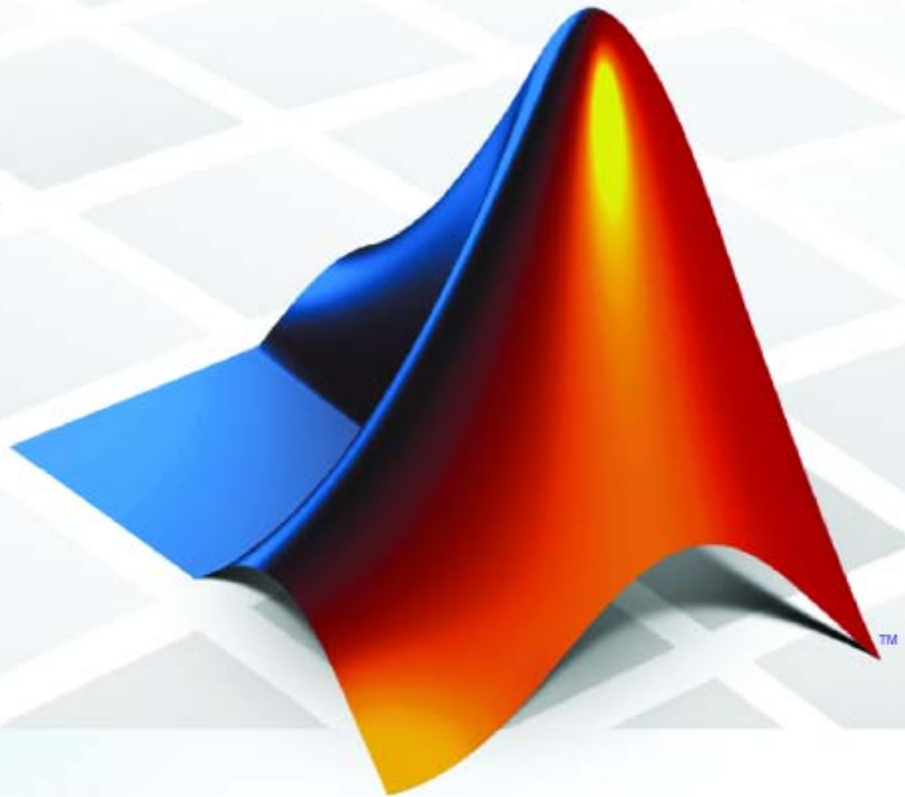


# PolySpace® Products for C++ 7

## User's Guide



## How to Contact The MathWorks



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



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



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

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

© COPYRIGHT 1999–2009 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

### Trademarks

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

### Patents

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

### Revision History

March 2008	Online Only	Revised for Version 5.1 (Release 2008a)
October 2008	Online Only	Revised for Version 6.0 (Release 2008b)
March 2009	Online Only	Revised for Version 7.0 (Release 2009a)

## Introduction to PolySpace Products

### 1

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

## Choosing How to Use PolySpace Software

### 2

<b>How to Use This Chapter</b> .....	2-2
<b>Applying PolySpace Verification to Your Development Process</b> .....	2-4
Overview of the PolySpace Approach .....	2-4
Standard Development Process .....	2-9
Rigorous Development Process: Introducing Tools and Coding Rules .....	2-12
A Quality/Qualification Approach .....	2-15
Code Acceptance Criterion .....	2-16
Choosing the Type of Verification You Want to Perform ..	2-17

### 3

<b>Analyzing C++ Classes</b> .....	3-2
Overview .....	3-2
Why Provide a Class Analyzer .....	3-2
<b>How the Class Analyzer Works</b> .....	3-3
Overview .....	3-3
Sources to be Verified .....	3-4
Architecture of the Generated main .....	3-4
Log File .....	3-5
Characteristics of a Class and Messages in the Log File ..	3-6
Behavior of Global variables and members .....	3-6
Methods and Class Specificities .....	3-10
<b>Types of Classes</b> .....	3-13
Simple Class .....	3-13
Simple Inheritance .....	3-15
Multiple Inheritance .....	3-16
Abstract Classes .....	3-17
Virtual Inheritance .....	3-18
Other Types of Classes .....	3-19

## Setting Up a Verification Project

### 4

<b>Creating a Project</b> .....	4-2
What Is a Project? .....	4-2
Project Directories .....	4-3
Opening PolySpace Launcher .....	4-3
Specifying Default Directory .....	4-6
Creating New Projects .....	4-8
Opening Existing Projects .....	4-9
Specifying Source Files .....	4-10
Specifying Include Directories .....	4-12
Specifying Results Directory .....	4-14
Specifying Analysis Options .....	4-15
Configuring Text and XML Editors .....	4-16

Saving the Project .....	4-17
<b>Setting Up Project to Check Coding Rules .....</b>	<b>4-19</b>
PolySpace JSF C++ Checker Overview .....	4-19
Checking Compliance with JSF++ Coding Rules .....	4-19
Creating a JSF++ Rules File .....	4-20
Excluding Files from JSF++ Checking .....	4-22
<b>Setting Up Project for Generic Target Processors .....</b>	<b>4-24</b>
Project Model Files .....	4-24
Creating Project Model Files .....	4-25
Viewing Existing Generic Targets .....	4-25
Defining Generic Targets .....	4-26
Deleting a Generic Target .....	4-28
Common Generic Targets .....	4-28
Creating a Configuration File from a PolySpace Project Model File .....	4-30

## Emulating Your Runtime Environment

# 5

<b>Setting Up a Target .....</b>	<b>5-2</b>
Target/Compiler Overview .....	5-2
Specifying Target/Compilation Parameters .....	5-2
Predefined Target Processor Specifications (size of char, int, float, double...) .....	5-3
Generic Target Processors .....	5-5
Compiling Operating System Dependent Code (OS-target issues) .....	5-5
Ignoring or Replacing Keywords Before Compilation .....	5-9
How to Gather Compilation Options Efficiently .....	5-12
<b>Applying Data Ranges to External Variables and Stub     Functions (DRS) .....</b>	<b>5-14</b>
Overview of Data Range Specifications (DRS) .....	5-14
Specifying Data Ranges .....	5-14
File Format .....	5-15
Variable Scope .....	5-17
Performing Efficient Module Testing with DRS .....	5-19

## Preparing Source Code for Verification

# 6

<b>Stubbing</b> .....	6-2
Stubbing Overview .....	6-2
Manual vs. Automatic Stubbing .....	6-2
Deciding which Stub Functions to Provide .....	6-3
Stubbing Examples .....	6-6
Specifying Call Sequence .....	6-8
Constraining Data with Stubbing .....	6-9
Recoding Specific Functions .....	6-12
<b>Preparing Code for Variables</b> .....	6-14
How are Variables Initialized .....	6-14
Data and Coding Rules .....	6-15
Variables: Declaration and Definition .....	6-15
How Can I Model Variable Values External to My Application? .....	6-16
<b>Preparing Code for Built-in Functions</b> .....	6-18
Overview .....	6-18
Stubs of stl Functions .....	6-18
Stubs of libc Functions .....	6-18
<b>Types Promotion</b> .....	6-20
Unsigned Types Promoted to Signed .....	6-20
Promotion Rules in Operators .....	6-21

## Running a Verification

# 7

<b>Types of Verification</b> .....	7-2
------------------------------------	-----

<b>Running Verifications on PolySpace Server</b> .....	<b>7-3</b>
Starting Server Verification .....	7-3
What Happens When You Run Verification .....	7-4
Managing Verification Jobs Using the PolySpace Queue Manager .....	7-5
Monitoring Progress of Server Verification .....	7-6
Viewing Verification Log File on Server .....	7-9
Stopping Server Verification Before It Completes .....	7-11
Removing Verification Jobs from Server Before They Run .....	7-12
Changing Order of Verification Jobs in Server Queue .....	7-13
Purging Server Queue .....	7-13
Changing Queue Manager Password .....	7-15
Sharing Server Verifications Between Users .....	7-15
<b>Running Verifications on PolySpace Client</b> .....	<b>7-19</b>
Starting Verification on Client .....	7-19
What Happens When You Run Verification .....	7-20
Monitoring the Progress of the Verification .....	7-21
Stopping Client Verification Before It Completes .....	7-22
<b>Running Verifications from Command Line</b> .....	<b>7-24</b>
Launching Verifications in Batch .....	7-24
Managing Verifications in Batch .....	7-24

## Troubleshooting Verification Problems

# 8

<b>Verification Process Failed Errors</b> .....	<b>8-2</b>
Overview .....	8-2
Hardware Does Not Meet Requirements .....	8-2
You Did Not Specify the Location of Included Files .....	8-2
PolySpace Software Cannot Find the Server .....	8-3
Limit on Assignments and Function Calls .....	8-4
<b>Compile Errors</b> .....	<b>8-6</b>
Overview .....	8-6
Examining the Compile Log .....	8-6
Includes .....	8-8

Specific Keyword or Extended Keyword .....	8-8
Initialization of Global Variables .....	8-10
<b>Dialect Issues</b> .....	8-12
ISO versus Default Dialects .....	8-12
CFront2 and CFront3 Dialects .....	8-14
Visual Dialects .....	8-15
GNU Dialect .....	8-17
<b>Link Messages</b> .....	8-20
STL Library C++ Stubbing Errors .....	8-20
Lib C Stubbing Errors .....	8-21
<b>Troubleshooting Using the Preprocessed .ci Files</b> ....	8-25
Overview .....	8-25
Example of <i>ci</i> File .....	8-25
Troubleshooting Methodology .....	8-27
<b>Reducing Verification Time</b> .....	8-30
How Far has the Verification Progressed? How Can I Predict the Duration? .....	8-30
An Ideal Application Size .....	8-32
Why Should there be an Optimum Size? .....	8-33
Switch the Antivirus Off .....	8-34
Tuning PolySpace Parameters .....	8-34
Selecting a Subset of Code .....	8-35
A Decision Algorithm to Speed-Up a Verification: Hints and Troubleshooting .....	8-40
What are the Benefits of these Methods? .....	8-45

## Reviewing Verification Results

# 9

<b>Before You Review PolySpace Results</b> .....	9-2
Overview: Understanding PolySpace Results .....	9-2
Why Gray Follows Red and Green Follows Orange .....	9-3
What is the Message and What does it Mean? .....	9-4
What is the C++ Explanation? .....	9-5



<b>Opening Verification Results</b> .....	<b>9-8</b>
Downloading Results from Server to Client .....	9-8
Opening Verification Results .....	9-11
Exploring the Viewer Window .....	9-11
Selecting Viewer Mode .....	9-15
Setting Character Encoding Preferences .....	9-15
<b>Reviewing Results in Assistant Mode</b> .....	<b>9-19</b>
What Is Assistant Mode? .....	9-19
Switching to Assistant Mode .....	9-19
Selecting the Methodology and Criterion Level .....	9-20
Exploring Methodology for C++ .....	9-21
Defining a Custom Methodology .....	9-23
Reviewing Checks .....	9-24
<b>Reviewing Results in Expert Mode</b> .....	<b>9-26</b>
What Is Expert Mode? .....	9-26
Switching to Expert Mode .....	9-26
Selecting a Check to Review .....	9-26
Displaying the Calling Sequence .....	9-28
Tracking Review Progress .....	9-29
Making the Reviewed Column Visible .....	9-30
Filtering Checks .....	9-33
Types of Filters .....	9-33
Creating a Custom Filter .....	9-35
<b>Generating Reports of Verification Results</b> .....	<b>9-37</b>
<b>Using PolySpace Results</b> .....	<b>9-41</b>
Review Runtime Errors: Fix Red Errors .....	9-41
Red Checks Where Gray Checks were Expected .....	9-42
Potential Side Effect of a Red Error .....	9-44
Review Dead Code Checks: Why Gray Code is Interesting .....	9-45
Selective Orange Review: Finding the Maximum Number of Bugs in One Hour .....	9-47
Exhaustive Orange Review at Unit Phase .....	9-49
Exhaustive Orange Review at Integration Phase .....	9-50
Integration Bug Tracking .....	9-52

<b>Understanding Orange Checks</b> .....	<b>10-2</b>
What is an Orange Check? .....	10-2
Sources of Orange Checks .....	10-3
Determining Cause of Orange Checks .....	10-5
<b>Reducing Orange Checks in Your Results</b> .....	<b>10-6</b>
Options to Reduce Orange Checks .....	10-6
Generic Objectives: A Balance Between Precision and Verification Time .....	10-7
Applying Coding Rules to Reduce Orange Checks .....	10-8
Varying the Precision Level .....	10-8
Applying Software Safety Level Wisely .....	10-9
Adding Precision Constraints at the Periphery Via Stubs .....	10-11
Tuning Advanced Parameters .....	10-13
Applying Data Ranges .....	10-13
Considering the Effects of Application Code Size .....	10-14
<b>Reviewing Orange Checks</b> .....	<b>10-15</b>
Selective Orange Review .....	10-15
Performing a Selective Orange Review .....	10-16
Exhaustive Orange Review .....	10-17
Performing an Exhaustive Orange Review .....	10-18

## Day to Day Use

<b>PolySpace In One Click Overview</b> .....	<b>11-2</b>
<b>Using PolySpace In One Click</b> .....	<b>11-3</b>
PolySpace In One Click Workflow .....	11-3
Setting the Active Project .....	11-3
Launching Verification .....	11-5
Using the Taskbar Icon .....	11-9

<b>PolySpace JSF C++ Checker Overview</b> .....	12-2
<b>Using the PolySpace JSF C++ Checker</b> .....	12-3
Setting Up JSF++ Checking .....	12-3
Running a Verification with JSF++ Checking .....	12-7
<b>Supported Rules</b> .....	12-11
Code Size and Complexity .....	12-12
Environment .....	12-12
Libraries .....	12-13
Pre-Processing Directives .....	12-14
Header Files .....	12-15
Style .....	12-15
Classes .....	12-19
Namespaces .....	12-23
Templates .....	12-23
Functions .....	12-23
Comments .....	12-25
Declarations and Definitions .....	12-25
Initialization .....	12-26
Types .....	12-27
Constants .....	12-27
Variables .....	12-27
Unions and Bit Fields .....	12-28
Operators .....	12-28
Pointers and References .....	12-30
Type Conversions .....	12-31
Flow Control Standards .....	12-32
Expressions .....	12-33
Memory Allocation .....	12-35
Fault Handling .....	12-35
Portable Code .....	12-35
<b>Rules Not Checked</b> .....	12-36
Code Size and Complexity .....	12-37
Rules .....	12-37
Environment .....	12-37
Libraries .....	12-38
Header Files .....	12-38

Style .....	12-38
Classes .....	12-39
Namespaces .....	12-40
Templates .....	12-41
Functions .....	12-41
Comments .....	12-42
Initialization .....	12-42
Types .....	12-43
Unions and Bit Fields .....	12-43
Operators .....	12-43
Type Conversions .....	12-43
Expressions .....	12-43
Memory Allocation .....	12-44
Portable Code .....	12-44
Efficiency Considerations .....	12-44
Miscellaneous .....	12-45
Testing .....	12-45

## PolySpace C++ Add-in for Visual Studio

### 13

<b>Overview</b> .....	13-2
<b>Using PolySpace Software Within Visual Studio</b> .....	13-3
Overview .....	13-3
PolySpace Parameters Within Visual Studio .....	13-3
Performing a File Verification from Visual Studio .....	13-7
Verifying Classes .....	13-11
The Configuration File and Default Options .....	13-14
<b>Verifying an Entire Project</b> .....	13-16

## Code Verification for Eclipse IDE

### 14

<b>Overview</b> .....	14-2
-----------------------	------

**Using PolySpace Software Within Eclipse IDE** ..... 14-3  
PolySpace Features in the Eclipse Editor ..... 14-3  
Verifying Files from Eclipse IDE ..... 14-5

**Glossary**



**Index**





# Introduction to PolySpace Products

---

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

## Introduction to PolySpace Products

In this section...
“The Value of PolySpace Verification” on page 1-2
“How PolySpace Verification Works” on page 1-4
“Product Components” on page 1-6
“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 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](http://www.mathworks.com/access/helpdesk/help/toolbox/polyspace/polyspace.html)

## **The MathWorks Online**

For additional information and support, see:

[www.mathworks.com/products/polyspace](http://www.mathworks.com/products/polyspace)





# Choosing How to Use PolySpace Software

---

- “How to Use This Chapter” on page 2-2
- “Applying PolySpace Verification to Your Development Process” on page 2-4

## How to Use This Chapter

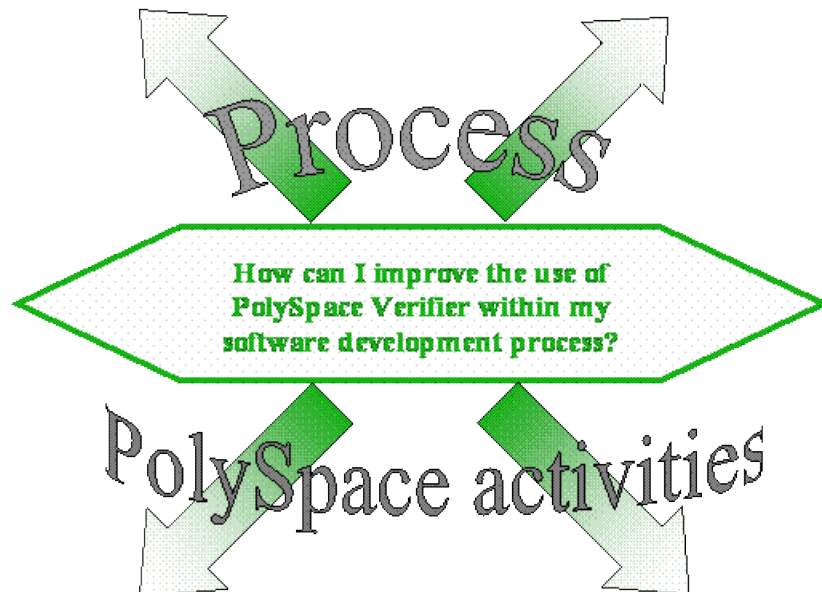
This chapter is designed for **Project managers, quality managers, and developers** who want to integrate PolySpace verification into their project development cycle. It explains how to apply PolySpace verification to each phase of the typical project lifecycle.

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

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

**How can I use PolySpace in my current process?**

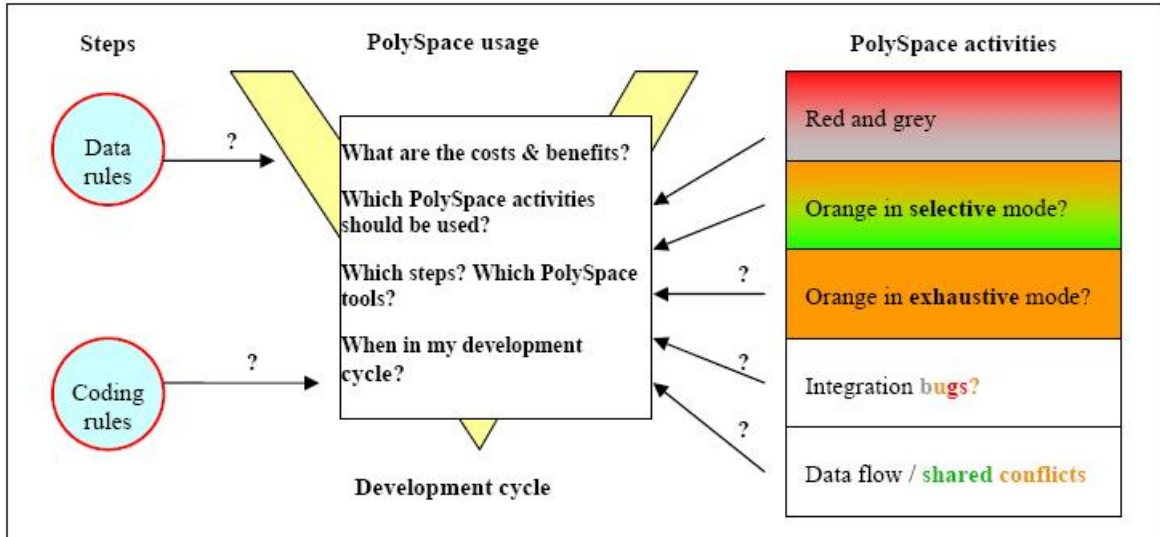
**How can I change the process to get more out of PolySpace?**



**On given results, how can I find the maximum number of anomalies?**

**How can I get the best results?**

This guide suggests answers to the following questions:



## Applying PolySpace Verification to Your Development Process

In this section...
“Overview of the PolySpace Approach” on page 2-4
“Standard Development Process” on page 2-9
“Rigorous Development Process: Introducing Tools and Coding Rules” on page 2-12
“A Quality/Qualification Approach” on page 2-15
“Code Acceptance Criterion” on page 2-16
“Choosing the Type of Verification You Want to Perform” on page 2-17

### Overview of the PolySpace Approach

PolySpace verification supports two objectives at the same time:

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

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

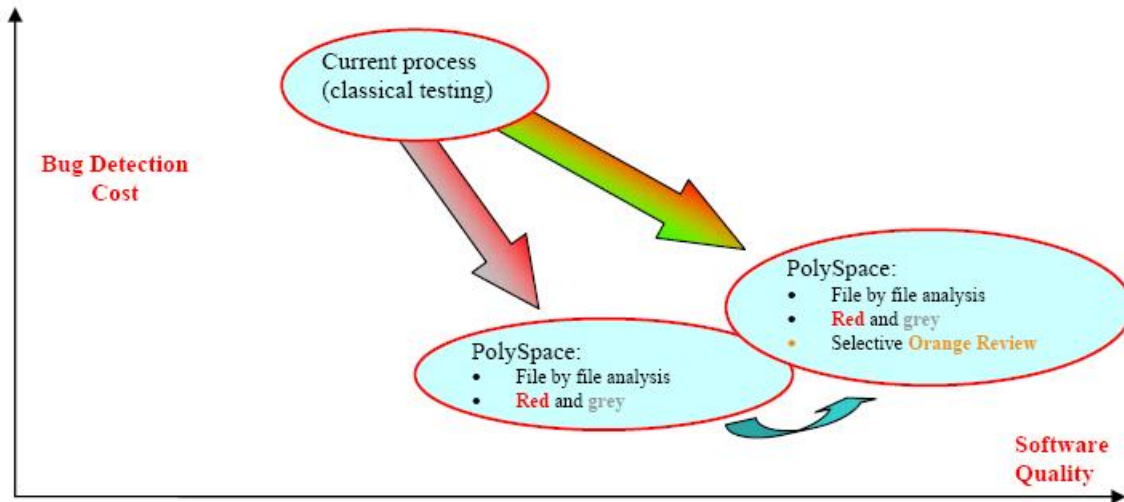
---

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

---

### When No Coding Rules Are Adopted

**During coding**, there are two recommended approaches:



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

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

---

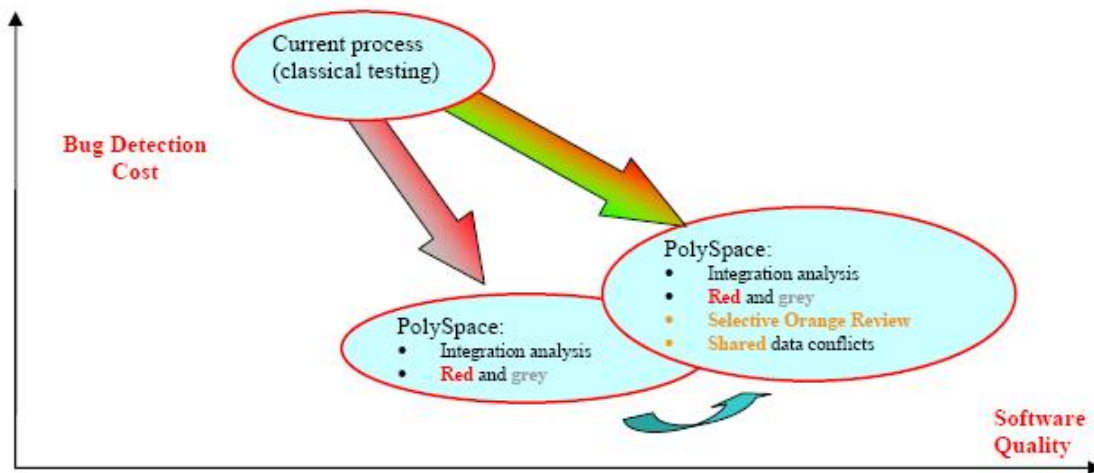
**Note** Using PolySpace verification on a single file is efficient. Even though the verification has no knowledge of the file context, experience shows that 50% of bugs detected by PolySpace verification can be found locally.

---



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

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



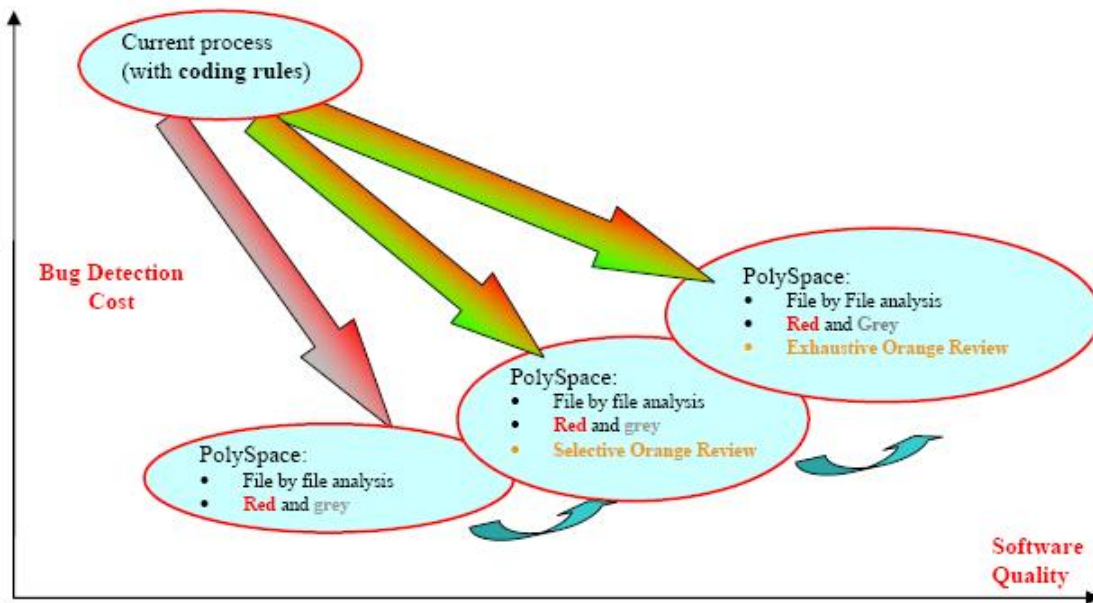
Again, the first approach is to use only the red and gray results — fix the red bugs, and check the dead code.

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

### **When Coding Rules Have Been Adopted**

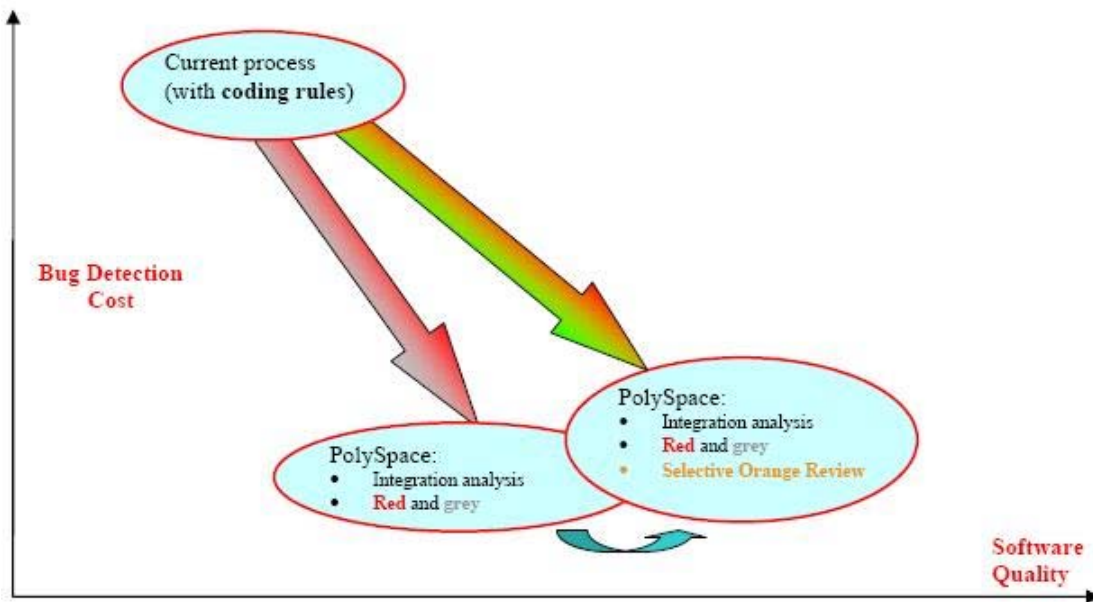
The main difference between this process and the previous process is in respect to the cost of bug detection. When PolySpace verification is used in conjunction with coding rules, the costs of bug detection are much lower.

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



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

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



---

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

---

### In a Certification Context

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

### As an Acceptance Tool

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



## Standard Development Process

### Overview

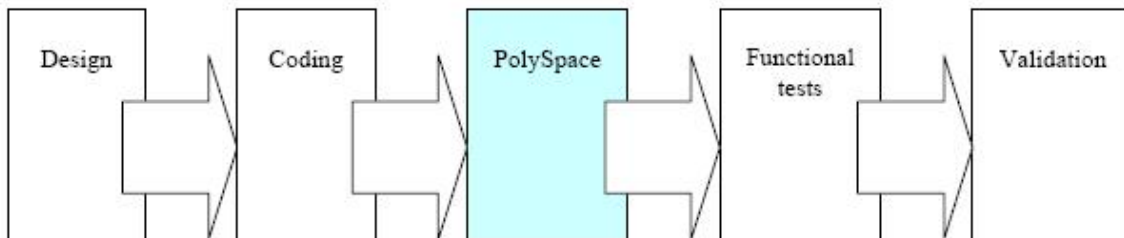
This approach is mainly for consideration by a project manager rather than a quality manager. It aims to improve productivity rather than to prove the quality of the application being analyzed.

### The Software Development Process

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

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

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



### The Objective of Using PolySpace Verification

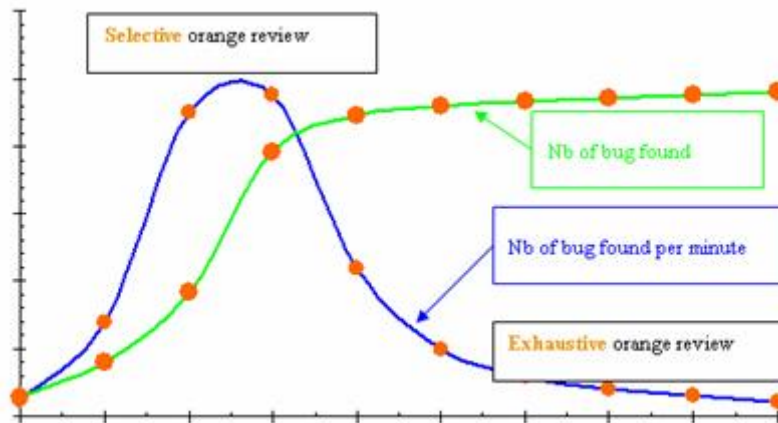
PolySpace verification will be used to improve the software quality and productivity. It will help the developer to find and fix bugs much quicker than the existing process. It will also improve the software quality by finding bugs which would otherwise be likely to remain in the software after delivery.

It does not prove the robustness of the code because the prime objective is to deliver code of at least similar quality to before, but to ensure that code is

produced in a predictable time frame with controlled and minimized delay and costs. Another approach for this purpose is described in the next section.

### The PolySpace Approach

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



### Cost/Benefits of a Selective Orange Review

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

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

### A Complementary Approach

A second approach is also possible which, unlike the first, focuses only on an increase in quality. If coding rules are applied, this second approach will

turn into a cheap and productive one as described by the second arrow on the illustration.

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

### **Integration with Configuration Management Tools**

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

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

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

### **Costs and Benefits**

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

- Red checks and bugs in gray checks. The number of bugs found thanks to these colors can vary from one user to another, but experience shows that on average, around 40 percent of verifications will reveal one or more red errors and/or will reveal bugs in gray code.
- Orange checks. Experience suggests that the time needed to find one bug per file varies from 5 minutes to 1 hour, and is typically around 30 minutes. This represents an average of two minutes per orange check review, and a total of 20 orange checks per package in Ada and 60 orange checks per file in C or C++.

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

- **75% of bugs are local in this type of code** — the selective orange review at integration phase reveals a of integration bugs, and the rest () of local bugs. Finding real integration bugs might require another process which requires coding rules to be efficient.
- **Setup time** — the time needed to setup the verification can be higher due to a lack of coding rules. Code modifications might be needed. Most of these modifications cannot be automatic without changes in the process.
- **Anomalies and complexity** — In this configuration, any particular file will receive more orange checks (about twice as many). These oranges are likely to be anomalies, and will make the orange check review more time consuming.
- **An exhaustive orange review can take 25 man-days for a 50000 line project** — This would represent the effort where the aspiration is for bug free software, assuming that a 50000 line application contains about 3000 orange checks

## Rigorous Development Process: Introducing Tools and Coding Rules

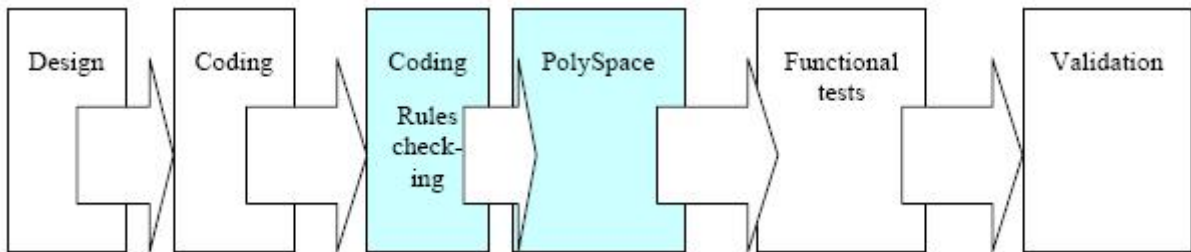
### Overview

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

## The Software Development Process

This section describes how to use PolySpace verification within a development process.

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



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

## The PolySpace Approach

Use PolySpace on a file by file verification basis.

- The “main” used to analyze each file is very often **automatically generated by the project**, and not by PolySpace (unlike the standard approach).
- **Initialization ranges** should be applied to input data. For instance, if a variable “x” is read by functions in the file, and if x can be initialized to any value between 1 and 10, this information should be included as part of the verification.
- **[Optional]** Some properties of output variables might be checked. For instance, if a variable “y” is returned by a function in the file and should always be returned with a value in the range 1 to 100, then PolySpace can flag instances where that range of values might be breached.
- Red errors will be fixed and gray code examined, and an exhaustive orange review will be completed.
- The usage of permissive options is not advisable at this stage.

---

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

---

### A Complementary Approach

A second approach is also possible. Use PolySpace at integration phase to track integration bugs, and review:

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

### Costs and Benefits

With this approach, using PolySpace to find bugs will typically provide the following benefits

- 3-5 orange checks per file, 3 gray checks per file yielding an average of 1 bug per file. Typically, 2 of these oranges might represent the same bug, and another might represent an anomaly.
- An average of 2 verifications by PolySpace per file is typical before the file can be checked-in to the configuration management system.
- The average verification time is about 15 minutes.

---

**Note** If the development process includes data rules which determine how the data flow are designed, the benefits might even be higher. The data rules would implicitly reduce the potential for PolySpace to find integration bugs.

---

With this approach, using PolySpace verification to find integration bugs might bring the following results. On a typical 50000 line project:

- A selective orange check review might reveal **one integration bug per hour of orange** code review and takes about after 6 hours, which long enough to review the main orange points throughout the whole

application. This represents a step towards an exhaustive orange check review. Spending more time is unlikely to be efficient, and wont guarantee that no bugs remain.

- An exhaustive orange review takes between 4 and 6 days, given that a 50000 lines of code application might contain about 400-800 orange checks.

## A Quality/Qualification Approach

### Overview

Quality managers are likely to be interested in this approach.

### The Software Development Process

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

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

### The Objective of Using PolySpace Verification

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

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

### The PolySpace Approach

- For data and control flow verification and shared data detection, PolySpace can be used on the whole application or on a subsection of the application.

- For robustness unit tests (as opposed to functional unit tests), PolySpace might be used in the same way as the method applied to the Rigorous development process.

### **Costs and Benefits**

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

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

## **Code Acceptance Criterion**

### **Overview**

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

### **The Software Development Process**

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

### **The Objective of Using PolySpace Verification**

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

### **The PolySpace Approach**

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

- No compilation errors
- No compilation warning errors



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

## Choosing the Type of Verification You Want to Perform

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

- **Robustness Verification** – Prove that the software works under all conditions, including “abnormal” conditions. This can be thought of as “worst case” analysis.
- **Contextual Verification** – Prove that the software works under normal working conditions. This can limit the amount of analysis that needs to be done by providing the software with the ranges of various parameters, so that the code only needs to be verified within these ranges.

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

When performing contextual verification, you can use several PolySpace options to reduce the number of orange checks. You can use DRS to specify the ranges for your variables, thereby limiting the verification to these cases. You also can create a very detailed main generator.

It is important to note that DRS should be used specifically to perform contextual verification, it is not simply a means to reduce oranges.



# PolySpace Class Analyzer

---

- “Analyzing C++ Classes” on page 3-2
- “How the Class Analyzer Works” on page 3-3
- “Types of Classes” on page 3-13

## 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 3-10

### 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` directory:

```
*****
***
*** 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;
    OtherClass (int intx);    // code is missing
public:
    int getMember(void) {return x;}; // NIV is warning
};
class MyClass
{
    OtherClass m_loc;
public:
    MyClass(int intx) : m_loc(0) {};
    void show(void) {
        int wx, wl;
        wx = m_loc.getMember();
```

```

    w1 = wx*wx + 2;    // Possible overflows because OtherClass
                      // member is assigned to full range
};
};

```

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 class. In the example below, the class `Point` derives 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.

## 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” on page 3-13  
“Simple Inheritance” on page 3-15  
“Multiple Inheritance” on page 3-16  
“Abstract Classes” on page 3-17  
“Virtual Inheritance” on page 3-18  
“Other Types of Classes” on page 3-19

### Simple Class

Consider the following class:

Stack.h

```
#define MAXARRAY 100

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

public:
    int Top (void);
    int IsEmpty (void);
    int 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:

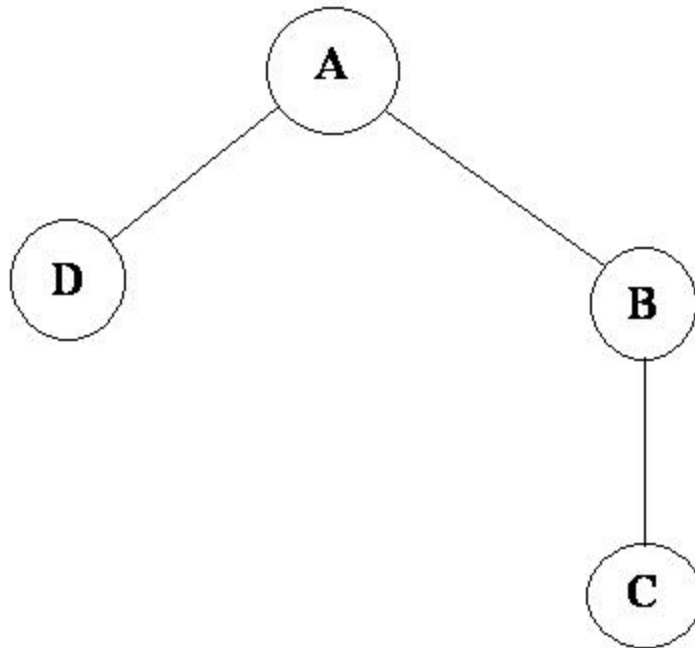


- The `stack::push` method may write after the last element of the array, resulting in the OBAI orange check at line 28.
- If called before `push`, the `stack::top` method will access element -1, resulting in the OBAI and NIV checks at line 13.

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.

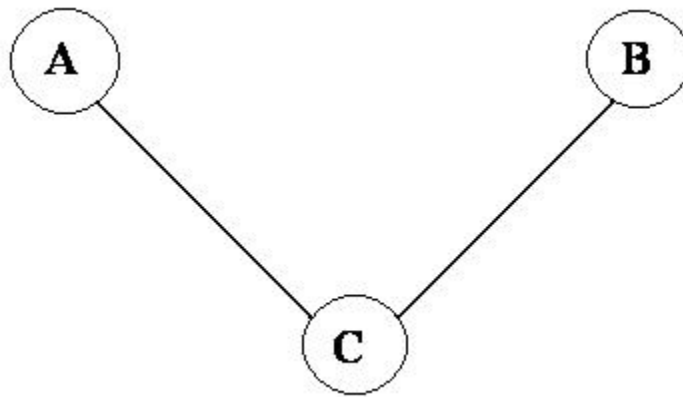
In a case such as this, PolySpace allows you to run the following verifications:

- 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:



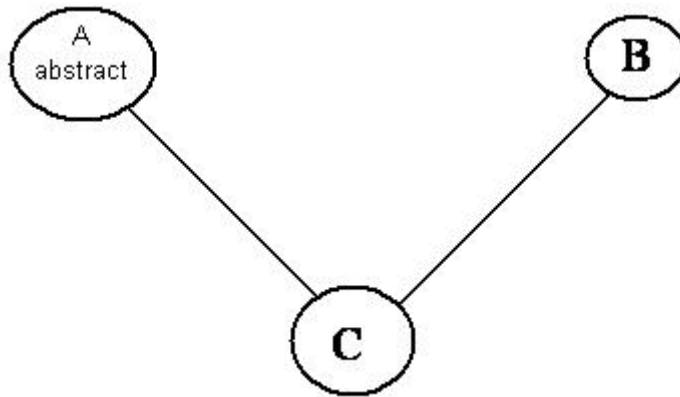
A and B are base classes of C.

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.

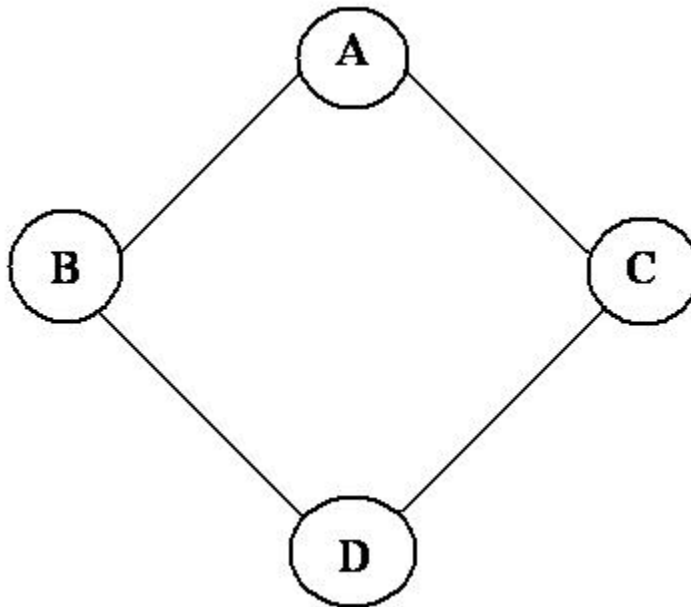
A and B are base classes of C.

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.

# Setting Up a Verification Project

---

- “Creating a Project” on page 4-2
- “Setting Up Project to Check Coding Rules” on page 4-19
- “Setting Up Project for Generic Target Processors” on page 4-24

# Creating a Project

In this section...
“What Is a Project?” on page 4-2
“Project Directories” on page 4-3
“Opening PolySpace Launcher” on page 4-3
“Specifying Default Directory” on page 4-6
“Creating New Projects” on page 4-8
“Opening Existing Projects” on page 4-9
“Specifying Source Files” on page 4-10
“Specifying Include Directories” on page 4-12
“Specifying Results Directory” on page 4-14
“Specifying Analysis Options” on page 4-15
“Configuring Text and XML Editors” on page 4-16
“Saving the Project” on page 4-17

## What Is a Project?

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

A project includes:

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

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

A project file has one of the following file types:



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

## Project Directories

Before you begin verifying your code with PolySpace software, you must know the location of your 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 directory, and then in that directory, create separate directories for the source files, include files, and results. For example:

```
polyspace_project/
```

- sources
- includes
- results

## Opening PolySpace Launcher

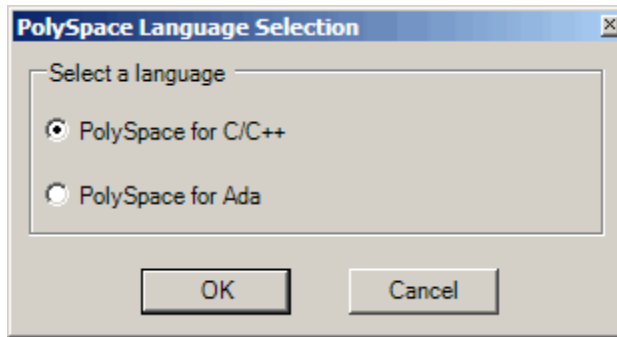
You use the PolySpace Launcher to create a project and start a verification.

To open the PolySpace Launcher:

- 1 Double-click the PolySpace Launcher icon.

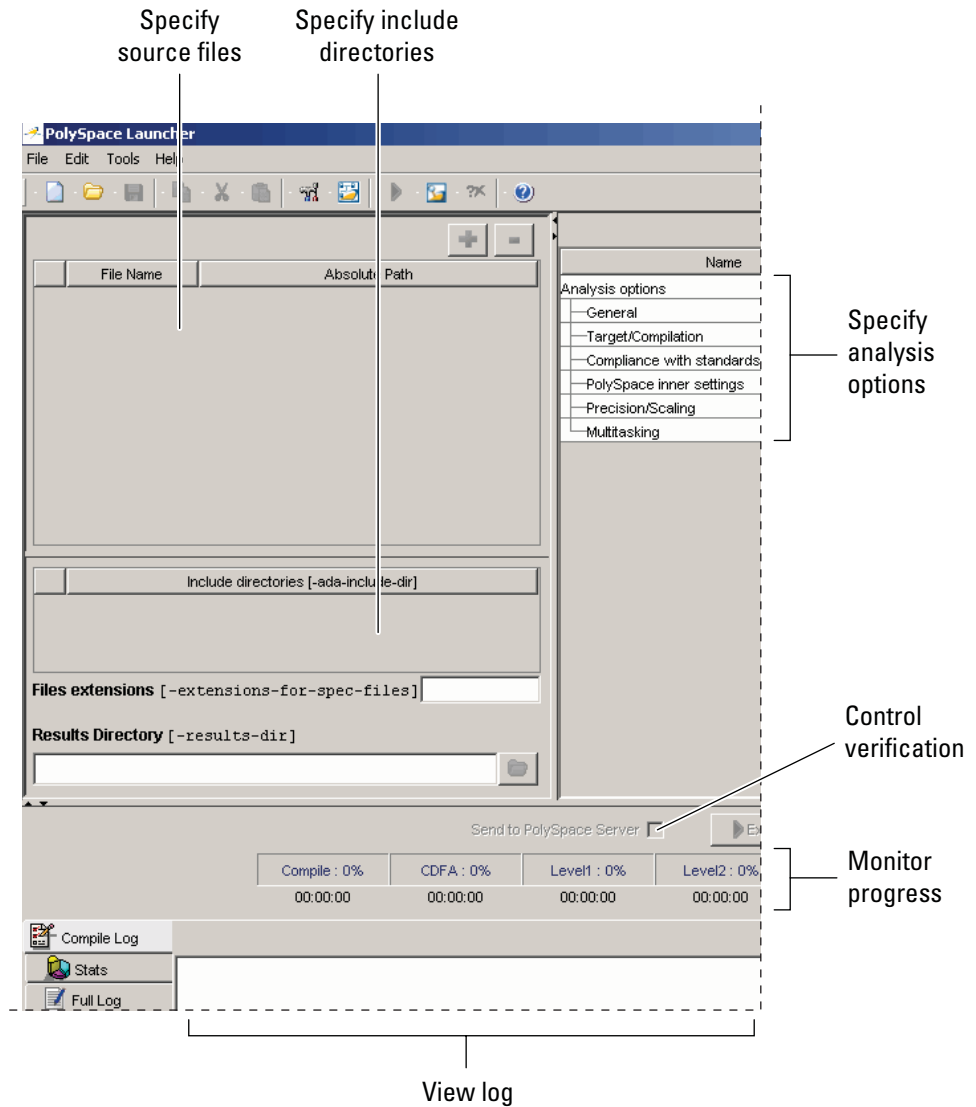


- 2 If you have both PolySpace for C/C++ and PolySpace for Ada products on your system, the **PolySpace Language Selection** dialog box will appear.



Select **PolySpace for C/C++**, then lick **OK**.

The PolySpace Launcher window appears:



The Launcher window has three main sections.

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

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

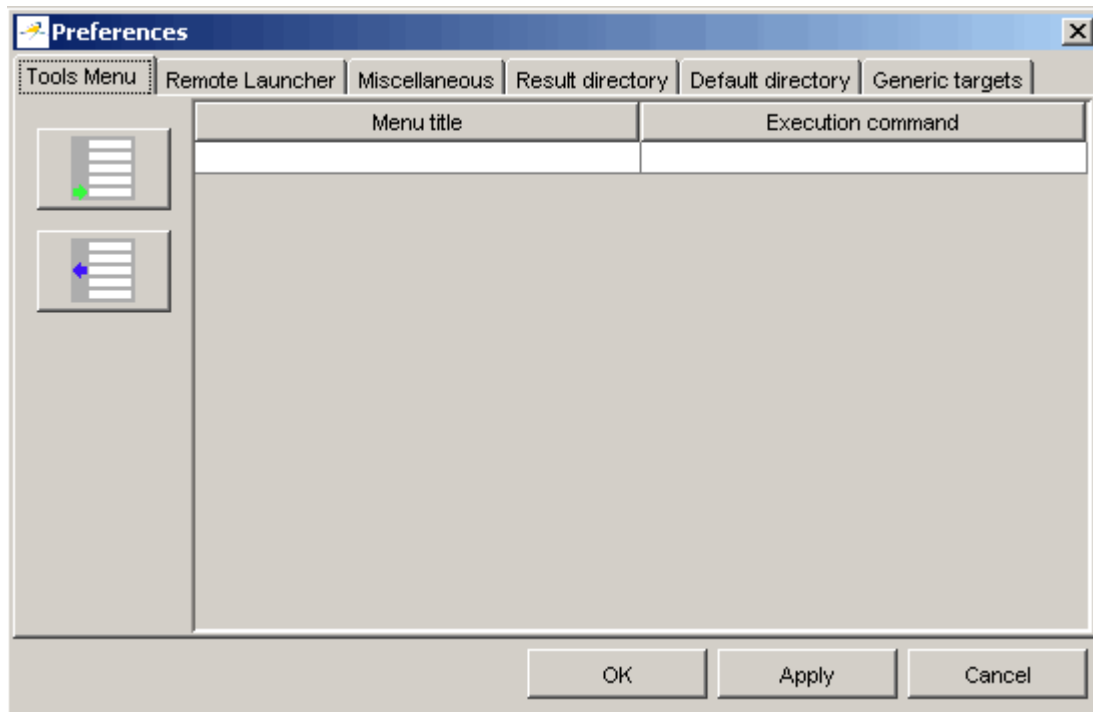
### Specifying Default Directory

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

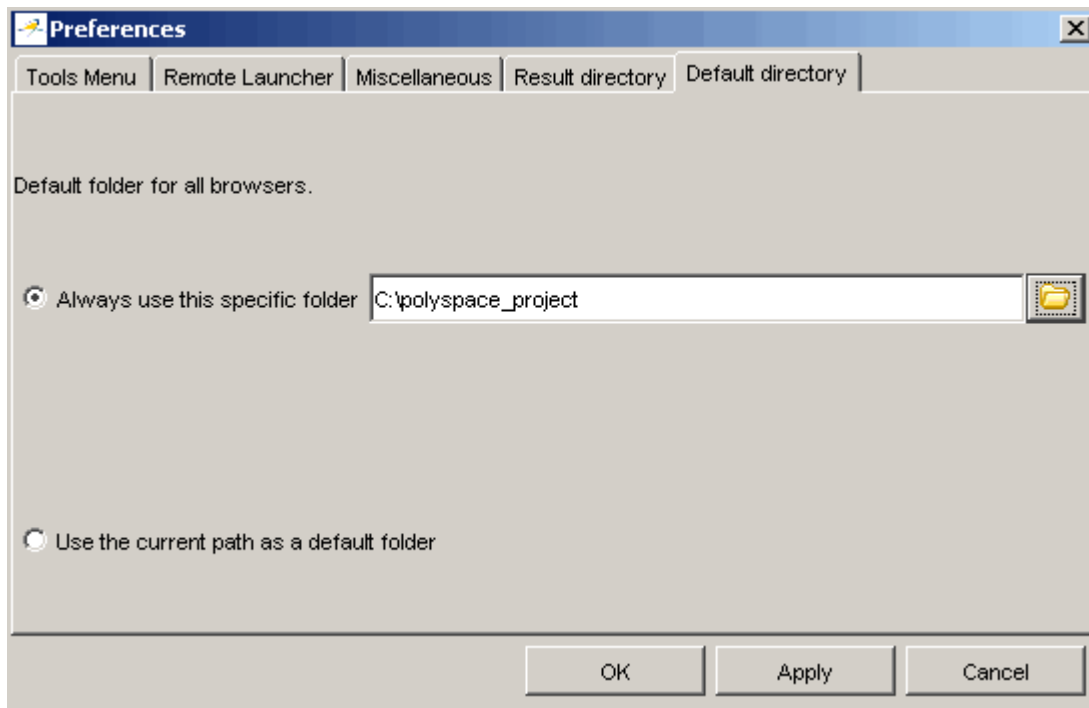
To change the default directory to the project directory:

- 1 Select **Edit > Preferences**.

The **Preferences** dialog box appears.



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



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

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

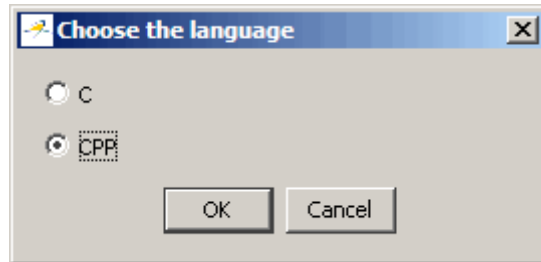
**5** Click **OK** to apply the changes and close the dialog box.

### Creating New Projects

To create a new project:

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

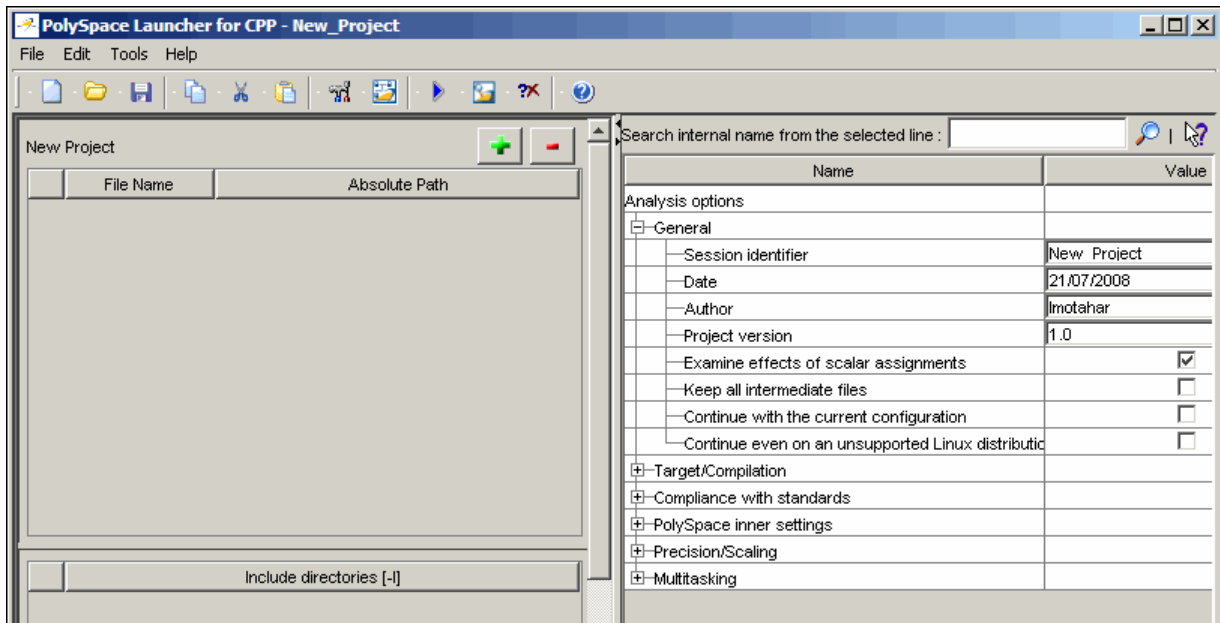
The **Choose the language** dialog box appears:



- 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

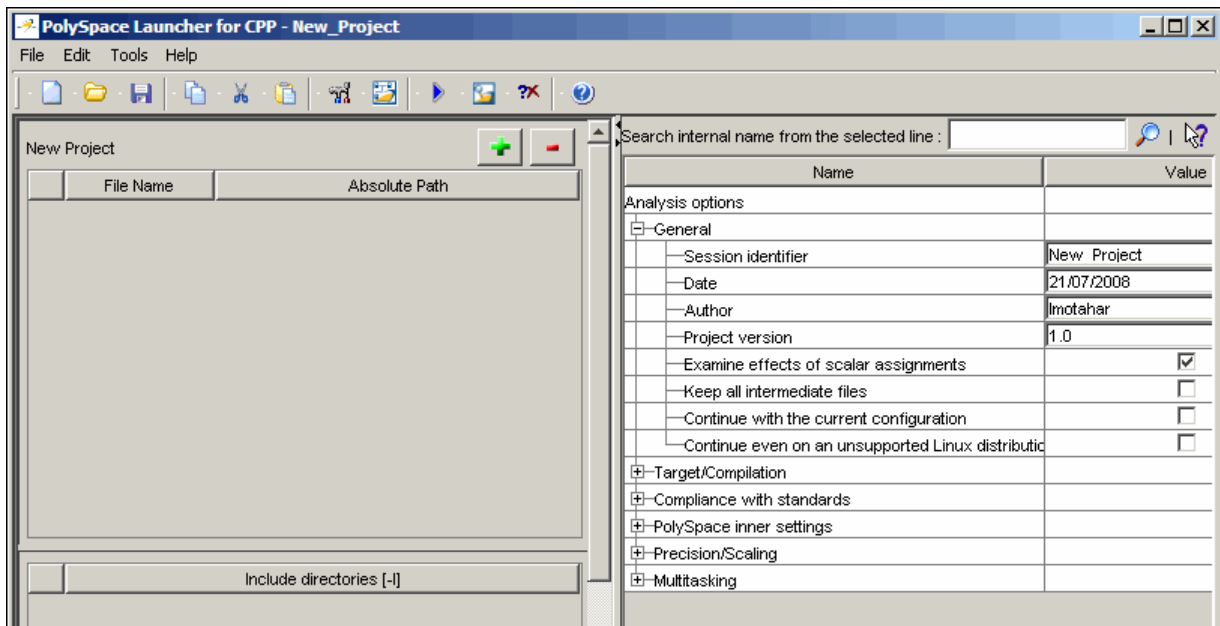
To open an existing project:

- 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

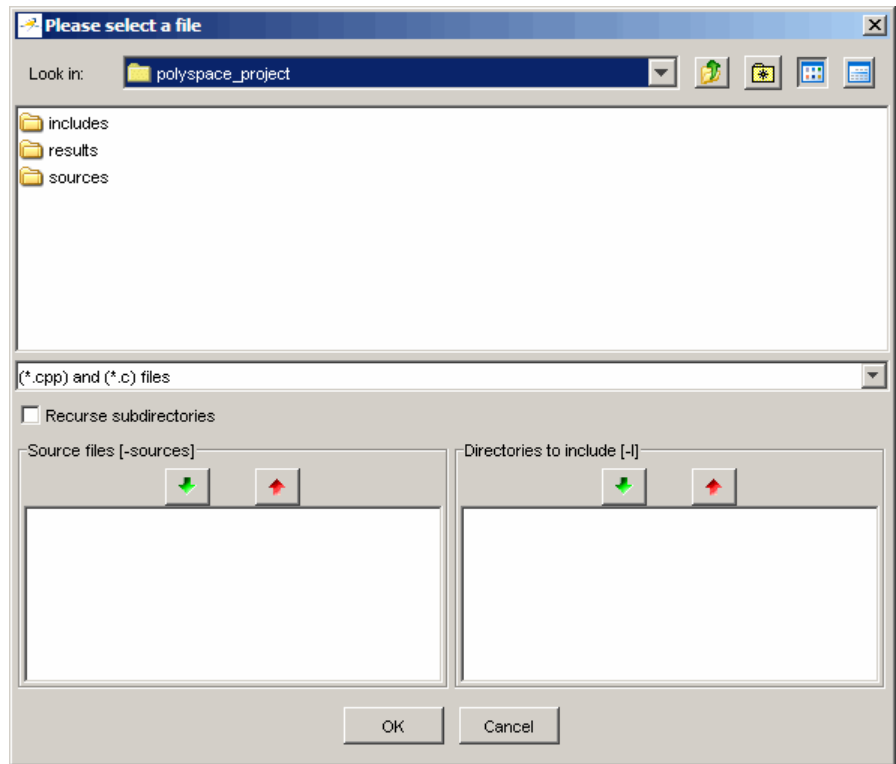
To specify the source files for your project:

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



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

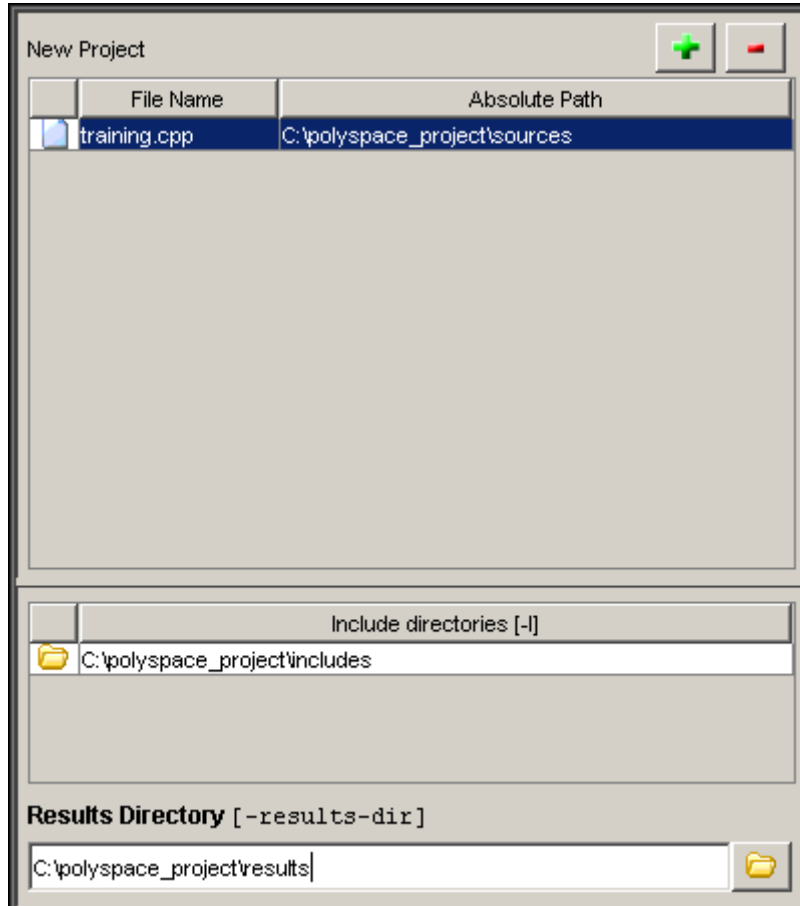
---

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

---

- 4 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.



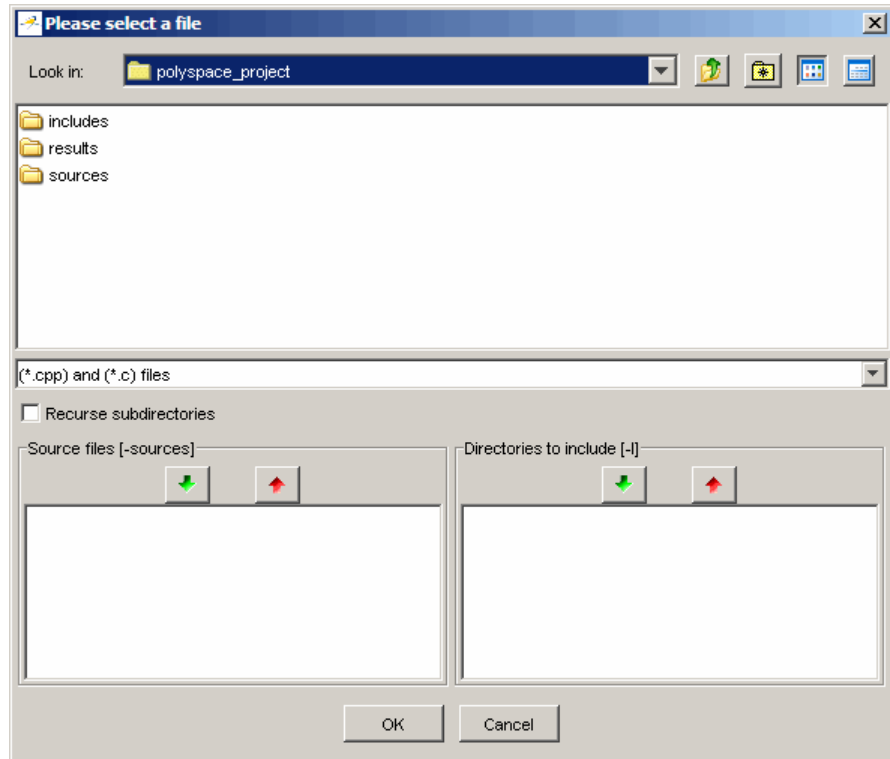
### Specifying Include Directories

To specify the include directories 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.



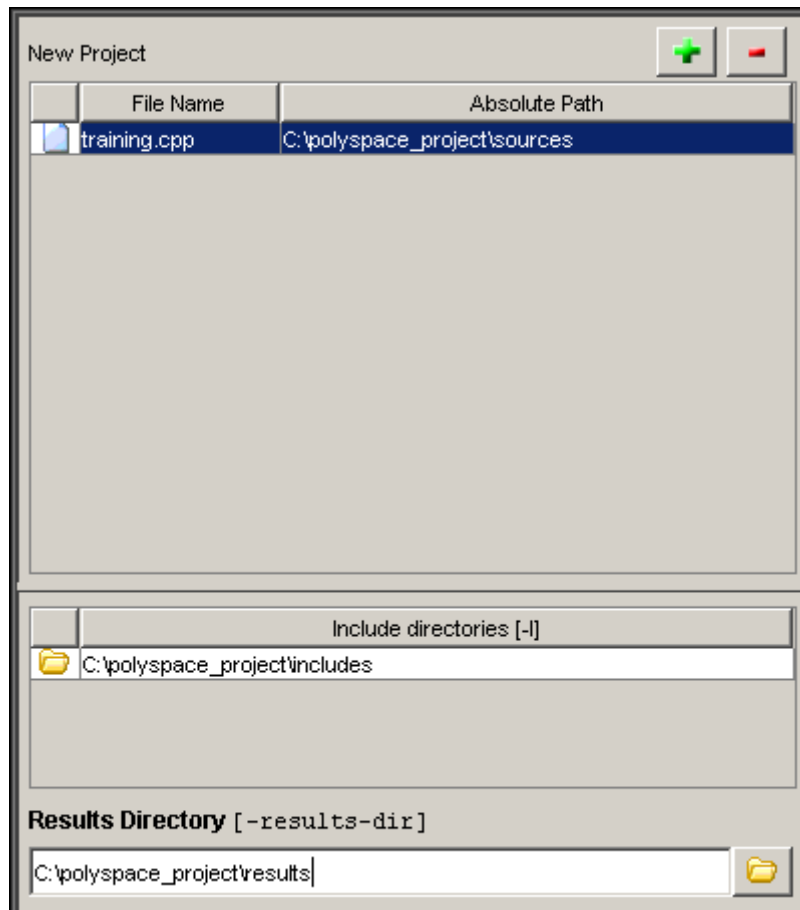
- 2** In the **Look in** field, navigate to your project directory.
- 3** Select the directory containing the include files for your project, then click the green down arrow button in the **Directories to include** section.



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

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

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

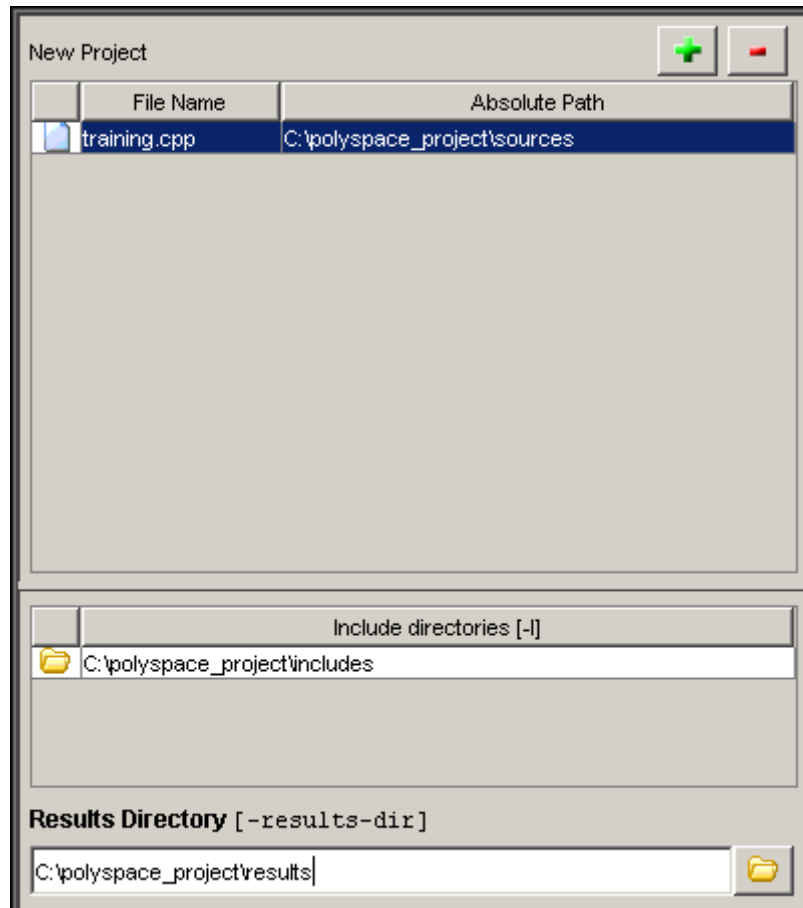


### Specifying Results Directory

To specify the results directory for the project:

- 1 In the **Results Directory** section of the Launcher window, specify the full path of the directory that will contain your verification results. For example: 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	Example Project	-prog
Date	11/12/2008	-date
Author	user name	-author
Project version	1.0	-verif-version
Keep all intermediate files	<input type="checkbox"/>	-keep-all-files
Continue with the current configuration	<input checked="" type="checkbox"/>	-continue-with-existing-host
Continue even on an unsupported Linux	<input type="checkbox"/>	-allow-unsupported-linux
[+] Target/Compilation		
[+] Compliance with standards		
[+] PolySpace inner settings		
[+] Precision/Scaling		
[+] Multitasking		

3 Specify the appropriate general parameters for your project.

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

## Configuring Text and XML Editors

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

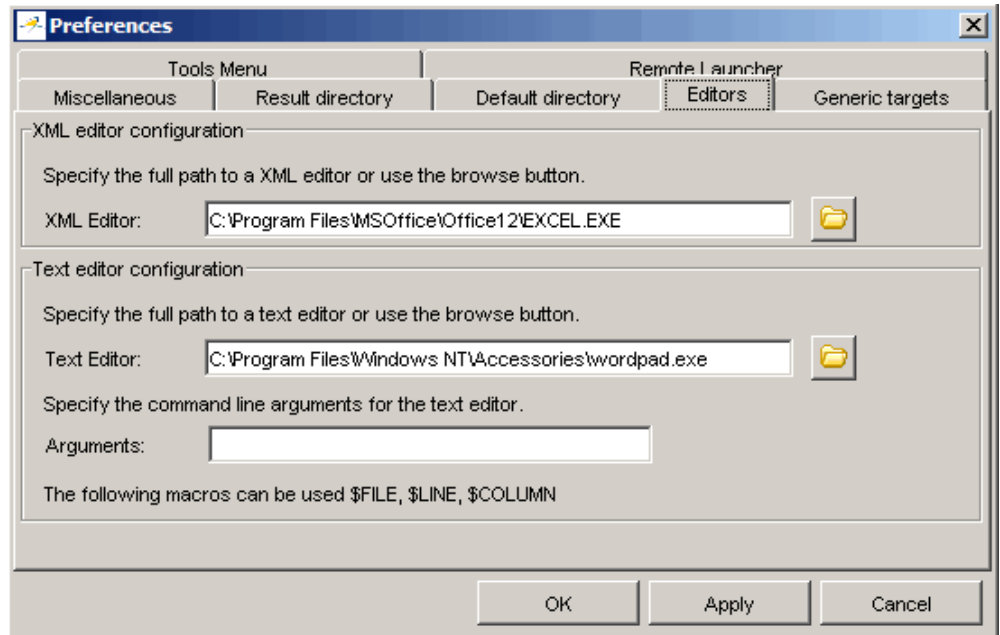
To configure your text and .XML editors:

1 Select **Edit > Preferences**.

The Preferences dialog box opens.

2 Select the **Editors** tab.

The Editors tab opens.

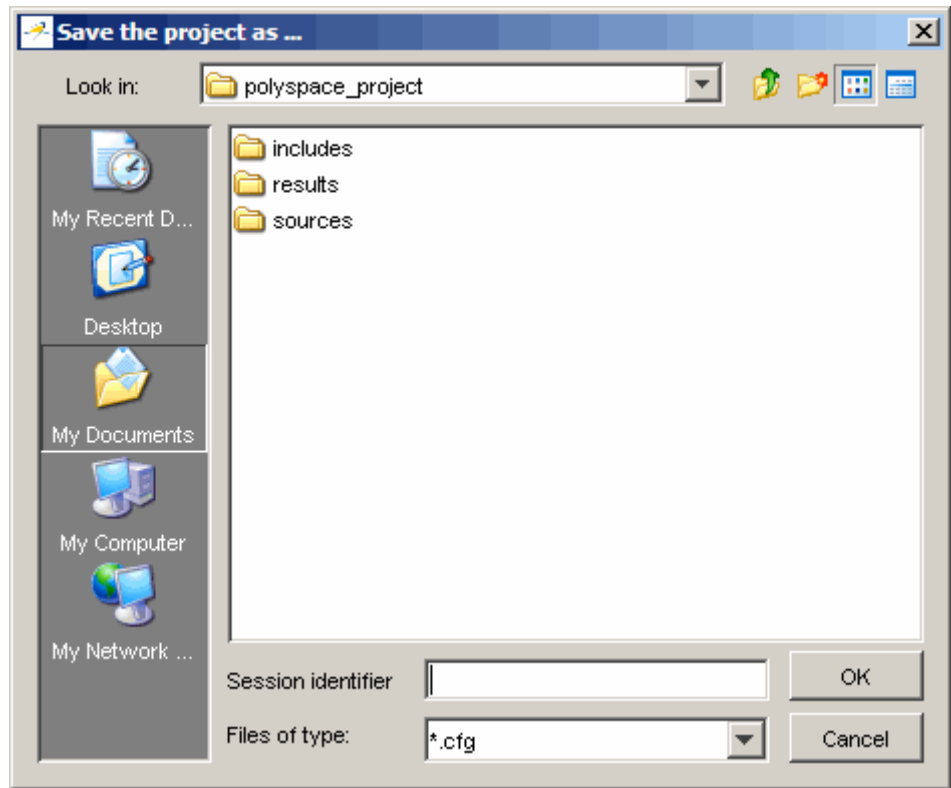


- 3** Specify an XML editor to use to view JSF reports.
- 4** Specify a Text editor to use to view source files from the Viewer logs.
- 5** Click **OK**.

## 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 directory.
- 3** In **Session identifier**, enter a name for your project.
- 4** Click **OK** to save the project and close the dialog box.



## Setting Up Project to Check Coding Rules

### In this section...

“PolySpace JSF C++ Checker Overview” on page 4-19

“Checking Compliance with JSF++ Coding Rules” on page 4-19

“Creating a JSF++ Rules File” on page 4-20

“Excluding Files from JSF++ Checking” on page 4-22

### 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. The JSF C++ checker can check 120 of the 221 JSF++ programming rules .

---

**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](http://www.jsf.mil/downloads/documents/JSF_AV_C++_Coding_Standards_Rev_C.doc).

---

### Checking Compliance with JSF++ Coding Rules

To check JSF++ 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 JSF++ violations, you run the verification again.

To set the JSF++ checking option:

- 1 In the Analysis options, select **Compliance with standards > Check JSF-C++: 2005 rules**.

The software displays the two JSF++ options: `jsf-coding-rules` and `includes-to-ignore`.

<input type="checkbox"/> Check JSF-C++: 2005 rules	<input checked="" type="checkbox"/>		
Rules configuration		...	<code>-jsf-coding-rules</code>
Files and directories to ignore		...	<code>-includes-to-ignore</code>

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

- 2 Select the **Check JSF-C++: 2005 rules** check box.
- 3 Specify which JSF++ rules to check and which, if any, files to exclude from the checking.

---


**Note** For more information on using the JSF C++ checker, see Chapter 12, “JSF C++ Checker”.

---

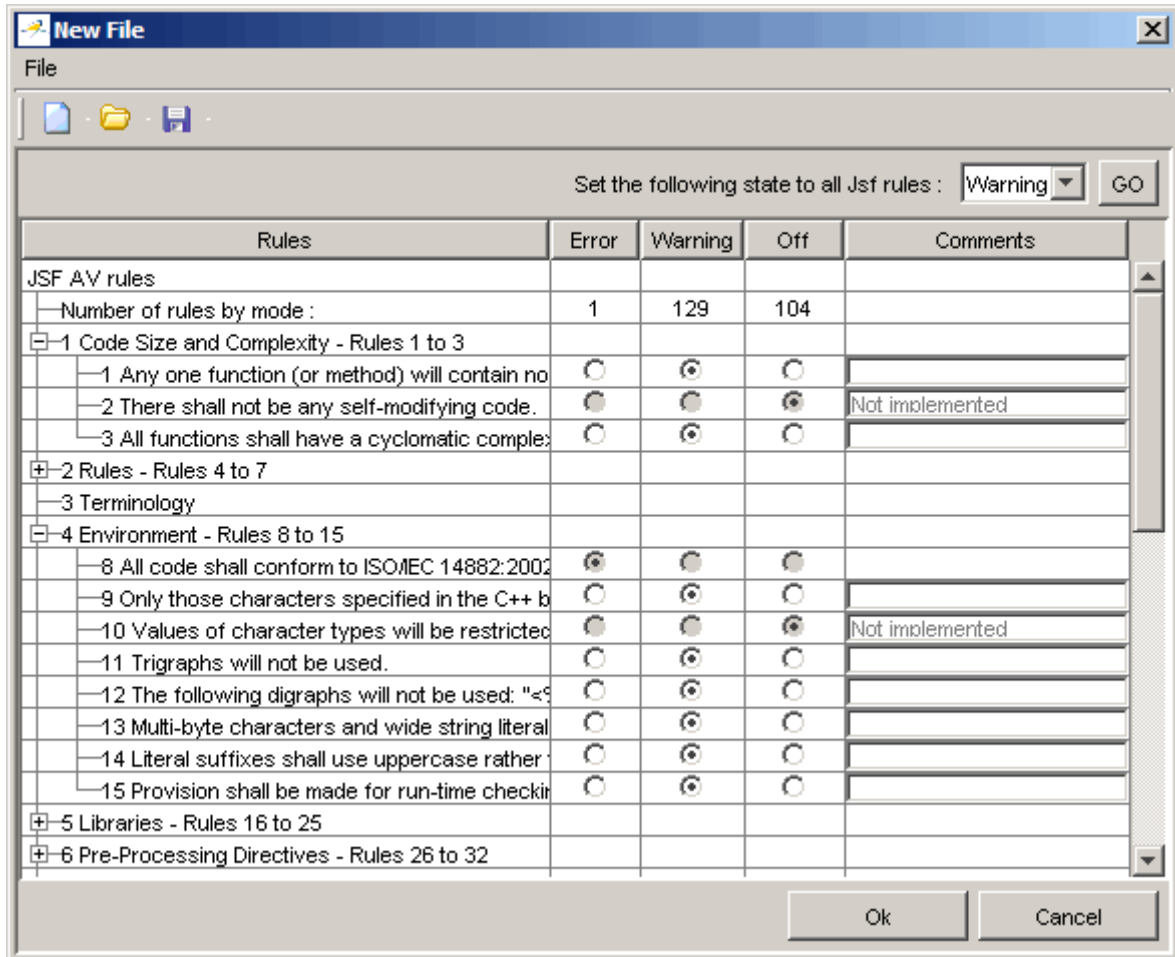
### Creating a JSF++ Rules File

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 button  to the right of the **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.

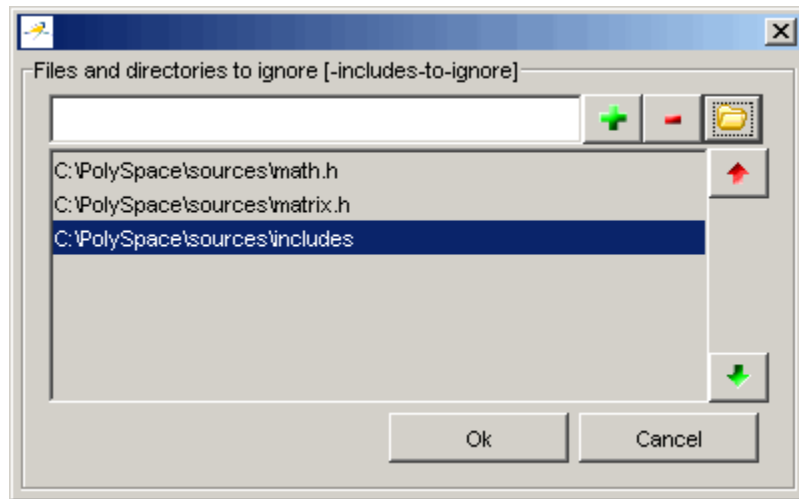
### Excluding Files from JSF++ Checking

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 directories to ignore** option.

The **Files and directories to ignore (includes-to-ignore)** dialog box opens.



- 2 Click the folder icon 

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

- 3 Select the files or directories you want to exclude.
- 4 Click **OK**.

The select files and directories 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” on page 4-24
“Creating Project Model Files” on page 4-25
“Viewing Existing Generic Targets” on page 4-25
“Defining Generic Targets” on page 4-26
“Deleting a Generic Target ” on page 4-28
“Common Generic Targets” on page 4-28
“Creating a Configuration File from a PolySpace Project Model File” on page 4-30

## 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 directories, and a results directory 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.

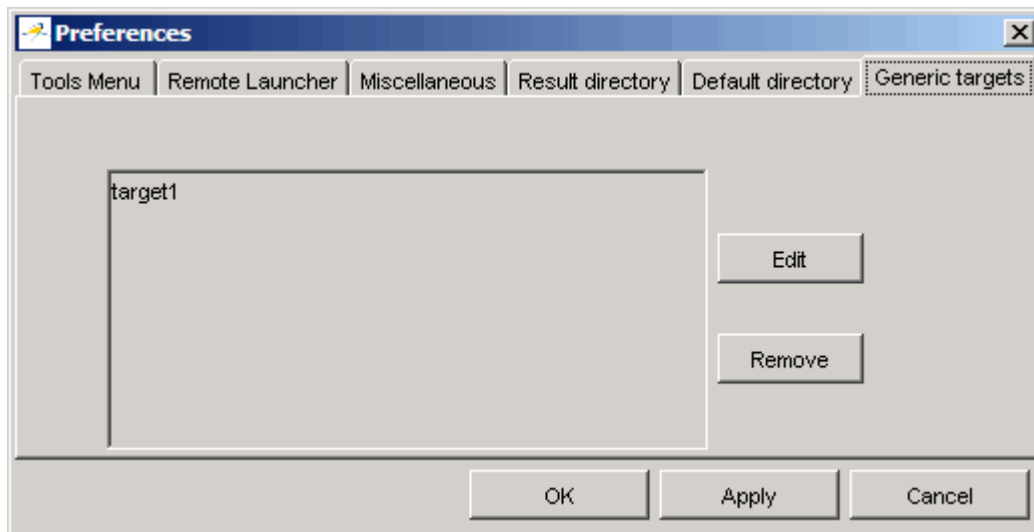
To view existing generic targets:

- 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

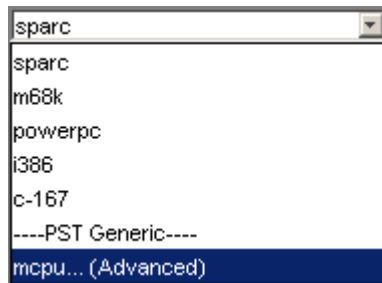
If your application is designed for a custom target processor, you can configure many basic characteristics of the target by selecting the PST Generic target, and specifying the characteristics of your processor.

To configure a generic target:

To define a generic target:

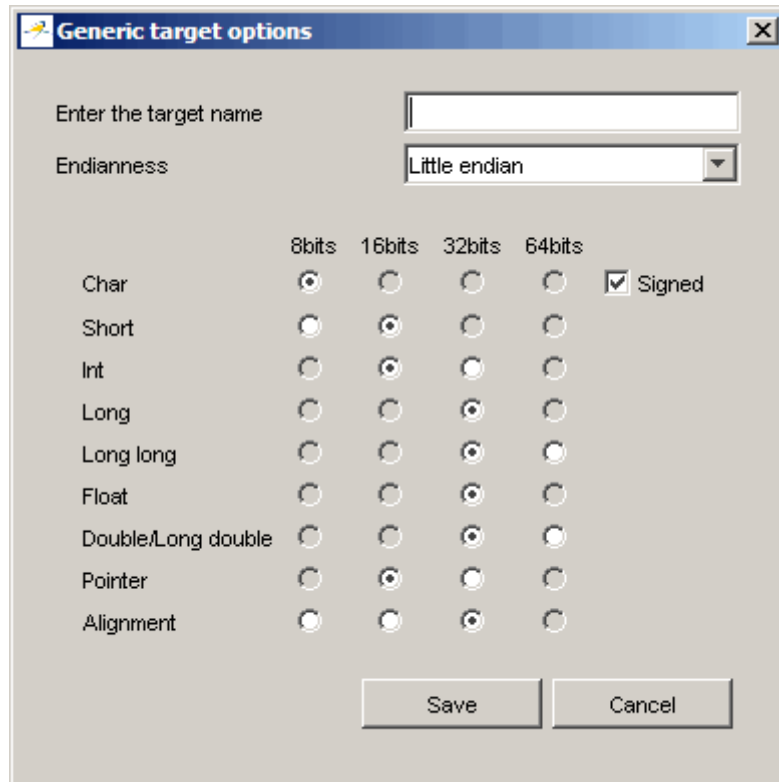
- 1** In **Analysis options**, expand **Target/Compilation**.
- 2** Click the down arrow to open the **Target processor type** menu.





**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 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.

### 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.

### 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]`.

### 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 directory and source files.

---

**Note** If you enter the results directory 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
- “Applying Data Ranges to External Variables and Stub Functions (DRS)” on page 5-14

## Setting Up a Target

In this section...
“Target/Compiler Overview” on page 5-2
“Specifying Target/Compilation Parameters” on page 5-2
“Predefined Target Processor Specifications (size of char, int, float, double...)” on page 5-3
“Generic Target Processors” on page 5-5
“Compiling Operating System Dependent Code (OS-target issues)” on page 5-5
“Ignoring or Replacing Keywords Before Compilation” on page 5-9
“How to Gather Compilation Options Efficiently” on page 5-12

### 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 <sup>1</sup>	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.

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

---

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

## Generic Target Processors

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.

For more information, see “Setting Up Project for Generic Target Processors” on page 4-24.

## 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 5-5
- “My Target Application Runs on Linux” on page 5-8
- “My Target Application Runs on Solaris” on page 5-8
- “My Target Application Runs on Vxworks” on page 5-9
- “My Target Application Does Not Run on Linux, vxworks nor Solaris” on page 5-9

### 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	<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

<b>-OS-target</b>	<b>Compilation flags</b>	<b>-include file</b>	<b>Minimum set of options</b>
	<pre>-D__unix__ -Dlinux -D__linux__ -D__linux__ -Di386 -D__i386 -D__i386__ -Di686 -D__i686__ -D__i686__ -Dpentiumpro -D__pentiumpro -D__pentiumpro__</pre>		<p>been installed in the directory &lt;product_dir&gt;</p>
vxWorks	<pre>-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__unix__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__ -D__svr4 -D__SVR4</pre>	<pre>&lt;product_dir&gt;/ cininclude/ pstvxworks. h</pre>	<pre>polyspace-[desktop-]cpp \ -OS-target vxworks \ -I /your_path_to/ Vxworks_include_directories</pre>

<b>-OS-target</b>	<b>Compilation flags</b>	<b>-include file</b>	<b>Minimum set of options</b>
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__unix__ -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 -D_POSIX_SOURCE		polyspace-[desktop-]cpp \ -OS-target no-predefined-OS \ \ -I /your_path_to/ MyTarget_include_directories

-OS-target	Compilation flags	-include file	Minimum set of options
	-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 directory `/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 directory `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
```

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_directories
```

### **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_directories
```

## **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 files**.
- 3** Select `myTpl.pl` using the browse button.

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

### **Content of the myTpl.pl file**

```
#!/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-z0-9] Any single character in the set
# [^a-z0-9] Any character not in set
# \d A digit same as
# \D A non digit same as [^0-9]
# \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? 0 or 1 occurrence of x
# x* 0 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 localisation 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);
```

## Applying Data Ranges to External Variables and Stub Functions (DRS)

### In this section...

“Overview of Data Range Specifications (DRS)” on page 5-14

“Specifying Data Ranges” on page 5-14

“File Format” on page 5-15

“Variable Scope” on page 5-17

“Performing Efficient Module Testing with DRS” on page 5-19

“Reducing Oranges with DRS” on page 5-20

### Overview of Data Range Specifications (DRS)

By default, PolySpace verification assumes that all data inputs are set to their full range. Therefore, nearly any operation on these inputs could produce an overflow. The Data Range Specifications (DRS) module allows you to set external constraints on global variables and stub function return values. This can substantially reduce the number of orange checks in the verification results.

---

**Note** You can only apply data ranges to variables with external linkages (see “Variable Scope” on page 5-17) and stubbed functions.

---

### 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 5-15.
- 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`.

Static variables are not supported by DRS. The following table summarizes possible uses:

	<b>init</b>	<b>permanent</b>	<b>globalassert</b>	<b>comments</b>
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)

	<b>init</b>	<b>permanent</b>	<b>globalassert</b>	<b>comments</b>
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	No effect	No effect	No effect	
Complete structure	No effect	No effect	No effect	
Array cell	No effect	No effect	No effect	Example: array[0], array[10] ...
Stubbed function return	No effect	Ok	No effect	Stubbed function returning integral or floating point

---

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

---



---

**Note** DRS can initialize arrays of structures, structures of arrays, etc., as 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).

---

## 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	<b>Reduces</b> the number of oranges, (compared with a standard PolySpace verification)	Input data that were full range are set to a smaller range.	↓	↑
Outputs	globalassert	<b>Increases</b> 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	<b>Increases</b> the number of oranges, (compared with a standard PolySpace verification)	Data that were constant are set to a wider range.	↑	↓

## Reducing Oranges with DRS

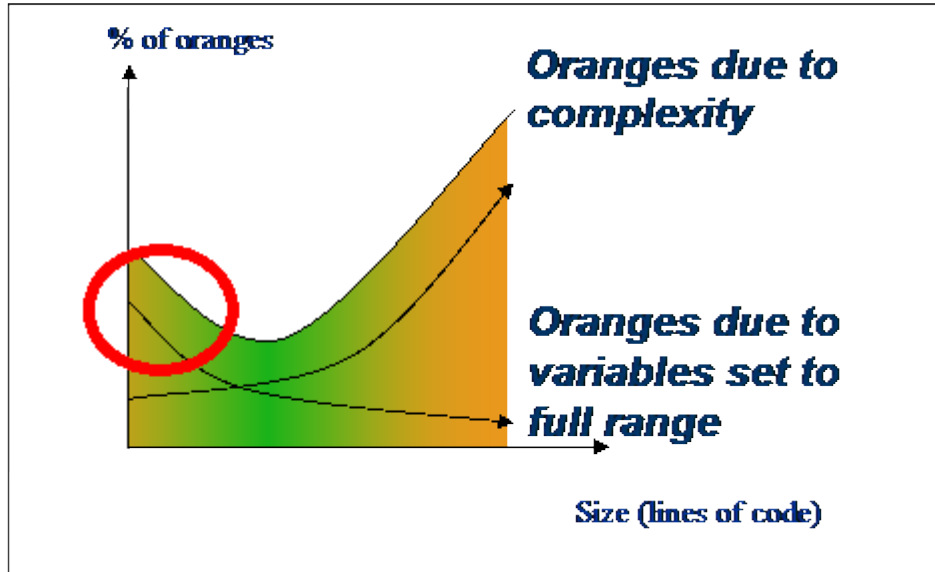
When performing robustness (worst case) verification, data inputs are always set to their full range. Therefore, every operation on these inputs, even a simple “one\_input + 10” can produce an overflow, as the range of one\_input varies between the min and the max of the type.

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?

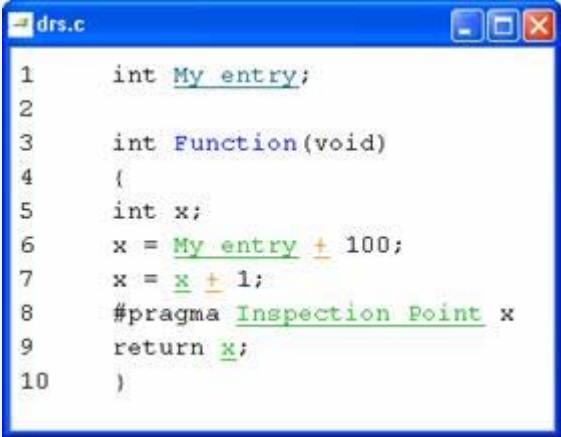
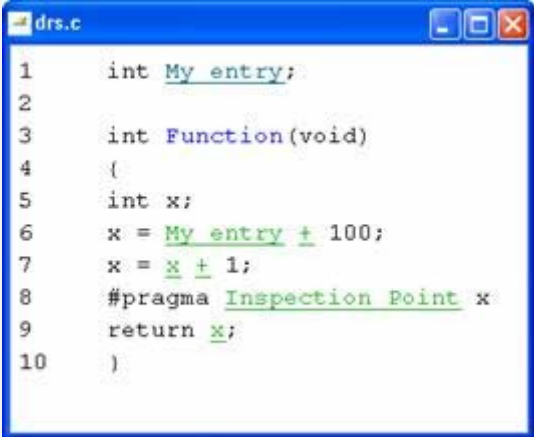
Removing oranges caused by full-range (worst-case) data can drastically reduce the total number of orange checks, especially when used on verifications of small files or modules. However, the number of orange checks caused by code complexity is not effected by DRS. For more information on oranges caused by code complexity, see “Considering the Effects of Application Code Size” on page 8-46, and “Why Should there be an Optimum Size?” on page 8-33.

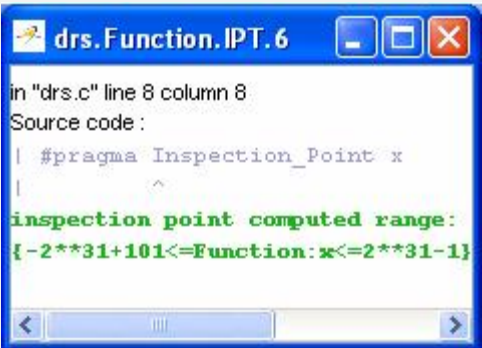
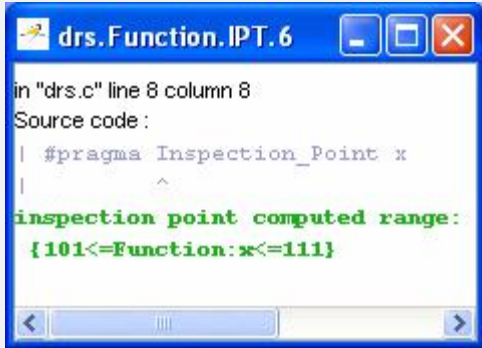
This section describes how DRS reduces oranges on files or modules only.

### 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
 <pre> 1  int My_entry; 2 3  int Function(void) 4  { 5  int x; 6  x = My_entry + 100; 7  x = x + 1; 8  #pragma Inspection Point x 9  return x; 10 } </pre>	 <pre> 1  int My_entry; 2 3  int Function(void) 4  { 5  int x; 6  x = My_entry + 100; 7  x = x + 1; 8  #pragma Inspection Point x 9  return x; 10 } </pre>
<ul style="list-style-type: none"> <li>• With “My_entry” being full range, the addition “+” is orange,</li> <li>• the result “x” is equal to all values between [min+100 max]</li> <li>• Due to previous computations, x+1 can here overflow too, making the addition “+”orange.</li> </ul>	<ul style="list-style-type: none"> <li>• With “My_entry” being bounded to [0,10], the addition “+” is green</li> <li>• the result “x” is equal to [100,110]</li> <li>• Due to previous computations, x+1 can NOT overflow here, making the addition “+” green again.</li> </ul>

<b>Without DRS</b>	<b>With DRS – 2 Oranges Removed + Return Statement More Accurate</b>
And the returned result is between [min+101 max]	And the returned result is between [101,111]
 <pre>drs.Function.IPT.6 in "drs.c" line 8 column 8 Source code :   #pragma Inspection_Point x   inspection point computed range: {-2**31+101&lt;=Function:x&lt;=2**31-1}</pre>	 <pre>drs.Function.IPT.6 in "drs.c" line 8 column 8 Source code :   #pragma Inspection_Point x   inspection point computed range: {101&lt;=Function:x&lt;=111}</pre>



# Preparing Source Code for Verification

---

- “Stubbing” on page 6-2
- “Preparing Code for Variables” on page 6-14
- “Preparing Code for Built-in Functions ” on page 6-18
- “Types Promotion” on page 6-20

# Stubbing

### In this section...

“Stubbing Overview” on page 6-2

“Manual vs. Automatic Stubbing” on page 6-2

“Deciding which Stub Functions to Provide” on page 6-3

“Stubbing Examples” on page 6-6

“Specifying Call Sequence” on page 6-8

“Constraining Data with Stubbing” on page 6-9

“Recoding Specific Functions” on page 6-12

## 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
<pre>void main(void) {     a = 1;     b = 0;     a_missing_function(&amp;a, b);     b = 1 / a;     // orange division }</pre>	<pre>void a_missing_function (int *x, int y;) { *x = y; }  void main(void) {     a = 1;     b = 0;     a_missing_function(&amp;a, b);     b = 1 / a;     // red division</pre>	<pre>void a_missing_function (int *x, int y;) { }  void main(void) {     a = 1;     b = 0;     a_missing_function(&amp;a, b);     b = 1 / a;     // green division</pre>

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 used to compute some mathematical algorithm. It is probably a Boolean status flag and if

<b>File:</b> connection_lib	<b>Function:</b> Lib_connection_create
-----------------------------	--

so is likely to be stored and compared to 0 or 1. The default stub would therefore have no detrimental effect.

<b>File:</b> connection_lib		<b>Function:</b> 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

```
void send_message(T *);
void main(void)
```

```
{
  int i;
  T x = {10, 20};

  send_message(&x);
  i = x.b /x.a; // orange with the default stubbing
}
```

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_CALIB_1] = { 1, 50, 75, 87, 95, 97, 100} ;
T Constant_1 = { 0, 0,
                MAX_Calibration_Constant,
                &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 6-14

“Data and Coding Rules” on page 6-15

“Variables: Declaration and Definition” on page 6-15

“How Can I Model Variable Values External to My Application?” on page 6-16

### 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  $x \sim [-2^{31}, 2^{31}-1]$   
y = 1 / x; // green because  $x \sim [-2^{31} - 1] \cup [1, 2^{31}-1]$ 
```

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; //  $x \sim [-2^{31}, 2^{31}-1]$ , 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 6-14 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 6-18
“Stubs of stl Functions” on page 6-18
“Stubs of libc Functions” on page 6-18

### 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

### In this section...

“Unsigned Types Promoted to Signed” on page 6-20

“Promotion Rules in Operators” on page 6-21

### 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 than 7. And this big value can never be  $\leq 7$ , 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.

# 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 7-19
- “Running Verifications from Command Line” on page 7-24

## Types of Verification

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

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

## Running Verifications on PolySpace Server

### In this section...

“Starting Server Verification” on page 7-3

“What Happens When You Run Verification” on page 7-4

“Managing Verification Jobs Using the PolySpace Queue Manager” on page 7-5

“Monitoring Progress of Server Verification” on page 7-6

“Viewing Verification Log File on Server” on page 7-9

“Stopping Server Verification Before It Completes” on page 7-11

“Removing Verification Jobs from Server Before They Run” on page 7-12

“Changing Order of Verification Jobs in Server Queue” on page 7-13

“Purging Server Queue” on page 7-13

“Changing Queue Manager Password” on page 7-15

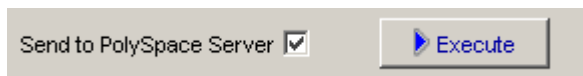
“Sharing Server Verifications Between Users” on page 7-15

### 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 **Execute** button in the middle of the Launcher window.



---

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

---

#### 4 Click **Execute**.

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

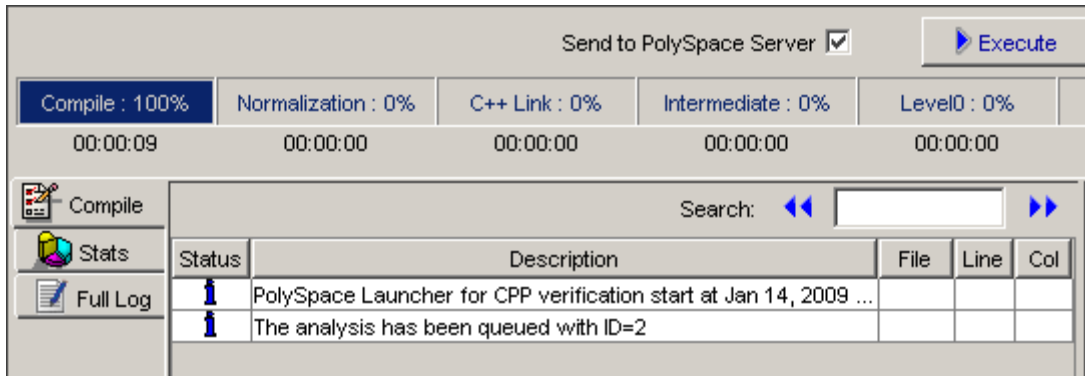
The verification has three main phases:

- 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 Client/Server for C++ User’s Guide*.
- 3 Analyzing the code for run-time errors and generating color-coded diagnostics.

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



- A message dialog box tells you that the verification completed. This message means that the part of the verification that takes place on the client is complete. The rest of the verification runs on the server.
- A message in the log area tells you that the verification was transferred to the server and gives you the identification number (Analysis ID) for the verification. For the following verification, the identification number is 1.



## Managing Verification Jobs Using the PolySpace Queue Manager

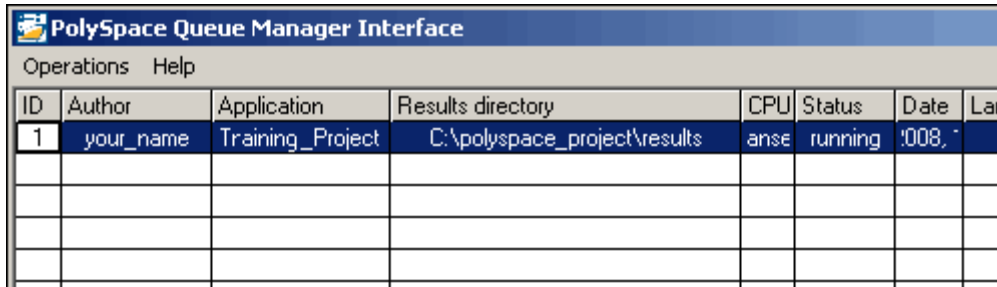
You manage all server verifications using the PolySpace Queue Manager (also called the PolySpace Spooler). The PolySpace Queue Manager allows you to move jobs within the queue, remove jobs, monitor the progress of individual verifications, and download results.

To manage verification jobs on the PolySpace Server:

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

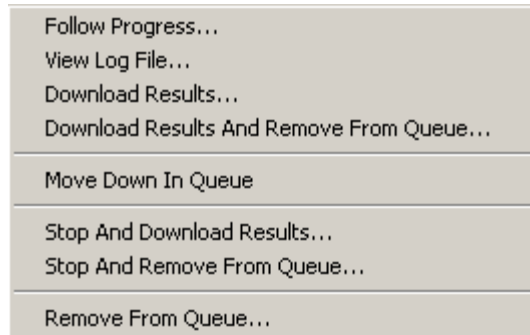


The **PolySpace Queue Manager Interface** opens.




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

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



- 3 Select the appropriate option from the context menu.

---

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

---

## Monitoring Progress of Server Verification

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

To view a log file on the server:

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

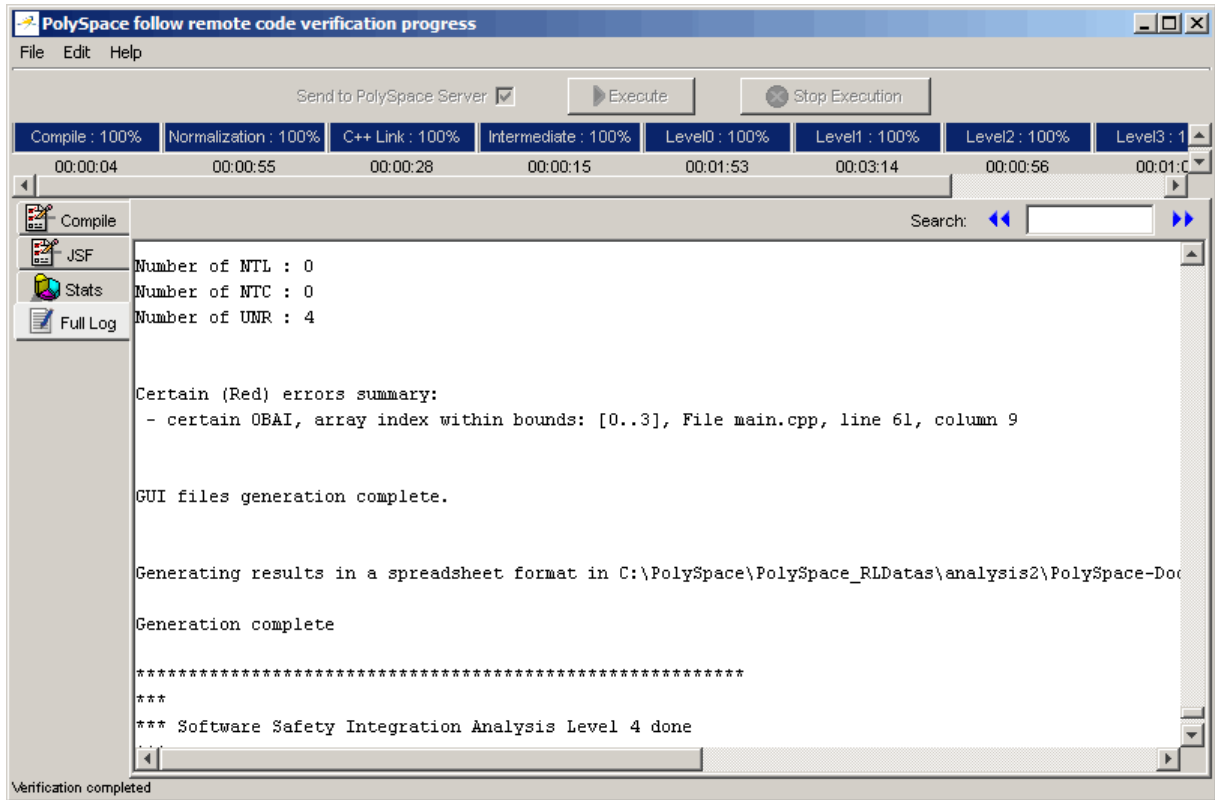


The **PolySpace Queue Manager Interface** opens.

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

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

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




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

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

- 3 Click the **Compile Log** button to display compile phase messages and errors. You can search the log by entering search terms in the **Search in**

the **log** box and clicking the left arrows to search backward or the right arrows to search forward. Click on any message in the log to get details about the message.

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

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

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

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

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

## Viewing Verification Log File on Server

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

To view a log file on the server:

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



The **PolySpace Queue Manager Interface** opens.



## Stopping Server Verification Before It Completes

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

To stop a server verification:

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



The **PolySpace Queue Manager Interface** opens.

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

- 2 Right-click the job you want to monitor, and select one of the following options:
  - **Kill and download results** — Stops the verification immediately and downloads any preliminary results. The status of the verification changes from “running” to “aborted”. The verification remains in the queue.
  - **Kill and remove from queue** — Stops the verification immediately and removes it from the queue.





## Changing Order of Verification Jobs in Server Queue

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

To move a job within the server queue:

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



The **PolySpace Queue Manager Interface** opens.

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

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

The job is moved down in the queue.

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

## 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.

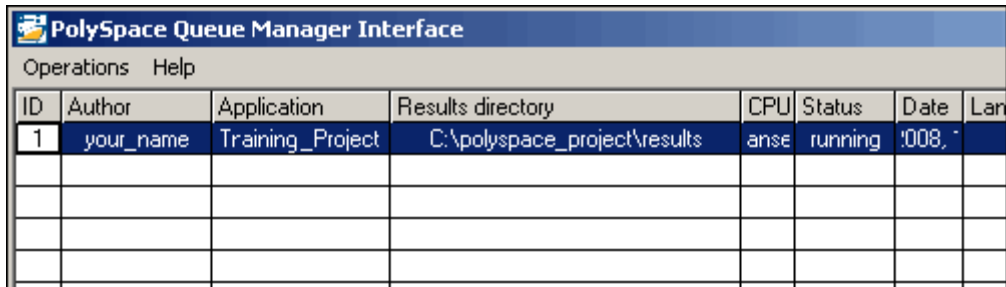
---

To purge the server queue:

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

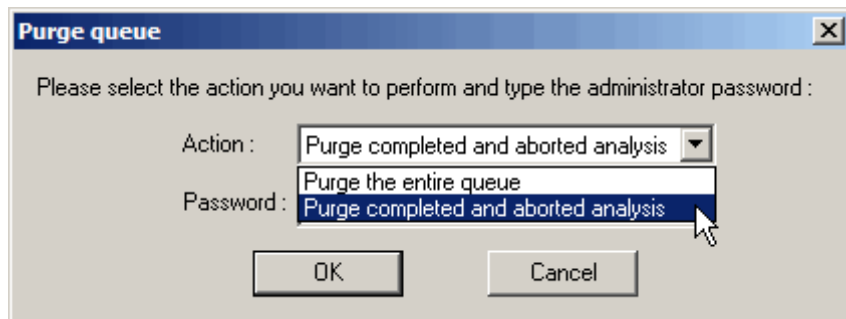


The **PolySpace Queue Manager Interface** opens.

A screenshot of the PolySpace Queue Manager Interface window. It has a title bar with the application name and a menu bar with "Operations" and "Help". Below the menu bar is a table with columns: ID, Author, Application, Results directory, CPU, Status, Date, and Language. The first row contains the following data: ID: 1, Author: your\_name, Application: Training\_Project, Results directory: C:\polyspace\_project\results, CPU: anse, Status: running, Date: '008, Language: .

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

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



- 3 Select one of the following options:

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

**4** Enter the Queue Manager Password.

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

The server queue is purged.

## Changing Queue Manager Password

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

---

**Note** The default password is administrator.

---

To set the Queue Manager password:

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

The PolySpace Queue Manager Interface opens.

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

The Change Administrator Password dialog box opens.

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

The password is changed.

## Sharing Server Verifications Between Users

### Security of Jobs in Server Queue

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

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

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

### **analysis-keys.txt File**

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

- **UNIX**<sup>®</sup> — `/home/<username>/PolySpace`
- **Windows**<sup>®</sup> — `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.

A screenshot of the PolySpace Queue Manager Interface window. The window title is "PolySpace Queue Manager Interface" and it has menu options "Operations" and "Help". Below the menu is a table with columns: ID, Author, Application, Results directory, CPU, Status, Date, and Lan. The first row contains the following data: ID: 1, Author: your\_name, Application: Training\_Project, Results directory: C:\polyspace\_project\results, CPU: anse, Status: running, Date: '008, '.

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

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

## Running Verifications on PolySpace Client

In this section...
“Starting Verification on Client” on page 7-19
“What Happens When You Run Verification” on page 7-20
“Monitoring the Progress of the Verification” on page 7-21
“Stopping Client Verification Before It Completes” on page 7-22

### 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.

---

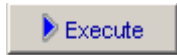
**Note** Because a verification on a client can process only a limited number of variable assignments and function calls, the source code should have no more than 800 lines of code.

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 **Execute** button.



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

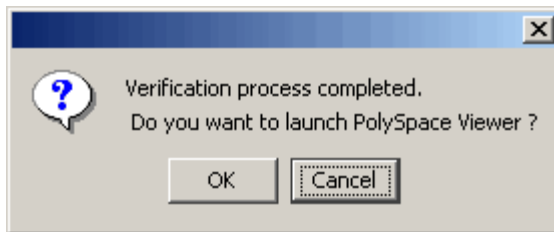
The progress bar and logs area of the Launcher window become active.

---

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

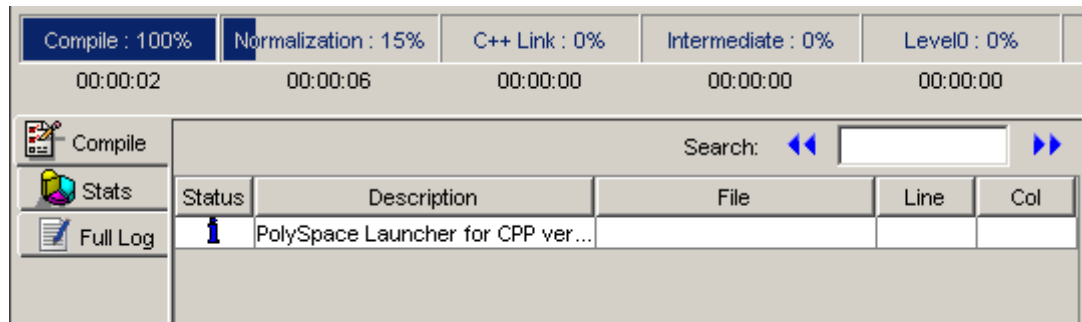


“Generate a Main Using a Given Class” in the *PolySpace Client/Server for C++ User’s Guide*.

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

## Monitoring the Progress of the Verification

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



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

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

To view the logs:

- 1 The compile log is displayed by default.

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

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

- 3 Click the refresh button  to update the stats log display as the verification progresses.
- 4 Click the **Full Log** button to display messages, errors, and statistics for all phases of the verification.

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

### Stopping Client Verification Before It Completes

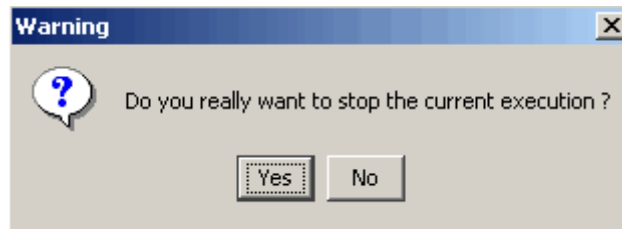
You can stop the verification before it completes. If you stop the verification, results will be incomplete, and if you start another verification, the verification starts over from the beginning.

To stop a verification:

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



A warning dialog box appears.



- 2 Click **Yes**.

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

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

---

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

---

## Running Verifications from Command Line

In this section...
“Launching Verifications in Batch” on page 7-24
“Managing Verifications in Batch” on page 7-24

### 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/` directory. 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 directory.
  - `[-f]` force download (without interactivity)
  - `-admin -p <password>` allows administrator to download results.
  - `[-server <name>[:port]]` selects a specific Queue Manager.
  - `[-v|version]` gives release number.
- `psqueue-kill <id>` — kill an identified verification.
- `psqueue-purge all|ended` — remove all completed verifications from the queue.
- `psqueue-dump` — gives the list of all verifications in the queue associated with the default Queue Manager.
- `psqueue-move-down <id>` — move down an identified verification in the Queue.
- `psqueue-remove <id>` — remove an identified verification in the queue.
- `psqueue-get-qm-server` — give the name of the default Queue Manager.
- `psqueue-progress <id>`: give progression of the currently identified and running verification.
  - `[-open-launcher]` display the log in the graphical user interface of launcher.
  - `[-full]` give full log file.
  - `psqueue-set-password <password> <new password>` — change administrator password.
- `psqueue-check-config` — check the configuration of Queue Manager.
  - `[-check-licenses]` check for licenses only.
- `psqueue-upgrade` — Allow to upgrade a client side (see the PolySpace Installation Guide in the `<PolySpace Common Dir>/Docs` directory).
  - `[-list-versions]` give the list of available release to upgrade.

- [-install-version *<version number>* [-install-dir *<directory>*]] [-silent] allow to install an upgrade in a given directory and in silent.

---

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

---

# Troubleshooting Verification Problems

---

- “Verification Process Failed Errors” on page 8-2
- “Compile Errors” on page 8-6
- “Dialect Issues” on page 8-12
- “Link Messages” on page 8-20
- “Troubleshooting Using the Preprocessed .ci Files” on page 8-25
- “Reducing Verification Time” on page 8-30

## 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 8-2
“PolySpace Software Cannot Find the Server” on page 8-3
“Limit on Assignments and Function Calls” on page 8-4

### Overview

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

### Hardware Does Not Meet Requirements

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

[www.mathworks.com/products/polyspaceclientc/requirements.html](http://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 files are missing or you did not specify the location of included files.



include.h: No such file or directory

For information on how to specify the location of include files, see “Creating New Projects” on page 4-8.

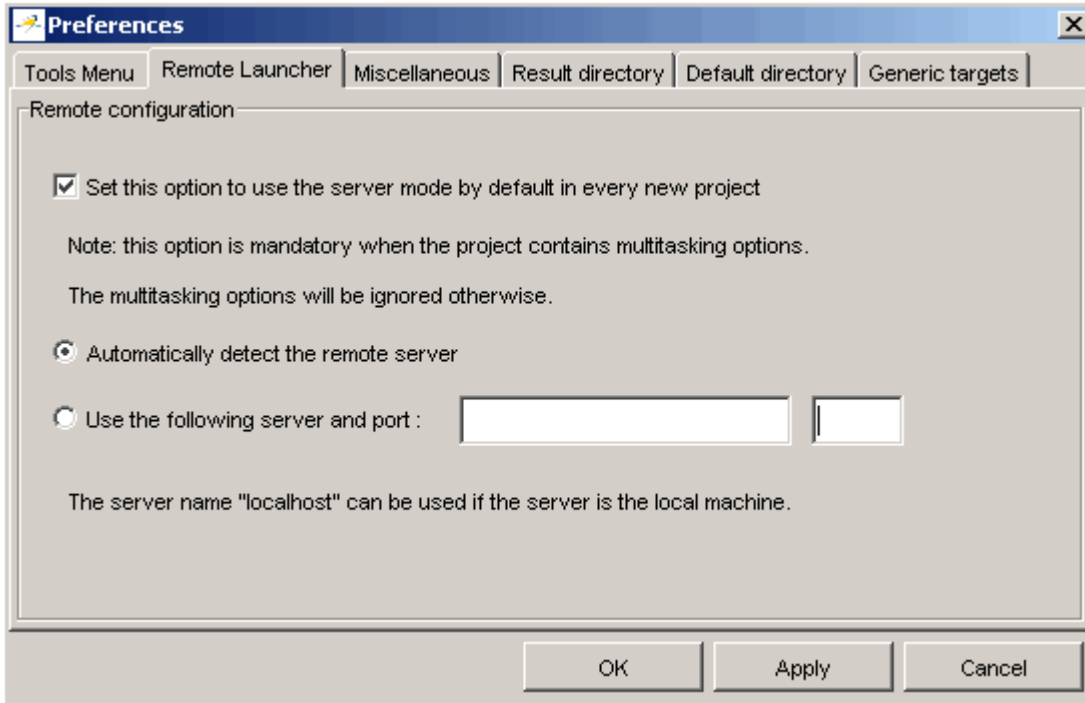
## **PolySpace Software Cannot Find the Server**

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

Error: Unknown host :

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

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



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

## Limit on Assignments and Function Calls

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

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

```
*** License error: number of assignments and function calls is
```

```
too big for -unit mode (5534 v.s 2000).  
*** Aborting.
```

PolySpace Client for 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” on page 8-6
“Examining the Compile Log” on page 8-6
“Includes” on page 8-8
“Specific Keyword or Extended Keyword” on page 8-8
“Initialization of Global Variables” on page 8-10

### Overview

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

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

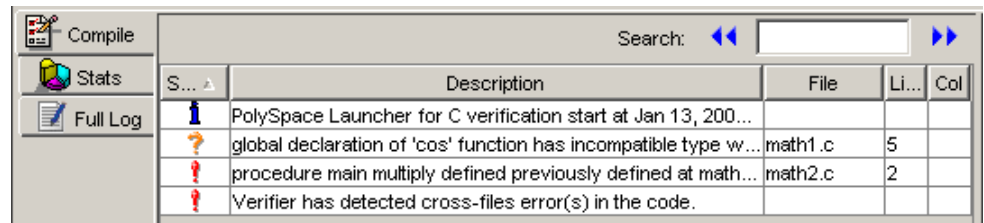
### Examining the Compile Log

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

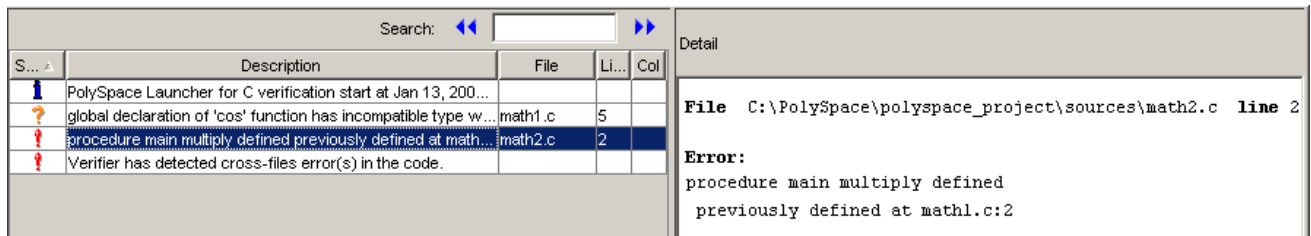
To examine errors in the Compile log:

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

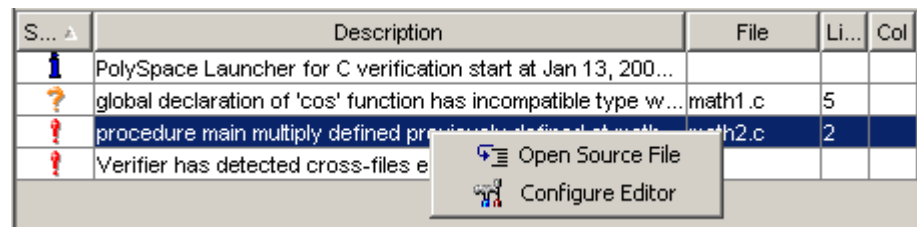
A list of compile phase messages appear in the log part of the window.



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



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



The file opens in your text editor.

**Note** You must configure a text editor before you can open source files. See “Configuring Text and XML Editors” on page 4-16.

- 4 Correct the error and run the verification again.

## 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.

## Specific Keyword or Extended Keyword

- “Specific Keyword” on page 8-8
- “Non ANSI Keywords” on page 8-9

### Specific Keyword

Compilers of specific application are defined their own keyword. A classic example is the compiler for micro controller as IAR or Keil compiler.

Original code:

keyword.h	keyword.cpp
<pre>class keyword { public:   int far m_val;   keyword (int val); };</pre>	<pre>#include keyword.h keyword::keyword(int val) {   m_val = 0;    if (val &gt; 10 )     m_val = -1; }</pre>

Error message:

```
Verifying keyword.cpp
"../sources/keyword.h", line 7: error: expected a ";"
    int far m_val;
      ^

"../sources/keyword.cpp", line 6: error: identifier "m_val" is undefined
    m_val = 0;
      ^

2 errors detected in the compilation of "CPP-ALL/SRC/MACROS/keyword.cpp".
```

You need to use the option `-D` to avoid taking these keywords into account:  
`-D far=`

## Non ANSI Keywords

You might have the same error message as for a regular compilation error, as discussed previously when using some non ANSI keyword containing for example `@` as first character. But in this case, the problem cannot be addressed by means of a compilation flag, nor a `-include` file. In this case, you need to use the post-preprocessing command.

- 1 Create a file called `ABC.txt`, and save it under `c:\PolySpace`
- 2 Open it with an ASCII editor, and copy and paste the following text:

```
#!/bin/sh
sed "s/titi/toto/g" |
sed "s/@interrupt//g"
```

- 3 In the launcher, specify the absolute path and file name in the `-post-preprocessing-command` field using browse button on a Windows system.

---

**Note** Under Linux, you must:

- enter the full path, such as `/home/poly/working_dir/ABC.txt`, and
  - make sure this file has execution permissions by typing: `chmod 755 ABC.txt`.
-

- 4** Launch a verification on the example “my\_file.cpp” below, and confirm that the compilation phase generates no errors.

```
void main(void)
{
  @
  interrupt // will be removed by the command

  int titi; // will be replaced by int toto
  int r=0; r++; toto++;
}
```

- 5** To confirm that the right transformation has been performed, open the expanded file “my\_file.ci” which is located in the directory “<results\_folder>/CPP-ALL/my\_file.ci”

## Initialization of Global Variables

When a data member of a class is declared static in the class definition, then 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.cpp
"./sources/test.h", line 33: error: data member initializer is not allowed
  static int m_number = 0;
      ^
1 error detected in the compilation of "Test.cpp".
```

Corrected code:



in file Test.h	in file Test.cpp
<pre>class Test { public: static int m_number; };</pre>	<pre>int Test::m_number = 0;</pre>

---

**Note** Some dialects, other than those supported by PolySpace Client for C/C++, accept the default initialization of static data member during the declaration.

---

## Dialect Issues

### In this section...

“ISO versus Default Dialects” on page 8-12

“CFront2 and CFront3 Dialects” on page 8-14

“Visual Dialects” on page 8-15

“GNU Dialect” on page 8-17

### ISO versus Default Dialects

The 5 common permissiveness options used by PolySpace software are described in this paragraph when using `-dialect iso`:

Original code (file `permissive.cpp`):

```
class B {} ;
class A
{
friend B ;
enum e ;
void f() { long float ff = 0.0 ;}
enum e { OK = 0, KO } ;
};
template <class T>
struct traits
{
typedef T * pointer ;
typedef T * pointer ;
} ;
template<class T>
struct C
{
typedef traits<T>::pointer pointer ;
} ;
int main()
{
C<int> c ;
}
```

- Using dialect iso, 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 will disappear if the `-dialect default` option is activated.

## 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 ({} ) which 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 will appear if the compiler is based on a Visual® dialect (including visual8).

## Import Directory

When a Visual application uses #import directives, the Visual C++ compiler generates a header file which contains some definitions. These header files have a .tlh extension and PolySpace for C/C++ requires the directory 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 0;
}
```

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” directory (usually Debug or Release). Therefore, in order to provide those files, the application needs to be built first. Then, the option `-import-dir=<build directory>` 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
<pre>#pragma pack(4)  #include "type.h"</pre>	<pre>struct A {     char c ;     int i ; } ;</pre>	<pre>#pragma pack(2)  #include "type.h"</pre>

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:

```
void f(int n) { char tmp[n] ; ... }
```

- Anonymous structures:

```
union A { struct { double x ; double y ; double z } ;  
double tab[3] ; } a ; 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`, `__builtin_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 %%a1,$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

In this section...
“STL Library C++ Stubbing Errors” on page 8-20
“Lib C Stubbing Errors” on page 8-21

### STL Library C++ Stubbing Errors

PolySpace software provides an efficient implementation of all functions in the Standard Template Library. The Standard Template Library (STL) and platforms may have different declarations and definitions, otherwise the error messages below appears.

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 option *-no-stub-stl* avoid this error message. Then, you need to add the directory containing definitions of own STL library as a directory to include using option *-I*.

The preceding message can also appear with the directory 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, as 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>/CPP-ALL/CPP-STUBS/__polyspace_stdstubs.c", line 2996: error:
declaration of function "memcpy" is incompatible with a declaration in another
translation unit (parameters do not match)
    the other declaration is at line 4 of "/sources/Copy.cpp"
extern void * __pst_profile_memcpy (void *s1, const void *s2, size_t n) ;
extern "C" void * memcpy (void *s1, const void *s2, size_t n)
    ^

detected during compilation of secondary translation unit "CPP-ALL/
SRC/MACROS/__polyspace_stdstubs.c"
```

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 permissive option `-no-extern-C`. It will remove all declaration `extern "C"`

### **Standard Stubs**

It could also happen that the compiler (used) does not provide exact ANSI prototypes for a given C function of the standard libC library.

Original code:

```
#include <signal.h>
extern "C" {
    extern void (*signal (int, void (*)(int)))(int);
}
class Copy
{
public:
    Copy() {};

};
```

Error message:

```
Pre-linking C++ sources ...
c:\results\CPP-ALL\CPP-STUBS\__polyspace__stdstubs.c:891: error:
a value of type "void (*) (...) C" cannot be used to initialize
an entity of type "_polyspace_signal_function_type"
    _polyspace_signal_function_type res = (void (*)(...))(-1);
    ^
c:\results\CPP-ALL\CPP-STUBS\__polyspace__stdstubs.c:922: error:
a value of type "void (*) (...) C" cannot be assigned to an entity
of type "_polyspace_signal_function_type"
    res = (void (*)(...))1;
    ^
```

In the previous example and associated error message, a problem occurs in the `__polyspace__stdstubs.c` file. At line 891 of this file, located in `<results directory>/CPP-ALL/SRC`, the prototype of `signal` function does not match the one given in the original code. In this example, the code to analyze does not follow the Standard ANSI function prototype on function `signal`.

It is possible to use compiler prototypes by deactivating standard prototype provided by ANSI. To do so, you have to add the flag `POLYSPACE_NO_STANDARD_STUBS` to the verification using `-D` option: `-D POLYSPACE_NO_STANDARD_STUBS`. All functions declared in `assert.h`, `ctype.h`, `errno.h`, `locale.h`, `math.h`, `setjmp.h`, `signal.h`, `stdio.h`, `stdarg.h`, `stdlib.h`, `string.h` and `time.h` will be taken into account.

## Functional Limitations on Some of 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, for which PolySpace uses built-in code, do not set `errno` but instead generate a red error when a range or domain error occurs (see examples with **NTC** checks).

You can also use the compile option `POLYSPACE_STRICT_ANSI_STANDARD_STUBS` (-D flag) which will only deactivate extensions to ANSI C standard `libc`. 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 `siglongjmp` are concerned.

## Troubleshooting Using the Preprocessed .ci Files

In this section...
“Overview” on page 8-25
“Example of <i>ci</i> File” on page 8-25
“Troubleshooting Methodology” on page 8-27

### Overview

In the preceding paragraphs, common types of compile or linking errors messages have been detailed. They are associated to C++ dialects, or specific options used by the dialect (for instance 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 directory>/CPP-ALL/SRC/MACROS` or `<results directory>/ALL/SRC/MACROS`. They have a `.ci` extension and they will 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
<pre> #include "Extension.h"  Extension::Extension(int val) {     m_val = 0;     ABS(val);      if (val &gt; MAX_VALUE )         m_val = -1; }  #ifdef _DEBUG void Extension::message(char*) {} #else void print(char*) {} #endif </pre>	<pre> #define MAX_VALUE 10 #define ABS(x) ((x)&lt;0?(x):- (x))  class Extension { public:     int m_val;     Extension(int val);  #ifdef _DEBUG     void message(char*); #else     void print(char*); #endif }; </pre>

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 to 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 Check whether 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
<pre>#define DEBUG  #include "Test.h"  class Child1 : public Test { public:     Child1();     Child1(int val);      void search(int val); };</pre>	<pre>#undef DEBUG  #include "Test.h"  class Child2 : public Test { public:     Child2();     Child2(int val);      void qshort(int val);  protected:     int m_status; };</pre>	<pre>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; };</pre>

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...

“How Far has the Verification Progressed? How Can I Predict the Duration?” on page 8-30

“An Ideal Application Size” on page 8-32

“Why Should there be an Optimum Size?” on page 8-33

“Switch the Antivirus Off” on page 8-34

“Tuning PolySpace Parameters” on page 8-34

“Selecting a Subset of Code” on page 8-35

“A Decision Algorithm to Speed-Up a Verification: Hints and Troubleshooting” on page 8-40

“What are the Benefits of these Methods?” on page 8-45

### How Far has the Verification Progressed? How Can I Predict the Duration?

The duration of a verification is impacted by:

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

The fact that so many factors are involved make it impossible to derive a precise formula to calculate verification duration. Instead, PolySpace software provides textual output to illustrate how much progress has been made (available under Linux and Windows). This progress text is located in the “product\_installation\_dir”/tools/ and is called polyspace-stats.

**Example**

```
/cygdrive/C/PolySpace/2.4/Verifier/tools/polyspace-stats
my_log_file.txt
```

```

/cygdrive/c/PolySpace_Results
$ /cygdrive/C/PolySpace/2.4/Verifier/tools/polyspace-stats PolySpace_2_4_1_21_N
ew_Project_03_25_2004-11h50.log
PolySpace Verifier 2_4_1_21 :

Username      : Marc
Hostname      : laptop
Results directory : /cygdrive/c/PolySpace_Results

Number of files      :      1
Number of lines      :     27
Number of lines without comments :    20

The completed passes are the following :
- C sources verification : 0:00:40
- C to intermediate language translation : 0:00:16
- IL compilation : 0:00:20
- Control and Data Flow Analysis [1/3] : 0:00:07
- Control and Data Flow Analysis [2/3] : 0:00:01
- Control and Data Flow Analysis [3/3] : 0:00:02
- Control and Data Flow Analysis : 0:00:50

Currently in Level 1 Software Safety Analysis :
4 .atz files out of a total of 4 were analysed for this pass : 0:00:46

Please refer to file:/cygdrive/c/PolySpace_Results/PolySpace_2_4_1_21_New_Projec
t_03_25_2004-11h50.log for further information.

$

```

Consider the area displaying:

Currently in Level 1 Software Safety Analysis

4 .atz files out of 4 were analysed for this pass: 00:00:46

It can be deduced that

- The proportion of files analyzed for this integration level (4/4)
- The elapsed time : 46 seconds

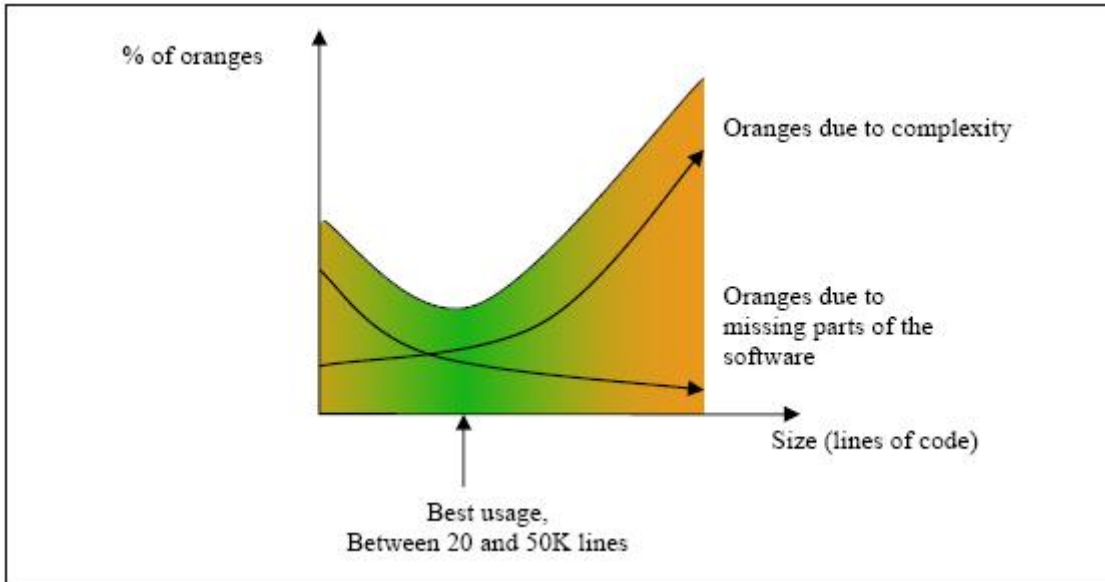
The remaining verification duration can be deduced by extrapolating from this data by considering the number of files and passes still to be completed.

### **An Ideal Application Size**

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 made by PolySpace software. 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. However, they also mean that the benefits derived from verifying the whole of a large application have to be balanced against the loss of precision which results.

**This is why it is recommended to 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 twenty (for C++) and fifty thousand lines of code (for C and Ada). For such applications, approximations should not be too significant. Take care that some times verification time should **not be reasonable**.

Experience suggests that subdividing an application prior to verification will normally have a **beneficial impact on selectivity** — that is, more **red**, **green** and **gray** checks, fewer **orange** unproven and therefore more efficient bug detection.



### A compromise between selectivity and size

## Why Should there be an Optimum Size?

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

- Makes broader approximations, producing more oranges
- Can take much more time to analyze the application.

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

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

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

- Stop the use of PolySpace verification at this stage (a lot of the benefits have been achieved already), or
- Analyze subsets of the code.

### **Switch the Antivirus Off**

Disabling or switching off any third party antivirus software for the duration of a verification can reduce the verification time by up to forty percent.

### **Tuning PolySpace Parameters**

here is a compromise to be made to balance the time required to perform a verification, and the time required to review the results. Launching PolySpace verification with the following options will allow the time taken for verification to be reduced but will compromise the precision of the results which will therefore take longer to review. It is suggested that the parameters should be used in the sequence shown - that is, if the first suggestion does not increase the speed of verification sufficiently then introduce the second, and so on.

- Switch from -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 big arrays are used, set the `-no-fold` option.

For example, an appropriate launching command might be

```
polyspace-c -O0 -respect-types-in-globals -k-limiting 0
```



## Selecting a Subset of Code

If a project is subdivided for verification purposes, then the total verification time will be considerably shorter for the sum of the parts than for the whole project considered in one pass. A logical way to set about splitting the project in this way is to consider data flow.

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

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

Consider the examples below.

### Example 1

```
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" may be formatted data, due to a very complex manipulation of x.

Since x is volatile, a probable consequence will be that y will contain all possible formatted data. An approximation could be to remove the procedure `complete_treatment_based_on_x` completely, and let automatic stubbing work. "y" will then be considered 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!
}
```

### **Consequences of Subdividing Code**

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

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

- **Error management modules.** These modules often contain a big array of structures that are accessed through an API, but return only a Boolean value. By removing the API code and retaining the prototype,

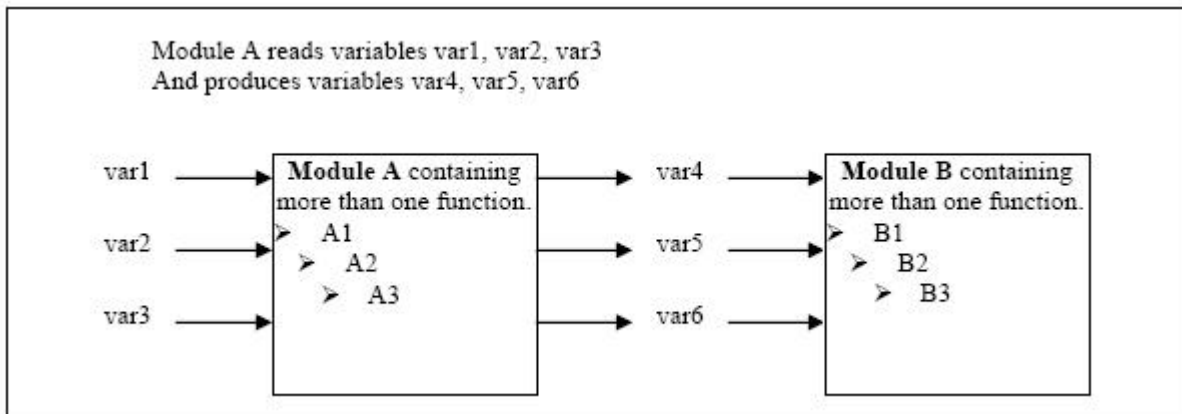
the automatically generated stub will be assumed to return a value in the range  $[-2^{31}, 2^{31}-1]$ , which includes 1 and 0. The procedure will be considered to return all possible answers, just like reality;

- **Buffer management for mailboxes coming from missing code.**

Suppose an application reads a huge buffer of 1024 char, and then uses it to populate 3 small arrays of data, using a very complicated algorithm before passing it to the main module. If the buffer is excluded from the verification and the arrays are initialized with random values instead, then the verification of the remaining code will just be the same.

### Subdivision According to Data-Flow

Consider the following example.



In this application, variables 1, 2 and 3 can vary between the following ranges:

Var1	Between 0 and 10
Var2	Between 1 and 100
Var3	Between -10 and 10

#### Specification of Module A:

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

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

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

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

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

#### The consequences:

- (-) A slight loss of precision on the B module verification, because now all combinations for variables 4, 5 and 6 are considered:
  - It includes all of the possible combinations.
  - It also includes those that would have been restricted by the A module verification.

For example, If the B module included the test

“If var2 is equal to 0, than  $\text{var4} > \text{var5} > 5$ ”

then the dead code on any subsequent “else” clause would not be detected.

- (+) An in depth investigation of the code is not necessary to isolate a meaningful subset. It means that a logical split is possible for any application, in accordance with the logic of the data
- (+) The results remain valid (because there no need to remove (say) a thread that will change shared data)
- (+) The complexity of the code is reduced by a significant factor
- (+) The maximum precision level can be retained.

#### **Typical examples of removable components:**

- Error management modules. A function `has_an_error_already_occurred` might return TRUE or FALSE. Such a module may contain a big array of structures which are accessed through an API. The removal of the API code with the retention of the prototype will result in the PolySpace verification producing a stub which returns  $[-2^{31}, 2^{31}-1]$ . This clearly includes 1 and 0 (yes and no). The procedure `has_an_error_already_occurred` will therefore return all possible answers, just like the code would at execution time.
- Buffer management for mailboxes coming from missing code. Suppose a large buffer of 1024 char is read, and the data is then collated into 3 small arrays of data using a very complicated algorithm. This data is then given to a main module for treatment. For the verification, the buffer can be removed and the 3 arrays initialized with random values.
- Display modules.

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

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

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

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

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

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

- x could be declared as volatile in order to take into account all possible executions. Otherwise x will take only its initial value or x variable will remain constant, and T2s verification will be a subset of possible execution paths. You might have precise results, but it will only include one *scenario* among all possible states for the variable x.
- x could be initialized to the whole possible range [10;15], and then the T2entry-point called. This is accurate if x is calibration data.

### **Subdivide According to Files**

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

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

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

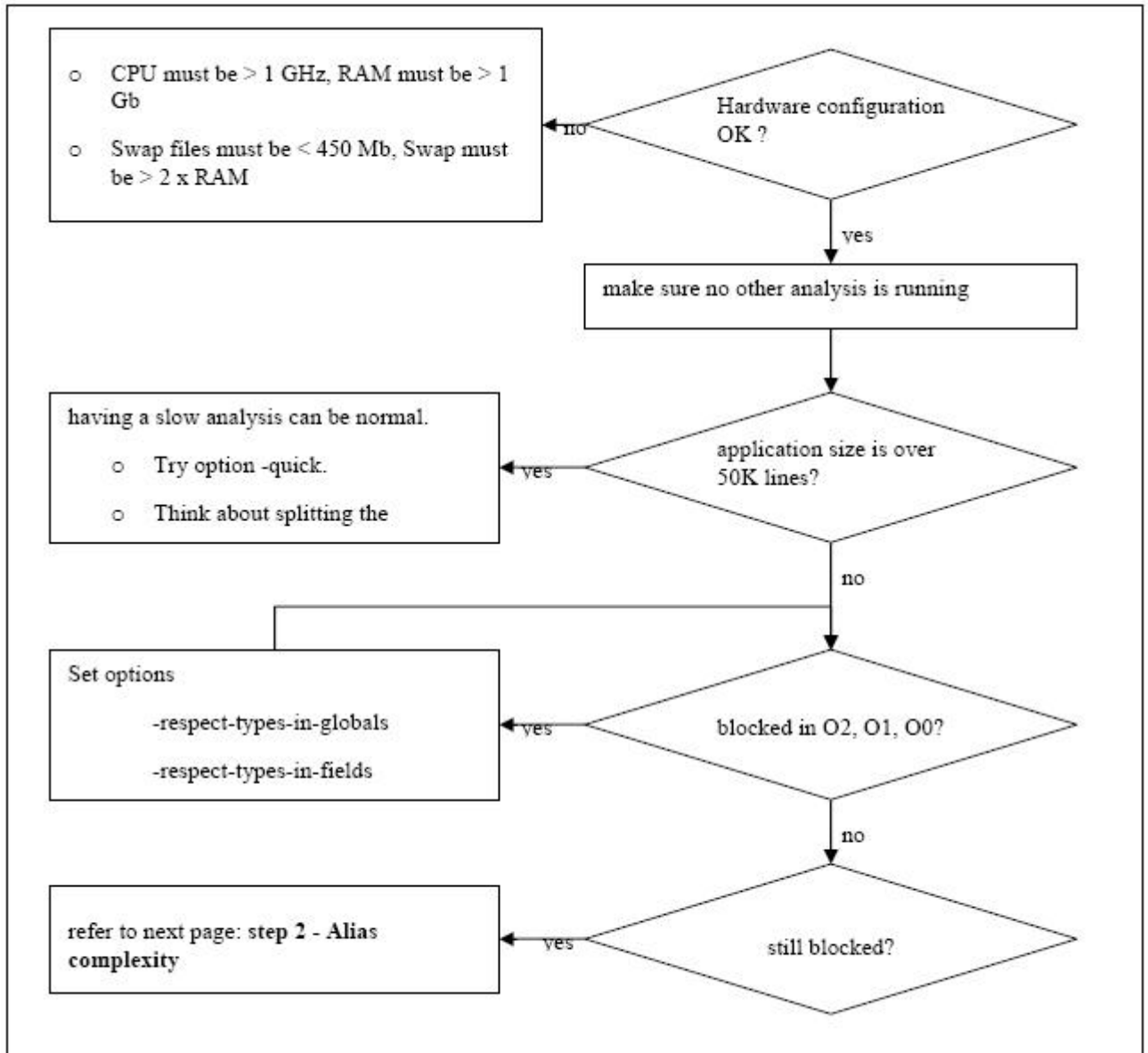
### **A Decision Algorithm to Speed-Up a Verification: Hints and Troubleshooting**

This chapter suggests methods to reduce the duration of a particular verification, while minimizing the need to compromise the launch parameters or the precision of the results.

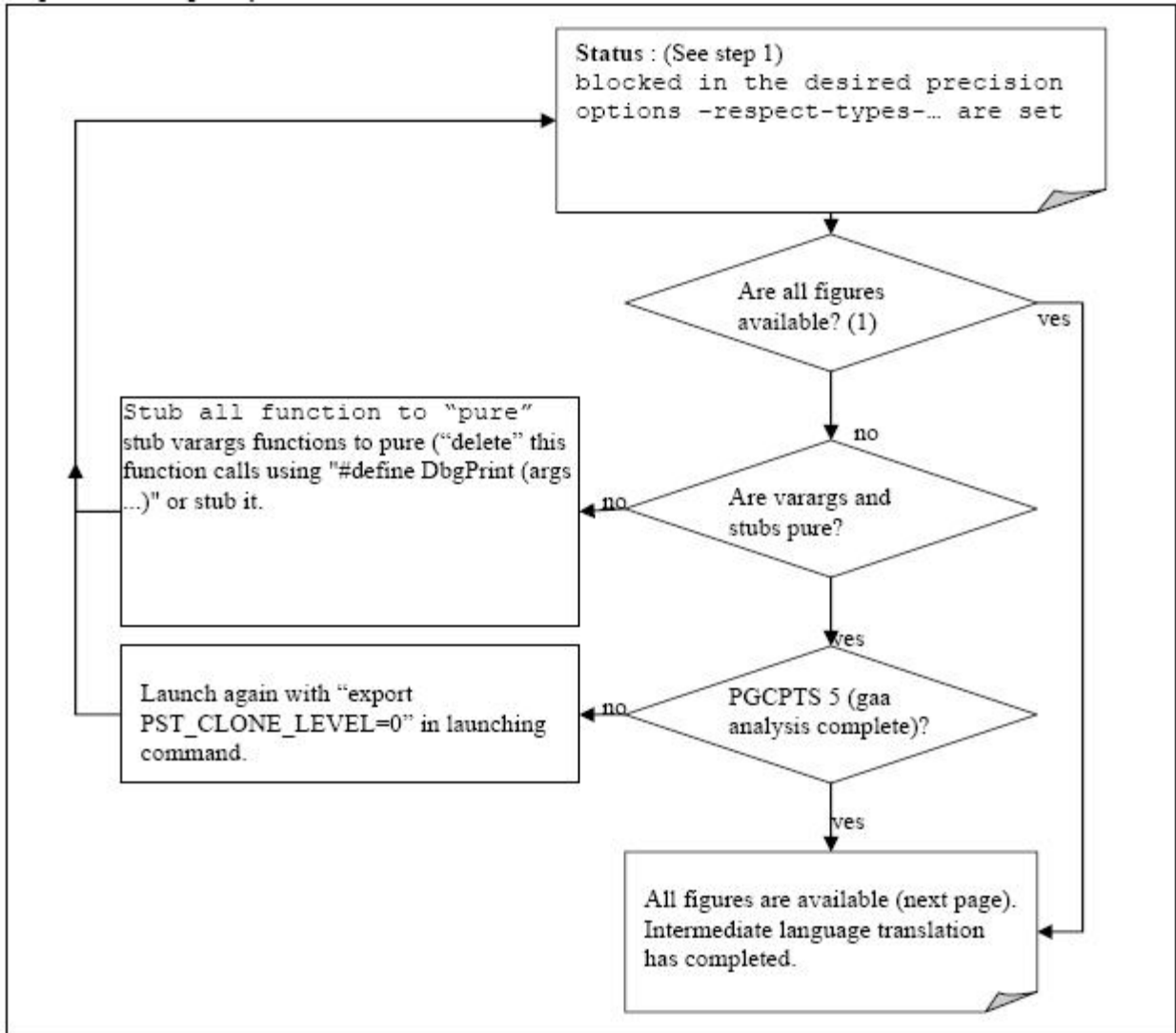
The size of a code sample which can be effectively analyzed can be increased by tuning the tool be optimized for that sample. Beyond that point, subdividing the code or choosing a lower precision level will bring better results (-O1, -O0).

Suppose that for a given set of code, the intermediate language translation does not finish.

#### **Step 1: standard scaling options**



### Step 2: alias complexity



A typical set of statistics is shown below. They are be found for any application by using the “polyspace-stats -v” command, at any point after the intermediate language translation has been completed.

Some stats on aliases use:



```
Number of alias writes: 2672
Number of must-alias writes: 0
Number of alias reads: 0
Number of invisibles: 60
Number of global invisibles: 3808
Stats about alias writes:
  biggest sets of alias writes: Variable_1 (45), Variable_1 (32)
  procedures that write the biggest sets of aliases: procedure_f_1
(583), procedure_f_2 (369), procedure_f_3 (264)
```

You can reduce the pointers complexity by inlining the following functions :

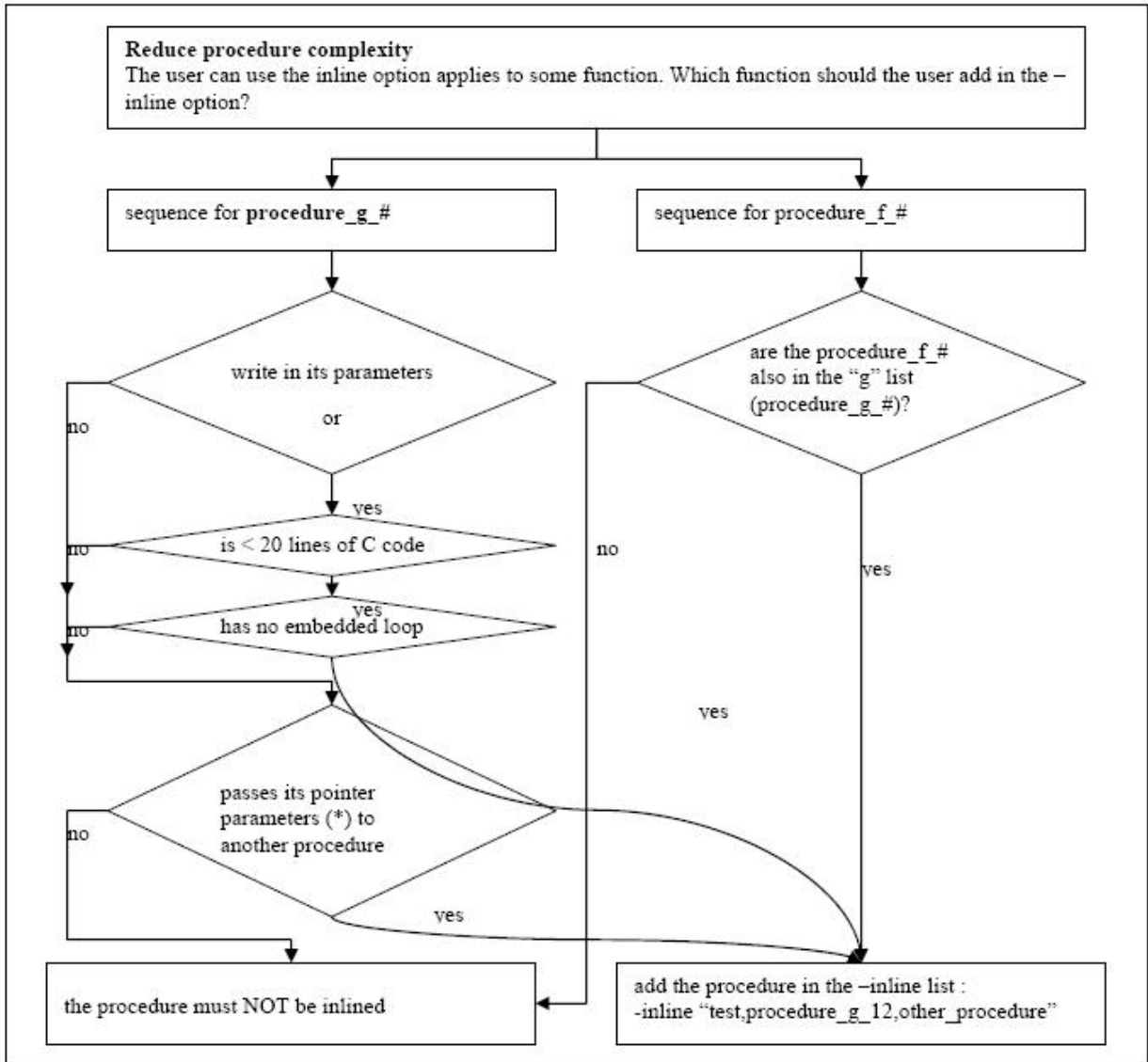
```
procedure_g_1      procedure_g_2
procedure_g_3
```

From this point, there are three possible routes to take. In order of preference, they are

- Reduce procedure complexity
- Reduce task complexity
- Reduce variable complexity

and then restart the verification.

### **Reduce procedure complexity**



For example, does it pass its pointer parameters to another procedure?

YES	NO	NO
<pre>void f(int *p) {   f2(p) }</pre>	<pre>void f(int q)</pre>	<pre>void f(int *r) {   *r = 12 }</pre>

### Reduce task complexity

If 2 or more tasks are present, and particularly if there are more than 10000 alias reads:

Set the `-lightweight-thread-model` option, which will

- Reduce task complexity, and
- Reduce verification time

There are some downsides:

- It causes more oranges and a slight loss of precision on reads of shared variables through pointers
- The dictionary may omit some read/write accesses.

### Reduce variable complexity

If the types are complex	Set the <code>-k-limiting [0-2]</code> option. Begin with 0. Go up to 1, or 2 in order to gain precision
If there are large arrays	Setting the <code>-no-fold</code> option can solve the problem.

## What are the Benefits of these Methods?

It may be desirable to split the code

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

The problems subdivision may bring are that

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

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

When the code consists of a small subset of a larger project, a lot of procedures will be automatically stubbed. This is done according to the specification or prototype of the missing functions, and therefore PolySpace verification assumes that all possible values for the parameter type can be returned.

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

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

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

### **Considering the Effects of Application Code Size**

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

For instance, in a relatively small application, PolySpace 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  $[-2 ; 2] \cup \{10\} \cup [15 ; 17] \cup \{25\}$ . Here, the same division appears as an orange check.

If the complexity of the internal data becomes even greater later in the verification, PolySpace 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 PolySpace will adjust the level of simplification, for example:

- -O0 — shorter computation time,
  - -O2 — less orange warnings.
  - -O3 — less orange warnings and bigger computation time.
-



# Reviewing Verification Results

---

- “Before You Review PolySpace Results” on page 9-2
- “Opening Verification Results” on page 9-8
- “Reviewing Results in Assistant Mode” on page 9-19
- “Reviewing Results in Expert Mode” on page 9-26
- “Generating Reports of Verification Results” on page 9-37
- “Using PolySpace Results” on page 9-41

## Before You Review PolySpace Results

### In this section...

“Overview: Understanding PolySpace Results” on page 9-2

“Why Gray Follows Red and Green Follows Orange” on page 9-3

“What is the Message and What does it Mean?” on page 9-4

“What is the C++ Explanation?” on page 9-5

### Overview: Understanding PolySpace Results

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

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

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

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



## Why Gray Follows Red and Green Follows Orange

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

In the example below, consider why:

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

```

void red(void)                extern int Read_An_Input(void);
{
int x;                        void propagate(void)
x = 1 / x ;                   {
x = x + 1;                    int X;
}                               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  $X$ ,  $X$  has not been initialized. Therefore the corresponding check (Non Initialized Variable) on  $X$  is red.
- As a result, all possible execution paths are stopped, because they all produce an RTE. 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, an “out of bounds” error is possible since  $X$  can equal  $-3$  as well as  $3$ .
- Subsequently, all conditions leading to an RTE are assumed to have been truncated — they are no longer considered in the verification. Therefore, on the following line, all executions in which  $X = [-2^{31}, -1]$  and  $[100, 2^{31}-1]$  are stopped.

- Consequently, 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.

---

**Note** When writing manual stubs, you can use this property of PolySpace software to restrict data input values. For more information on how to assign ranges of variables, see “Reduce the cloud of points” on page 10-11.

---

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

PolySpace software numbers checks in the same order it followed during execution of the code.

Consider the instruction:

```
x++;
```

PolySpace first checks for a potential NIV (Non Initialized Variable) for  $x$ , and then checks the potential OVFL (overflow). This mimics the actual execution sequence.

Understanding these sequences can help you understand the message presented by PolySpace, and what that message implies.

Consider an orange NIV on  $x$  in the test:

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

You might conclude that the verification has not kept 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

Using the viewer, you can see the category of each check by clicking on it. When you examine an orange check, any value of a variable that would result in a runtime 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  $x$  has been 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 doesn't know anything about  $x$ .
- if " $x > 100$ " **DOES** means that PolySpace doesn't know whether  $X$  has been initialized.

The two rules to remember when reviewing results are:

- Focus on the message given by PolySpace software.
- Do not jump to conclusions.

## What is the C++ Explanation?

Verification results depend entirely on the code that was verified. When interpreting the results, do not consider:

- Any physical action from the environment in which the code is intended to operate.

- Any configuration that is not part of the verification.
- Any reason other than the code itself.

Remember, the only thing the verification considers is the C code submitted to it.

Consider the example below, 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 runtime error, meaning that subsequent lines will be dealing with only  $x = [0, 99]$
- The orange check on the division also truncates all execution paths that lead to a runtime error, so all instances where  $x = 0$  are also stopped.

Therefore, for the code execution path after the orange division sign,  $x = [1; 99]$ .

- Thus,  $x$  is never equal to 0 **at this line**. Therefore, the array access is green ( $y(x - 1)$ ).

---

**Note** For the array access at the previous line ( $y[x]$ ), we have  $X \sim [-2^{31}, 2^{31}-1]$  – 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

In this section...
“Downloading Results from Server to Client” on page 9-8
“Opening Verification Results” on page 9-11
“Exploring the Viewer Window” on page 9-11
“Selecting Viewer Mode” on page 9-15
“Setting Character Encoding Preferences” on page 9-15

### Downloading Results from Server to Client

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

---

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

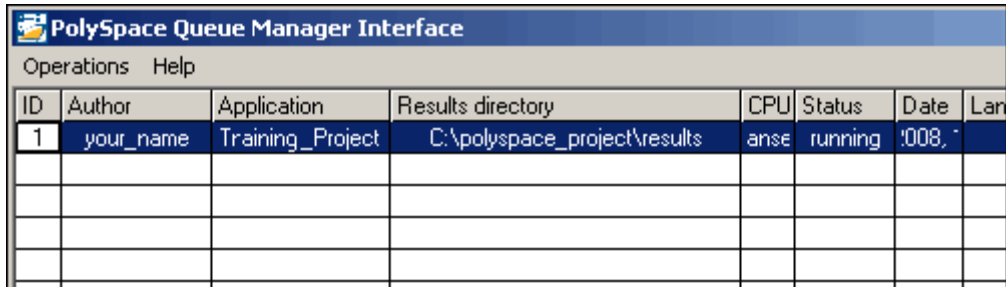
---

To download verification results to your client system:

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



The **PolySpace Queue Manager Interface** opens.



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

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

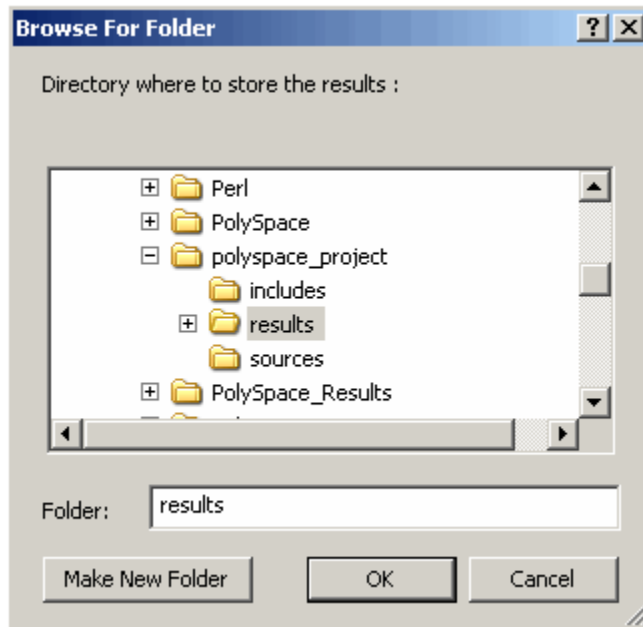
- 2 Right-click the job you want to view, then select **Download Results** from the context menu.

---

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

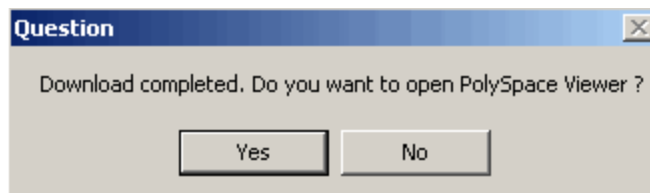
---

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



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

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



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

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



## Opening Verification Results

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

---

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

---

To open the verification results:

- 1 Double-click the PolySpace Viewer icon:



- 2 Select **File > Open**.
- 3 In the **Please select a file dialog box**, select the results file you want to view.
- 4 Click the **Open** button.

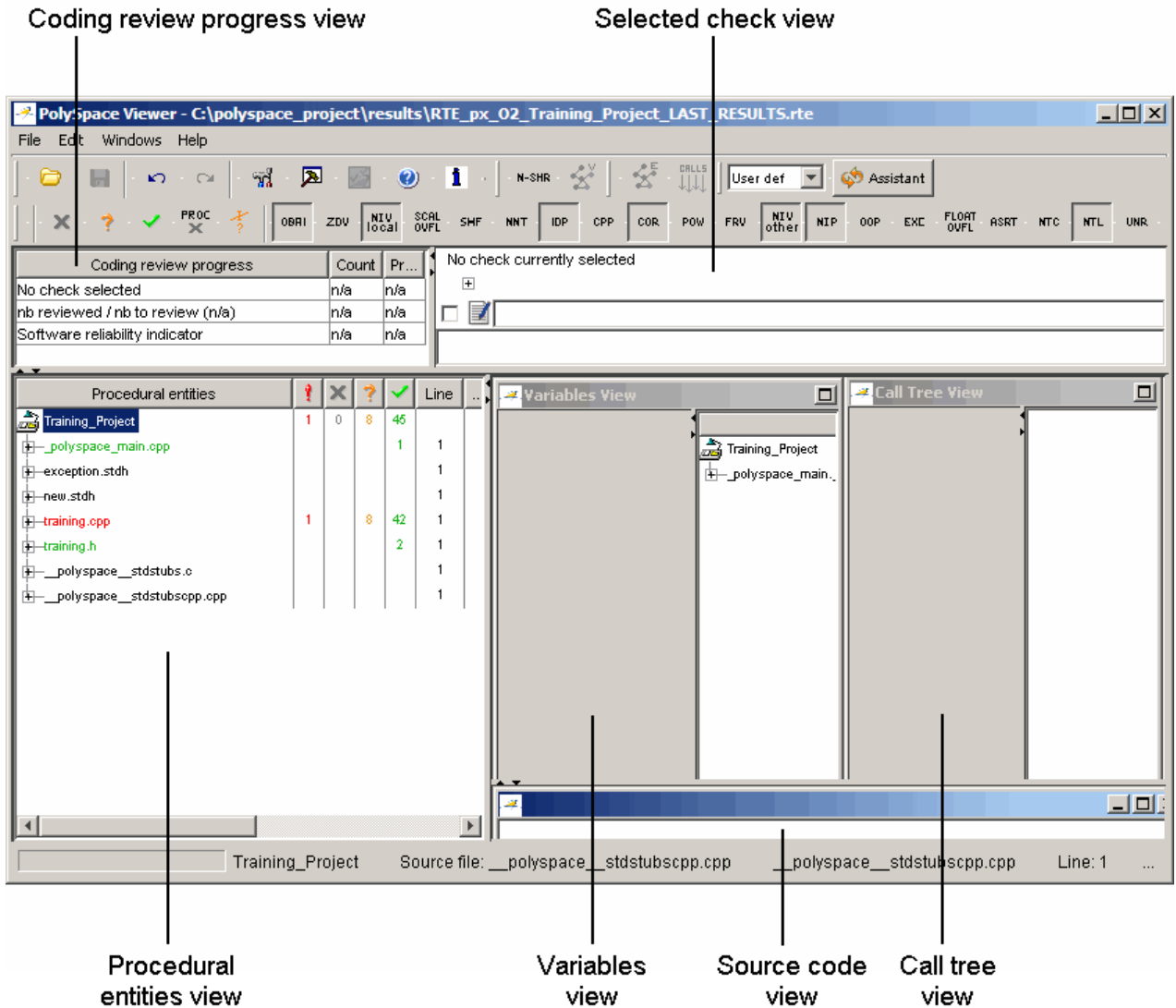
The results appear in the Viewer window.

## Exploring the Viewer Window

- “Overview” on page 9-11
- “Procedural Entities View” on page 9-13

### Overview

The PolySpace Viewer looks like:



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



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

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

You can resize or hide any of these sections.

## Procedural Entities View

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

Procedural entities						Line	...	%
Training_Project		1	0	8	45			85
+	_polyspace_main.cpp				1	1		100
+	-exception.stdh					1		0
+	-new.stdh					1		0
+	-training.cpp	1		8	42	1		84
+	-training.h				2	1		100
+	__polyspace__stdstubs.c					1		0
+	__polyspace__stdstubs.cpp					1		0

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

Column Heading	Indicates
	Number of red checks (for operations where an error always occurs)
	Number of gray checks (for unreachable code)
	Number of orange checks (warnings for operations where an error might occur)
	Number of green checks (for operations where an error never occurs)
	Total number of red, green, and gray checks (an indication of the level of proof)

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

---

**Note** You can select which columns appear in the procedural entities view by editing the preferences. To learn how to add a **Reviewed** column, see “Making the Reviewed Column Visible” on page 9-30.

---

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

## Selecting Viewer Mode

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

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

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



## Setting Character Encoding Preferences

If the source files you want to verify were created on an operating system that uses different character encoding than your current system (for example, when viewing files containing Japanese characters), you will receive an error message when you view the source file or run certain macros.

The **Character encoding** option allows you to view source files created on an operating system that uses different character encoding than your current system.

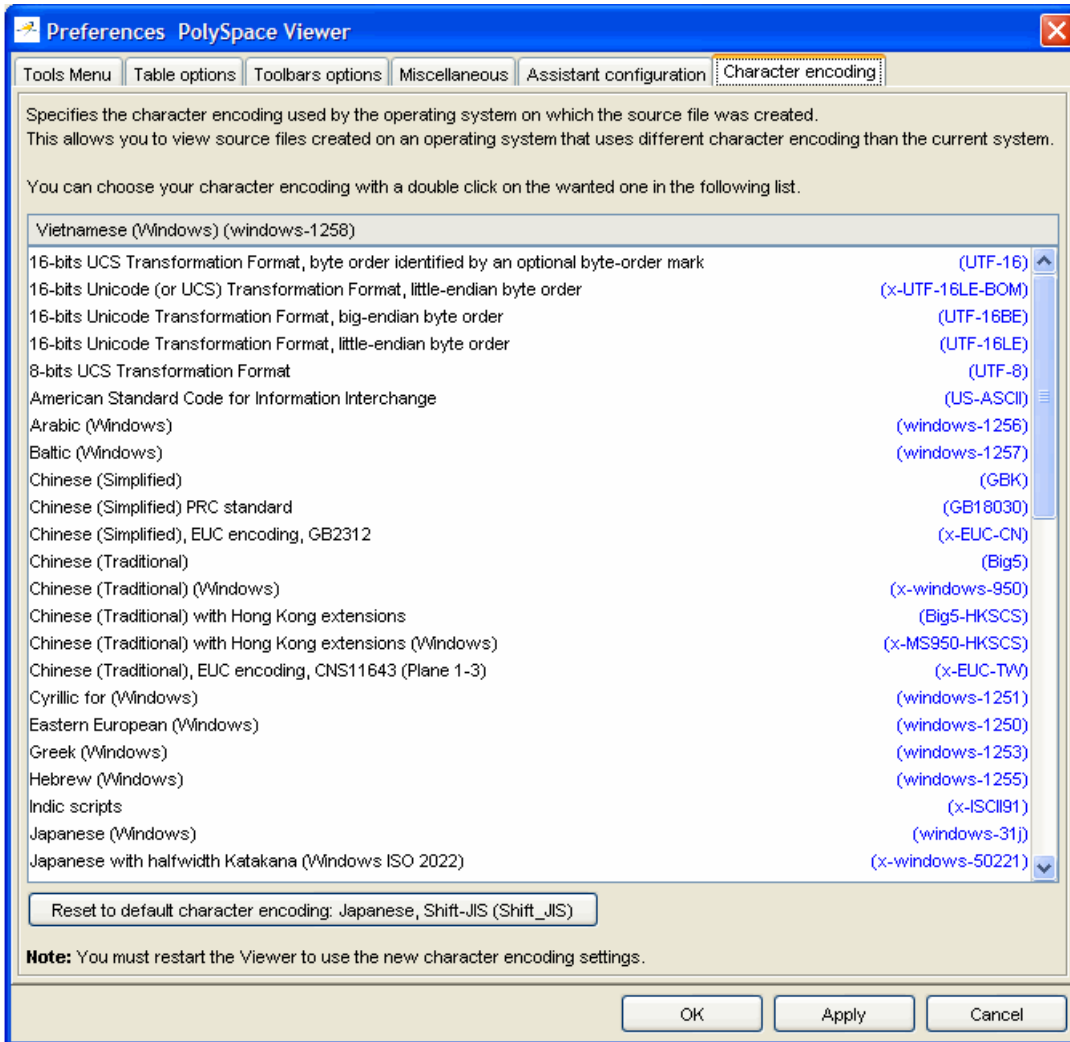
To set the character encoding for a source file:

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

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

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

The Character encoding tab appears.



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

---

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

---

**5** Close and restart the Viewer.



## Reviewing Results in Assistant Mode

### In this section...

- “What Is Assistant Mode?” on page 9-19
- “Switching to Assistant Mode” on page 9-19
- “Selecting the Methodology and Criterion Level” on page 9-20
- “Exploring Methodology for C++” on page 9-21
- “Defining a Custom Methodology” on page 9-23
- “Reviewing Checks” on page 9-24

### What Is Assistant Mode?


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

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

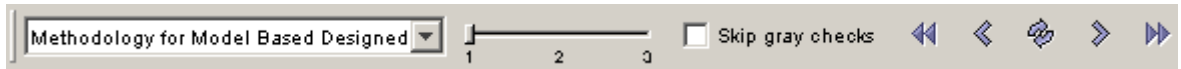
For more information about methodologies and criterion levels, see “Selecting the Methodology and Criterion Level” on page 9-20.

### Switching to Assistant Mode

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

- Click the Viewer mode button . The button features a small orange icon of a person with arms raised and the word "Assistant" to its right.

The Viewer window toolbar displays controls specific to assistant mode.



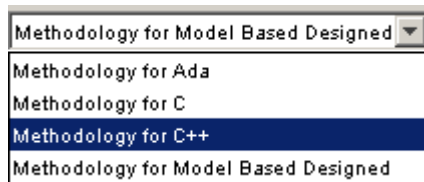
The controls for assistant mode include:

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

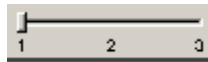
## Selecting the Methodology and Criterion Level

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

- 1 Select **Methodology for C++** from the methodology menu.



- 2 Select the appropriate level on the level slider.



For the configuration Methodology for C++, the three levels are:

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

These three levels correspond to phases of the development process.

## Exploring Methodology for 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 appears.

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

The configuration for Methodology for C++ appears.

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

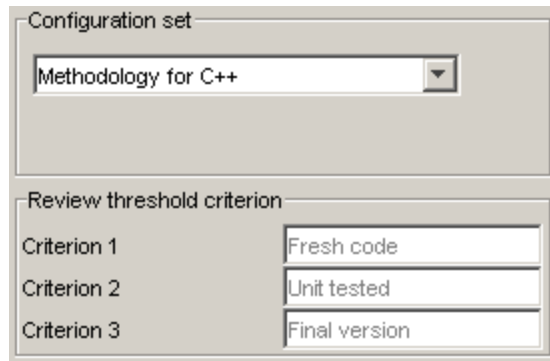
aneous Assistant configuration

Number of checks to review

	Criterion 1	Criterion 2	Criterion 3
Common			
ZDV	5	20	ALL
NIVL	10	50	ALL
S-OVFL	10	50	ALL
COR		10	10
POW	5	10	ALL
NIV		0	10
F-OVFL	5	10	20
ASRT		5	20
C & C++ only			
OBAI	10	20	ALL
SHF	5	10	ALL
IDP		10	20
NIP		10	20

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).



For the configuration Methodology for C++, the criterion names are:

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

These names correspond to phases of the development process.

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

## Defining a Custom Methodology

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

To define a custom methodology:

- 1 Select **Edit > Preferences**.

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

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

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

The **Create a new set** dialog box appears.

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

### Reviewing Checks

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

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


---

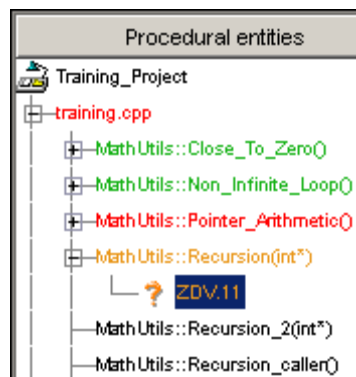
**Note** You can skip gray checks by selecting the **Skip gray checks** check box in the toolbar.

---

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

To navigate through these checks:

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



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

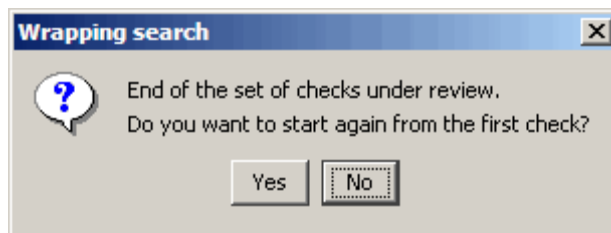
---

**Note** You can display the calling sequence and track review progress. See “Reviewing Results in Expert Mode” on page 9-26.

---

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

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



- 4 Click No.

## Reviewing Results in Expert Mode

In this section...
“What Is Expert Mode?” on page 9-26
“Switching to Expert Mode” on page 9-26
“Selecting a Check to Review” on page 9-26
“Displaying the Calling Sequence” on page 9-28
“Tracking Review Progress” on page 9-29
“Making the Reviewed Column Visible” on page 9-30
“Filtering Checks” on page 9-33
“Types of Filters” on page 9-33
“Creating a Custom Filter” on page 9-35

### What Is Expert Mode?

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

### Switching to Expert Mode

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

- Click the Viewer mode button:



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

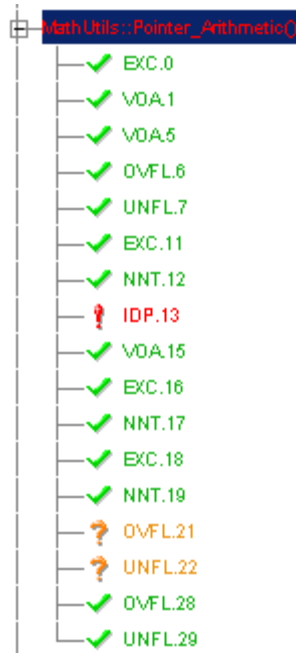
### Selecting a Check to Review

To review a check in expert mode:



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

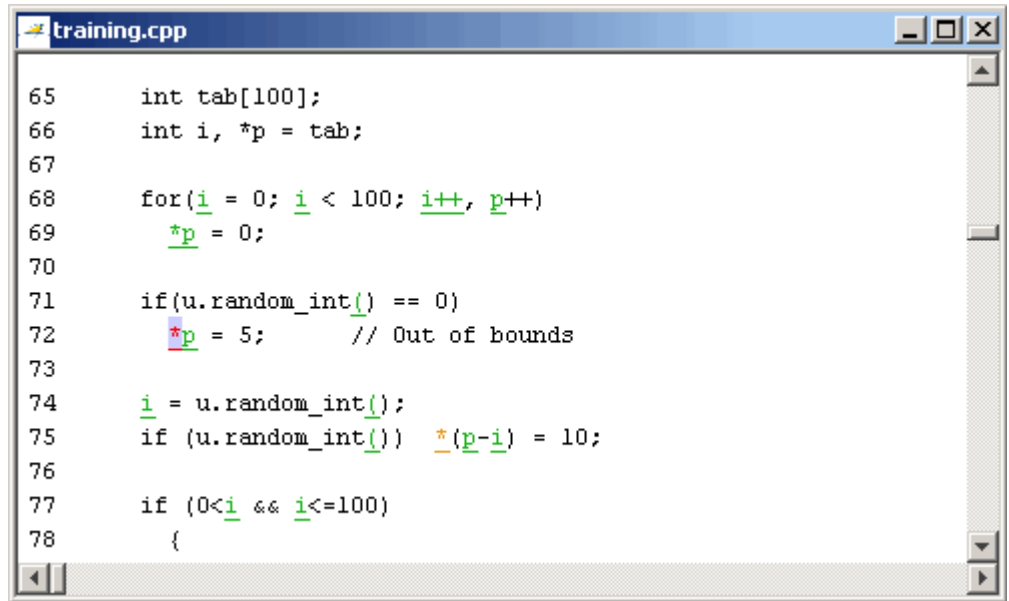
A color-coded list of the checks performed on the procedure appears:



Each item in the list of checks has an acronym that identifies the type of check and a number. For example, in 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 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 Click the colored check in the code.

An message box appears describing the error.

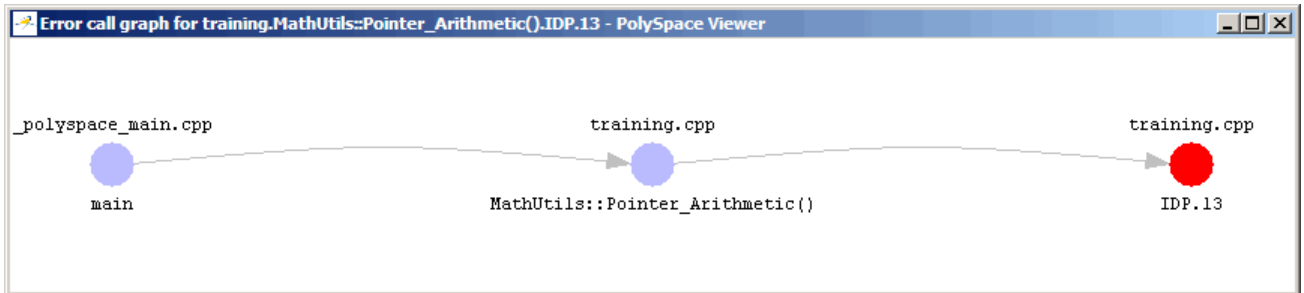
## Displaying the Calling Sequence

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

- 1 Expand the procedure containing the check you want to review.
- 2 Click the check you want to review.
- 3 Click the call graph button in the toolbar.



A window displays the call graph.



The call graph displays the code associated with the check.

## Tracking Review Progress

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

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

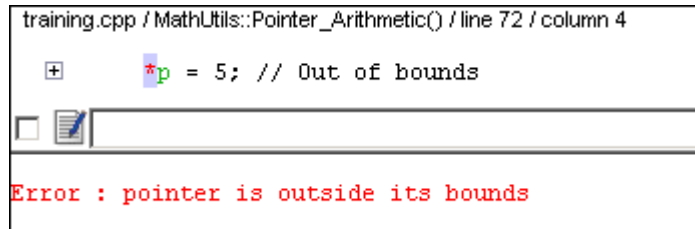
A table with statistics about the review progress for that category and severity of error appear in the upper-left part of the window.

Coding review progress	Count	Progress
nb IDP reviewed / nb IDP to review (Red)	0/1	0
nb reviewed / nb to review (Red)	0/1	0
Software reliability indicator	83/97	85

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 having the color and category of the current check. In this example, it 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 having 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.

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



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

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

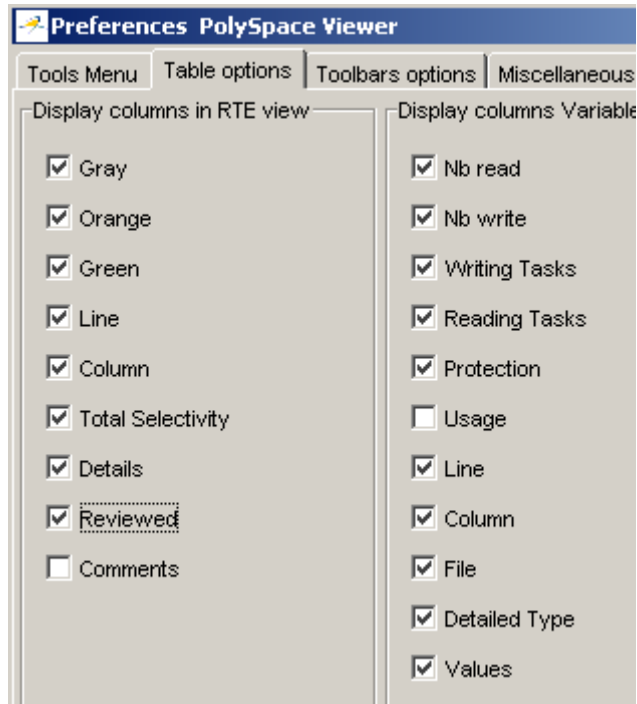
Coding review progress	Count	Progress
nb IDP reviewed / nb IDP to review (Red)	1/1	100
nb reviewed / nb to review (Red)	1/1	100
Software reliability indicator	83/97	85

## Making the Reviewed Column Visible

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

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

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



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

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

Procedural entities					Line	...	%	Details	Reviewed
Training_Project	1	0	8	45			85		<input type="checkbox"/>
+_polyspace_main.cpp				1	1		100	_polyspa...	<input type="checkbox"/>
-exception.stdh					1		0	exception...	<input type="checkbox"/>
-new.stdh					1		0	new.stdh	<input type="checkbox"/>
-training.cpp	1		8	42	1		84	training.cpp	<input type="checkbox"/>
+MathUtils::Close_To_Zero()			3	9	12	16	75	training.cpp	<input type="checkbox"/>
+MathUtils::Non_Infinite_Loop()				6	39	15	100	training.cpp	<input type="checkbox"/>
+MathUtils::Pointer_Arithmetic()	1		2	11	61	16	86	training.cpp	<input type="checkbox"/>
✓ EXC.0				1	61			function ...	<input type="checkbox"/>
✓ VDA.1					68		6		<input type="checkbox"/>
✓ VDA.5					68		22		<input type="checkbox"/>
✓ OVFL.6				1	68		23	[+] overf...	<input type="checkbox"/>
✓ UNFL.7				1	68		23	[+] under...	<input type="checkbox"/>
✓ EXC.11				1	71		17	call to ra...	<input type="checkbox"/>
✓ NNT.12				1	71		17	this-point...	<input type="checkbox"/>
! IDP.13	1				72		4	pointer wi...	<input checked="" type="checkbox"/>
✓ VDA.15					74		2		<input type="checkbox"/>
✓ EXC.16				1	74		18	call to ra...	<input type="checkbox"/>
✓ NNT.17				1	74		18	this-point...	<input type="checkbox"/>

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

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

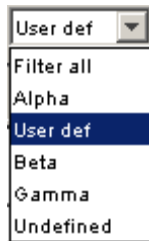
- Selects the check box for that check in the current check view (upper-right part of the window).
- Updates the counts in the coding review progress view (upper-left part of the window).

## Filtering Checks

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

The default filter is User def.

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



## Types of Filters

There are three types of filters:

- “Individual Filters” on page 9-33
- “Composite Filters” on page 9-34
- “Custom Filters” on page 9-34

## Individual Filters

You can use an individual filter to display or hide a given check category, such as VOA. When a filter is enabled, that check category does not display. For example, when the VOA filter is enabled, VOA checks do not display. When the filter is disabled, that check category displays. For example, when the VOA filter is disabled, VOA checks display. You can also filter by check color. To enable or disable an individual filter, click the toggle button for that filter on the toolbar.

**Tip** The tooltip for a filter button tells you what filter the button is for and whether the filter is enabled or disabled.

**Note** When you filter a check category, some red checks with that category will still display.

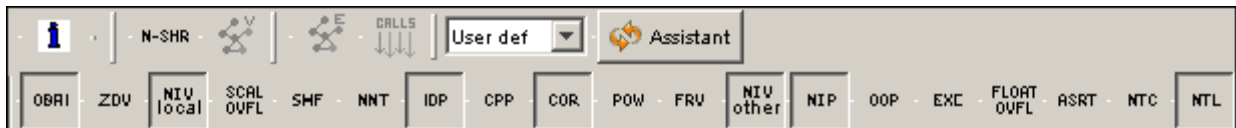
### Composite Filters

Composite filters combine individual filters, allowing you to display or hide groups of checks.

Use this filter...	To...
Alpha	Display all checks
Beta	Hide NIV, NIVL, NIP, Scalar OVFL, and Float OVFL checks
Gamma	Display red and gray checks
User def	Hide checks as defined in a custom filter that you can modify

### Custom Filters

The custom filter is a composite filter that you define. It appears on the composite filter menu as `User def` and is the default composite filter. By default, the custom filter hides the OBAI, NIV local, IDP, COR, IRV, NIV other, NIP, and NTL checks as shown in the following figure.



To modify the custom filter, see “Creating a Custom Filter” on page 9-35.



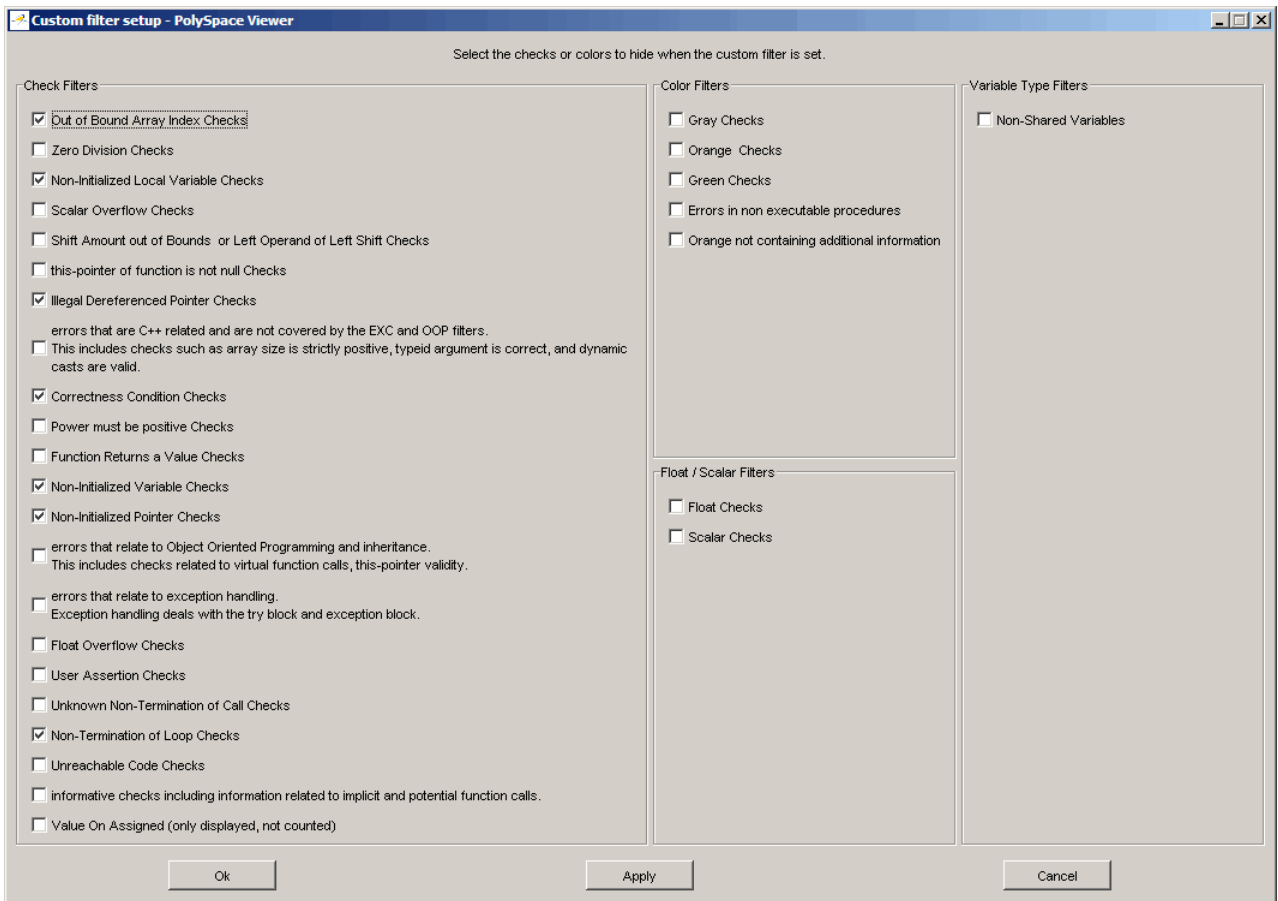
## Creating a Custom Filter

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

To modify the custom filter:

- 1 Select `User def` from the composite filters menu.
- 2 Select **Edit > Custom filters**.

The **Custom filter setup** dialog box appears.



- 3 Clear the filters for the checks that you want to display. For example, if you clear the **Out of Bound Array Index Checks** box, these checks display.

---

**Note** You do not have to change any of the selections for this tutorial.

---

- 4 Select the filters for the checks that you do not want to display.
- 5 Click **OK** to apply the changes and close the dialog box.

PolySpace software saves the custom filter definition in the Viewer preferences.

## Generating Reports of Verification Results

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

To generate an Excel report of your verification results:

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

The directory should have the following files:

```
Example_Project_Call_Tree.txt
Example_Project_RTE_View.txt
Example_Project_Variable_View.txt
Example_Project-NON-SCALAR-TABLE-APPENDIX.ps
PolySpace_Macros.xls
```

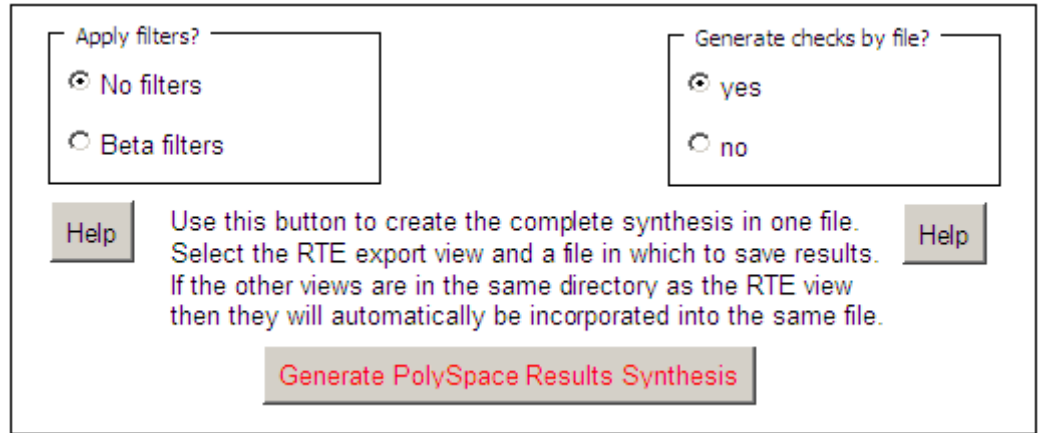
The first three files correspond to the call tree, RTE, and variable views in the PolySpace Viewer window.

- 2 Open the macros file `PolySpace_Macros.xls`.

A security warning dialog appears.

- 3 Click **Enable Macros**.

A spreadsheet appears. The top part of the spreadsheet looks like:



- 4 Specify the report options you want, then click **Generate PolySpace Results Synthesis**.

The synthesis report combines the RTE, call tree, and variables views into one report.

The **Where is the PolySpace RTE View text file** dialog box appears.

- 5 In **Look in**, navigate to the PolySpace-Doc folder in your results directory. For example: `polyspace_project\results\PolySpace-Doc`.
- 6 Select `Project_RTE_View.txt`.
- 7 Click **Open** to close the dialog box.

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

- 8 Keep the default file name and file type.
- 9 Click **Save** to close the dialog box and start the report generation.

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

Microsoft Excel - Training_Project-Synthesis.xls	
File Edit View Insert Format Tools Data Window Help Adob	
A	
1	<b>Procedural entities</b>
2	Training_Project
3	-?extern
4	-acos
5	X UNP.0
6	-acosh
7	X UNP.0
8	-asin
9	X UNP.0
10	-asinh
11	X UNP.0
12	-atan2
13	X UNP.0
14	-atanh
15	X UNP.0
16	-cos
17	X UNP.0
18	-cosh
19	X UNP.0
20	-exp
21	X UNP.0
22	-log
23	X UNP.0
24	-pow
25	X UNP.0
26	-sin
27	X UNP.0
28	-sinh
29	X UNP.0
30	-sqrt

RTE Checks Sheet 1 / Launching Options / Check Synthesis / CH

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

Microsoft Excel - Training_Project-Synthesis.xls						
File Edit View Insert Format Tools Data Window Help						
	A	B	C	D	E	F
1	<b>RTE Statistics</b>					
2	<b>Check category</b>	<b>Check detail</b>	<b>R</b>	<b>O</b>	<b>Gy</b>	<b>Gr</b>
3	OBAI	Out of Bounds Array Index	0	0	0	0
4	NIVL	Uninitialized Local Variable	0	0	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

RTE Checks Sheet 1 Launching Options **Check Synthesis**

## Using PolySpace Results

### In this section...

“Review Runtime Errors: Fix Red Errors” on page 9-41

“Red Checks Where Gray Checks were Expected” on page 9-42

“Potential Side Effect of a Red Error” on page 9-44

“Review Dead Code Checks: Why Gray Code is Interesting” on page 9-45

“Selective Orange Review: Finding the Maximum Number of Bugs in One Hour” on page 9-47

“Exhaustive Orange Review at Unit Phase” on page 9-49

“Exhaustive Orange Review at Integration Phase” on page 9-50

“Integration Bug Tracking” on page 9-52

### Review Runtime Errors: Fix Red Errors

All Runtime Errors highlighted by PolySpace verification are determined by reference to the language standard, and are sometimes implementation dependant — that is, they may be acceptable for a particular compiler but unacceptable according to the language standard.

Consider an overflow on a type restricted from -128 to 127. The computation of  $127+1$  cannot be 128, but depending on the environment a “wrap around” might be performed with a resulting value of -128.

This result is of course mathematically incorrect. If the value represents the altitude of a plane, this could result in a disaster.

By default, PolySpace verification doesn’t make assumptions about the way a variable is used. Any deviation from the recommendations of the language standard is treated as a **red error**, and must therefore be corrected.

PolySpace verification identifies two kinds of red checks

- Red errors which are *compiler-dependant* in a specific way. On some occasions a PolySpace option may be used to allow particular compiler specific behavior, and on others the code must be corrected in order to

comply. An example of a PolySpace option to permit compiler specific behavior would be the option to force “IN/OUT” ADA function parameters to be initialized. Examples in C include options to deal with constant overflows, shift operation on negative values, etc.

- All other red errors **must be fixed**. They are bugs.

Most of the bugs you’ll find are easy to correct once they are identified. PolySpace verification identifies bugs regardless of their consequence, or of the ease with which they can be corrected.

### **Red Checks Where Gray Checks were Expected**

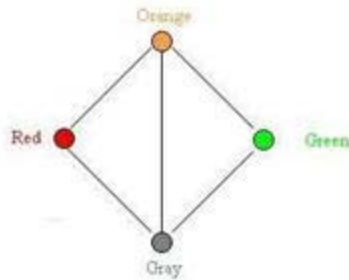
By default, PolySpace continues verification when it finds a red error. This is used to deal with two primary circumstances:

- 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 a 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;
int variable_it_points_to;

void big_red(void)
{
  int r;
  int my_zero = 0;
  if (condition==1)
    r = 1 / my_zero; // red ZDV
  ...
  ... // hundreds of lines
}

void other_function(void)
{
  if (condition==1)
    *global_ptr = 12;
}
```

```
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

## Review Dead Code Checks: Why Gray Code is Interesting

- “Functional Bugs Can Be Found in Gray Code” on page 9-45
- “Structural Coverage” on page 9-47

### Functional Bugs Can Be Found in Gray Code

PolySpace verification finds different types of dead code. Common examples include:

- Defensive code which is never reached

- Dead code due to a particular configuration
- Libraries which are not used to their full extent in a particular context
- Dead code resulting from bugs in the source code.

The causes of dead code listed in the examples below are taken from critical applications of embedded software by PolySpace verification.

- A lack of parenthesis and operand priorities in the testing clause can change the meaning significantly.

- Consider a line of code such as

```
IF NOT a AND b OR c AND d
```

Now consider how misplaced parentheses might influence how that line behaves

```
IF NOT (a AND b OR c AND d)
```

```
IF (NOT (a) AND b) OR (c AND d))
```

```
IF NOT (a AND (b OR c) AND d)
```

- The test of variable inside a branch where the conditions are never met;
- An unreachable “else” clause where the wrong variable is tested in the “if” statement
- A variable that is supposed to be local to the file but instead is local to the function
- Wrong variable prototyping leading to a comparison which is always false (say)

As is the case for red errors, the consequence of dead code and the effort needed to deal with it is unpredictable. It can vary

- From one week effort of functional testing on target, trying to build a scenario going into that branch, and wondering why the functional behavior is altered, to
- A 3 minutes code review discovering the bug.

Again, as for red errors, PolySpace doesn’t measure the impact of dead code.

The tool provides a list of dead code. A short code review will enable you to place each entry from that list into one of the five categories from the beginning of this chapter. Doing will identify known dead code and uncover real bugs.

**PolySpace experience is that at least 30% of gray code reveals real bugs.**

### **Structural Coverage**

PolySpace software always performs upper approximations of all possible executions. Therefore even if a line of code is shown in green, there remains a possibility that it is a dead portion of code. Because PolySpace verification made an upper approximation, it could not conclude that the code was dead, but it could conclude that no runtime error could be found.

PolySpace verification will find around 80% of dead code that the developer would find by doing structural coverage.

PolySpace verification is intended to be used as a productivity aid in dead code detection. It detects dead code which might take days of effort to find by any other means.

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

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

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

### Choosing What to Review

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

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

---

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

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

---

### Reviewing Oranges Quickly

While performing a selective orange review:

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

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

### Performing a Selective Orange Review

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

To perform a selective orange review:

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

2 Click **Filter all** .

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



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

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

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

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

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

7 Select another type of RTE and repeat the procedure.

---

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

---

## Exhaustive Orange Review at Unit Phase

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

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

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

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

---

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

---

### **Exhaustive Orange Review at Integration Phase**

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

#### **Cost**

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

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



## Method

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

Consider the possible reasons for an orange check:

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

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

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

Consider an example below. Suppose that

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

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

**Basic Imprecision.** On some rare occasions, a module will contain a lot of *basic imprecision* due to approximations made by PolySpace. (For more information, see “Sources of Orange Checks” on page 10-3 and “Approximations Used During Verification” in the *PolySpace Products for C Reference*).

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

Examples of extra activities might be

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

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

### Integration Bug Tracking

By default, integration bug tracking can be achieved by applying the selective orange methodology to integrated code. Each error category will be more likely to reveal integration bugs, depending on the chosen coding rules for the project.

For instance, consider a function receives two unbounded integers. The presence of an overflow can only be checked at integration phase, since at unit phase the first mathematical operation will reveal an orange check.

Consider these two circumstances:

- When integration bug tracking is performed in isolation, a selective orange review will highlight most integration bugs. In this case, a PolySpace verification has been performed integrating tasks.
- When integration bug tracking is performed together with an exhaustive orange review at unit phase, a PolySpace verification has been performed on one or more files.

In this second case, an exhaustive orange review will already have been performed file by file. Therefore, at integration phase **only checks that have turned from green to another color** are worth assessing.

For instance, if a function takes a structure as an input parameter, the standard hypothesis made at unit level is that the structure is well initialized. This will consequentially display a green NIV check at the first read access to a field. But this might not be true at integration time, where this check can turn orange if any context does not initialize these fields.

These orange checks will reveal integration bugs.



# Managing Orange Checks

---

- “Understanding Orange Checks” on page 10-2
- “Reducing Orange Checks in Your Results” on page 10-6
- “Reviewing Orange Checks” on page 10-15

## Understanding Orange Checks

### In this section...

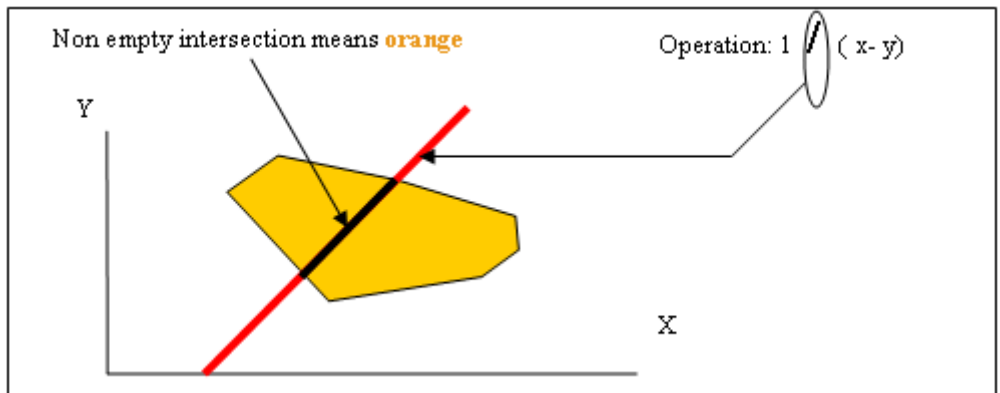
“What is an Orange Check?” on page 10-2

“Sources of Orange Checks” on page 10-3

“Determining Cause of Orange Checks” on page 10-5

### What is an Orange Check?

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



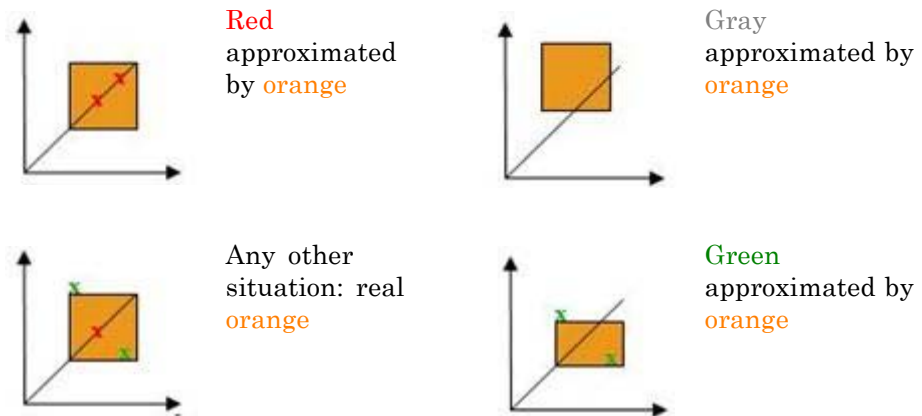
### Graphical Representation of an Orange Check

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

---

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

---



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

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

## Sources of Orange Checks

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

- **Potential bug** — an orange check can represent a real bug.  
Example - loop with division by zero
- **Inconclusive check** — an orange check can represent a situation where PolySpace verification is unable to conclude whether a problem exists. It is sometimes in the nature of software code that it cannot be concluded

whether there is a potential error. In the example below, the task T1 can be started before or after T2, so PolySpace verification cannot conclude without the calling sequence being defined.

- Consider a variable X initialized to 0, and two concurrent tasks T1 and T2.
- Suppose that T1 assigns a value of 12 to variable X
- Now suppose that T2 divides a local variable by X. The division is shown as an orange check because T1 can be started before or after T2 (so a division by zero is possible).
- **Data set issue** — an orange check resulting from a theoretical set of data. PolySpace verification considers all combinations of input data rather than *one* particular combination (that is, it uses an upper approximation of the data set). Therefore a check may be colored orange as the result of a combination of input values which is analyzed by PolySpace, but which will not be possible at execution time.
  - Consider three variables X, Y and Z which can vary between 1 and 1000
  - Now suppose that the code computes a value of  $X*Y*Z$  on a type 16 bits. The result can potentially overflow. It may be known when the code is developed that the variables cant all take the value 1000 at the same time, but this information is not available to PolySpace software. The code will be colored orange, accordingly.
- **Basic imprecision** — an orange check can be due to an imprecise approximation.
  - Consider that X, before the function call, can have the following values: -5, -3, 8, or any value in range [10 . . . 20].
  - This means that 0 has been excluded from the set of possible values for X. Therefore, PolySpace software will approximate X in the range [-5 . . . 20], instead of the previous unions of values, because of imprecision and optimization.
  - In this case, calling the function  $x = 1/x$  leads to an orange ZDV. PolySpace is not able to prove the absence of a run-time error.



## Determining Cause of Orange Checks

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

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

## Reducing Orange Checks in Your Results

### In this section...

“Options to Reduce Orange Checks” on page 10-6

“Generic Objectives: A Balance Between Precision and Verification Time” on page 10-7

“Applying Coding Rules to Reduce Orange Checks” on page 10-8

“Varying the Precision Level” on page 10-8

“Applying Software Safety Level Wisely” on page 10-9

“Adding Precision Constraints at the Periphery Via Stubs” on page 10-11

“Tuning Advanced Parameters” on page 10-13

“Applying Data Ranges” on page 10-13

“Considering the Effects of Application Code Size” on page 10-14

### Options to Reduce Orange Checks

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

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

## Generic Objectives: A Balance Between Precision and Verification Time

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

PolySpace verification needs to be fast and precise.

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

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

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

---

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

---

---

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

---

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

---

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

---

### Applying Coding Rules to Reduce Orange Checks

The number of **orange checks** per file strongly depends on the coding style used in the project.

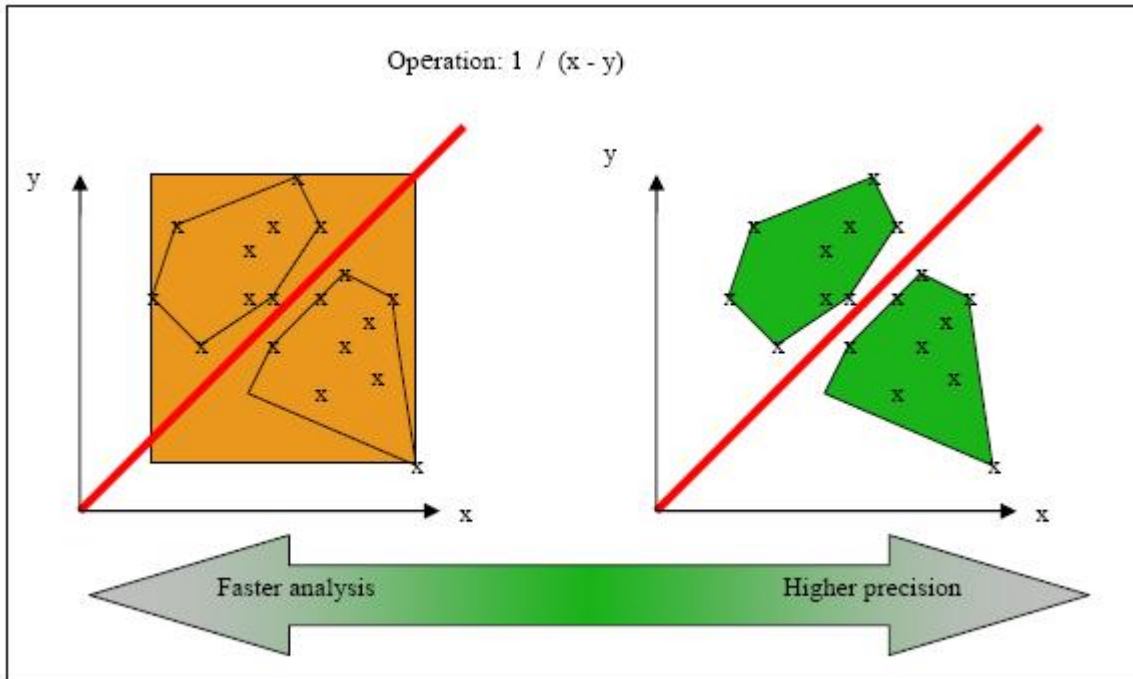
If your code follows JSF++ coding rules, the number of checks per file will usually decrease substantially.

For more information on the checking JSF++ coding rules, see Chapter 12, “JSF C++ Checker”.

### Varying the Precision Level

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

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



### Vary the Precision Rate

## Applying Software Safety Level Wisely

### Abstract

What are the differences between verification levels

### Explanation

There follows an example of the distinction between Safety Analysis levels 1, 2 and 3. The deeper the verification goes, the more precise it is. Depending on the backward/forward dependencies, oranges will be solved at the Safety Analysis level 1, and some later in level 2 or 3.

- **One way to effect precision is to select which algorithm will model your cloud of points.** The modelling is internal, and represented by a precision level ranging from 0 to 2. You can select a particular precision

level for a specific body, which might differ from the default value for the rest of the code.

- **The level of a verification is the depth of verification of PolySpace Verification.** It starts with Safety Analysis 1 (which approximates to unit verification) and normally goes up to level 4 (although it can go further if exceptional circumstances require it). Each iteration corresponds to a deeper level of propagation of calling and called context, as illustrated below. A level of iteration is selected for the whole application and unlike the precision level, it cannot be varied on a body-by-body basis.

PolySpace verification performs 4 levels of Software Safety Analysis by default. Below is an example of the distinction between Safety Analysis levels 1, 2 and 3; the deeper the verification goes, the more precise it is. Depending on the backward/forward dependencies, oranges will be resolved into red, green or gray at the Safety Analysis level 1 or later in level 2, 3 or 4.

The level of a verification represents the number of iterations performed by PolySpace verification. Each iteration corresponds to a deeper level of propagation of calling and called contexts. As an example, a division by an input parameter of a function might produce an orange during Level 1 verification and then subsequently turn into green during level 2 or 3. PolySpace software gains a more accurate knowledge of x when the value is propagated deeper. Unlike the precision which is tuned for specific modules, the level of safety verification is set for the whole application.

Safety Analysis Level 1	Safety Analysis Level 2	Safety Analysis Level 3
<pre>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, &amp;y); *t = 1.0/(v - 2.0 * x); }</pre>	<pre>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, &amp;y); *t = 1.0/(v - 2.0 * x); }</pre>	<pre>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, &amp;y); *t = 1.0/(v - 2.0 * x); }</pre>

Safety Analysis Level 1	Safety Analysis Level 2	Safety Analysis Level 3
<pre>float level2(float v) {   float t;   t = v;   level1(0.0, 1.0, &amp;t);   return t; } void main(void) {   float r,d;   d= level2(1.0);   r = 1.0 / (2.0 - d); }</pre>	<pre>float level2(float v) {   float t;   t = v;   level1(0.0, 1.0, &amp;t);   return t; } void main(void) {   float r,d;   d= level2(1.0);   r = 1.0 / (2.0 - d); }</pre>	<pre>float level2(float v) {   float t;   t = v;   level1(0.0, 1.0, &amp;t);   return t; } void main(void) {   float r,d;   d= level2(1.0);   r = 1.0 / (2.0 - d); }</pre>

## Adding Precision Constraints at the Periphery Via Stubs

Another way to increase the selectivity is to indicate to the PolySpace software that some variables (detailed below) might vary between some functional ranges instead of the full range of the considered type.

This primarily concerns two items from the language:

- Parameters passed to functions.
- Variables' content, mostly globals, which might change from one execution to another. Typically, these might include things like calibration data or mission specific data. These variables might be read directly within the code, or read through an API of functions.

### Reduce the cloud of points

Stubs do not need to model the details of the functions or procedures involved. They only need to represent the effect that the code might have on the remainder of the system.

If a function is supposed to return an integer, the default automatic stubbing will stub it on the assumption that it can potentially take any value from the full type of an integer.

Given that PolySpace models data ranges throughout the code it verifies, it will obviously produce more precise, informative results, provided that the data it considers from the “outside world” is representative of the data that can be expected when the code is implemented. There is a certain number of mechanisms available to model such a data range within the code itself, and three possible approaches are presented here. There is no particular advantage in using one approach or another (except, perhaps, that the assertions in the first two will usually generate orange checks) – it is largely down to personal preference.

with volatile and assert	with assert and without volatile	without assert, without volatile, without "if"
<pre>#include &lt;assert.h&gt;  int stub(void) {     volatile int random;     int tmp;     tmp = random;     assert(tmp&gt;=1 &amp;&amp; tmp&lt;=10 );     return</pre>	<pre>#include &lt;assert.h&gt;  extern int other_func(void); int stub(void) {     int tmp;     tmp= other_func();     assert(tmp&gt;=1 &amp;&amp; tmp&lt;=10);     return }</pre>	<pre>extern int other_func(void); int stub(void) {     int tmp;     do {tmp= other_func();}     while (tmp&lt;1    tmp&gt;10);     return tmp; }</pre>

### Increase the Number of Red and Green Checks

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

```
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;
}

```

- By relying 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 **green**.
- A **red** division could only be achieved with a manual stub.

Applying fine-level modelling of constraints in primitives and outside functions at the application periphery will propagate more precision throughout the application, which will result in a higher selectivity rate (more proven colors, i.e. more **red**+ **green** + gray)

## Tuning Advanced Parameters

The Advanced Parameters provide a degree of control over some aspects of PolySpace internal tuning. These are provided to allow the user to concentrate verification time on specific aspects of the software. For example, the user can decide whether or not to expand arrays and records by modelling each element as a separate variable.

These options are specific to each language. Refer to “*Precision/Scaling Options*” in the *PolySpace Products for C Reference*.

```

-0(0-3)
-modules-precision mod1:0(0-3)[,mod2:0(0-3)[,...]]

```

## Applying Data Ranges

By default, PolySpace verification assumes that all data inputs are set to their full range. Therefore, nearly any operation on these inputs could produce an overflow. The Data Range Specifications (DRS) module allows you to set

external constraints on global variables and stub function return values. This can substantially reduce the number of orange checks in the verification results.

For more information, see “Applying Data Ranges to External Variables and Stub Functions (DRS)” on page 5-14.

## 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 instance, 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...

“Selective Orange Review” on page 10-15

“Performing a Selective Orange Review” on page 10-16

“Exhaustive Orange Review” on page 10-17

“Performing an Exhaustive Orange Review” on page 10-18

## Selective Orange Review

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

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

## Choosing What to Review

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

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

---

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

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

---

### Review Oranges Quickly

While performing a selective orange review:

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

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

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

### Performing a Selective Orange Review

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

To perform a selective orange review:

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

- 2 Click **Filter all**  .

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



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

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

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

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

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

**7** Select another type of RTE and repeat the procedure.

---

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

---

## Exhaustive Orange Review

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

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

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

### Performing an Exhaustive Orange Review

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

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

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

Consider the possible reasons for an orange check:

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

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

### Inconclusive

The most interesting type of inconclusive check is identified when PolySpace verification states that the code is too complicated. In such a case it is usually true that most **orange checks** in the problem file are related, and that patient navigation will always draw the user back to a same cause — perhaps a function or a variable modified many times. Experience suggests that such

situations often focus on functions or variables which have also caused trouble earlier in the development cycle.

Consider an example below. Suppose that

- a *signed* is an integer between  $-2^{31}$  and  $2^{31}-1$
- an *unsigned* is an integer between 0 and  $2^{32}-1$
- The variable "Computed\_Speed" is copied into a signed, and afterward into an unsigned, then signed, then added to another variable, and finally produces 20 orange overflows (OVFL).

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

### **Basic Imprecision**

On some rare occasions, a module will contain a lot of *basic imprecision* due to approximations made by PolySpace. (For more information, see "Sources of Orange Checks" on page 10-3 and "Approximations Used During Verification" in the *PolySpace Products for C Reference*).

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

Examples of extra activities might be

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

### **Real Bugs and Data Sets**

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



# Day to Day Use

---

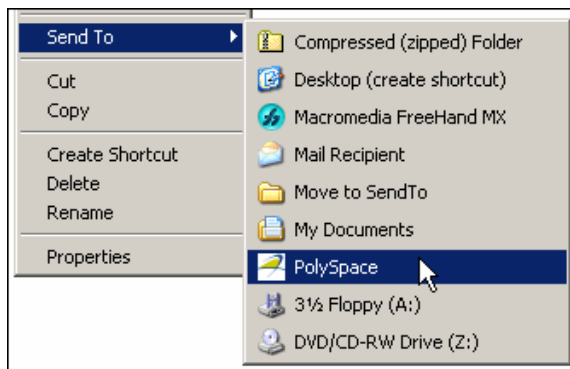
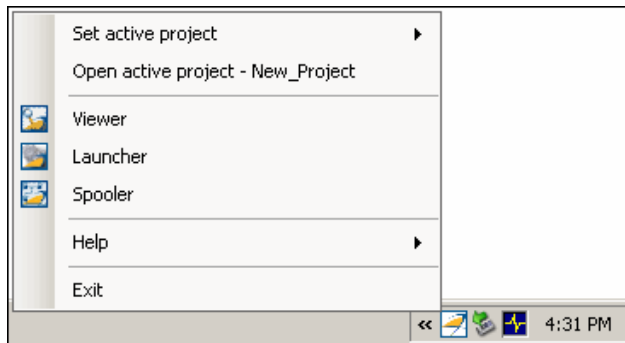
- “PolySpace In One Click Overview” on page 11-2
- “Using PolySpace In One Click” on page 11-3

## PolySpace In One Click Overview

Most developers verify the same files multiple times (writing new code, unit testing, integration), and usually need to run verifications on multiple project files using the same set of options. In a Microsoft Windows environment, PolySpace In One Click provides a convenient way to streamline your work when verifying several files using the same set of options.

Once you have set up a project file with the options you want, you designate that project as the *active project*, and then send the source files to PolySpace software for verification. You do not have to update the project with source file information.

On a Windows systems, the plug-in provides a PolySpace Toolbar in the Windows Taskbar, and a **Send To** option on the desktop pop-up menu:



## Using PolySpace In One Click

### In this section...

“PolySpace In One Click Workflow” on page 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

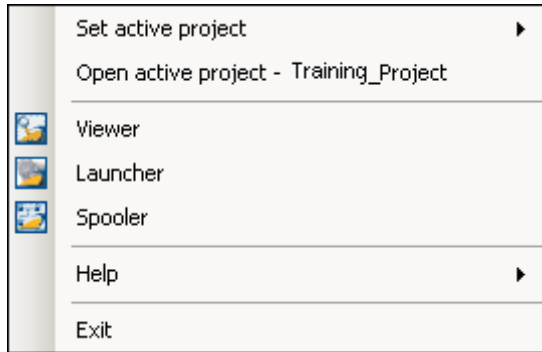
The active project is the project that PolySpace In One Click uses to verify the files that you select. Once you have set an active project, it remains active until you change the active project. PolySpace software uses the analysis options from the project; it does not use the source files or results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

To set the active project:

- 1 Right-click the PolySpace In One Click icon in the taskbar area of your Windows desktop:

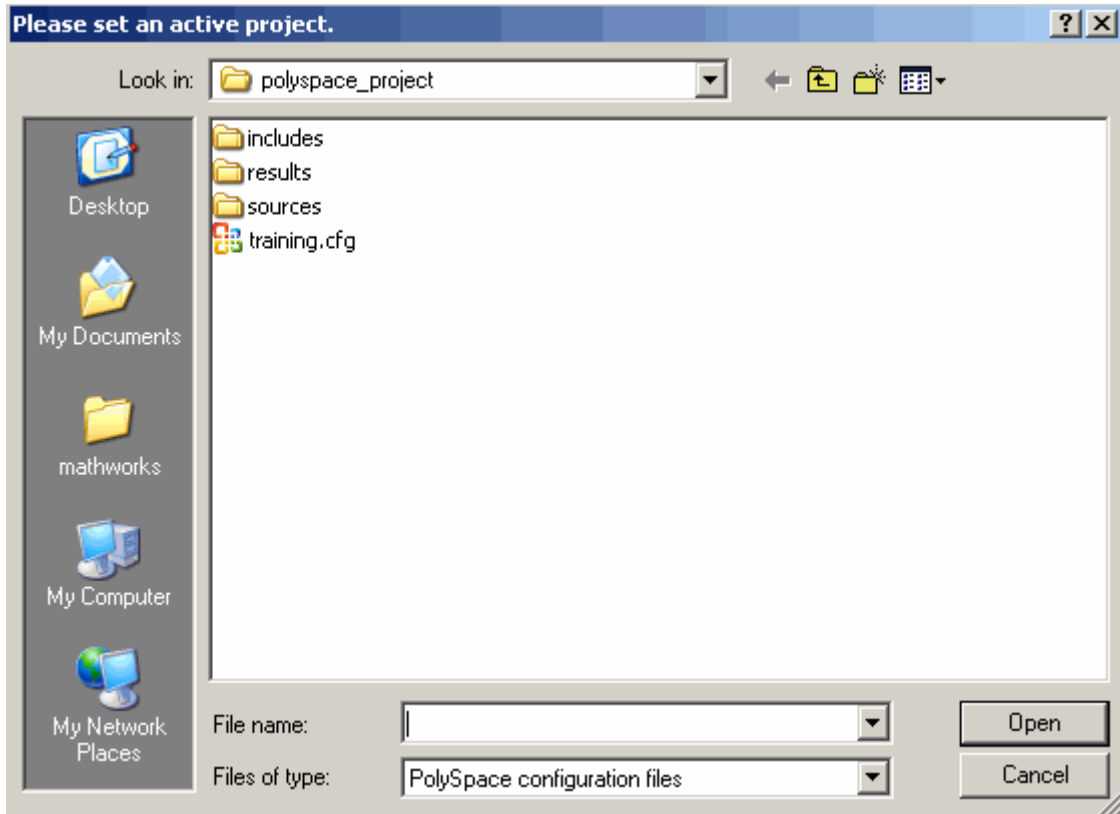


The context menu appears.



**2** Select **Set active project > Browse** from the menu.

The **Please set an active project** dialog box appears:



- 3 Select the project you want to use as the active project.
- 4 Click **Open** to apply the changes and close the dialog box.

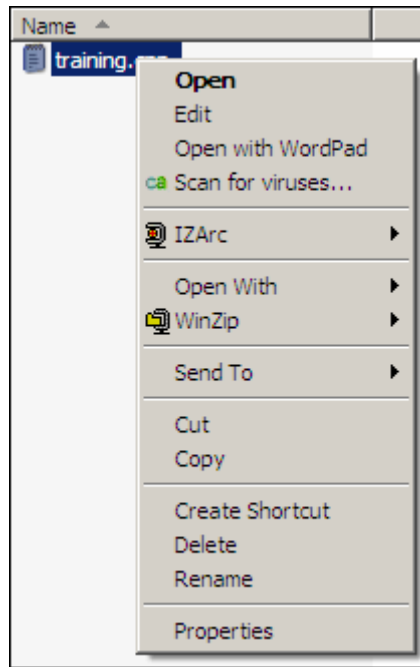
## Launching Verification

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

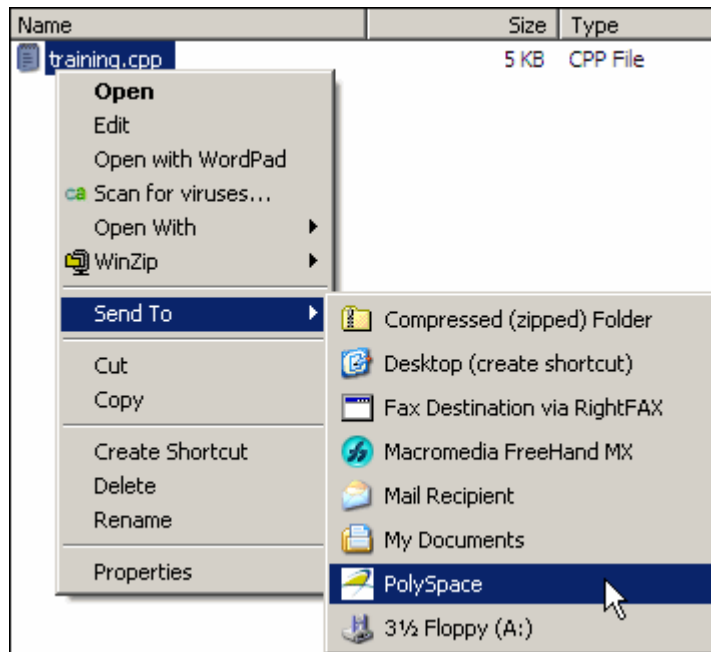
To send a file to PolySpace software for verification:

- 1 Navigate to the directory containing the source files you want to verify.
- 2 Right-click the file you want to verify.

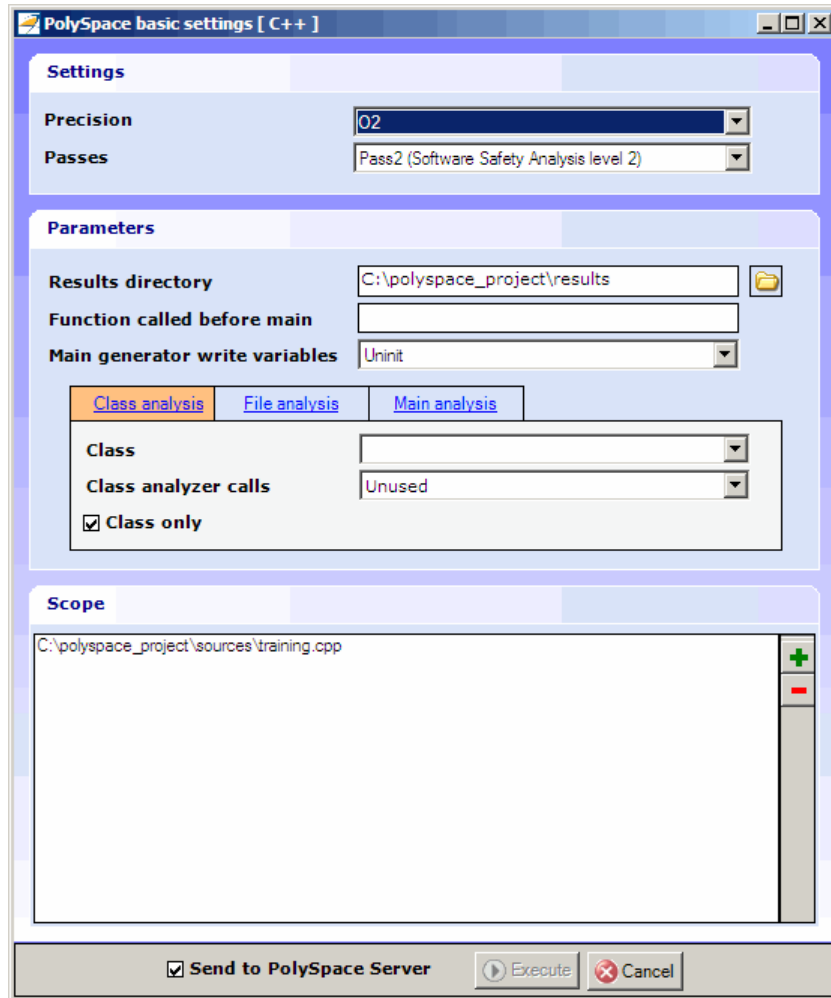
The context menu appears.



**3** Select **Send To > PolySpace**.



The **PolySpace basic settings** dialog box appears.




---

**Note** The options you specify the basic settings dialog box override any options set in the configuration file. These options are also preserved between verifications.

---

- 4 Enter the appropriate parameters for your verification.



- 5 Leave the default values for the other parameters.
- 6 Click **Execute**.

The verification starts and the verification log appears.

```

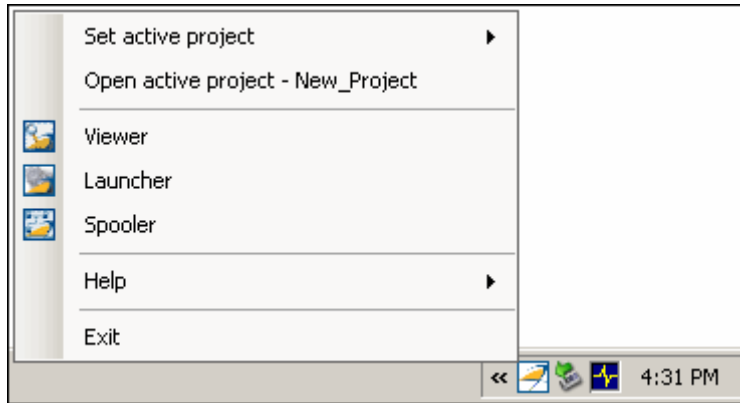
C:\polyspace_project\results
Number of lines      : 202
Number of lines without comments : 106
Number of lines with libraries : 6876
*****
***
*** C++ source compliance checking done
***
*****
Ending at: Jul 22, 2008 14:6:17
Generating remote file
Done
User time for polyspace-cpp: 1.9real, 1.9u + 0s
***
*** End of PolySpace Verifier analysis
***
Adding the analysis to the queue ...
Transferring the archive to the server ...
.....
Transfer completed.
Analysis ID : 1
The analysis has been queued. You may follow its progress using the spooler.

The analysis has been successfully done

```

## Using the Taskbar Icon

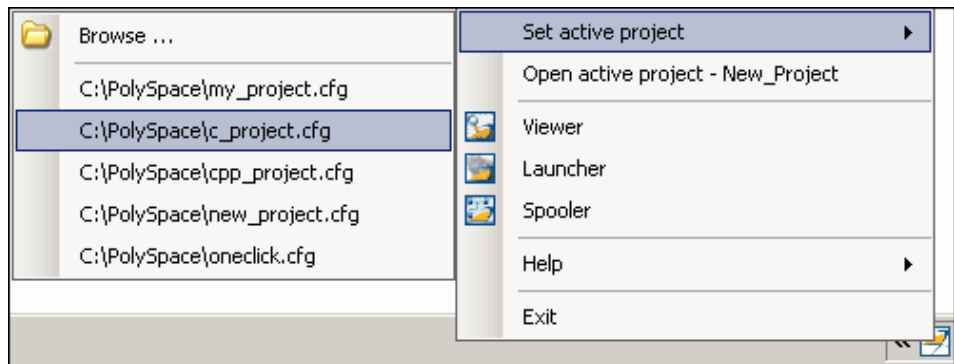
The PolySpace in One Click Taskbar icon allows you to access various software features.



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

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

A standard file browser allows you to choose the configuration file. If you have multiple configuration files, you can quickly switch between them using the browse history.



---

**Note** No configuration file is selected by default. You can create an empty file with a .cfg extension.

---

- **Open active project** — Opens the active configuration file. This allows you to update the project using the standard PolySpace Launcher graphical interface. It allows you to specify all PolySpace common options, including directives of compilation, options, and paths of standard and specific headers. It does not affect the precision of a verification or the results directory.
- **Viewer** — Opens the PolySpace viewer. This allows you to review verification results in the standard graphical interface. In order to load results into the viewer, you must choose a verification to review in the Verification Log window.
- **Launcher** — Opens the PolySpace Launcher. This allows you to launch a verification using the standard PolySpace graphical interface.
- **Spooler** — Opens the PolySpace Spooler. If you selected a server verification in the “PolySpace Preferences” dialog box, the spooler allows you to follow the status of the verification.



# JSF C++ Checker

---

- “PolySpace JSF C++ Checker Overview” on page 12-2
- “Using the PolySpace JSF C++ Checker” on page 12-3
- “Supported Rules ” on page 12-11
- “Rules Not Checked” on page 12-36

## **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. The JSF C++ checker can check 120 of the 221 JSF++ programming rules .

---

**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](http://www.jsf.mil/downloads/documents/JSF_AV_C++_Coding_Standards_Rev_C.doc).

---

## Using the PolySpace JSF C++ Checker

### In this section...

“Setting Up JSF++ Checking” on page 12-3

“Running a Verification with JSF++ Checking” on page 12-7

### Setting Up JSF++ Checking

- “Activating the JSF C++ Checker” on page 12-3
- “Creating a JSF++ Rules File” on page 12-4
- “Excluding Files from JSF++ Checking” on page 12-6

### 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 > Check JSF-C++: 2005 rules**.

The software displays the two JSF++ options: `jsf-coding-rules` and `includes-to-ignore`.

<input type="checkbox"/> Check JSF-C++: 2005 rules	<input checked="" type="checkbox"/>		
Rules configuration		...	-jsf-coding-rules
Files and directories to ignore		...	-includes-to-ignore


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

- 4 Select the **Check JSF-C++: 2005 rules** check box.

### Creating a JSF++ Rules File

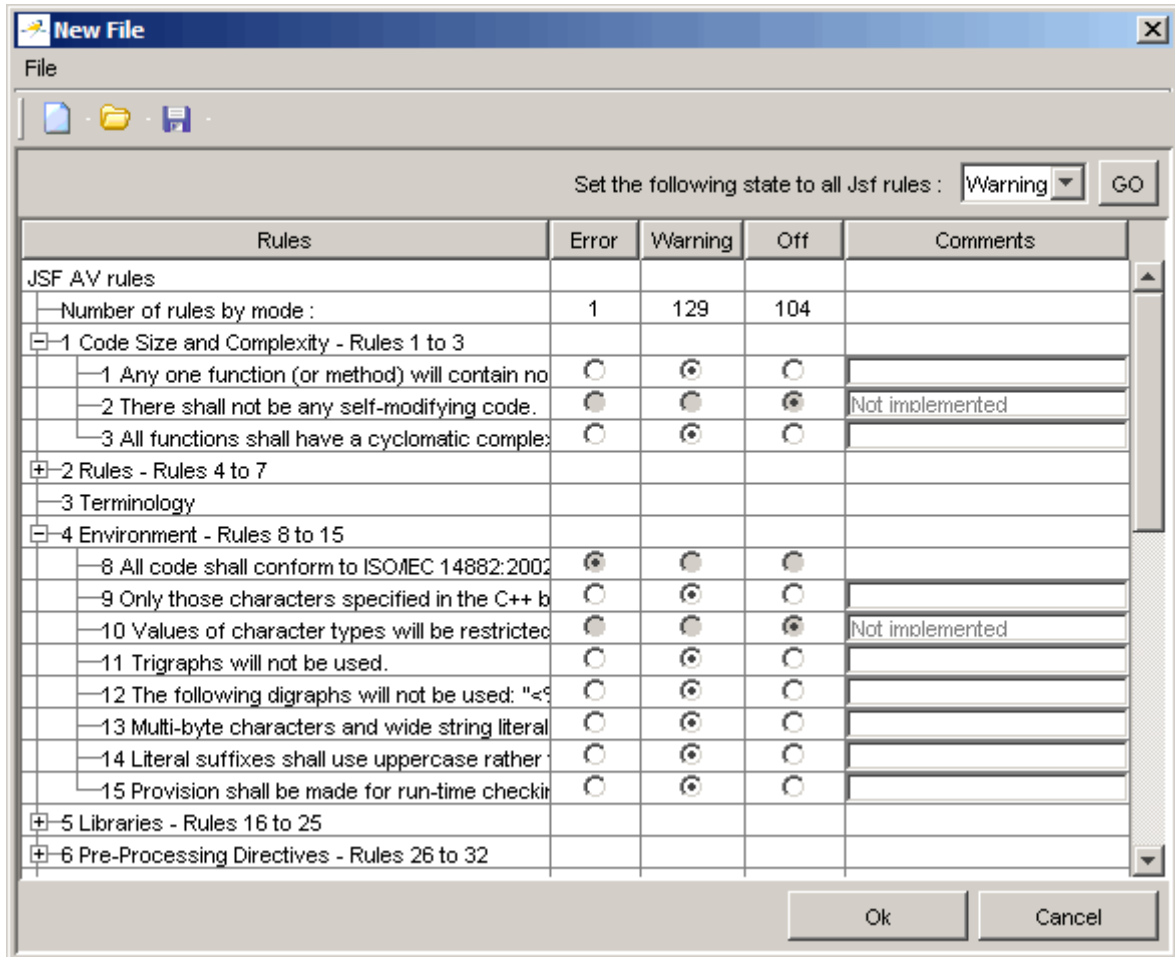
You must have a rules file to run a verification with JSF++ checking. You can use an existing file or create a new one.

To create a new rules file:

- 1 Click the button  to the right of the **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.

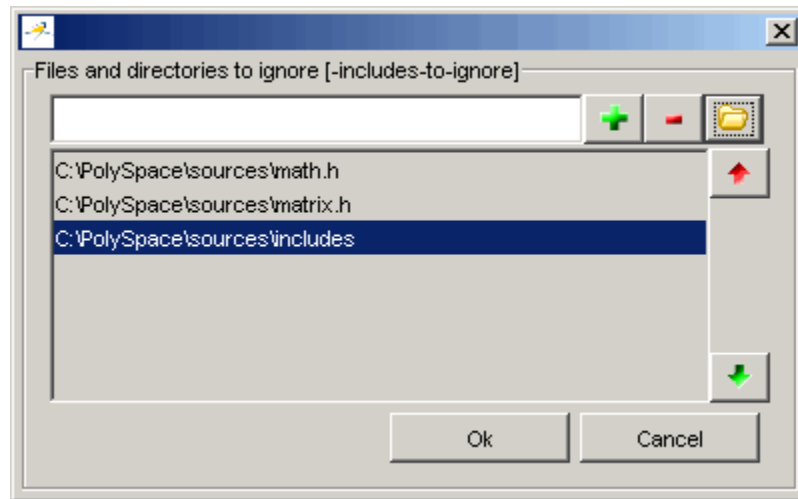
### Excluding Files from JSF++ Checking

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 directories to ignore** option.

The **Files and directories to ignore (includes-to-ignore)** dialog box opens.



- 2 Click the folder icon .

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

- 3 Select the files or directories you want to exclude.
- 4 Click **OK**.

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

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

## Running a Verification with JSF++ Checking

- “Starting the Verification” on page 12-7
- “Examining the JSF Log” on page 12-8

### Starting the Verification


When you run a verification with the **Check JSF-C++:2005 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.

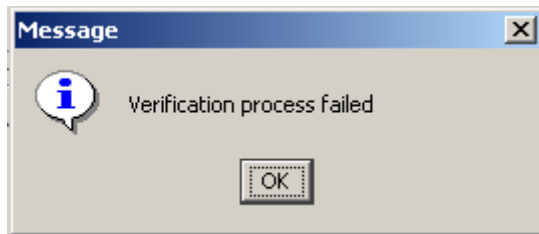
---

**Note** Some rules address run-time errors.

---

To start the verification:

- 1 Click the **Execute** button .
- 2 If you see a caution that PolySpace software will remove existing results from the results directory, click **Yes** to continue and close the message dialog box.
- 3 If the verification fails because of JSF++ violations. A message dialog box appears.



- 4 Click **OK**.

### Examining the JSF Log

To examine the JSF++ violations:

- 1 Click the **JSF** button in the log area of the Launcher window.  
A list of JSF++ violations appear in the log part of the window.

Status	Rule	File	Line	Col
?	180	training.cpp	21	8
!	191	training.cpp	46	19
?	180	training.cpp	109	15
?	180	training.cpp	142	38

- 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.

Status	Rule	File	Line	Col
?	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 ca...

**File:** C:\PolySpace\polyspace\_project\sources\training.cpp line 46 (column 19)

**Source code**

```

|     if (y > big) break;
|                   ^

```

- Right click the row containing the violation of rule 191, then select Open Source File.

Status	Rule	File	Line	Col
?	180	training.cpp	21	8
!	191	tra	46	19
?	180	tra	109	15
?	180	tra	142	38

- Open Source File
- 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 4-16.

- 4 Correct any violations reported in the log, then run the verification again to ensure compliance with all JSF++ rules.

## Supported Rules

<b>In this section...</b>
“Code Size and Complexity” on page 12-12
“Environment” on page 12-12
“Libraries” on page 12-13
“Pre-Processing Directives” on page 12-14
“Header Files” on page 12-15
“Style” on page 12-15
“Classes” on page 12-19
“Namespaces” on page 12-23
“Templates” on page 12-23
“Functions” on page 12-23
“Comments” on page 12-25
“Declarations and Definitions” on page 12-25
“Initialization” on page 12-26
“Types” on page 12-27
“Constants” on page 12-27
“Variables” on page 12-27
“Unions and Bit Fields” on page 12-28
“Operators” on page 12-28
“Pointers and References” on page 12-30
“Type Conversions” on page 12-31
“Flow Control Standards” on page 12-32
“Expressions” on page 12-33
“Memory Allocation” on page 12-35
“Fault Handling” on page 12-35
“Portable Code” on page 12-35

## Code Size and Complexity

N.	JSF++ Definition	Comments
1	Any one function (or method) <b>will</b> 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 <b>shall</b> 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 <b>shall</b> 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 <b>will</b> be used.	
11	Trigraphs <b>will not</b> be used.	
12	The following digraphs <b>will not</b> 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 <b>will not</b> be used.	Report L 'c' and L "string" and use of wchar_t.
14	Literal suffixes <b>shall</b> use uppercase rather than lowercase letters.	
15	Provision <b>shall</b> be made for run-time checking (defensive programming).	Done with RTE checks in the Verifier.



## Libraries

N.	JSF++ Definition	Comments
17	The error indicator <code>errno</code> <b>shall not</b> be used.	<code>errno</code> should not be used as a macro or a global with external "C" linkage.
18	The macro <code>offsetof</code> , in library <code>&lt;stddef.h&gt;</code> , <b>shall not</b> be used.	<code>offsetof</code> should not be used as a macro or a global with external "C" linkage.
19	<code>&lt;locale.h&gt;</code> and the <code>setlocale</code> function <b>shall not</b> be used.	<code>setlocale</code> and <code>localeconv</code> should not be used as a macro or a global with external "C" linkage.
20	The <code>setjmp</code> macro and the <code>longjmp</code> function <b>shall not</b> be used.	<code>setjmp</code> and <code>longjmp</code> should not be used as a macro or a global with external "C" linkage.
21	The signal handling facilities of <code>&lt;signal.h&gt;</code> <b>shall not</b> be used.	<code>signal</code> and <code>raise</code> should not be used as a macro or a global with external "C" linkage.
22	The input/output library <code>&lt;stdio.h&gt;</code> <b>shall not</b> be used.	all standard functions of <code>&lt;stdio.h&gt;</code> should not be used as a macro or a global with external "C" linkage.
23	The library functions <code>atof</code> , <code>atoi</code> and <code>atol</code> from library <code>&lt;stdlib.h&gt;</code> <b>shall not</b> be used.	<code>atof</code> , <code>atoi</code> and <code>atol</code> should not be used as a macro or a global with external "C" linkage.
24	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> from library <code>&lt;stdlib.h&gt;</code> <b>shall not</b> be used.	<code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> should not be used as a macro or a global with external "C" linkage.
25	The time handling functions of library <code>&lt;time.h&gt;</code> <b>shall not</b> be used.	<code>clock</code> , <code>difftime</code> , <code>mktime</code> , <code>asctime</code> , <code>ctime</code> , <code>gmtime</code> , <code>localtime</code> and <code>strftime</code> 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 <b>shall</b> be used: <code>#ifndef</code> , <code>#define</code> , <code>#endif</code> , <code>#include</code> .	
27	<code>#ifndef</code> , <code>#define</code> and <code>#endif</code> <b>will</b> be used to prevent multiple inclusions of the same header file. Other techniques to prevent the multiple inclusions of header files <b>will not</b> be used.	Detects the patterns <code>#if !defined</code> , <code>#pragma once</code> , <code>#ifdef</code> , and missing <code>#define</code> .
28	The <code>#ifndef</code> and <code>#endif</code> pre-processor directives <b>will</b> 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 <code>#ifndef</code> .
29	The <code>#define</code> pre-processor directive <b>shall not</b> 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: <ul style="list-style-type: none"> <li>• 29.def : The <code>#define</code> pre-processor directive shall not be used to create inline macros.</li> <li>• 29.use : Inline functions shall be used instead of inline macros</li> </ul>
30	The <code>#define</code> pre-processor directive <b>shall not</b> be used to define constant values. Instead, the <code>const</code> qualifier <b>shall</b> be applied to variable declarations to specify constant values.	Reports <code>#define</code> of simple constants.
31	The <code>#define</code> pre-processor directive <b>will</b> only be used as part of the technique to prevent multiple inclusions of the same header file.	Detects use of <code>#define</code> that are not used to guard for multiple inclusion, assuming that rules 35 and 27 are not violated.
32	The <code>#include</code> pre-processor directive <b>will</b> only be used to include header (*.h) files.	

## Header Files

N.	JSF++ Definition	Comments
33	The <code>#include</code> directive <b>shall</b> use the <code>&lt;filename.h&gt;</code> notation to include header files.	
35	A header file <b>will</b> contain a mechanism that prevents multiple inclusions of itself.	
39	Header files ( <code>*.h</code> ) <b>will not</b> 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 <b>will</b> be kept to a length of 120 characters or less.	
42	Each expression-statement <b>will</b> be on a separate line.	Reports when two consecutive expression statements are on the same line.
43	Tabs <b>should</b> 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) <b>will not</b> rely on significance of more than 64 characters.	

N.	JSF++ Definition	Comments
47	Identifiers <b>will not</b> begin with the underscore character <code>'_'</code> .	
48	Identifiers <b>will not</b> differ by: <ul style="list-style-type: none"> <li>• Only a mixture of case</li> <li>• The presence/absence of the underscore character</li> <li>• The interchange of the letter 'O'; with the number '0' or the letter 'D'</li> <li>• The interchange of the letter 'I'; with the number '1' or the letter 'l'</li> <li>• The interchange of the letter 'S' with the number '5'</li> <li>• The interchange of the letter 'Z' with the number 2</li> <li>• The interchange of the letter 'n' with the letter 'h'</li> </ul>	Checked regardless of scope. Not checked between macros and other identifiers. Messages in log file: <ul style="list-style-type: none"> <li>• Identifier "Idf1" (file1.cpp line 11 column c1) and "Idf2" (file2.h line 12 column c2) only differ by the presence/absence of the underscore character.</li> <li>• Identifier "Idf1" (file1.cpp line 11 column c1) and "Idf2" (file2.h line 12 column c2) only differ by a mixture of case.</li> <li>• Identifier "Idf1" (file1.cpp line 11 column c1) and "Idf2" (file2.h line 12 column c2) only differ by letter 'O', with the number '0'.</li> </ul>
50	The first word of the name of a class, structure, namespace, enumeration, or type created with <code>typedef</code> <b>will</b> begin with an uppercase letter. All others letters <b>will</b> be lowercase.	Messages in log file: <ul style="list-style-type: none"> <li>• The first word of the name of a class will begin with an uppercase letter.</li> <li>• The first word of the namespace of a class will begin with an uppercase letter.</li> </ul>
51	All letters contained in function and variables names <b>will</b> be composed entirely of lowercase letters.	Messages in log file: <ul style="list-style-type: none"> <li>• All letters contained in variable names will be composed entirely of lowercase letters.</li> <li>• All letters contained in function names will be composed entirely of lowercase letters.</li> </ul>

N.	JSF++ Definition	Comments
52	Identifiers for constant and enumerator values <b>shall</b> be lowercase.	Messages in log file: <ul style="list-style-type: none"> <li>• Identifier for enumerator value shall be lowercase.</li> <li>• Identifier for template constant parameter shall be lowercase.</li> </ul>
53	Header files <b>will</b> always have file name extension of ".h".	.H is allowed if you set the option -dos.
53.1	The following character sequences <b>shall</b> not appear in header file names: ', \, /*, //, or " .	
54	Implementation files <b>will</b> 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 <b>will</b> be declared in that order.	
58	When declaring and defining functions with more than two parameters, the leading parenthesis and the first argument <b>will</b> 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 <b>shall</b> always be enclosed in braces, even if the braces form an empty block.	<p>Messages in log file:</p> <ul style="list-style-type: none"> <li>• The statements forming the body of an if statement shall always be enclosed in braces.</li> <li>• The statements forming the body of an else statement shall always be enclosed in braces.</li> <li>• The statements forming the body of a while statement shall always be enclosed in braces.</li> <li>• The statements forming the body of a do ... while statement shall always be enclosed in braces.</li> <li>• The statements forming the body of a for statement shall always be enclosed in braces.</li> </ul>
60	Braces ("{}") which enclose a block <b>will</b> 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 <b>will</b> 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.	<p>Reports when the following characters are not directly connected to a white space:</p> <ul style="list-style-type: none"> <li>• .</li> <li>• -&gt;</li> <li>• !</li> <li>• ~</li> <li>• -</li> <li>• ++</li> <li>• —</li> </ul> <hr/> <p><b>Note</b> A violation will be reported for "." used in float/double definition.</p> <hr/>

## Classes

N.	JSF++ Definition	Comments
67	Public and protected data <b>should</b> 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 <b>will</b> be performed through the member initialization list rather than through assignment in the body of a constructor.	<p>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:</p> <p>Initialization of nonstatic class members "&lt;field&gt;" will be performed through the member initialization list.</p>
75	Members of the initialization list <b>shall</b> be listed in the order in which they are declared in the class.	
76	A copy constructor and an assignment operator <b>shall</b> be declared for classes that contain pointers to data items or nontrivial destructors.	<p>Messages in log file:</p> <ul style="list-style-type: none"> <li>• no copy constructor and no copy assign</li> <li>• no copy constructor</li> <li>• no copy assign</li> </ul>
77.1	The definition of a member function <b>shall not</b> 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 <b>shall</b> define a virtual destructor.	
79	All resources acquired by a class shall be released by the class's destructor.	<p>Reports when the number of "new" called in a constructor is greater than the number of "delete" called in its destructor.</p> <hr/> <p><b>Note</b> A violation is raised even if "new" is done in a "if/else".</p> <hr/>



N.	JSF++ Definition	Comments
81	The assignment operator shall handle self-assignment correctly.	<p>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.</p> <p>A violation is raised when the if statement is followed by a statement other than the return statement.</p>
82	An assignment operator <b>shall</b> return a reference to <code>*this</code> .	<p>The following operators should return <code>*this</code> on method, and <code>*first_arg</code> on plain function.</p> <pre>operator= operator+= operator-= operator*= operator &gt;&gt;= operator &lt;&lt;= operator /= operator %= operator  = operator &amp;= operator ^= prefix operator++ prefix operator--</pre> <p>Does not report when no return exists.</p> <p>No special message if type does not match.</p> <p>Messages in log file:</p> <ul style="list-style-type: none"> <li>• An assignment operator shall return a reference to <code>*this</code>.</li> <li>• An assignment operator shall return a reference to its first arg.</li> </ul>

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 <b>shall</b> only be allowed in the following restricted form: n interfaces plus m private implementations, plus at most one protected implementation.	<p>Messages in log file:</p> <ul style="list-style-type: none"> <li>• Multiple inheritance on public implementation shall not be allowed: <code>&lt;public_base_class&gt;</code> is not an interface.</li> <li>• Multiple inheritance on protected implementation shall not be allowed : <code>&lt;protected_base_class_1&gt;</code></li> <li>• <code>&lt;protected_base_class_2&gt;</code> are not interfaces.</li> </ul>
88.1	A stateful virtual base <b>shall</b> be explicitly declared in each derived class that accesses it.	
89	A base class <b>shall not</b> be both virtual and non-virtual in the same hierarchy.	
94	An inherited nonvirtual function <b>shall not</b> be redefined in a derived class.	<p>Does not report for destructor. Message in log file:</p> <p>Inherited nonvirtual function %s shall not be redefined in a derived class.</p>
95	An inherited default parameter <b>shall never</b> be redefined.	
96	Arrays <b>shall not</b> 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 <b>shall not</b> 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 !=) <b>shall</b> 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 <code>main()</code> , <b>should</b> be placed in some namespace.	
99	Namespaces <b>will not</b> be nested more than two levels deep.	

## Templates

N.	JSF++ Definition	Comments
104	A template specialization <b>shall</b> be declared before its use.	Reports the actual compilation error message.

## Functions

N.	JSF++ Definition	Comments
107	Functions <b>shall</b> always be declared at file scope.	
108	Functions with variable numbers of arguments <b>shall not</b> 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 <b>will not</b> be used.	
111	A function <b>shall not</b> return a pointer or reference to a non-static local object.	Simple cases without alias effect detected.
113	Functions <b>will</b> have a single exit point.	Reports first return, or once per function.
114	All exit points of value-returning functions <b>shall</b> be through return statements.	
116	Small, concrete-type arguments (two or three words in size) <b>should</b> be passed by value if changes made to formal parameters should not be reflected in the calling function.	Report constant parameters references with <code>sizeof &lt;= 2 * sizeof(int)</code> . Does not report for copy-constructor.
119	Functions <b>shall</b> 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 <b>should</b> 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 ( <code>//</code> ) 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 <b>shall not</b> 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.	<p>Reports when:</p> <ul style="list-style-type: none"> <li>• A global variable is used in only one function.</li> <li>• A local variable is not used in a statement (<code>expr</code>, <code>return</code>, <code>init ...</code>) of the same level of its declaration (in the same block) or is not used in two sub-statements of its declaration.</li> </ul> <hr/> <p><b>Note</b></p> <ul style="list-style-type: none"> <li>• Non-used variables are reported.</li> <li>• Initializations at definition are ignored (not considered an access)</li> </ul> <hr/>

N.	JSF++ Definition	Comments
137	All declarations at file scope should be static where possible.	
138	Identifiers <b>shall not</b> 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 <b>shall not</b> be used.	
141	A class, structure, or enumeration <b>will not</b> be declared in the definition of its type.	

### Initialization

N.	JSF++ Definition	Comments
142	All variables <b>shall</b> be initialized before use.	Done with <b>NIV</b> and <b>LOCAL_NIV</b> checks in the Verifier.
144	Braces <b>shall</b> 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 <b>shall not</b> 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 <b>shall not</b> 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) <b>shall not</b> be used.	
150	Hexadecimal constants <b>will</b> be represented using all uppercase letters.	
151	Numeric values in code <b>will not</b> 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 <b>shall not</b> be allowed on the same line.	

## Unions and Bit Fields

N.	JSF++ Definition	Comments
153	Unions <b>shall not</b> be used.	
154	Bit-fields <b>shall</b> have explicitly unsigned integral or enumeration types only.	
156	All the members of a structure (or class) <b>shall</b> 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: <ul style="list-style-type: none"> <li>• The right hand operand of a &amp;&amp; operator shall not contain side effects.</li> <li>• The right hand operand of a    operator shall not contain side effects.</li> </ul>
158	The operands of a logical && or    <b>shall</b> be parenthesized if the operands contain binary operators.	Messages in log file: <ul style="list-style-type: none"> <li>• The operands of a logical &amp;&amp; shall be parenthesized if the operands contain binary operators.</li> <li>• The operands of a logical    shall be parenthesized if the operands contain binary operators.</li> </ul>



N.	JSF++ Definition	Comments
		Exception for: X    Y    Z , Z&&Y &&Z
159	Operators   , &&, and unary & <b>shall not</b> be overloaded.	Messages in log file: <ul style="list-style-type: none"> <li>• Unary operator &amp; shall not be overloaded.</li> <li>• Operator    shall not be overloaded.</li> <li>• Operator &amp;&amp; shall not be overloaded.</li> </ul>
160	An assignment expression <b>shall</b> be used only as the expression in an expression statement.	Only simple assignment, not +=, ++, etc.
162	Signed and unsigned values <b>shall not</b> be mixed in arithmetic or comparison operations.	
163	Unsigned arithmetic <b>shall not</b> be used.	
164	The right hand operand of a shift operator <b>shall</b> 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 <b>shall not</b> have a negative value.	Detects constant case +. Verifier used for dynamic cases.
165	The unary minus operator <b>shall not</b> be applied to an unsigned expression.	
166	The sizeof operator <b>will not</b> be used on expressions that contain side effects.	
168	The comma operator <b>shall not</b> 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 <b>shall not</b> 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: <ul style="list-style-type: none"> <li>• the same object,</li> <li>• the same function,</li> <li>• members of the same object, or</li> <li>• elements of the same array (including one past the end of the same array).</li> </ul>	Reports when relational operator are used on pointer types (casts ignored).
173	The address of an object with automatic storage <b>shall not</b> be assigned to an object which persists after the object has ceased to exist.	
174	The null pointer <b>shall not</b> be de-referenced.	Done with <b>IDP</b> checks in Verifier.
175	A pointer <b>shall not</b> be compared to NULL or be assigned NULL; use plain 0 instead.	Reports usage of NULL macro in pointer contexts.
176	A typedef <b>will</b> 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 <b>should</b> be avoided.	<p>Reports user defined conversion function, non-explicit constructor with one parameter or default value for others (even undefined ones). Does not report copy-constructor.</p> <p>Additional message for constructor case: This constructor should be flagged as "explicit".</p>
178	<p>Down casting (casting from base to derived class) <b>shall</b> only be allowed through one of the following mechanism:</p> <ul style="list-style-type: none"> <li>• Virtual functions that act like dynamic casts (most likely useful in relatively simple cases).</li> <li>• Use of the visitor (or similar) pattern (most likely useful in complicated cases).</li> </ul>	Reports explicit down casting, <code>dynamic_cast</code> included. (No special case for visitor pattern.)
179	A pointer to a virtual base class <b>shall not</b> be converted to a pointer to a derived class.	Reports this specific down cast. Allows <code>dynamic_cast</code> .
180	Implicit conversions that may result in a loss of information <b>shall not</b> be used.	<p>Reports the following implicit casts :</p> <pre>integer =&gt; smaller integer unsigned =&gt; smaller or eq signed signed =&gt; smaller or eq un-signed integer =&gt; float float =&gt; integer</pre> <p>Does not report for cast to bool reports for implicit cast on constant done with the options <code>-detect-unsigned-overflows</code> or <code>-ignore-constant-overflows</code>.</p>
181	Redundant explicit casts <b>will not</b> be used.	Reports useless cast: <code>cast T to T</code> . Casts to equivalent typedefs are also reported.

N.	JSF++ Definition	Comments
182	Type casting from any type to or from pointers <b>shall not</b> be used.	Does not report when Rule 181 applies.
184	Floating point numbers <b>shall not</b> 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) <b>shall</b> be used instead of the traditional C-style casts.	

### Flow Control Standards

N.	JSF++ Definition	Comments
186	There <b>shall</b> be no unreachable code.	Done with gray checks in the Verifier.
187	All non-null statements <b>shall</b> potentially have a side-effect.	
188	Labels <b>will not</b> be used, except in switch statements.	
189	The goto statement <b>shall not</b> be used.	
190	The continue statement <b>shall not</b> be used.	
191	The break statement <b>shall not</b> 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.

<b>N.</b>	<b>JSF++ Definition</b>	<b>Comments</b>
193	Every non-empty case clause in a switch statement <b>shall</b> be terminated with a break statement.	
194	All switch statements that do not intend to test for every enumeration value <b>shall</b> contain a final default clause.	Reports only for missing default.
195	A switch expression <b>will</b> not represent a Boolean value.	
196	Every switch statement <b>will</b> have at least two cases and a potential default.	
197	Floating point variables <b>shall not</b> be used as loop counters.	Assumes 1 loop parameter.
198	The initialization expression in a for loop <b>will</b> 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 <b>will</b> 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 <b>will not</b> be used; a while loop will be used instead.	
201	Numeric variables being used within a <i>for</i> 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

<b>N.</b>	<b>JSF++ Definition</b>	<b>PolySpace Comments</b>
202	Floating point variables <b>shall not</b> be tested for exact equality or inequality.	Reports only direct equality/inequality. Check done for all expressions.

N.	JSF++ Definition	PolySpace Comments
203	Evaluation of expressions <b>shall not</b> lead to overflow/underflow.	Done with the <b>SCAL-OVFL</b> and <b>FLOAT-OVFL</b> checks in the Verifier.
204	<p>A single operation with side-effects shall only be used in the following contexts:</p> <ul style="list-style-type: none"> <li>• by itself</li> <li>• the right-hand side of an assignment</li> <li>• a condition</li> <li>• the only argument expression with a side-effect in a function call</li> <li>• condition of a loop</li> <li>• switch condition</li> <li>• single part of a chained operation</li> </ul>	<p>Reports when:</p> <ul style="list-style-type: none"> <li>• A side effect is found in a return statement</li> <li>• A side effect exists on a single value, and only one operand of the function call has a side effect.</li> </ul>
204.1	The value of an expression shall be the same under any order of evaluation that the standard permits.	<p>Reports when:</p> <ul style="list-style-type: none"> <li>• Variable is written more than once in an expression</li> <li>• Variable is read and write in sub-expressions</li> <li>• Volatile variable is accessed more than once</li> </ul> <hr/> <p><b>Note</b> Read-write operations such as ++, are only considered as a write.</p> <hr/>
205	The volatile keyword <b>shall not</b> 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) <b>shall not</b> occur after initialization.	Reports calls to C library functions: <code>malloc / calloc / realloc / free</code> and all <code>new/delete</code> operators in functions or methods.

## Fault Handling

N.	JSF++ Definition	Comments
208	C++ exceptions <b>shall not</b> be used.	Reports <code>try</code> , <code>catch</code> , <code>throw spec</code> , and <code>throw</code> .

## Portable Code

N.	JSF++ Definition	Comments
209	The basic types of <code>int</code> , <code>short</code> , <code>long</code> , <code>float</code> and <code>double</code> <b>shall not</b> be used, but specific-length equivalents should be <code>typedef</code> 'd accordingly for each compiler, and these type names used in the code.	Only allows use of basic types through direct <code>typedefs</code> .
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 <b>will not</b> be used.	Reports: <code>p + I</code> <code>p - I</code> <code>p++</code> <code>p--</code> <code>p+=</code> <code>p-=</code> Allows <code>p[i]</code> .

## Rules Not Checked

**In this section...**

“Code Size and Complexity” on page 12-37

“Rules” on page 12-37

“Environment” on page 12-37

“Libraries” on page 12-38

“Header Files” on page 12-38

“Style” on page 12-38

“Classes” on page 12-39

“Namespaces” on page 12-40

“Templates” on page 12-41

“Functions” on page 12-41

“Comments” on page 12-42

“Initialization” on page 12-42

“Types” on page 12-43

“Unions and Bit Fields” on page 12-43

“Operators” on page 12-43

“Type Conversions” on page 12-43

“Expressions” on page 12-43

“Memory Allocation” on page 12-44

“Portable Code” on page 12-44

“Efficiency Considerations” on page 12-44

“Miscellaneous” on page 12-45

“Testing” on page 12-45



## 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: <ul style="list-style-type: none"> <li>• approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)</li> </ul>
5	To break a “will” or a “shall” rule, the following approvals must be received by the developer: <ul style="list-style-type: none"> <li>• approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)</li> <li>• approval from the software product manager (obtained by the unit approval in the developmental CM tool)</li> </ul>
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.
69	<p>A member function that does not affect the state of an object (its instance variables) will be declared const.</p> <p>Member functions should be const by default. Only when there is a clear, explicit reason should the const modifier on member functions be omitted.</p>
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	<p>The invariant for a class should be:</p> <ul style="list-style-type: none"> <li>• A part of the postcondition of every class constructor,</li> <li>• A part of the precondition of the class destructor (if any),</li> <li>• A part of the precondition and postcondition of every other publicly accessible operation.</li> </ul>
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.

N.	JSF++ Definition
90	Heavily used interfaces should be minimal, general and abstract.
91	Public inheritance will be used to implement “is-a” relationships.
92	<p>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:</p> <ul style="list-style-type: none"> <li>• Preconditions of derived methods must be at least as weak as the preconditions of the methods they override.</li> <li>• Postconditions of derived methods must be at least as strong as the postconditions of the methods they override.</li> </ul> <p>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.</p>
93	“has-a” or “is-implemented-in-terms-of” relationships will be modeled through membership or non-public inheritance.

## Namespaces

N.	JSF++ Definition
100	<p>Elements from a namespace should be selected as follows:</p> <ul style="list-style-type: none"> <li>• using declaration or explicit qualification for few (approximately five) names,</li> <li>• using directive for many names.</li> </ul>

## Templates

N.	JSF++ Definition
101	Templates shall be reviewed as follows: <ol style="list-style-type: none"> <li><b>1</b> with respect to the template in isolation considering assumptions or requirements placed on its arguments.</li> <li><b>2</b> with respect to all functions instantiated by actual arguments.</li> </ol>
102	Template tests shall be created to cover all actual template instantiations.
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: <ul style="list-style-type: none"> <li>• <b>117.1</b> – An object should be passed as <code>const T&amp;</code> if the function should not change the value of the object.</li> <li>• <b>117.2</b> – An object should be passed as <code>T&amp;</code> if the function may change the value of the object.</li> </ul>
118	Arguments should be passed via pointers if NULL values are possible: <ul style="list-style-type: none"> <li>• <b>118.1</b> – An object should be passed as <code>const T*</code> if its value should not be modified.</li> <li>• <b>118.2</b> – An object should be passed as <code>T*</code> if its value may be modified.</li> </ul>
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.

<b>N.</b>	<b>JSF++ Definition</b>
124	Trivial forwarding functions should be inlined.
125	Unnecessary temporary objects should be avoided.

## Comments

<b>N.</b>	<b>JSF++ Definition</b>
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

<b>N.</b>	<b>JSF++ Definition</b>
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: <ol style="list-style-type: none"> <li data-bbox="246 1367 372 1397">1 by itself</li> <li data-bbox="246 1428 728 1458">2 the right-hand side of an assignment</li> <li data-bbox="246 1489 409 1519">3 a condition</li> </ol>

N.	JSF++ Definition
	<p><b>4</b> the only argument expression with a side-effect in a function call</p> <p><b>5</b> condition of a loop</p> <p><b>6</b> switch condition</p> <p><b>7</b> single part of a chained operation</p>

## 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

<b>N.</b>	<b>JSF++ Definition</b>
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

<b>N.</b>	<b>JSF++ Definition</b>
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.



# PolySpace C++ Add-in for Visual Studio

---

- “Overview” on page 13-2
- “Using PolySpace Software Within Visual Studio” on page 13-3
- “Verifying an Entire Project” on page 13-16

### Overview

This chapter describes how to use PolySpace Client for C/C++ software while integrated in the Microsoft® Visual C++® .NET (see the *PolySpace Installation Guide* for compatibility information).

The PolySpace C++ add-in for Visual Studio® provides automatic source code verification and bug detection in source code developed inside the Visual IDE. It includes the following main features:

- An automatic setting of PolySpace project configuration file derived from your Visual project settings.
- A direct launching of C++ file verification from Visual IDE.
- A report of PolySpace compilation findings back to the IDE.

## Using PolySpace Software Within Visual Studio

In this section...
“Overview” on page 13-3
“PolySpace Parameters Within Visual Studio” on page 13-3
“Performing a File Verification from Visual Studio” on page 13-7
“Verifying Classes” on page 13-11
“The Configuration File and Default Options” on page 13-14

### Overview

The PolySpace for Visual .NET plug-in allows you to launch C++ verifications inside the Visual C++® IDE whether those verifications are local or remote.

Launching options are integrated within the Visual editor through a PolySpace menu and a toolbox.

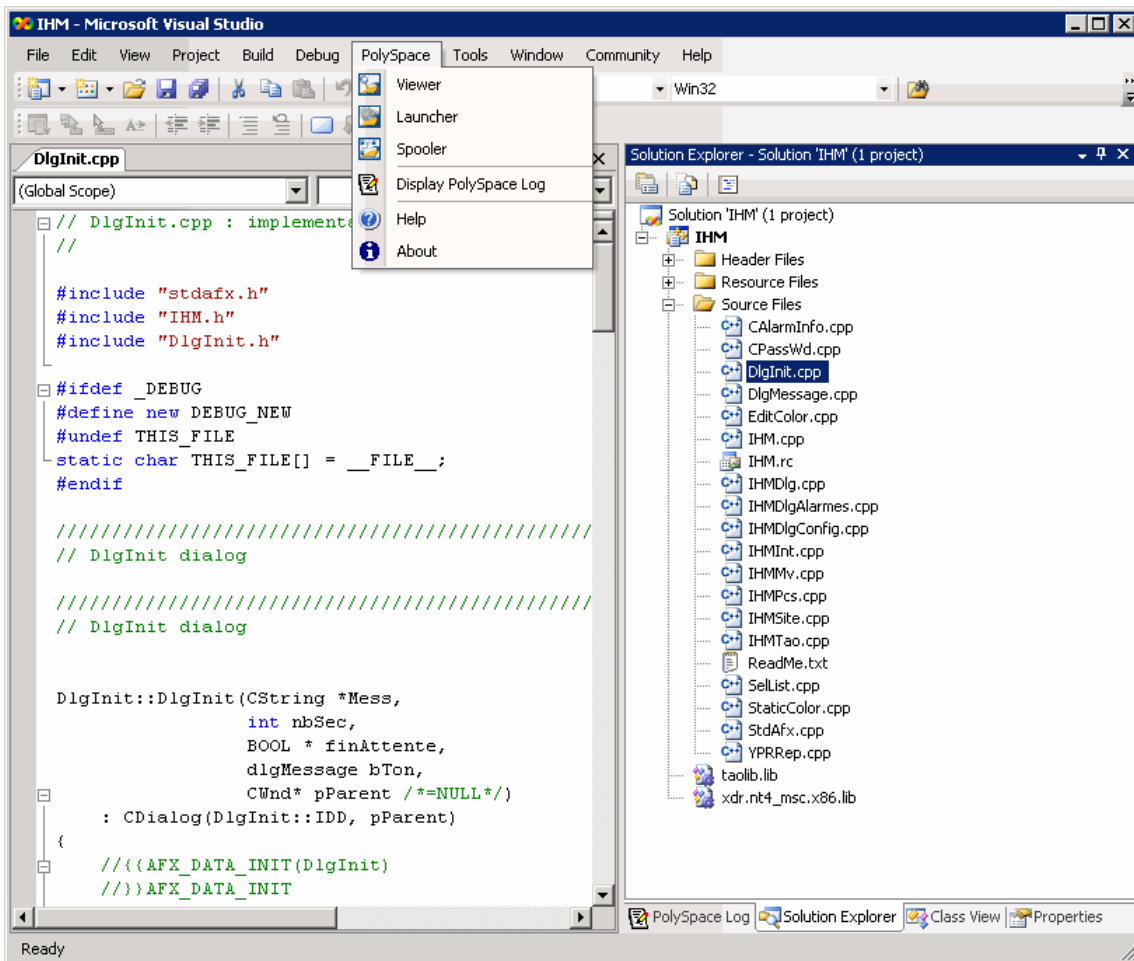
---

**Note** Some components of the plug-in are not automatically docked at installation. They must be manually moved where user wants them. Next time the interface will open, the components will be at the same place.

---

## PolySpace Parameters Within Visual Studio

When the PolySpace Visual C++ plug-in has been installed, a Polyspace menu and a PolySpace Log tab are displayed inside the Visual Studio IDE. Those tools are used to start local or remote verification on current classes and files of a Visual Studio C++ project (see next figure) without getting off your own development environment.



**PolySpace Menu and PolySpace Log Tab in Visual Studio®**

## PolySpace Menu

- Select **PolySpace > Launcher** to open the PolySpace launcher on the **last** configuration file updated in Visual.

---

**Note** The consistency is not checked with the current project and a warning message is always displayed. The “cfg” file could not correspond to “cfg” file of the current project.

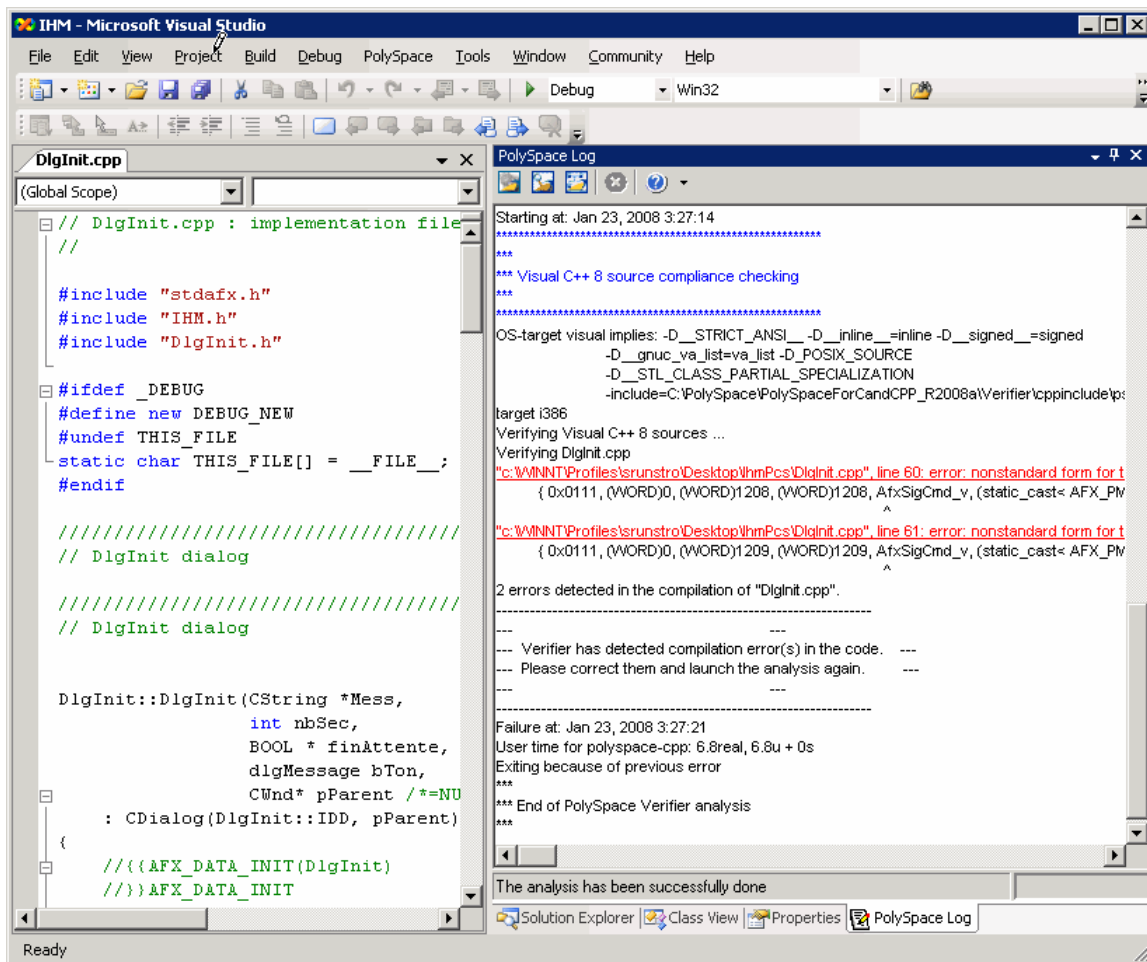
---

- Select **PolySpace > Spooler** to start the PolySpace spooler. This tool is used to manage PolySpace jobs that are performed on remote servers.  
For more information, see “Running Verifications on PolySpace Server” on page 7-3.
- Select **PolySpace > Viewer** to open the PolySpace Viewer with the **last** available results. If the verification has been done on the server, downloading them first is required before clicking on this button.
- Select **PolySpace > Display PolySpace Log** to view the progress of a verification (see below).
- Select **PolySpace > Help** to open the *PolySpace Client/Server for C++ User Guide* (PDF format).
- Select **PolySpace > About** to display the release number of the PolySpace for Visual Studio plug-in.

## PolySpace Log Tab

After you launch a verification, the **PolySpace Log** displays a progress report. Compilation errors are highlighted as links. Click a link to display the file and line number that includes the error.

Click the X in the PolySpace Log toolbar to stop the verification. For a server verification, this option works only during the compilation phase — before it is sent to the server. However, you can select **PolySpace > Spooler** and stop the verification from the spooler dialog.

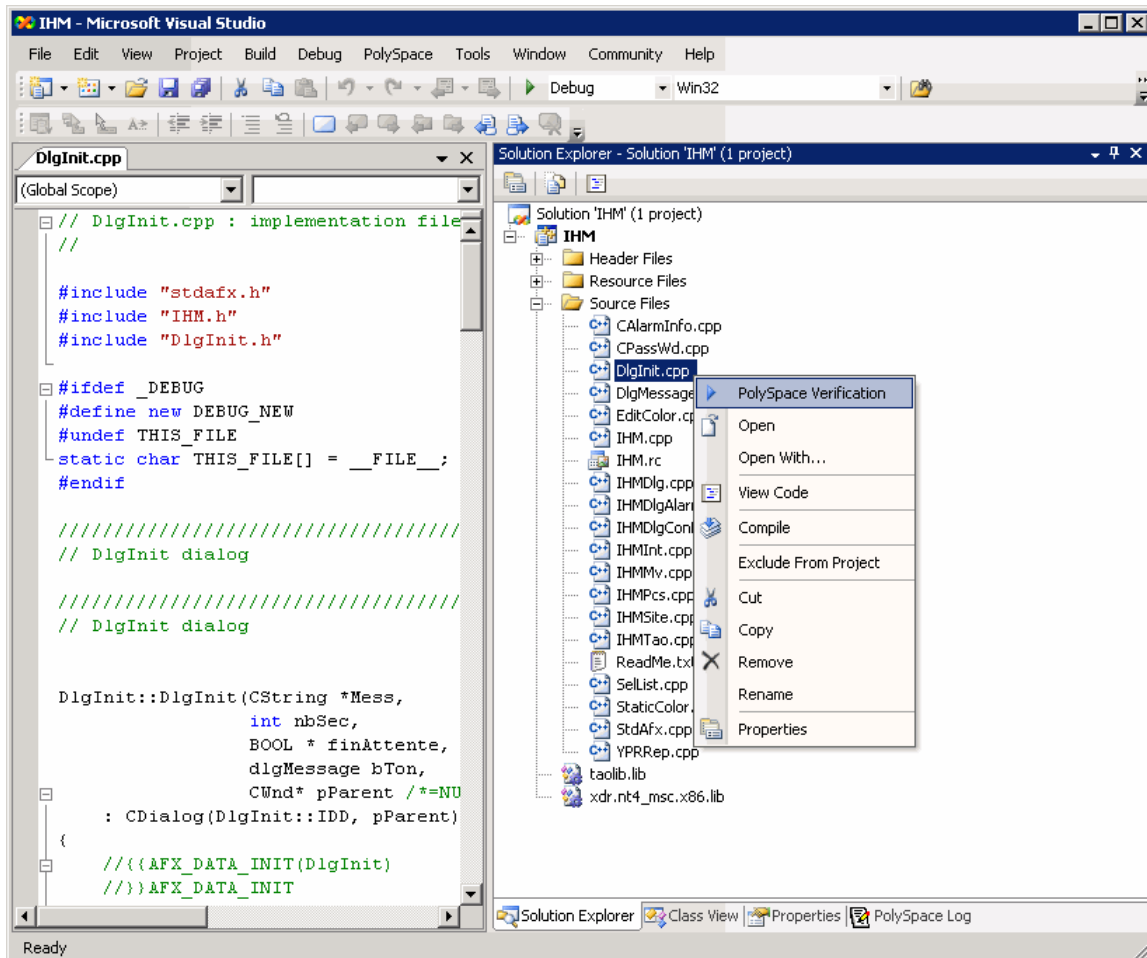


**The PolySpace® Log Window**



## PolySpace Right-Click Menu

You can right-click on files in the Solution Explorer to perform PolySpace Verification (see picture below).



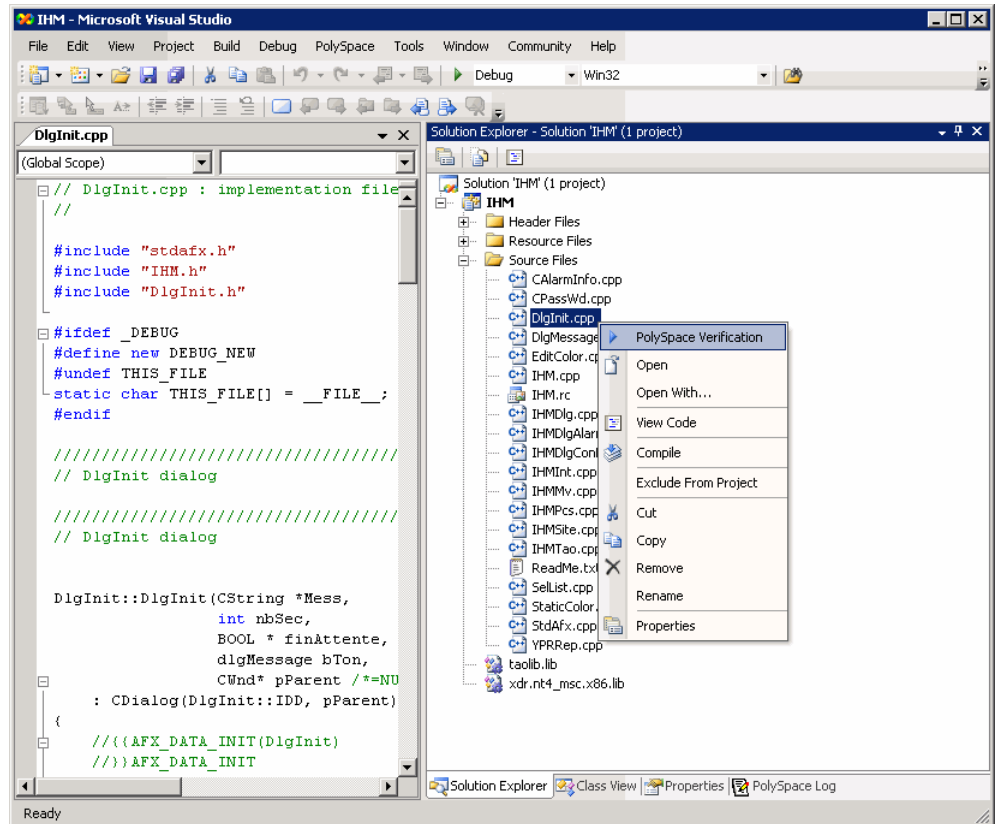
### PolySpace® Verification Menu Option

Select a file, then right-click and select PolySpace Verification to launch a verification on that file.

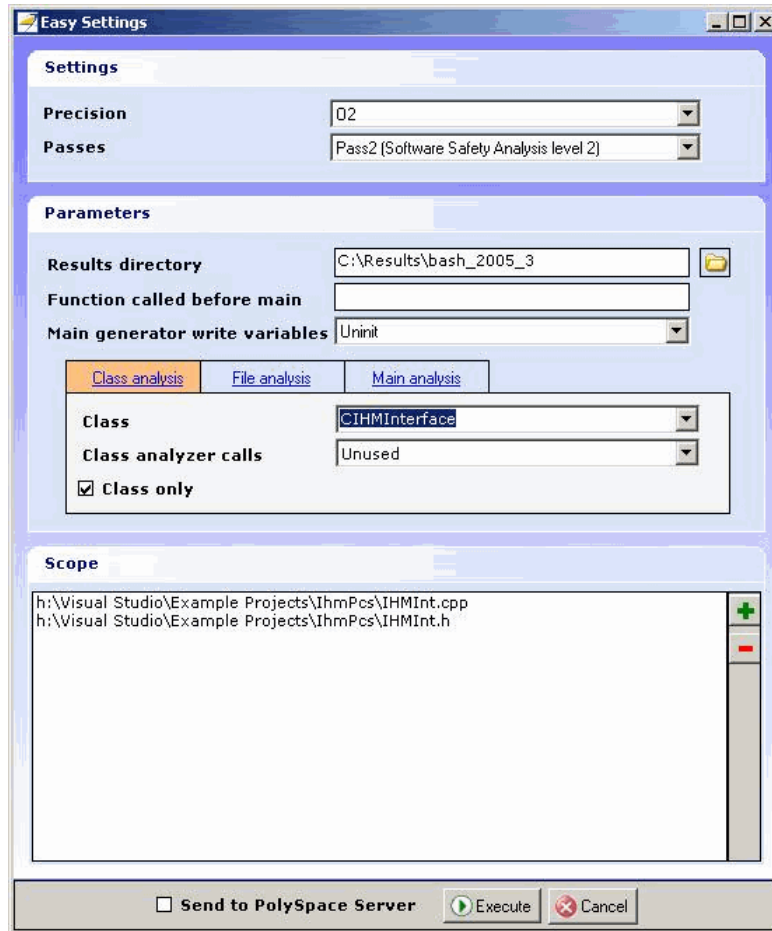
## Performing a File Verification from Visual Studio

To start your first verification:

- 1** Select **File > New > Project > New > Project Console Win32** to create a new project space
- 2** Enter the name `CppExample`, then save it in an appropriate location.  
For example: `C:\PolySpace\Visual`.  
  
Some files and a Project Console Win32 are created.
- 3** Select the **Browse the solution** tab.
- 4** Right-click the project name, then select **Add > Add existing element** from the pop-up menu.
- 5** Add `matrix.cpp` and `matrix.h` (located in `<PolySpaceProduct>/Examples/Demo_Cpp_Long/sources`) to the `CppExample` project .
- 6** In the Visual Studio Solution Explorer, select the `matrix.cpp` file.
- 7** Right-click `matrix.cpp` file, then select **PolySpace Verification**.



The Easy Settings dialog box opens, allowing you to set parameters including precision of the current verification and a result folder (See next figure).



### Easy Settings Window

- 8 Select the basic options for the current class verification:
  - **Settings** window allows you to select precision (-o) and level of verification (-t0);
  - **Parameters** window allows you to select:
    - **Results directory** – (-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 behavior of the analysis (`-class-only` and `-class-analyzer`).
  - **File analysis** tab – Allows you to choose a file analysis with associated option (`-main-generator-calls`).
  - **Main analysis** tab – Allows you to choose a partial integration analysis by choosing the name of the “main” (`-main`).
  - **Scope** window allows you to modify the list of files and classes to verify.
- 9 Click **Execute** to start the verification.

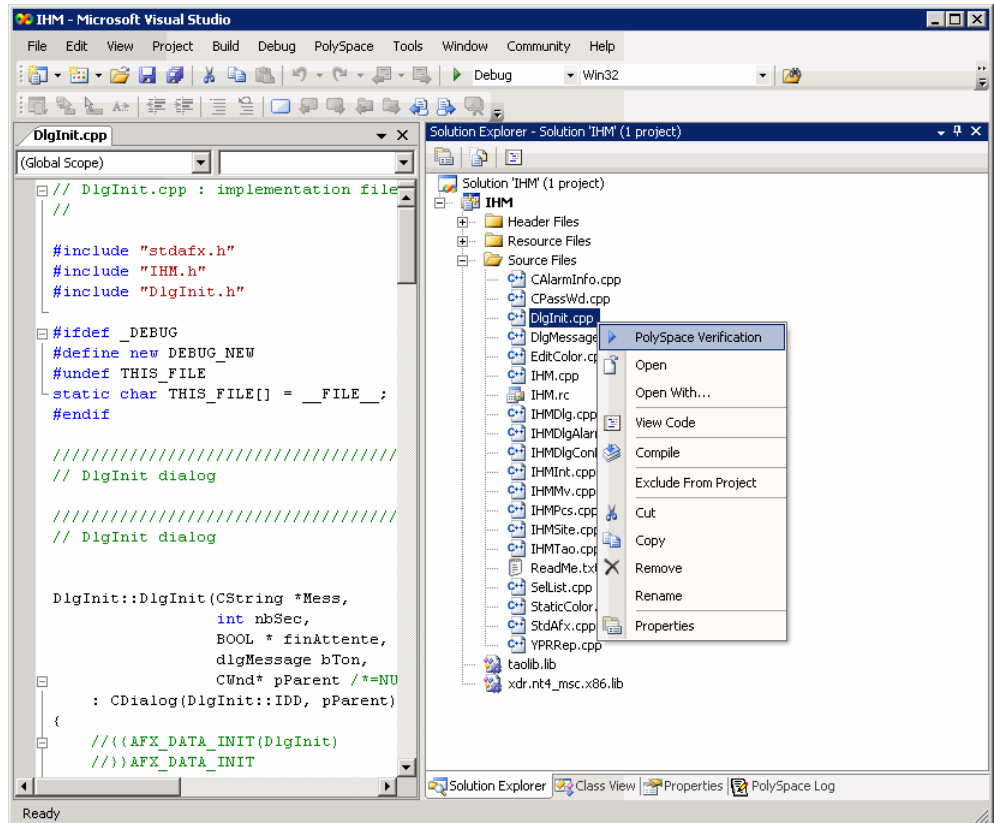
You can follow the progress of the verification in the **PolySpace Log** tab, and later using the PolySpace Spooler if remote launching has been enabled.

## Verifying Classes

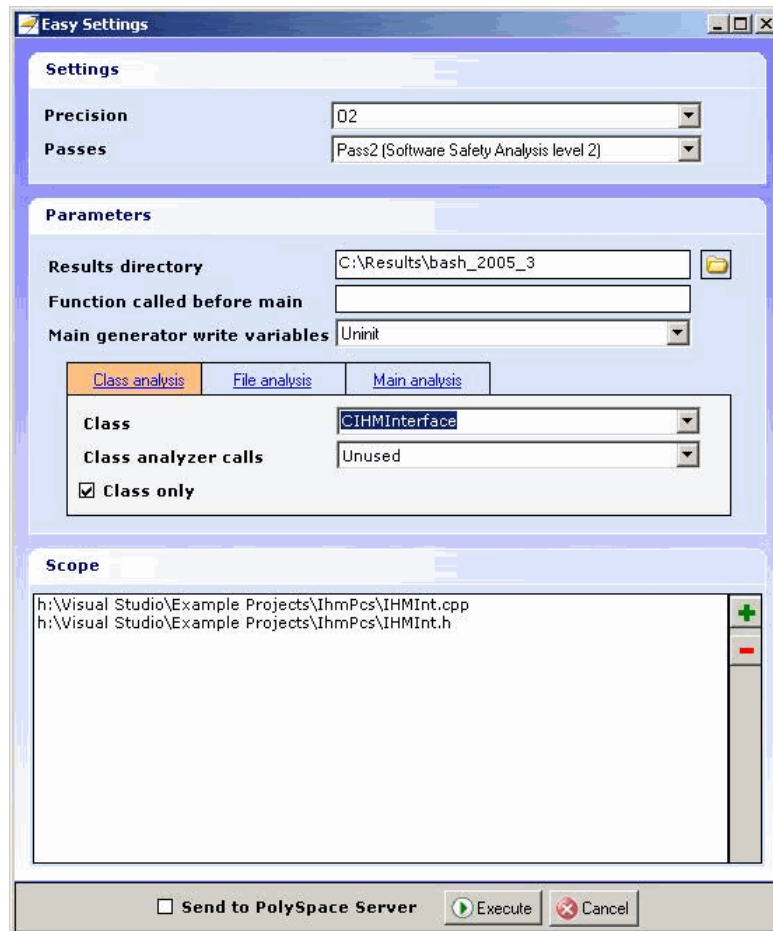
You can verify a C++ class by modifying the scope option in the Easy Settings dialog box.


To verify a class:

- 1 In the Visual Studio Solution Explorer, right-click a file, then select **PolySpace Verification**.

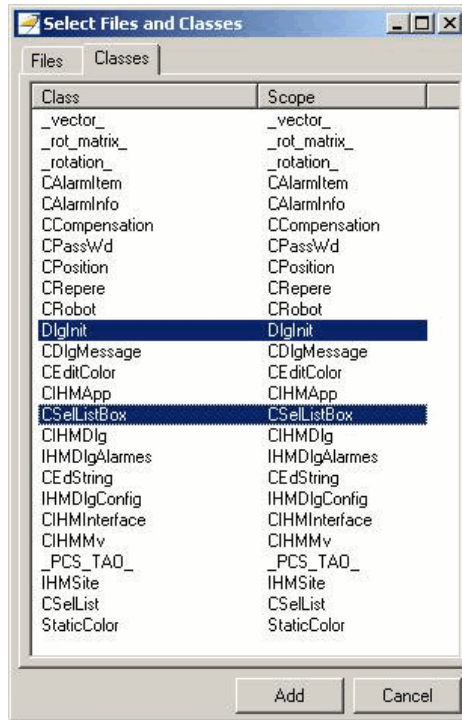


The Easy Settings dialog box opens.



2 Click  in the Scope window.

The Select Files and Classes dialog box appears.



- 3 Select the classes you want to analyze, then click **Add**.
- 4 Click **Execute** in the Easy Settings dialog box to start the verification.

You can follow the progress of the verification in the **PolySpace Log** tab, and later using the PolySpace Spooler if remote launching has been enabled.



## The Configuration File and Default Options

Some options are set by default and some others are directly extracted from the Visual project and set in the associated PolySpace configuration file.

- The list of Visual options extracted from the project file is:

Visual 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]

- Sources and includes directories (-I) are also extracted automatically from Visual options.
- Default options passed to the kernel depends of the Visual Studio release:  
-dialect Visual7.1 (or -dialect visual8) -OS-target Visual -target i386  
-desktop

Standard PolySpace options can be set by clicking on the “Launcher” right click menu (or from the PolySpace menu).

It starts the standard graphical interface polspace-launcher on a particular PolySpace configuration file (with .cfg extension).

Every option selected, will be taken into account during the verification, except the list of options set in the “PolySpace Basic Setting [C++]” window.

### Verifying an Entire Project

The launching of PolySpace on an entire project can only be made through the PolySpace Launcher using the “Launcher” command. In this case, the option `-main` must be set manually.

# Code Verification for Eclipse IDE

---

- “Overview” on page 14-2
- “Using PolySpace Software Within Eclipse IDE” on page 14-3

### Overview

The PolySpace Client is integrated with the Eclipse Integrated Development Environment (IDE) through the PolySpace C/C++ plug-in for Eclipse IDE (for compatibility information, see “PolySpace Plug-In Requirements” in the *PolySpace Installation Guide*).

This plug-in provides PolySpace source code verification and bug detection functionality for source code developed within Eclipse IDE. Features include the following:

- A contextual menu that allows you to launch a verification of one or more files.
- Views in the Eclipse editor that allow you to set verification parameters and monitor verification progress.

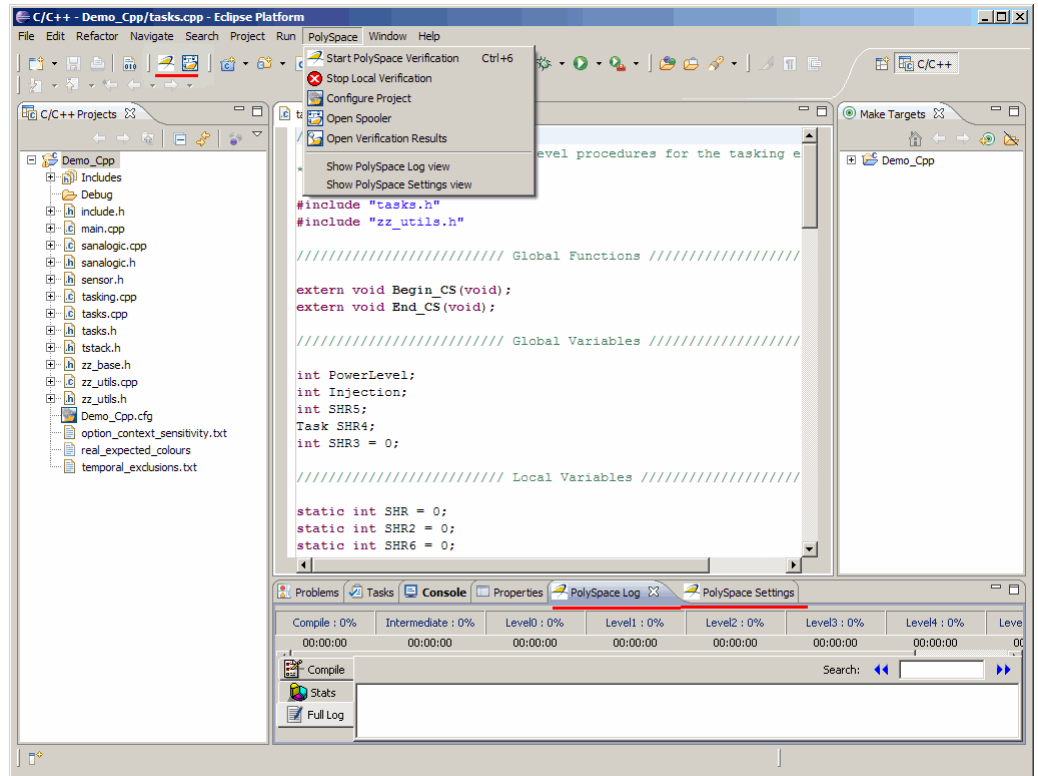
## Using PolySpace Software Within Eclipse IDE

In this section...
“PolySpace Features in the Eclipse Editor” on page 14-3
“Verifying Files from Eclipse IDE” on page 14-5

### **PolySpace Features in the Eclipse Editor**

Once the PolySpace C/C++ plug-in for Eclipse IDE is installed, the following are available in the Eclipse editor:

- A PolySpace menu
- Buttons on the toolbar to launch a verification and open the PolySpace spooler
- PolySpace Log and Setting views



## Eclipse Editor with PolySpace®

### PolySpace Menu

From this drop-down menu, you can select:

- **PolySpace > Start PolySpace Verification** to launch the verification of the files selected within the current project.
- **PolySpace > Stop Local Verification** to halt a verification that is in progress.
- **PolySpace > Configure Project** to set up or modify a PolySpace project configuration.
- **PolySpace > Open Spooler** to start the PolySpace spooler. This tool is used to manage PolySpace jobs that are performed on remote servers.

For more information, see “Running Verifications on PolySpace Server” on page 7-3.

- **PolySpace > Open Verification Results** to open the PolySpace Viewer with the **last** available results. If the verification has been done on the server, you should download the results first.
- **PolySpace > Show PolySpace Log view** to observe the progress of a verification.
- **PolySpace > Settings view** to view or modify verification parameters.

## Verifying Files from Eclipse IDE

To start a verification using the Eclipse editor:

- 1** Select a project within the **C/C++ Projects** view. If your source files do not belong to an Eclipse project, then you can create one using the Eclipse editor:
  - a** Select **File > New > C++ Project**.
  - b** Ensure that the **Use default location** check box is not selected.
  - c** Using **Browse**, navigate to the folder containing your source files, for example, `C:\Test\Source_cpp`.
  - d** In the **Project name** field, enter a name, for example, `Demo_Cpp`.
  - e** Click **Finish**. An Eclipse project is created.
- 2** In the **PolySpace Settings** view, click **Advanced** and specify the following Target/Compilation options:
  - `-I`
  - `-OS-target`
  - `-D`
  - `-dialect`

In addition, if you want to verify a class, you must specify its name:

- a** Under **PolySpace inner settings**, expand **Generate a main for a given class**, and select the corresponding check box.

- b** Enter the class name in the **Value** field. This specifies the `-class-analyzer` option.

For information about these options, see “Options Description” in the PolySpace Products for C++ Reference.

- 3** Within the **C/C++ Projects** view, select the file(s) that you want to verify.
- 4** Either right-click and select **Start PolySpace Verification**, or select **PolySpace > Start PolySpace Verification**.

You can follow the progress of a verification in the **PolySpace Log** view. If an error or warning is produced, you can double-click it to go to the corresponding location in the source code.

Use the Viewer (**PolySpace > Open Verification Results**) to examine the results of the verification.



**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.



## A

- active project
  - definition 11-3
  - setting 11-3
- analysis options 4-15
  - generic targets 4-26
  - JSF++ compliance 4-19
- assistant mode
  - criterion 9-20
  - custom methodology 9-23
  - methodology 9-20
  - methodology for C 9-20
  - methodology for C++ 9-21
  - overview 9-19
  - reviewing checks 9-24
  - selection 9-19
  - use 9-19 9-24

## C

- call graph 9-28
- call tree view 9-11
- calling sequence 9-28
- cfg. *See* configuration file
- client 1-6 7-2
  - installation 1-6
  - verification on 7-19
- Client
  - overview 1-6
- coding review progress view 9-11 9-29
- color-coding of verification results 1-3 9-13
- compile
  - log 8-6
- compile log
  - Launcher 7-21
  - Spooler 7-5
- compile phase 7-3
- compliance
  - JSF++ 1-2 4-19
- composite filters 9-34

- configuration file
  - definition 4-2
- custom methodology
  - definition 9-23

## D

- default directory
  - changing in preferences 4-6
- desktop file
  - definition 4-2
- directories
  - includes 4-10 4-12 4-14
  - results 4-10 4-12 4-14
  - sources 4-10 4-12 4-14
- downloading
  - results 9-8
- dsk. *See* desktop file

## E

- expert mode
  - filters 9-33
    - composite 9-34
    - individual 9-33
  - overview 9-26
  - selection 9-26
  - use 9-26

## F

- files
  - includes 4-10 4-12 4-14
  - results 4-10 4-12 4-14
  - source 4-10 4-12 4-14
- filters 9-33
  - alpha 9-34
  - beta 9-34
  - custom
    - modification 9-34 to 9-35
    - use 9-34 to 9-35

- gamma 9-34
- individual 9-33
- user def 9-34

## G

- generic target processors
  - adding 4-25
  - definition 4-26
  - deleting 4-28

## H

- hardware requirements 8-2
- help
  - accessing 1-8

## I

- installation
  - PolySpace Client for C/C++ 1-6
  - PolySpace products 1-6
  - PolySpace Server for C/C++ 1-6

## J

- JSF C++ compliance
  - file exclusion 4-22 12-6
  - rules file 4-20 12-4
- JSF++ compliance 1-2
  - analysis option 4-19
  - checking 4-19
  - log 12-8

## L

- Launcher
  - monitoring verification progress 7-21
  - opening 4-3
  - starting verification on client 7-19
  - starting verification on server 7-3

- viewing logs 7-21
- window 4-3
  - overview 4-3
  - progress bar 7-21

- licenses

- obtaining 1-6

- logs

- compile

- Launcher 7-21

- Spooler 7-5

- full

- Launcher 7-21

- Spooler 7-5

- stats

- Launcher 7-21

- Spooler 7-5

- viewing

- Launcher 7-21

- Spooler 7-5

## M

- methodology for C 9-20
- methodology for C++ 9-21

## P

- PolySpace Client

- overview 1-6

- PolySpace Client for C/C++

- installation 1-6

- license 1-6

- PolySpace In One Click

- active project 11-3

- overview 11-2

- sending files to PolySpace software 11-5

- starting verification 11-5

- use 11-2

- PolySpace products for C++

- components 1-6

- installation 1-6
    - licenses 1-6
    - overview 1-2
    - related products 1-6
    - user interface 1-6
  - PolySpace project model file
    - creation 4-25
    - definition 4-25
    - use 4-24
  - PolySpace Queue Manager Interface. *See* Spooler
  - PolySpace Server
    - overview 1-6
  - PolySpace Server for C/C++
    - installation 1-6
    - license 1-6
  - ppm. *See* PolySpace project model file
  - preferences
    - Launcher
      - default directory 4-6
      - default server mode 7-3
      - generic targets 4-25
      - server detection 8-3
    - Viewer
      - assistant configuration 9-21
      - display columns in RTE view 9-30
  - procedural entities view 9-11
    - reviewed column 9-30
  - product overview 1-2
  - progress bar
    - Launcher window 7-21
  - project
    - creation 4-2
    - definition 4-2
    - directories
      - includes 4-3
      - results 4-3
      - sources 4-3
    - file types
      - configuration file 4-2
      - desktop file 4-2
      - PolySpace project model file 4-2
      - saving 4-17
  - project model file. *See* PolySpace project model file
- R**
- related products 1-6
    - PolySpace products for linking to Models 1-7
    - PolySpace products for verifying Ada code 1-7
    - PolySpace products for verifying C code 1-7
  - reports
    - generation 9-37
  - results
    - directory 4-10 4-12 4-14
    - downloading from server 9-8
    - opening 9-11
    - report generation 9-37
  - reviewed column 9-30
  - rte view. *See* procedural entities view
- S**
- selected check view 9-11
  - server 1-6 7-2
    - detection 8-3
    - information in preferences 8-3
    - installation 1-6 8-3
    - verification on 7-3
  - Server
    - overview 1-6
  - source code view 9-11
  - Spooler
    - monitoring verification progress 7-5
    - removing verification from queue 9-8
    - use 7-5
    - viewing log 7-5

**T**

troubleshooting failed verification 8-2

**V**

variables view 9-11

verification

Ada code 1-7

C code 1-7

C++ code 1-2

client 7-2

compile phase 7-3

failed 8-2

monitoring progress

Launcher 7-21

Spooler 7-5

phases 7-3

results

color-coding 1-3

opening 9-11

report generation 9-37

reviewing 9-8

running 7-2

running on client 7-19

running on server 7-3

starting

from Launcher 7-2 7-19

from PolySpace In One Click 7-2 11-5

stopping 7-22

troubleshooting 8-2

with JSF C++ checking 12-7

Viewer

modes

selection 9-15

opening 9-11

window

call tree view 9-11

coding review progress view 9-11

overview 9-11

procedural entities view 9-11

selected check view 9-11

source code view 9-11

variables view 9-11