to the variable: • **MAIN GENERATOR** – Variable range is assigned depending on the settings of the main generator options -main-generator-writes-variables and -no-def-init-glob. • **IGNORE** – Variable is not assigned to any range, even if a range is specified. • **INIT** – Variable is assigned to the specified range only at initialization, and keeps the range until first write. • **PERMANENT** – Variable is permanently assigned to the specified range. If the variable is assigned outside this range during the program, no warning is provided. Use the globalassert mode if you need a warning. User-defined functions support only INIT mode. Stub functions support only PERMANENT mode.

Column	Settings
Init Range	Specifies the minimum and maximum values for the variable. You can use the keywords min and max to denote the minimum and maximum values of the variable type. For example, for the type long, min and max correspond to -2^31 and 2^31-1 respectively. You can also use hexadecimal values. For example: `0x12..0x100`
Initialize Pointer	Not applicable for C++ projects.
Init Allocated	Not applicable for C++ projects.
# Allocated Objects	Not applicable for C++ projects.
Global Assert	Specifies whether to perform an assert check on the variable at global initialization, and after each assignment.
Global Assert Range	Specifies the minimum and maximum values for the range you want to check.

Specifying Data Ranges Using Existing DRS Configuration

Once you have created a DRS configuration file for a Project, you can reuse the data ranges for subsequent verifications.

To specify an existing DRS configuration file for your Project:

1 Open the Project.

2 In the Analysis options section of the Launcher window, select **PolySpace inner settings > Stubbing**.

3 In the **Variable range setup** row, select the browse button ...

The PolySpace DRS configuration dialog box opens.
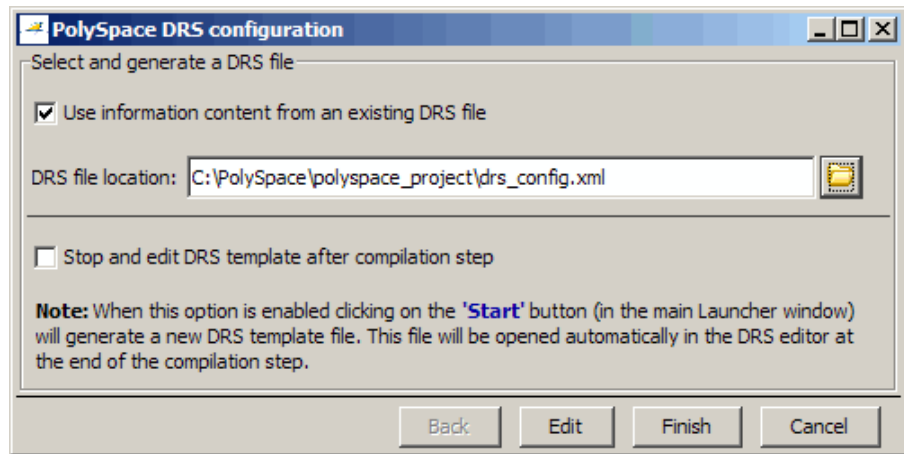
4 Select **Use information content from an existing DRS file**.

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

6 Click **Finish**.

   The PolySpace DRS configuration dialog box closes.

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

The software uses the specified DRS configuration file the next time you launch a verification.

Editing Existing DRS Configuration

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

To edit an existing DRS configuration:

1 Open the Project.

2 In the Analysis options section of the Launcher window, select **PolySpace inner settings > Stubbing**.

3 In the **Variable range setup** row, select the browse button [...].

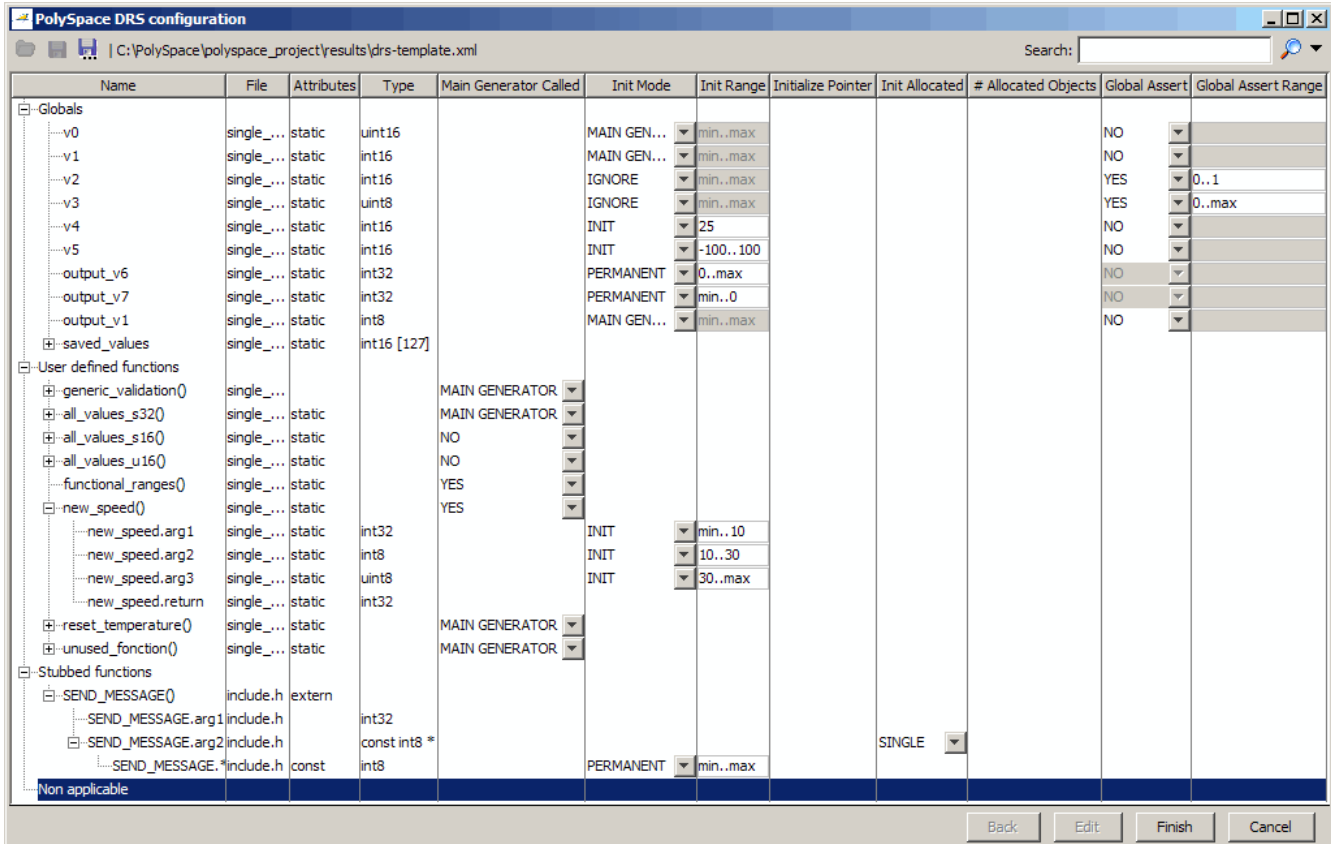The PolySpace DRS configuration dialog box opens.
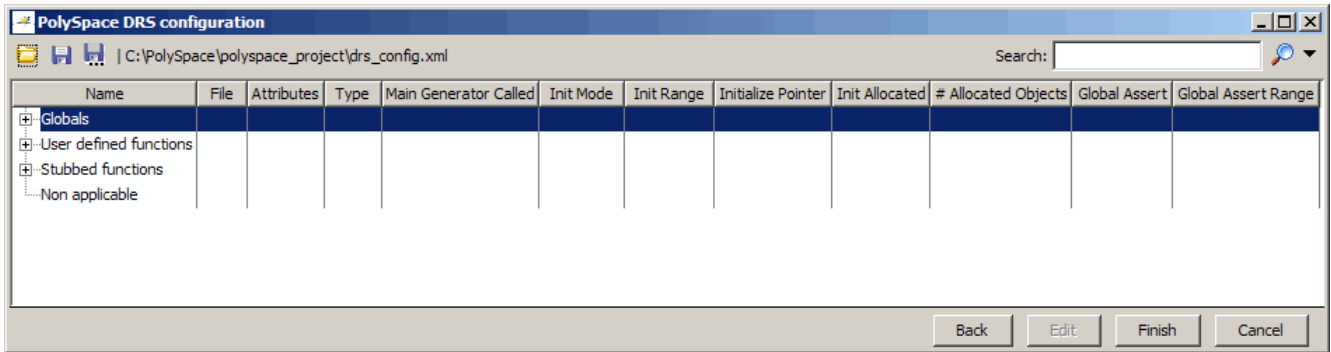
4 Specify the data ranges for global variables, user-defined function inputs, and stub-function return values.

5 Click ![Save DRS icon] (Save DRS), to save your DRS configuration file.

6 Click **Finish**.

The PolySpace DRS configuration dialog box closes.

Specifying Data Ranges Using Text Files

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

You can specify data ranges using the PolySpace DRS configuration interface (see "Specifying Data Ranges" on page 5-33), or you can provide a text file that contains a list of variables and data ranges.

Note If you used the DRS feature prior to R2010a, you created a text file to specify data ranges. The format of this file has not changed. You can use your existing DRS text file to specify data ranges.

To specify data ranges using a DRS text file:

1 Create a DRS text file containing the list of global variables (or functions) and their associated data ranges, as described in "File Format" on page 5-34.

2 Open your Project.

3 In the Analysis options section of the Launcher window, select **PolySpace inner settings > Stubbing**.

4 In the **Variable range setup** row, select the browse button [...].

The PolySpace DRS configuration dialog box opens.
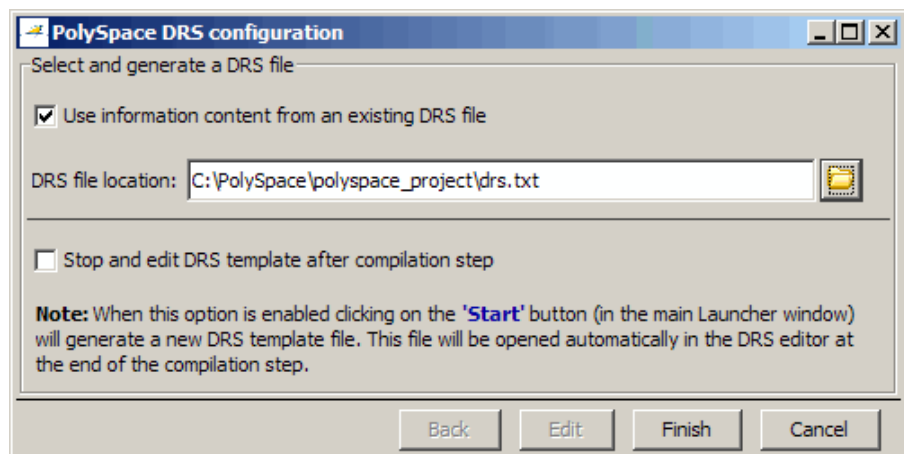
5 Select **Use information content from an existing DRS file**.

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

7 Click **Finish**.

The PolySpace DRS configuration dialog box closes.

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

When you launch a verification, the software automatically merges the data ranges in the text file with a DRS template for the project, and saves the information in the file drs-template.xml, located in your results folder.

You can continue to use the DRS text file for future verifications, or change the **DRS file location** to specify the generated file drs-template.xml (See "Specifying Data Ranges Using Existing DRS Configuration" on page 5-27).

If you specify the .xml template, you can then edit your data ranges using the DRS configuration interface (see "Editing Existing DRS Configuration" on page 5-28).

DRS Text File Format

The DRS file contains a list of global variables and associated data ranges. The point during verification at which the range is applied to a variable is controlled by the mode keyword: init, permanent, or globalassert.

The DRS file must have the following format:

variable_name min_value max_value <init|permanent|globalassert>

function_name.return min_value max_value permanent

- *variable_name* — The name of the global variable.
- *min_value* — The minimum value for the variable.
- *max_value* — The maximum value for the variable.

- `init` — The variable is assigned to the specified range only at initialization, and keeps it until first write.

- `permanent` — The variable is permanently assigned to the specified range. If the variable is assigned outside this range during the program, no warning is provided. Use the `globalassert` mode if you need a warning.

- `globalassert` — After each assignment, an assert check is performed, controlling the specified range. The assert check is also performed at global initialization.

- *function_name* — The name of the stub function.

Tips for Creating DRS Text Files

- You can use the keywords `"min"` and `"max"` to denote the minimum and maximum values of the variable type. For example, for the type long, min and max correspond to -2^31 and 2^31-1 respectively.

- You can use hexadecimal values. For example, `x 0x12 0x100 init`.

- Supported column separators are tab, comma, space, or semicolon.

- To insert comments, use shell style "#".

- `init` is the only mode supported for user-defined function arguments.

- `permanent` is the only mode supported for stub function output.

- Function names may be C or C++ functions with blanks or commas. For example, `f(int, int)`.

- Function names can be specified in the short form ("f") as long as no ambiguity exists.

- The function returns either an integral (including enum and bool) or floating point type. If the function returns an integral type and you specify the range as a floating point [`v0.x, v1.y`], the software applies the integral interval [`(int)v0-1, (int)v1+1`].

Example DRS Text File

In the following example, the global variables are named `x`, `y`, `z`, `w`, and `v`.

```
x  12  100    init
```

```
y   0   10000  permanent
z   0   1      globalassert
w   min max    permanent
v   0   max    globalassert
arrayOfInt  -10 20  init
s1.id       0   max init
array.c2    min 1   init
car.speed   0   350 permanent
bar.return  -100 100 permanent

# x is defined between [12;100] at initialization
# y is permanently defined between [0,10000] even any assignment
# z is checked in the range [0;1] after each assignment
# w is volatile and full range on its declaration type
# v is positive and checked after each assignment.
# All cells arrayOfInt are defined between [-10;20] at initializsation
# s1.id is defined between [0;2^31-1] at initialisation.
# All cells array[i].c2 are defined between [-2^31;1] at initialization
# Speed of Struct car is permanently defined between 0 and 350 Km/h
# function bar returns -100..100
```

Specifying Data Ranges

You activate the DRS feature using the option **Variable range setup**
(-data-range-specification).

To use the DRS feature:

1 Create a DRS file containing the list global variables (or functions) and
their associated data ranges, as described in "File Format" on page 5-34.

2 In the Analysis options section of the Launcher window, select **PolySpace
inner settings > Stubbing**.

3 In the **Variable range setup parameter**, select the DRS file that you
want to use.

File Format

The DRS file contains a list of global variables and associated data ranges. The point during verification at which the range is applied to a variable is controlled by the mode keyword: `init`, `permanent`, or `globalassert`.

The DRS file must have the following format:

```
variable_name min_value max_value <init|permanent|globalassert>
function_name.return min_value max_value permanent
```

variable_name val_min val_max <init|permanent|globalassert>

- *variable_name* — The name of the global variable.

- *min_value* — The minimum value for the variable.

- *min_value* and *max_value* — The minimum and maximum values for the variable. You can use the keywords "`min`" and "`max`" to denote the minimum and maximum values of the variable type. For example, for the type long, min and max correspond to -2^31 and 2^31-1 respectively.

- `init` — The variable is assigned to the specified range only at initialization, and keeps it until first write.

- `permanent` — The variable is permanently assigned to the specified range. If the variable is assigned outside this range during the program, no warning is provided. Use the `globalassert` mode if you need a warning.

- `globalassert` — After each assignment, an assert check is performed, controlling the specified range. The assert check is also performed at global initialization.

- *function_name* — The name of the stub function.

Tips

- You can use the keywords "`min`" and "`max`" to denote the minimum and maximum values of the variable type. For example, for the type long, min and max correspond to -2^31 and 2^31-1 respectively.

- You can use hexadecimal values. For example, x 0x12 0x100 init.

- Supported column separators are tab, comma, space, or semi-column.

- To insert comments, use shell style "#".

- Functions must be stubbed functions (no provided body).

- `permanent` is the only supported mode for functions.

- Function names may be C or C++ functions with blanks or commas. For example, `f(int, int)`.

- Function names can be specified in the short form ("f") as long as no ambiguity exists.

- The function returns either an integral (including enum and bool) or floating point type. If the function returns an integral type and you specify the range as a floating point [v0.x, v1.y], the software applies the integral interval [(int)v0-1, (int)v1+1].

Example

In the following example, the global variables are named x, y, z, w, array, and v.

```
x 12 100 init       # x is defined between [12;100] at \
          initialisation
y 0 10000 permanent   # y is permanently defined between \
          [0,10000] even any possible assignment.
z 0 1 globalassert    # z is checked in the range [0;1] after \
          each assignment
w min max permanent   # w is volatile and full range on its \
          declaration type
v 0 max globalassert   # v is positive and checked after each \
          assignment.
arrayOfInt -10 20 init  # All cells are defined between [-10;20] \
          at initialisation
s1.id 0 max init     # s1.id is defined between [0;2^31-1] at \
          initialisation.
array.c2 min 1 init    # All cells array[i].c2 are defined \
          between [-2^31;1] at initialisation
car.speed 0 350 permanent # Speed of Struct car is permanently \
            defined between 0 and 350 Km/h
bar.return -100 100 permanent # function bar returns -100..100
```

Variable Scope

DRS supports variables with external linkages, const variables, and defined variables. In addition, extern variables are supported with the option `-allow-undef-variables`.

The following table summarizes possible uses:

	init	**permanent**	**globalassert**	**comments**
Integer	Ok	Ok	Ok	`char`, `short`, `int`, `enum`, `long` and `long long` If you define a range in floating point form, rounding is applied.
Real	Ok	Ok	Ok	`float`, `double` and `long double` If you define a range in floating point form, rounding is applied.
Volatile	No effect	Ok	Full range	Only for `int` and `real`
Structure field	Ok	Ok	Ok	Only for `int` and `real` fields, including arrays or structures of `int` or `real` fields (see below)

	init	permanent	globalassert	comments
Structure field in array	Ok	No effect	No effect	Only when leaves are int or real. Moreover the syntax is the following: `<array_name>.<field_name>`
Array	Ok	Ok	Ok	Only for int and real fields, including structures or arrays of integer or real fields (see below)
Pointer	No effect	No effect	No effect	
Union field	Ok	No effect	Ok	See "DRS Support for Union Members" on page 5-38.
Complete structure	No effect	No effect	No effect	
Array cell	No effect	No effect	No effect	Example: array[0], array[10] …
User-defined function arguments	Ok	No effect	No effect	Main generator calls the function with arguments in the specified range
Stubbed function return	No effect	Ok	No effect	Stubbed function returning integer or floating point

Every variable (or function) and associated data range will be written in the log file during the compile phase of PolySpace verification. If PolySpace software does not support the variable, a warning message is displayed.

Note If you use DRS to set a data range on a const global variable that is used in another variable declaration (for example as an array size), the variable that uses the global variable you ranged is not ranged itself.

DRS Support for Structures

DRS can initialize arrays of structures, structures of arrays, etc., as the long as the last field is explicit (structures of arrays of integers, for example).

However, DRS cannot initialize a structure itself — you can only initialize the fields. For example, `"s.x 20 40 init"` is valid, but `"s 20 40 init"` is not (because PolySpace cannot determine what fields to initialize).

DRS Support for Union Members

In init mode, the software applies the last range in DRS to the union members at the given offset.

In globalassert mode, the software checks every globalassert in DRS for a given offset within the union at every assignment to the union variable at that offset.

For example:

```
union position {
 int   sunroof;
  int   window;
  int   locks;
} positionData;
```

DRS:

```
positionData.sunroof 0 100 globalassert
positionData.window -100 0 globalassert
```

```
positionData.locks  -1  1 globalassert
```

An assignment to `positionData.locks` (or other members) will perform assertion checking on the ranges 0 to 100, -100 to 0, and -1 to 1.

Performing Efficient Module Testing with DRS

DRS allows you to perform efficient static testing of modules. This is accomplished by adding design level information missing in the source-code.

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

- Input data are consumed
- Output data are produced
- Constant calibrations are used during black box execution influencing intermediate results and output data.

Using the DRS feature, you can define:

- The nominal range for input data
- The expected range for output data
- The generic specified range for calibrations

These definitions then allow PolySpace software to perform a single static verification that performs two simultaneous tasks:

- answering questions about robustness and reliability
- checking that the outputs are within the expected range, which is a result of applying black-box tests to a module

In this context, you assign DRS keywords according to the type of data (inputs, outputs, or calibrations).

Type of Data	DRS Mode	Effect on Results	Why?	Oranges	Selectivity
Inputs (entries)	permanent	**Reduces** the number of oranges, (compared with a standard PolySpace verification)	Input data that were full range are set to a smaller range.	↓	↑
Outputs	globalassert	**Increases** the number of oranges, (compared with a standard PolySpace verification)	More verification is introduced into the code, resulting in both more orange checks and more green checks.	↑	→
Calibration	init	**Increases** the number of oranges, (compared with a standard PolySpace verification)	Data that were constant are set to a wider range.	↑	↓

Reducing Oranges with DRS

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

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

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

This not only eliminates the local overflow orange, but also results in more accuracy in the data. This accuracy is then propagated through the rest of the code.

Using DRS removes the oranges located in the red circle below.
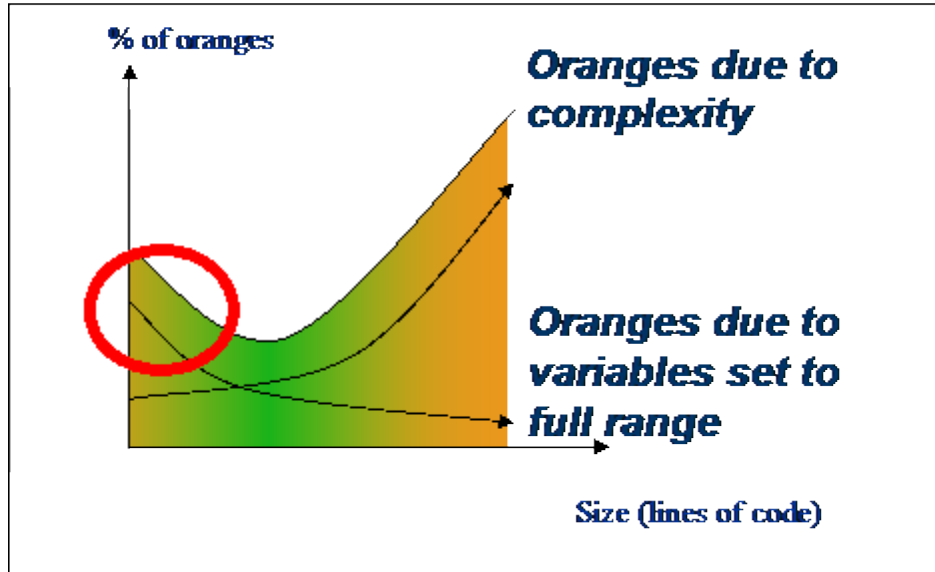
Why Is DRS Most Effective on Module Testing?
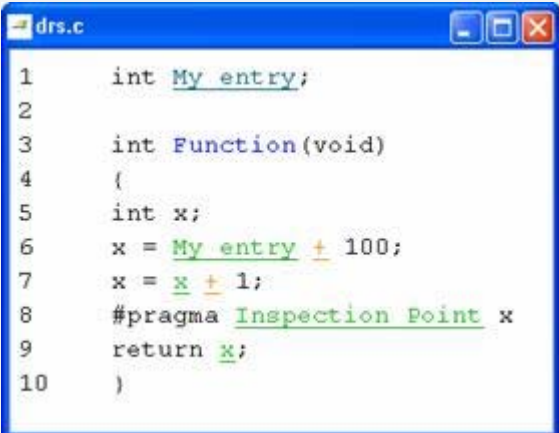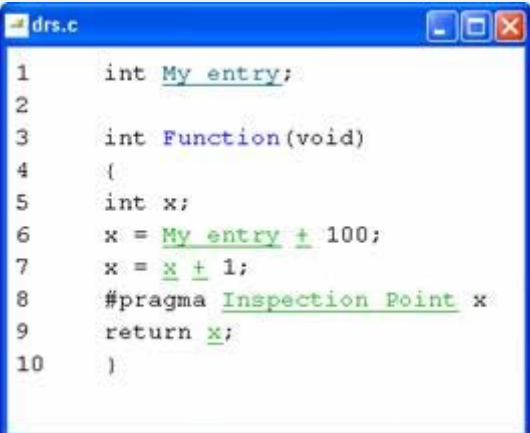
Removing oranges caused by full-range (worst-case) data can drastically reduce the total number of orange checks, especially when used on verifications of small files or modules. However, the number of orange checks caused by code complexity is not effected by DRS. For more information on oranges caused by code complexity, see "Considering the Effects of Application Code Size" on page 8-37, and "An Ideal Application Size" on page 8-36.
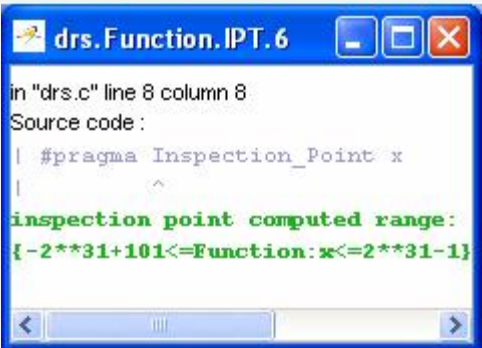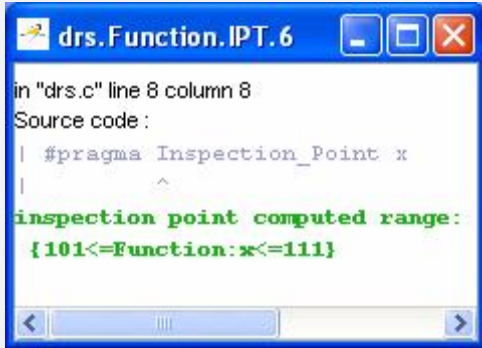
This section describes how DRS reduces oranges on files or modules only.

Example

The following example illustrates how DRS can reduce oranges. Suppose that in the real world, the input "My_entry" can vary between 0 and 10.

PolySpace verification produces the following results: one with DRS and one without.

Without DRS	With DRS — 2 Oranges Removed + Return Statement More Accurate
• With "*My_entry*" being full range, the addition "**+**" is orange, • the result "x" is equal to all values between [min+100 max] • Due to previous computations, x+1 can here overflow too, making the addition "**+**"orange.	• With "*My_entry*" being bounded to [0,10], the addition "+" is green • the result "x" is equal to [100,110] • Due to previous computations, x+1 can NOT overflow here, making the addition "+" green again.

Without DRS	With DRS — 2 Oranges Removed + Return Statement More Accurate
And the returned result is between [min+101 max]	And the returned result is between [101,111]

drs.Function.IPT.6 ⬛ ⬜ ❌ in "drs.c" line 8 column 8 Source code : | #pragma Inspection_Point x | ^ inspection point computed range: {-2**31+101<=Function:x<=2**31-1} ◀ ▬▬▬ ▶	**drs.Function.IPT.6** ⬛ ⬜ ❌ in "drs.c" line 8 column 8 Source code : | #pragma Inspection_Point x | ^ inspection point computed range: {101<=Function:x<=111} ◀ ▬▬▬ ▶

6

Preparing Source Code for Verification

- "Stubbing" on page 6-2
- "Preparing Code for Variables" on page 6-15
- "Preparing Code for Built-in Functions " on page 6-19
- "Types Promotion" on page 6-21
- "Highlighting Known Coding Rule Violations and Run-Time Errors" on page 6-24

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.

By default, PolySpace software automatically stubs functions. 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.

For Example:

```
void main(void)
{
 a=1;
 b=0;
 a_missing_function(&a, b);
 b = 1 / a;
}
```

Due to the reliance on the software's default stub, the division is shown with an orange warning because a is assumed to be anywhere in the full permissible integer range (including 0). If the function was commented out, then the division would be a green "/ ". A red "/ " could only be achieved with a manual stub.

Deciding which Stub Functions to Provide

In the following section, *procedure_to_stub* can represent either procedure or a sequence of assembly instructions which would be automatically stubbed in the absence of a manual stub.

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 (void *procedure*(void)). PolySpace needs no concept of timing since it takes into account all possible scheduling and interleaving of concurrent execution. There is therefore no need to stub functions that set or reset a timer. Simply declare the variable representing time as volatile.

- An I/O access: maybe to a hardware port, a sensor, a read/write of a file, a read of an EEPROM, or a write to a volatile variable. There is no need to stub a write access. If you wish to do so, simply stub a write access to an empty action (void *procedure*(void)). Stub read accesses to "read all possible values (volatile)".

- A write to a global variable. In this case, you may need to consider which procedures or functions write to it and why. Do not stub the concerned *procedure_to_stub* if:

  - The variable is volatile;

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

  - The 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. This is easily achieved by copying the value into a local variable.

In general, follow the Data Flow and remember that:

- PolySpace only cares about the C code which is provided;

- PolySpace need not be informed of timing constraints because all possible sequencing is taken into account;

- You can refer to execution hypotheses made by PolySpace for a complete list of constraints.

Example
The following example shows a header for a missing function (which might occur, for example, if the code is a subset of a project.) The missing function

copies the value of the src parameter to dest so there would be a division by
zero - a runtime error - at run time.

```
void main(void)
{
 a = 1;
 b = 0;
 a_missing_function(&a, b);
 b = 1 / a;
}
```

Due to reliance on the PolySpace 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 a green "**/** ". A red "**/** " could only be achieved with a manual
stub.

Default Stubbing	Manual Stubbing	Function Ignored
```void main(void)``` ``` { ``` ``` a = 1; ``` ``` b = 0; ``` ``` a_missing_function(&a, b); ``` ``` b = 1 / a; ``` ``` // orange division ``` ``` } ```	```void a_missing_function``` ```(int *x, int y;)``` ```{ *x = y; }``` ```void main(void)``` ```{``` ``` a = 1;``` ``` b = 0;``` ``` a_missing_function(&a, b);``` ``` b = 1 / a;``` ``` // red division```	```void a_missing_function``` ```(int *x, int y;)``` ```{ }``` ```void main(void)``` ```{``` ``` a = 1;``` ``` b = 0;``` ``` a_missing_function(&a, b);``` ``` b = 1 / a;``` ``` // green division```

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

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

Stub automatically in the knowledge that no runtime error will be ever introduced by automatic stubbing; to minimize preparation time.

Stubbing Examples

Example: Specification
The following examples consider the pros and cons of manual and automatic stubbing.

Here is the first example:

```
typedef struct _c {
int cnx_id;
int port;
int data;
} T_connection ;

int Lib_connection_create(T_connection *in_cnx) ;
int Lib_connection_open (T_connection *in_cnx) ;
```

File: connection_lib		Function: Lib_connection_create
param in	None	
param in/out	in_cnx	all fields might be changed in case of a success
returns	int	0 : failure of connection establishment
		1 : success

Note   Default stubbing is suitable here.

Here are the reasons why:

- The content of the *in_cnx* structure might be changed by this function.

- The possible return values of 0 or 1 compared to the full range of an integer wont have much impact on the Run-Time Error aspect. It is unlikely that the results of this operation will be

File: connection_lib	Function: Lib_connection_create

used to compute some mathematical algorithm. It is probably a Boolean status flag and if so is likely to be stored and compared to 0 or 1. The default stub would therefore have no detrimental effect.

File: connection_lib		Function: Lib_connection_open
param in	T_connection *in_cnx	in_cnx->cnx_id is the only parameter used to open the connection, and is a read-only parameter.
		cnx_id, port and data remain unchanged
param in/out	None	
returns	int	0 : failure of connection establishment
		1 : success

Note Default stubbing works here but manual stubbing would give more benefit.

Here are the reasons why:

- For the return value, default stubbing would be applicable as explained in the previous example.

- Since the structure is a read-only parameter, it will be worth stubbing it manually to accurately reflect the behavior of the missing code. Benefits: PolySpace will find more red and gray code

> **Note** Even in the examples above, it concerns some C code like; stubs of functions members in classes follow same behavior.

Colored Source Code Example

```
1     typedef struct _c {
2     int a;
```

```
3       int b;
4       } T;
5
6       void send_message(T *);
7       void main(void)
8       {
9       int i;
10      T x = {10, 20};
11      send_message(&x);
12      i = x.b /x.a;  // orange with the default stubbing
13      }
```

Suppose that it is known that send_message does not write into its argument.
The division by x.a will be orange if default stubbing is used, warning of
a potential division by zero. A manual stub which accurately reflects the
behavior of the missing code will result in a green division instead, thus
increasing the selectivity.

Manual stubbing examples for send_message:

```
void send_message(T *) {}
```

In this case, an empty function would be a sound manual stub.

Specifying Call Sequence

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

Colored Source Code Example

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

```
static char x;
static int y;
```

```
void f(void)
{
y = 300;
}

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

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

Example of Call Sequence

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

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

Constraining Data with Stubbing

- "Default Behavior of Global Data" on page 6-9

- "Constraining the Data" on page 6-10

- "Applying the Technique" on page 6-10

- "Integer Example" on page 6-11

Default Behavior of Global Data

Initially, consider how PolySpace handles the verification of global variables.

There is a maximum range of values which may be assigned to each variable as defined by its type. By default, PolySpace assigns that full range for each

global variable, ensuring that a meaningful verification of such a variable can take place even when the functions that write to it are not included. If a range of values was not considered in these circumstances, such a variable would be assumed to have a value of zero throughout.

This default launching mode is often adequate, but it is sometimes useful to specify that the range of values which may be assigned to some variables is to be limited to what is appropriate on a functional level. These ranges will be propagated to the whole call tree, and hence will limit the number of "impossible values" which are considered throughout the verification.

This thinking does not just apply to global variables; it is equally appropriate where such a variable is passed as a parameter to a function, or where return values from stubbed functions are under consideration.

To some extent, the effectiveness of this technique is limited by compromises made by PolySpace to deal with issues of code complexity. For instance, it cannot be assumed that all of these ranges will be propagated throughout all function calls. Sometimes, perhaps as a result of complex function interactions or constructions where PolySpace is known to be imprecise, the potential value of a variable will assume its full "type" range despite this technique having been applied.

Constraining the Data

PolySpace experience is that restricting such as global variables to a functional range is a useful technique. However, it is not always fruitful and it is therefore recommended only where its application is not too labour intensive - that is, where its implementation can be automated.

The technique therefore requires

- A knowledge of the variables and the maximum ranges they may take in practice.

- A data dictionary in electronic format from which the variable names and their minimum and maximum values can be extracted.

Applying the Technique

To apply the technique:

1 Create the range setting stubs:

   a create 6 functions for each type (8,16 or 32 bits, signed and unsigned)

   b declare 6 global volatile variables for each type

   c write the functions which returns sub-ranges (an example follows)

2 Gather the initialization of all relevant variables into a single procedure

3 Call this procedure at the beginning of the main. This should replace any existing initialization code.

Integer Example

```
volatile int tmp;

int polyspace_return_range(int min_value, int max_value)
{
int ret_value;

ret_value = tmp;
assert (ret_value>=min_value && ret_value<=max_value);

return ret_value;
}
void init_all(void)
{
x1 = polyspace_return_range(1,10);
x2 = polyspace_return_range(0,100);
x3 = polyspace_return_range(-10,10);
}

void main(void)
{
init_all();

while(1)
 {
 if (tmp) function1();
 if (tmp) function2();
```

```
  // ...
  }
}
```

Recoding Specific Functions

Once data ranges have been specified (above), it may be beneficial to recode some functions in support of them.

Sometimes, perhaps as a result of complex function interactions or constructions where PolySpace is known to be imprecise, the potential value of a variable will assume its full "type" range data ranges having been restricted. Recoding those complex functions will address this issue.

Identify in the modules:

- API which read global variables through pointers

  Replace this API:

  ```
  typedef struct _points {
  int x,y,nb;
  char *p;
  }T;

  #define MAX_Calibration_Constant_1 7
  char Calibration_Constant_1[MAX_Calibration_Constant_1] =    \
  { 1, 50, 75, 87, 95, 97, 100} ;
  T Constant_1 = { 0, 0,
    MAX_Calibration_Constant_1,
    &Calibration_Constant_1[0] } ;

  int read_calibration(T * in, int index)
  {
  if ((index <= in->nb) && (index >=0)) return in->p[index];
  }

  void interpolation(int i)
  {
  int a,b;
  ```

```
 a= read_calibration(&Constant_1,i);
 }
```

With this one:

```
char Constant_1 ;

#define read_calibration(in,index) *in

void main(void)
{
Constant_1 = polyspace_return_range(1, 100);
}

void interpolation(int i)
{
int a,b;

a= read_calibration(&Constant_1,i);
}
```

- Points in the source code which expand the data range perceived by PolySpace

- Functions responsible for full range data, as shown by the VOA (Value on assignment) check.

  if direct access to data is responsible, define the functions as macros.

  ```
  #define read_from_data(param) read_from_data##param

  int read_from_data_my_global1(void)
  { return [a functional range for my_global1]; }

  Char read_from_data_my_global2(void)
  { }
  ```

- stub complicated algorithms, calibration read accesses and API functions reading global data - as usual. For instance, if an algorithm is iterative - stub it.

- variables

- where the data range held by each element of an array is the same, replace that array with a single variable.

- where the data range held by each element of an array differs, separate it into discrete variables.

Preparing Code for Variables

In this section...
"How are Variables Initialized" on page 6-15
"Data and Coding Rules" on page 6-16
"Variables: Declaration and Definition" on page 6-16
"How Can I Model Variable Values External to My Application?" on page 6-17

How are Variables Initialized

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

Extern

PolySpace works on the principle that a global or static extern variable could take any value within the range of its type.

```
extern int x;
int y;
y = 1 / x; // orange because x ~ [-2^31, 2^31-1]
y = 1 / x; // green because x ~ [-2^31 -1] U [1, 2^31-1]
```

Refer to Chapter 9, "Reviewing Verification Results" for more information on color propagation.

For extern structures containing field(s) of type "pointer to function", this principle leads to red errors in the viewer. In this case, the resulting default behavior is that these pointers don't point to any valid function. For results to be meaningful here, you may well need to define these variables explicitly.

Volatile

```
volatile int x; // x ~ [-2^31, 2^31-1], although x has not been
initialized
```

- If x is a global variable, the NIV is green

- If x is a local variable, the NIV is always orange

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

```
#define X (* ((int *)0x20000))
```

- X = 100;
- y = 1 / X; // NIV on X is orange
- int *p = (int *)0x20000;
- *p = 100;
- y = 1 / *p ; // NIV on *p is orange

Data and Coding Rules
Data rules are design rules which dictate how modules and/or files interact with each other.

For instance, consider global variables. It is not always apparent which global variables are produced by a given file, or which global variables are used by that file. The excessive use of global variables can lead to resulting problems in a design, such as

- File APIs (or function accessible from outside the file) with no procedure parameters;
- The requirement for a formal list of variables which are produced and used, as well as the theoretical ranges they can take as input and/or output values.

Variables: Declaration and Definition
The definition and declaration of a variable are two discrete but related operations which are frequently confused.

Declaration
A declaration provides information about the type of the function or variable.

- for a function, the prototype: `int f(void);`

- for an external variable: `extern int x;`

If the function or variable is used in a file where it has not been declared, a compilation error will result.

Definition

A definition provides:

- for a function, the body of the function has been written: `int f(void) { return 0; }`

- for a variable, a part of memory has been reserved for the variable: `int x;` or `extern int x=0;`

When a variable is not defined, the `-allow-undef-variable` is required to start the verification. Where that option is used, PolySpace will consider the variable to be initialized, and to potentially take any value in its full range (see "How are Variables Initialized" on page 6-15 section).

When a function is not defined, it is stubbed automatically.

How Can I Model Variable Values External to My Application?

There are three main considerations.

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

A volatile variable can be defined as a variable which does not respect following axiom:

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

Thus the value of a volatile variable is "unknown". It can be any value that can be represented by a variable of its type, and that value can change at any time - even between 2 successive memory accesses.

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

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

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

Preparing Code for Built-in Functions

In this section...
"Overview " on page 6-19
"Stubs of stl Functions" on page 6-19
"Stubs of libc Functions" on page 6-19

Overview

PolySpace stubs all functions which are not defined within the verification. PolySpace provides for all the functions defined in the stl, in the standard libc, an accurate stub taking into account functional aspect of the function.

Stubs of stl Functions

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

Note All allocation functions found in the code to analyze like new, new[], delete and delete[] are replaced by internal and optimized stubs of new and delete. A warning is given in the log file when such replace occurs.

Stubs of libc Functions

Concerning the libc, all these functions are declared in the standard list of headers and can be redefined using its own definition by invalidating the associated set of functions:

• Using `D POLYSPACE_NO_STANDARD_STUBS` for all functions declared in Standard ANSI headers: assert.h, ctype.h, errno.h, locale.h, math.h, setjmp.h ('setjmp' and 'longjmp' functions are partially implemented – see <polyspaceProduct>/cinclude/__polyspace__stdstubs.c), signal.h ('signal' and 'raise' functions are partially implemented – see <polyspaceProduct>/cinclude/__polyspace__stdstubs.c), stdio.h, stdarg.h, stdlib.h, string.h, and time.h.

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

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

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

Types Promotion

Unsigned Types Promoted to Signed

It is important to understand the circumstances under which signed integers are promoted to unsigned.

For example, the execution of the following piece of code would produce an assertion failure and a core dump.

```
#include <assert.h>
int main(void) {
 int x = -2;
 unsigned int y = 5;
 assert(x <= y);
}
```

Consider the range of possible values (interval) of x in this second example. Again, this code would cause assertion failure:

```
volatile int random;
unsigned int y = 7;
int x = random;
assert ( x >= -7 && x <= y );
```

However, given that the interval range of x after the second assertion is **not [-7 .. 7]**, but rather **[0 .. 7]**, the following assertion would hold true.

```
assert (x>=0 && x<=7);
```

Implicit promotion explains this behavior.

In fact, in the second example x <= y is implicitly:

```
((unsigned int) x) <= y /* implicit promotion because y is unsigned */
```

A negative cast into unsigned gives a big value, which has to be bigger that 7. And this big value can never be <= 7, and so the assertion can never hold true.

Promotion Rules in Operators

Knowledge of the rules applying to the standard operators of the C language will help you to analyze those orange and **red** checks which relate to overflows on type operations. Those rules are:

- Unary operators operate on the type of the operand;

- Shifts operate on the type of the left operand;

- Boolean operators operate on Booleans;

- Other binary operators operate on a common type. If the types of the 2 operands are different, they are promoted to the first common type which can represent both of them.

So, be careful of constant types, and also when analyzing any operation between variables of different types without an explicit cast.

Consider the integral promotion aspect of the ANSI standard. On arithmetic operators like +, -, *, %and / , an integral promotion is applied on both operands. From the PolySpace point of view, that can imply an OVFL or a UNFL orange check.

Example

```
2 extern char random_char(void);
3 extern int random_int(void);
4
5 void main(void)
6 {
7  char c1 = random_char();
8  char c2 = random_char();
9  int i1 = random_int();
10  int i2 = random_int();
11
12  i1 = i1 + i2;   // A typical OVFL/UNFL on a + operator
13  c1 = c1 + c2;   // An OVFL/UNFL warning on the c1 assignment
[from int32 to int8]
```

```
14 }
```

Unlike the addition of two integers at line 12, an implicit promotion is used in the addition of the two chars at line 13. Consider this second "equivalence" example.

```
2 extern char random_char(void);
3
4 void main(void)
5 {
6  char c1 = random_char();
7  char c2 = random_char();
8
9  c1 = (char)((int)c1 + (int)c2);  // Warning OVFL: due to
integral promotion
10 }
```

An orange check represents a warning of a potential overflow (OVFL), generated on the (char) cast [from int32 to int8]. A green check represents a verification that there is no possibility of any overflow (OVFL) on the +operator.

In general, integral promotion requires that the abstract machine should promote the type of each variable to the integral target size before realizing the arithmetic operation and subsequently adjusting the assignment type. See the equivalence example of a simple addition of two *char* (above).

Integral promotion respects the size hierarchy of basic types:

- *char (signed or not)* and *signed short* are promoted to *int*.

- *unsigned short* is promoted to *int* only if *int* can represent all the possible values of an *unsigned short*. If that is not the case (perhaps because of a 16-bit target, for example) then *unsigned short* is promoted to *unsigned int*.

- Other types like *(un)signed int*, *(un)signed long int* and *(un)signed long long int* promote themselves.

Highlighting Known Coding Rule Violations and Run-Time Errors

In this section...
"Annotating Code to Indicate Known Coding Rule Violations" on page 6-24
"Annotating Code to Indicate Known Run-Time Errors" on page 6-26

Annotating Code to Indicate Known Coding Rule Violations

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

- Hide known coding rule violations while analyzing new coding rule violations.

- Inform other users of known coding rule violations.

To annotate your code before running a verification:

1 Open your source file using a text editor.

2 Locate the code that violates a coding rule.

3 Insert the required comment.

```
#include <cassert>
#include <typeinfo>
// polyspace<JSF: 9> This is a known rule violation
#include "sanalogic.h"
#include "zz_utils.h"
```

See also "Syntax for Coding Rule Violations" on page 6-25.

4 Save your file.

5 Start the verification. If your comments do not conform to the prescribed syntax, the software produces a warning and the comments do not appear in the Launcher.

When the verification is complete, or stops because of a compilation error, you can view all coding rule violations in the Launcher by clicking **MISRA C++** or **JSF C++**.

In the **Acronym**, **User Acronym** and **Justification** columns, the information that you provide within your code comments is now visible. In addition, the **Justified** cell contains Yes.

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

Syntax for Coding Rule Violations
To apply comments to a single line of code, use the following syntax:

```
// polyspace<Cpp_Rule_Set: Rule1[,Rule2[,…]] [:  Predefined
Acronym [:  User-defined Acronym]] > [Additional text]
```

where

- Square brackets *[ ]* indicate optional information.

- *Cpp_Rule_Set* is, depending on your rule checker, either MISRA-CPP or JSF.

- *Rule1, Rule2,* … are rules in your rules file (for example, misra-rules.msr). You can also specify ALL, which covers every coding rule.

- *Predefined Acronym*, for example, MIN and DEF, allows you to categorize the coding rule violation with a PolySpace acronym. To see the complete list of PolySpace acronyms, in the Preferences PolySpace Viewer dialog box, click the **Acronyms** tab.

- *User-defined Acronym* allows you to categorize the coding rule violation with an acronym that you define in the Preferences PolySpace View dialog box, on the **Acronyms** tab. You can specify *User-defined Acronym* only if you specify *Predefined Acronym*.

- *[Additional text]* appears in the **Justification** column of the **MISRA CPP** or **JSF** view of the Launcher. Use this text to provide information about the coding rule violations.

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

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

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

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

```
// polyspace:begin<Cpp_Rule_Set: Rule1[,Rule2[,…]] [:  Predefined
Acronym [:  User-defined Acronym]] > [Additional text]

...  Code section ...

// polyspace:end<Cpp_Rule_Set: Rule1[,Rule2[,…]] [:  Predefined
Acronym [:  User-defined Acronym]] > [Additional text]
```

Annotating Code to Indicate Known Run-Time Errors

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

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

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

To 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 (u.random_int())
    // polyspace<RTE: CPP : OTH : EGC> This is a known run-time error
    return (typeid(this) ==  typeid (*ptr));
```

See also "Syntax for Run-Time Errors" on page 6-28.

4 Save your file.

5 Start the verification. If your comments do not conform to the prescribed syntax, the software produces a warning and the comments do not appear in the Viewer.

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

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

Procedural entities	!	✗	?	✓	Line	Col	@	Details	Reviewed	Acronym	User Acronym	Comment
Demo_Cpp	2	10	34	181			85		☐			
exception.stdh					1		0	exception.stdh	☐			
main.cpp	1	7	18	85	1		84	main.cpp	☐			
new.stdh					1		0	new.stdh	☐			
sanalogic.cpp	1	1		15	1		100	sanalogic.cpp	☐			
SAnalogic::TypeInfo()	1	1		15	12	15	100	sanalogic.cpp	☐			
EXC.0				✓	1	12		function does not throw	☐			
FRV.1				✓	1	12		function returns a value	☐			
EXC.2				✓	1	14	12	call to SAnalogic does not throw	☐			
EXC.3				✓	1	15	18	call to SAnalogic does not throw	☐			
ASRT.4				✓	1	17	2	user assertion is verified	☐			
ASRT.5				✓	1	18	2	user assertion is verified	☐			
EXC.6				✓	1	20	9	call to Sensor does not throw	☐			
NNT.7				✓	1	23	18	this-pointer of random_int is not null	☐			
EXC.8				✓	1	23	18	call to random_int does not throw	☐			
CPP.9	!				1	25	29	Error : incorrect typeid argument	☑	OTH	EGC	This is a known run-time error
OOP.10		✗			1	25	29	Unreachable check : this-pointer type is correct	☐			
NIP.11				✓	1	25	38	Pointer is initialized	☐			

Syntax for Run-Time Errors

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

```
// polyspace<RTE: RunTimeError1[,RunTimeError2[,…]] [:
Predefined Acronym [: User-defined Acronym]]> [Additional text]
```

where

- Square brackets *[ ]* indicate optional information.

- *RunTimeError1, RunTimeError2, …* are formal PolySpace checks, for example, OBAI, IDP, and ZDV. You can also specify ALL, which covers every check.

- *Predefined Acronym*, for example, MIN and DEF, allows you to categorize the run-time error with a PolySpace acronym. To see the complete list of PolySpace acronyms, in the Preferences PolySpace Viewer dialog box, click the **Acronyms** tab.

- *User-defined Acronym* allows you to categorize the run-time error with an acronym that you define in the Preferences PolySpace View dialog box, on the **Acronyms** tab. You can specify *User-defined Acronym* only if you specify *Predefined Acronym*.

- *[Additional text]* appears in the **Comment** column of the procedural entities view of the Viewer. Use this text to provide information about the run-time errors.

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

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

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

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

```
// polyspace:begin<RTE: RunTimeError1[,RunTimeError2[,…]] [:
Predefined Acronym [:  User-defined Acronym]] > [Additional text]

...  Code section ...

// polyspace:end<RTE: RunTimeError1[,RunTimeError2[,…]] [:
Predefined Acronym [:  User-defined Acronym]] > [Additional text]
```

Running a Verification

- "Types of Verification" on page 7-2
- "Running Verifications on PolySpace Server" on page 7-3
- "Running Verifications on PolySpace Client" on page 7-22
- "Running Verifications from Command Line" on page 7-27

Types of Verification

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

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

Running Verifications on PolySpace Server

In this section...

Starting Server Verification

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

To start a verification that runs on a server:

1 Open the Launcher.

2 Open the project containing the files you want to verify. For more information, see Chapter 4, "Setting Up a Verification Project".

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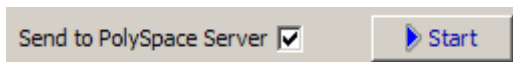
> **Note** If you select **Set this option to use the server mode by default in every new project** in the Remote Launcher pane of the preferences, the **Send to PolySpace Server** check box is selected by default when you create a new project.

4 Click **Start**.

The verification starts. For information on the verification process, see "What Happens When You Run Verification" on page 7-4.

> **Note** If you see the message `Verification process failed`, click **OK** and go to "Verification Process Failed Errors" on page 8-2.

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

6 For information on downloading and viewing your results, see "Opening Verification Results" on page 9-8.

What Happens When You Run Verification

The verification has three main phases:

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

2 Generating a main if it does not find a main and the **Generate a Main** option is selected. For more information about generating a main, see "Generate a Main Using a Given Class" in the *PolySpace Products for C++ Reference*.

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

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

- A message dialog box tells you that the verification completed. This message means that the part of the verification that takes place on the client is complete. The rest of the verification runs on the server.

- A message in the log area tells you that the verification was transferred to the server and gives you the identification number (Analysis ID) for the verification. For the following verification, the identification number is 1.
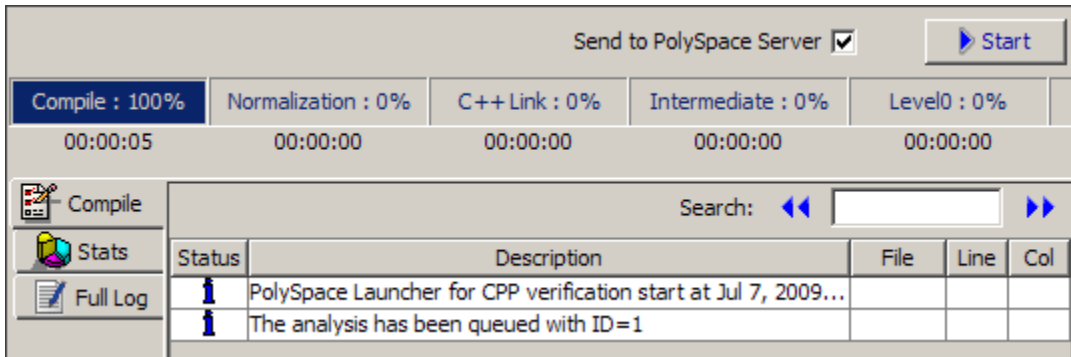
Running Verification Unit-by-Unit

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

To run a unit-by-unit verification:

1 In the Launcher, ensure that the **Send to PolySpace Server** check box is selected.
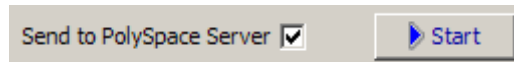
2 In the Analysis options, expand **PolySpace inner settings**.

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

⊟ PolySpace inner settings				
⊟ Run a verification unit by unit		☑		-unit-by-unit
Unit by unit common source		C:\PolySpace\cpp_	...	-unit-by-unit-common-source

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

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

The Unit by unit common source dialog box opens.
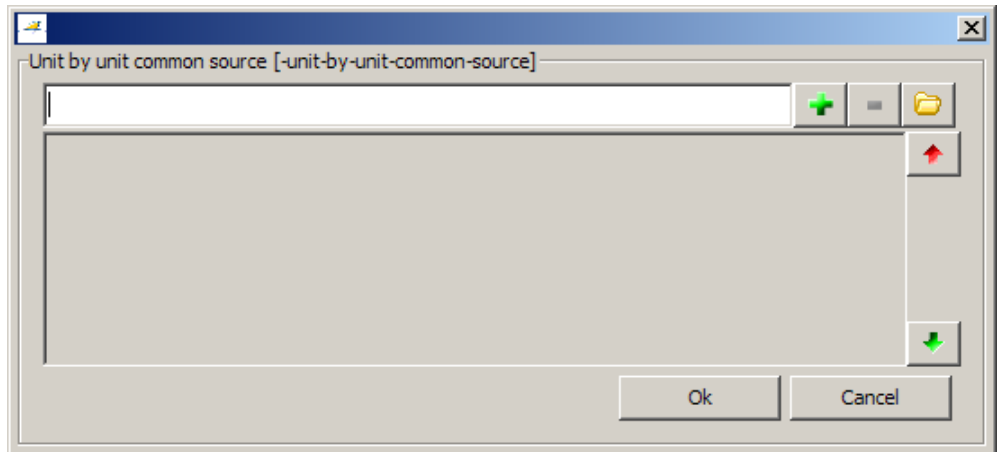
6 Click the folder icon 🗁 .

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

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

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

8 Click **Ok**.

9 Click **Start**.

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

Managing Verification Jobs Using the PolySpace Queue Manager

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

To manage verification jobs on the PolySpace Server:

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

The **PolySpace Queue Manager Interface** opens.

ID	Author	Application	Results folder	CPU	Status	Date	Language
1	Polyspace	Demo_Cpp	C:\PolySpace\PolySpaceForCandCPP_R...	runstroms	completed	04-Sep-2009, 16:32:23	CPP
4	PolySpace	Demo_C	C:\PolySpace\PolySpaceForCandCPP_R...	runstroms	completed	14-Dec-2009, 15:29:08	C
5	polyspace	Demo_C_Singl...	C:\PolySpace\PolySpaceForCandCPP_R...	runstroms	running	14-Dec-2009, 15:33:38	C
6	username	Example_Project	C:\PolySpace\polyspace_project\results		queued	14-Dec-2009, 15:34:41	C

Connected to Queue Manager localhost   User mode

2 Right-click any job in the queue to open the context menu for that verification.
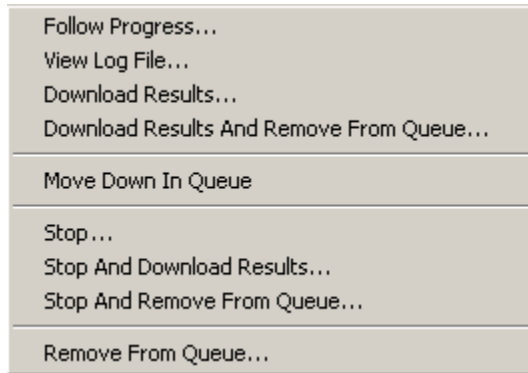
3 Select the appropriate option from the context menu.

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

Monitoring Progress of Server Verification

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

To view a log file on the server:

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

The **PolySpace Queue Manager Interface** opens.

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

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

A Launcher window labeled **PolySpace follow remote analysis progress for CPP** appears.
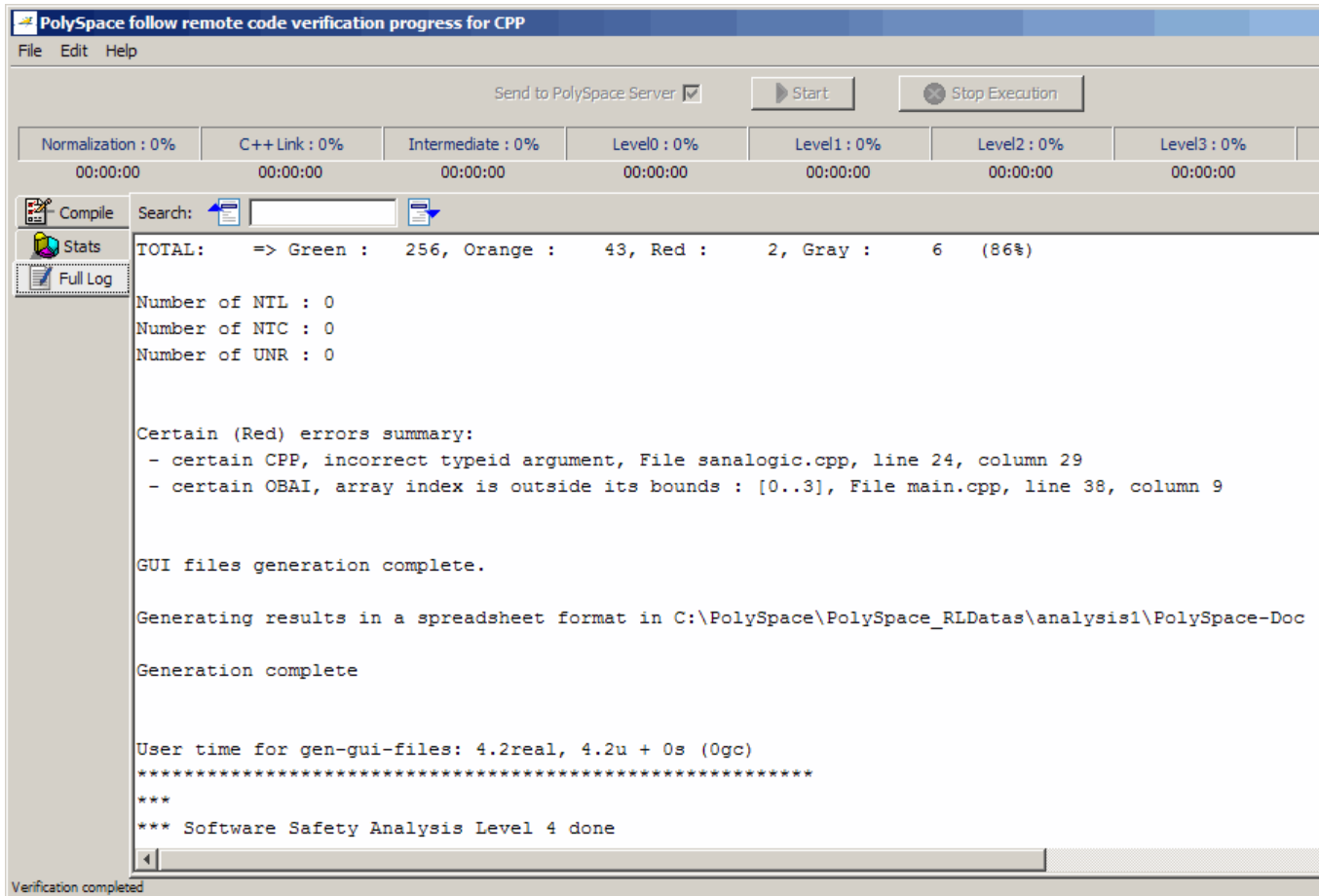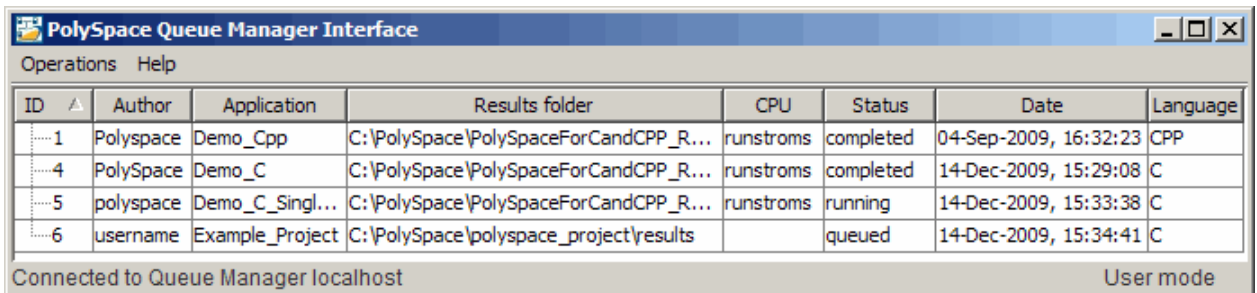
You can monitor the progress of the verification by watching the progress bar and viewing the logs at the bottom of the window. The word `processing` appears under the current phase. The progress bar highlights each completed phase and displays the amount of time for that phase.

The logs report additional information about the progress of the verification. The information appears in the log display area at the bottom of the window. The full log displays by default. It display messages, errors, and statistics for all phases of the verification. You can search the full log

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

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

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

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

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

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

ID △	Author	Application	Results folder	CPU	Status	Date	Language
1	Polyspace	Demo_Cpp	C:\PolySpace\PolySpaceForCandCPP_R...	runstroms	completed	04-Sep-2009, 16:32:23	CPP
4	PolySpace	Demo_C	C:\PolySpace\PolySpaceForCandCPP_R...	runstroms	completed	14-Dec-2009, 15:29:08	C
5	polyspace	Demo_C_Singl...	C:\PolySpace\PolySpaceForCandCPP_R...	runstroms	running	14-Dec-2009, 15:33:38	C
6	username	Example_Project	C:\PolySpace\polyspace_project\results		queued	14-Dec-2009, 15:34:41	C

Connected to Queue Manager localhost   User mode

Viewing Verification Log File on Server

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

To view a log file on the server:

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

The **PolySpace Queue Manager Interface** opens.

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

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

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

Stopping Server Verification Before It Completes

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

To stop a server verification:

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

The **PolySpace Queue Manager Interface** opens.

ID △	Author	Application	Results folder	CPU	Status	Date	Language
1	Polyspace	Demo_Cpp	C:\PolySpace\PolySpaceForCandCPP_R...	runstroms	completed	04-Sep-2009, 16:32:23	CPP
4	PolySpace	Demo_C	C:\PolySpace\PolySpaceForCandCPP_R...	runstroms	completed	14-Dec-2009, 15:29:08	C
5	polyspace	Demo_C_Singl...	C:\PolySpace\PolySpaceForCandCPP_R...	runstroms	running	14-Dec-2009, 15:33:38	C
6	username	Example_Project	C:\PolySpace\polyspace_project\results		queued	14-Dec-2009, 15:34:41	C

Connected to Queue Manager localhost   User mode

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

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

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

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

Removing Verification Jobs from Server Before They Run

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

Note If the job has started running, you must stop the verification before you can remove the job (see "Stopping Server Verification Before It Completes" on page 7-13). Once you have aborted a verification, you can remove it from the queue.

To remove a job from the server queue:

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

The **PolySpace Queue Manager Interface** opens.

ID △	Author	Application	Results folder	CPU	Status	Date	Language
1	Polyspace	Demo_Cpp	C:\PolySpace\PolySpaceForCandCPP_R...	runstroms	completed	04-Sep-2009, 16:32:23	CPP
4	PolySpace	Demo_C	C:\PolySpace\PolySpaceForCandCPP_R...	runstroms	completed	14-Dec-2009, 15:29:08	C
5	polyspace	Demo_C_Singl...	C:\PolySpace\PolySpaceForCandCPP_R...	runstroms	running	14-Dec-2009, 15:33:38	C
6	username	Example_Project	C:\PolySpace\polyspace_project\results		queued	14-Dec-2009, 15:34:41	C

Connected to Queue Manager localhost   User mode

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

The job is removed from the queue.

Changing Order of Verification Jobs in Server Queue
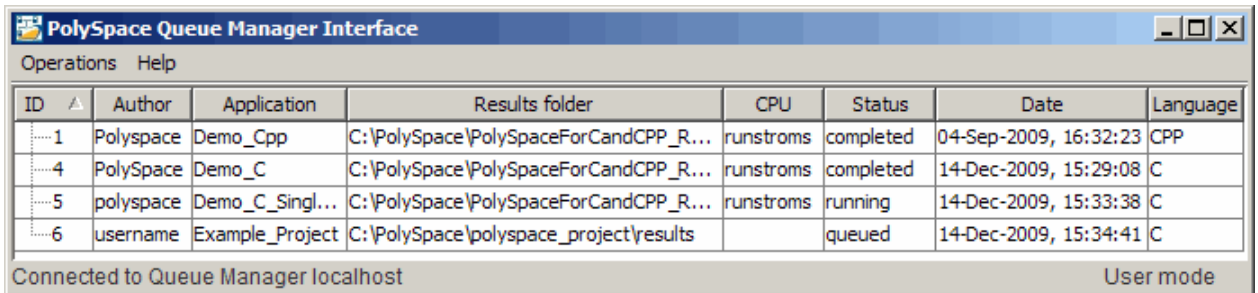
You can change the priority of verification jobs in the server queue to determine the order in which the jobs run.

To move a job within the server queue:

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

The **PolySpace Queue Manager Interface** opens.

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

The job is moved down in the queue.

3 Repeat this process to reorder the jobs as necessary.

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

Purging Server Queue

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

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

To purge the server queue:

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

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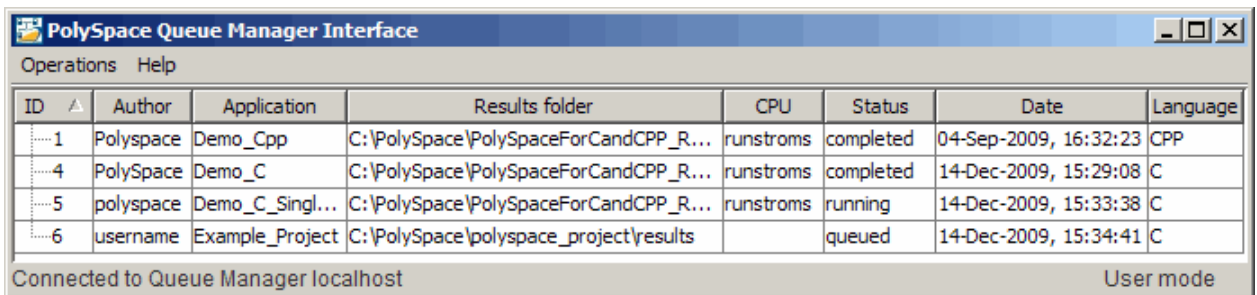
2 Select **Operations > Purge queue**. The Purge queue dialog box opens.
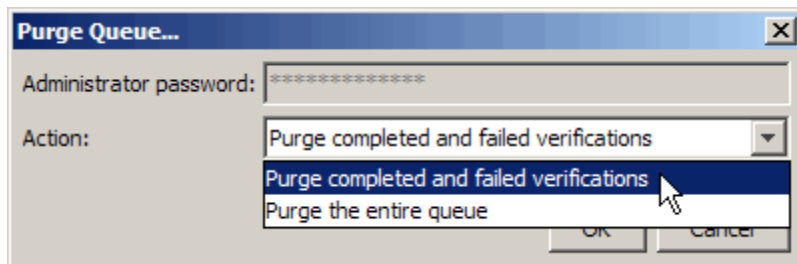
3 Select one of the following options:

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

- **Purge completed and aborted analysis** — Removes all completed and aborted 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 `administrator`.

To set the Queue Manager password:

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

The PolySpace Queue Manager Interface opens.

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

The Change Administrator Password dialog box opens.

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

The password is changed.

Sharing Server Verifications Between Users

Security of Jobs in Server Queue

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

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

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

analysis-keys.txt File

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

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

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

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

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

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

The fields in the file are tab-separated.

The file cannot contain blank lines.

Example:

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

Sharing Verifications Between Accounts

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

To share a verification with another user:

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

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

The second user can then download or manage the verification.

Magic Key to Share Verifications

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

The format for a magic key is as follows:

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

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

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

If analysis-keys.txt File is Lost or Corrupted

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

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

To use administrator mode:

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

The **PolySpace Queue Manager Interface** opens.

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

3 Enter the Queue Manager **Password**.

4 Click **OK**.

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

Running Verifications on PolySpace Client

In this section...
"Starting Verification on Client" on page 7-22
"What Happens When You Run Verification" on page 7-23
"Monitoring the Progress of the Verification" on page 7-24
"Stopping Client Verification Before It Completes" on page 7-25

Starting Verification on Client

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

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

If you launch a verification on C++ code containing more than 3,000 assignments and calls, the verification will stop and you will receive an error message.

To start a verification that runs on a client:

1 Open the Launcher.

2 Open the project containing the files you want to verify. For more information, see Chapter 4, "Setting Up a Verification Project".

3 Ensure that the **Send to PolySpace Server** check box is not selected.

4 If you see a warning that multitasking is not available when you run a verification on the client, click **OK** to continue and close the message box. This warning only appears when you clear the **Send to PolySpace Server** check box.

5 Click the **Start** button. ▶ Start

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

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

Note If you see the message Verification process failed, click **OK** and go to "Verification Process Failed Errors" on page 8-2.

---

7 When the verification completes, a message dialog box appears telling you that the verification is complete and asking if you want to open the Viewer.
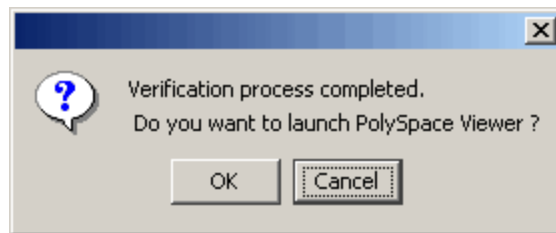
8 Click **OK** to open your results in the Viewer.

For information on viewing your results, see "Opening Verification Results" on page 9-8.

What Happens When You Run Verification

The verification has three main phases:

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

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

"Generate a Main Using a Given Class" in the *PolySpace Products for C++ Reference*.

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

Monitoring the Progress of the Verification

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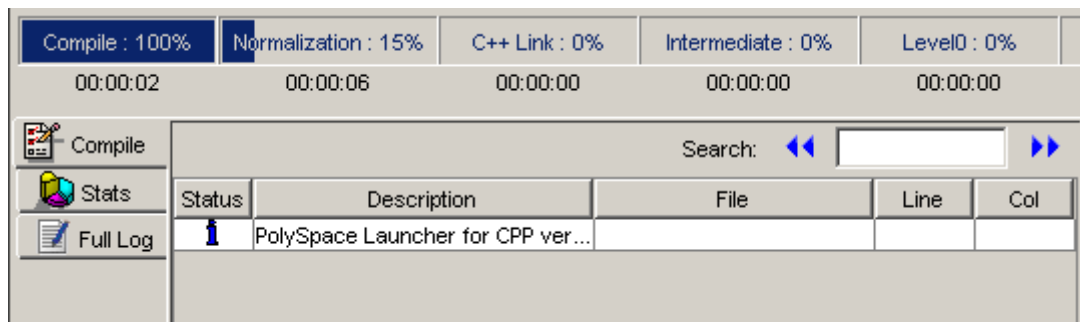
The progress bar highlights the current phase in blue and displays the amount of time and completion percentage for that phase.

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

To view the logs:

1 The compile log is displayed by default.

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

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

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

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

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

Stopping Client Verification Before It Completes

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

To stop a verification:

1 Click the **Stop Execution** button.

A warning dialog box appears.
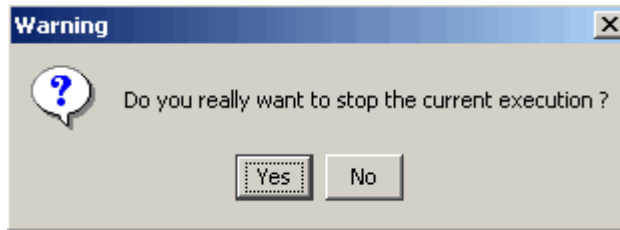
2 Click **Yes**.

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

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

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

Running Verifications from Command Line

Launching Verifications in Batch

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

All these commands begin with the following prefixes:

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

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

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

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

---

Managing Verifications in Batch

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

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

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

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

  - [-f] force download (without interactivity)

  - -admin -p *<password>* allows administrator to download results.

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

  - [-v|version] gives release number.

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

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

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

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

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

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

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

  - [-open-launcher] display the log in the graphical user interface of launcher.

  - [-full] give full log file.

- ▪ psqueue-set-password *<password>* *<new password>* — change
  administrator password.

- psqueue-check-config — check the configuration of Queue Manager.

  - ▪ [-check-licenses] check for licenses only.

- psqueue-upgrade — Allow to upgrade a client side (see the PolySpace
  Installation Guide in the <PolySpace Common Dir>/Docs folder).

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

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

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

8

Troubleshooting Verification Problems

Verification Process Failed Errors

In this section...
"Overview" on page 8-2
"Hardware Does Not Meet Requirements" on page 8-2
"You Did Not Specify the Location of Included Files" on page 8-3
"PolySpace Software Cannot Find the Server" on page 8-3
"Viewing Error Information When the Verification Stops" on page 8-4
"Limit on Assignments and Function Calls" on page 8-5

Overview

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

Message	See
`Errors found when verifying host configuration`	"Hardware Does Not Meet Requirements" on page 8-2
`catastrophic error:  could not open source file "include.h"` (where `include.h` represents the included file)	"You Did Not Specify the Location of Included Files" on page 8-3
`Error:  Unknown host :`	"PolySpace Software Cannot Find the Server" on page 8-3
`License error:  number-of assignments and function calls is too big for -unit mode`	"Limit on Assignments and Function Calls" on page 8-5

Hardware Does Not Meet Requirements

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

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

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

Errors found when verifying host configuration.

You can:

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

You Did Not Specify the Location of Included Files

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

include.h: No such file or folder

For information on how to specify the location of include files, see "Creating New Projects" on page 4-7.

PolySpace Software Cannot Find the Server

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

Error: Unknown host :

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

1 Select **Edit > Preferences**.

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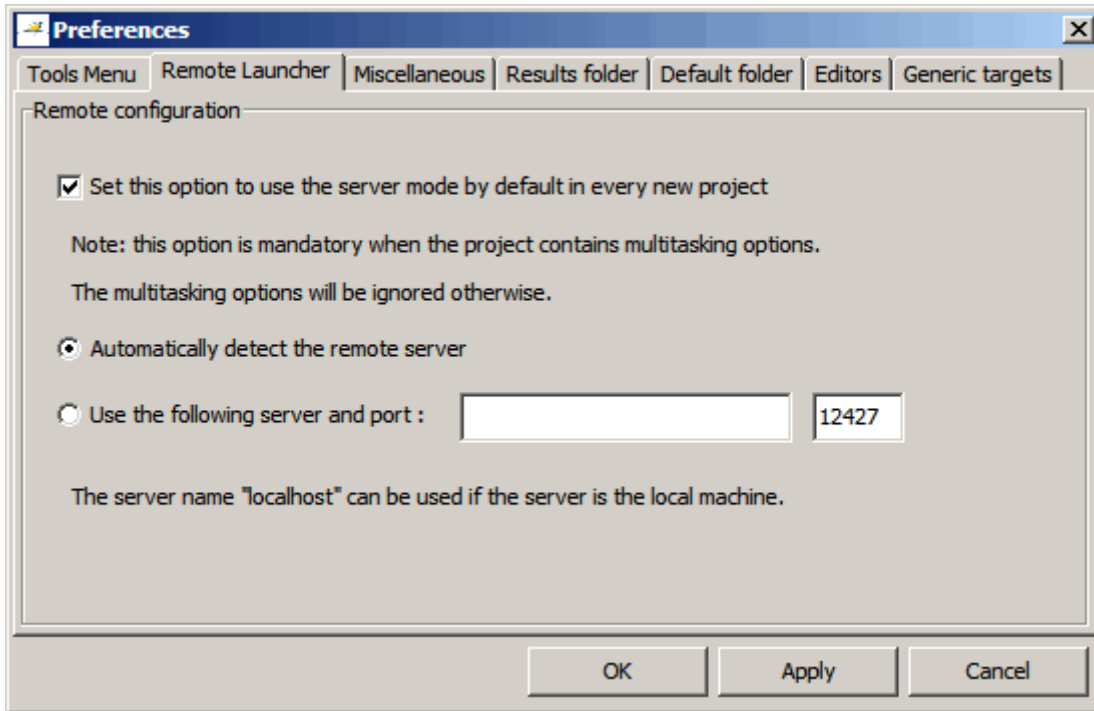
By default, PolySpace software automatically finds the server. You can specify the server by selecting **Use the following server and port** and providing the server name and port. For information about setting up a server, see the *PolySpace Installation Guide*.

Viewing Error Information When the Verification Stops

If the PolySpace software provides no graphical result, it lists the errors and their locations at the end of the log file. To find the errors, scroll through the verification log file, starting at the end and working backwards.

The following example shows the log file information that results if you use the -class-analyzer *argument*, but the verification cannot find *argument* in the source code:

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

```
***
*** Beginning Visual C++ 8 source normalization
***
************************************************************
**** Visual C++ 8 source normalization - 1 (Loading)
------------------------------------------------------------------------
User Program Error: Argument of option -class-analyzer not found.
|                   Class or typedef MyClass does not exist.
|Please correct the program and restart the verifier.
------------------------------------------------------------------------
------------------------------------------------------------------------
---                                                                   ---
---   Verifier has encountered an internal error.                     ---
---   Please contact your technical support.                          ---
---                                                                   ---
------------------------------------------------------------------------
Failure at: Sep 24, 2009 17:16:26
User time for polyspace-cpp: 25.6real, 25.6u + 0s (0gc)
Error: Exiting because of previous error
***
*** End of PolySpace Verifier analysis
***
```

Limit on Assignments and Function Calls

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

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

*** License error: number of assignments and function calls is
too big for -unit mode (5534 v.s 3000).
*** Aborting.
```

PolySpace Client for C/C++ software can only verify C++ code with up to
3,000 assignments and calls.

To verify code containing more than 3,000 assignments and calls, launch your verification on the PolySpace Server for C/C++.

Compile Errors

In this section...

Overview

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

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

Examining the Compile Log

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

To examine errors in the Compile log:

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

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

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

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

The file opens in your text editor.

Note You must configure a text editor before you can open source files. See "Configuring Text and XML Editors" on page 4-16.

4 If you do not understand the error information in the **Detail** pane, right-click the row for the message and select **Open Preprocessed File**.

This action opens the .ci file that the PolySpace software uses to compile the source file. The contents of this file helps you understand the compilation error.

5 The two compilation errors in this example occurred because the compiler encountered unexpected type definitions. To correct the errors in this example:

- invalid combination of type specifiers

  in the line

  typedef unsigned long wchar_t;

  By default, wchar_t is a C++ keyword. For some C++ compilers, wchar_t is not defined, and you can use it as a typedef. The PolySpace compiler allows you to define wchar_t as a typedef using the -wchar-t-is typedef option.

- first parameter of allocation function must be of type "size_t"

  in the line

  void * operator new(unsigned long mysize);

  the new operator has been prototyped with an unsigned long parameter, which requires that size_t be unsigned long. To fix this compile error, set the -size-t-is-unsigned-long option.

6 Rerun the verification.

Includes

As for the C language, access to the standard header files must be provided when the applications use the standard library.

The original code uses standard header files, but a message can appear:

```
Error message:
file.cpp", line 1: catastrophic error: could not open source file "iostream.h"
file.cpp:
 1 #include "iostream.h"
```

Use the -I option to include the correct header files, including the header files of the compiler.

Initialization of Global Variables

When a data member of a class is declared static in the class definition, it is a *static member* of the class. Static data members are initialized and destroyed outside the class, as they exist even when no instance of the class has been created.

```
class Test
{
public:

 static int m_number = 0;
};
```

Error message:

```
Verifying test_ko.cpp
/sources/test_ko.cpp, line 4: error: a member with an in-class
initializer must be const
|   static int m_number = 0;
|                     ^

1 error detected in the compilation of "test_ko.cpp".
```

Corrected code:

in file Test.h	in file Test.cpp
```class Test { public: static int m_number; };```	```int Test::m_number = 0;```

Note Some dialects, other than those supported by PolySpace Client for C/C++, accept the default initialization of static data member during the declaration.

#error directive

The PolySpace software can terminate during compilation with an unsupported platform #error. This error means that the software does not recognize the header data types due to missing compilation flags.

Message

```
#error directive:  !Unsupported platform; stopping!
```

Code Used

```
#if defined(__BORLANDC__) || defined(__VISUALC32__)
#   define MYINT int // then use the int type
#elif defined(__GNUC__) // GCC doesn't support myint
#   define MYINT long // but uses 'long' instead
#else
#   error !Unsupported platform; stopping!
#endif
```

Solution

In the PolySpace software, all compilation directives must be explicit. In this example, the compilation stops because you did not specify the __BORLANDC__, or the __VISUALC32__, or the __GNUC__ compilation flags. To fix this error, in

the **Target/Compilation** section, under **Analysis options**, for the **Defined Preprocessor Macros** option, specify one of those three compilation flags and restart the verification.

Dialect Issues

ISO versus Default Dialects

The ISO dialect strictly follows the ISO/IEC 14882:1998 ANSI C++ standard. If you specify the -dialect iso option, the PolySpace compiler might produce permissiveness errors. The following code contains five common permissiveness errors that occur if you specify the -dialect iso option. These errors are explained in detail following the code.

If you do not specify the -dialect option, the PolySpace compiler uses a default dialect that many C++ compilers use; the default dialect is more permissive with regard to the C++ standard.

Original code (file permissive.cpp):

```
1
2   class B {} ;
3   class A
4   {
5   friend B ;
6   enum e ;
7   void f() { long float ff = 0.0 ;}
8   enum e { OK = 0, KO } ;
9   };
10  template <class T>
11  struct traits
12  {
13  typedef T * pointer ;
14  typedef T * pointer ;
15  } ;
16  template<class T>
```

```
17  struct C
18  {
19  typedef traits<T>::pointer pointer ;
20  } ;
21  int main()
22  {
23  C<int> c ;
23  }
```

- Using -dialect iso, line 5 should be: friend class B:

    ```
    "./sources/permissive.cpp", line 5: error: omission of "class"
    is nonstandard
      friend B ;
    ```

- Using -dialect iso, the line 6 must be removed:

    ```
    "./sources /permissive.cpp", line 6: error: forward declaration
    of enum type
    is nonstandard
      enum  e ;
          ^
    ```

- Using -dialect iso, line 7 should be: double ff = 0.0:

    ```
    "./sources/permissive.cpp", line 7: error: invalid combination
    of type
    specifiers
      long float ff = 0.0 ;
      ^
    ```

- Using -dialect iso, line 14 needs to be removed:

    ```
    "./sources/permissive.cpp", line 14: error: class member typedef
    may not be
    redeclared
      typedef T *  pointer ; // duplicate !
            ^
    ```

- Using -dialect iso, line 21 needs to be changed by: typedef *typename* traits<T>::pointer pointer

```
"./sources/permissive.cpp", line 21: error: nontype
"traits<T>::pointer [with T=T]" is not a type name
  typedef traits<T>::pointer pointer ;
```

All these error messages disappear if you specify the -dialect default option.

CFront2 and CFront3 Dialects

As mentioned at the beginning of this section, cfront2 and cfront3 dialects were already being used before the publication of the ANSI C++ standard in 1998. Nowadays, these two dialects are used to compile legacy C++ code.

If the cfront2 or cfront3 options are not selected, you may get the common error messages below.

Variable Scope Issues

The ANSI C++ standard specifies that the scope of the declarations occurring inside loop definition is local to the loop. However some compilers may assume that the scope is local to the bloc ({ }) that contains the loop.

Original code:

```
for (int i = 0; i < maxval; i++) {...}
if (i == maxval) {
 ...
}
```

Error message:

```
Verifying Test.cpp
"../sources/Test.cpp", line 26: error: identifier "i" is undefined
  if (i == maxval) {
    ^
```

Note This kind of construction has been allowed by compilers until 1999, before the standard became more strict.

"bool" Issues

Standard type may need to be turned into boolean type.

Original code:

```
enum bool
 {
   FALSE=0,
   TRUE
 };
class CBool
{
public:
 CBool ();
 CBool (bool val);
 bool m_val;
};
```

Error message:

```
Verifying C++ sources ...
Verifying CBool.cpp
"../sources/CBool.h", line 4: error: expected either a definition
or a tag name
 enum  bool
    ^
```

Visual Dialects

The following messages appears if the compiler is based on a Visual® dialect
(including visual8).

Import Folder

When a Visual application uses #import directives, the Visual C++ compiler
generates a header file that contains some definitions. These header files have
a .tlh extension, and PolySpace for C/C++ requires the folder containing
those files.

Original code:

```
#include "stdafx.h"
```

```
#include <comdef.h>
#import <MsXml.tlb>
MSXML::_xml_error e ;
MSXML::DOMDocument* doc ;
int _tmain(int argc, _TCHAR* argv[])
{
 return O;
}
```

Error message:

```
"../sources/ImportDir.cpp", line 7: catastrophic error: could not
open source file "./MsXml.tlh"
 #import <MsXml.tlb>
         ^
```

The Visual C++ compiler generates these files in its "build-in" folder (usually Debug or Release). Therefore, in order to provide those files, the application needs to be built first. Then, the option -import-dir=<*build folder*> must be set with a correct path folder.

pragma Pack
Using a different value with the compile flag (#pragma pack) can lead to a linking error message.

Original code:

test1.cpp	type.h	test2.cpp
#pragma pack(4) #include "type.h"	struct A { char c ; int i ; } ;	#pragma pack(2) #include "type.h"

Error message:

```
Pre-linking C++ sources ...
"../sources/type.h", line 2: error: declaration of class "A" had
```

```
a different meaning during compilation of "CPP-ALL/SRC/MACROS/test1.cpp"
(class types do not match)
 struct A
    ^
   detected during compilation of secondary translation unit
"CPP-ALL/SRC/MACROS/test2.cpp"
```

The option `-ignore-pragma-pack` is mandatory to continue the verification.

GNU Dialect

The GNU dialect is based on GCC 3.4. The GNU dialect supports the keyword `__asm__ __volatile__`, which is used to support inline functions. For example, the *<sys/io.h>* header includes many inline functions. The GNU dialect supports these inline functions.

PolySpace software supports the following GNU elements:

- Variable length arrays

- Anonymous structures:

```
void f(int n) { char tmp[n] ; /* ... */ }

union A {
 struct {
  double x ;
  double y ;
  double z ;
 };
 double tab[3];
} a ;


void main(void) {

 assert(&(a.tab[0]) == &(a.x)) ;

}
```

- All other syntactic constructions allowed by GCC, except as noted below

Partial Support

Zero-length arrays have the same support as in Visual Mode. They are allowed when used through a pointer, but not in a local variable.

Syntactic Support Only

PolySpace software provides syntactic support for the following options, but not semantic support:

- `__attribute__(...)` is allowed, but generally not taken into account.

- No special stubs are computed for predeclared functions such as `__builtin_cos`, `__builin_exit`, and `__builtin_fprintf`).

Not Supported

The following options are not supported:

- The keyword `__thread`

- Statement expressions:

  ```
  int i = ({ int tmp ; tmp = f() ; if (tmp > 0 ) { tmp = 0 ; } tmp ; })
  ```

- Taking the address of a label:

  ```
  { L : void *a = &&L ; goto *a ; }
  ```

- General C99 features supported by default in GCC, such as complex built-in types (`__complex__`, `__real__`, etc.).

- Extended designators initialization:

  ```
  struct X { double a; int b[10] } x = { .b = { 1, [5] =2 },
  .b[3] = 1, .a = 42.0 };
  ```

- Nested functions

Examples

Example 1: `_asm_volatile_` keyword

In the following example, for the inb_p function to correctly manage the return of the local variable _v, the __asm__ __volatile__ keyword is used as follows:

```
extern inline unsigned char
inb_p (unsigned short port)
{
  unsigned char _v;

  __asm__ __volatile__ ("inb %w1,%0\noutb %%al,$0x80":"=a"
          (_v):"Nd" (port));
  return _v;
}
...
```

Example 2: Anonymous Structure

The following example shows an unnamed structure supported by GNU:

```
class x
{
public:

  struct {
  unsigned int a;
  unsigned int b;
  unsigned int c;
  };
  unsigned short pcia;
  enum{
  ea = 0x1,
  eb = 0x2,
  ec = 0x3
  };

  struct {
  unsigned int z1;
```

```
     unsigned int z2;
     unsigned int z3;
     unsigned int z4;
     };
};

int main(int argc, char *argv[])
{
  class x myx;

  myx.a = 10;
  myx.z1 = 11;
  return(0);
}
```

Link Messages

STL Library C++ Stubbing Errors

PolySpace software provides an efficient implementation of all functions in the Standard Template Library (STL). The STL and platforms may have different declarations and definitions; otherwise, the following error messages appear:

Original code:

```
#include <map>

struct A
{
 int m_val;
};

struct B
{
 int m_val;
 B& operator=(B &) ;
};

typedef std::map<A, B> MAP ;

int main()
{
 MAP m ;
 A a ;
 B b ;

 m.insert(std::make_pair(a,b)) ;
}
```

Error message:

```
Verifying template.cpp
"<Product>/Verifier/cinclude/new_stl/map", line 205: error: no operator
"=" matches these operands
 operand types are: pair<A, B> = const map<A, B, less<A>>::value_type
 { volatile int random_alias = 0 ; if (random_alias) *((pair<Key, T> * )
_pst_elements) = x ; } ; // read of x is done here

 detected during instantiation of
"pair<__pst__generic_iterator<bidirectional_iterator_tag, pair<const Key,
T>>, bool> map<Key, T, Compare>::insert(const map<Key, T, Compare>::
value_type &) [with Key=A, T=B, Compare=less<A>]" at line 23 of "/cygdrive/
c/_BDS/Test-Polyspace/sources/template.cpp"
```

Using the -no-stub-stl option avoids this error message. Then, you need to add the folder containing definitions of own STL library as a folder to include using the option -I.

The preceding message can also appear with the folder names:

```
"<Product>/cinclude/new_stl/map", line 205: error: no operator "="
matches these operands

"<Product>/cinclude/pst_stl/vector", line 64: error: more than one
operator "=" matches these operands:
```

Be careful that other compile or linking troubles can appear with your own template definitions.

Lib C Stubbing Errors

Extern C Functions

Some functions may be declared inside an extern C { } bloc in some files, but not in others. In this case, the linkage is different which causes a link error, because it is forbidden by the ANSI standard.

Original code:

```
extern "C" {
 void* memcpy(void*, void*, int);
```

```
}
class Copy
{
public:
 Copy() {};
 static void* make(char*, char*, int);
};
void* Copy::make(char* dest, char* src, int size)
{
 return memcpy(dest, src, size);
}
```

Error message:

```
Pre-linking C++ sources ...

<results_dir>/test.cpp, line 2: error: declaration of function "memcpy"
is incompatible with a declaration in another translation unit
(parameters do not match)
|           the other declaration is at line 4096 of "__polyspace__stdstubs.c"
|    void* memcpy(void*, void*, int);
|          ^
|          detected during compilation of secondary translation unit "test.cpp"
```

The function memcpy is declared as an external "C" function and as a C++ function. It causes a link problem. Indeed, function management behavior differs whether it relates to a C or a C++ function.

When such error happens, the solution is to homogenize declarations, i.e. add extern "C" { } around previous listed C functions.

Another solution consists in using the permissive option -no-extern-C. It removes all extern "C" declarations.

Functional Limitations on Some Stubbed Standard ANSI Functions

- signal.h is stubbed with functional limitations: signal and raise functions do not follow the associated functional model. Even if the function

raise is called, the stored function pointer associated to the signal number is not called.

- No jump is performed even if the `setjmp` and `longjmp` functions are called.

- `errno.h` is partially stubbed. Some math functions do not set `errno`, but instead, generate a red error when a range or domain error occurs with **ASRT** checks.

You can also use the compile option `POLYSPACE_STRICT_ANSI_STANDARD_STUBS` (`-D` flag). This option only deactivates extensions to ANSI C standard libC, including the functions `bzero`, `bcopy`, `bcmp`, `chdir`, `chown`, `close`, `fchown`, `fork`, `fsync`, `getlogin`, `getuid`, `geteuid`, `getgid`, `lchown`, `link`, `pipe`, `read`, `pread`, `resolvepath`, `setuid`, `setegid`, `seteuid`, `setgid`, `sleep`, `sync`, `symlink`, `ttyname`, `unlink`, `vfork`, `write`, `pwrite`, `open`, `creat`, `sigsetjmp`, `__sigsetjmp`, and `siglongjmpare`.

Troubleshooting Using the Preprocessed .ci Files

Overview

The preceding paragraphs explained the common types of compile or linking errors messages. They are associated with C++ dialects, or specific options used by the dialect (for example, Microsoft Visual C++ with the option -import-dir).

Nevertheless, sometimes the error messages are not sufficient to find the cause of problems. Indeed, they do not correspond to common error messages listed above.

PolySpace, as others compilers, transforms a source code to a preprocessed code. These files are located in the folder: *<results folder>*/CPP-ALL/SRC/MACROS or *<results folder>*/ALL/SRC/MACROS. They have a .ci extension, and they help to understand and find precisely the error problem.

Example of *ci* File

A *.ci file is a copy of original file containing whole header files inside a unique file:

- Compile flags activate some parts of code,

- Macro commands are expanded,

- Arguments which are described as #define xxx, are replaced by their owned definition,

- etc.

Extension.cpp	Extension.h
```#include "Extension.h"``` ```Extension::Extension(int val)``` ```{``` ``` m_val = 0;``` ``` ABS(val);``` ``` if (val > MAX_VALUE)``` ``` m_val = -1;``` ``` }``` ```#ifdef _DEBUG``` ```void Extension::message(char*) {}``` ```#else``` ``` void print(char*) {}``` ``` #endif```	```#define MAX_VALUE 10``` ```#define ABS(x) ((x)<0?(x):-(x))``` ```class Extension``` ```{``` ```public:``` ``` int m_val;``` ``` Extension(int val);``` ```#ifdef _DEBUG``` ``` void message(char*);``` ```#else``` ``` void print(char*);``` ```#endif``` ```};```

The associated file Extension.ci uses the compile flag _DEBUG:

```
# 1 "../sources/extension.cpp"
# 1 "<Product>/Verifier/cinclude/polyspace_std_decls.h" 1

# 1 "../sources/extension.cpp" 2
# 1 "../sources/extension.h" 1
class Extension
{
public:
 int m_val;
 Extension(int val);

 message(char*);     // _DEBUG activates the message member function

};

# 2 "../sources/extension.cpp" 2

Extension::Extension(int val)
```

```
 {
  m_val = 0;
  ((val)<0?(val): -(val));  // EXPANDED MACRO ABS

  if (val > 10 )     // MAX_VALUE REPLACED BY 10
   m_val = -1;
  }

 void Extension::message(char*) {}
```

Analyzing these files with the compile flag `-D _DEBUG` expands the code fully and may help you find the problems quickly.

Troubleshooting Methodology

This guide is designed to help understanding errors messages, as well as the differences between your compiler and PolySpace:

1 Check whether the compile error messages come from a dialect problem.

2 Verify that linking error messages are related or not to:

- A C++ stubbing error which could be resolved by an option (like `-no-stl-stubs`)

- C stubbing error which could be resolved by an option or a compilation flag like `POLYSPACE_NO_STANDARD_STUBS` or `POLYSPACE_STRICT_ANSI_STANDARD_STUBS`.

3 Check the preprocessed `*.ci` files to see the expanded files. Looking at the preprocessed code can help to find errors faster.

Example with these original codes:

Child1.c	Child2.c	Test.h

`#define DEBUG`
 	
`#include "Test.h"`
 	
`class Child1 : public Test`
`{`
`public:`
` Child1();`
` Child1(int val);`
 	
` void search(int val);`
 	
`};`
	`#undef DEBUG`
	
	`#include "Test.h"`
	
	`class Child2 : public Test`
	`{`
	`public:`
	` Child2();`
	` Child2(int val);`
	
	` void qshort(int val);`
	
	`protected:`
	` int m_status;`
	`};`
		`class Test`
		`{`
		`public:`
		
		` Test();`
		` Test(int val);`
		
		` int getVal();`
		` void setVal(int val);`
		
		`#ifdef DEBUG`
		` void algorithm(int val,`
		`int max);`
		`#endif`
		
		`private:`
		` int m_val;`
		`};`

Error message:

```
Pre-linking C++ sources ...
"../sources/test.h", line 4: error: declaration of function
"Test::Test(const Test &)" does not match function
"Test::algorithm" during compilation of "CPP-ALL/SRC/
MACROS/Child2.cpp" (one may have been removed due to #define)
   class  Test
     ^
  detected during compilation of secondary translation unit
"CPP-ALL/SRC/MACROS/Child2.cpp"
```

In this example it is clear that DEBUG is defined in child1.c but not in child2.c, which creates two different definition of the class test.

The solution can also come up by comparing the two *.ci files:

Test.ci	Child2.ci
<pre>....

1 "../sources/Test.cpp" 2
1 "../sources/test.h" 1

class Test
{
public:
 Test();
 Test(int val);

 int getVal();
 void setVal(int val);

 void algorithm(int val, int max);

private:
 int m_val;
};

2 "../sources/Test.cpp" 2

....</pre> | <pre>....

1 "../sources/Child2.cpp" 2
1 "../sources/Child2.h" 1
1 "../sources/test.h" 1

class Test
{
public:
 Test();
 Test(int val);

 int getVal();
 void setVal(int val);

private:
 int m_val;
};

2 "../sources/Child2.h" 2

....</pre> |

Looking at the preprocessed code can help to find errors faster.

Reducing Verification Time

In this section...

Factors Impacting Verification Time

These factors affect how long it takes to run a verification:

- The size of the code

- The number of global variables

- The nesting depth of the variables (the more nested they are, the longer it takes)

- The depth of the call tree of the application

- The intrinsic complexity of the code, particularly with regards to pointer manipulation

Because many factors impact verification time, there is no precise formula for calculating verification duration. Instead, PolySpace software provides graphical and textual output to indicate how the verification is progressing (available on Windows and Linux platforms).

Displaying Verification Status Information

For *server* verifications, you can use the PolySpace Queue Manager to follow the progress of your verification. For more information, see "Monitoring Progress of Server Verification" on page 7-8.

For *client* verifications, you can monitor the progress of your verification using the progress bar and **Stats** log in the Launcher. For more information, see "Monitoring the Progress of the Verification" on page 7-24.
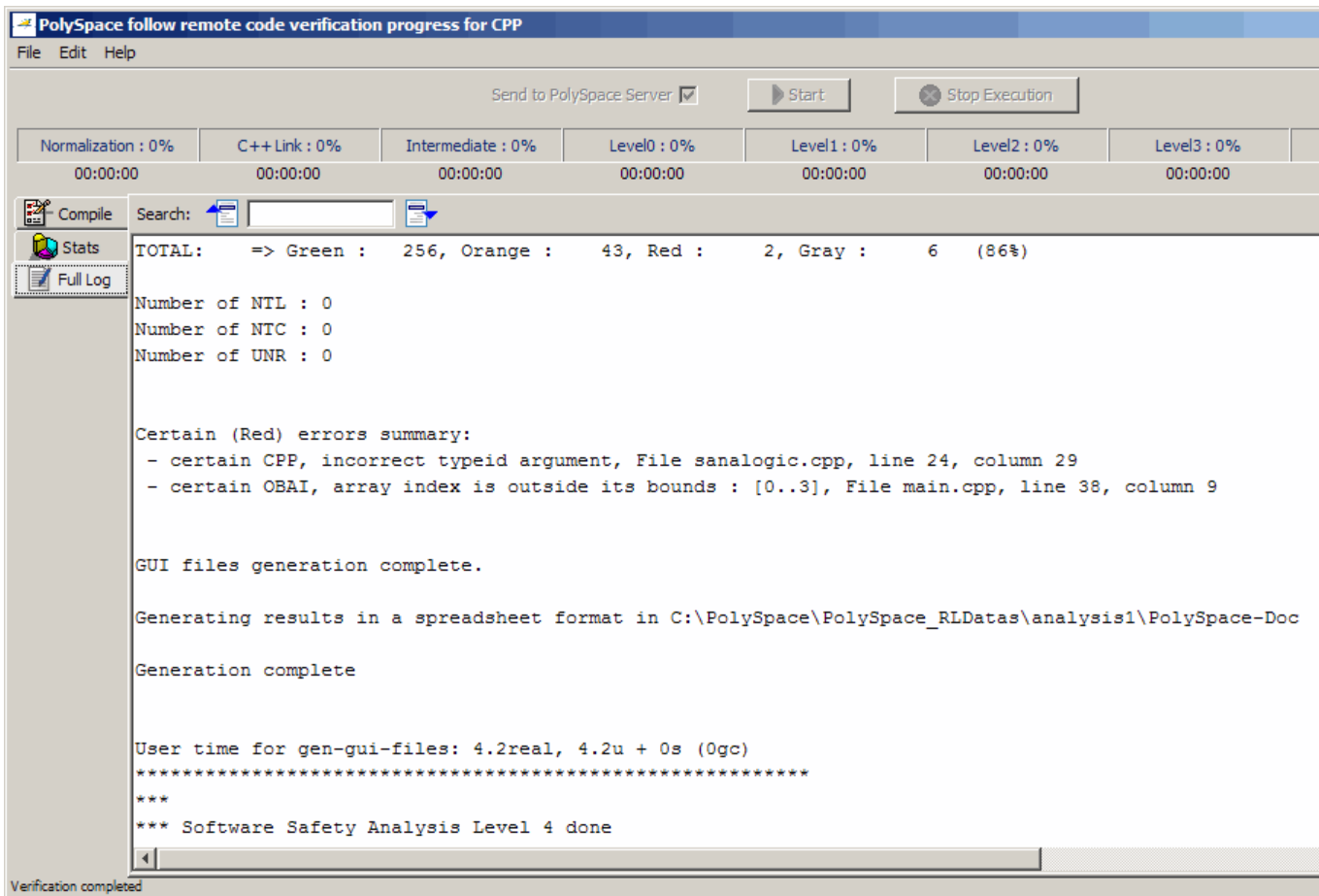
The progress bar highlights each completed phase and displays the amount of time for that phase. You can estimate the remaining verification time by extrapolating from this data, and considering the number of files and passes remaining.

Techniques for Improving Verification Performance

This section suggests methods to reduce the duration of a particular verification, with minimal compromise for the launch parameters or the precision of the results.

You can increase the size of a code sample for effective analysis by tuning the tool for that sample. Beyond that point, subdividing the code or choosing a lower precision level offers better results (-O1, -O0).

You can use several techniques to reduce the amount of time required for a verification:

- "Turning Antivirus Software Off" on page 8-35

- "Tuning PolySpace Parameters" on page 8-35

- "Subdividing Code" on page 8-35

- "Reducing Task Complexity" on page 8-46

- "Reducing Variable Complexity" on page 8-46

- "Choosing Lower Precision" on page 8-46

You can combine these techniques. See the performance-tuning flow chart in "Standard Scaling Options Flow Chart" on page 8-34.

Standard Scaling Options Flow Chart

Step 1: standard scaling options

- CPU must be > 1 GHz.
- Memory must be > 1 GB x #processors.
- Swap files must be > 1 GB or
  >= min(4 GB, memory size).
- /tmp must be > 10 MB.

No ← Hardware configuration OK?

Yes

Make sure no other verification is running.

- Slow verification can be normal.
- Consider splitting the application or using -unit-by-unit verification.

Yes ← Application > 20K lines?

No

- Slow verification can be normal using a generated main.
- Consider using -unit-by-unit verification.
- Consider usering class verification.

Yes ← Application > 5K lines?

Yes

- If you have passed level 0, you have meaningful results at level 0; open the PolySpace Viewer.
- Set tuning options when relevant.
- Do class verifications with the -class-only option.

Yes ← Still blocked?

Turning Antivirus Software Off

Disabling or switching off any third-party antivirus software for the duration of a verification can reduce the verification time by up to 40%.

Tuning PolySpace Parameters

Impact of Parameter Settings

Compromise to balance the time required to perform a verification and the time required to review the results. Launching PolySpace verification with the following options reduces the time taken for verification. However, these parameter settings compromise the precision of the results. The less precise the results of the verification, the more time you can spend reviewing the results.

Recommended Parameter Tuning

The MathWorks™ suggests that you use the parameters in the sequence listed. If the first suggestion does not increase the speed of verification sufficiently, try the second suggestion, and so on.

- Switch from `-O2` to a lower precision;

- Set the `-respect-types-in-globals` and `-respect-types-in-fields` options;

- Set the `-k-limiting` option to 2, then 1, or 0;

- Manually stub missing functions which write into their arguments.

- If some code uses some large arrays, use the `-no-fold` option.

For example, an appropriate launching command is

```
polyspace-c -O0 -respect-types-in-globals -k-limiting 0
```

Subdividing Code

- "An Ideal Application Size" on page 8-36

- "Benefits of Subdividing Code" on page 8-36

An Ideal Application Size

People have used PolySpace software to analyze numerous applications with greater than 100,000 lines of code.

There always is a compromise between the time and resources required to analyze an application, and the resulting selectivity. The larger the project size, the broader the approximations PolySpace software makes. Broader approximations produce more oranges. Large applications can require you to spend much more time analyzing the results and your application.

These approximations enable PolySpace software to extend the range of project sizes it can manage, to perform the verification further, and to solve traditionally incomputable problems. Balance the benefits derived from verifying a whole large application against the loss of precision that results.

Benefits of Subdividing Code

Subdividing a large application into smaller subsets of code provides several benefits. You:

- Quickly isolate a meaningful subset
- Keep all functional modules
- Can maintain a high precision level (for example, level O2)
- Reduce the number of orange items
- Get correct results are correct because you do not need to remove any thread affecting change shared data
- Reduce the code complexity considerably

Possible Issues with Subdividing Code

Subdividing code can lead to these problems:

- Orange checks can result from a lack of information regarding the relationship between modules, tasks, or variables.

- Orange checks can result from using too wide a range of values for stubbed functions.

- Some loss of precision; the verification considers all possible values for a variable.

When the Application is Incomplete. When the code consists of a small subset of a larger project, PolySpace software automatically stubs many procedures. PolySpace bases the stubbing on the specification or prototype of the missing functions. PolySpace verification assumes that all possible values for the parameter type are returnable.

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

Now consider a procedure `f` that modifies its input parameters `a` and `b`. `f` passes both parameters by reference. Suppose `a` can be from 0 through 10, and `b` any value between –10 and 10. In an automatically stubbed function, the combination `a=10` and `b=10` is possible, even if it is not possible with the real function. This situation introduces orange checks in a code snippet such as `1/(a*b - 100)`, where the division would be orange.

- So, even with precise stubbing, verification of a small section of code can introduce extra orange checks. However, the net effect from reducing the complexity is to reduce the total number of orange checks.

- With default stubbing, the increase in the number of orange checks as the result of this phenomenon tends to be more pronounced.

Considering the Effects of Application Code Size. PolySpace can make approximations when computing the possible values of the variables, at any point in the program. Such an approximation use a superset of the actual possible values.

For instance, in a relatively small application, PolySpace software can retain detailed information about the data at a particular point in the code. For

example, the variable VAR can take the values { -2 ; 1 ; 2 ; 10 ; 15 ; 16 ; 17 ; 25 }. If the code uses VAR to divide, the division is green (because 0 is not a possible value).

If the program is large, PolySpace software simplifies the internal data representation by using a less precise approximation, such as [-2 ; 2] U {10} U [15 ; 17] U {25}. Here, the same division appears as an orange check.

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

This phenomenon increases the number of orange warnings when the size of the program becomes large.

Recommended Approach

The MathWorks recommends that you begin with file-by-file verifications (when dealing with C language), package-by-package verifications (when dealing with Ada language), and class-by-class verifications (when dealing with C++ language).

The maximum application size is between 20,000 (for C++) and 50,000 lines of code (for C and Ada). For such applications of that size, approximations are not too significant. However, sometimes verification time is extensive.

Experience suggests that subdividing an application before verification normally has a beneficial impact on selectivity. The verification produces more red, green and gray checks, and fewer unproven Orange checks. This subdivision approach makes bug detection more efficient.
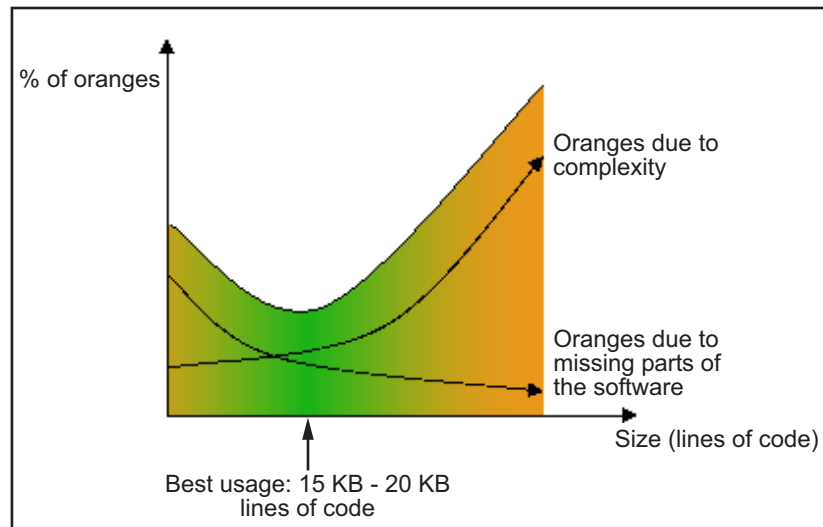
A compromise between selectivity and size

PolySpace verification is most effective when you use is as early as possible in the development process, before any other form of testing.

When you analyze a small module (for example, a file, piece of code, or package) using PolySpace software, focus on the red and gray checks. **Orange** unproven checks at this stage are interesting, because most of them deal with robustness of the application. The Orange checks change to red, gray, or green as the project progresses and you integrate more modules.

In the integration process, code can become so large (50,000 lines of code or more). This amount of code can cause the verification to take an unreasonable amount of time. You have two options:

- Stop using PolySpace verification at this stage (you have gained many benefits already).

- Analyze subsets of the code.

Selecting a Subset of Code

Subdividing a project for verification takes considerably less verification time for the sum of the parts than for the whole project considered in one pass. Consider data flow when you subdivide the code.

Consider two distinct concepts:

- Function entry points — Function entry points refer to the PolySpace execution model, because they start concurrently, without any assumption regarding sequence or priority. They represent the beginning of your call tree.

- Data entry points — Regard lines in the code that acquire data as data entry points.

Example 1

```
int complete_treatment_based_on_x(int input)
{
 thousand of line of computation...
}
```

Example 2

```
void main(void)
{
 int x;
 x = read_sensor();
 y = complete_treatment_based_on_x(x);
}
```

Example 3

```
#define REGISTER_1 (*(int *)0x2002002)
void main(void)
{
 x = REGISTER_1;
 y = complete_treatment_based_on_x(x);
}
```

In each case, the x variable is a data entry point and y is the consequence of such an entry point. y can be formatted data, due to a complex manipulation of x.

Because x is volatile, a probable consequence is that y contains all possible formatted data. You can remove the procedure complete_treatment_based_on_x completely, and let automatic stubbing work. The verification process considers y as potentially taking any value in the full range data (see "Stubbing" on page 6-2).

```
//removed definition of complete_treatment_based_on_x
void main(void)
{
 x = ... // what ever
 y = complete_treatment_based_on_x(x); // now stubbed!
}
```

Typical Examples of Removable Components, According to the Logic of the Data. Here are some examples of removable components, based on the logic of the data:

- **Error management modules** often contain a large array of structures accessed through an API, but return only a Boolean value. Removing the API code and retaining the prototype causes the automatically generated stub to return a value in the range [-2^31, 2^31-1], which includes 1 and 0. PolySpace considers the procedure able to return all possible answers, just like reality.

- **Buffer management for mailboxes coming from missing code** — Suppose an application reads a huge buffer of 1024 char. The application then uses the buffer to populate three small arrays of data, using a complicated algorithm before passing it to the main module. If the verification excludes the buffer, and initializes the arrays with random values instead, then the verification of the remaining code is just the same.

- Display modules

Subdivision According to Data Flow. Consider the following example.
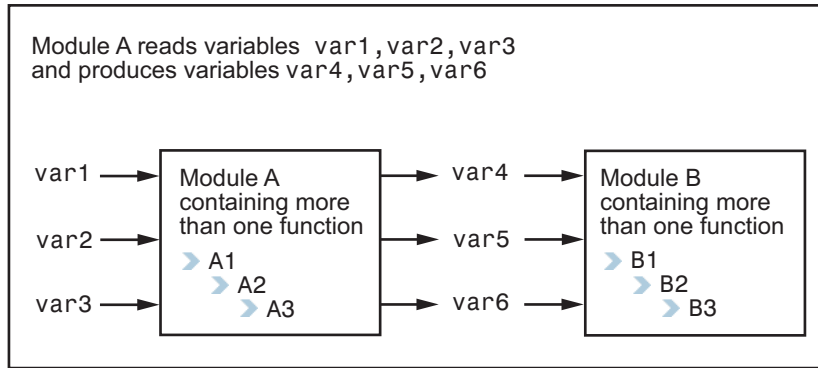
In this application, `var1`, `var2`, and `var3` can vary between the following ranges:

`var1`	From 0 through 10
`var2`	From 1 through 100
`var3`	From −10 through 10

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

As a result, `var4`, `var5`, and `var6` have the following specifications:

Ranges	`var4`	Between −60 and 110
	`var5`	From 0 through 12
	`var6`	From 0 through 100
Properties	And a set of properties between variables	• If `var2` is equal to 0, than `var4 > var5 > 5`.
		• If `var3` is greater than 4, than `var4 < var5 < 12`
		• ...

Subdivision in accordance with data flow allows you to analyze modules `A` and `B` separately.

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

- `B` uses variables 4, 5 and 6 initialized respectively to `[-60;110]`, `[0;12]` and `[-10;10]`

The consequences are:

- A slight loss of precision on the `B` module verification, because now PolySpace considers all combinations for `var4`, `var5`, and `var6`. It includes all possible combinations, even those combinations that the module `A` verification restricts.

  For example, if the `B` module included the test

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

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

- An in-depth investigation of the code is not necessary to isolate a meaningful subset. It means that a logical split is possible for any application, in accordance with the logic of the data.

- The results remain valid, because there no requirement to remove (for example) a thread that changes shared data.

- The code is less complex.

- You can maintain the maximum precision level.

Typical examples of removable components:

- Error management modules. A function `has_an_error_already_occurred` can return TRUE or FALSE. Such a module can contain a large array of structures accessed through an API. Removing API code with the retention of the prototype results in the PolySpace verification producing a stub that returns `[-2^31, 2^31-1]`. That result clearly includes 1 and 0 (yes and no). The procedure `has_an_error_already_occurred` returns all possible answers, just like the code would at execution time.

- Buffer management for mailboxes coming from missing code. Suppose the code reads a large buffer of 1024 char and then collates the data into three small arrays of data, using a complicated algorithm. It then gives this data to a main module for treatment. For the verification, PolySpace can remove the buffer and initialize the arrays with random values.

- Display modules.

Subdivide According to Real-Time Characteristics. Another way to split an application is to isolate files which contain only a subset of tasks, and to analyze each subset separately.

If a verification initiates using only a few tasks, PolySpace loses information regarding the interaction between variables.

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

If T1 modifies x and reads it at a particular moment, the values of x impact subsequent operations in T2.

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

- You could declare x as volatile to take into account all possible executions. Otherwise, x takes only its initial value or x variable remains constant, and verification of T2 is a subset of possible execution paths. You can get precise results, but it includes one scenario among all possible states for the variable x.

- You could initialize x to the whole possible range [10;15], and then call the T2 entry-point. This approach is accurate if x is calibration data.

Subdivide According to Files. This method is simple, but it can produce good results when you are trying to find red errors and bugs in gray code.

Simply extract a subset of files and perform a verification using one of these approaches:

- Use the principal entry point in this subset as the main.

- Create a `main` that calls randomly all functions that the subset of the code does not call.

Reducing Task Complexity

If the code contains two or more tasks, and particularly if there are more than 10,000 alias reads, set the `-lightweight-thread-model` option. This option reduces:

- Task complexity
- Verification time

There are some down sides:

- It causes more oranges and a slight loss of precision on reads of shared variables through pointers.
- The dictionary can omit some read/write accesses.

Reducing Variable Complexity

Variable Characteristic	Action
The types are complex.	Set the `-k-limiting [0-2]` option. Begin with 0. Go up to 1 or 2 in order to gain precision.
There are large arrays	Set the `-no-fold` option.

Choosing Lower Precision

The amount of simplification applied to the data representations depends on the required precision level (O0, O2), PolySpace software adjusts the level of simplification. For example:

- `-O0` — shorter computation time
- `-O2` — less orange warnings
- `-O3` — less orange warnings and bigger computation time

Obtaining Configuration Information

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

Configuration information includes:

- Hardware configuration
- Operating system
- PolySpace licenses
- Specific version numbers for PolySpace products

To obtain your configuration information, enter the following command:

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

The configuration information appears.

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

Removing Preliminary Results Files

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

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

To remove preliminary results files:

1 Open the project containing the results you want to delete in the Launcher.

2 Select **Tools > Clean Results.**

The preliminary results files are deleted.

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

9

Reviewing Verification Results

Before You Review PolySpace Results

In this section...

Overview: Understanding PolySpace Results

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

- **Red** – Indicates code that always has an error (errors occur every time the code is executed).

- **Gray** – Indicates unreachable code (dead code).

- **Orange** – Indicates unproven code (code might have a run-time error).

- **Green** – Indicates code that never has a run-time error (safe code).

When you analyze these colors, remember these rules:

- An instruction is verified only if no run-time error is detected in the previous instruction.

- The verification assumes that each run-time error causes a "core dump." The corresponding instruction is considered to have stopped, even if the actual run-time execution of the code might not stop. This means that red checks are always followed by gray checks, and orange checks only propagate the green parts through to subsequent checks.

- Focus on the verification message. Do not jump to false conclusions. You must understand the color of a check step by step, until you find the root cause of a problem.

- Determine the cause by examining the actual code. Do not focus on what the code is supposed to do.

Why Gray Follows Red and Green Follows Orange

Gray checks follow **red** checks, and **green** checks are propagated out of **orange** ones.

In the following example, consider why:

- The gray checks follow the **red** in the red function.

- There are **green** checks relating to the array.

```
void red(void)          extern int Read_An_Input(void);
{                       void propagate(void)
int x;                  {
x = 1 / x ;              int X;
x = x + 1;               int y[100];
}                        X = Read_An_Input();
                         y[X] = 0; // [array index within bounds]
                         y[X] = 0;
                        }
```

Consider each line of code for the red function:

- When PolySpace divides by *X*, X is not initialized. Therefore the corresponding check (Non Initialized Variable) on X is red.

- As a result, PolySpace stops all possible execution paths because they all produce an RTE. Therefore the subsequent instructions are gray (unreachable code).

Now, consider each line of code for the propagate function:

- *X* is assigned the return value of Read_An_Input. After this assignment, X = [-2^31, 2^31-1].

- At the first array access, you might see an "out of bounds" error because X can equal -3 as well as 3.

- Subsequently, all conditions leading to an RTE are truncated — they are no longer considered in the verification. On the following line, all executions in which X = [-2^31, -1] and [100, 2^31-1] are stopped.

- At the next instruction, X = [0, 99].

- Therefore, at the second array access, the check is green because X = [0, 99].

Summary

Green checks can be propagated out of orange checks.

The Message and What It Means

PolySpace software numbers checks to correspond to the code execution order.

Consider the instruction:

```
x++;
```

PolySpace first checks for a potential NIV (Non Initialized Variable) for x, and then checks the potential OVFL (overflow). This action mimics the actual execution sequence.

Understanding these sequences can help you understand the message presented by PolySpace, and what that message means.

Consider an orange NIV on x in the test:

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

You might conclude that the verification does not keep track of the value of x. However, consider the context in which the check is made:

```
extern int read_an_input(void);

void main(void)
{
int x;
if (read_an_input()) x = 100;
if (x > 101) // [orange on the NIV : non initialised variable ]
 { x++; } // gray code
}
```

Explanation

You can see the category of each check by clicking it in the Viewer. When you examine an orange check, you see that any value of a variable that results in a run-time error (RTE) is not considered further. However, as this example NIV (Non Initialized Variable) shows, any value that does not cause an RTE is verified on subsequent lines.

The correct interpretation of this verification result is that if x is initialized, the only possible value for it is 100. Therefore, x can never be both initialized and greater than 101, so the rest of the code is gray. This conclusion may be different from what you first suspect.

Summary

In summary:

- if "(x > 100)" does **NOT** mean that PolySpace does not know anything about x.

- if "(x > 100)" **DOES** means that PolySpace does not know whether X is initialized.

When you review results, remember:

- Focus on the PolySpace software message.

- Do not assume any conclusions.

The C++ Explanation

Verification results depend entirely on the code that you are verifying. When interpreting the results, do not consider:

- Any physical action from the environment in which the code operates.

- Any configuration that is not part of the verification.

- Any reason other than the code itself.

The only thing that the verification considers is the C++ code submitted to it.

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

```
extern int read_an_input(void);

void main(void)
{
 int x;
 int y[100];
 x = read_an_input();
 y[x ] = 0; // [array index within bounds]
 y[x-1] = (1 / X) + X ;
 if (x == 0)
  y[x] = 1; // gray code on this line
}
```

You can see that:

- The line containing the access to the y array is unreachable.

- Therefore, the test to assess whether `x = 0` is always false.

- **The initial conclusion is that "the test is always false."** You might conclude that this results from input data that is not equal to 0. However, `Read_An_Input` can be any value in the full integer range, so this is not the correct explanation.

Instead, consider the execution path leading to the gray code:

- The orange check on the array access (y[x]) truncates any execution path leading to a run-time error, meaning that subsequent lines deal with only x = [0, 99].

- The orange check on the division also truncates all execution paths that lead to a run-time error, so all instances where x = 0 are also stopped. Therefore, for the code execution path after the orange division sign, `x = [1; 99]`.

- x is never equal to 0 **at this line**. The array access is green (y (x − 1).

Note For the array access at the previous line (y[x]), we have X ~ [-2^31, 2^31-1] – hence the orange on (1 /X) .

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 stores the results on the server. To view your results, download the results file from the server to the client.

Note If you download results before the verification is complete, you get partial results and the verification continues.
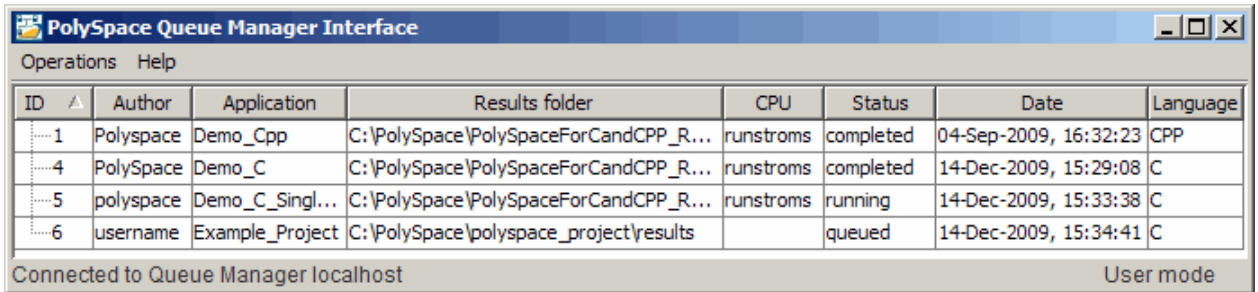
To download verification results to your client system:

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

The **PolySpace Queue Manager Interface** opens.

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

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

The Browse For Folder dialog box opens.
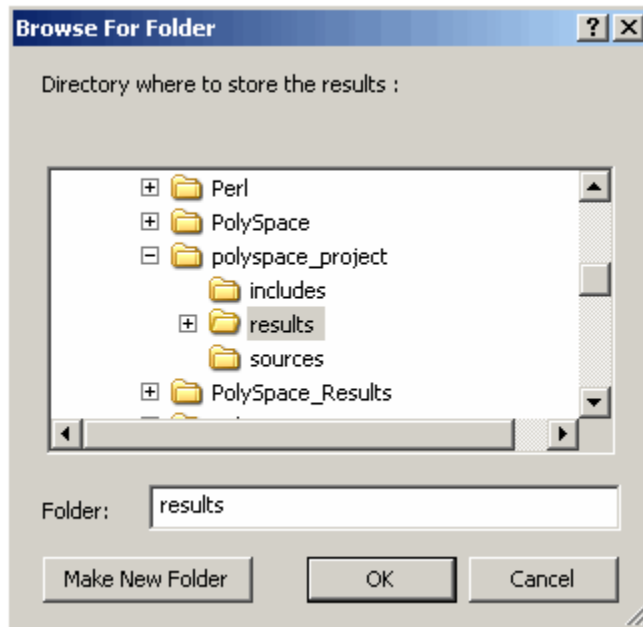
3 Select the folder into which you want to download results.

4 Click **OK** to download the results and close the dialog box.

When the download is complete, a dialog box opens asking if you want to open the PolySpace Viewer.
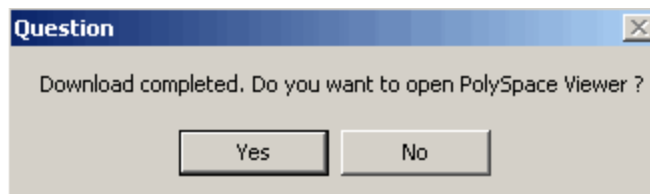
5 Click **Yes** to open the results.

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

Downloading Results Using Command Line

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

To download your results, enter the following command:

```
<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 Viewer.

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

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

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

For more information on managing verification jobs from the command line, see "Managing Verifications in Batch" on page 7-27.

Downloading Results from Unit-by-Unit Verifications

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

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

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

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

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

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

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

PolySpace Unit By Unit Results Synthesis

	Green	Orange	Inputs Orange	Dark Orange	Red	Grey	Total	Selectivity	Results	Log file
Source compliance phase results										Open log file
Unit single_file_analysis	97	8	8		2	4	111	93%	Open results	Open log file
Unit main	12	5	3				17	75%	Open results	Open log file
Unit example	99	10	10		5	77	191	95%	Open results	Open log file
Unit tasks2	30	2	2				32	94%	Open results	Open log file
Unit initialisations	52	6				3	61	90%	Open results	Open log file
Unit tasks1	33	5	1				38	87%	Open results	Open log file
	323	36	24	0	7	84	450	92%		

Opening Verification Results

Use the PolySpace Viewer to review the results of your verification.

To open the verification results:

1 Double-click the PolySpace Viewer icon.

2 Select **File** > **Open**.

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

4 Click **Open**.

The results appear in the Viewer window.

Exploring the Viewer Window

- "Overview" on page 9-13
- "Procedural Entities View" on page 9-15
- "Source Code View" on page 9-17
- "Coding Review Progress View" on page 9-19
- "Selected Check View" on page 9-20
- "Variables View" on page 9-20
- "Call Tree View" on page 9-23

Overview
The PolySpace Viewer looks like the following graphic.

Coding review progress view   Selected check view

Procedural   Variables   Source code   Call tree
entities view   view   view   view

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

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

This View...	Displays...
Procedural entities view (lower left)	List of the diagnostics (checks) for each file and function in the project
Source code view (lower right)	Source code for a selected check in the procedural entities view
Coding review progress view (upper left)	Statistics about the review progress for checks with the same type and category as the selected check
Selected check view (upper right)	Details about the selected check
Variables view	Information about the global variables declared in the source code
Call tree view	Tree structure of function calls

You can resize or hide any of these sections.

Procedural Entities View

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

Procedural entities	!	✖	?	✔	Line	...	‰	
Training_Project	1	0	10	48			83	
⊞ _polyspace_main.cpp				1	1		100	_polyspace_main.cpp
⊞ exception.stdh				1	1		0	exception.stdh
⊞ new.stdh				1	1		0	new.stdh
⊞ training.cpp	1		10	45	1		82	training.cpp
⊞ training.h				2	1		100	training.h
⊞ __polyspace__stdstubs.c				1	1		0	__polyspace__stdstub
⊞ __polyspace__stdstubscpp.cpp				1	1		0	__polyspace__stdstub

The file `example.c` is red because its has a run-time error. PolySpace software assigns to a file the color of the most severe error found in that file. The first column of the table is the procedural entity (the file or function). The following table describes some of the other columns in the procedural entities view.

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

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

> **Note** You can select which columns appear in the procedural entities view by right-clicking on the Procedural entities column heading, and selecting the columns you want to display..

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

Source Code View

The source code view (in the lower-right part of the Viewer window), shows the source code with colored checks highlighted.

```
training.cpp                                                          _ | 8 | X
98    void MathUtils::Recursion_2(int* depth)
99    {
100     Recursion (depth);
101   }
102
103   /* if depth<0, recursion will lead to division by zero */
104   void MathUtils::Recursion (int* depth)
105   {
106     float advance;
107
108     *depth = *depth + 1;
109     advance = 1.0/(float)(*depth);  // potential division by zero
110
111     if (*depth < 5
112       {
113         Recursion_
114       }
115   }
116
117   void MathUtils::Recursion_caller()
118   {
119     Utils  u;
120     int x=u.random_int();
121
122     x = -4;
123     if (u.random_int() > 0)
124       Recursion( &x );   // always encounters a division by zero
125
```

operator / on type float 64
  left: 1.0
  right: [-2.1475E^{+9} .. -9.9999E^{-1}] or 0.0 or [2.9999 .. 5.0001E^{+1}]
  result: [-1.0001 .. -4.6566E^{-10}] or [1.9999E^{-2} .. 3.3334E^{-1}]

Tooltips. Placing your cursor over a check displays a tooltip that provides
ranges for variables, operands, function parameters, and return values. For
more information on tooltips, see "Using Range Information in the Viewer"
on page 9-68.

Additional Information on checks. Clicking a check opens a message box
that provides more information about the check.

Coding Review Progress View

The Coding review progress view (upper-left part of the Viewer window),
displays statistics about how many checks you have reviewed. As you review
checks, the software updates these statistics.

The **Count** column displays a ratio and the **Progress** column displays the
equivalent percentage.

The first row displays the ratio of reviewed checks to total checks that have
the same color and category of the current check. In this example, the first
row displays the ratio of reviewed red IDP checks to total red IDP errors in
the project.

The second row displays the ratio of reviewed checks to total checks that have
the color of the current check. In this example, this is the ratio of red errors
reviewed to total red errors in the project.

The third row displays the ratio of the number of green checks to the total
number of checks, providing an indicator of the reliability of the software.

Selected Check View

The Selected check view (upper-right part of the Viewer window) displays information about the current check.

```
training.cpp / MathUtils::Pointer_Arithmetic() / line 72 / column 4

  ⊞        *p = 5; // Out of bounds

 ☑ NOW ▼ [        ▼] 📝 Fix Now

Error : pointer is outside its bounds
    dereference of variable 'p' (pointer to int 32, size: 32 bits):
        pointer is not null
        points to 4 bytes at offset 400 in allocated buffer of 400 bytes
        may point to variable or field of variable in: {MathUtils::Pointer_Arithmetic():tab}
```

When reviewing checks, you use the Selected check view to mark checks as **Reviewed**, and enter comments to describe the results of your review. This helps you track the progress of your review and avoid reviewing the same check twice.

For more information, see "Reviewing and Commenting Checks " on page 9-48.

Variables View

The variables view displays global variables and where in the source code they are read or written to.

Non Shared Variables. Click the Non-Shared Variables button ![N-SHR] in the Variables View toolbar to show or hide non-shared variables.

Concurrent Access Graph. Click the Show Access Graph button ![icon] in the Variables View toolbar to display a graph of read and write access for the selected variable.
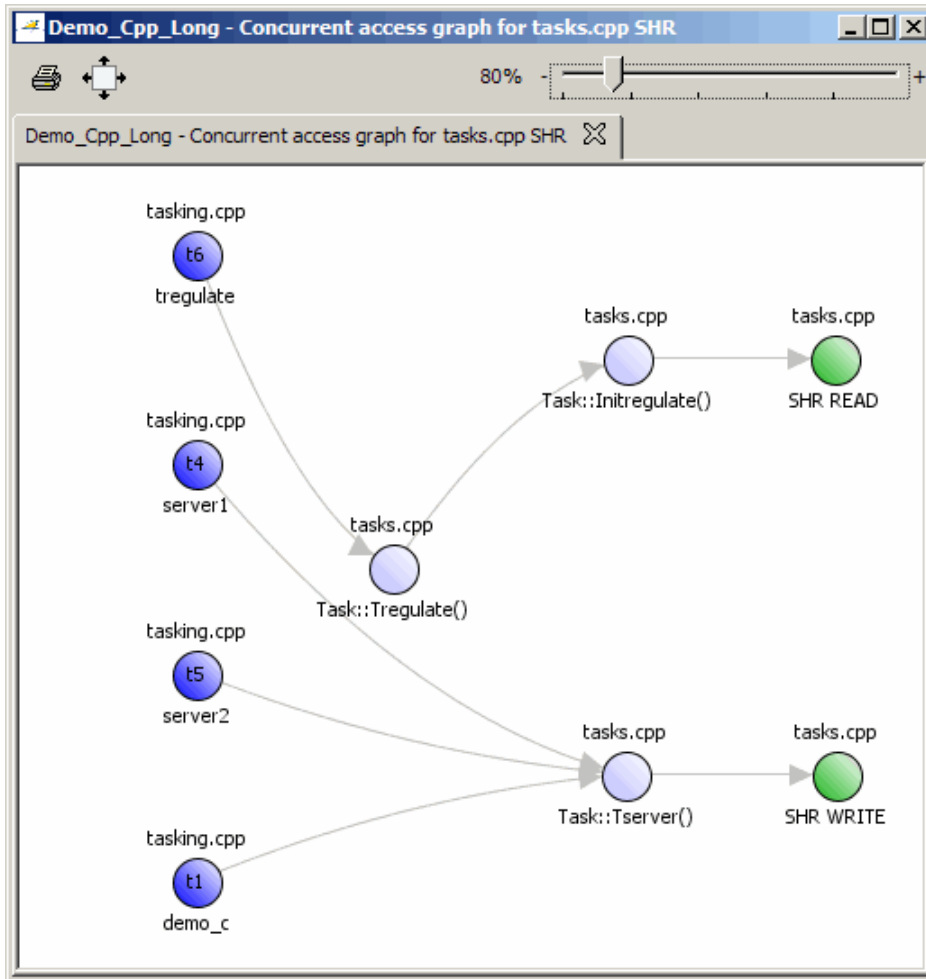
For more information, see "Displaying the Access Sequence for Variables" on page 9-41.

Legend Information. To display the legend for a variable, right-click the variable and select **Show legend**.
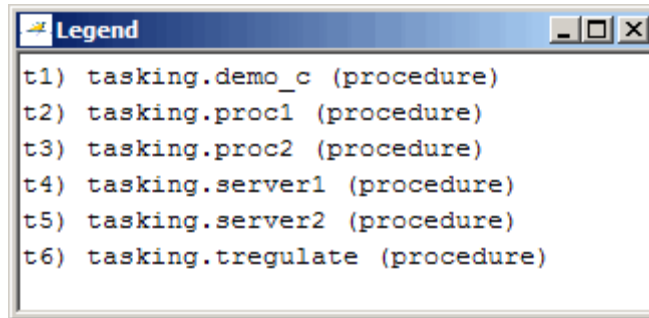
Call Tree View

The call tree view displays the call tree of functions in the source code. You can use the call tree view to easily navigate up and down the call tree.
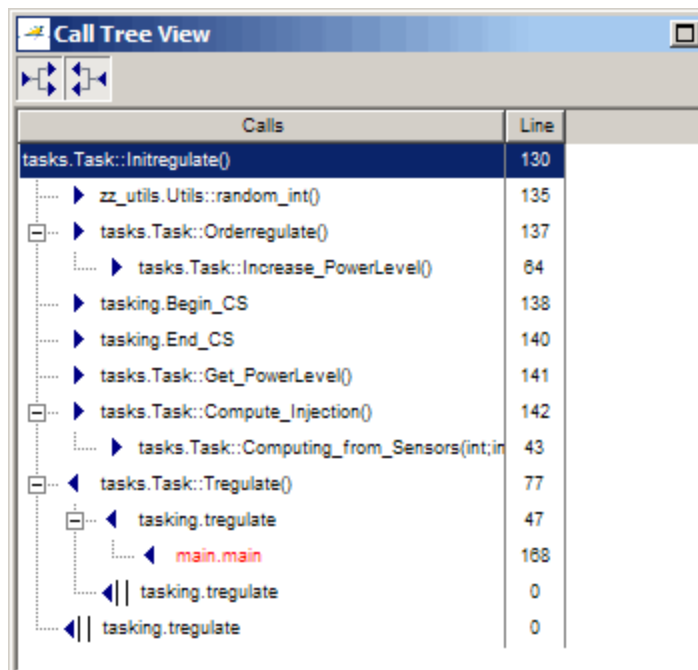
Callers and Callees. Click the buttons in the Call Tree View toolbar to show or hide callers and callees.

Function Definitions. To go directly to the definition of a function, right-click the function call and select **Go to definition**.

Selecting Viewer Mode

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

- In expert mode, you decide how you review the results.

- In assistant mode, PolySpace software guides you through the results.

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

Searching Results in Viewer

You can search your results and source code using the Search feature in the Viewer toolbar.

The Search toolbar allows you to quickly enter search terms, specify search options, and set the scope for your search.
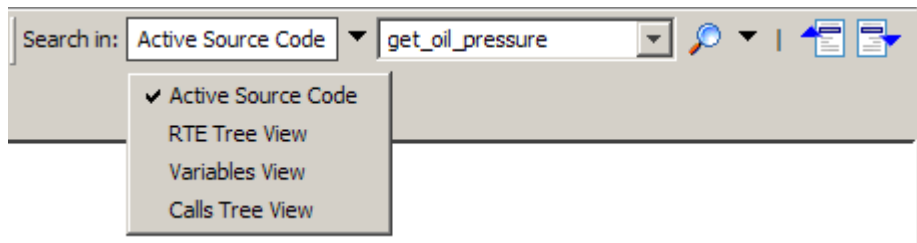
You can limit the scope of your search to one of four Viewer areas:

- Active Source code view
- Procedural entities (RTE) view
- Variables view
- Call tree view

Setting Character Encoding Preferences

If the source files that you want to verify are created on an operating system that uses different character encoding than your current system (for example, when viewing files containing Japanese characters), you receive an error message when you view the source file or run certain macros.

The **Character encoding** option allows you to view source files created on an operating system that uses different character encoding than your current system.

To set the character encoding for a source file:

1 In the Viewer, select **Edit > Preferences**.

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

2 Select the **Character encoding** tab.
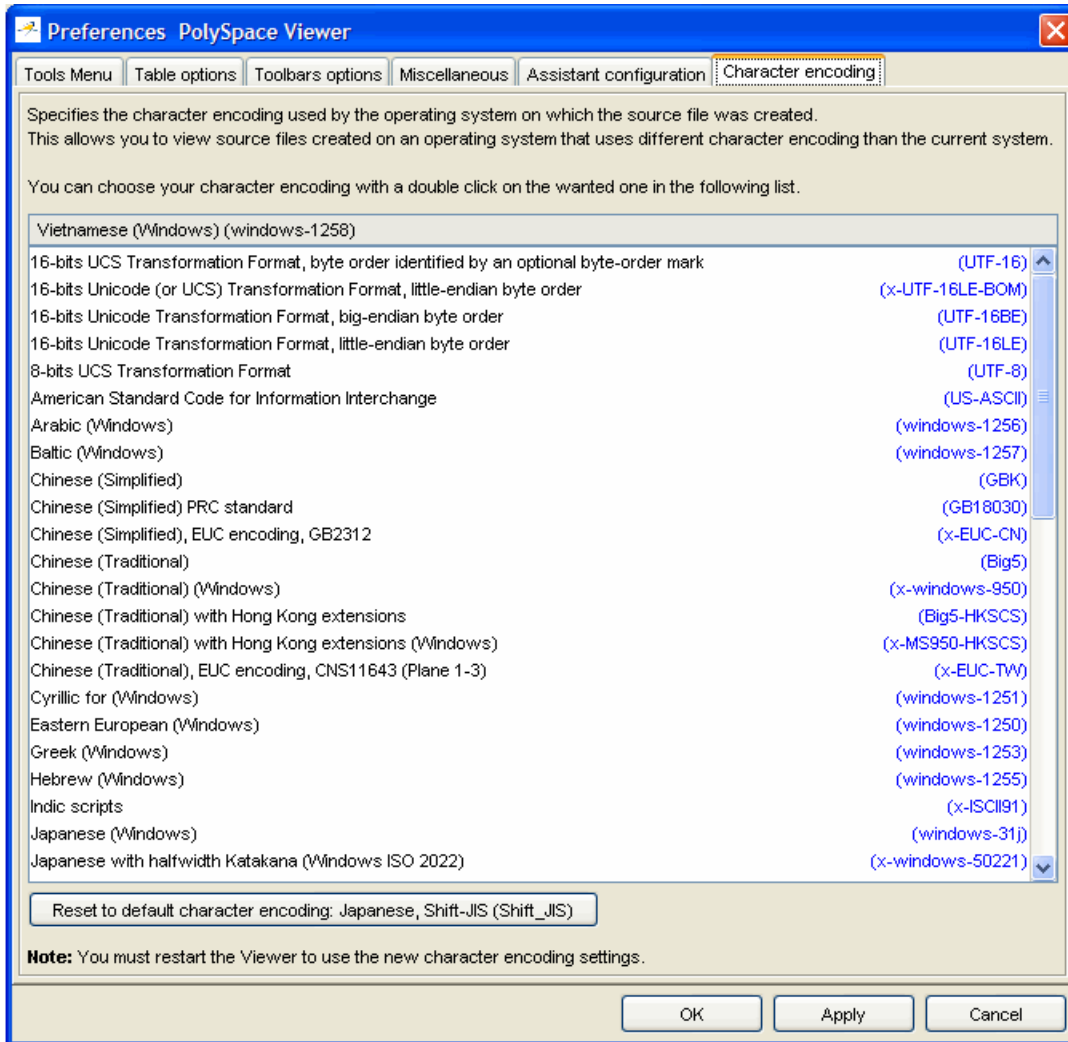
3 Select the character encoding used by the operating system on which the source file was created.

4 Click **OK**.

5 Close and restart the Viewer to use the new character encoding settings.

Reviewing Results in Assistant Mode

What Is Assistant Mode?

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

1 All red checks (an error always occurs).

2 Orange checks known to produce errors in some situations (dark orange). For example, red for one call to a procedure and green for another.

3 Some gray checks (UNR checks that are not nested within dead code blocks).

4 Other orange checks (according to the selected methodology and criterion level).

For more information about methodologies and criterion levels, see "Selecting the Methodology and Criterion Level" on page 9-28.

Switching to Assistant Mode

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

- Click the Viewer mode button [Assistant] .

  The Viewer window toolbar displays controls specific to assistant mode.

  | Methodology for Model Based Designed ▼ | 1   2   3 | ☐ Skip gray checks   ◀◀ ◀ ❀ ▶ ▶▶ |
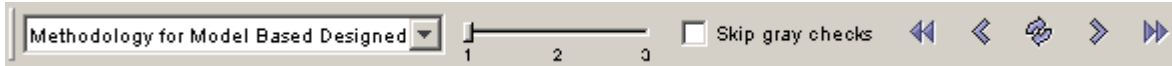
  The controls for assistant mode include:

- A menu to select the review methodology for orange checks.

- A slider to select the criterion level within that methodology.

- A check box for omitting gray checks.

- Arrows for navigating through the reviews.

Selecting the Methodology and Criterion Level

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

1 From the methodology menu, select **Methodology for C++**.

| Methodology for Model Based Designed ▼ |
| Methodology for Ada |
| Methodology for C |
| Methodology for C++ |
| Methodology for Model Based Designed |
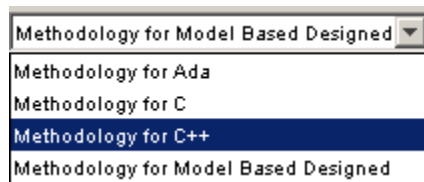
2 Select the appropriate level on the level slider.

| 1   2   3 |
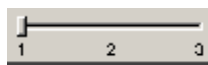
For the configuration Methodology for C++, this table describes the three levels.

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

These three levels correspond to phases of the development process.

Exploring Methodology for C++

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

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

To examine the configuration for **Methodology for C++:**

1 Select **Edit > Preferences**.

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

2 Select the **Assistant configuration** tab.

You see the configuration for Methodology for C++.

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

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

In the lower-left part of the dialog box, the section **Review threshold criterion** contains text that appears in the tooltip for the criterion slider on the Viewer toolbar (in assistant mode).
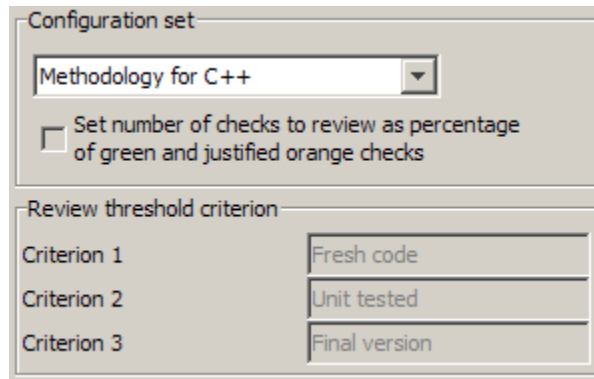
The table describes the criterion names for the configuration Methodology for C++.

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

These names correspond to phases of the development process.

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

Defining a Custom Methodology

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

Custom methodologies can specify either a specific number of orange checks to review, or a minimum percentage of orange checks that must be reviewed. This percentage is calculated as:
(green checks + reviewed orange checks) / (green checks + total orange checks).

To define a custom methodology:

1 Select **Edit > Preferences**.

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

2 Select the **Assistant configuration** tab.

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

The Create a new set dialog box opens.

4 Enter a name for the new configuration set, then click **Enter**.

5 If you would like to review orange checks by percentage, select **Set number of checks to review as percentage of green and justified orange checks**.

6 Enter a name for each criteria level.

7 Enter the total number of checks (or percentage of checks) to review for each type of check, and each criteria level.

Preferences PolySpace Viewer

Tools Menu | Toolbars options | Miscellaneous | Acronyms | Assistant configuration | Character encoding

This configuration menu allows the definition of different configurations to use with the results review assistant.

It allows:
o Creation of a new configuration set,
o Definition of the names for the three different review criteria (used as tool tips of the slider),
o Definition of the number of checks to be reviewed for each category.

By default the number of checks is the maximum number to review and can be:
 - A positive number up to 9999,
 - The word all (or All or ALL) to select all the checks,
 - The word auto (or Auto or AUTO) for automatic check selection (Ada only)

When the checkbox below is selected the number represents the percentage of green and justified orange checks to obtain.
The percentage is evaluated as follows:
 (Green + Orange reviewed) / (Green + Orange).
In this case the value must be a number between 0 and 100.

Configuration set

My Custom Set

☑ Set number of checks to review as percentage of green and justified orange checks

Review threshold criterion

Criterion 1	Quality Level 2
Criterion 2	Quality Level 3
Criterion 3	Quality Level 4

Percentage of green and justified orange checks to obtain

	Criterion 1	Criterion 2	Criterion 3
Common			
ZDV	10	25	100
NIVL	25	50	100
S-OVFL	25	50	100
COR	0	25	100
NIV	0	0	100
F-OVFL	25	25	100
ASRT	0	10	100
C & C++ only			
OBAI	25	50	100
SHF	10	25	100
IDP	0	25	100
NIP	0	25	100
C only			
IRV			
C++ only			
NNT	10	25	100
CPP	10	25	100
FRV	25	50	100
OOP	0	0	100
EXC	0	10	100
Ada only			
EXCP			
POW			

OK | Apply | Cancel

8 Click **OK** to save the methodology and close the dialog box.

Reviewing Checks

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

1 All reds.

2 All blocks of gray checks (the first check in each unreachable function).

Note You can omit gray checks. In the toolbar, select the **Skip gray checks** check box.

---

3 Orange checks, according to the methodology and criterion level and that you select.

To navigate through these checks:

1 Click the forward arrow ⟩.

- The procedural entities view (lower left), expands to show the current check.
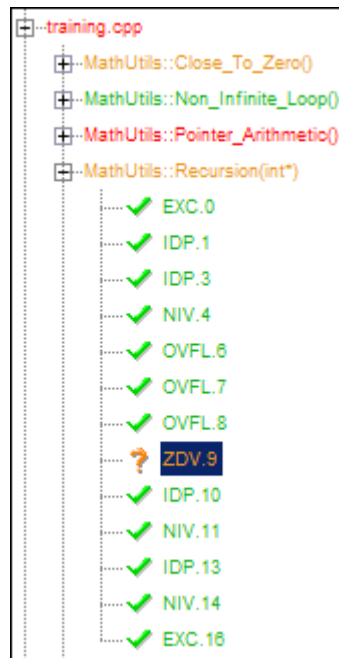
- The source code view (lower right) displays the source code for this check.

- The current check view (upper right) displays information about this check.

Note You can display the calling sequence and track review progress. See "Reviewing Results in Expert Mode" on page 9-37.

---

2 Review the current check.

3 Continue to click the forward arrow until you have gone through all of the checks.

After the last check, a dialog box opens asking if you want to start again from the first check.
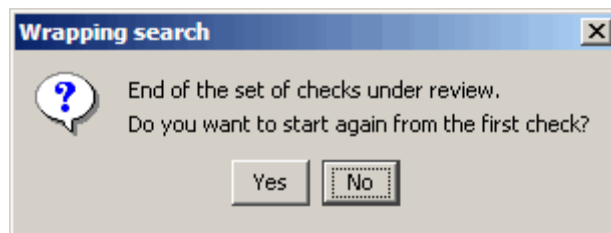
4 Click **No**.

Saving Review Comments

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

To save your review comments:

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

Your comments are saved with the verification results.

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

Reviewing Results in Expert Mode

In this section...
"What Is Expert Mode?" on page 9-37
"Switching to Expert Mode" on page 9-37
"Selecting a Check to Review" on page 9-37
"Displaying the Call Sequence for a Check" on page 9-41
"Displaying the Access Sequence for Variables" on page 9-41
"Filtering Checks" on page 9-43
"Types of Filters" on page 9-43
"Creating a Custom Filter" on page 9-45
"Saving Review Comments" on page 9-46

What Is Expert Mode?

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

Switching to Expert Mode

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

- Click the Viewer mode button:

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

Selecting a Check to Review

To review a check in expert mode:

1 In the procedural entities section of the window, expand any file containing checks.

2 Expand the procedure containing the check that you want to review.
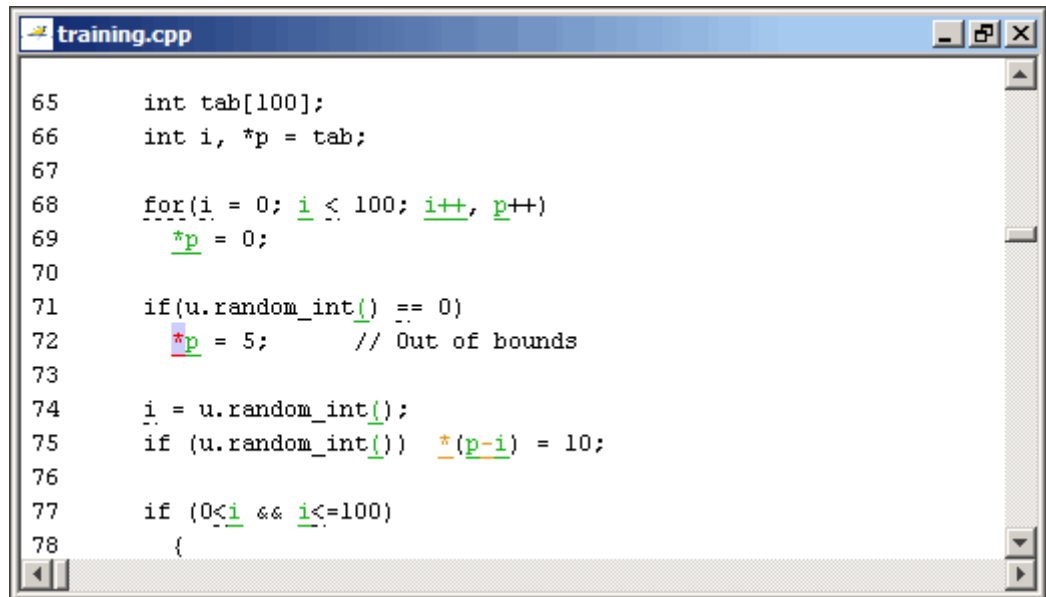
You see a color-coded list of the checks:
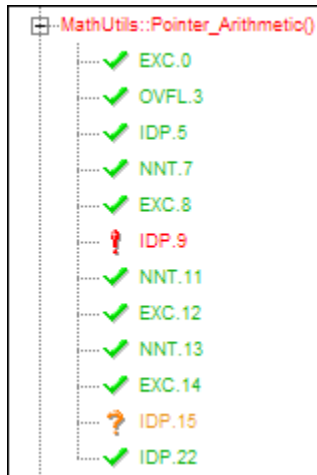
Each item in the list of checks has an acronym that identifies the type of check and a number. For example, `IDP.11`, IDP stands for Illegal Dereferenced Pointer. For more information about different types of checks, see "Check Descriptions"in the *PolySpace Products for C Reference.*

3 Click the check that you want to review.

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

```
65      int tab[100];
66      int i, *p = tab;
67
68      for(i = 0; i < 100; i++, p++)
69        *p = 0;
70
71      if(u.random_int() == 0)
72        *p = 5;        // Out of bounds
73
74      i = u.random_int();
75      if (u.random_int())  *(p-i) = 10;
76
77      if (0<i && i<=100)
78        {
```

4 Place your cursor over any colored check in the code.

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

5 In the code, click the red check.

You see a message box that describes the error.

After you review a check, you can enter comments to describe the results of your review. You can also mark the check as Reviewed to help track your review progress. For more information, see "Tracking Review Progress" on page 9-47.

Displaying the Call Sequence for a Check

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

1 Expand the procedure containing the check that you want to review.

2 Select the check that you want to review.

3 In the toolbar, click the call graph button.

A window displays the call graph.
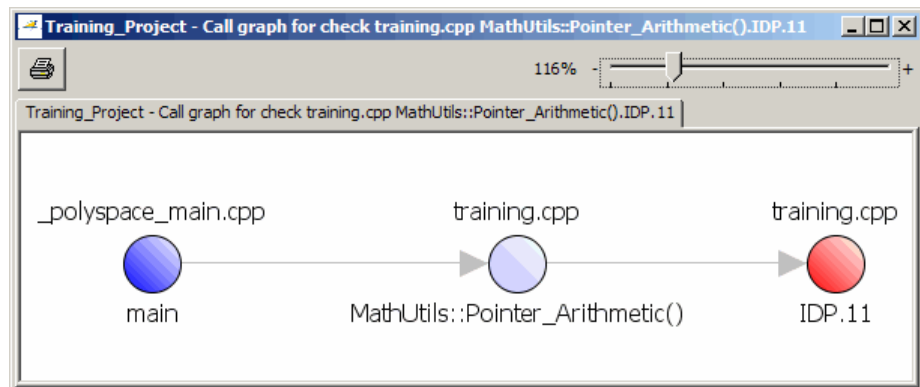
The call graph displays the code associated with the check.

Displaying the Access Sequence for Variables

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

To see the access graph:

1 Select the Variables View.

2 Select the variable that you want to view.

3 In the Variables View toolbar, click the Show Access Graph button.

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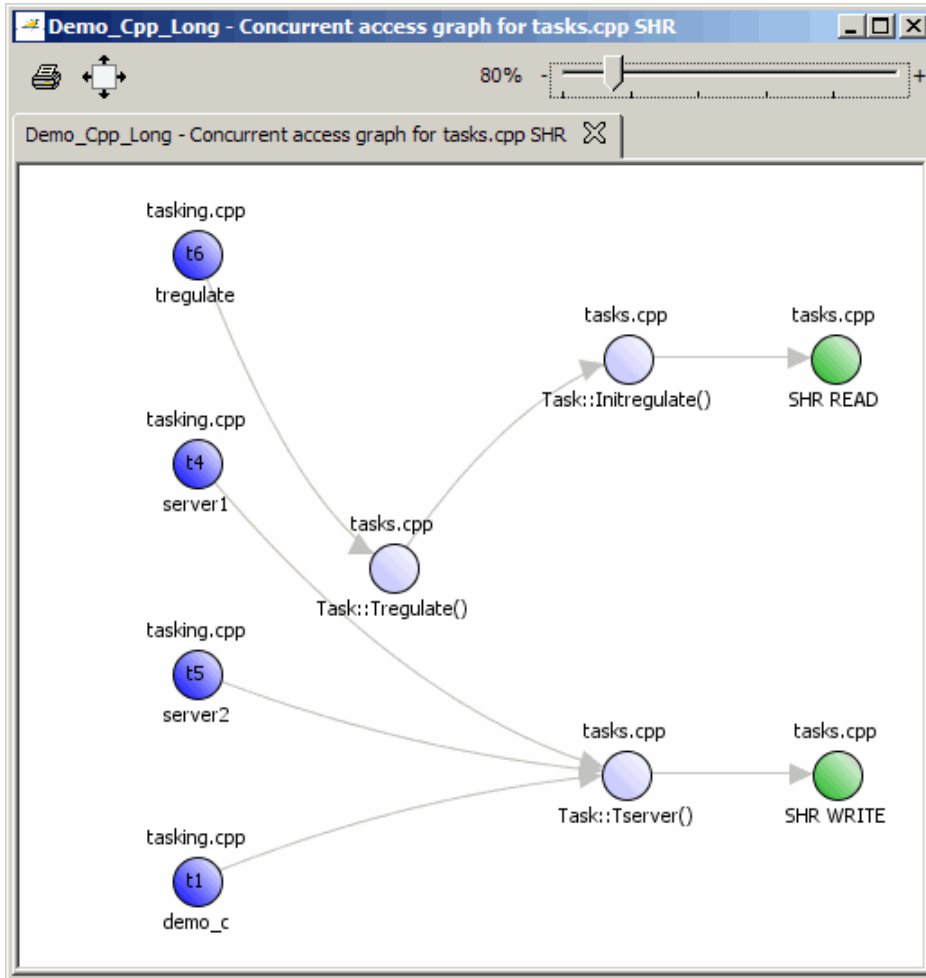
The access graph displays the read and write access for the variable.

4 Click any object in the graph to navigate to that function in the Procedural entities view and Source code view.

Filtering Checks

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

The default filter is `User def`.

To filter checks, select a filter from the filter menu.
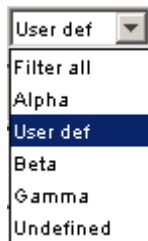
Types of Filters

There are three types of filters:

Individual Filters

You can use individual filters to display or hide a given check category, such as IDP. When a filter is enabled, you do not see that check category. For example, when the IDP filter is enabled, you do not see IDP checks. When the filter is disabled, you see IDP checks.

You can also filter by check color, and can hide checks in nonexecutable procedures.

To enable or disable an individual filter, click the toggle button for that filter on the toolbar.

Tip The tooltip for a filter button indicates to you what filter the button is for and whether the filter is enabled or disabled.

Note When you filter a check category, you do see some red checks with that category.

Composite Filters

Composite filters combine individual filters, allowing you to show or hide groups of checks.

Use This Filter...	To...
Alpha	Show all checks
Beta	Hide NIV, NIVL, NIP, Scalar OVFL, and Float OVFL checks
Gamma	Show red and gray checks
User def	Hide checks as defined in a custom filter that you can modify

Custom Filters

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

By default, the custom filter hides the OBAI, NIV IDP, COR, NIV other, NIP, and NTL checks, as shown in the following figure.
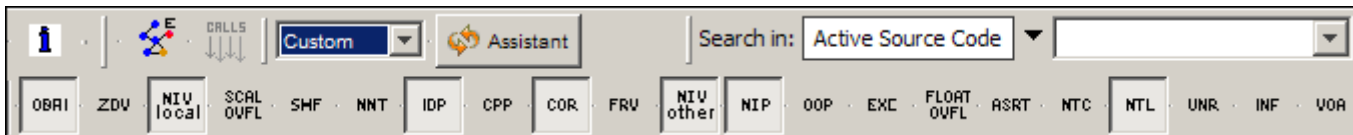
To modify the custom filter, see "Creating a Custom Filter" on page 9-45.

Creating a Custom Filter

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

To modify the custom filter:

1 From the composite filters menu, select `User def`.

2 Select **Edit > Custom filters**.

The **Custom filter setup** dialog box opens.
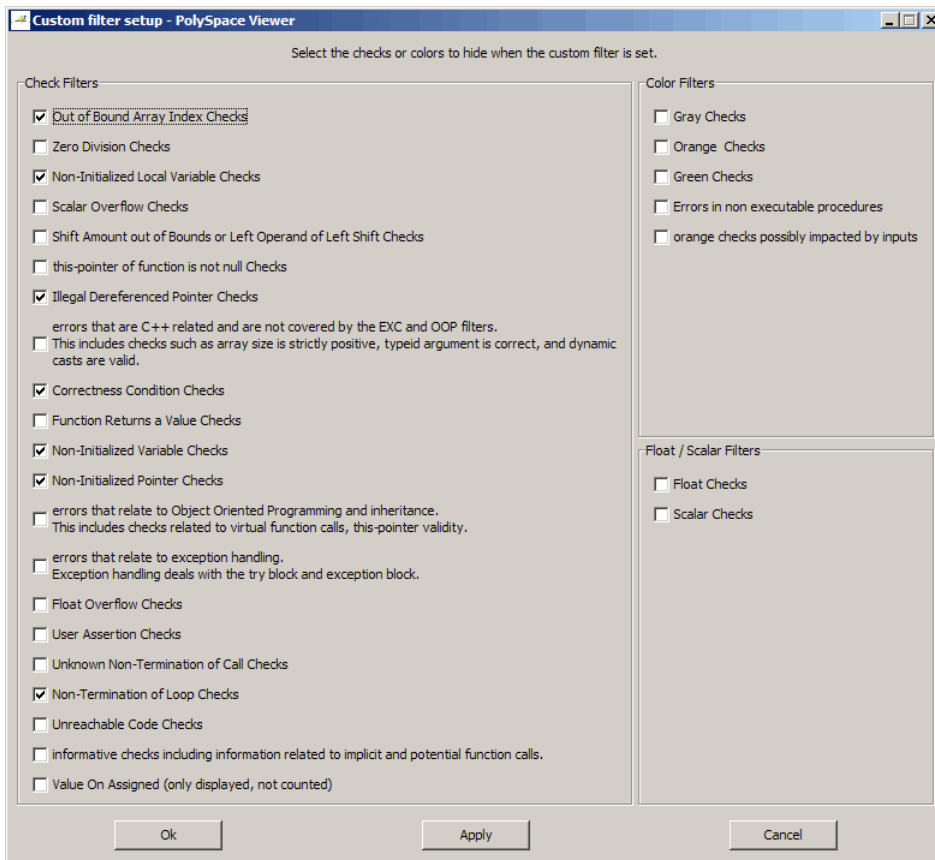
3 Clear the filters for the checks that you want to display. For example, if you clear the **Out of Bound Array Index Checks** box, you see the OBAI checks.

Note You do not have to change any of the selections for this tutorial.

---

4 Select the filters for the checks that you do not want to display.

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

PolySpace software saves the custom filter definition in the Viewer preferences.

Saving Review Comments

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

To save your review comments:

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

Your comments are saved with the verification results.

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

Tracking Review Progress

In this section...

Checking Coding Review Progress

When you select a check in either Assistant or Expert mode, the Coding review progress view (upper-left part of the Viewer window), displays statistics about the review progress for that category and severity of error.

Coding review progress	Count	Progress
num IDP reviewed / num IDP to review (Red)	0/1	0
num reviewed / num to review (Red)	0/1	0
Software reliability indicator	88/102	86

The **Count** column displays a ratio and the **Progress** column displays the equivalent percentage.

The first row displays the ratio of reviewed checks to total checks that have the same color and category of the current check. In this example, the first row displays the ratio of reviewed red IDP checks to total red IDP errors in the project.

The second row displays the ratio of reviewed checks to total checks that have the color of the current check. In this example, this is the ratio of red errors reviewed to total red errors in the project.

The third row displays the ratio of the number of green checks to the total number of checks, providing an indicator of the reliability of the software.

When you select the Reviewed checkbox for the check, the software updates the ratios of errors reviewed to total errors in the **Coding review progress** part of the window.

Coding review progress	Count	Progress
num IDP reviewed / num IDP to review (Red)	1/1	100
num reviewed / num to review (Red)	1/1	100
Software reliability indicator	88/102	86

Reviewing and Commenting Checks

When reviewing checks in either Assistant or Expert mode, you can mark checks **Reviewed**, and enter comments to describe the results of your review. This helps you track the progress of your review and avoid reviewing the same check twice.

To reviewed and comment a check:

1 Select the check that you want to review.

The Selected check view (upper-right part of the Viewer window) displays information about the current check.

```
training.cpp / MathUtils::Pointer_Arithmetic() / line 72 / column 4

   ⊞         *p = 5; // Out of bounds

☑ NOW ▼            ▼  📝  Fix Now

Error : pointer is outside its bounds
    dereference of variable 'p' (pointer to int 32, size: 32 bits):
         pointer is not null
         points to 4 bytes at offset 400 in allocated buffer of 400 bytes
         may point to variable or field of variable in: {MathUtils::Pointer_
```
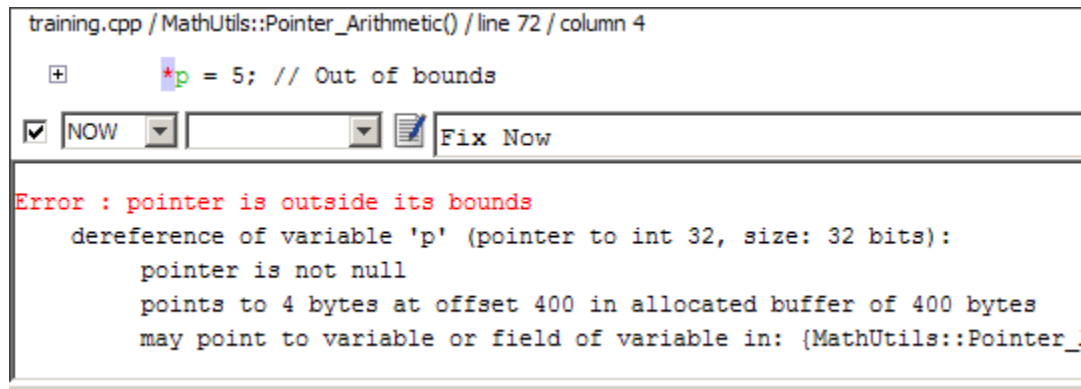
2 After you review the check, select an acronym to describe the check in the Predefined acronyms menu:

- **NOW** – Bug to fix now.

- **NXT** – Bug to fix in Next Release

- **ROB** – Robustness Issue

- **DEF** – Defensive Code

- **MIN** – Minor quality issue

- **OTH** – Other

Note You can also define your own acronyms. See "Defining Custom Acronyms " on page 9-49.

3 In the comment box, enter additional information about the check.

4 Select the check box to indicate that you have reviewed this check.

The software updates the ratios of errors reviewed to total errors in the **Coding review progress** view of the Viewer window.

Coding review progress	Count	Progress
num IDP reviewed / num IDP to review (Red)	1/1	100
num reviewed / num to review (Red)	1/1	100
Software reliability indicator	88/102	86

Defining Custom Acronyms

In addition to the Predefined Acronyms for reviewing checks, you can define your own acronyms. Once you define an acronym, you can select it from the user-defined acronym menu in the Selected check view.

To define custom acronyms:

1 Select **Edit** > **Preferences**.

2 Select the **Acronyms** tab.
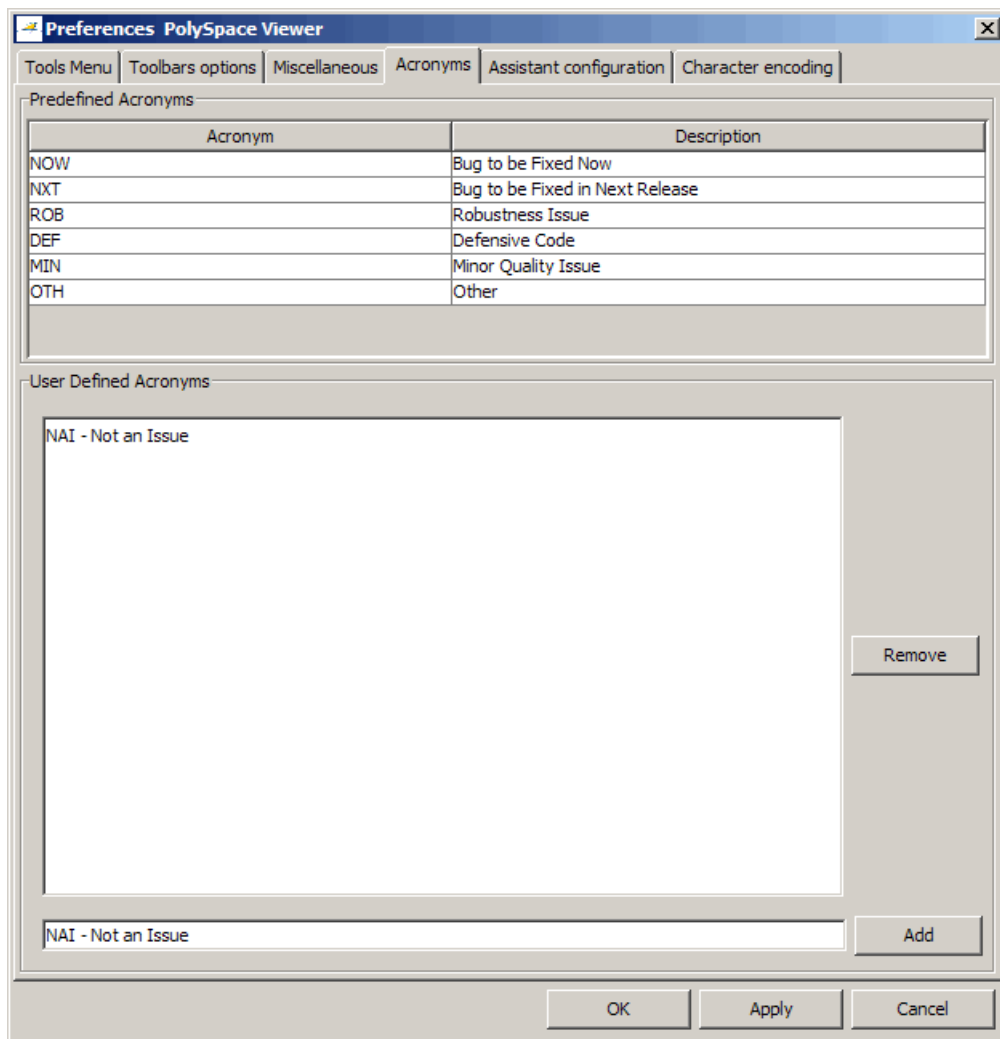
3 Enter your new acronym at the bottom of the dialog box, then click **Add**.

The new acronym appears in the User Defined Acronyms list.

4 Click **OK** to save your changes and close the dialog box.

When reviewing checks, you can select the new acronym from the user-defined acronyms menu in the Selected check view.

Tracking Reviewed Checks in Procedural Entities View

The **Procedural entities** view in the Viewer displays which checks you have reviewed and the predefined acronym you used to describe each check.

Procedural entities	!	✗	?	✓	Line	Col	8	Details	Reviewed	Acronym
⊞···tasks.h				2	1		100	tasks.h	☐	
⊟···training.cpp	4	7	5	62	1		94	training.cpp	☐	
⊞···MathUtils::Close_To_Zero()			3	8	12	16	73	training.cpp	☐	
⊞···MathUtils::Non_Infinite_Loop()				5	39	15	100	training.cpp	☐	
⊟···MathUtils::Pointer_Arithmetic()	1			8	61	16	100	training.cpp	☐	
├···✓ EXC.0				1	61			function does no...	☐	
├···✓ OVFL.3				1	68	23		operation [+] on ...	☐	
├···✓ NNT.7				1	71	17		this-pointer of ra...	☐	
├···✓ EXC.8				1	71	17		call to random_in...	☐	
├···❗ IDP.9	1				72	4		Error : pointer is ...	☑	NOW
├···✓ NNT.11				1	74	18		this-pointer of ra...	☐	
├···✓ EXC.12				1	74	18		call to random_in...	☐	
├···✓ NNT.13				1	75	18		this-pointer of ra...	☐	
└···✓ EXC.14				1	75	18		call to random_in...	☐	
⊞···MathUtils::Recursion(int*)			1	5	104	16	83	training.cpp	☐	
⊞···MathUtils::Recursion_2(int*)				2	98	16	100	training.cpp	☐	

Tip If you do not see the Reviewed column, resize the **Procedural entities** view to display the column. If it does not appear, right click the **Procedural entities** column heading and select **Reviewed**.

You can select the **Reviewed** check box to mark a check as reviewed. Selecting this check box also automatically:

- Selects the check box for that check in the current check view (upper-right part of the window).

- Updates the counts in the coding review progress view (upper-left part of the window).

Importing and Exporting Review Comments

In this section...

Reusing Review Comments

After you have reviewed verification results on a module, you can reuse your review comments with subsequent verifications of the same module. This allows you to avoid reviewing the same check twice, or to compare results over time.

The PolySpace Viewer allows you to either:

- Export review comments from the current results to another set of results.
- Import review comments from another set of results into the current results.

You can also generate a report that compares the source code and verification results from two verifications, and highlights differences in the results.

Note If the code has changed since the previous verification, the imported comments may not be applicable to your current results. For example, the justification for an orange check may no longer be relevant to the current code. Open the Import/Export Report to see changes that affect your review comments.

Exporting Review Comments to Other Verification Results

After you have reviewed verification results, you can export your review comments for use with other verifications of the same module, allowing you to avoid reviewing the same check twice.

Caution   The comments you export replace any existing comments in the selected results.

To export review comments to other verification results:

1 Select **File > Export checks and comments**.

2 Navigate to the folder containing the other results file.

3 Select the results (.RTE) file, then click **Open**.

   The review comments from the current results are exported into the selected results.

Note   If the code has changed between the two verifications, the exported comments may not be applicable to the other results. For example, the justification for an orange check may no longer be relevant to the current code.

Importing Review Comments from Previous Verifications

If you have previously reviewed verification results for a module and saved your comments, you can import those comments into the current verification, allowing you to avoid reviewing the same check twice.

Caution   The comments you import replace any existing comments in the current results.

To import review comments from a previous verification:

1 Open your most recent verification results in the Viewer.

2 Select **File > Import checks and comments**.

3 Navigate to the folder containing your previous results.

4 Select the results (.RTE) file, then click **Open**.

The review comments from the previous results are imported into the current results.

Once you import checks and comments, the **go to next check** icon in assistant mode will skip any reviewed checks, allowing you to review only checks that you have not reviewed previously. If you want to view reviewed checks, click the **go to next reviewed check** icon.

Note If the code has changed since the previous verification, the imported comments may not be applicable to your current results. For example, the justification for an orange check may no longer be relevant to the current code.

Viewing Checks and Comments Report

Importing review comments from a previous verification can be extremely useful, since it allows you to avoid reviewing checks twice, and to compare verification results over time.

However, if your code has changed since the previous verification, or if you have upgraded to a new version of the software, the imported comments may not be applicable to your current results. For example, the color of a check may have changed, or the justification for an orange check may no longer be relevant to the current code.

The Import/Export checks and comments report allows you to compare the source code and verification results from a previous verification to the current verification, and highlights differences in the results.

To view the Import/Export checks and comments report:

1 Open your most recent verification results in the Viewer.

2 Select **File > Import checks and comments**.

3 Select the results (.RTE) file from your previous verification, then click **Open**.

The review comments from the previous results are imported into the current results.

4 Select **File > Open Import/Export Report**.

The Import/Export checks and comments report opens, highlighting differences in the two results, such as unmatched lines and changes to the color of checks.
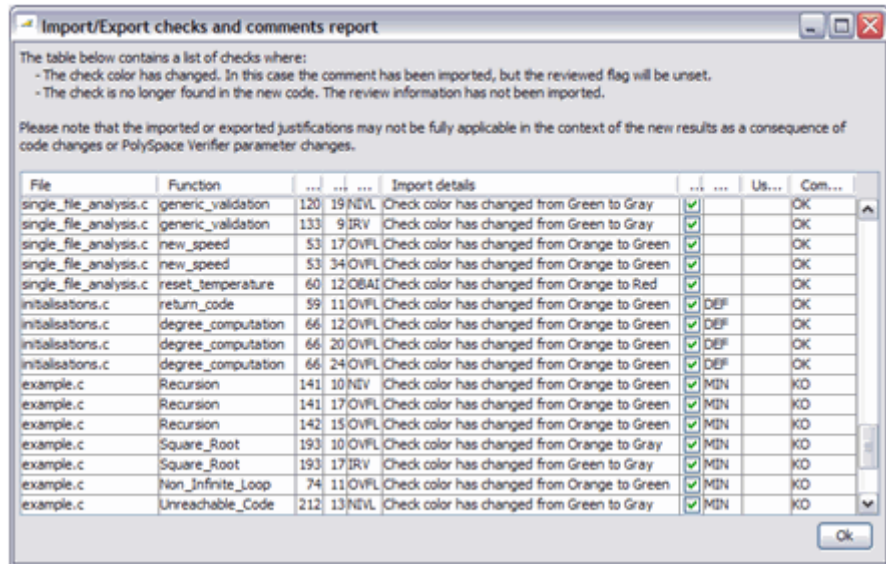
If the color of a check changes, the previous review comments are imported, but the check is not marked as reviewed.

If a check no longer appears in the code, the report highlights the change, but the software does not import any comments on the check.

Generating Reports of Verification Results

In this section...
"PolySpace Report Generator Overview" on page 9-58
"Generating Verification Reports" on page 9-59
"Automatically Generating Verification Reports" on page 9-62
"Generating Excel Reports" on page 9-62

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 MISRA C++ or JSF++ 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.

The PolySpace Report Generator allows you to generate verification reports in the following formats:

- HTML

- PDF

- RTF

- DOC (Microsoft® Word)

- XML

Note Microsoft Word is not available on UNIX platforms. RTF format is used instead.

Generating Verification Reports

You can generate reports for any verification results using the PolySpace Report Generator.

To generate a verification report:

1 In the Viewer, open your verification results.

2 Select **Reports > Run Report**.

The Run Report dialog box opens.
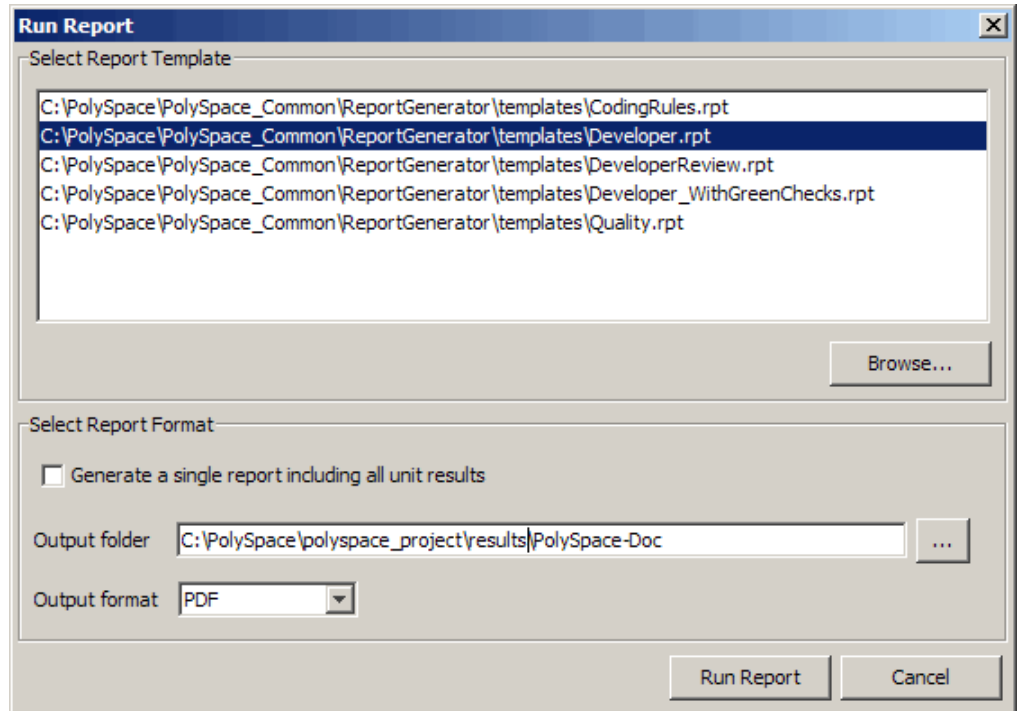
3 In the Select Report Template section, select the type of report that you want to run.

4 If your results are part of a unit-by-unit verification, you can generate a report for the current unit results, or for the entire project. Select **Generate a single report including all unit results** to combine all unit results in the report.

5 Select the Output folder in which to save the report.

6 Select the Output format for the report.

7 Click **Run Report**.

The software creates the specified report.

> **Note** If you generate an RTF format report on a Linux system, the software does not open the report at the end of the generation process.

Running the Report Generator from the Command Line

You can also run the Report Generator, with options, from the command line, for example:

```
C:\>\PolySpace\PolySpace_Common\ReportGenerator\wbin\report-generator
-template path -format type -results-dir folder_paths
```

For information about the available options, see the following sections.

-template *path*. Specify the *path* to a valid Report Generator template file, for example,

```
C:\PolySpace\PolySpace_Common\ReportGenerator\templates\Developer.rpt
```

Other supplied templates are CodingRules.rpt, Developer_WithGreenChecks.rpt, DeveloperReview.rpt, and Quality.rpt.

-format *type*. Specify the format *type* of the report. Use HTML, PDF, RTF, WORD, or XML. The default is RTF.

-help or -h. Displays help information.

-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, then 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"
```

where *folder1, folder2, ...* are the file paths to the folders that contain the results of your verifications (normal or unit-by-unit). For example,

```
"C:\Results1,C:\Recent\results,C:\Old"
```

If you do not specify a folder path, the software uses verification results from the current folder.

Automatically Generating Verification Reports

You can specify that PolySpace software automatically generate reports for each verification using an option in the Launcher .

To automatically generate reports for each verification:

1 In the Launcher, open your project.

2 In the Analysis options section of the Launcher window, expand **General**.

   You see the General options.

3 Select **Report Generation**.

4 Select the **Report template name**.

5 Select the **Output format** for the report.

6 Save your project.

Generating Excel Reports

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

> **Note** Excel reports do not use the PolySpace Report Generator.

To generate an Excel report of your verification results:

1 In your results folder, navigate to the `PolySpace-Doc` folder. For example:`polypace_project\results\PolySpace-Doc`.

The folder should have the following files:

```
Example_Project_Call_Tree.txt
Example_Project_RTE_View.txt
Example_Project_Variable_View.txt
Example_Project-NON-SCALAR-TABLE-APPENDIX.ps
PolySpace_Macros.xls
```

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

2 Open the macros file `PolySpace_Macros.xls`.

You see a security warning dialog box.

3 Click **Enable Macros**.

A spreadsheet opens. The top part of the spreadsheet looks like the following figure.
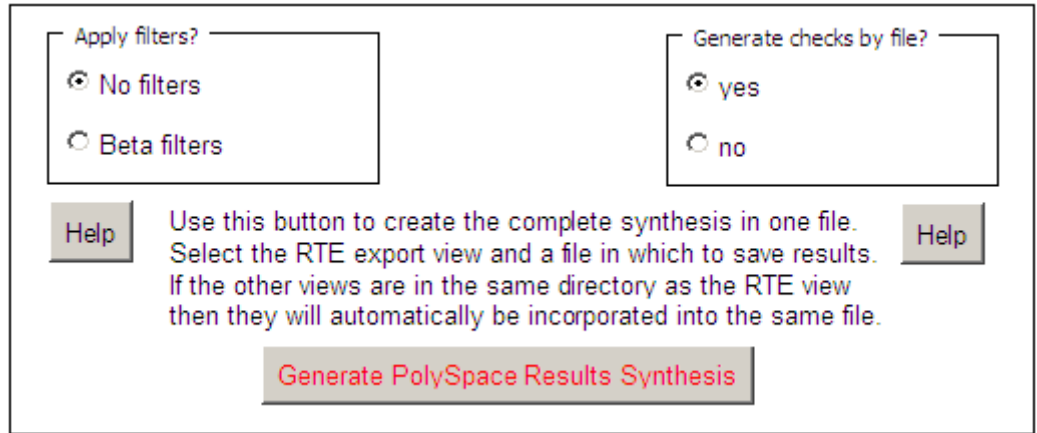
4 Specify the report options that you want, then click **Generate PolySpace Results Synthesis**.

The synthesis report combines the RTE, call tree, and variables views into one report.

The **Where is the PolySpace RTE View text file** dialog box opens.

5 In **Look in**, navigate to the PolySpace-Doc folder in your results folder. For example:polypace_project\results\PolySpace-Doc.

6 Select *Project*_RTE_View.txt.

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

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

8 Keep the default file name and file type.

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

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

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

	A	B	C	D	E	F
1		**RTE Statistics**				
2	**Check category**	**Check detail**	**R**	**O**	**Gy**	**Gr**
3	OBAI	Out of Bounds Array Index	0	0	0	0
4	NIVL	Uninitialized Local Variable	0	0	0	19
5	IDP	Illegal Dereference of Pointer	1	2	0	5
6	NIP	Uninitialized Pointer	0	0	0	12
7	NIV	Uninitialized Variable	0	3	0	0
8	IRV	Initialized Value Returned	0	0	0	0
9	COR	Other Correctness Conditions	0	0	0	2
10	ASRT	User Assertion Failure	0	0	0	1
11	POW	Power Must Be Positive	0	0	0	0
12	ZDV	Division by Zero	0	1	0	3
13	SHF	Shift Amount Within Bounds	0	0	0	0
14	OVFL	Overflow	0	3	0	4
15	UNFL	Underflow	0	1	0	6
16	UOVFL	Underflow or Overflow	0	3	0	2
17	EXCP	Arithmetic Exceptions	0	0	0	0
18	NTC	Non Termination of Call	0	0	0	0
19	k-NTC	Known Non Termination of Call	0	0	0	0
20	NTL	Non Termination of Loop	0	0	0	0

Microsoft Excel - Training_Project-Synthesis.xls

File   Edit   View   Insert   Format   Tools   Data   Window   Help

RTE Checks Sheet 1 / Launching Options \ **Check Synthesis**

Using PolySpace Results

Review Runtime Errors: Fix Red Errors

All Runtime Errors highlighted by PolySpace verification are determined by reference to the language standard, and are sometimes implementation dependant — that is, they may be acceptable for a particular compiler but unacceptable according to the language standard.

Consider an overflow on a type restricted from -128 to 127. The computation of 127+1 cannot be 128, but depending on the environment a "wrap around" might be performed to give a result of -128.

This result is mathematically incorrect, and could have serious consequences if, for example, the computation represents the altitude of a plane.

By default, PolySpace verification does not make assumptions about the way you use a variable. Any deviation from the recommendations of the language standard is treated as a red error, and must therefore be corrected.

PolySpace verification identifies two kinds of red checks:

- Red errors which are *compiler-dependant* in a specific way. A PolySpace option may be used to allow particular compiler specific behavior. An example of a PolySpace option to permit compiler specific behavior is the option to force "IN/OUT" ADA function parameters to be initialized.

Examples in C include options to deal with constant overflows, shift operation on negative values, and so on.

- You must fix all other red errors. They are bugs.

Most of the bugs you find are easy to correct once the software identifies them. PolySpace verification identifies bugs regardless of their consequence, or how difficult they may be to correct.

Using Range Information in the Viewer

Viewing Range Information

You can see range information associated with variables and operators within the source code view. Place the cursor over an operator or variable. A tooltip message displays the range information, if it is available.

Note The displayed range information represents a superset of dynamic values, which the software computes using static methods.

If a line of code is entirely the same color, selecting (clicking) the line opens the Expanded Source Code window. Place your cursor over the required operator or variable in this window to view range information. In addition, you can select the line in the Expanded Source Code window to display error or warning messages (along with range information) in the selected check view.

In the source code view, if a line of code contains different colored checks, then selecting a check displays the error or warning message along with range information in the selected check view.

Note Computing range information for reads and operators may take a long time. You can reduce verification time by limiting the amount of range information displayed in verification results. See "-less-range-information" in the *PolySpace Products for C++ Reference Guide*.

Interpreting Range Information

The software uses the following syntax display range information of variables:

```
name (data_type) : [min1 .. max1] or [min2 .. max2] or [min3 .. max3] or exact value
```

In the following example,

```
30      {
31        int temp;
32        PowerLevel = -10000;
33
34        RTE        assignment of variable 'PowerLevel' (int 32): -10000
35
```

the tooltip message indicates the variable `PowerLevel` is a 32-bit integer with the value −10000.

In the next example,

```
140
141     *depth = *depth + 1;
142     advance = 1.0f/(float)(*depth);   /* potential division by zero */
143
144            assignment of variable 'advance' (float 32): [-1.0001 .. -4.6566E⁻¹⁰] or [1.9999E⁻² .. 3.3334E⁻¹]
```

the tooltip message indicates that the variable `advance` is a 32-bit `float` that lies between either −1.0001 and −4.6566E-10 or 1.9999E-2 and 3.3334E-1

The tooltip message also indicates whether the variable occupies the full range:

```
37
38        temp = read_on_bus();
39        switch(temp)
                          returned value of read_on_bus (int 32): full-range [-2³¹ .. 2³¹-1]
40            {
```
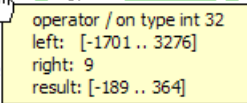
The tooltip message indicates that the returned value of the function read_on_bus is a 32-bit integer that that occupies the full range of the data type, -2147483648 to 2147483647.

With operators, the software displays associated information. Consider the following example:

```
50
51    static s32 new_speed(s32 in, s8 ex_speed, u8 c_speed)
52    {
53      return (in / 9 + ((s32)ex_speed + (s32)c_speed) / 2 );
54    }
                    operator / on type int 32
55                  left:  [-1701 .. 3276]
56    static char re   right: 9              v3)
                       result: [-189 .. 364]
57    {
```

The tooltip message for the division operator / indicates that the :

- Operation is performed on 32-bit integers

- Dividend (left) is a value between −1701 and 3276

- Divisor (right) is an exact value, 9

- Quotient (result) lies between −189 and 364

Note You can run a pass0 (Software Safety Analysis level 0) verification to produce results quickly. See "-from verification-phase" and "-to verification-phase" in the *PolySpace Products for C++ Reference Guide*. However, with a pass0 verification, the software generates range information that is either a constant or full-range for the data type of the expression.

Diagnosing Errors with Range Information

You can use range information to diagnose errors. Consider the function table_loop() in the following example:

There is one red check in the **Procedural entities** view:

Clicking the red check, `OBAI.62` in the **Procedural entities** view or `[` on line 38 in the source code view, displays an error message and range information in the selected check view:

The error message shows that the array size is 4, but the array index is negative, with a value of –1.

Placing the cursor over the subtraction operator - in the source code view shows the following:

The operands are 0 and 1 respectively, which gives a negative result.

Placing the cursor over j reveals the reason for the negative index:

```
36      // Error: array index is outside its bounds
37      if (u.random_int())
38        array[j-1]->Draw();
39                    variable 'j' (int 32): 0
40      return  true;
41   }
42
```

j has the value 0.

Using Pointer Information in the Viewer

Within the source code view, you can see information about pointers to variables or functions. If you place the cursor over a pointer, dereference character ([, ->, *), function call, or function declaration, a tooltip message displays pointer information. For example:

```
102       if(get_oil_pressure() > 0)
103         {
104           *p = 5; /* Out of bounds */
105         }  dereference of variable 'p' (pointer to int 32, size: 32 bits):
106       else      pointer is not null
107         {        points to 4 bytes at offset 400 in allocated buffer of 400 bytes
108               may point to variable or field of variable in: {Pointer_Arithmetic:array}
109         }
110       }
```

If you click the pointer check (IDP, NIP), dereference character, function call, or function declaration, the software also displays the pointer information in the selected check view.

```
example.c / Pointer_Arithmetic / line 104 / column 10

   ⊞       *p = 5; /* Out of bounds */

□ [   ▼] [      ▼] 📝 [

Error : pointer is outside its bounds
    dereference of variable 'p' (pointer to int 32, size: 32 bits):
        pointer is not null
        points to 4 bytes at offset 400 in allocated buffer of 400 bytes
        may point to variable or field of variable in: {Pointer_Arithmetic:array}
```
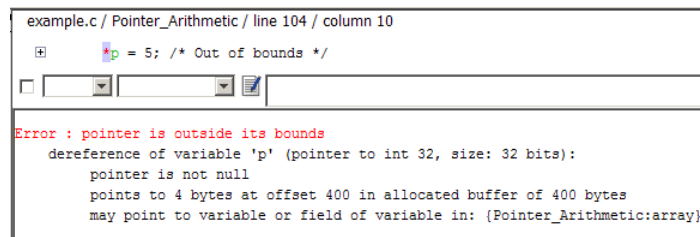
For a pointer to a variable, on separate lines in the tooltip message, the software displays:

• The pointer name, data type of the variable, and size of the data type in bits.

- A comment that indicates whether the pointer is null, is not null, or may be null. See also "Messages on Dereferences" on page 9-76.

- The number of bytes that the pointer accesses, the offset position of the pointer in the allocated buffer, and the size of this buffer in bytes.

- A comment that indicates whether the pointer *may* point to dynamically allocated memory.

- The names of the variables at which the pointer may point.

Note Tooltip messages display only lines that contain meaningful information. For example, when a pointer is initialized by the main generator, the tooltip does not display lines for offset and aliases.

For pointer to a function, the software displays:

- The pointer name.

- A comment that indicates whether the pointer is null, is not null, or may be null.

- The names of the functions that the pointer may point to, and a comment indicating whether the functions are well or badly typed (whether the number or types of arguments in a function call are compatible with the function definition).

Note Computing pointer information may take a long time. You can disable the display of pointer information by selecting the option no-pointer-information. See "-no-pointer-information" in the *PolySpace Products for C++ Reference Guide*.

You can use pointer information when analyzing, for example, red and orange IDP and NIP checks. In the following example, placing the cursor over the orange check shows that offset position may lie outside the bounds of the pointer.

```
112     i = get_bus_status();

113

114     if (i >= 0)   {j(p-i) = 10;}

115                       dereference of expression (pointer to int 32, size: 32 bits):
116     if ((0 <= i) &&       pointer is not null
117       {                   points to 4 bytes at offset [? .. 400] in allocated buffer of 400 bytes
118         p = p - i;        may point to variable or field of variable in: {Pointer_Arithmetic:array}
119         *p = 5;         /   safe pointer access   /
120       }
```

Messages on Dereferences
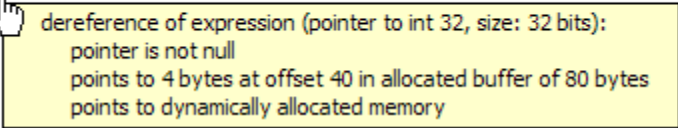
Tooltip messages on dereferences give information about the expression that is dereferenced.

Consider the following code:

```
int *p = (int*) malloc ( sizeof(int) * 20 );
p[10] = 0;
```

In the verification results, the tooltip on "[" displays information about the expression that is dereferenced.

```
23
24      int *p = (int*) malloc ( sizeof(int) * 20 );
25      p[10] = 0;
26           dereference of expression (pointer to int 32, size: 32 bits):
27    }           pointer is not null
                  points to 4 bytes at offset 40 in allocated buffer of 80 bytes
28                points to dynamically allocated memory
29
```

On `p[10]`, the expression dereferenced is `p + 10 * sizeof(int)`, so the tooltip message displays the following:

- The dereferenced pointer is at offset `40`.

   Explanation: `p` has offset `0`, so `p+10` has offset `40` (`10 * sizeof(int)`).

- The dereferenced pointer is not null.

   Explanation: `p` is null, but `p+10` is not null ($0+40 \neq 0$).

The software reports an orange dereference check (IDP) on `p[10]` because `malloc` may have put `NULL` into `p`. In that case, `p + 10 * sizeof(int)` is not null, but it is not properly allocated.

Red Checks Where Gray Checks were Expected

By default, PolySpace continues verification when it finds a red error. This is used to deal with two primary circumstances:

- A red error appears in code which was expected to be dead code.

- A red error appears which was expected, but the verification is required to continue.

PolySpace performs an upper approximation of variables. Consequently, it may be true that PolySpace verifies a particular branch of code as though it was accessible, despite the fact that it could never be reached during "real life" execution. In the example below, there is an attempt to compare elements in an array, and PolySpace is not able to conclude that the branch was unreachable. PolySpace may conclude that an error is present in a line of code, even when that code cannot be reached.

Consider the figure below.
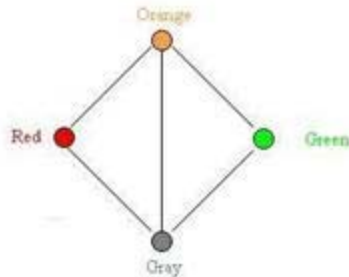
As a result of imprecision, each color shown can be approximated by a color immediately above it in the grid. It is clear that green or red checks can be approximated by orange ones, but the approximation of gray checks is less obvious.

During PolySpace verification, data values possible at execution time are represented by supersets including those values - and possibly more besides.

Gray code represents a situation where no valid data values exist. Imprecision means that such situation can be approximated

- by an empty superset;
- by a nonempty super set, members of which may generate checks of any color.

And hence PolySpace cannot be guaranteed to find all dead code in a verification.

However, there is no problem in having gray checks approximated by red ones. Where any red error is encountered, all instructions which follow it in the relevant branch of execution are aborted as usual. At execution time, it is also true that those instructions would not be executed.

Consider the following example:

```
if (condition) then action_producing_a_red;
```

After the "if" statement, the only way execution can continue is if the condition is false; otherwise a **red check** would be produced. Therefore, after this branch the condition is always false. For that reason, the code verification continues, even with a specific error. Remember that this propagates values throughout your application. None of the execution paths leading to a run-time error will continue after the error and if the red check is a real problem rather than an approximation of a gray check, then the verification will not be representative of how the code will behave when the red error has been addressed.

It is applicable on the current example:

```
1 int a[] = { 1,2,3,4,5,7,8,9,10 };
2  void main(void)
3  {
4  int x=0;
5  int tmp;
6  if (a[5] > a[6])
7  tmp = 1 /x; // RED ERROR [scalar division by zero] in gray code
8  }
```

Potential Side Effect of a Red Error

This section explains why when a red error has been found the verification continues but some cautions need to be taken. Consider this piece of code:

```
int *global_ptr;                            void other_function(void)
int variable_it_points_to;
                                            {
void big_red(void)
{                                           if (condition==1)
int r;
int my_zero = 0;                             *global_ptr = 12;
if (condition==1)
 r = 1 / my_zero; // red ZDV                }

...

... // hundreds of lines

global_ptr = &variable_it_points_to;

other_function();

}
```

PolySpace works by propagating data sets representing ranges of possible values throughout the call tree, and throughout the functions in that call tree. Sometimes, PolySpace internally subdivides the functions for verification, and the propagation of the data ranges need several iterations (or integration levels) to complete. That effect can be observed by examining the color of the checks on completion of each of those levels. It can sometimes happen that:

- PolySpace will detect gray code which exists due to a terminal RTE which will not be flagged in red until a subsequent integration level.

- PolySpace flags a NTC in red with the content in gray. This red NTC is the result of an imprecision, and should be gray.

Suppose that an NTC is hard to understand at given integration level (level 4):

- If other red checks exist at level 4, fix them and restart the verification

- Otherwise, look back through the results from each previous level to see whether other red errors can be located. If so, fix them and restart the verification

Why Review Dead Code Checks

- "Functional Bugs in Gray Code" on page 9-80
- "Structural Coverage" on page 9-81

Functional Bugs in Gray Code

PolySpace verification finds different types of dead code. Common examples include:

- Defensive code which is never reached
- Dead code due to a particular configuration
- Libraries which are not used to their full extent in a particular context
- Dead code resulting from bugs in the source code.

The causes of dead code listed in the following examples are taken from critical applications of embedded software by PolySpace verification.

- A lack of parenthesis and operand priorities in the testing clause can change the meaning significantly.
- Consider a line of code such as:

  ```
  IF NOT a AND b OR c AND d
  ```

  Now consider how misplaced parentheses might influence how that line behaves:

  ```
  IF NOT (a AND b OR c AND d)

  IF (NOT (a) AND b) OR (c AND d))

  IF NOT (a AND (b OR c) AND d)
  ```
- The test of variable inside a branch where the conditions are never met
- An unreachable "else" clause where the wrong variable is tested in the "if" statement

- A variable that should be local to the file but instead is local to the function

- Wrong variable prototyping leading to a comparison which is always false (say)

As is the case for red errors, the consequences of dead code and how much time you must spend on it is unpredictable. For example, it can be:

- A one-week effort of functional testing on target, trying to build a scenario going into that branch.

- A three-minute code review discovering the bug.

Again, as for red errors, PolySpace does not measure the impact of dead code.

The tool provides a list of dead code. A short code review enables you to place each entry from that list into one of the five categories from the beginning of this chapter. Doing so identifies known dead code and uncovers real bugs.

Using **PolySpace shows that at least 30% of gray code reveals real bugs.**

Structural Coverage

PolySpace software always performs upper approximations of all possible executions. Therefore, if a line of code is shown in green, there is a possibility that it is a dead portion of code. Because PolySpace verification made an upper approximation, it does not conclude that the code is dead, but it could conclude that no run-time error is found.

PolySpace verification finds around 80% of dead code that the developer finds by doing structural coverage.

Use PolySpace verification as a productivity aid in dead code detection. It detects dead code which might take days of effort to find by any other means.

Reviewing Orange Checks

Orange checks indicate *unproven code*. This means that the code can neither be proven safe, nor can it be proven to contain a runtime error.

The number of orange checks you review is determined by several factors, including:

- The stage of the development process
- Your quality objectives

There are also actions you can take to reduce the number of orange checks in your results.

For information on managing orange checks in your results, see Chapter 10, "Managing Orange Checks".

Integration Bug Tracking

By default, you can achieve integration bug tracking by applying the selective orange methodology to integrated code. Each error category reveals integration bugs, depending on the coding rules that you choose for the project.

For instance, consider a function that receives two unbounded integers. The presence of an overflow can be checked only at integration phase because at unit phase the first mathematical operation reveals an orange check.

Consider these two circumstances:

- When you carry out integration bug tracking in isolation, a selective orange review highlights most integration bugs. A PolySpace verification is performed integrating tasks.
- When you carry out integration bug tracking together with an exhaustive orange review at unit phase, a PolySpace verification is performed on one or more files.

In this second case, an exhaustive orange review already has been performed, file by file. Therefore, at integration phase, assess **only checks that have turned from green to another color** .

For instance, if a function takes a structure as an input parameter, the standard hypothesis made at unit level is that the structure is well initialized. This consequentially displays a green NIV check at the first read access to

a field. But this might not be true at integration time, where this check can turn orange if any context does not initialize these fields.

These orange checks reveal integration bugs.

Managing Orange Checks

- "Understanding Orange Checks" on page 10-2
- "Too Many Orange Checks?" on page 10-12
- "Reducing Orange Checks in Your Results" on page 10-14
- "Reviewing Orange Checks" on page 10-25

Understanding Orange Checks

In this section...
"What is an Orange Check?" on page 10-2
"Sources of Orange Checks" on page 10-6

What is an Orange Check?

Orange checks indicate *unproven code*. This means that the code can neither be proven safe, nor can it be proven to contain a runtime error.

PolySpace verification does not simply try to find bugs, it attempts to prove the absence or existence of run time errors. Therefore, all code starts out as unproven prior to verification. The verification then attempts to prove that the code is either correct (green), is certain to fail (red), or is unreachable (gray). Any remaining code stays unproven (orange).

Code often remains unproven in situations where some paths fail while others succeed. For example, consider the following instruction:

```
X = 1 / (X - Y);
```

Does a division-by-zero error occur?

The answer clearly depends on the values of X and Y. However, there are an almost infinite number of possible values. Creating test cases for all possible values is not practical.

X = Y (Division by zero error)

Actual states of operation
X / (X - Y)
(nearly infinite)
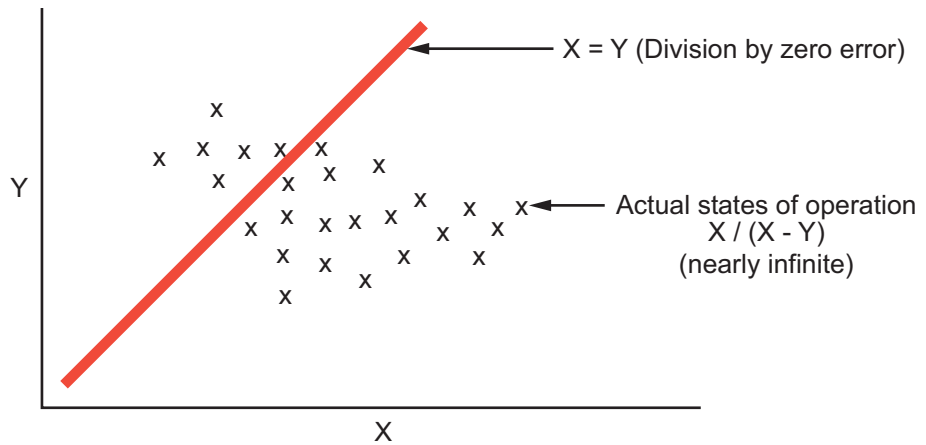
Although it is not possible to test every value for each variable, the target computer and programming language provide limits on the possible values of the variables. PolySpace verification uses these limits to compute a *cloud of points* (upper-bounded convex polyhedron) that contains all possible states for the variables.

Convex polyhedron containing all possible states of X / (X - Y)
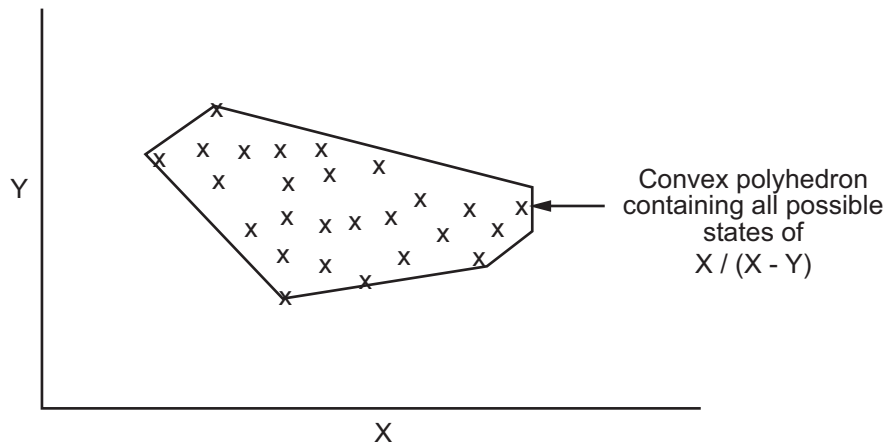
PolySpace verification than compares the data set represented by this polyhedron to the error zone. If the two data sets intersect, the check is orange.
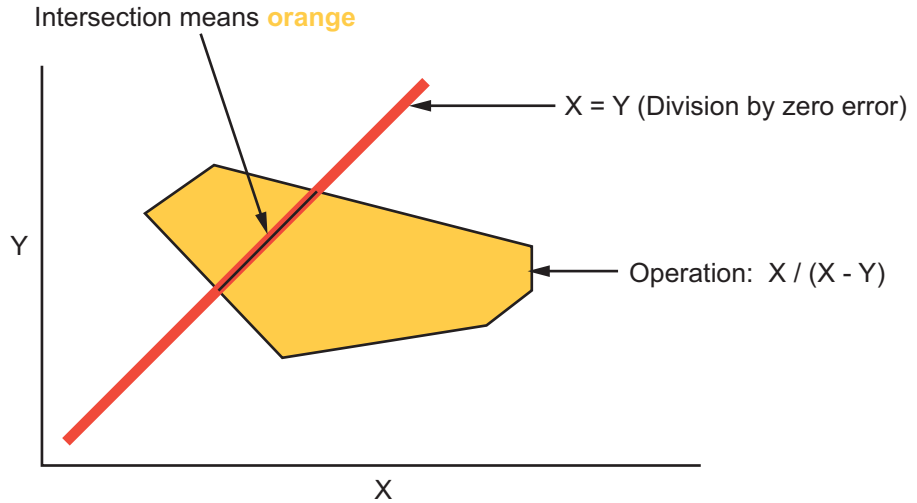
Graphical Representation of an Orange Check

A true orange check represents a situation where some paths fail while others succeed. However, because the data set used in the verification is an approximation of actual values, an orange check may actually represent a check of any other color, as shown below.
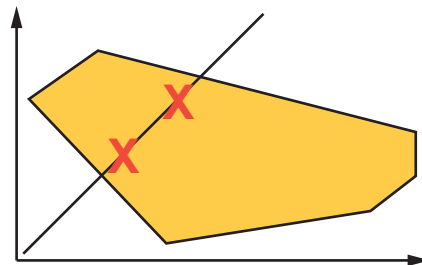
Red approximated by orange
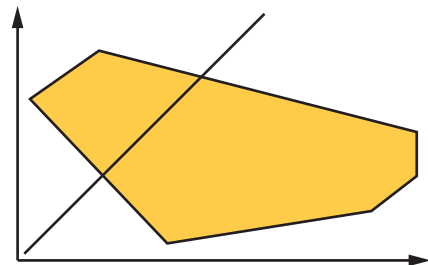
Gray approximated by orange
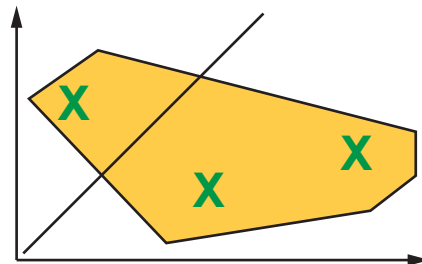
Green approximated by orange
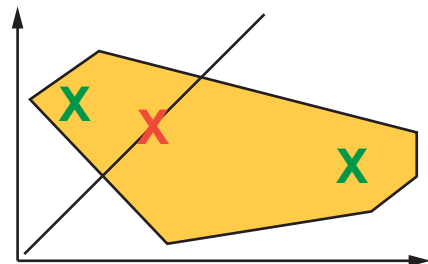
Any other situation (true orange)

PolySpace reports an orange check any time the two data sets intersect, regardless of the actual values. Therefore, you may find orange checks that represent bugs, while other orange checks represent code that is safe.

You can resolve some of these orange checks by increasing the precision of your verification, or by adding execution context, but often you must review the results to determine the source of an orange check.

Sources of Orange Checks

Orange checks can be separated into two categories:

Orange Checks Due to Code Issues

Most orange checks are caused by issues in the code. These oranges may represent real bugs, or could indicate theoretical issues that cannot actually occur in your application.

Orange checks due to code issues can be caused by:

Potential Bug. An orange check can reveal code which will fail in some circumstances. These types of orange checks are called true orange, and often represent real bugs.

For example, consider a function `Recursion()`:

- `Recursion()` takes a parameter, increments it, then divides by it.

- This sequence of actions loops through an indirect recursive call to `Recursion_recurse()`.

If the initial value passed to `Recursion()` is negative, then the recursive loop will at some point attempt a division by zero. Therefore, the division operation causes an orange ZDV.

When an orange check indicates a potential bug, you can usually identify the cause quickly. The range information provided in the viewer can help you identify whether the orange represents a bug that should be fixed. See "Using Range Information in the Viewer" on page 9-68.

If the orange represents a situation that cannot actually occur (for example, the initial value above cannot be negative), you have several options:

- Comment the orange check and ignore it.
- Modify the code to take constraints into account.
- Constrain the data ranges used in the verification using DRS (contextual verification).

Data Set Issue. An orange check can result from a theoretical set of data that cannot actually occur.

PolySpace verification uses an *upper approximation* of the data set, meaning that it considers all combinations of input data rather than any particular combination. Therefore, an orange check may result from a combination of input values that is not possible at execution time.

For example, consider three variables *X*, *Y*, and *Z*:

- Each of these variables is defined as being between 1 and 1,000.
- The code computes X*Y*Z on a 16-bit data type.
- The result can potentially overflow, so it causes an orange OVFL.

When developing the code, you may know that the three variables cannot all take the value 1,000 at the same time, but this information is not available to the verification. Therefore, the multiplication is orange.

When an orange check is caused by a data set issue, it is usually possible to identify the cause quickly. The range information provided in the viewer can help you identify whether the orange represents a bug that should be fixed. See "Using Range Information in the Viewer".

After identifying a data set issue, you have several options:

- Comment the orange check and ignore it.
- Modify the code to take data constraints into account.
- Constrain the data ranges that are verified using DRS (contextual verification).

Function Sequence. An orange check can occur if the verification cannot conclude whether a problem exists.

In some code, it is impossible to conclude whether an error exists without additional information, such as the function sequence.

For example, consider a variable *X*, and two functions, F1and F2:

- F1 assigns X = 12.
- F2 divides a local variable by X.
- The automatically generated main (F0) initializes *X* to 0.
- The generated main then randomly calls the functions, similar to the following:

```
If (random)
 Call F1
 Call F2
Else
 Call F2
 Call F1
```

A division by zero error is possible because F1 can be called before or after F2, so the division causes an orange ZDV. The verification cannot determine if an error will occur unless you define the call sequence.

Many inconclusive orange checks take some time to investigate, due to the complexity of the code. When an orange check is caused by function sequence, you have several options:

- Provide manual stubs for some functions.
- Use -main-generator options to describe the function call sequence, or to specify a function called before the main.
- Write defensive code to prevent potential problems.
- Comment the orange check and ignore it.

Orange Checks Due to Tool Issues

Some orange checks are caused by limitations of the verification process itself.

In these cases, the orange check is a false positive, because the code does not contain an actual bug. However, these types of oranges may suggest design issues with the code.

Orange checks due to tool issues can be caused by:

- "Code Complexity" on page 10-9
- "Basic Imprecision" on page 10-10

Code Complexity. An orange check can occur when the code structure is too complicated to be verified by PolySpace software.

When code is extremely complex, the verification cannot conclude whether a problem exists, and therefore reports an inconclusive orange check in the results.

For example, consider a variable *Computed_Speed*.

- *Computed_Speed* is first copied into a signed integer (between -2^31 and 2^31-1).
- *Computed_Speed* is then copied into an unsigned integer (between 0 and 2^31-1).
- *Computed_Speed* is next copied into a signed integer again.
- Finally, *Computed_Speed* is added to another variable.

PolySpace verification reports an orange OVFL on the addition.

This type of orange check is a false positive, because the scenario does not cause a real bug. However, it does suggest that the code may be poorly designed.

Orange checks caused by code complexity often take some time to investigate, but generally share certain characteristics. Code complexity problems usually result in multiple orange checks in the same module. These checks are often

related, and analysis identifies a single cause — perhaps a function or a variable modified many times.

In these cases, you may want to recode to ensure there is no risk, depending on the criticality of the function and the required speed of execution.

To limit the number of orange checks caused by code complexity, you can:

- Enforce coding rules during development
- Perform unit-by-unit verification to verify smaller sections of code.

Note The MathWorks recommends enforcing compliance with coding standards to reduce code complexity. For more information, see "MISRA Checker".

Basic Imprecision. An orange check can be caused by imprecise approximation of the data set used for verification.

Static verification uses approximations of software operations and data. For certain code constructions, these approximations can lead to a loss of precision, and therefore cause orange checks in the verification results.

For example, consider a variable *X*:

- Before the function call, *X* is defined as having the following values: `-5`, `-3`, `8`, or any value in range `[10...20]`.
  This means that `0` has been excluded from the set of possible values for *X*.
- However, due to optimization (especially at low precision levels), the verification approximates *X* in the range `[-5...20]`, instead of the previous set of values.
- Therefore, calling the function `x = 1/x` causes an orange ZDV.

PolySpace verification is unable to prove the absence of a run-time error in this case.

In cases of basic imprecision, you may be able to resolve orange checks by increasing the precision level. If this does not resolve the orange check, however, verification cannot help directly. You need to review the code to determine if there is an actual problem.

To limit the number of orange checks caused by basic imprecision, avoid code constructions that cause imprecision.

For more information, see "Approximations Used During Verification"in the *PolySpace Products for C++ Reference*.

Too Many Orange Checks?

In this section...
"Do I Have Too Many Orange Checks?" on page 10-12
"How to Manage Orange Checks" on page 10-13

Do I Have Too Many Orange Checks?

If the goal of code verification is to prove the absence of run time errors, you may be concerned by the number of orange checks (unproven code) in your results.

In reality, asking "Do I have too many orange checks?" is not the right question. There is not an ideal number of orange checks that applies for all applications, not even zero. Whether you have too many orange checks depends on:

- **Development Stage** – Early in the development cycle, when verifying the first version of a software component, you may want to focus exclusively on finding red errors, and not consider orange checks. As development of the same component progresses, however, you may want to focus more on orange checks.

- **Application Requirements** – There are actions you can take during coding to produce more provable code. However, writing provable code often involves compromises with code size, code speed, and portability. Depending on the requirements of your application, you may decide to optimize code size, for example, at the expense of more orange checks.

- **Quality Goals** – PolySpace software can help you meet quality goals, but it cannot define those goals for you. Before you verify code, you must define quality goals for your application. These goals should be based on the criticality of the application, as well as time and cost constraints.

It is these factors that ultimately determine how many orange checks are acceptable in your results, and what you should do with the orange checks that remain.

Thus, a more appropriate question is "How do I manage orange checks?"

This question leads to two main activities:

- Reducing the number of orange checks

- Working with orange checks

How to Manage Orange Checks

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

Similarly, you cannot successfully manage orange checks simply by using PolySpace options. To manage orange checks effectively, you must take actions while coding, when setting up your verification project, and while reviewing verification results.

To successfully manage orange checks, perform each of the following steps:

1 Define your quality objectives to set overall goals for application quality. See "Defining Quality Objectives" on page 2-5.

2 Set PolySpace analysis options to match your quality objectives. See "Specifying Options to Match Your Quality Objectives" on page 4-19.

3 Define a process to reduce orange checks. See "Reducing Orange Checks in Your Results" on page 10-14.

4 Apply the process to work with remaining orange checks. See "Reviewing Orange Checks" on page 10-25.

Reducing Orange Checks in Your Results

In this section...
"Overview: Reducing Orange Checks" on page 10-14
"Applying Coding Rules to Reduce Orange Checks" on page 10-15
"Improving Verification Precision" on page 10-15
"Stubbing Parts of the Code Manually" on page 10-20
"Considering Contextual Verification" on page 10-23
"Considering the Effects of Application Code Size" on page 10-23

Overview: Reducing Orange Checks

There are several actions you can take to reduce the number of orange checks in your results.

However, it is important to understand that while some actions increase the quality of your code, others simply change the number of orange checks reported by the verification, without improving code quality.

Actions that reduce orange checks and improve the quality of your code:

- **Apply coding rules** – Coding rules are the most efficient means to reduce oranges, and can also improve the quality of your code.

Actions that reduce orange checks through increased verification precision:

- **Set precision options** – There are several PolySpace options that can increase the precision of your verification, at the cost of increased verification time.
- **Implement manual stubbing** – Manual stubs that accurately model the behavior of missing functions can increase the precision of the verification.

Options that reduce orange checks but do not improve code quality or the precision of the verification:

- **Constrain data ranges** – You can use data range specifications (DRS) to limit the scope of a verification to specific variable ranges, instead of considering all possible values. This reduces the number of orange checks, but does not improve the quality of the code. Therefore, DRS should be used specifically to perform contextual verification, not simply to reduce orange checks.

Each of these actions have trade-offs, either in development time, verification time, or the risk of errors. Therefore, before taking any of these actions, it is important to define your quality objectives, as described in "Defining Quality Objectives" on page 2-5.

It is your quality objectives that determine how many orange checks are acceptable in your results, what actions you should take to reduce the number of orange checks, and what you should do with any orange checks that remain.

Applying Coding Rules to Reduce Orange Checks

The number of orange checks in your results depends strongly on the coding style used in the project. Applying coding rules can both reduce the number of orange checks in your verification results, and improve the quality of your code. Coding rules are the most efficient way to reduce orange checks.

PolySpace software allows you to check MISRA C++ or JSF++ coding rules during verification. If your code complies with these coding rules, the total number of orange checks will decrease substantially, and the percentage of orange checks representing real bugs will increase.

Improving Verification Precision

Improving the precision of a verification can reduce the number of orange checks in your results, although it does not affect the quality of the code itself.

There are a number of PolySpace options that affect the precision of the verification. The trade off for this improved precision is increased verification time.

The following sections describe how to improve the precision of your verification:

Balancing Precision and Verification Time

When performing code verification, you must find the right balance between precision and verification time. Consider the two following extremes:

- If a verification runs in one minute but contains only orange checks, the verification is not useful because each check must be reviewed manually.

- If a verification contains no orange checks (only gray, red, and green), but takes six months to run, the verification is not useful because of the time spent waiting for the results.

Higher precision yields more proven code (red, green, and gray), but takes longer to complete. The goal is therefore to get the most precise results in the time available. Factors that influence this compromise include the time available for verification, the time available to review results, and the stage in the development cycle.

For example, consider the following scenarios:

- **Unit testing** – Before going to lunch, a developer starts verification. After returning from lunch the developer reviews verification results for one hour.

- **Integration testing** – Verification runs automatically on nightly builds of modules or software components.

These scenarios require a developer to use PolySpace software in different ways. Generally, the first verification should use the lowest precision mode, while subsequent verifications increase the precision level.

Setting the Analysis Precision Level

The analysis **Precision Level** specifies the mathematical algorithm used to compute the cloud of points (polyhedron) containing all possible states for the variables.

Although changing the precision level does not affect the quality of your code, orange checks caused by low precision become green when verified with higher precision.
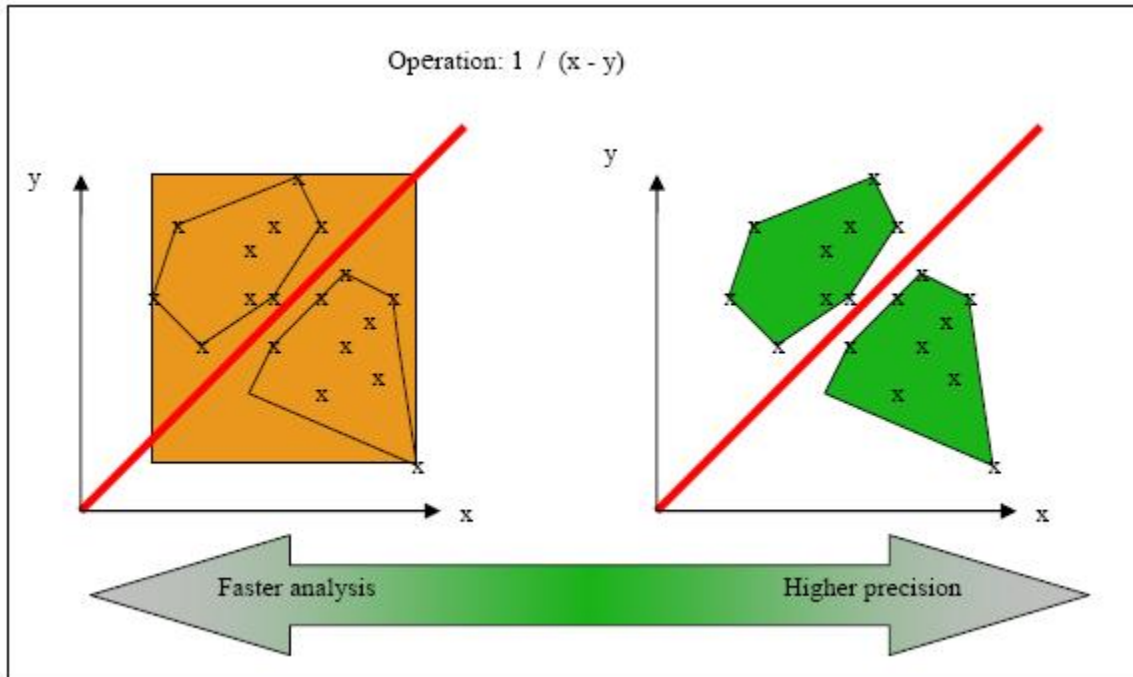
Affect of Precision Rate on Orange Checks

To set the precision level:

1 In the Analysis options section of the Launcher window, select **Precision/Scaling > Precision**.

2 Select the -O0, -O1, -O2 or -O3 precision level the Precision Level drop-down list.

For more information, see "-O(0-3)"in the *PolySpace Products for C++ Reference*.

Setting Software Safety Analysis Level

The Software Safety Analysis level of your verification specifies how many times the abstract interpretation algorithm passes through your code. The deeper the verification goes, the more precise it is.

There are 5 Software Safety Analysis levels (pass0 to pass4). By default, verification proceeds to pass4, although it can go further if required. Each iteration results in a deeper level of propagation of calling and called context.

To set the Software Safety Analysis level:

1 In the Analysis options section of the Launcher window, select **Precision/Scaling > Precision**.

2 Select the appropriate level in the **To end of** drop-down list.

For more information, see "-to verification-phase" in the *PolySpace Products for C++ Reference*.

Example: Orange Checks and Software Safety Analysis Level

The following example shows how orange checks are resolved as verification proceeds through Software Safety Analysis levels 0 and 1.

Safety Analysis Level 0	Safety Analysis Level 1
``#include <stdlib.h>

void ratio (float x, float *y)
{
 *y=(abs(x-*y))/(x+*y);
}

void level1 (float x,
     float y, float *t)
{ float v;
 v = y;
 ratio (x, &y);
 *t = 1.0/(v - 2.0 * x);
}``` | ```#include <stdlib.h>

void ratio (float x, float *y)
{
 *y=(abs(x-*y))/(x+*y);
}

void level1 (float x,
     float y, float *t)
{ float v;
 v = y;
 ratio (x, &y);
 *t = 1.0/(v - 2.0 * x);
}``` |

Safety Analysis Level 0	Safety Analysis Level 1
``float level2(float v)
{
 float t;
 t = v;
 level1(0.0, 1.0, &t);
 return t;
}

void main(void)
{
 float r,d;
 d= level2(1.0);
 r = 1.0 / (2.0 - d);
}``` | ```float level2(float v)
{
 float t;
 t = v;
 level1(0.0, 1.0, &t);
 return t;
}

void main(void)
{
 float r,d;
 d= level2(1.0);
 r = 1.0 / (2.0 - d);
}``` |

In this example, division by an input parameter of a function produces an orange during Level 0 verification, but turns to green during level 1. The verification gains more accurate knowledge of x as the value is propagated deeper.

Other Options that Can Improve Precision

The following options can also improve verification precision:

- "Improve precision of interprocedural analysis" on page 10-20

- "Sensitivity context" on page 10-20

- "Inline" on page 10-20

Note Changing these options does not affect the quality of the code itself. Improved precision can reduce the number of orange checks, but will increase verification time.

Improve precision of interprocedural analysis. This option causes the verification to propagate information within procedures earlier than usual. This improves the precision within each Software Safety Analysis level, meaning that some orange checks are resolved in level 1 instead of later levels.

However, using this option increases verification time exponentially. In some cases this could cause a level 1 verification to take longer than a level 4 verification.

For more information, see "-path-sensitivity-delta number"in the *PolySpace Products for C++ Reference*.

Sensitivity context. This option splits each check within a procedure into sub-checks, depending on the context of a call. This improves precision for discrete calls to the procedure. For example, if a check is red for one call to the procedure and green for another, both colors will be revealed.

For more information, see "-context-sensitivity "proc1[,proc2[,...]]""in the *PolySpace Products for C++ Reference*.

Inline. This option creates clones of a each specified procedure for each call to it. This reduces the number of aliases in a procedure, and can improve precision in some situations.

However, using this option can duplicate large amounts of code, leading to increased verification time and other scaling problems.

For more information, see "-inline "proc1[,proc2[,...]]""in the *PolySpace Products for C++ Reference*.

Stubbing Parts of the Code Manually

Manually stubbing parts of your code can reduce the number of orange checks in your results. However, manual stubbing generally does not improve the quality of your code, it only changes the results.

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

If a function is supposed to return an integer, the default automatic stubbing will stub it on the assumption that it can potentially take any value from the full type of an integer.

The following sections describe how to reduce orange checks using manual stubbing:

- "Manual vs. Automatic Stubbing" on page 10-21
- "Emulating Function Behavior with Manual Stubs" on page 10-22

Manual vs. Automatic Stubbing

There are two types of stubs in PolySpace verification:

- **Automatic stubs** – The software automatically creates stubs for unknown functions based on the function's prototype (the function declaration). Automatic stubs do not provide insight into the behavior of the function, but are very conservative, ensuring that the function does not cause any runtime errors.

- **Manual stubs** – You create these stub functions to model the behavior of the missing functions, and manually include them in the verification with the rest of the source code. Manual stubs can better model missing functions, or they can be empty.

By default, PolySpace software automatically stubs functions. However, because automatic stubs are conservative, they can lead to more orange checks in your results.

Stubbing Example

The following example shows the effect of automatic stubbing.

```
void main(void)
{
 a=1;
 b=0;
 a_missing_function(&a, b);
 b = 1 / a;
}
```

Due to automatic stubbing, the verification assumes that *a* can be any integer, including 0. This produces an orange check on the division.

If you provide an empty manual stub for the function, the division would be green. This reduces the number of orange checks in the result, but does not improve the quality of the code itself. The function could still potentially cause an error.

However, if you provide a detailed manual stub that accurately models the behavior of the function, the division could be any color, including red.

Emulating Function Behavior with Manual Stubs

You can improve both the speed and selectivity of your verification by providing manual stubs that accurately model the behavior of missing functions. The trade-off is time spent writing the stubs.

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

Example

This example shows a header for a missing function (which may occur when the verified code is an incomplete subset of a project).

```
int a,b;
int *ptr;
void a_missing_function(int *dest, int src);
/* should copy src into dest */
void main(void)
{
 a = 1;
 b = 0;
 a_missing_function(&a, b);
 b = 1 / a;
}
```

The missing function copies the value of the src parameter to dest, so there is a division by zero error.

However, automatic stubbing always shows an orange check, because a is assumed to have any value in the full integer range. Only an accurate manual stub can reveal the true red error.

Using manual stubs to accurately model constraints in primitives and outside functions propagates more precision throughout the application, resulting in fewer orange checks.

Considering Contextual Verification

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

PolySpace software also allows you to perform *contextual verification*, proving that the software works under normal working conditions. When performing contextual verification, you use the data range specifications (DRS) module to set external constraints on global variables and stub function return values, and the code is verified within these ranges.

Contextual verification can substantially reduce the number of orange checks in your verification results, but it does not improve the quality of your code.

Note DRS should be used specifically to perform contextual verification, it is not simply a means to reduce oranges.

For more information, see "Specifying Data Ranges for Variables and Functions (Contextual Verification)" on page 5-21.

Considering the Effects of Application Code Size

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

For example, in a relatively small application, PolySpace might retain very detailed information about the data at a particular point in the code, so

that for example the variable VAR can take the values { -2; 1; 2; 10; 15; 16; 17; 25 }. If VAR is used to divide, the division is green (because 0 is not a possible value).

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

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

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

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

- -O0: shorter computation time. Focus only red and gray.
- -O2: less orange warnings.
- -O3: less orange warnings and bigger computation time.

Reviewing Orange Checks

In this section...
"Overview: Reviewing Orange Checks" on page 10-25
"Defining Your Review Methodology" on page 10-25
"Performing Selective Orange Review" on page 10-27
"Importing Review Comments from Previous Verifications" on page 10-30
"Commenting Code to Provide Information During Review" on page 10-31
"Performing an Exhaustive Orange Review" on page 10-32

Overview: Reviewing Orange Checks

After you define a process that matches your quality objectives, you do not have too many orange checks. You have the correct number of orange checks for your quality model.

At this point, the goal is not to eliminate orange checks, it is to work efficiently with them.

Working efficiently with orange checks involves:

- Defining a review methodology to work consistently with orange checks

- Reviewing orange checks efficiently

- Importing comments to avoid duplicating review effort

Defining Your Review Methodology

Before reviewing verification results, you should configure a methodology for your project. The methodology defines both the type and number of orange checks you need to review to meet three criteria levels.

Number of checks to review			
	Criterion 1	Criterion 2	Criterion 3
Common			
ZDV	5	20	ALL
NIVL	10	50	ALL
S-OVFL	10	50	ALL
COR	0	10	10
NIV	0	0	10
F-OVFL	5	10	20
ASRT	0	5	20
C & C++ only			
OBAI	10	20	ALL
SHF	5	10	ALL
IDP	0	10	20
NIP	0	10	20
C only			
IRV	5	20	ALL

Sample Review Methodology

The criteria levels displayed in the methodology represent quality levels you defined as part of the quality objectives for your project.

Note For information on setting the quality levels for your project, see Chapter 2.

After you configure a methodology, each developer uses it to review verification results. This ensures that all users apply the same standards when reviewing orange checks in each stage of the development cycle.

For more information on defining a methodology, see "Selecting the Methodology and Criterion Level" on page 9-28.

Performing Selective Orange Review

Once you have defined a methodology for your project, you can use assistant mode to perform a *selective orange review*.

The number and type of orange checks you review is determined by your methodology and the quality level you are trying to achieve. As a project progresses, the quality level (and number of orange checks to review) generally increases.

For example, you may perform a level 1 review in the early stages of development, when trying to improve the quality of freshly written code. Later, you may perform a level 2 review as part of unit testing.

In general, the goal of a selective orange review is to find the maximum number of bugs in a short period of time. Many orange checks take only a few seconds to understand. Therefore, to maximize the number of bugs you can identify, you should focus on those checks you can understand quickly, spending no more than 5 minutes on each check. Checks that take longer to understand are left for later analysis.

To perform a selective orange review:

1 Click the **Assistant** button in the Viewer to select assistant mode.

The Viewer window toolbar displays the assistant mode controls.

2 Select the methodology for your project from the methodology menu.
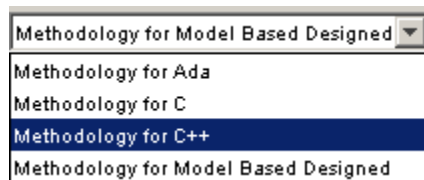
3 Select the appropriate quality level for your review using the level slider.

4 Navigate through the checks by clicking the forward arrow .

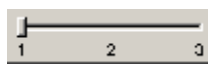**5** Perform a quick code review on each orange check, spending no more than 5 minutes on each.

Your goal is to quickly identify whether the orange check is a:

- **potential bug** – code which will fail under some circumstances.

- **inconclusive check** – a check that requires additional information to resolve, such as the call sequence.

- **data set issue** – a theoretical set of data that cannot actually occur.

See "Sources of Orange Checks" on page 10-6 for more information on each of these causes.

Note If an orange check is too complicated to explain quickly, it may be an inconclusive check caused by complex code structure, or the result of basic imprecision (approximation of the data set used for verification). These types of checks often take a substantial amount of time to understand.

---

6 If you cannot identify a cause within 5 minutes, move on to the next check.

Note Your goal is to find the maximum number of bugs in a short period of time. Therefore, you want to identify the source of as many orange checks as possible, while leaving more complex situations for future analysis.

---

7 Once you understand the cause of an orange check, select the check box to indicate that you have reviewed the check.

```
training.cpp / MathUtils::Recursion(int*) / line 109 / column 15

  ⊞        advance = 1.0/(float)(*depth); // potential division by zero

☑ OTH  ▾ Not an Issue  ▾ 📝 Value of depth cannot be negative

Warning : float division by zero may occur
    operator / on type float 64
        left:   1.0
        right:  [-2.1475E⁺⁹ .. -9.9999E⁻¹] or 0.0 or [2.9999 .. 5.0001E⁺¹]
        result: [-1.0001 .. -4.6566E⁻¹⁰] or [1.9999E⁻² .. 3.3334E⁻¹]
operation [/] on float does not overflow in FLOAT64 range
    operator / on type float 64
     left: 1.0
        right: [-2.1475E⁺⁹ .. -9.9999E⁻¹] or 0.0 or [2.9999 .. 5.0001E⁺¹]
        result: [-1.0001 .. -4.6566E⁻¹⁰] or [1.9999E⁻² .. 3.3334E⁻¹]
```

8 Select an acronym to describe the check in the Predefined acronyms menu:

- **NOW** – Bug to fix now.
- **NXT** – Bug to fix in Next Release
- **ROB** – Robustness Issue
- **DEF** – Defensive Code
- **MIN** – Minor quality issue
- **OTH** – Other

Note You can also define your own acronyms, which then appear in the user-defined acronym menu. For more information see "Defining Custom Acronyms " on page 9-49.

---

9 Enter a comment for the reviewed check in the text box, indicating the results of your review.

10 Continue to click the forward arrow until you have reviewed all of the checks identified by the assistant.

11 Select **File > Save checks and comments** to save your review comments.

Importing Review Comments from Previous Verifications

Once you have reviewed verification results for a module and saved your comments, you can import those comments into subsequent verifications of the same module, allowing you to avoid reviewing the same check twice.

To import review comments from a previous verification:

1 Open your most recent verification results in the Viewer.

2 Select **File > Import checks and comments**.

3 Navigate to the folder containing your previous results.

4 Select the results (.RTE) file, then click **Open**.

The review comments from the previous results are imported into the current results.

Once you import checks and comments, the **go to next check** ⟫ icon in assistant mode will skip any reviewed checks, allowing you to review only checks that you have not reviewed previously. If you want to view reviewed checks, click the **go to next reviewed check** ⟫ icon.

Note If the code has changed since the previous verification, the imported comments may not be applicable to your current results. For example, the justification for an orange check may no longer be relevant to the current code. To see the changes that affect your review comments, open the Import/Export Report.

For more information, see "Importing and Exporting Review Comments" on page 9-53.

Commenting Code to Provide Information During Review

You can place comments in your code to provide information on known issues. When reviewing results, you can use these comments to:

- Highlight and quickly understand issues identified in previous verifications

- Identify and skip previously reviewed checks.

This allows you to avoid reviewing the same check twice, and focus your review on new issues.

You must annotate your code before running a verification:

```
if (u.random_int())
    // polyspace<RTE: CPP : OTH : EGC> This is a known run-time error
    return (typeid(this) ==  typeid (*ptr));
```

In the Viewer, the **Acronym**, **User Acronym** and **Comment** columns display your code comments. In addition, in the **Reviewed** column , the check box is selected.

For more information, see "Highlighting Known Coding Rule Violations and Run-Time Errors" on page 6-24.

Performing an Exhaustive Orange Review

Up to 80% of orange checks can be resolved using multiple iterations of the process described in "Performing Selective Orange Review" on page 10-27. However, for extremely critical applications, you may want to resolve all orange checks. Exhaustive orange review is the process for resolving the remaining orange checks.

An exhaustive orange review is generally conducted later in the development process, during the unit testing or integration testing phase. The purpose of an exhaustive orange review is to analyze any orange checks that were not resolved during previous selective orange reviews, to identify potential bugs in those orange checks.

You must balance the time and cost of performing an exhaustive orange review against the potential cost of leaving a bug in the code. Depending on your quality objectives, you may or may not want to perform an exhaustive orange review.

Cost of Exhaustive Orange Review

During an exhaustive orange review, each orange check takes an average of 5 minutes to review. This means that 400 orange checks require about four days of code review, and 3,000 orange checks require about 25 days.

However, if you have already completed several iterations of selective orange review, the remaining orange checks are likely to be more complex than average, increasing the average time required to resolve them.

Exhaustive Orange Review Methodology

Performing an exhaustive orange review involves reviewing each orange check individually. As with selective orange review, your goal is to identify whether the orange check is a:

- **potential bug** – code which will fail under some circumstances.
- **inconclusive check** – a check that requires additional information to resolve, such as the call sequence.
- **data set issue** – a theoretical set of data that cannot actually occur.

- **Basic imprecision** – checks caused by imprecise approximation of the data set used for verification.

Note See "Sources of Orange Checks" on page 10-6 for more information on each of these causes.

Although you must review each check individually, there are some general guidelines to follow.

1 Start your review with the modules that have the highest selectivity in your application.

   If the verification finds only one or two orange checks in a module or function, these checks are probably not caused by either inconclusive verification or basic imprecision. Therefore, it is more likely that these orange checks contain actual bugs. In general, these types of orange checks can also be resolved more quickly.

2 Next, examine files that contain a large percentage of orange checks compared to the rest of the application. These files may highlight design issues.

   Often, when you examine modules containing the most orange checks, those checks will prove inconclusive. If the verification is unable to draw a conclusion, it often means the code is very complex, which can mean low robustness and quality. See "Code Complexity" on page 10-9.

3 For all files you review, spend the first 10 minutes identifying checks that you can quickly categorize (such as potential bugs and data set issues), similar to what you do in a selective orange review.

   Even after performing a selective orange review, a significant number of checks can be resolved quickly. These checks are more likely than average to reflect actual bugs.

4 Spend the next 40 minutes of each hour tracking more complex bugs.

   If an orange check is too complicated to explain quickly, it may be an inconclusive check caused by complex code structure, or the result of basic

imprecision (approximation of the data set used for verification). These types of checks often take a substantial amount of time to understand. See "Basic Imprecision" on page 10-10.

5 Depending on the results of your review, correct the code or comment it to identify the source of the orange check.

Inconclusive Verification and Code Complexity

The most interesting type of inconclusive check occurs when verification reveals that the code is too complicated. In these cases, most orange checks in a file are related, and careful analysis identifies a single cause — perhaps a function or a variable modified many times. These situations often focus on functions or variables that have caused problems earlier in the development cycle.

For example, consider a variable *Computed_Speed*.

- *Computed_Speed* is first copied into a signed integer (between -2^{31} and $2^{31}-1$).

- *Computed_Speed* is then copied into an unsigned integer (between 0 and $2^{31}-1$).

- *Computed_Speed* is next copied into a signed integer again.

- Finally, *Computed_Speed* is added to another variable.

The verification reports 20 orange overflows (OVFL).

This scenario does not cause a real bug, but the development team may know that this variable caused trouble during development and earlier testing phases. PolySpace verification also identified a problem, suggesting that the code is poorly designed.

Resolving Orange Checks Caused by Basic Imprecision

On rare occasions, a module may contain many orange checks caused by imprecise approximation of the data set used for verification. These checks are usually local to functions, so their impact on the project as a whole is limited.

In cases of basic imprecision, you may be able to resolve orange checks by increasing the precision level. If this does not resolve the orange check, however, verification cannot help directly.

In these cases, PolySpace software can only assist you through the call tree and dictionary. The code needs to be reviewed using alternate means. These alternate means may include:

- Additional unit tests
- Code review with the developer
- Checking an interpolation algorithm in a function
- Checking calibration data

For more information on basic imprecision, see "Sources of Orange Checks" on page 10-6.

Day to Day Use

PolySpace In One Click Overview

Most developers verify the same files multiple times (writing new code, unit testing, integration), and usually need to run verifications on multiple project files using the same set of options. In a Microsoft Windows environment, PolySpace In One Click provides a convenient way to streamline your work when verifying several files using the same set of options.

Once you have set up a project file with the options you want, you designate that project as the *active project*, and then send the source files to PolySpace software for verification. You do not have to update the project with source file information.

On a Windows systems, the plug-in provides a PolySpace Toolbar in the Windows Taskbar, and a **Send To** option on the desktop pop-up menu:
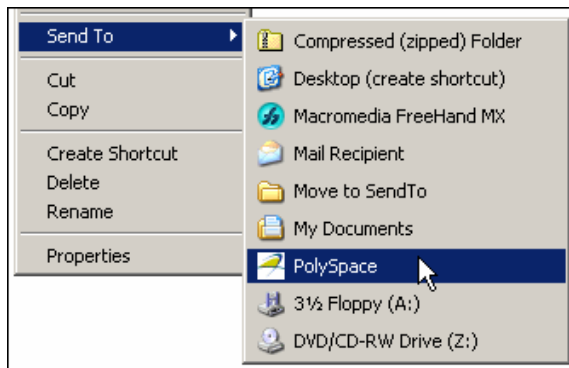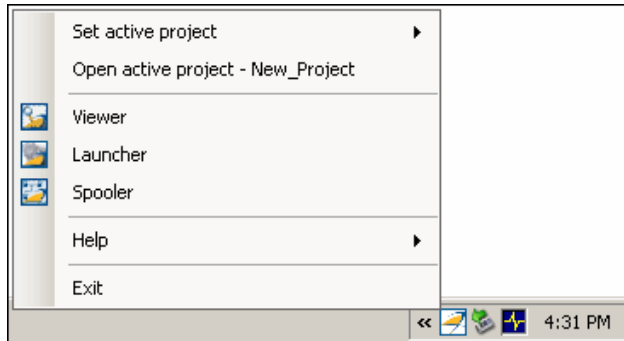
Using PolySpace In One Click

In this section...
"PolySpace In One Click Workflow" on page 11-3
"Setting the Active Project" on page 11-3
"Launching Verification" on page 11-5
"Using the Taskbar Icon" on page 11-9

PolySpace In One Click Workflow

Using PolySpace In One Click involves two steps:

1 Setting the active project.

2 Sending files to PolySpace software for verification.

Setting the Active Project

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

To set the active project:

1 Right-click the PolySpace In One Click icon in the taskbar area of your Windows desktop:

The context menu appears.
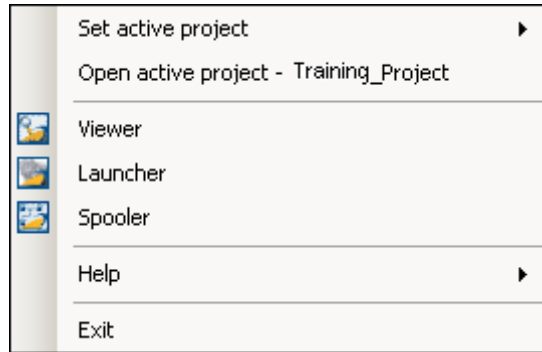
2 Select **Set active project > Browse** from the menu.

The **Please set an active project** dialog box appears:
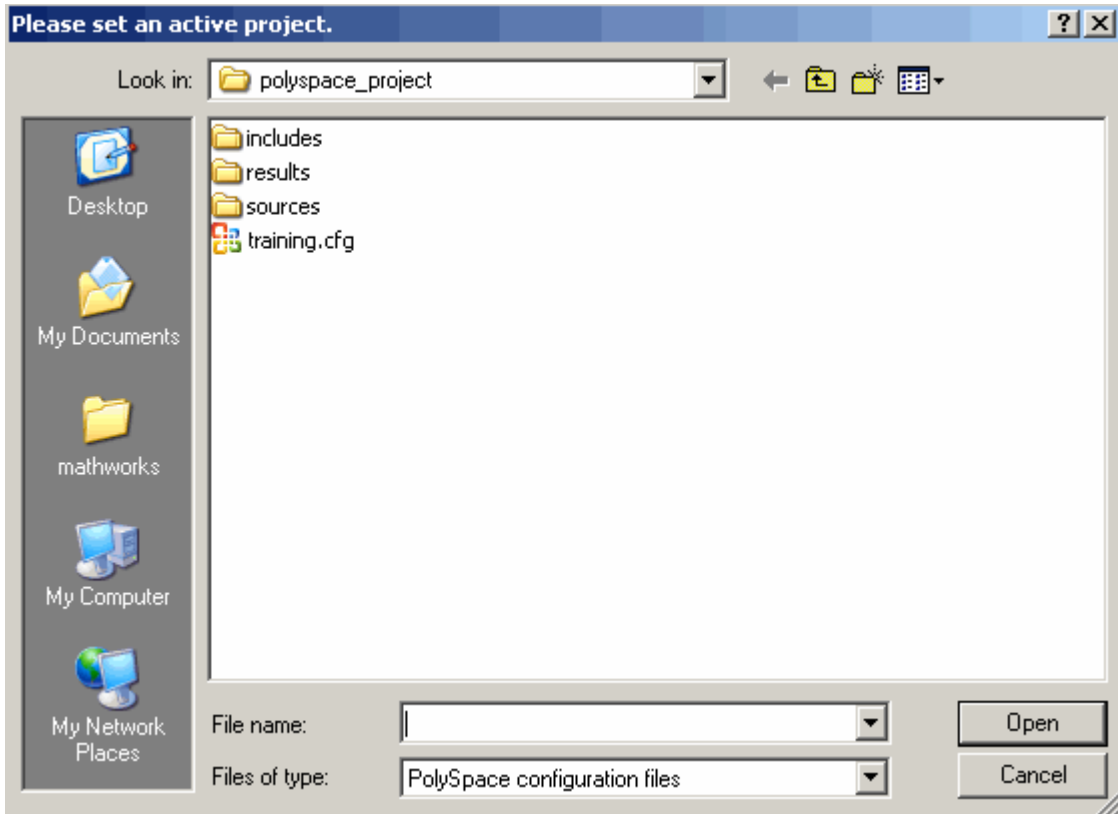
3 Select the project you want to use as the active project.

4 Click **Open** to apply the changes and close the dialog box.

Note You can also set the active project by right-clicking on a project file (.cfg or .dsk) file and selecting **Send To > PolySpace**.

Launching Verification

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

To send a file to PolySpace software for verification:

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

2 Right-click the file you want to verify.

3 Select **Send To > PolySpace**.
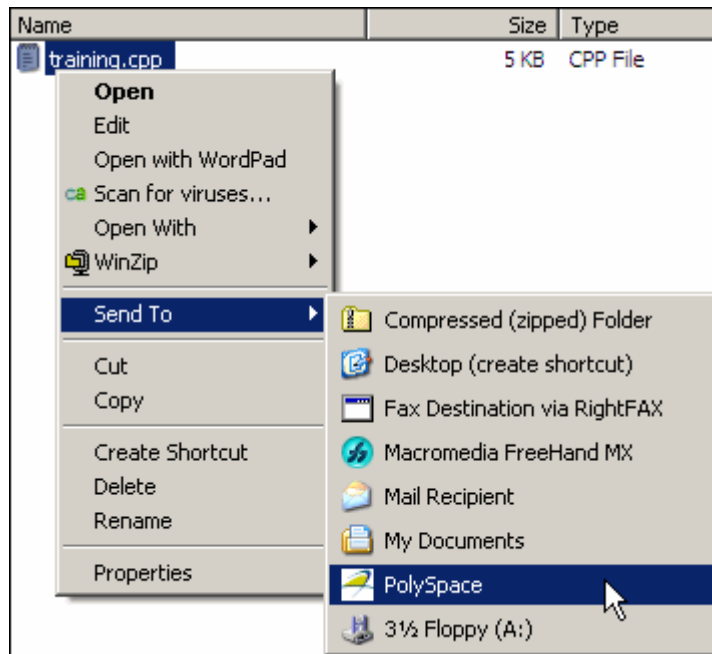
The **PolySpace basic settings** dialog box appears.
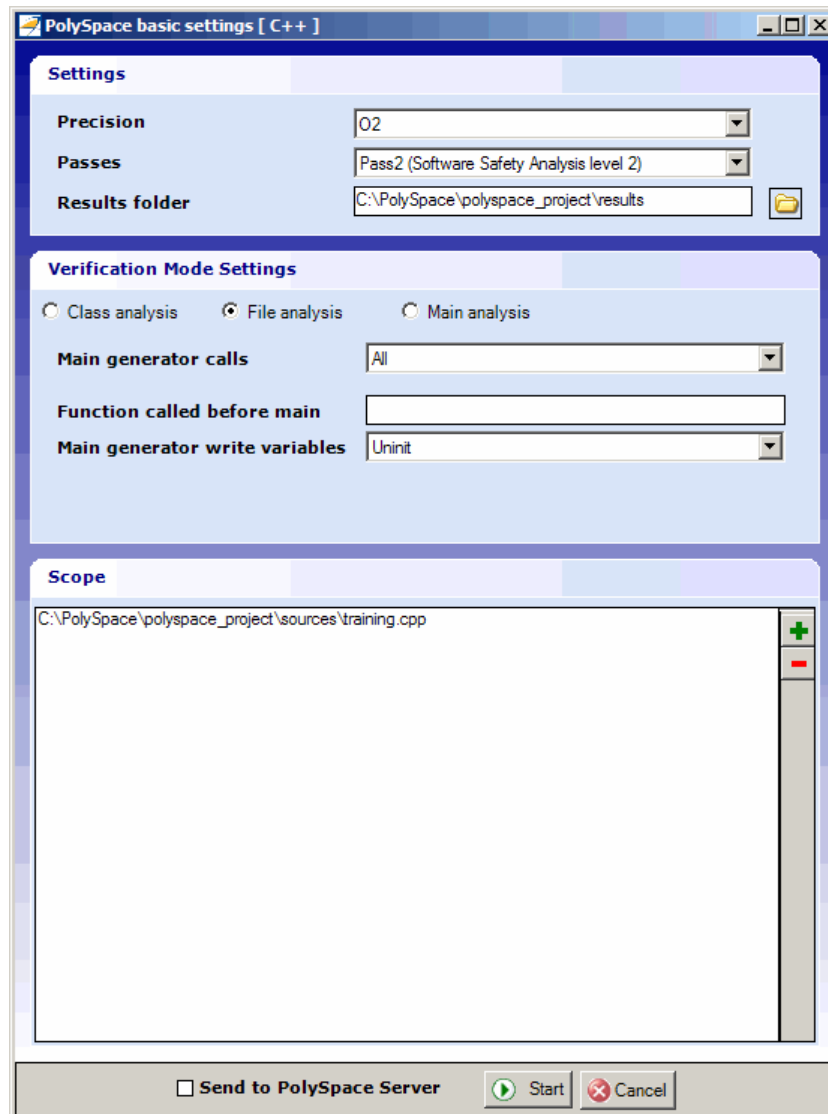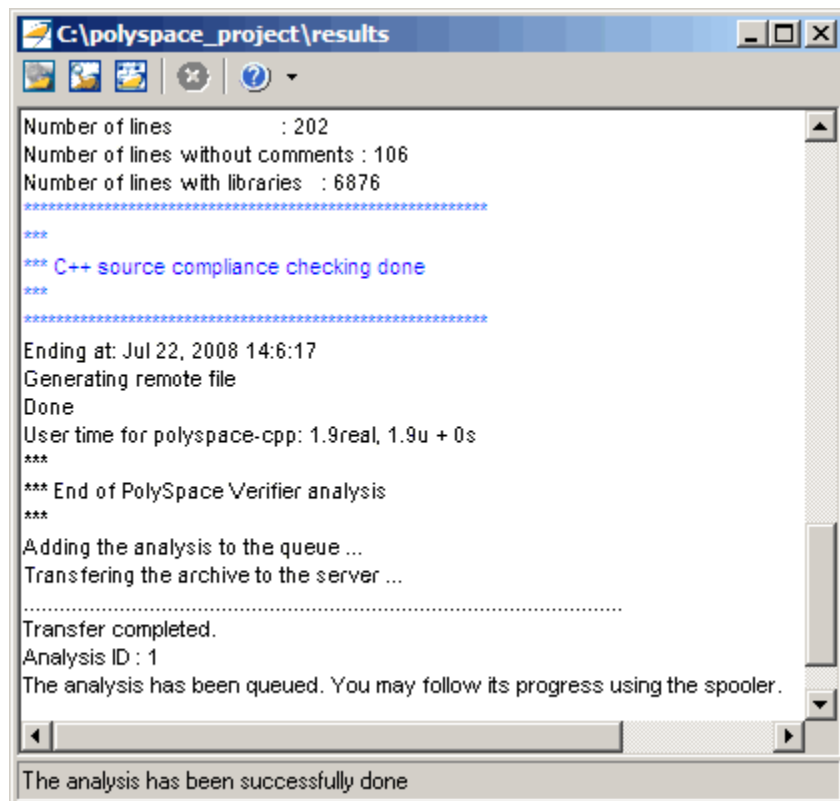
> **Note** The options you specify the basic settings dialog box override any options set in the configuration file. These options are also preserved between verifications.

4 Enter the appropriate parameters for your verification.

5 Click **Start**.

The verification starts and the verification log appears.

Using the Taskbar Icon

The PolySpace in One Click Taskbar icon allows you to access various software features.
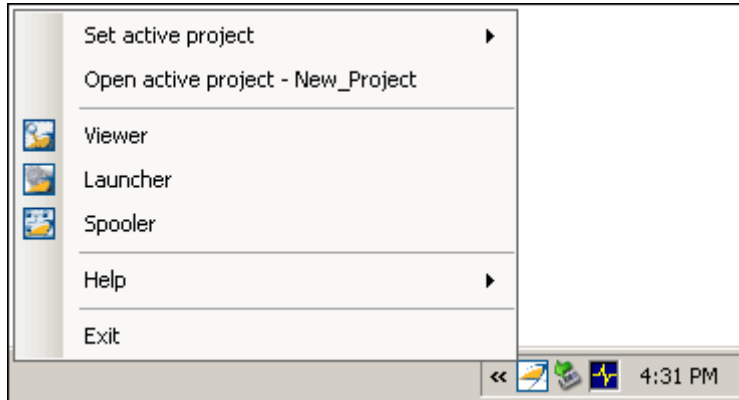
Click the PolySpace Taskbar Icon, then select one of the following options:

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

  A standard file browser allows you to choose the configuration file. If you have multiple configuration files, you can quickly switch between them using the browse history.
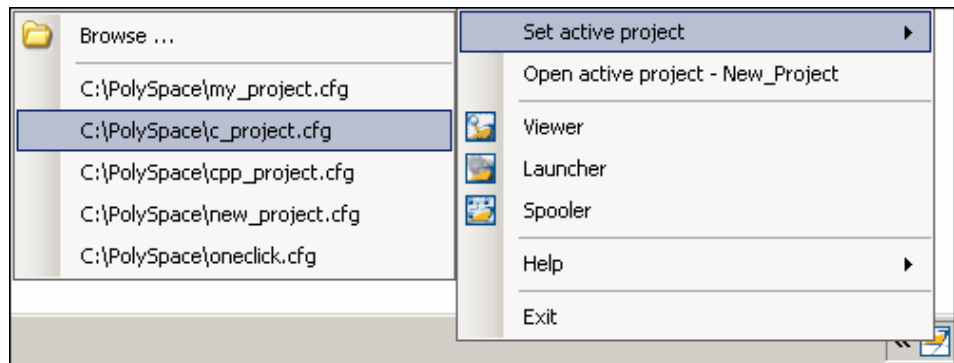
> **Note** No configuration file is selected by default. You can create an empty file with a .cfg extension.

- **Open active project** — Opens the active configuration file. This allows you to update the project using the standard PolySpace Launcher graphical interface. It allows you to specify all PolySpace common options, including directives of compilation, options, and paths of standard and specific headers. It does not affect the precision of a verification or the results folder.

- **Viewer** — Opens the PolySpace viewer. This allows you to review verification results in the standard graphical interface. In order to load results into the viewer, you must choose a verification to review in the Verification Log window.

- **Launcher** — Opens the PolySpace Launcher. This allows you to launch a verification using the standard PolySpace graphical interface.

- **Spooler** — Opens the PolySpace Spooler. If you selected a server verification in the "PolySpace Preferences" dialog box, the spooler allows you to follow the status of the verification.

Checking Coding Rules

Overview of PolySpace Code Analysis

PolySpace software allows you to analyze code to demonstrate compliance with established C++ coding standards (MISRA C++:2008 or JSF++:2005).

Applying coding rules can both reduce the number of orange checks in your verification results, and improve the quality of your code. Coding rules are the most efficient way to reduce orange checks.

While creating a project, you specify both the coding standard, and individual rules to enforce. PolySpace software then performs code analysis before starting verification, and reports any errors or warnings in the Launcher log.

Checking MISRA C++ Coding Rules

In this section...
"PolySpace MISRA C++ Checker Overview" on page 12-3
"Setting Up MISRA C++ Checking" on page 12-3
"Running a Verification with MISRA C++ Checking" on page 12-9

PolySpace MISRA C++ Checker Overview

The PolySpace MISRA C++ checker helps you comply with the MISRA C++:2008 coding standard.[2]

The PolySpace MISRA C++ checker enables PolySpace software to provide messages when MISRA C++ rules are not respected. Most messages are reported during the compile phase of a verification. The MISRA C++ checker can check 167 of the 228 MISRA C++ coding rules .

Note The PolySpace MISRA C++ checker is based on MISRA C++:2008 – "Guidelines for the use of the C++ language in critical systems." For more information on these coding standards, see http://www.misra-cpp.com.

Setting Up MISRA C++ Checking

- "Activating the MISRA C++ Checker" on page 12-4
- "Creating a MISRA C++ Rules File" on page 12-4
- "Excluding Files from MISRA C++ Checking" on page 12-6
- "Configuring Text and XML Editors" on page 12-7

---

2. MISRA is a registered trademark of MISRA Ltd., held on behalf of the MISRA Consortium.

Activating the MISRA C++ Checker

You activate the MISRA C++ checker using the options `misra-cpp` and `includes-to-ignore`. These options can be set at the command line, or through the PolySpace Launcher user interface.

To activate the MISRA C++ Checker:

1 Open the PolySpace Client for C/C++ window by double-clicking the PolySpace Launcher icon on your desktop:

2 Open the project you want to use.

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

The software displays the two MISRA C++ rules checker options: `misra-cpp` and `includes-to-ignore`.

⊟ Coding rules checker			
⊟ Check JSF C++ rules	☐		
JSF C++ rules configuration		...	-jsf-coding-rules
⊟ Check MISRA C++ rules	☑		
MISRA C++ rules configuration		...	-misra-cpp
Files and folders to ignore		...	-includes-to-ignore

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

4 Select the **Check MISRA C++ rules** check box.

Creating a MISRA C++ Rules File

You must have a rules file to run a verification with MISRA C++ checking. You can use an existing file or create a new one.

To create a new rules file:

1 Click the browse button [...] to the right of the **MISRA C++ rules configuration** option.

The New File window opens, allowing you to create a new MISRA C++ rules file, or open an existing file.
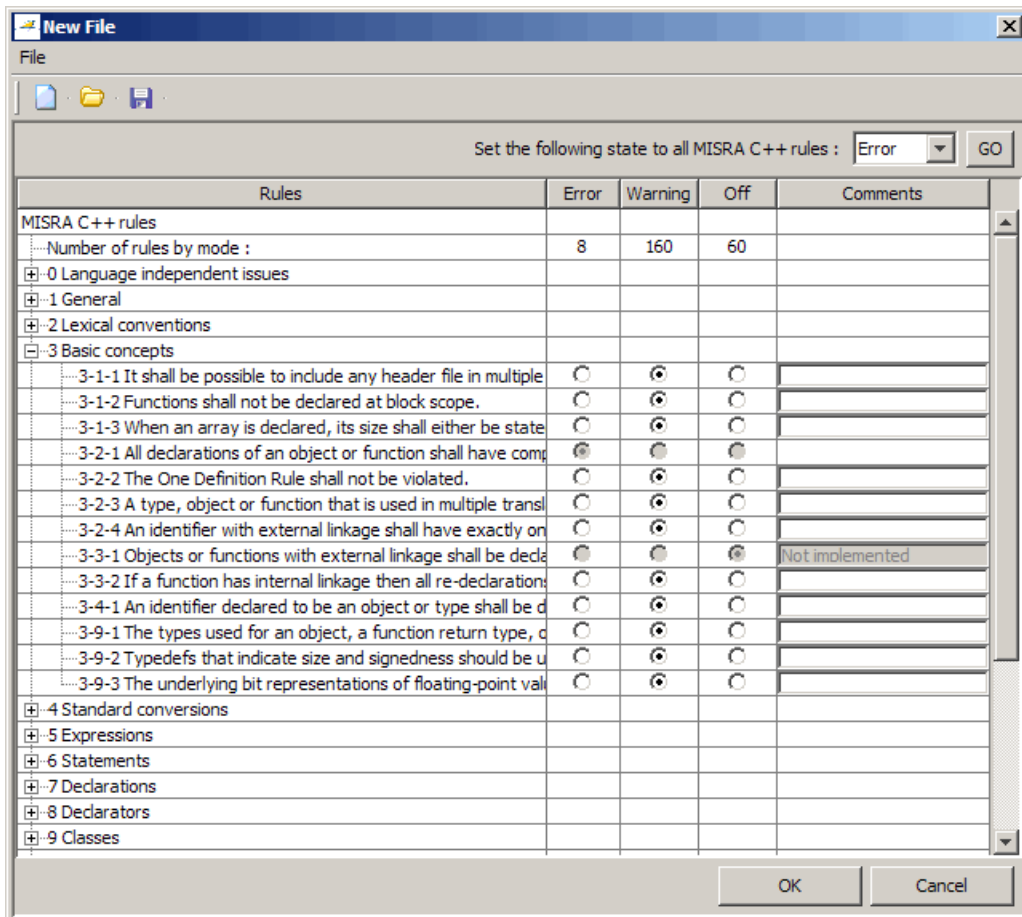
2 For each MISRA C++ rule, specify one of these states:

State	Causes the verification to...
Error	End after the compile phase when this rule is violated.
Warning	Display warning message and continue verification when this rule is violated.
Off	Skip checking of this rule.

Note The default state for most rules is Warning. The state for rules that have not yet been implemented is Off. Some rules always have state Error (you cannot change the state of these).

3 Click **OK** to save the rules and close the window.

The **Save as** dialog box opens.

4 In **File**, enter a name for your rules file.

5 Click **OK** to save the file and close the dialog box.

Excluding Files from MISRA C++ Checking

You can exclude files from MISRA C++ checking. For example, you may want to exclude some included files.

To exclude files from MISRA C++ checking:

1 Click the button ⬚ to the right of the **Files and folders to ignore** option.

The Files and folders to ignore (includes-to-ignore) dialog box opens.
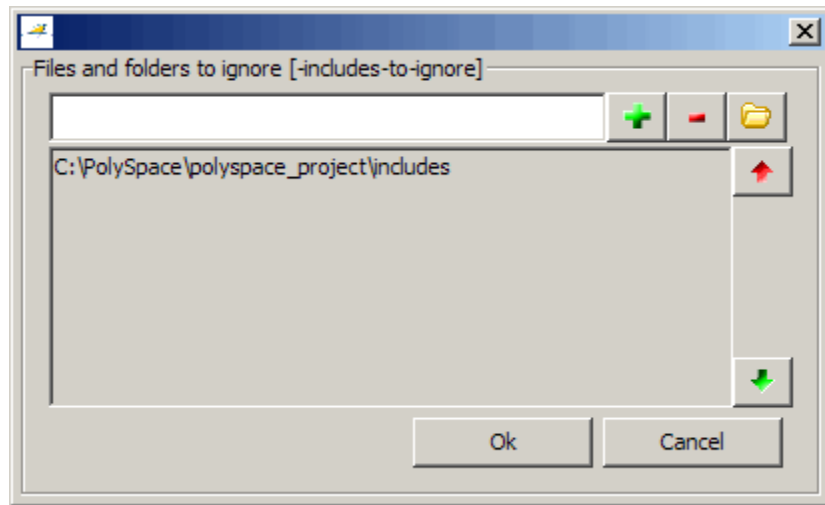
2 Click the folder icon ![folder icon].

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

3 Select the files or folders you want to exclude.

4 Click **OK**.

The select files and folders appear in the list of files to ignore.

5 Click **OK** to close the dialog box.

Configuring Text and XML Editors

Before you check MISRA C++ rules, you should configure your text and XML editors in the Launcher. Configuring text and XML editors in the Launcher allows you to view source files and MISRA C++ reports directly from the MISRA C++ log in the launcher.

To configure your text and .XML editors:

1 Select **Edit > Preferences**.

The Preferences dialog box opens.

2 Select the **Editors** tab.

The Editors tab opens.
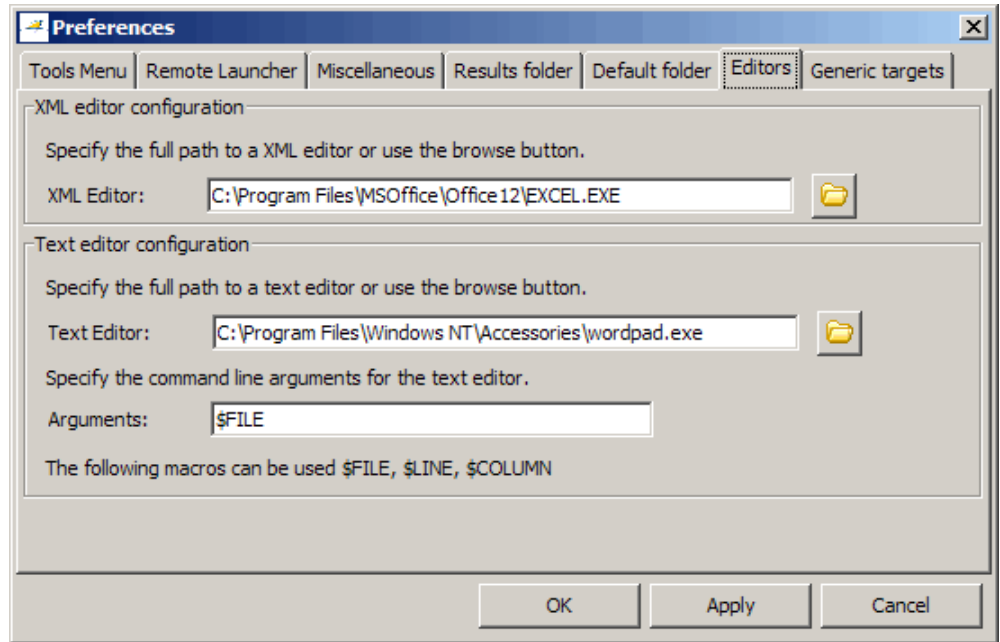
3 Specify an XML editor to use to view MISRA C++ reports. For example:

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

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

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

5 Specify command line arguments for the text editor. For example:

```
$FILE
```

6 Click **OK**.

Running a Verification with MISRA C++ Checking

- "Starting the Verification" on page 12-9
- "Examining the MISRA C++ Log" on page 12-10

Starting the Verification

When you run a verification with the **Check MISRA C++ rules** option selected, the verification checks most of the MISRA C++ rules during the compile phase. If there is a violation of a rule with state `Error`, the verification stops.

Note Some rules address run-time errors.

To start the verification:

1 Click the **Start** button ▶ Start.

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

3 If the verification fails because of MISRA C++ violations. A message dialog box appears.
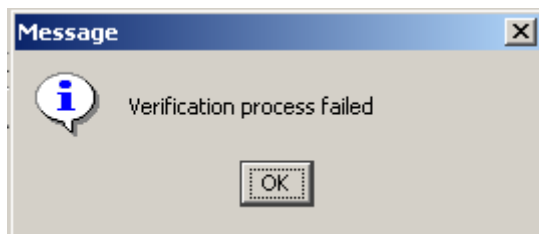
4 Click **OK**.

Examining the MISRA C++ Log

To examine the MISRA C++ violations:

1 Click the **MISRA C++** button in the log area of the Launcher window.

A list of MISRA C++ violations appear in the log part of the window.

2 Click on any of the violations to see a description of the violated rule, the full path of the file in which the violation was found, and the source code containing the violation.

3 Right click the row containing the violation of rule 0–1–2 , then select Open Source File.

Status	Rule	File	Line	Col	Justified
？	5-0-6	training.cpp	21	6	
！	0-				
？	5-				
？	5-				

⊊≣ Open Source File
Add Pre-Justification to Clipboard
Open MISRA-CPP Report
🏗 Configure Editor

The source file opens in your text editor.

Note You must configure a text editor before you can open source files.
See "Configuring Text and XML Editors" on page 12-16.

---

4 Correct any violations reported in the log, then run the verification again
to ensure compliance with all MISRA C++ rules.

Checking JSF++ Coding Rules

In this section...
"PolySpace JSF C++ Checker Overview" on page 12-12
"Setting Up JSF++ Checking" on page 12-12
"Running a Verification with JSF++ Checking" on page 12-18

PolySpace JSF C++ Checker Overview

The PolySpace JSF C++ checker helps you comply with the Joint Strike Fighter Air Vehicle C++ coding standards (JSF++). These coding standards were developed by Lockheed Martin® for the JSF program, and are designed to improve the robustness of C++ code, and improve maintainability.

The PolySpace JSF C++ checker enables PolySpace software to provide messages when JSF++ rules are not respected. Most messages are reported during the compile phase of a verification.

Note The PolySpace JSF C++ checker is based on JSF++:2005.
For more information on these coding standards, see
http://www.jsf.mil/downloads/documents/JSF_AV_C++_Coding_Standards_Rev_C.doc.

Setting Up JSF++ Checking

- "Activating the JSF C++ Checker" on page 12-12
- "Creating a JSF++ Rules File" on page 12-13
- "Excluding Files from JSF++ Checking" on page 12-15
- "Configuring Text and XML Editors" on page 12-16

Activating the JSF C++ Checker

You activate the JSF C++ checker using the options `jsf-coding-rules` and `includes-to-ignore`. These options can be set at the command line, or through the PolySpace Launcher user interface.

To activate the JSF C++ Checker:

1 Open the PolySpace Client for C/C++ window by double-clicking the PolySpace Launcher icon on your desktop:

2 Open the project you want to use.

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

The software displays the JSF C++ rules checker options, `-jsf-coding-rules` and `-includes-to-ignore`.

⊟ Coding rules checker			
⊟ Check JSF C++ rules	☑		
JSF C++ rules configuration		...	-jsf-coding-rules
⊟ Check MISRA C++ rules	☐		
MISRA C++ rules configuration		...	-misra-cpp
Files and folders to ignore		...	-includes-to-ignore

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

4 Select the **Check JSF C++: rules** check box.

Creating a JSF++ Rules File

You must have a rules file to run a verification with JSF++ checking. You can use an existing file or create a new one.

To create a new rules file:

1 Click the browse button ... to the right of the **JSF C++ rules configuration** option.

The New File window opens, allowing you to create a new JSF++ rules file, or open an existing file.
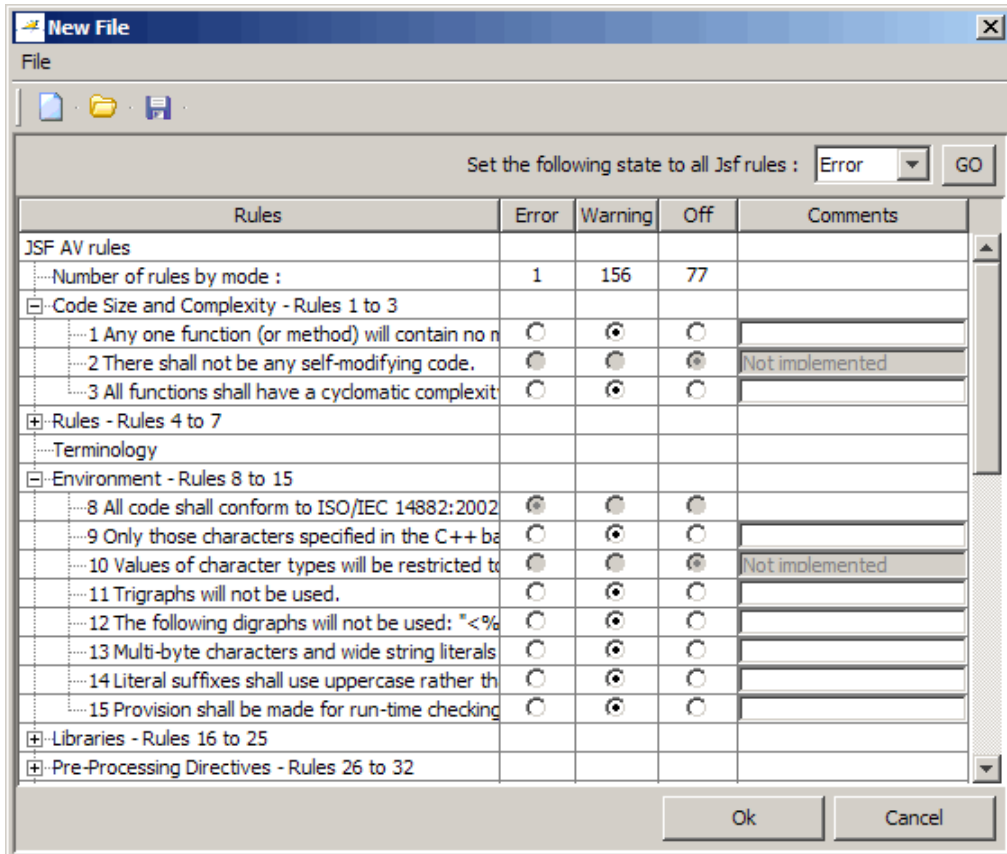
2 For each JSF++ rule, specify one of these states:

State	Causes the verification to...
Error	End after the compile phase when this rule is violated.
Warning	Display warning message and continue verification when this rule is violated.
Off	Skip checking of this rule.

> **Note** The default state for most rules is Warning. The state for rules that
> have not yet been implemented is Off. Some rules always have state Error
> (you cannot change the state of these).

3 Click **OK** to save the rules and close the window.

The **Save as** dialog box opens.

4 In **File**, enter a name for your rules file.

5 Click **OK** to save the file and close the dialog box.

> **Note** If your project uses a dialect other than ISO, some JSF++ coding
> rules may not be completely checked. For example, AV Rule 8: "All code
> shall conform to ISO/IEC 14882:2002(E) standard C++."

Excluding Files from JSF++ Checking

You can exclude files from JSF++ checking. For example, you may want to
exclude some included files.

To exclude files from JSF++ checking:

1 Click the button [...] to the right of the **Files and folders to ignore** option.

The Files and folders to ignore (includes-to-ignore) dialog box opens.
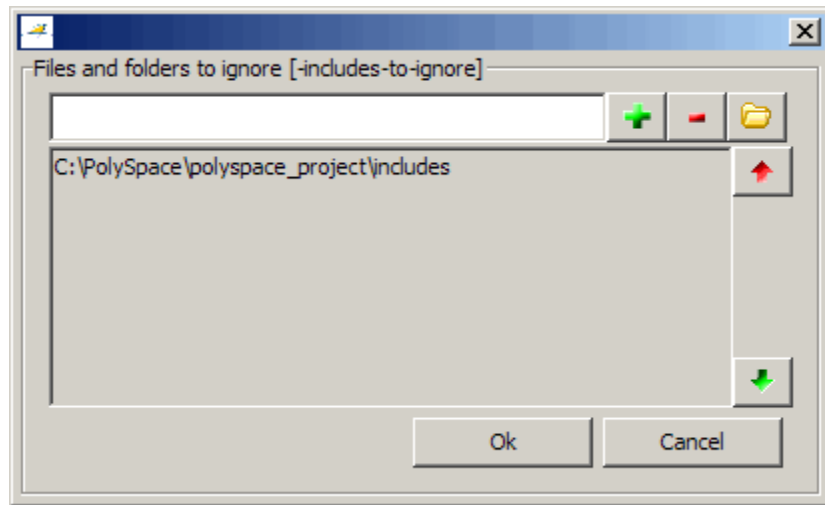
2 Click the folder icon .

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

3 Select the files or folders you want to exclude.

4 Click **OK**.

The select files and folders appear in the list of files to ignore.

5 Click **OK** to close the dialog box.

Configuring Text and XML Editors

Before you check JSF++ rules, you should configure your text and XML editors in the Launcher. Configuring text and XML editors in the Launcher allows you to view source files and JSF C++ reports directly from the JSF C++ log in the launcher.

To configure your text and .XML editors:

1 Select **Edit > Preferences**.

The Preferences dialog box opens.

2 Select the **Editors** tab.

The Editors tab opens.
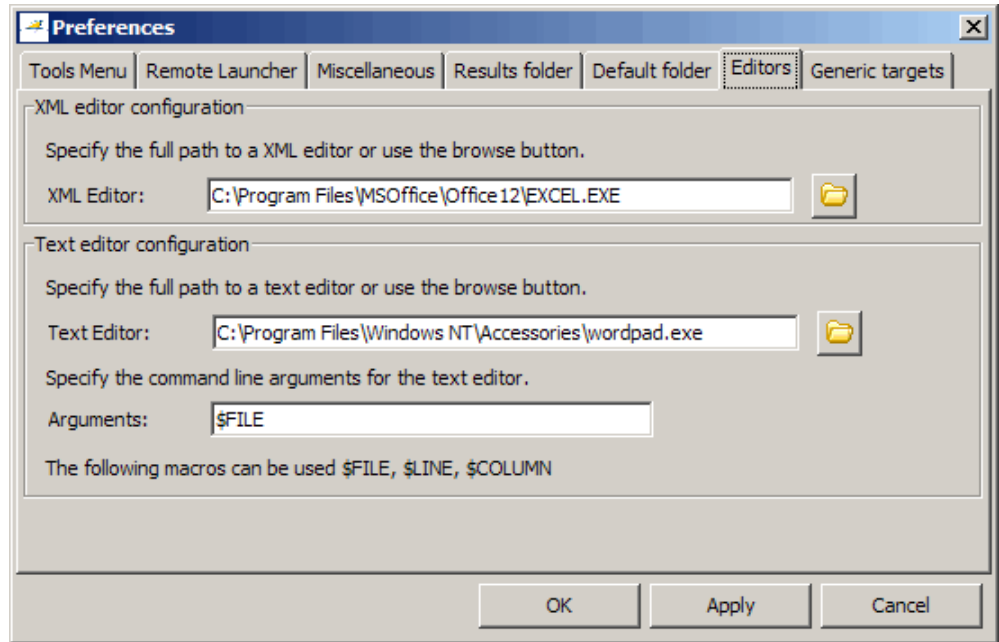
3 Specify an XML editor to use to view JSF C++ reports. For example:

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

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

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

5 Specify command line arguments for the text editor. For example:

```
$FILE
```

6 Click **OK**.

Running a Verification with JSF++ Checking

- "Starting the Verification" on page 12-18
- "Examining the JSF C++ Log" on page 12-18

Starting the Verification

When you run a verification with the **Check JSF C++ rules** option selected, the verification checks most of the JSF++ rules during the compile phase. If there is a violation of a rule with state `Error`, the verification stops.

Note Some rules address run-time errors.

To start the verification:

1 Click the **Start** button [▶ Start].

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

3 If the verification fails because of JSF++ violations. A message dialog box appears.
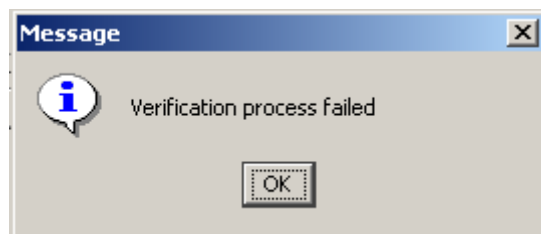
4 Click **OK**.

Examining the JSF C++ Log

To examine the JSF++ violations:

1 Click the **JSF C++** button in the log area of the Launcher window.

A list of JSF++ violations appear in the log part of the window.

	Filter			Hide justifed violated rules					**i**
Compile	Status	Rule	File	Line	Col	Justified	Acronym	User Acronym	Justification
JSF C++	?	180	training.cpp	21	8				
Stats	!	191	training.cpp	46	19				
Full Log	?	180	training.cpp	109	15				
	?	180	training.cpp	142	38				

2 Click on any of the violations to see a description of the violated rule, the full path of the file in which the violation was found, and the source code containing the violation.

Filter		Hide justifed violated rules						**i**
Status	Rule	File	Line	Col	Justified	Acronym	User Acronym	Justification
?	180	training.cpp	21	8				
!	191	training.cpp	46	19				
?	180	training.cpp	109	15				
?	180	training.cpp	142	38				

Detail

```
Rule: 191 (Error): The break statement shall not be used (except to terminate the cases of a switch statement).

File: C:\PolySpace\polyspace_project\sources\training.cpp line 46 (column 19)

Source code
|      if (y > big) break;
                    ^
```
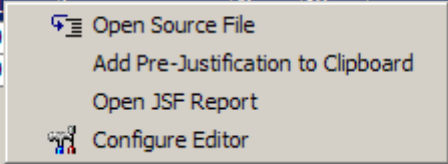
3 Right click the row containing the violation of rule 191 , then select Open Source File.

Status	Rule	File	Line	Col	Justified
?	180	training.cpp	21	8	
!	191				
?	180				
?	180				

Open Source File

Add Pre-Justification to Clipboard

Open JSF Report

Configure Editor

The source file opens in your text editor.

Note You must configure a text editor before you can open source files. See "Configuring Text and XML Editors" on page 12-16.

---

4 Correct any violations reported in the log, then run the verification again to ensure compliance with all JSF++ rules.

Supported Coding Rules

In this section...

Supported MISRA C++ Coding Rules

Language Independent Issues

N.	MISRA Definition	Comments
0-1-1	A project shall not contain unreachable code.	
0-1-2	A project shall not contain infeasible paths.	
0-1-7	The value returned by a function having a non- void return type that is not an overloaded operator shall always be used.	
0-1-10	Every defined function shall be called at least once.	Detects if static functions are not called in their translation unit. Other cases are detected by the Verifier.

General

N.	MISRA Definition	Comments
1-0-1	All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".	

Lexical Conventions

N.	MISRA Definition	Comments
2-3-1	Trigraphs shall not be used.	
2-5-1	Digraphs should not be used.	
2-7-1	The character sequence /* shall not be used within a C-style comment.	

N.	MISRA Definition	Comments
2-10-1	Different identifiers shall be typographically unambiguous.	
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.	No detection for logical scopes: fields or member functions hiding outer scopes identifiers or hiding ancestors members.
2-10-3	A typedef name (including qualification, if any) shall be a unique identifier.	No detection accross namespaces.
2-10-4	A class, union or enum name (including qualification, if any) shall be a unique identifier.	No detection accross namespaces.
2-10-5	The identifier name of a non-member object or function with static storage duration should not be reused.	For functions the detection is only on the definition where there is a declaration.
2-10-6	If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.	If the identifier is a function and the function is both declared and defined then the violation is reported only once.
2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.	
2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.	
2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.	
2-13-4	Literal suffixes shall be upper case.	

Basic Concepts

N.	MISRA Definition	Comments
3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.	
3-1-2	Functions shall not be declared at block scope.	
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.	
3-2-1	All declarations of an object or function shall have compatible types.	
3-2-2	The One Definition Rule shall not be violated.	Report type, template, and inline function defined in source file
3-2-3	A type, object or function that is used in multiple translation units shall be declared in one and only one file.	
3-2-4	An identifier with external linkage shall have exactly one definition.	
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.	
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.	
3-9-1	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.	Comparison is done between current declaration and last seen declaration.

N.	MISRA Definition	Comments
3-9-2	typedefs that indicate size and signedness should be used in place of the basic numerical types.	No detection in non-instantiated templates.
3-9-3	The underlying bit representations of floating-point values shall not be used.	

Standard Conversions

N.	MISRA Definition	Comments
4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.	
4-5-3	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator. N	

Expressions

N.	MISRA Definition	Comments
5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.	
5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions.	
5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type.	Assumes that ptrdiff_t is signed integer

N.	MISRA Definition	Comments
5-0-4	An implicit integral conversion shall not change the signedness of the underlying type.	Assumes that ptrdiff_t is signed integerIf the conversion is to a narrower integer with a different sign then Misra Cpp 5-0-4 takes precedence over Misra Cpp 5-0-6.
5-0-5	There shall be no implicit floating-integral conversions.	This rule takes precedence over 5-0-4 and 5-0-6 if they apply at the same time.
5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type.	If the conversion is to a narrower integer with a different sign then Misra Cpp 5-0-4 takes precedence over Misra Cpp 5-0-6.
5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression.	
5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.	
5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.	
5-0-10	If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	
5-0-15	Array indexing shall be the only form of pointer arithmetic.	Warning on operations on pointers. (p+I, I+p and p-I, where p is a pointer and I an integer, p[i] accepted).
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.	Report when relational operator are used on pointers types (casts ignored).
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.	
5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type.	

N.	MISRA Definition	Comments
5-2-1	Each operand of a logical && or || shall be a postfix - expression.	During preprocessing, violations of this rule are detected on the expressions in #if directives. Allowed exception on associativity (a && b && c), (a || b || c).
5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.	
5-2-3	Casts from a base class to a derived class should not be performed on polymorphic types.	
5-2-4	C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.	
5-2-5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.	
5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.	No violation if pointer types of operand and target are identical.
5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.	"Extended to all pointer conversions including between pointer to struct object and pointer to type of the first member of the struct type. Indirect conversions through non-pointer type (e.g. int) are not detected."
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.	Exception on zero constants. Objects with pointer type include objects with pointer to function type.
5-2-9	A cast should not convert a pointer type to an integral type.	

N.	MISRA Definition	Comments
5-2-10	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	
5-2-11	The comma operator, && operator and the || operator shall not be overloaded.	
5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer.	
5-3-1	Each operand of the ! operator, the logical && or the logical || operators shall have type bool.	
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	
5-3-3	The unary & operator shall not be overloaded.	
5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects.	No warning on volatile accesses and function calls
5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	
5-14-1	The right hand operand of a logical && or || operator shall not contain side effects.	No warning on volatile accesses and function calls.
5-18-1	The comma operator shall not be used.	
5-19-1	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	

Statements

N.	MISRA Definition	Comments
6-2-1	Assignment operators shall not be used in sub-expressions.	
6-2-2-	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.	
6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character.	
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.	
6-4-1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.	
6-4-2	All if ... else if constructs shall be terminated with an else clause.	Detects also cases where the last if is in the block of the last else (same behavior as JSF, stricter than MISRA C). Example: "if ... else { if ...{}}" raises the rule
6-4-3	A switch statement shall be a well-formed switch statement.	Return statements are considered as jump statements.
6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	
6-4-5	An unconditional throw or break statement shall terminate every non - empty switch-clause.	

N.	MISRA Definition	Comments
6-4-6	The final clause of a switch statement shall be the default-clause.	
6-4-7	The condition of a switch statement shall not have bool type.	
6-4-8	Every switch statement shall have at least one case-clause.	
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.	
6-5-3	The loop-counter shall not be modified within condition or statement.	Detect only direct assignments if for_index is known (see 6-5-1).
6-6-3	The continue statement shall only be used within a well-formed for loop.	Assumes 6.5.1 to 6.5.6: so it is implemented only for supported 6_5_x rules.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.	
6-6-5	A function shall have a single point of exit at the end of the function.	At most one return not necessarily as last statement for void functions.

Declarations

N.	MISRA Definition	Comments
7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations.	
7-3-3	There shall be no unnamed namespaces in header files.	
7-3-4	using-directives shall not be used.	
7-3-5	Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.	

N.	MISRA Definition	Comments
7-3-6	using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.	
7-4-3	Assembly language shall be encapsulated and isolated.	
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.	
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	
7-5-3	A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.	
7-5-4	Functions should not call themselves, either directly or indirectly.	

Declarators

N.	MISRA Definition	Comments
8-0-1	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.	
8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.	

N.	MISRA Definition	Comments
8-4-1	Functions shall not be defined using the ellipsis notation.	
8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.	
8-4-3	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	
8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.	
8-5-1	All variables shall have a defined value before they are used.	NIV given by verifier and error messages for obvious cases
8-5-2	Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.	
8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	

Classes

N.	MISRA Definition	Comments
9-3-1	const member functions shall not return non-const pointers or references to class-data.	Class-data for a class is restricted to all non-static member data.
9-3-2	Member functions shall not return non-const handles to class-data.	Class-data for a class is restricted to all non-static member data.
9-5-1	Unions shall not be used.	
9-6-2	Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.	

N.	MISRA Definition	Comments
9-6-3	Bit-fields shall not have enum type.	
9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit.	

Derived Classes

N.	MISRA Definition	Comments
10-1-1	Classes should not be derived from virtual bases.	
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.	Assumes 10.1.1 not required
10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.	
10-2-1	All accessible entity names within a multiple inheritance hierarchy should be unique.	No detection between entities of different kinds (member functions against data members, …).
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.	Member functions that are virtual by inheritance are also detected.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword. l.	
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtua	

Member Access Control

N.	MISRA Definition	Comments
11-0-1	Member data in non- POD class types shall be private.	

Special Member Functions

N.	MISRA Definition	Comments
12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.	
12-1-2	All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.	
12-1-3	All constructors that are callable with a single argument of fundamental type shall be declared explicit.	
12-8-2	The copy assignment operator shall be declared protected or private in an abstract class.	

Templates

N.	MISRA Definition	Comments
14-5-2	A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.	
14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.	
14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->	
14-6-2	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.	

N.	MISRA Definition	Comments
14-7-3	All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.	
14-8-1	Overloaded function templates shall not be explicitly specialized.	All specializations of overloaded templates are rejected even if overloading occurs after the call.
14-8-2	The viable function set for a function call should either contain no function specializations, or only contain function specializations.	

Exception Handling

N.	MISRA Definition	Comments
15-0-2	An exception object should not have pointer type.	NULL not detected (see 15-1-2).
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.	
15-1-2	NULL shall not be thrown explicitly.	
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.	
15-3-2	There should be at least one exception handler to catch all otherwise unhandled exceptions.	Detect that there is no try/catch in the main and that the catch does not handle all exceptions. No detection if no "main" (desktop mode?).
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.	

N.	MISRA Definition	Comments
15-3-5	A class type exception shall always be caught by reference.	
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.	
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.	
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.	
15-5-1	A class destructor shall not exit with an exception.	Limit detection to throw and catch that are internals to the destructor; rethrows are partially processed; no detections in nested handlers
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).	Limit detection to throw that are internals to the function; rethrows are partially processed; no detections in nested handlers.

Preprocessing Directives

N.	MISRA Definition	Comments
16-0-1	#include directives in a file shall only be preceded by other preprocessor directives or comments.	
16-0-2	Macros shall only be #define 'd or #undef 'd in the global namespace.	

N.	MISRA Definition	Comments
16-0-3	#undef shall not be used.	
16-0-4	Function-like macros shall not be defined.	
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	
16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.	
16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.	
16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.	
16-1-1	The defined preprocessor operator shall only be used in one of the two standard forms.	
16-1-2	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	
16-2-1	The pre-processor shall only be used for file inclusion and include guards.	The rule is raised for #ifdef/#define if the file is not an include file.
16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.	
16-2-3	Include guards shall be provided.	
16-2-4	The ', ", /* or // characters shall not occur in a header file name.	
16-2-5	The \ character should not occur in a header file name.	

N.	MISRA Definition	Comments
16-2-6	The #include directive shall be followed by either a <filename> or "filename" sequence.	
16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.	
16-3-2	The # and ## operators should not be used.	

Library Introduction

N.	MISRA Definition	Comments
17-0-1	Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.	
17-0-2	The names of standard library macros and objects shall not be reused.	
17-0-5	The setjmp macro and the longjmp function shall not be used.	

Language Support Library

N.	MISRA Definition	Comments
18-0-2	The library functions atof, atoi and atol from library <cstdlib> shall not be used.	
18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.	The option -dialect iso must be used to detect violations (e.g.:exit).
18-0-4	The time handling functions of library <ctime> shall not be used.	
18-2-1	The macro offsetof shall not be used.	

N.	MISRA Definition	Comments
18-4-1	Dynamic heap memory allocation shall not be used.	
18-7-1	The signal handling facilities of <csignal> shall not be used.	

Diagnostic Library

N.	MISRA Definition	Comments
19-3-1	The error indicator errno shall not be used.	

Input/output Library

N.	MISRA Definition	Comments
27-0-1	The stream input/output library <cstdio> shall not be used.	

MISRA C++ Rules Not Checked

Language Independent Issues

N.	MISRA Definition	Comments
0–1–3	A project shall not contain unused variables.	
0-1-4	A project shall not contain non-volatile POD variables having only one use.	
0-1-5	A project shall not contain unused type declarations.	
0-1-6	A project shall not contain instances of non-volatile variables being given values that are never subsequently used.	
0-1-8	All functions with void return type shall have external side effect(s).	
0-1-9	There shall be no dead code.	
0-1-11	There shall be no unused parameters (named or unnamed) in non- virtual functions.	
0-1-12	There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.	
0-2-1	An object shall not be assigned to an overlapping object.	

N.	MISRA Definition	Comments
0-3-1	Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.	
0-3-2	If a function generates error information, then that error information shall be tested.	
0-4-1	Use of scaled-integer or fixed-point arithmetic shall be documented.	
0-4-2	Use of floating-point arithmetic shall be documented.	
0-4-3	Floating-point implementations shall comply with a defined floating-point standard.	

General

N.	MISRA Definition	Comments
1-0-2	Multiple compilers shall only be used if they have a common, defined interface.	
1-0-3	The implementation of integer division in the chosen compiler shall be determined and documented.	

Lexical Conventions

N.	MISRA Definition	Comments
2-2-1	The character set and the corresponding encoding shall be documented.	
2-7-2	Sections of code shall not be "commented out" using C-style comments.	
2-7-3	Sections of code should not be "commented out" using C++ comments.	
2-13-5	Narrow and wide string literals shall not be concatenated.	

Basic Concepts

N.	MISRA Definition	Comments
3-3-1	Objects or functions with external linkage shall be declared in a header file.	

Standard Conversions

N.	MISRA Definition	Comments
4-5-2	Expressions with type enum shall not be used as operands to built- in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.	
4-10-1	ULL shall not be used as an integer value.	
4-10-2	Literal zero (0) shall not be used as the null-pointer-constant.	

Expressions

N.	MISRA Definition	Comments
5-0-11	The plain char type shall only be used for the storage and use of character values.	
5-0-12	signed char and unsigned char type shall only be used for the storage and use of numeric values.	
5-0-13	The condition of an if-statement and the condition of an iteration- statement shall have type bool.	
5-0-14	The first operand of a conditional-operator shall have type bool.	
5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	
5-0-17	Subtraction between pointers shall only be applied to pointers that address elements of the same array.	
5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type.	
5-17-1	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.	

Statements

N.	MISRA Definition	Comments
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.	
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.	
6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression.	
6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.	
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.	
6-6-2	The goto statement shall jump to a label declared later in the same function body.	

Declarations

N.	MISRA Definition	Comments
7-1-1	A variable which is not modified shall be const qualified.	
7-1-2	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.	
7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	

N.	MISRA Definition	Comments
7-3-2	The identifier main shall not be used for a function other than the global function main.	
7-4-1	All usage of assembler shall be documented.	
7-4-2	Assembler instructions shall only be introduced using the asm declaration.	

Classes

N.	MISRA Definition	Comments
9-3-3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.	
9-6-1	When the absolute positioning of bits representing a bit-field is required, then the behavior and packing of bit-fields shall be documented.	

Special Member Functions

N.	MISRA Definition	Comments
12-8-1	A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member.	

Templates

N.	MISRA Definition	Comments
14-5-1	A non-member generic function shall only be declared in a namespace that is not an associated namespace.	
14-7-1	All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.	
14-7-2	For any given template specialization, an explicit instantiation of the template with the template-arguments used in the specialization shall not render the program ill-formed.	

Exception Handling

N.	MISRA Definition	Comments
15-0-1	Exceptions shall only be used for error handling.	
15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	
15-3-1	Exceptions shall be raised only after start-up and before termination of the program.	
15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	
15-5-3	The terminate() function shall not be called implicitly.	

Preprocessing Directives

N.	MISRA Definition	Comments
16-6-1	All uses of the #pragma directive shall be documented.	

Library Introduction

N.	MISRA Definition	Comments
17-0-3	The names of standard library functions shall not be overridden.	
17-0-4	All library code shall conform to MISRA C++.	

Language Support Library

N.	MISRA Definition	Comments
18-0-1	The C library shall not be used.	
18-0-5	The unbounded functions of library <cstring> shall not be used.	

Supported JSF C++ Coding Rules

Code Size and Complexity

N.	JSF++ Definition	Comments
1	Any one function (or method) **will** contain no more than 200 logical source lines of code (L-SLOCs).	Message in log file: *<function name>* has *<num>* logical source lines of code.
3	All functions **shall** have a cyclomatic complexity number of 20 or less.	Message in log file: *<function name>* has cyclomatic complexity number equal to *<num>*

Environment

N.	JSF++ Definition	Comments
8	All code **shall** conform to ISO/IEC 14882:2002(E) standard C++.	Reports the compilation error message
9	Only those characters specified in the C++ basic source character set **will** be used.	
11	Trigraphs **will not** be used.	
12	The following digraphs **will not** be used: <%, %>, <:, :>, %:, %:%:.	Message in log file: The following digraph will not be used: *\<digraph\>* Reports the digraph. If the rule level is set to warning, the digraph will be allowed even if it is not supported in -dialect iso
13	Multi-byte characters and wide string literals **will not** be used.	Report L'c' and L"string" and use of wchar_t.
14	Literal suffixes **shall** use uppercase rather than lowercase letters.	
15	Provision **shall** be made for run-time checking (defensive programming).	Done with RTE checks in the Verifier.

Libraries

N.	JSF++ Definition	Comments
17	The error indicator errno **shall not** be used.	errno should not be used as a macro or a global with external "C" linkage.
18	The macro offsetof, in library <stddef.h>, **shall not** be used.	offsetof should not be used as a macro or a global with external "C" linkage.
19	<locale.h> and the setlocale function **shall not** be used.	setlocale and localeconv should not be used as a macro or a global with external "C" linkage.

N.	JSF++ Definition	Comments
20	The `setjmp` macro and the `longjmp` function **shall not** be used.	`setjmp` and `longjmp` should not be used as a macro or a global with external "C" linkage.
21	The signal handling facilities of `<signal.h>` **shall not** be used.	`signal` and `raise` should not be used as a macro or a global with external "C" linkage.
22	The input/output library `<stdio.h>` **shall not** be used.	all standard functions of `<stdio.h>` should not be used as a macro or a global with external "C" linkage.
23	The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>` **shall not** be used.	`atof`, `atoi` and `atol` should not be used as a macro or a global with external "C" linkage.
24	The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` **shall not** be used.	`abort`, `exit` `getenv` and `system` should not be used as a macro or a global with external "C" linkage.
25	The time handling functions of library `<time.h>` **shall not** be used.	`clock`, `difftime`, `mktime`, `asctime`, `ctime`, `gmtime`, `localtime` and `strftime` should not be used as a macro or a global with external "C" linkage.

Pre-Processing Directives

N.	JSF++ Definition	Comments
26	Only the following pre-processor directives **shall** be used: `#ifndef`, `#define`, `#endif`, `#include`.	
27	`#ifndef`, `#define` and `#endif` **will** be used to prevent multiple inclusions of the same header file. Other techniques to prevent the multiple inclusions of header files **will not** be used.	Detects the patterns `#if !defined`, `#pragma once`, `#ifdef`, and missing `#define`.

N.	JSF++ Definition	Comments
28	The `#ifndef` and `#endif` pre-processor directives **will** only be used as defined in AV Rule 27 to prevent multiple inclusions of the same header file.	Detects any use that does not comply with AV Rule 27. Assuming 35/27 is not violated, reports only `#ifndef`.
29	The `#define` pre-processor directive **shall not** be used to create inline macros. Inline functions shall be used instead.	Rule is split into two parts: the definition of a macro function (29.def) and the call of a macrofunction (29.use).Messages in log file: • `29.1 :  The #define pre-processor directive shall not be used to create inline macros.` • `29.2 :  Inline functions shall be used intead of inline macros`
30	The `#define` pre-processor directive **shall not** be used to define constant values. Instead, the `const` qualifier **shall** be applied to variable declarations to specify constant values.	Reports `#define` of simple constants.
31	The `#define` pre-processor directive **will** only be used as part of the technique to prevent multiple inclusions of the same header file.	Detects use of `#define` that are not used to guard for multiple inclusion, assuming that rules 35 and 27 are not violated.
32	The `#include` pre-processor directive **will** only be used to include header (*.h) files.	

Header Files

N.	JSF++ Definition	Comments
33	The #include directive **shall** use the <filename.h> notation to include header files.	
35	A header file **will** contain a mechanism that prevents multiple inclusions of itself.	
39	Header files (*.h) **will not** contain non-const variable definitions or function definitions.	Reports definitions of global variables / function in header.

Style

N.	JSF++ Definition	Comments
40	Every implementation file shall include the header files that uniquely define the inline functions, types, and templates used.	Reports when type, template, or inline function is defined in source file.
41	Source lines **will** be kept to a length of 120 characters or less.	
42	Each expression-statement **will** be on a separate line.	Reports when two consecutive expression statements are on the same line.
43	Tabs **should** be avoided.	
44	All indentations will be at least two spaces and be consistent within the same source file.	Reports when a statement indentation is not at least two spaces more than the statement containing it. Does not report bad indentation between opening braces following if/else, do/while, for, and while statements. NB: in final release it will accept any indentation
46	User-specified identifiers (internal and external) **will not** rely on significance of more than 64 characters.	

N.	JSF++ Definition	Comments
47	Identifiers **will not** begin with the underscore character '_'.	
48	Identifiers **will not** differ by: • Only a mixture of case • The presence/absence of the underscore character • The interchange of the letter 'O'; with the number '0' or the letter 'D' • The interchange of the letter 'I'; with the number '1' or the letter 'l' • The interchange of the letter 'S' with the number '5' • The interchange of the letter 'Z' with the number 2 • The interchange of the letter 'n' with the letter 'h'	Checked regardless of scope. Not checked between macros and other identifiers. Messages in log file: • `Identifier "Idf1" (file1.cpp line l1 column c1) and "Idf2" (file2.h line l2 column c2) only differ by the presence/absence of the underscore character.` • `Identifier "Idf1" (file1.cpp line l1 column c1) and "Idf2" (file2.h line l2 column c2) only differ by a mixture of case.` • `Identifier "Idf1" (file1.cpp line l1 column c1) and "Idf2" (file2.h line l2 column c2) only differ by letter 'O', with the number 'O'.`
50	The first word of the name of a class, structure, namespace, enumeration, or type created with `typedef` **will** begin with an uppercase letter. All others letters **will** be lowercase.	Messages in log file: • `The first word of the name of a class will begin with an uppercase letter.` • `The first word of the namespace of a class will begin with an uppercase letter.`
51	All letters contained in function and variables names **will** be composed entirely of lowercase letters.	Messages in log file: • `All letters contained in variable names will be composed entirely of lowercase letters.` • `All letters contained in function names will be composed entirely of lowercase letters.`

N.	JSF++ Definition	Comments
52	Identifiers for constant and enumerator values **shall** be lowercase.	Messages in log file: • `Identifier for enumerator value shall be lowercase.` • `Identifier for template constant parameter shall be lowercase.`
53	Header files **will** always have file name extension of ".h".	`.H` is allowed if you set the option `-dos`.
53.1	The following character sequences **shall** not appear in header file names: ', \, /*, //, or ".	
54	Implementation files **will** always have a file name extension of ".cpp".	Not case sensitive if you set the option `-dos`.
57	The public, protected, and private sections of a class **will** be declared in that order.	
58	When declaring and defining functions with more than two parameters, the leading parenthesis and the first argument **will** be written on the same line as the function name. Each additional argument will be written on a separate line (with the closing parenthesis directly after the last argument).	Detects that two parameters are not on the same line, The first parameter should be on the same line as function name. Does not check for the closing parenthesis.

N.	JSF++ Definition	Comments
59	The statements forming the body of an if, else if, else, while, do ... while or for statement **shall** always be enclosed in braces, even if the braces form an empty block.	Messages in log file: • The statements forming the body of an if statement shall always be enclosed in braces. • The statements forming the body of an else statement shall always be enclosed in braces. • The statements forming the body of a while statement shall always be enclosed in braces. • The statements forming the body of a do ... while statement shall always be enclosed in braces. • The statements forming the body of a for statement shall always be enclosed in braces.
60	Braces ("{}") which enclose a block **will** be placed in the same column, on separate lines directly before and after the block.	Detects that statement-block braces should be in the same columns.
61	Braces ("{}") which enclose a block **will** have nothing else on the line except comments.	

N.	JSF++ Definition	Comments
62	The dereference operator '*' and the address-of operator '&' will be directly connected with the type-specifier.	Reports when there is a space between type and "*" "&" for variables, parameters and fields declaration.
63	Spaces will not be used around '.' or '->', nor between unary operators and operands.	Reports when the following characters are not directly connected to a white space: • . • -> • ! • ~ • - • ++ • — **Note** A violation will be reported for "." used in float/double definition.

Classes

N.	JSF++ Definition	Comments
67	Public and protected data **should** only be used in structs - not classes.	
68	Unneeded implicitly generated member functions shall be explicitly disallowed.	Reports when default constructor, assignment operator, copy constructor or destructor is not declared.
71.1	A class's virtual functions shall not be invoked from its destructor or any of its constructors.	Reports when a constructor or destructor directly calls a virtual function.

N.	JSF++ Definition	Comments
74	Initialization of nonstatic class members **will** be performed through the member initialization list rather than through assignment in the body of a constructor.	All data should be initialized in the initialization list except for array. Does not report that an assignment exists in `ctor` body.Message in log file: `Initialization of nonstatic class members "<field>" will be performed through the member initialization list.`
75	Members of the initialization list **shall** be listed in the order in which they are declared in the class.	
76	A copy constructor and an assignment operator **shall** be declared for classes that contain pointers to data items or nontrivial destructors.	Messages in log file: • `no copy constructor and no copy assign` • `no copy constructor` • `no copy assign`
77.1	The definition of a member function **shall not** contain default arguments that produce a signature identical to that of the implicitly-declared copy constructor for the corresponding class/structure.	Does not report when an explicit copy constructor exists.
78	All base classes with a virtual function **shall** define a virtual destructor.	
79	All resources acquired by a class shall be released by the class's destructor.	Reports when the number of "new" called in a constructor is greater than the number of "delete" called in its destructor. **Note** A violation is raised even if "new" is done in a "if/else".

N.	JSF++ Definition	Comments
81	The assignment operator shall handle self-assignment correctly.	Reports when copy assignment body does not begin with "if (this != arg)" A violation is not raised if an empty `else` statement follows the `if`, or the body contains only a return statement.
		A violation is raised when the `if` statement is followed by a statement other than the return statement.
82	An assignment operator **shall** return a reference to `*this`.	The following operators should return `*this` on method, and `*first_arg` on plain function.

		`operator=`
		`operator+=`
		`operator-=`
		`operator*=`
		`operator >>=`
		`operator <<=`
		`operator /=`
		`operator %=`
		`operator |=`
		`operator &=`
		`operator ^=`
		`prefix operator++`
		`prefix operator--`

		Does not report when no return exists.
		No special message if type does not match.
		Messages in log file:
		• `An assignment operator shall return a reference to *this.`
		• `An assignment operator shall return a reference to its first arg.`

N.	JSF++ Definition	Comments
83	An assignment operator shall assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).	Reports when a copy assignment does not assign all data members. In a derived class, it also reports when a copy assignment does not call inherited copy assignments.
88	Multiple inheritance **shall** only be allowed in the following restricted form: n interfaces plus m private implementations, plus at most one protected implementation.	Messages in log file: • `Multiple inheritance on public implementation shall not be allowed:` *`<public_base_class>`* `is not an interface.` • `Multiple inheritance on protected implementation shall not be allowed :` *`<protected_base_class_1>`* • *`<protected_base_class_2>`* `are not interfaces.`
88.1	A stateful virtual base **shall** be explicitly declared in each derived class that accesses it.	
89	A base class **shall not** be both virtual and non-virtual in the same hierarchy.	
94	An inherited nonvirtual function **shall not** be redefined in a derived class.	Does not report for destructor.Message in log file: `Inherited nonvirtual function %s shall not be redefined in a derived class.`
95	An inherited default parameter **shall never** be redefined.	
96	Arrays **shall not** be treated polymorphically.	Reports pointer arithmetic and array like access on expressions whose pointed type is used as a base class.

N.	JSF++ Definition	Comments
97	Arrays **shall not** be used in interface.	Only to prevent array-to-pointer-decay, Not checked on private methods
97.1	Neither operand of an equality operator (== or !=) **shall** be a pointer to a virtual member function.	Reports == and != on pointer to member function of polymorphic classes (cannot determine statically if it is virtual or not), except when one argument is the null constant.

Namespaces

N.	JSF++ Definition	Comments
98	Every nonlocal name, except main(), **should** be placed in some namespace.	
99	Namespaces **will not** be nested more than two levels deep.	

Templates

N.	JSF++ Definition	Comments
104	A template specialization **shall** be declared before its use.	Reports the actual compilation error message.

Functions

N.	JSF++ Definition	Comments
107	Functions **shall** always be declared at file scope.	
108	Functions with variable numbers of arguments **shall not** be used.	

N.	JSF++ Definition	Comments
109	A function definition should not be placed in a class specification unless the function is intended to be inlined.	Reports when there is no "inline" in the definition of a member function inside the class definition.
110	Functions with more than 7 arguments **will not** be used.	
111	A function **shall not** return a pointer or reference to a non-static local object.	Simple cases without alias effect detected.
113	Functions **will** have a single exit point.	Reports first return, or once per function.
114	All exit points of value-returning functions **shall** be through return statements.	
116	Small, concrete-type arguments (two or three words in size) **should** be passed by value if changes made to formal parameters should not be reflected in the calling function.	Report constant parameters references with `sizeof <= 2 * sizeof(int)`. Does not report for copy-constructor.
119	Functions **shall** not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	Direct recursion is reported statically. Indirect recursion reported through Verifier. Message in log file: `Function <F> shall not call directly itself.`
121	Only functions with 1 or 2 statements **should** be considered candidates for inline functions.	Reports inline functions with more than 2 statements.

Comments

N.	JSF++ Definition	Comments
126	Only valid C++ style comments (//) **shall** be used.	
133	Every source file will be documented with an introductory comment that provides information on the file name, its contents, and any program-required information (e.g. legal statements, copyright information, etc).	Reports when a file does not begin with two comment lines.

Declarations and Definitions

N.	JSF++ Definition	Comments
135	Identifiers in an inner scope **shall not** use the same name as an identifier in an outer scope, and therefore hide that identifier.	
136	Declarations should be at the smallest feasible scope.	Reports when: • A global variable is used in only one function. • A local variable is not used in a statement (expr, return, init ...) of the same level of its declaration (in the same block) or is not used in two sub-statements of its declaration. **Note** • Non-used variables are reported. • Initializations at definition are ignored (not considered an access)

N.	JSF++ Definition	Comments
137	All declarations at file scope should be static where possible.	
138	Identifiers **shall not** simultaneously have both internal and external linkage in the same translation unit.	
139	External objects will not be declared in more than one file.	Reports all duplicate declarations inside a translation unit. Reports when the declaration localization is not the same in
140	The register storage class specifier **shall not** be used.	
141	A class, structure, or enumeration **will not** be declared in the definition of its type.	

Initialization

N.	JSF++ Definition	Comments
142	All variables **shall** be initialized before use.	Done with **NIV** and **LOCAL_NIV** checks in the Verifier.
144	Braces **shall** be used to indicate and match the structure in the non-zero initialization of arrays and structures.	This covers partial initialization.
145	In an enumerator list, the '=' construct **shall not** be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	Generates one report for an enumerator list.

Types

N.	JSF++ Definition	Comments
147	The underlying bit representations of floating point numbers **shall not** be used in any way by the programmer.	Reports on casts with float pointers (except with void*).
148	Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.	Reports when non enumeration types are used in switches.

Constants

N.	JSF++ Definition	Comments
149	Octal constants (other than zero) **shall not** be used.	
150	Hexadecimal constants **will** be represented using all uppercase letters.	
151	Numeric values in code **will not** be used; symbolic values will be used instead.	Reports direct numeric constants (except integer/float value 1, 0) in expressions, non -const initializations. and switch cases. char constants are allowed. Does not report on templates non-type parameter.
151.1	A string literal shall not be modified.	Report when a char*, char[], or string type is used not as const.A violation is raised if a string literal (for example, " ") is cast as a non const.

Variables

N.	JSF++ Definition	Comments
152	Multiple variable declarations **shall not** be allowed on the same line.	

Unions and Bit Fields

N.	JSF++ Definition	Comments
153	Unions **shall not** be used.	
154	Bit-fields **shall** have explicitly unsigned integral or enumeration types only.	
156	All the members of a structure (or class) **shall** be named and shall only be accessed via their names.	Reports unnamed bit-fields (unnamed fields are not allowed).

Operators

N.	JSF++ Definition	Comments
157	The right hand operand of a && or || operator shall not contain side effects.	Assumes rule 159 is not violated.Messages in log file: • The right hand operand of a && operator shall not contain side effects. • The right hand operand of a || operator shall not contain side effects.
158	The operands of a logical && or || **shall** be parenthesized if the operands contain binary operators.	Messages in log file: • The operands of a logical && shall be parenthesized if the operands contain binary operators. • The operands of a logical || shall be parenthesized if the operands contain binary operators.

N.	JSF++ Definition	Comments
		Exception for: `X || Y || Z , Z&&Y &&Z`
159	Operators ||, &&, and unary & **shall not** be overloaded.	Messages in log file: • `Unary operator & shall not be overloaded.` • `Operator || shall not be overloaded.` • `Operator && shall not be overloaded.`
160	An assignment expression **shall** be used only as the expression in an expression statement.	Only simple assignment, not +=, ++, etc.
162	Signed and unsigned values **shall not** be mixed in arithmetic or comparison operations.	
163	Unsigned arithmetic **shall not** be used.	
164	The right hand operand of a shift operator **shall** lie between zero and one less than the width in bits of the left-hand operand (inclusive).	
164.1	The left-hand operand of a right-shift operator **shall not** have a negative value.	Detects constant case +. Verifier used for dynamic cases.
165	The unary minus operator **shall not** be applied to an unsigned expression.	
166	The sizeof operator **will not** be used on expressions that contain side effects.	
168	The comma operator **shall not** be used.	

Pointers and References

N.	JSF++ Definition	
169	Pointers to pointers should be avoided when possible.	Reports second-level pointers, except for arguments of main.
170	More than 2 levels of pointer indirection **shall not** be used.	Only reports on variables/parameters.
171	Relational operators shall not be applied to pointer types except where both operands are of the same type and point to: • the same object, • the same function, • members of the same object, or • elements of the same array (including one past the end of the same array).	Reports when relational operator are used on pointer types (casts ignored).
173	The address of an object with automatic storage **shall not** be assigned to an object which persists after the object has ceased to exist.	
174	The null pointer **shall not** be de-referenced.	Done with **IDP** checks in Verifier.
175	A pointer **shall not** be compared to NULL or be assigned NULL; use plain 0 instead.	Reports usage of NULL macro in pointer contexts.
176	A typedef **will** be used to simplify program syntax when declaring function pointers.	Reports non-typedef function pointers, or pointers to member functions for types of variables, fields, parameters. Returns type of function, cast, and exception specification.

Type Conversions

N.	JSF++ Definition	Comments
177	User-defined conversion functions **should** be avoided.	Reports user defined conversion function, non-explicit constructor with one parameter or default value for others (even undefined ones). Does not report copy-constructor.
		Additional message for constructor case:
		`This constructor should be flagged as "explicit".`
178	Down casting (casting from base to derived class) **shall** only be allowed through one of the following mechanism: • Virtual functions that act like dynamic casts (most likely useful in relatively simple cases). • Use of the visitor (or similar) pattern (most likely useful in complicated cases).	Reports explicit down casting, dynamic_cast included. (No special case for visitor pattern.)
179	A pointer to a virtual base class **shall not** be converted to a pointer to a derived class.	Reports this specific down cast. Allows dynamic_cast.

180	Implicit conversions that may result in a loss of information **shall not** be used.	Reports the following implicit casts :
		`integer => smaller integer`
		`unsigned => smaller or eq signed`
		`signed => smaller or eq un-signed`
		`integer => float`
		`float => integer`
		Does not report for cast to `bool` reports for implicit cast on constant done with the options `-detect-unsigned-overflows` or `-ignore-constant-overflows`.

N.	JSF++ Definition	Comments
181	Redundant explicit casts **will not** be used.	Reports useless cast: `cast T to T`. Casts to equivalent `typedefs` are also reported.
182	Type casting from any type to or from pointers **shall not** be used.	Does not report when Rule 181 applies.
184	Floating point numbers **shall not** be converted to integers unless such a conversion is a specified algorithmic requirement or is necessary for a hardware interface.	Reports `float->int` conversions. Does not report implicit ones.
185	C++ style casts (`const_cast`, `reinterpret_cast`, and `static_cast`) **shall** be used instead of the traditional C-style casts.	

Flow Control Standards

N.	JSF++ Definition	Comments
186	There **shall** be no unreachable code.	Done with gray checks in the Verifier.
187	All non-null statements **shall** potentially have a side-effect.	
188	Labels **will not** be used, except in switch statements.	
189	The `goto` statement **shall** not be used.	
190	The `continue` statement **shall not** be used.	
191	The `break` statement **shall not** be used (except to terminate the cases of a switch statement).	
192	All `if`, `else if` constructs will contain either a final `else` clause or a comment indicating why a final `else` clause is not necessary.	`else if` should contain an `else` clause.

N.	JSF++ Definition	Comments
193	Every non-empty `case` clause in a switch statement **shall** be terminated with a `break` statement.	
194	All `switch` statements that do not intend to test for every enumeration value **shall** contain a final `default` clause.	Reports only for missing `default`.
195	A `switch` expression **will** not represent a Boolean value.	
196	Every `switch` statement **will** have at least two cases and a potential `default`.	
197	Floating point variables **shall not** be used as loop counters.	Assumes 1 loop parameter.
198	The initialization expression in a `for` loop **will** perform no actions other than to initialize the value of a single `for` loop parameter.	Reports if `loop` parameter cannot be determined. Assumes Rule 200 is not violated. The `loop variable` parameter is assumed to be a variable.
199	The increment expression in a `for` loop **will** perform no action other than to change a single loop parameter to the next value for the loop.	Assumes 1 loop parameter (Rule 198), with non class type. Rule 200 must not be violated for this rule to be reported.
200	Null initialize or increment expressions in `for` loops **will not** be used; a `while` loop will be used instead.	
201	Numeric variables being used within a *for* loop for iteration counting shall not be modified in the body of the loop.	Assumes 1 loop parameter (AV rule 198), and no alias writes.

Expressions

N.	JSF++ Definition	PolySpace Comments
202	Floating point variables **shall not** be tested for exact equality or inequality.	Reports only direct equality/inequality. Check done for all expressions.
203	Evaluation of expressions **shall not** lead to overflow/underflow.	Done with the **SCAL-OVFL** and **FLOAT-OVFL** checks in the Verifier.
204	A single operation with side-effects shall only be used in the following contexts: • by itself • the right-hand side of an assignment • a condition • the only argument expression with a side-effect in a function call • condition of a loop • switch condition • single part of a chained operation	Reports when: • A side effect is found in a return statement • A side effect exists on a single value, and only one operand of the function call has a side effect.
204.1	The value of an expression shall be the same under any order of evaluation that the standard permits.	Reports when: • Variable is written more than once in an expression • Variable is read and write in sub-expressions • Volatile variable is accessed more than once **Note** Read-write operations such as ++, are only considered as a write.
205	The volatile keyword **shall not** be used unless directly interfacing with hardware.	Reports if volatile keyword is used.

Memory Allocation

N.	JSF++ Definition	Comments
206	Allocation/deallocation from/to the free store (heap) **shall not** occur after initialization.	Reports calls to C library functions: `malloc` / `calloc` / `realloc` / `free` and all `new`/`delete` operators in functions or methods.

Fault Handling

N.	JSF++ Definition	Comments
208	C++ exceptions **shall not** be used.	Reports `try`, `catch`, `throw spec`, and `throw`.

Portable Code

N.	JSF++ Definition	Comments
209	The basic types of `int`, `short`, `long`, `float` and `double` **shall not** be used, but specific-length equivalents should be `typedef`'d accordingly for each compiler, and these type names used in the code.	Only allows use of basic types through direct `typedefs`.
213	No dependence shall be placed on C++'s operator precedence rules, below arithmetic operators, in expressions.	Reports when a binary operation has one operand that is not parenthesized and is an operation with inferior precedence level. Reports bitwise and shifts operators that are used without parenthesis and binary operation arguments.

215	Pointer arithmetic **will not** be used.	Reports:
		`p + I`
		`p - I`
		`p++`
		`p--`
		`p+=`
		`p-=` Allows `p[i]`.

JSF++ Rules Not Checked

Code Size and Complexity

N.	JSF++ Definition
2	There shall not be any self-modifying code.

Rules

N.	JSF++ Definition
4	To break a "should" rule, the following approval must be received by the developer: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)
5	To break a "will" or a "shall" rule, the following approvals must be received by the developer: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool) • approval from the software product manager (obtained by the unit approval in the developmental CM tool)
6	Each deviation from a "shall" rule shall be documented in the file that contains the deviation). Deviations from this rule shall not be allowed, AV Rule 5 notwithstanding.
7	Approval will not be required for a deviation from a "shall" or "will" rule that complies with an exception specified by that rule.

Environment

N.	JSF++ Definition
10	Values of character types will be restricted to a defined and documented subset of ISO 10646 1.

Libraries

N.	JSF++ Definition
16	Only DO-178B level A [15] certifiable or SEAL 1 C/C++ libraries shall be used with safety-critical (i.e. SEAL 1) code.

Header Files

N.	JSF++ Definition
34	Header files should contain logically related declarations only.
36	Compilation dependencies should be minimized when possible.
37	Header (include) files should include only those header files that are required for them to successfully compile. Files that are only used by the associated .cpp file should be placed in the .cpp file—not the .h file.
38	Declarations of classes that are only accessed via pointers (*) or references (&) should be supplied by forward headers that contain only forward declarations.

Style

N.	JSF++ Definition
45	All words in an identifier will be separated by the '_' character.
49	All acronyms in an identifier will be composed of uppercase letters.
55	The name of a header file should reflect the logical entity for which it provides declarations.
56	The name of an implementation file should reflect the logical entity for which it provides definitions and have a ".cpp" extension (this name will normally be identical to the header file that provides the corresponding declarations.) At times, more than one .cpp file for a given logical entity will be required. In these cases, a suffix should be appended to reflect a logical differentiation.

Classes

N.	JSF++ Definition
64	A class interface should be complete and minimal.
65	A structure should be used to model an entity that does not require an invariant.
66	A class should be used to model an entity that maintains an invariant.

N.	JSF++ Definition
69	A member function that does not affect the state of an object (its instance variables) will be declared const. Member functions should be const by default. Only when there is a clear, explicit reason should the const modifier on member functions be omitted.
70	A class will have friends only when a function or object requires access to the private elements of the class, but is unable to be a member of the class for logical or efficiency reasons.
70.1	An object shall not be improperly used before its lifetime begins or after its lifetime ends.
71	Calls to an externally visible operation of an object, other than its constructors, shall not be allowed until the object has been fully initialized.
72	The invariant for a class should be: • A part of the postcondition of every class constructor, • A part of the precondition of the class destructor (if any), • A part of the precondition and postcondition of every other publicly accessible operation.
73	Unnecessary default constructors shall not be defined.
77	A copy constructor shall copy all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).
80	The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.
84	Operator overloading will be used sparingly and in a conventional manner.
85	When two operators are opposites (such as == and !=), both will be defined and one will be defined in terms of the other.
86	Concrete types should be used to represent simple independent concepts.
87	Hierarchies should be based on abstract classes.
90	Heavily used interfaces should be minimal, general and abstract.
91	Public inheritance will be used to implement "is-a" relationships.

N.	JSF++ Definition
92	A subtype (publicly derived classes) will conform to the following guidelines with respect to all classes involved in the polymorphic assignment of different subclass instances to the same variable or parameter during the execution of the system: • Preconditions of derived methods must be at least as weak as the preconditions of the methods they override. • Postconditions of derived methods must be at least as strong as the postconditions of the methods they override. In other words, subclass methods must expect less and deliver more than the base class methods they override. This rule implies that subtypes will conform to the Liskov Substitution Principle.
93	"has-a" or "is-implemented-in-terms-of" relationships will be modeled through membership or non-public inheritance.

Namespaces

N.	JSF++ Definition
100	Elements from a namespace should be selected as follows: • using declaration or explicit qualification for few (approximately five) names, • using directive for many names.

Templates

N.	JSF++ Definition
101	Templates shall be reviewed as follows: **1** with respect to the template in isolation considering assumptions or requirements placed on its arguments. **2** with respect to all functions instantiated by actual arguments.
102	Template tests shall be created to cover all actual template instantiations.

N.	JSF++ Definition
103	Constraint checks should be applied to template arguments.
105	A template definition's dependence on its instantiation contexts should be minimized.
106	Specializations for pointer types should be made where appropriate.

Functions

N.	JSF++ Definition
112	Function return values should not obscure resource ownership.
115	If a function returns error information, then that error information will be tested.
117	Arguments should be passed by reference if NULL values are not possible: • **117.1** – An object should be passed as `const T&` if the function should not change the value of the object. • **117.2** – An object should be passed as `T&` if the function may change the value of the object.
118	Arguments should be passed via pointers if NULL values are possible: • **118.1** – An object should be passed as `const T*` if its value should not be modified. • **118.2** – An object should be passed as `T*` if its value may be modified.
120	Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal
122	Trivial accessor and mutator functions should be inlined.
123	The number of accessor and mutator functions should be minimized.
124	Trivial forwarding functions should be inlined.
125	Unnecessary temporary objects should be avoided.

Comments

N.	JSF++ Definition
127	Code that is not used (commented out) shall be deleted.
128	Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented will not be allowed.
129	Comments in header files should describe the externally visible behavior of the functions or classes being documented.
130	The purpose of every line of executable code should be explained by a comment, although one comment may describe more than one line of code.
131	One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).
132	Each variable declaration, typedef, enumeration value, and structure member will be commented.
134	Assumptions (limitations) made by functions should be documented in the function's preamble.

Initialization

N.	JSF++ Definition
143	Variables will not be introduced until they can be initialized with meaningful values. (See also AV Rule 136, AV Rule 142, and AV Rule 73 concerning declaration scope, initialization before use, and default constructors respectively.)

Types

N.	JSF++ Definition
146	Floating point implementations shall comply with a defined floating point standard. The standard that will be used is the ANSI/IEEE Std 754 [1].

Unions and Bit Fields

N.	JSF++ Definition
155	Bit-fields will not be used to pack data into a word for the sole purpose of saving space.

Operators

N.	JSF++ Definition
167	The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.

Type Conversions

N.	JSF++ Definition
183	Every possible measure should be taken to avoid type casting.

Expressions

N.	JSF++ Definition
204	A single operation with side-effects shall only be used in the following contexts: **1** by itself **2** the right-hand side of an assignment **3** a condition **4** the only argument expression with a side-effect in a function call **5** condition of a loop **6** switch condition **7** single part of a chained operation

Memory Allocation

N.	JSF++ Definition
207	Unencapsulated global data will be avoided.

Portable Code

N.	JSF++ Definition
210	Algorithms shall not make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.).
210.1	Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.
211	Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.
212	Underflow or overflow functioning shall not be depended on in any special way.
214	Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.

Efficiency Considerations

N.	JSF++ Definition
216	Programmers should not attempt to prematurely optimize code.

Miscellaneous

N.	JSF++ Definition
217	Compile-time and link-time errors should be preferred over run-time errors.
218	Compiler warning levels will be set in compliance with project policies.

Testing

N.	JSF++ Definition
219	All tests applied to a base class interface shall be applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions /invariants shall be substituted in the derived class tests.
220	Structural coverage algorithms shall be applied against flattened classes.
221	Structural coverage of a class within an inheritance hierarchy containing virtual functions shall include testing every possible resolution for each set of identical polymorphic references.

Using PolySpace Software in Visual Studio

Verifying Code in Visual Studio

In this section...
"Creating a Visual Studio Project" on page 13-4
"Setting Up and Starting a PolySpace Verification in Visual Studio" on page 13-5
"Monitoring a Verification" on page 13-13
"Reviewing Verification Results in Visual Studio" on page 13-15
"Using the PolySpace Spooler" on page 13-15

You can apply the powerful code verification functionality of PolySpace software to code that you develop within the Visual Studio® Integrated Development Environment (IDE).

A typical workflow is:

1 Use the Visual Studio editor to create a project and develop code within this project.

2 Set up the PolySpace verification by configuring analysis options and settings, and then start the verification.

3 Monitor the verification.

4 Review the verification results.

Before you can verify code in Visual Studio, you must install the PolySpace plug-in for Visual.NET. For more information , see "PolySpace Plug-In Requirements" and "Installing the PolySpace C++ Add-In for Visual Studio" in the *PolySpace Installation Guide*.

Once you have installed the plug-in, in the Visual Studio editor, you can access:

• A **PolySpace** menu

• A **PolySpace Log** view
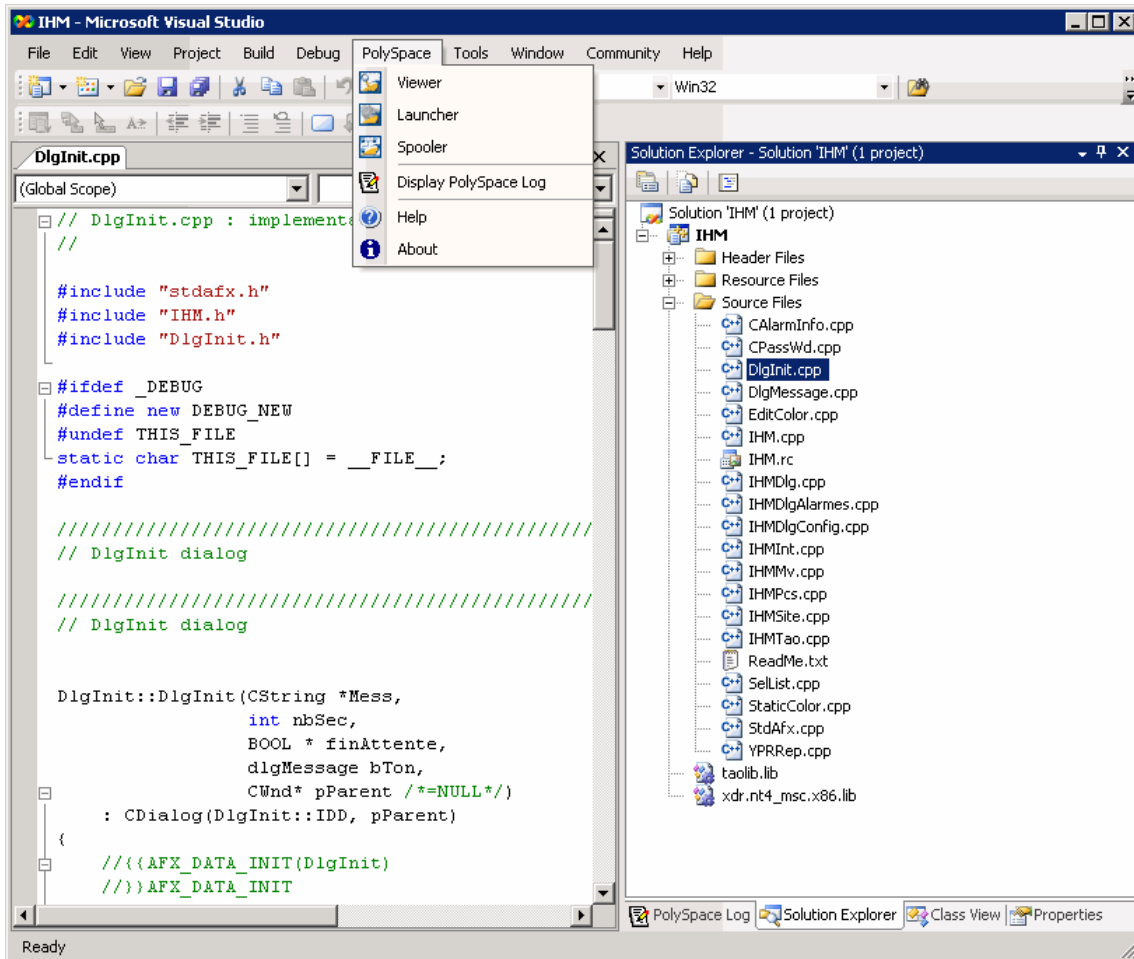
Creating a Visual Studio Project

If your source files do not belong to a Visual Studio project, you can create one using the Visual Studio editor:

1 Select **File > New > Project > New > Project Console Win32** to create a project space

2 Enter a project name, for example, CppExample.

3 Save this project in an appropriate location, for example, C:\PolySpace\Visual. The software creates some files and a Project Console Win32.

To add files to your project:

1 Select the **Browse the solution** tab.

2 Right-click the project name. From the pop-up menu, select **Add > Add existing element** .

3 Add the files you want (for example, matrix.cpp and matrix.h in *<PolySpaceProduct>*/Examples/Demo_Cpp_Long/sources) to the project (for example, CppExample).

Setting Up and Starting a PolySpace Verification in Visual Studio

To set up and start a verification:

1 In the Visual Studio **Solution Explorer** view, select one or more files that you want to verify.

2 Right-click the selection, and select **PolySpace Verification**.
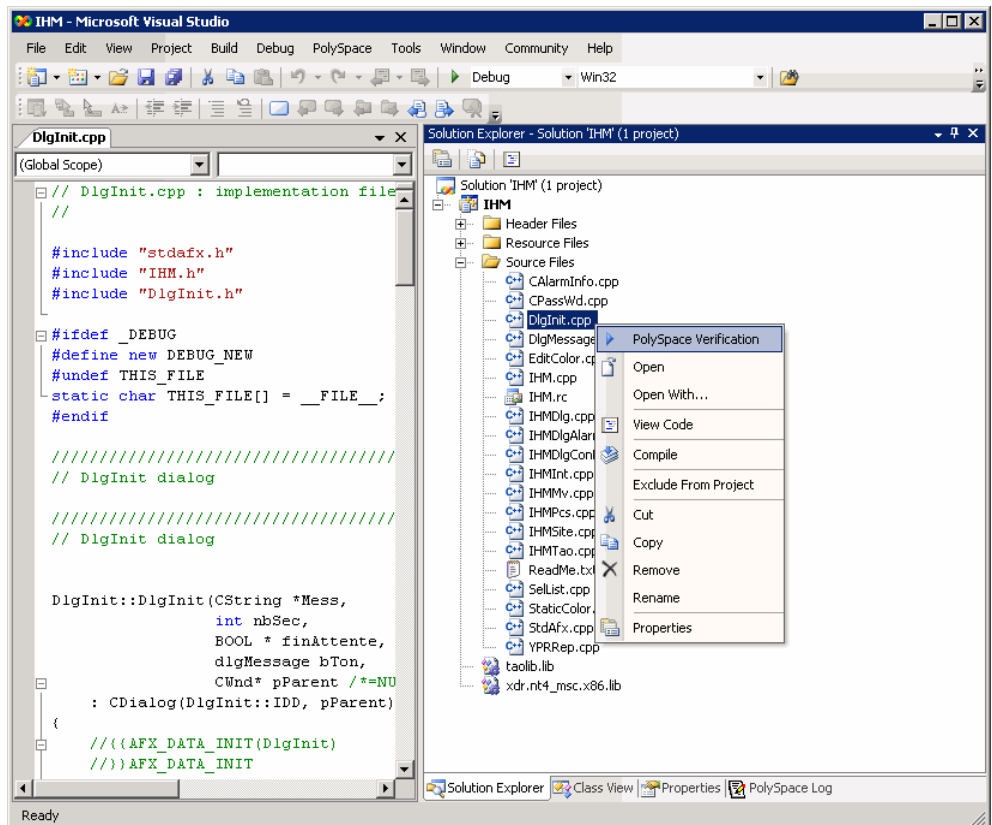
The Easy Settings dialog box opens.
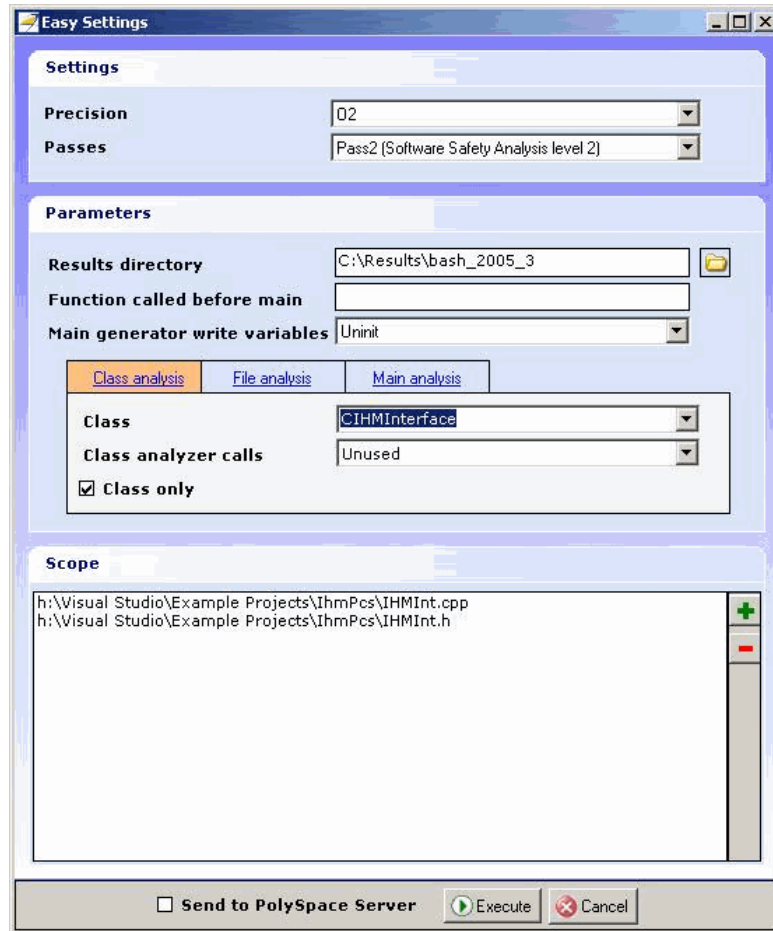
3 In the Easy Settings dialog box, you can specify the following options for your verification:

- Under **Settings**, in the **Precision** and **Passes** fields respectively, you specify precision (-O) and the level of verification (-to).

- Under **Parameters**, you can configure the following:

  - **Results folder** – You store verification results here (-results-dir).

  - **Function called before main** – A function, if any, called before all functions (-function-call-before-main)

- **Main generator write variables** – The type of initialization for global variables (`-main-generator-writes-variables`).

- **Class analysis** tab – By default, enables the class analysis with default options: the class to analyze (`-class-analyzer`) and associated options which can change the behavior of the analysis (`-class-only` and `-class-analyzer`).

- **File analysis** tab – Where you choose a file analysis with associated option (`-main-generator-calls`).

- **Main analysis** tab – Where you choose a partial integration analysis by choosing the name of the "main" (`-main`).

- Under **Scope**, you can modify the list of files and classes to verify.

For information on *how* to choose your options, see "Options Description" in the ***PolySpace Products for C++ Reference Guide***.

Note In the PolySpace Launcher window, you configure options that you cannot set in the Easy Settings dialog box. See "Setting Standard PolySpace Options" on page 13-11.

---

4 Click **Execute** to start the verification.

Verifying Classes

In the Easy Settings dialog box, you can verify a C++ class by modifying the scope option.

To verify a class:

1 In the Visual Studio Solution Explorer, right-click a file and select **PolySpace Verification**.
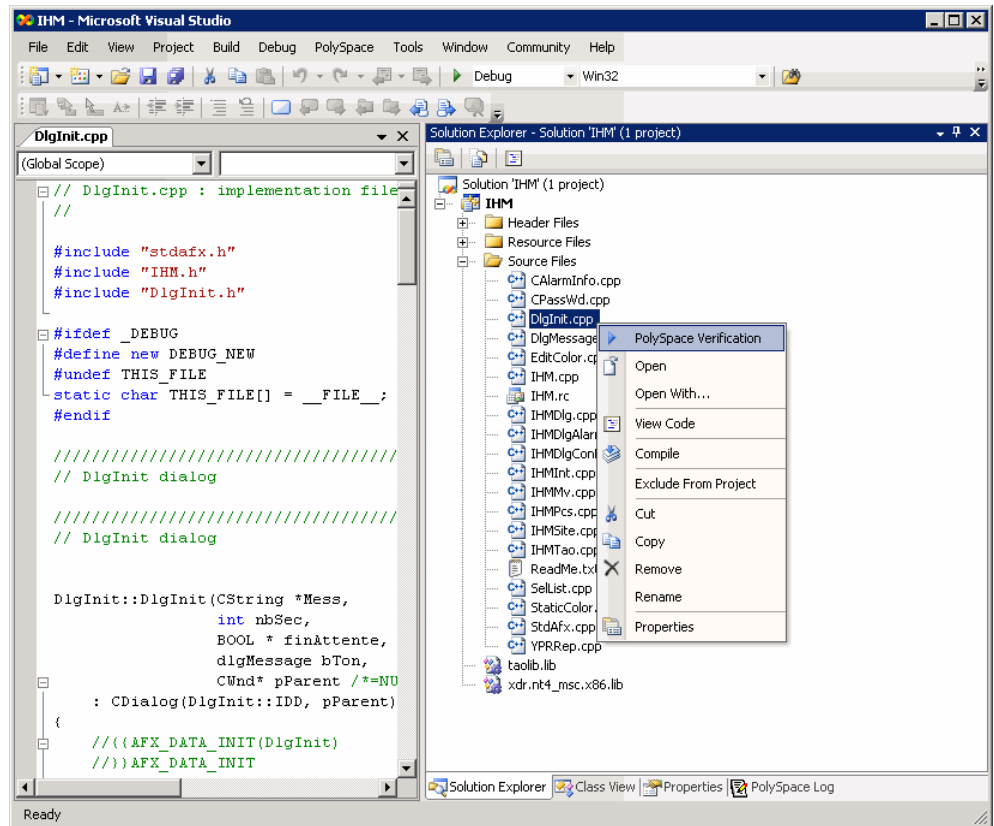
The Easy Settings dialog box opens.
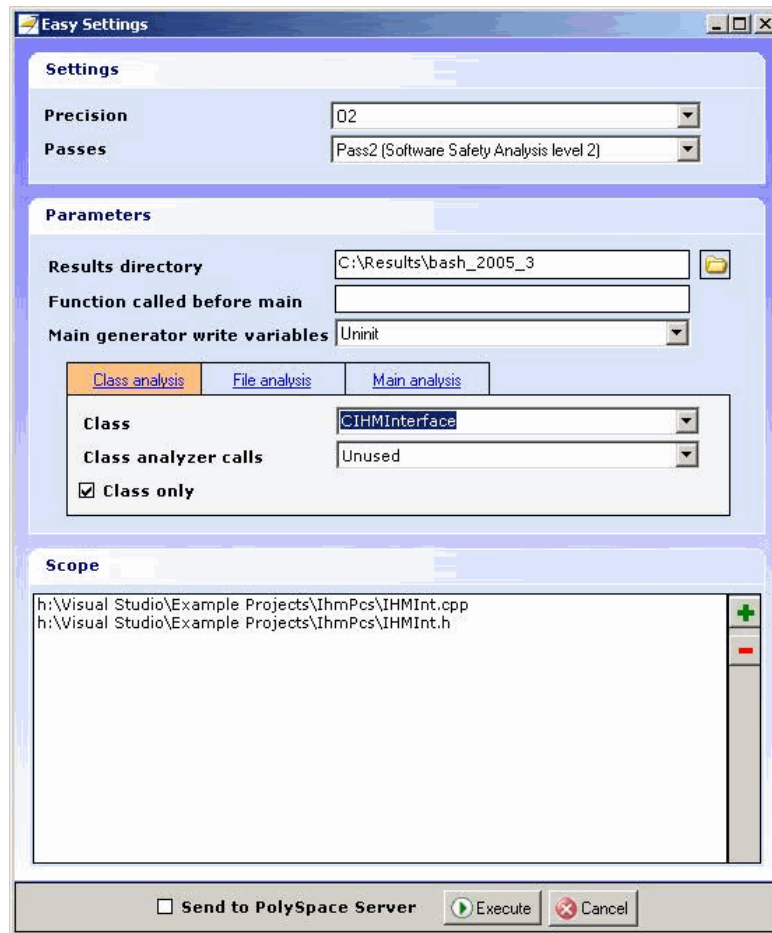
2 In the Scope window, click .

The Select Files and Classes dialog box opens.
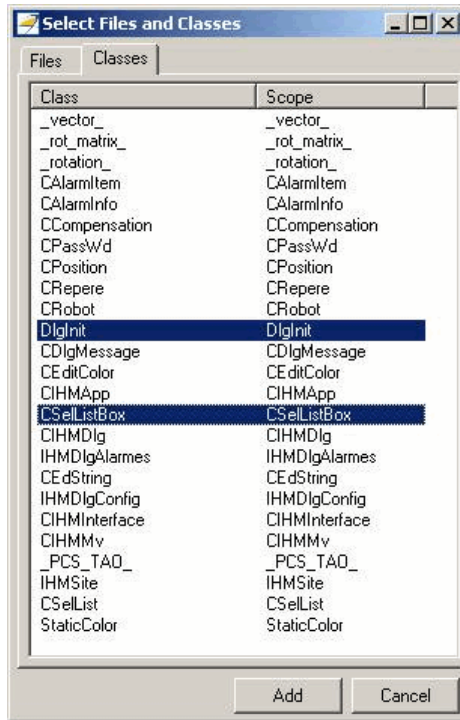
3 Select the classes that you want to analyze, then click **Add**.

4 In the Easy Settings dialog box, click **Execute** to start the verification.

Verifying an Entire Project

You can verify an entire project only through the PolySpace Launcher (select **PolySpace > Launcher**).

For information on using the PolySpace Launcher , see Chapter 7, "Running a Verification" in the *PolySpace Products for C++ User Guide*.

Setting Standard PolySpace Options

In the PolySpace Launcher window, you specify PolySpace verification options that you cannot set in the Easy Settings dialog box.

To open the PolySpace Launcher window, select **PolySpace > Launcher**. The software opens the PolySpace Launcher window using the**last** configuration (`.cfg`) file updated in Visual Studio. The software does not check the consistency of this configuration file with the current project, so it always displays a warning message. This message indicates that the `.cfg` file used by the PolySpace Launcher does not correspond to the `.cfg` file of the current project.

For information on *how* to choose your options, see "Options Description" in the ***PolySpace Products for C++ Reference Guide***.

The Configuration File and Default Options

Some options are set by default while others are extracted from the Visual Studio project and stored in the associated PolySpace configuration file.

- The following table shows Visual Studio options that are extracted automatically, and their corresponding PolySpace options:

Visual Studio Option	PolySpace Option
/D <name>	-D *<name>*
/U <name>	-U *<name>*
/MT	-D_MT
/MTd	-D_MT -D_DEBUG
/MD	-D_MT -D_DLL
/MDd	-D_MT -D_DLL -D_DEBUG
/MLd	-D_DEBUG
/Zc:wchar_t	-wchar-t-is keyword
/Zc:forScope	-for-loop-index-scope in
/FX	-support-FX-option-results
/Zp[1,2,4,8,16]	-pack-alignment-value [1,2,4,8,16]

- Source and include folders (-I) are also extracted automatically from the Visual Studio project.

- Default options passed to the kernel depend on the Visual Studio release:
  -dialect Visual7.1 (or -dialect visual8) -OS-target Visual -target i386 -desktop

Monitoring a Verification

Once you launch a verification, you can follow its progress in the **PolySpace Log** view.

Compilation errors are highlighted as links. Click a link to display the file and line that produced the error.
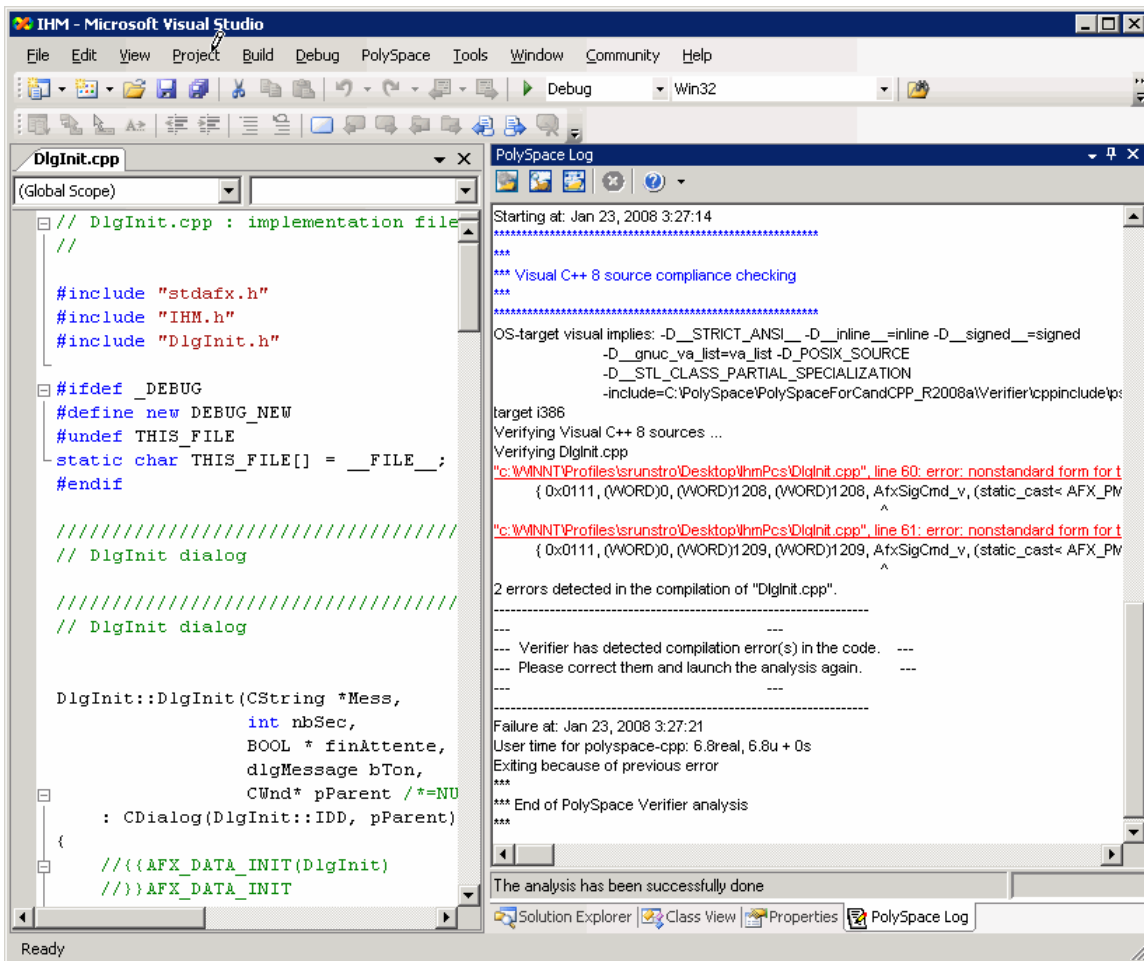
If the verification is being carried out on a server, use the PolySpace Spooler to follow the verification progress. Select **PolySpace > Spooler**, which opens the PolySpace Queue Manager Interface dialog box.

To stop a verification, on the **PolySpace Log** toolbar, click **X** . For a server verification, this option works only during the compilation phase, before the verification is sent to the server. After the compilation phase, you can select **PolySpace > Spooler** and in the PolySpace Queue Manager Interface dialog box, stop the verification.

For more information on the PolySpace Spooler, see "Managing Verification Jobs Using the PolySpace Queue Manager" on page 7-7 in the *PolySpace Products for C++ User Guide*.

Reviewing Verification Results in Visual Studio

Select **PolySpace > Viewer** to open the Viewer with the last available results. If verification has been carried out on a server, download the results before opening the Viewer.

For information on reviewing and understanding PolySpace verification results, see Chapter 9, "Reviewing Verification Results" in the ***PolySpace Products for C++ User Guide***.

Using the PolySpace Spooler

You can use the PolySpace spooler to manage jobs that are run on remote servers. To open the spooler, select **PolySpace > Spooler** .

For more information, see "Managing Verification Jobs Using the PolySpace Queue Manager" on page 7-7 in the ***PolySpace Products for C++ User Guide***.

Using PolySpace Software in the Eclipse IDE

Verifying Code in the Eclipse IDE

In this section...

You can apply the powerful code verification of PolySpace software to code that you develop within the Eclipse Integrated Development Environment (IDE).

A typical workflow is:

1 Use the Eclipse™ editor to create an Eclipse project and develop code within your project.

2 Set up the PolySpace verification by configuring analysis options and settings.

3 Start the verification and monitor the process.

4 Review the verification results.

Install the PolySpace plug-in for Eclipse IDE before you verify code in Eclipse IDE. For more information, see "PolySpace Plug-In Requirements" and "Installing the PolySpace Plug-In for Eclipse IDE" in the *PolySpace Installation Guide*.

Once you have installed the plug-in, in the Eclipse editor, you have access to:

• A **PolySpace** menu

• Toolbar buttons you use to launch a verification and open the PolySpace spooler

• **PolySpace Log** and **PolySpace Setting** views
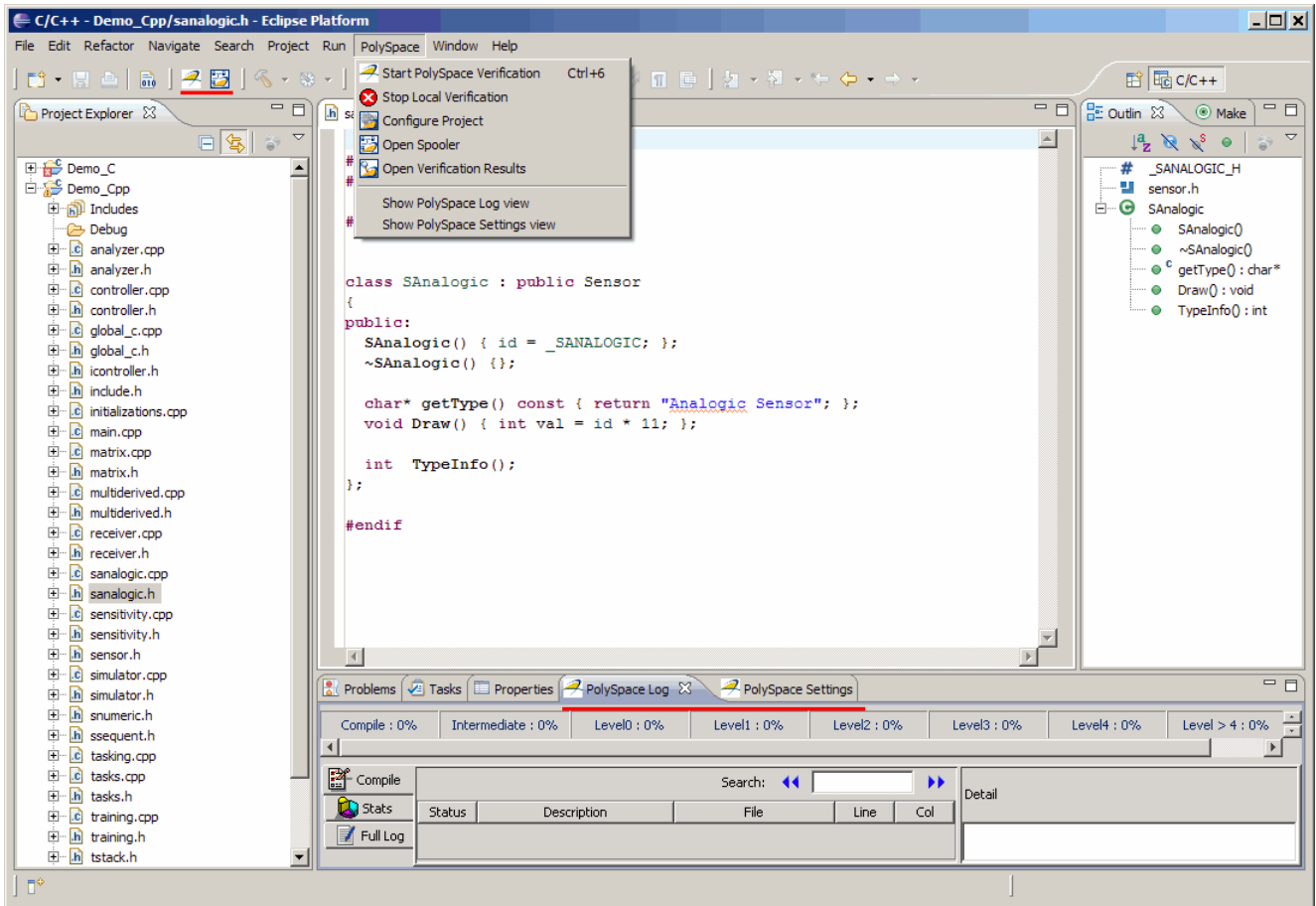
Creating an Eclipse Project

If your source files do not belong to an Eclipse project, create one using the Eclipse editor:

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

2 Clear the **Use default location** check box.

3 Click **Browse** to navigate to the folder containing your source files, for example, `C:\Test\Source_cpp`.

4 In the **Project name** field, enter a name, for example, `Demo_Cpp`.

5 In the **Project Type** tree, under **Executable**, select **Empty Project**.

6 Under **Toolchains**, select your installed toolchain, for example, `MinGW GCC`.

7 Click **Finish**. An Eclipse project is created.

For information on developing code within Eclipse IDE, refer to www.eclipse.org.

Setting Up PolySpace Verification with Eclipse Editor

Analysis Options

To specify analysis options for your verification :

1 In **Project Explorer**, select the project or files that you want to verify.

2 Select **PolySpace > Configure Project** to open the PolySpace Launcher for CPP window.

3 Under **Analysis options**, select your options for the verification process.

4 Save your options.

For information on *how* to choose your options, see "Options Description" in the ***PolySpace Products for C++ Reference Guide***

> **Note** The software automatically adds your Eclipse compiler options for include paths (-I) and symbol definitions (-D) to the list of analysis options.
>
> To view the -I and -D options in the Eclipse editor :
>
> **1** Select **Project > Properties** to open the Properties for Project dialog box.
>
> **2** In the tree, under **C/C++ General** , select **Paths and Symbols** .
>
> **3** Select **Includes** to view the -I options or **Symbols** to view the -D options.

Other Settings

In the **PolySpace Settings** view, specify:

- In the **Results folder** field, the location of your results folder.
- The required **Verification level**, for example, Level4.

If the item that you select in the **Project Explorer** is not a class, then you can also do the following in the **PolySpace Settings** view:

- Generate a main (if the item that you select does not contain one) by selecting the **Generate a main** check box. If you want to change the default behavior of the main generator, specify advanced settings through the -main-generator-calls option in the PolySpace Launcher for CPP window. Select **PolySpace > Configure Project** to open this window.
- Specify the -function-called-before-main option. In the **Startup function to call** field, enter the name of the function that you want to call before all selected functions in main.

Setting Up Verification for a Single Class

You can use the **PolySpace Settings** view to configure verification of a single class:

1 In **Project Explorer**, select your class.

2 In the **PolySpace Settings** view, select the **Verify the class contents only** check box.

This approach is equivalent to specifying the -class-analyzer and -class-only options. If necessary, you can use the PolySpace Launcher for CPP window (**PolySpace > Configure Project**) to specify other options, for example, -class-analyzer-calls.

Launching Verification from Eclipse Editor

To launch a PolySpace verification from the Eclipse editor:

1 Select the file, files, or class that you want to verify.

2 Either right-click and select **Start PolySpace Verification**, or select **PolySpace > Start PolySpace Verification**.

You can see the progress of the verification in the **PolySpace Log** view. If you see an error or warning, double-click it to go to the corresponding location in the source code.

To stop verification, select **PolySpace > Stop Local Verification**.

For more information on monitoring the progress of a verification, see Chapter 7, "Running a Verification" in the ***PolySpace Products for C++ User Guide***.

Reviewing Verification Results from Eclipse Editor

Use the PolySpace Viewer to examine results of the verification:

1 Select **PolySpace > Open Verification Results** to open the PolySpace Viewer.

2 If results are available in the specified **Results folder**, then these results appear automatically in the Viewer window.

For information on reviewing and understanding PolySpace verification results, see Chapter 9, "Reviewing Verification Results" in the ***PolySpace Products for C++ User Guide***.

Using the PolySpace Spooler

Use the PolySpace spooler to manage jobs running on remote servers. To open the spooler, select **PolySpace > Open Spooler** .

For more information, see "Managing Verification Jobs Using the PolySpace Queue Manager" on page 7-7 in the ***PolySpace Products for C++ User Guide***.

Glossary

Atomic

In computer programming, atomic describes a unitary action or object that is essentially indivisible, unchangeable, whole, and irreducible.

Atomicity

In a transaction involving two or more discrete pieces of information, either all of the pieces are committed or none are.

Batch mode

Execution of PolySpace from the command line, rather than via the launcher Graphical User Interface.

Category

One of four types of orange check: *potential bug, inconclusive check, data set issue* and *basic imprecision*

Certain error

See red error

Check

Test performed by PolySpace during verification, colored red, orange, green or gray in the viewer

Dead Code

Code which is inaccessible at execution time under all circumstances, due to the logic of the software executed before it.

Development Process

Development process used within a company to progress through the software development lifecycle.

Green check

Check found to be confirmed as error free.

Gray code

Dead code.

Imprecision

Approximations made during PolySpace verification, so that data values possible at execution time are represented by supersets including those values

mcpu

Micro Controller/Processor Unit

Orange warning

Check found to represent a possible error, which may be revealed on 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 to indicate what proportion of the verification has been completed. Could be considered as a "textual progress bar".

Red error

Check found to represent a definite error

Review

Inspection of the results produced by a PolySpace verification, using the Viewer.

Scaling option

Option applied when an application submitted to PolySpace Server proves to be bigger or more complex than is practical.

Selectivity

The ratio of (green + gray + red) / (total amount of checks)

Unreachable code

Dead code

Verification

In order to use a PolySpace tool, the code is prepared and a verification is launched which in turn produces results for review.

Index