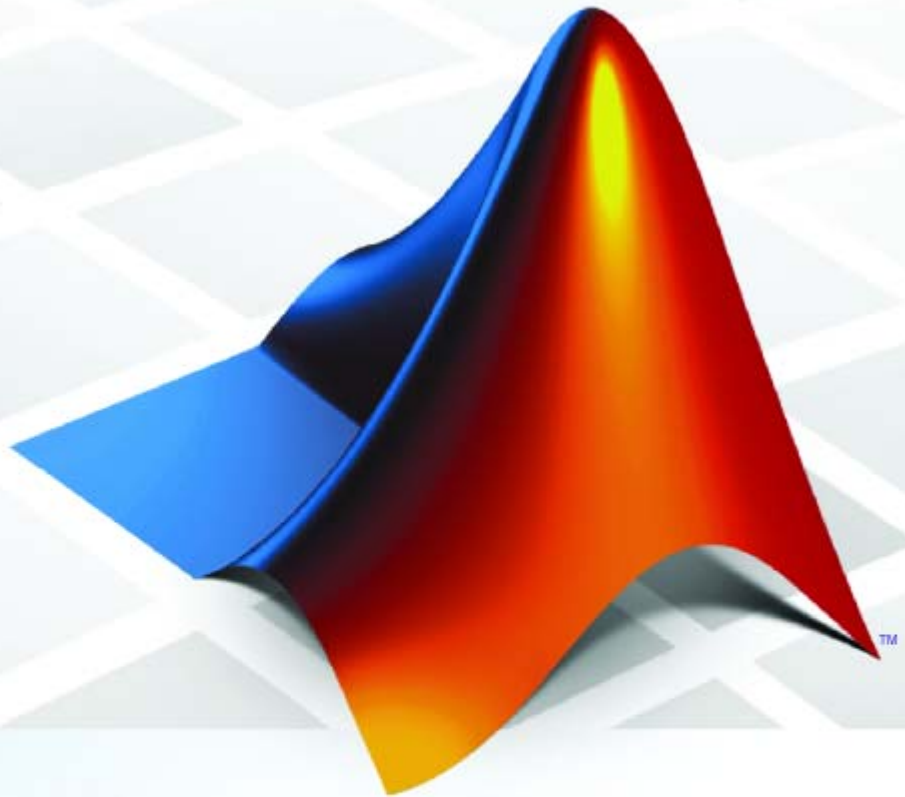


PolySpace® Products for C 7

User's Guide



How to Contact The MathWorks



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



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



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

PolySpace® Products for C User's Guide

© COPYRIGHT 1999–2009 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 for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

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

Introduction to PolySpace Products	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
PolySpace Documentation	1-8
About this Guide	1-8
Related Documentation	1-8

Choosing How to Use PolySpace Software

2

How to Use This Chapter	2-2
Applying PolySpace Verification to Your Development Process	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

Creating a Project	3-2
What Is a Project?	3-2
Project Directories	3-3
Opening PolySpace Launcher	3-3
Specifying Default Directory	3-6
Creating New Projects	3-8
Opening Existing Projects	3-10
Specifying Source Files	3-10
Specifying Include Directories	3-13
Specifying Results Directory	3-15
Specifying Analysis Options	3-16
Configuring Text and XML Editors	3-16
Saving the Project	3-17
Setting Up Project to Check Coding Rules	3-19
PolySpace MISRA Checker Overview	3-19
Checking Compliance with MISRA C Coding Rules	3-19
Creating a MISRA C Rules File	3-20
Excluding Files from the MISRA C Checking	3-23
Setting Up Project for Generic Target Processors	3-25
Project Model Files	3-25
Creating Project Model Files	3-26
Viewing Existing Generic Targets	3-26
Defining Generic Targets	3-27
Deleting a Generic Target	3-30
Common Generic Targets	3-30
Creating a Configuration File from a PolySpace Project Model File	3-31
Setting up Project to Automatically Test Orange	
Code	3-33
PolySpace Automatic Orange Tester	3-33
Enabling the Automatic Orange Tester	3-33

Emulating Your Runtime Environment

4

Setting Up a Target	4-2
Target/Compiler Overview	4-2
Specifying Target/Compilation Parameters	4-2
Predefined Target Processor Specifications (size of char, int, float, double...)	4-3
Generic Target Processors	4-5
Compiling Operating System Dependent Code (OS-target issues)	4-5
Address Alignment	4-9
Ignoring or Replacing Keywords Before Compilation	4-10
Verifying Code That Uses KEIL or IAR Dialects	4-13
How to Gather Compilation Options Efficiently	4-19
Verifying an Application Without a “Main”	4-22
Main Generator Overview	4-22
Automatically Generating a Main	4-23
Manually Generating a Main	4-23
Applying Data Ranges to External Variables and Stub Functions (DRS)	4-25
Overview of Data Range Specifications (DRS)	4-25
Specifying Data Ranges	4-25
File Format	4-26
Variable Scope	4-28
Performing Efficient Module Testing with DRS	4-30
Reducing Oranges with DRS	4-31

Preparing Source Code for Verification

5

Stubbing	5-2
Stubbing Overview	5-2
Manual vs. Automatic Stubbing	5-2
The Stubbing Options PURE and WORST	5-6
The Default and Alternative Behavior for Stubbing	5-6
Function Pointer Cases	5-8

Stubbing Functions with a Variable Argument Number ..	5-8
Finding Bugs in <code>_polyspace_stdstubs.c</code>	5-9
Preparing Code for Variables	5-11
Assigning Ranges to Variables/Assert?	5-11
Checking Properties on Global Variables at Any Point:	
Global assert	5-12
Modeling Variable Values External to my Application	5-15
How are Variables Initialized?	5-16
Verifying Code with Undefined or Undeclared Variables	
and Functions	5-17
Preparing Code for Built-in Functions	5-19
Preparing Multitasking Code	5-20
PolySpace Software Assumptions	5-20
Modelling Synchronous Tasks	5-21
Modelling Interruptions and Asynchronous	
Events/Tasks/Threads	5-23
Are Interruptions Maskable or Preemptive by Default? ...	5-25
Shared Variables	5-27
Mailboxes	5-31
Atomicity (Can an Instruction be Interrupted by	
Another)	5-34
Priorities	5-35
Verifying “Unsupported” Code	5-37
Ignoring Assembly Code	5-37
Dealing with Backward “goto” Statements	5-43
Types Promotion	5-46

Running a Verification

6

Types of Verification	6-2
Running Verifications on PolySpace Server	6-3
Starting Server Verification	6-3

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

Troubleshooting Verification Problems

7

Verification Process Failed Errors	7-2
Overview	7-2
Hardware Does Not Meet Requirements	7-2
You Did Not Specify the Location of Included Files	7-2
PolySpace Software Cannot Find the Server	7-3
Limit on Assignments and Function Calls	7-4
Compile Errors	7-6
Overview	7-6
Examining the Compile Log	7-6
Syntax error	7-8
Undeclared identifier	7-8
No such file or directory	7-9

Compilation errors with keywords: @interrupt, @address(0xABCDEF)	7-9
Link Messages	7-12
Overview	7-12
Function: Wrong Argument Type	7-12
Function: Wrong Argument Number	7-13
Variable: Wrong Type	7-14
Variable: Signed/Unsigned	7-14
Variable: Different Qualifier	7-15
Variable: Array Against Variable	7-15
Variable: Wrong Array Size	7-16
Missing Required Prototype for varargs	7-16
Stubbing Errors	7-17
Errors when Compiling _polyspace_stdstubs.c	7-17
Errors when Creating Automatic Stubs	7-22
Intermediate Language Errors	7-25
Reducing Verification Time	7-27
How Far has the Verification Progressed? How Can I Predict the Duration?	7-27
An Ideal Application Size	7-29
Why Should there be an Optimum Size?	7-30
Switch the Antivirus Off	7-31
Tuning PolySpace Parameters	7-31
Selecting a Subset of Code	7-32
A Decision Algorithm to Speed-Up a Verification: Hints and Troubleshooting	7-37
What are the Benefits of these Methods?	7-42

Reviewing Verification Results

8

Before You Review PolySpace Results	8-2
Overview: Understanding PolySpace Results	8-2
Why Gray Follows Red and Green Follows Orange	8-3
What is the Message and What does it Mean?	8-4

What is the C Explanation?	8-5
Opening Verification Results	8-8
Downloading Results from Server to Client	8-8
Opening Verification Results	8-11
Exploring the Viewer Window	8-11
Selecting Viewer Mode	8-15
Setting Character Encoding Preferences	8-15
Reviewing Results in Assistant Mode	8-17
What Is Assistant Mode?	8-17
Switching to Assistant Mode	8-17
Selecting the Methodology and Criterion Level	8-18
Exploring Methodology for C	8-19
Defining a Custom Methodology	8-21
Reviewing Checks	8-22
Reviewing Results in Expert Mode	8-25
What Is Expert Mode?	8-25
Switching to Expert Mode	8-25
Selecting a Check to Review	8-25
Displaying the Calling Sequence	8-27
Tracking Review Progress	8-28
Making the Reviewed Column Visible	8-30
Filtering Checks	8-33
Types of Filters	8-33
Creating a Custom Filter	8-35
Generating Reports of Verification Results	8-37
Using PolySpace Results	8-41
Review Runtime Errors: Fix Red Errors	8-41
Review Dead Code Checks: Why Gray Code is Interesting	8-42
Selective Orange Review: Finding the Maximum Number of Bugs in One Hour	8-44
Exhaustive Orange Review at Unit Phase	8-46
Exhaustive Orange Review at Integration Phase	8-47
Integration Bug Tracking	8-49
How to Find Bugs in Unprotected Shared Data	8-50
Dataflow Verification	8-51
Data and Coding Rules	8-51

Potential Side Effect of a Red Error	8-51
PolySpace Remembers the Relationships Between Variables	8-53
Why There Might be 2 Distinct Colors in a while/for Statement.	8-54

Managing Orange Checks

9

Understanding Orange Checks	9-2
What is an Orange Check?	9-2
Sources of Orange Checks	9-3
Determining Cause of Orange Checks	9-5
Reducing Orange Checks in Your Results	9-6
Options to Reduce Orange Checks	9-6
Generic Objectives: A Balance Between Precision and Verification Time	9-7
Applying Coding Rules to Reduce Orange Checks	9-8
Varying the Precision Level	9-13
Applying Software Safety Level Wisely	9-14
Adding Precision Constraints at the Periphery Via Stubs	9-15
Describing Multitasking Behavior Properly	9-17
Tuning Advanced Parameters	9-18
Applying Data Ranges	9-19
Reviewing Orange Checks	9-20
Selective Orange Review	9-20
Performing a Selective Orange Review	9-21
Exhaustive Orange Review	9-22
Performing an Exhaustive Orange Review	9-23
Automatically Testing Orange Code	9-26
Automatic Orange Tester Overview	9-26
Before Using the Automatic Orange Tester	9-29
Launching the Automatic Orange Tester	9-31
Reviewing the Test Results	9-35
Refining Data Ranges	9-39

Saving and Reusing Your Configuration	9-43
Exporting Data Ranges for PolySpace Verification	9-44
Configuring Compiler Options	9-45
Technical Limitations	9-46

Day to Day Use

10

PolySpace In One Click Overview	10-2
Using PolySpace In One Click	10-3
PolySpace In One Click Workflow	10-3
Setting the Active Project	10-3
Launching Verification	10-5
Using the Taskbar Icon	10-9

MISRA Checker

11

PolySpace MISRA Checker Overview	11-2
Setting Up MISRA C Checking	11-4
Checking Compliance with MISRA C Coding Rules	11-4
Creating a MISRA C Rules File	11-5
Excluding Files from the MISRA C Checking	11-7
Configuring Text and XML Editors	11-8
Running a Verification with MISRA C Checking	11-10
Starting the Verification	11-10
Examining the MISRA C Log	11-11
Opening MISRA-C Report	11-12
Rules Supported	11-14
Language Extensions	11-15
Character Sets	11-15
Identifiers	11-16

Types	11-17
Constants	11-17
Declarations and Definitions	11-18
Initialization	11-20
Arithmetic Type Conversion	11-20
Pointer Type Conversion	11-24
Expressions	11-25
Control Statement Expressions	11-28
Control Flow	11-29
Switch Statements	11-31
Functions	11-32
Pointers and Arrays	11-33
Structures and Unions	11-33
Preprocessing Directives	11-34
Standard Libraries	11-37
runtime Failures	11-39
Rules Partially Supported	11-40
Environment	11-40
Language Extension	11-41
Identifier	11-42
Declarations and Definitions	11-42
Expressions	11-43
Control Statement Expressions	11-45
Control Flow	11-46
Functions	11-47
Pointers and Arrays	11-48
Preprocessing Directives	11-49
Rules Not Checked	11-51
Environment	11-51
Language Extensions	11-52
Documentation	11-52
Types	11-53
Functions	11-54
Pointers and Arrays	11-54
Structures and Unions	11-55
Standard Libraries	11-55

Overview	12-2
Using PolySpace Software Within Eclipse IDE	12-3
PolySpace Features in the Eclipse Editor	12-3
Verifying Files from Eclipse IDE	12-5

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 MISRA C® standards.¹

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.

1. MISRA and MISRA C are registered trademarks of MISRA Ltd., held on behalf of the MISRA Consortium.

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 verification works by performing upper approximations. In other words, the computed variation domain of any program variable is always a superset of its actual variation domain. The direct consequence is that no runtime error (RTE) item to be checked can be missed by PolySpace verification.

Product Components

The PolySpace products for verifying 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™ for C/C++ 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, MISRA C Checker, Report Generator, and Automatic Orange Tester 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™ for C/C++ 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, 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*** – 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

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

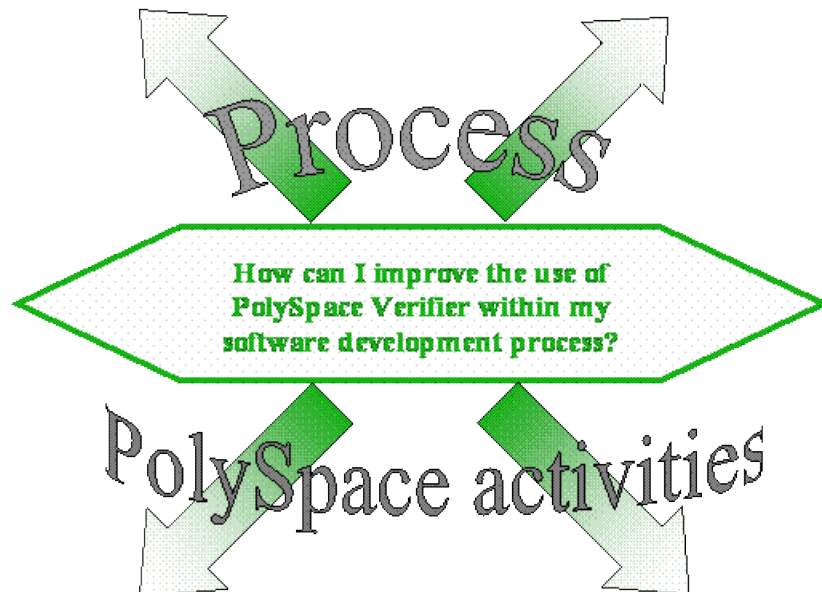
This chapter is designed for **Project managers, quality managers, and developers** who want to integrate PolySpace verification into their project development cycle. It explains how to apply PolySpace verification to each phase of the typical project lifecycle.

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?

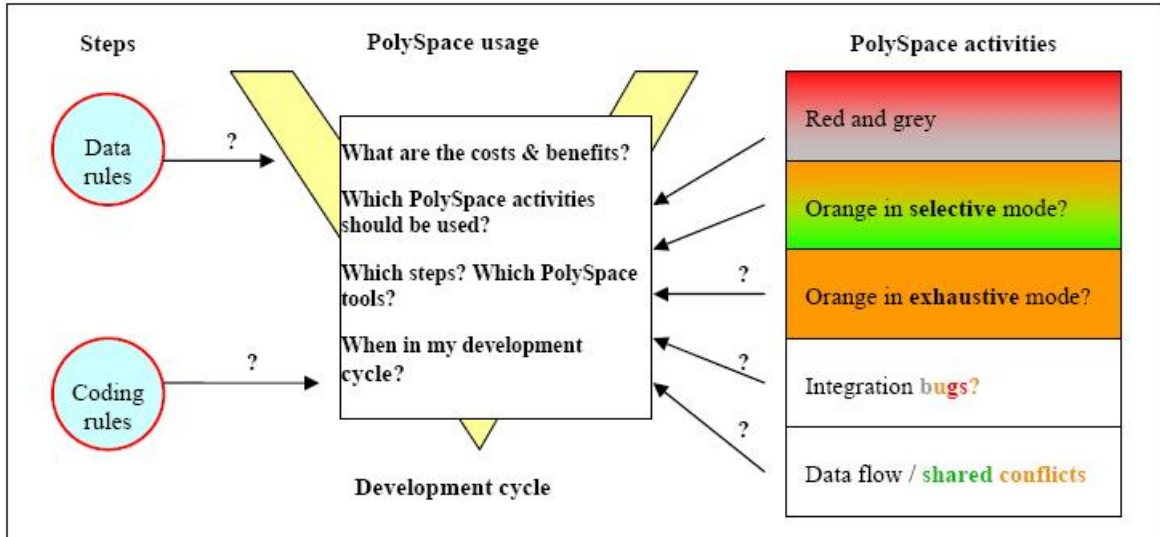
How can I change the process to get more out of PolySpace?



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:

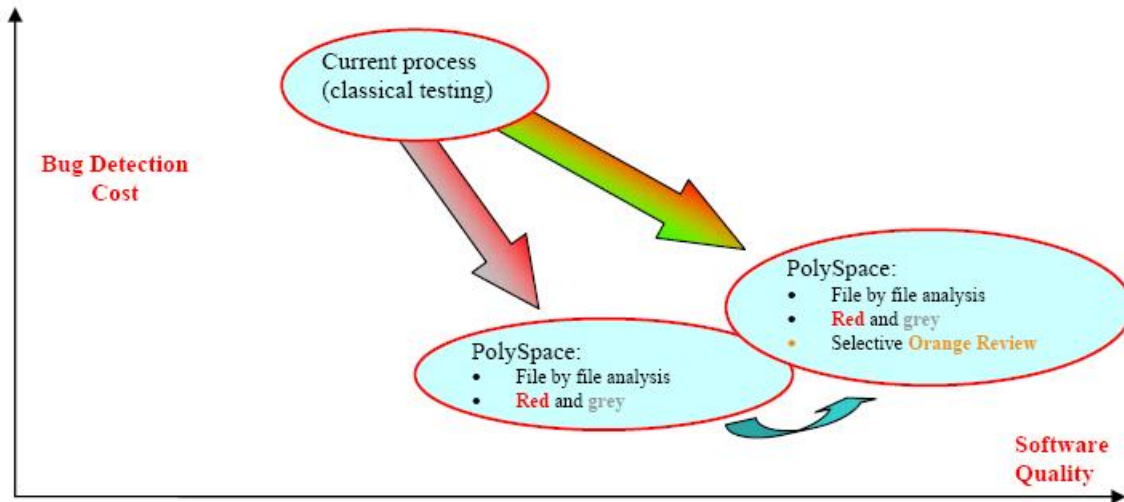
- Reducing the cost of testing and validation
- Improving software quality

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.

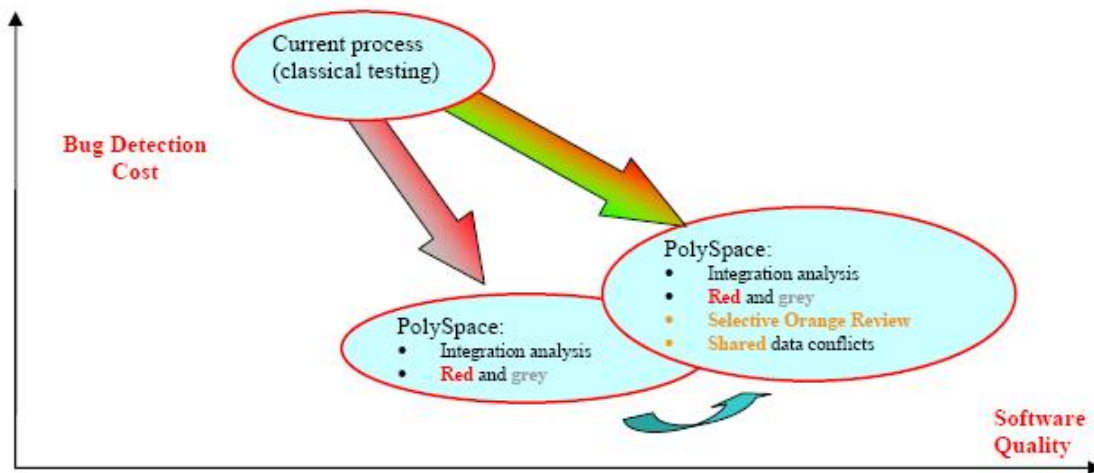
The second approach performs these activities, and adds a partial review of the orange warnings. The goal is to find as many bugs as possible in a limited amount of time. This approach finds more bugs, and therefore improves the overall quality of the software. It does involve more effort, but the amount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



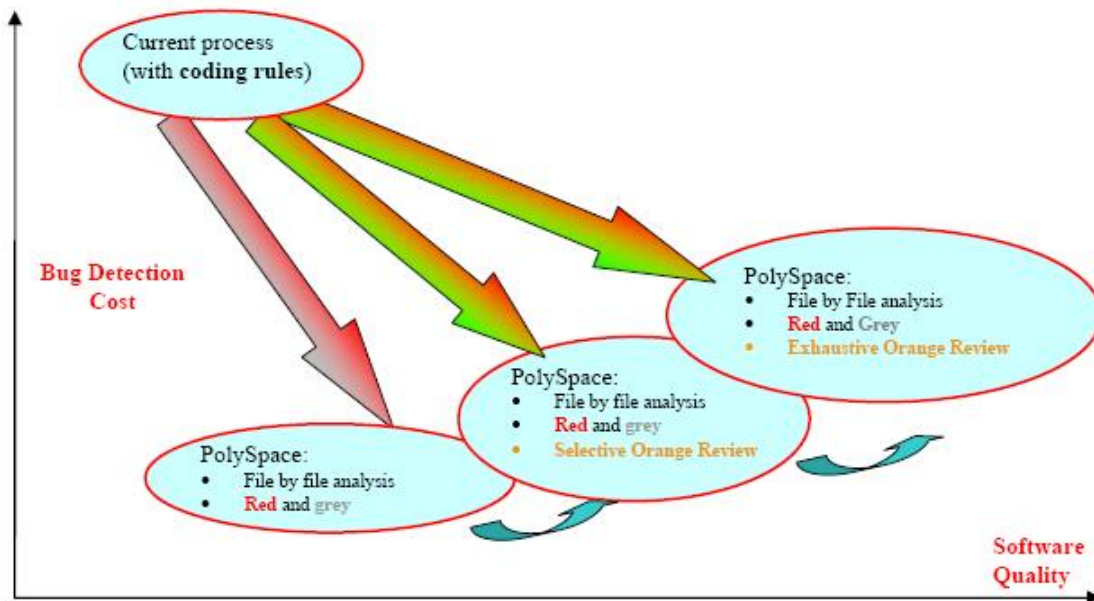
Again, the first approach is to use only the red and gray results — fix the red bugs, and check the dead code.

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

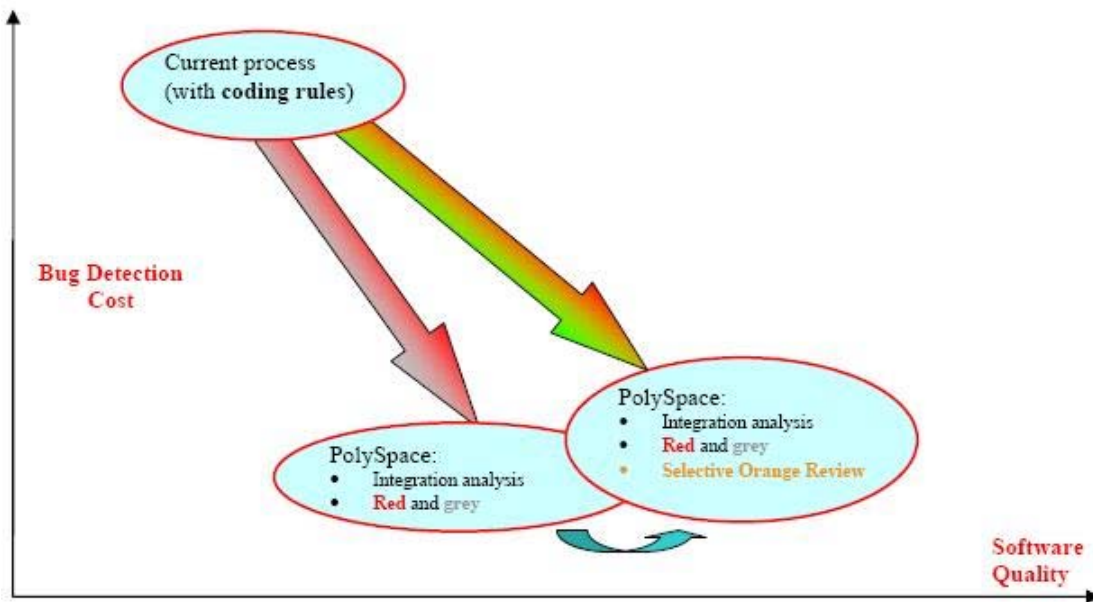
The main difference between this process and the previous process is in respect to the cost of bug detection. When PolySpace verification is used in conjunction with coding rules, the costs of bug detection are much lower.

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

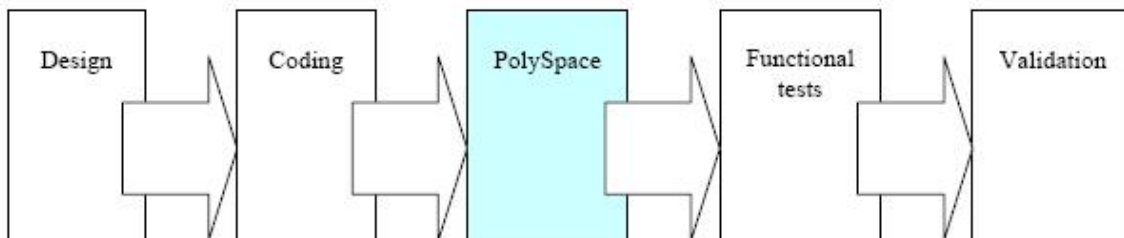
This approach is mainly for consideration by a project manager rather than a quality manager. It aims to improve productivity rather than to prove the quality of the application being analyzed.

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, 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.



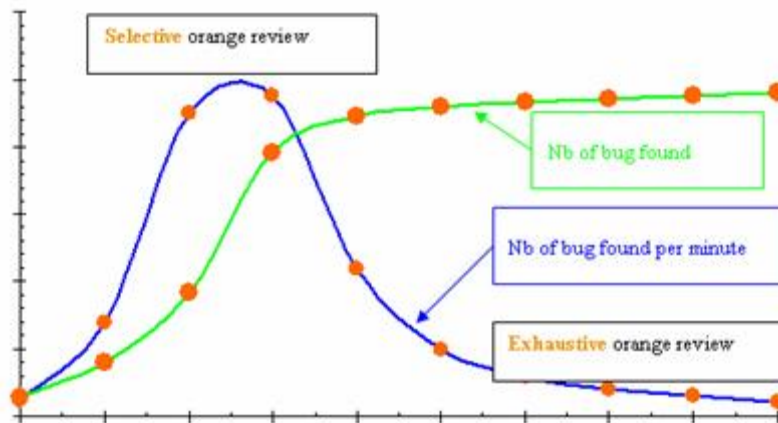
The Objective of Using PolySpace Verification

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 verification 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.

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

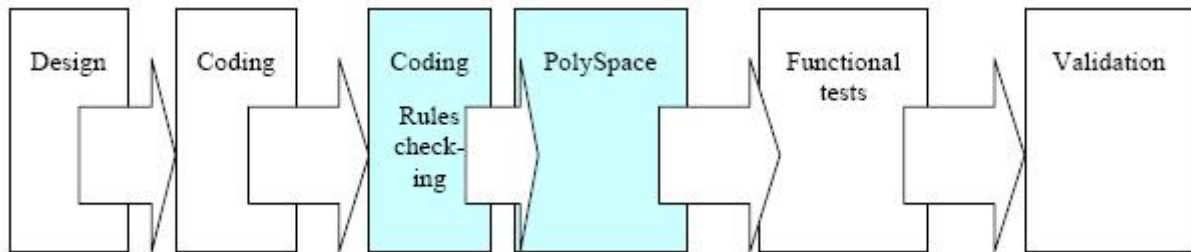
Overview

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.

Setting Up a Verification Project

- “Creating a Project” on page 3-2
- “Setting Up Project to Check Coding Rules” on page 3-19
- “Setting Up Project for Generic Target Processors” on page 3-25
- “Setting up Project to Automatically Test Orange Code” on page 3-33

Creating a Project

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

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:

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

Project Directories

Before you begin verifying your code with PolySpace software, you must know the location of your 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

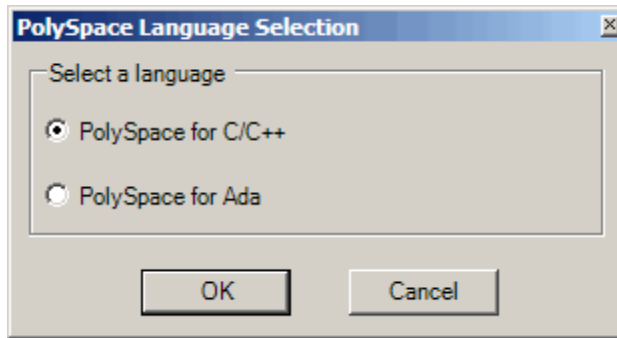
You use the PolySpace Launcher to create a project and start a verification.

To open the PolySpace Launcher:

- 1 Double-click the PolySpace Launcher icon.

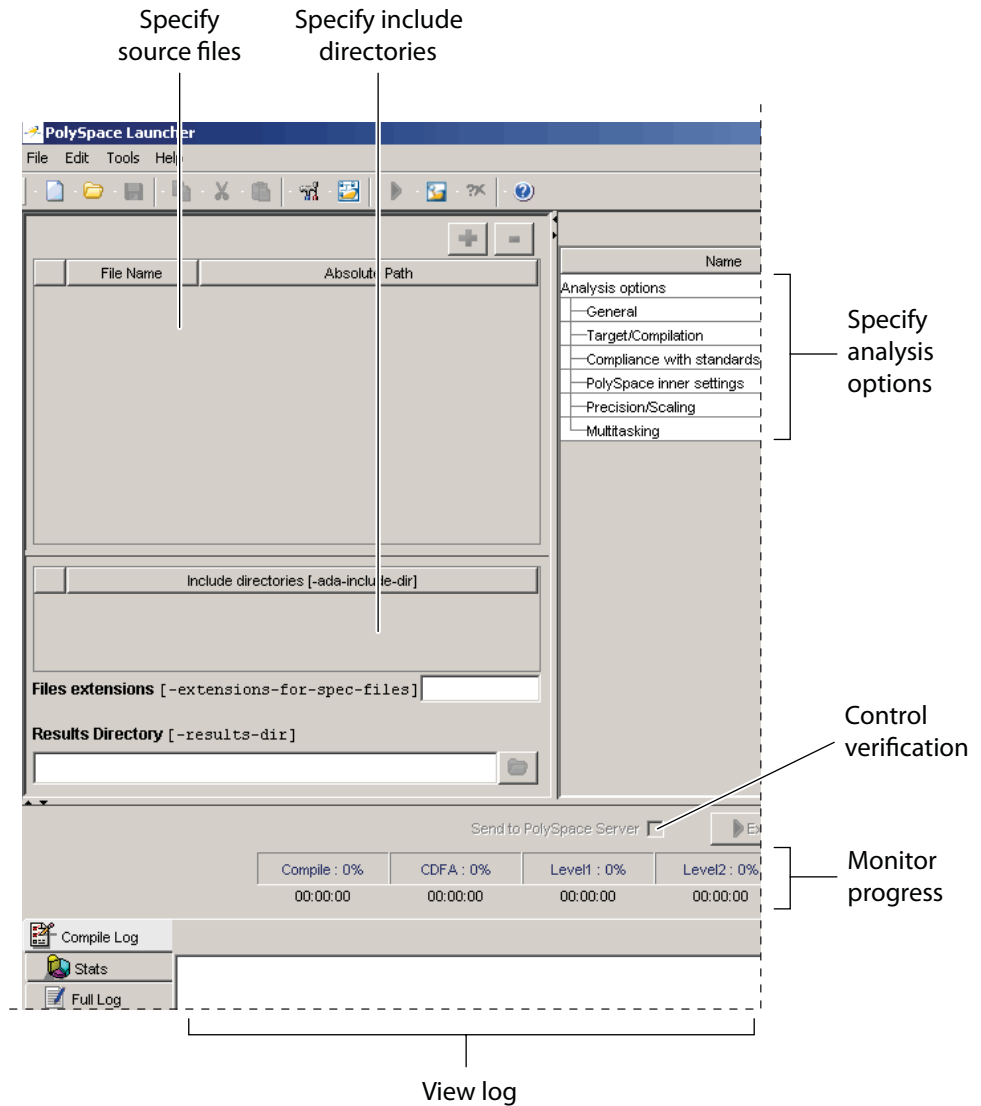


- 2 If you have both PolySpace Client for C/C++ and PolySpace Client for Ada products on your system, the **PolySpace Language Selection** dialog box will appear.



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

The PolySpace Launcher window appears:



The Launcher window has three main sections.

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

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

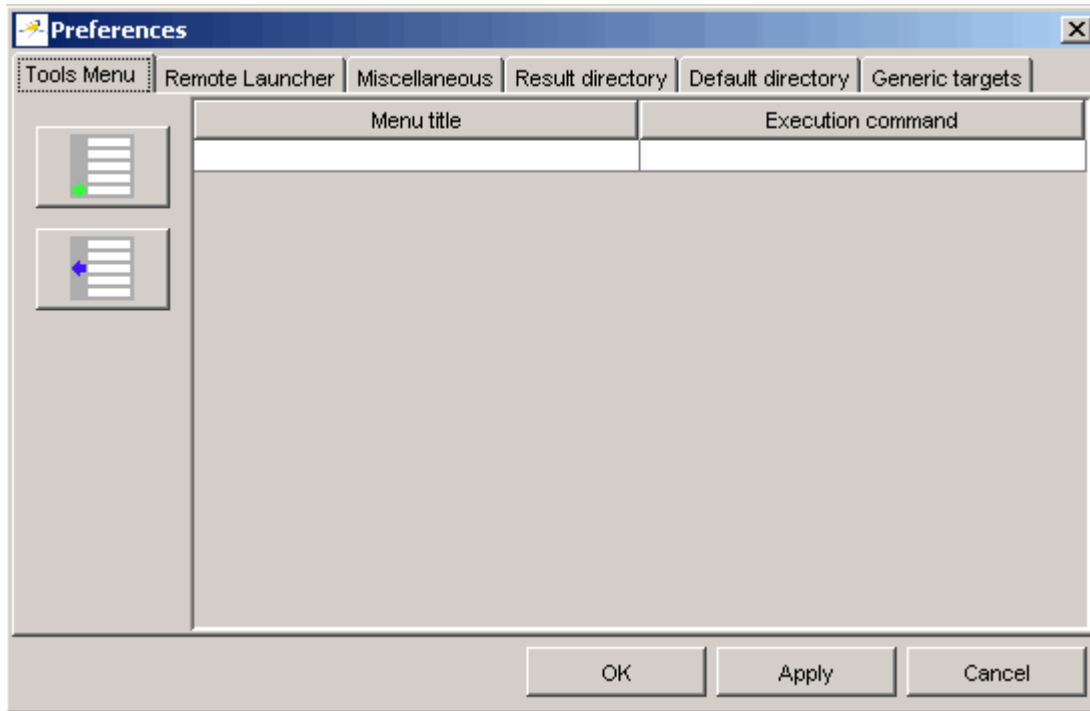
Specifying Default Directory

PolySpace software allows you to specify the default directory that appears in directory browsers in dialog boxes. If you do not change the default directory, the default directory is the installation directory. 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.

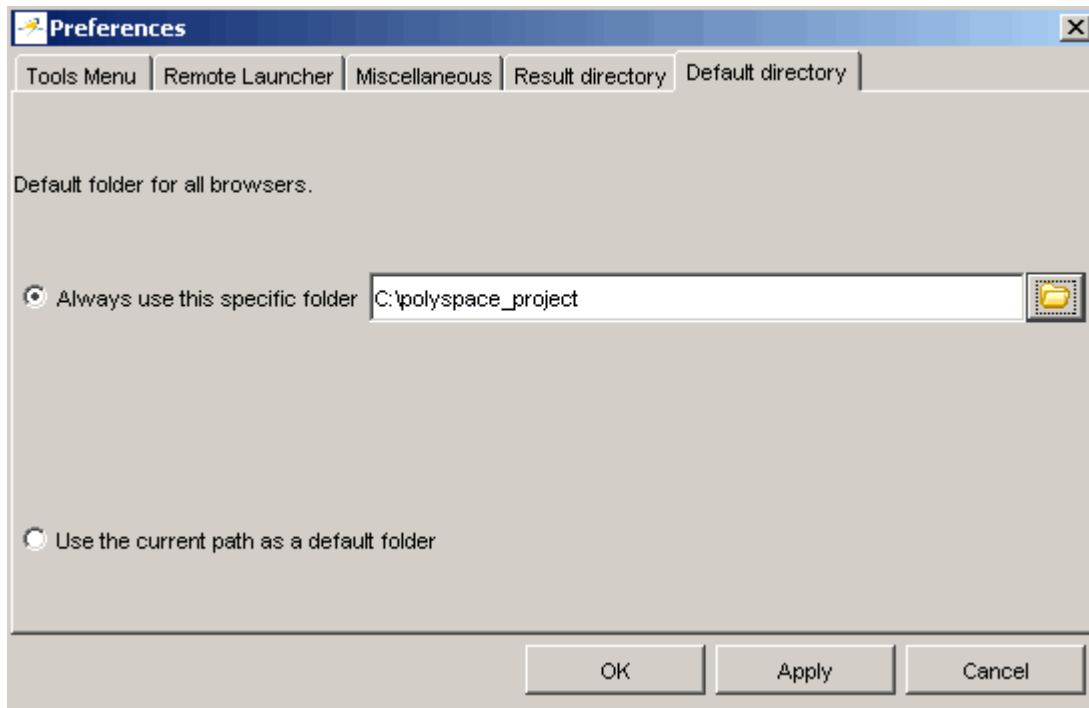
To change the default directory to the project directory:

- 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.

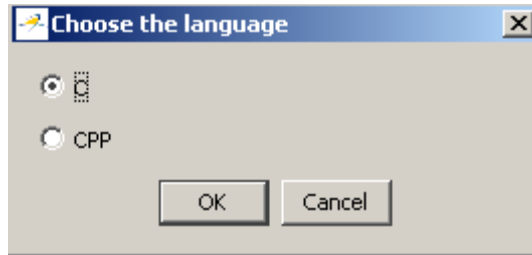
5 Click **OK** to apply the changes and close the dialog box.

Creating New Projects

To create a new project:

1 Select **File > New Project**.

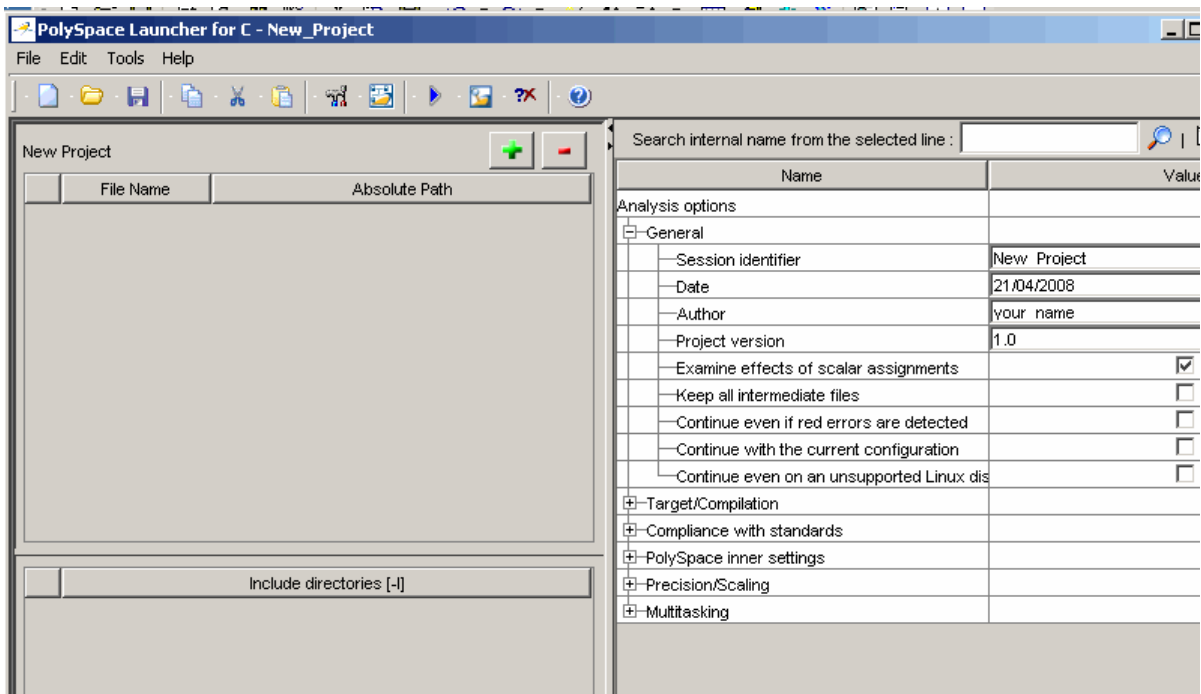
The **Choose the language** dialog box appears:



- 2 Select C, 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

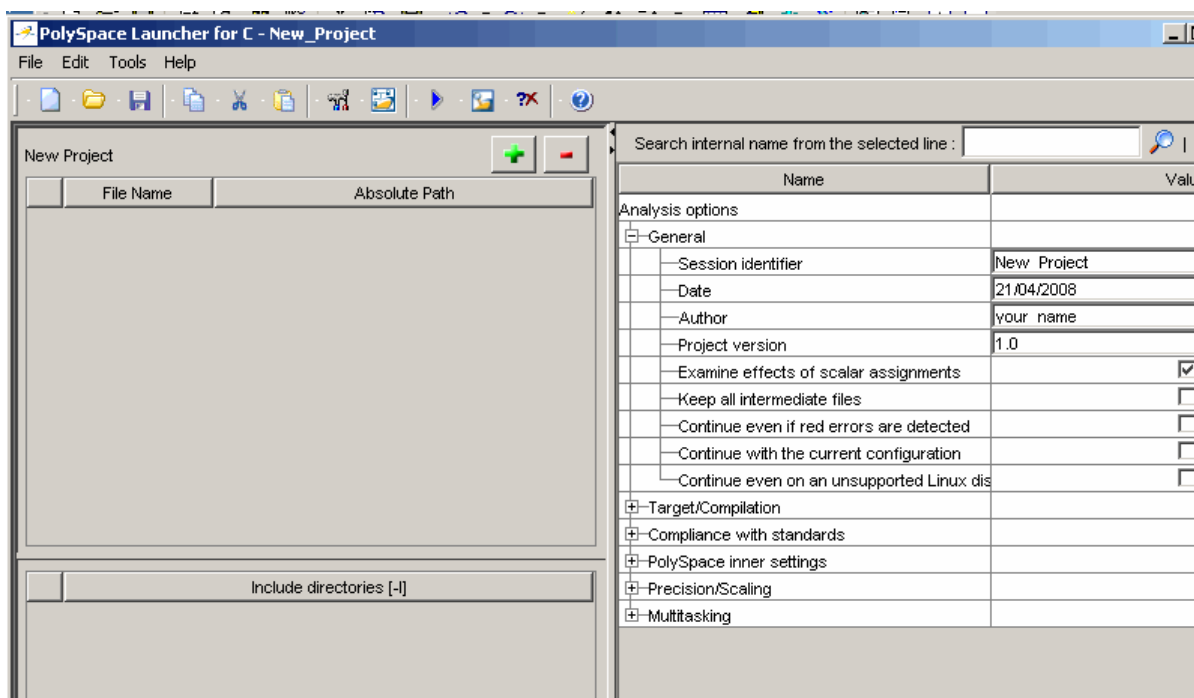
To open an existing project:

- 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.



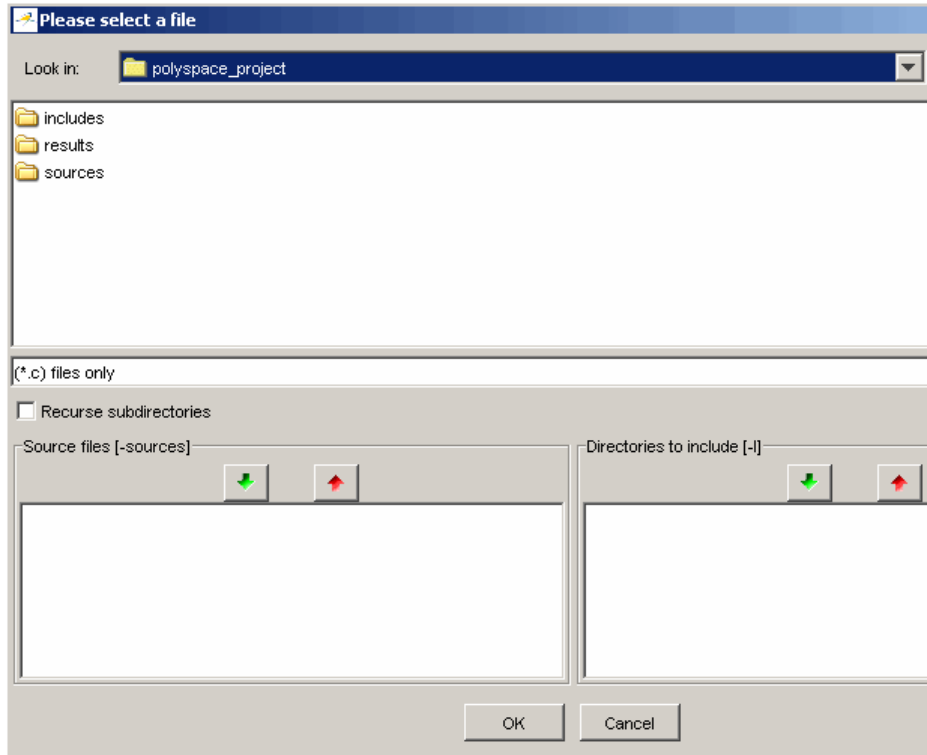
Specifying Source Files

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.

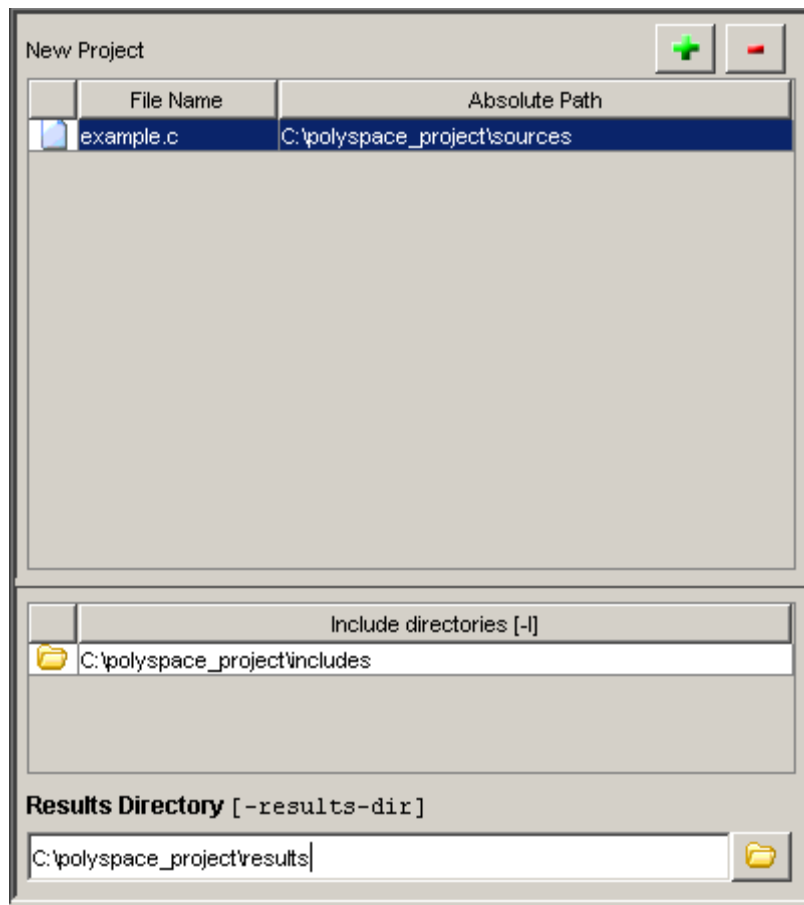


The path of each source files appear in the source files list.

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.



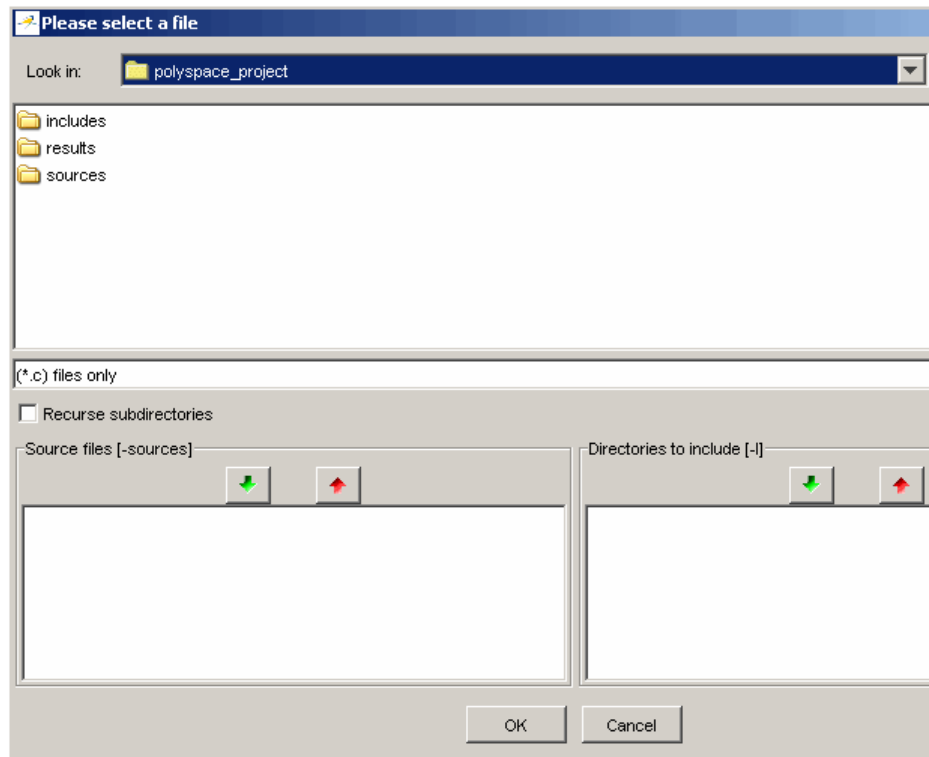
Specifying Include Directories

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.



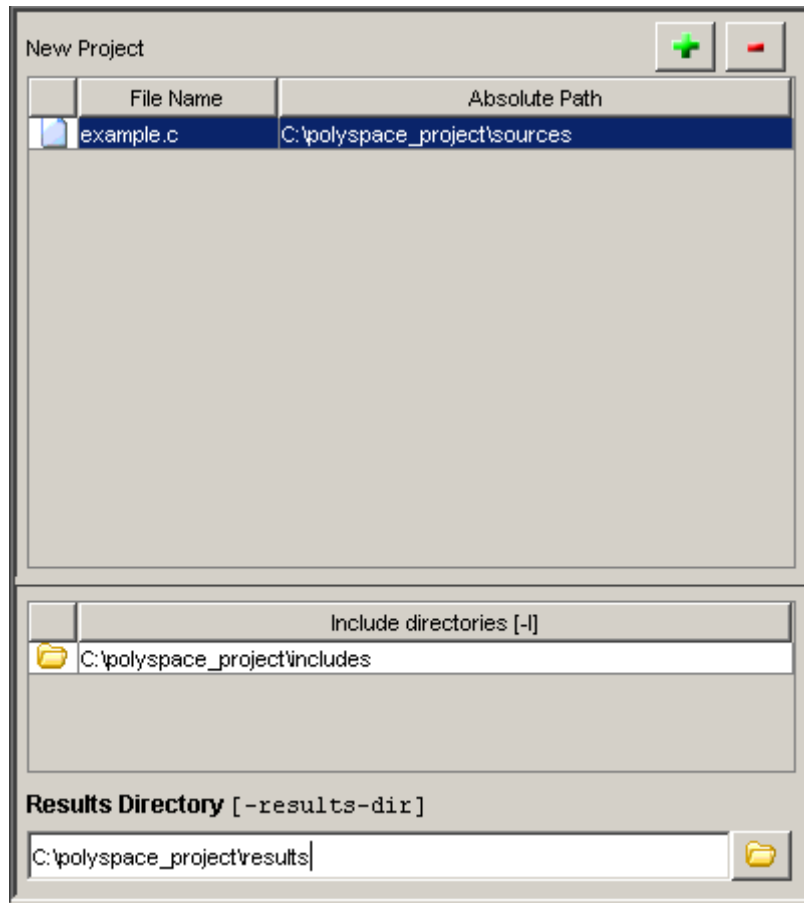
- 2 In the **Look in** field, navigate to your project directory.
- 3 Select the directory containing the include files for your project, then click the green down arrow button in the **Directories to include** section.



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.

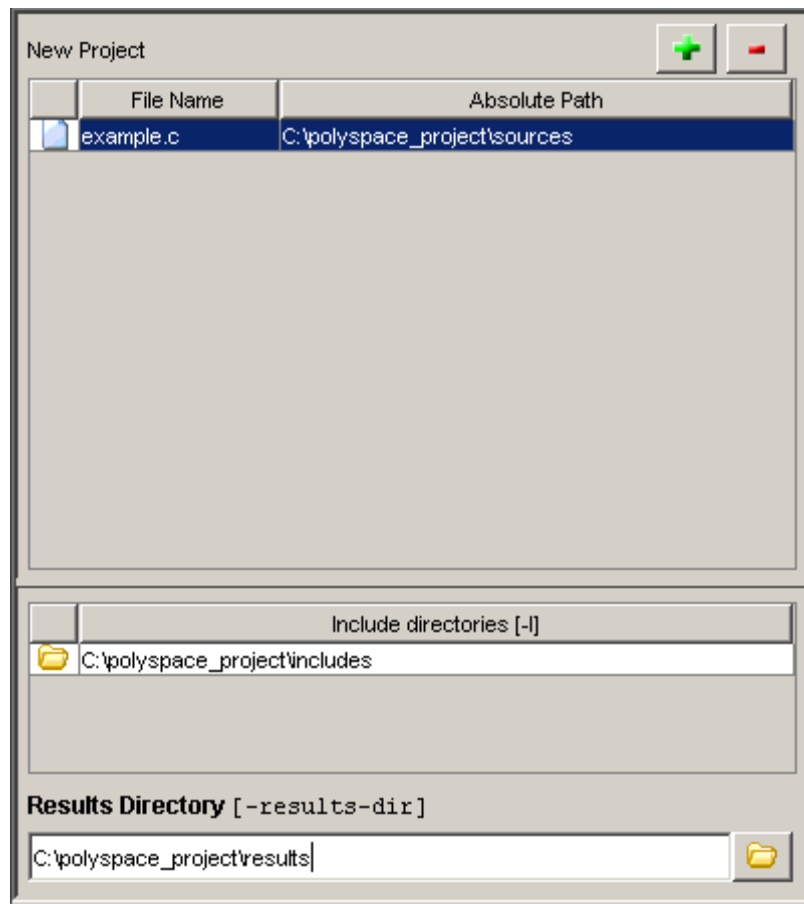


Specifying Results Directory

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 or 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 MISRA® reports directly from the Viewer logs.

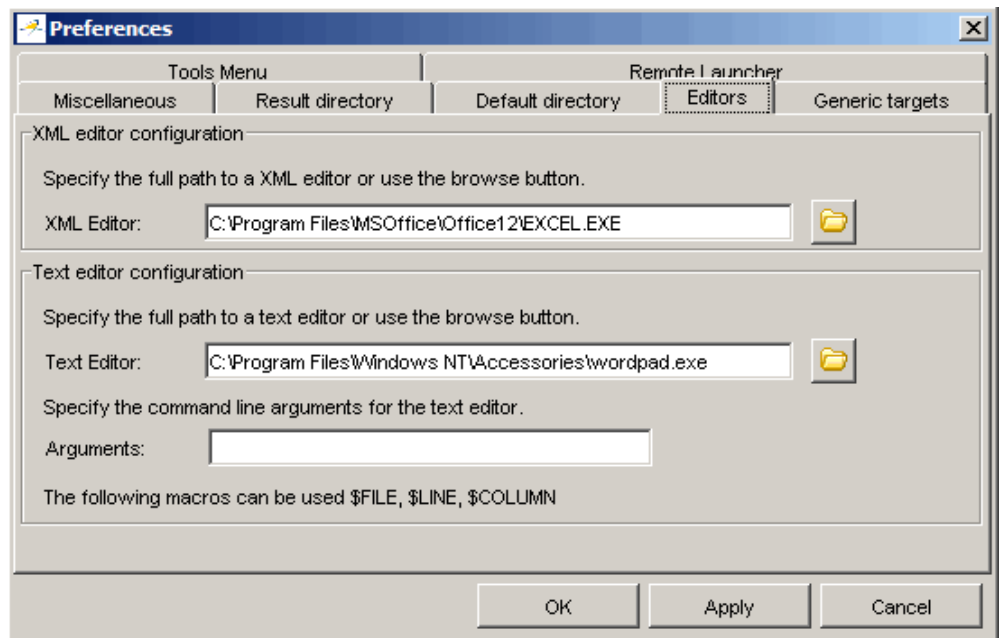
To configure your text and .XML editors:

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 MISRA-C reports.

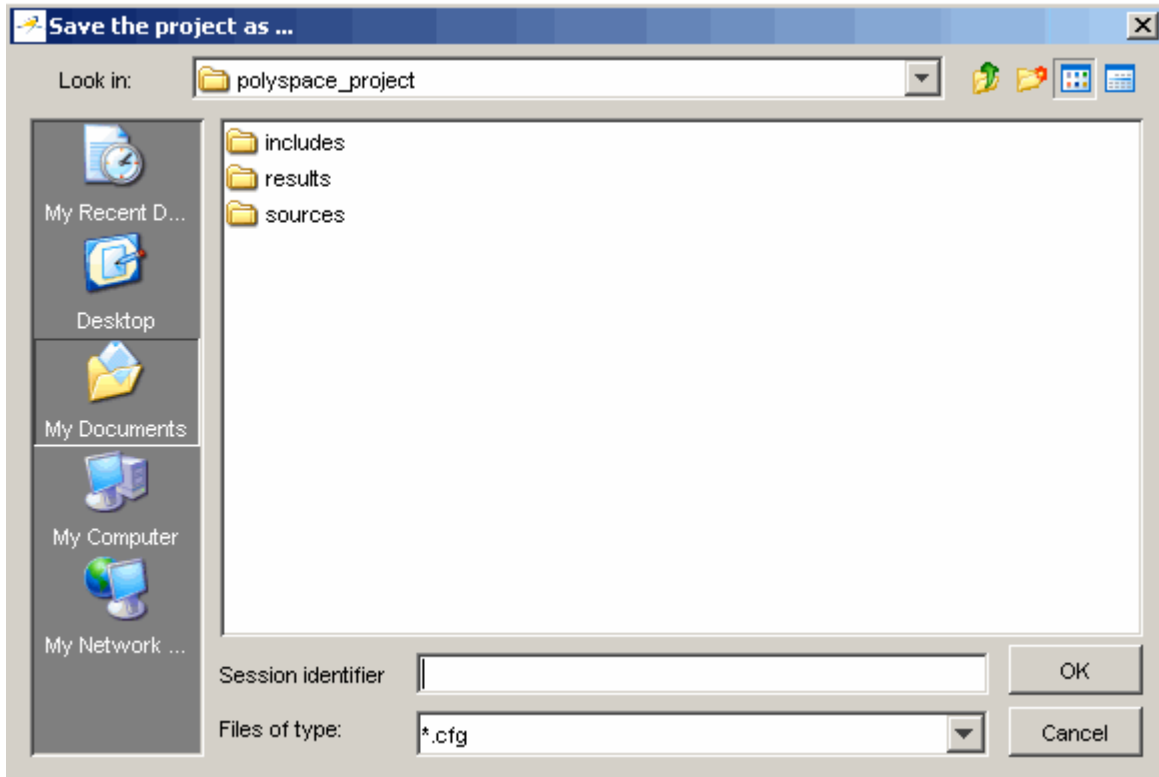
4 Specify a Text editor to use to view source files from the Viewer logs.

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 MISRA Checker Overview” on page 3-19
- “Checking Compliance with MISRA C Coding Rules” on page 3-19
- “Creating a MISRA C Rules File” on page 3-20
- “Excluding Files from the MISRA C Checking” on page 3-23

PolySpace MISRA Checker Overview

PolySpace software can check that C code complies with MISRA C 2004 standards.²

The MISRA checker enables PolySpace software to provide messages when MISRA C rules are not respected. Most messages are reported during the compile phase of a verification. The MISRA checker can check nearly all of the 141 MISRA C:2004 rules.

Note The PolySpace MISRA checker is based on MISRA C:2004 (<http://www.misra-c.com>).

Checking Compliance with MISRA C Coding Rules

To check MISRA C 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 MISRA C violations, you run the verification again.

To set the MISRA C checking option:

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

2. MISRA and MISRA C are registered trademarks of MISRA Ltd., held on behalf of the MISRA Consortium.

The Compliance with standards options appear.

2 Select the **Check MISRA-C:2004 rules** check box.

3 Expand the **Check MISRA-C:2004 rules** option.

Two options, **Rules configuration** and **Files and directories to ignore**, appear.

Name	Value		Internal name
Analysis options			
+ General			
+ Target/Compilation			
- Compliance with standards			
Code from DOS or Windows filesystem	<input checked="" type="checkbox"/>		-dos
+ Embedded assembler			
+ Strict	<input type="checkbox"/>		-strict
+ Permissive	<input type="checkbox"/>		-permissive
- Check MISRA-C:2004 rules	<input checked="" type="checkbox"/>		
Rules configuration		...	-misra2
Files and directories to ignore		...	-includes-to-ignore
+ KeilMAR support	default	▼	-dialect
+ PolySpace inner settings			
+ Precision/Scaling			
+ Multitasking			

4 Specify which MISRA C rules to check and which, if any, files to exclude from the checking.


Note For more information on using the MISRA C checker, see Chapter 11, “MISRA Checker”.

Creating a MISRA C Rules File

You must have a rules file to run a verification with MISRA C checking.

Opening a New Rules File

To open a new rules file:

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

A window for opening or creating a MISRA C rules file appears.

- 2 Select **File > New File**.

A table of rules appears.

3 Setting Up a Verification Project

Rules	Error	Warning	Off
MISRA C rules			
— Number of rules by mode :	7	1	134
<input checked="" type="checkbox"/> 1 Environment			
<input checked="" type="checkbox"/> 2 Language extensions			
<input checked="" type="checkbox"/> 3 Documentation			
<input checked="" type="checkbox"/> 4 Character sets			
<input checked="" type="checkbox"/> 5 Identifiers			
<input checked="" type="checkbox"/> 6 Types			
<input checked="" type="checkbox"/> 7 Constants			
<input checked="" type="checkbox"/> 8 Declarations and definitions			
<input checked="" type="checkbox"/> 9 Initialisation			
<input checked="" type="checkbox"/> 10 Arithmetic type conversions			
<input checked="" type="checkbox"/> 11 Pointer type conversions			
<input checked="" type="checkbox"/> 12 Expressions			
<input checked="" type="checkbox"/> 13 Control statement expressions			
<input checked="" type="checkbox"/> 14 Control flow			
<input checked="" type="checkbox"/> 15 Switch statements			
<input type="checkbox"/> 16 Functions			
<input type="radio"/> 16.1 Functions shall not be defined with variable numbers of arguments.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
<input type="radio"/> 16.2 Functions shall not call themselves, either directly or indirectly.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
<input checked="" type="radio"/> 16.3 Identifiers shall be given for all of the parameters in a function prototype.	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
<input type="radio"/> 16.4 The identifiers used in the declaration and definition of a function shall match.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
<input type="radio"/> 16.5 Functions with no parameters shall be declared with parameter type void.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
<input type="radio"/> 16.6 The number of arguments passed to a function shall match the number in the function prototype.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
<input checked="" type="radio"/> 16.7 A pointer parameter in a function prototype should be declared as pointer to const.	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>
<input type="radio"/> 16.8 All exit paths from a function with non-void return type shall have an explicit return statement.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
<input type="radio"/> 16.9 A function identifier shall only be used with either a preceding &, or with a preceding *.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
<input checked="" type="radio"/> 16.10 If a function returns error information, then that error information shall be checked.	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>
<input type="checkbox"/> 17 Pointer and arrays			
<input checked="" type="radio"/> 17.1 Pointer arithmetic shall only be applied to pointers that address an array.	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>
<input checked="" type="radio"/> 17.2 Pointer subtraction shall only be applied to pointers that address elements of an array.	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>
<input checked="" type="radio"/> 17.3 >, >=, <, <= shall not be applied to pointer types except where they point to the same array.	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>
<input type="radio"/> 17.4 Array indexing shall be the only allowed form of pointer arithmetic.	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
<input type="radio"/> 17.5 The declaration of objects should contain no more than 2 levels of pointer indirection.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
<input type="radio"/> 17.6 The address of an object with automatic storage shall not be assigned to a pointer.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
<input checked="" type="checkbox"/> 18 Structures and unions			
<input checked="" type="checkbox"/> 19 Preprocessing directives			
<input checked="" type="checkbox"/> 20 Standard libraries			
<input checked="" type="checkbox"/> 21 Run-time failures			

- 3** For each rule, you 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).

- 4** Click **OK** to save the rules and close the window.


The **Save as** dialog box opens.

- 5** In **File**, enter a name for the rules file.

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

Excluding Files from the MISRA C Checking

You can exclude files from MISRA C checking. You might want to exclude some included files. To exclude `math.h` from the MISRA C checking of the project `example.cfg`:

- 1** Click the button  to the right of the **Files and directories to ignore** option.
- 2** Click the folder icon.



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

3 Select the files or directories (such as include files) you want to ignore.

4 Click **OK**.

The selected files 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 3-25

“Creating Project Model Files” on page 3-26

“Viewing Existing Generic Targets” on page 3-26

“Defining Generic Targets” on page 3-27

“Deleting a Generic Target ” on page 3-30

“Common Generic Targets” on page 3-30

“Creating a Configuration File from a PolySpace Project Model File” on page 3-31

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.

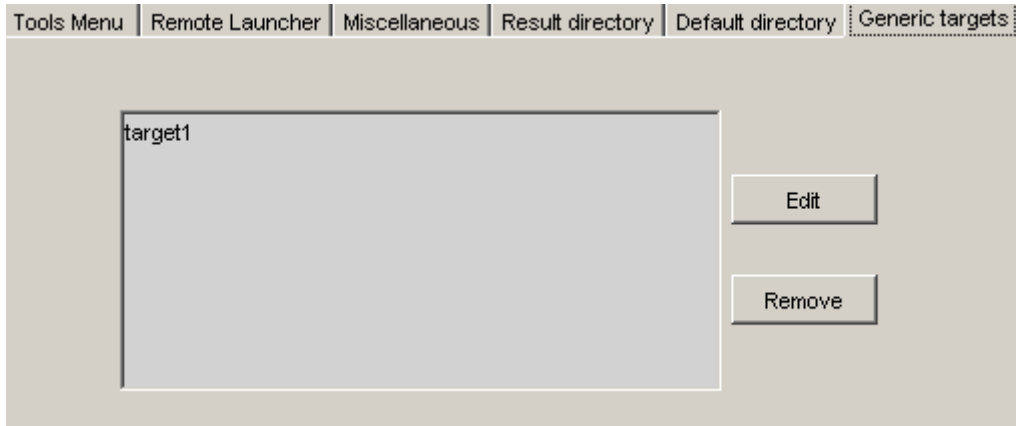
To view existing generic targets:

- 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

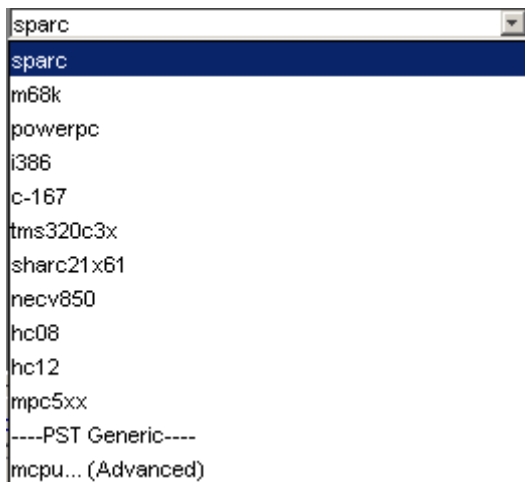
If your application is designed for a custom target processor, you can configure many basic characteristics of the target by selecting the PST Generic target, and specifying the characteristics of your processor.

To configure a generic target:

To define a generic target:

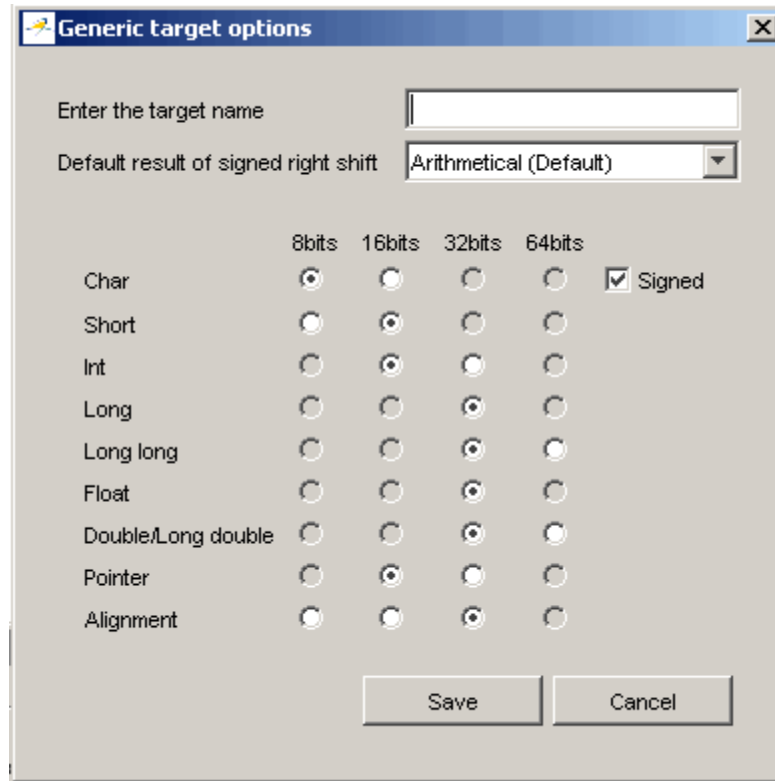
1 In **Analysis options**, expand **Target/Compilation**.

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

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.

Setting up Project to Automatically Test Orange Code

In this section...
“PolySpace Automatic Orange Tester” on page 3-33
“Enabling the Automatic Orange Tester” on page 3-33

PolySpace Automatic Orange Tester

The PolySpace Automatic Orange Tester dynamically stresses unproven code (orange checks) to identify runtime errors, and provides information to help you identify the cause of these errors.

The Automatic Orange Tester complements the results review in the Viewer by automatically creating test cases for all input variables in orange code, and then dynamically testing the code to find actual runtime errors.



For more information, see “Automatically Testing Orange Code” on page 9-26.

Enabling the Automatic Orange Tester

Before you can use the Automatic Orange Tester, you must run a PolySpace verification with the `-prepare-automatic-tests` option enabled. This option generates the data necessary to perform dynamic tests in the Automatic Orange Tester.

To enable the automatic orange tester:

- 1** In the Analysis Options window, expand the **PolySpace inner settings** menu.
- 2** Select the **Automatic Orange Tester** check box.

Search internal name from the selected line :  

Name	Value	Internal name
Analysis options		
+ General		
+ Target/Compilation		
+ Compliance with standards		
- PolySpace inner settings		
+ Generate a main	<input checked="" type="checkbox"/>	-main-generator
+ Stubbing		
+ Assumptions		
Automatic Orange Tester	<input checked="" type="checkbox"/>	-prepare-automatic-tests
+ Others		
+ Precision/Scaling		
+ Multitasking		

The `-prepare-automatic-tests` option is enabled.

For more information on using the Automatic Orange Tester, see “Automatically Testing Orange Code” on page 9-26.

Emulating Your Runtime Environment

- “Setting Up a Target” on page 4-2
- “Verifying an Application Without a “Main”” on page 4-22
- “Applying Data Ranges to External Variables and Stub Functions (DRS)” on page 4-25

Setting Up a Target

In this section...

“Target/Compiler Overview” on page 4-2

“Specifying Target/Compilation Parameters” on page 4-2

“Predefined Target Processor Specifications (size of char, int, float, double...)” on page 4-3

“Generic Target Processors” on page 4-5

“Compiling Operating System Dependent Code (OS-target issues)” on page 4-5

“Address Alignment” on page 4-9

“Ignoring or Replacing Keywords Before Compilation” on page 4-10

“Verifying Code That Uses KEIL or IAR Dialects” on page 4-13

“How to Gather Compilation Options Efficiently” on page 4-19

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

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

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 ³	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
tms320c3x	32	32	32	32	64	32	32	40 ⁴	32	signed	Little	int, long
sharc21x61	32	32	32	32	64	32	32 ⁵	64	32	signed	Little	int, long
NEC-V850	8	16	32	32	32	32	32	64	32	signed	Little	int
hc08 ⁶	8	16	16	32	32	32	32	32	16 ₇	unsigned	Big	int
hc12 ³	8	16	16	32	32	32	32	32	32 ₄	signed	Big	int
mpc5xx (#3)	8	16	32	32	64	32	32	32	32	signed	Big	int, long

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

3. The M68k family (68000, 68020, etc.) includes the “ColdFire” processor
4. All operations on long double values will be imprecise (that is, shown as orange).
5. On this target, a double may be 32 or 64 bits long. Only 32 bits double are supported.
6. Non ANSI C specified keywords and compiler implementation-dependent pragmas and interrupt facilities are not tokens into account by this support
7. all kinds of pointers (near or far pointer) have 2 bytes (hc08) or 4 bytes (hc12) of width physically.

Note The following table describes target processors that are not fully supported by PolySpace software, but for which you can still perform verification. In these cases, you should select the target processor listed in the “Nearest Processor” column. The characteristics that are not identical between the target processor and its equivalent are highlighted in red below. You should take these differences into account when reviewing verification results.

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 ⁸	32	signed	int, long	i386
tms320c2x	16	16	16	32	N/A	32	32	32	16	signed	int	Unsupported

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 3-25.

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 4-6
- “My Target Application Runs on Linux” on page 4-8
- “My Target Application Runs on Solaris” on page 4-8
- “My Target Application Runs on Vxworks” on page 4-9

8. All operations on long double values will be imprecise (that is, shown as orange).

- “My Target Application Does Not Run on Linux, vxworks nor Solaris” on page 4-9

List of Predefined Compilation Flags

These flags concern predefined OS-target: no-predefined-OS, linux, vxworks, Solaris and visual (-OS-target option).

OS-target	Compilation flags	-include file and content
no-predefined-OS	-D __STDC__	
visual	-D __STDC__	-include <product_dir>/cinclude/pst-visual.h
vxworks	-D __STDC__ -DANSI_PROTOTYPES -DSTATIC= -DCONST=const -D __STDC__ -D __GNUC__=2 -Dunix -D __unix -D __unix__ -Dsparc -D __sparc -D __sparc__ -Dsun -D __sun -D __sun__ -D __svr4__ -D __SVR4	-include <product_dir>/cinclude/pst-vxworks.h

OS-target	Compilation flags	-include file and content
linux	-D __STDC__ -D __GNUC__=2 -D __GNUC_MINOR__=6 -D __GNUC__=2 -D __GNUC_MINOR__=6 -D __ELF__ -D unix -D __unix -D __unix__ -D linux -D __linux -D __linux__ -D i386 -D __i386 -D __i386__ -D i686 -D __i686 -D __i686__ -D pentiumpro -D __pentiumpro -D __pentiumpro__	<product_dir>/cinclude/pst-linux.h
Solaris	-D __STDC__ -D __GNUC__=2 -D __GNUC_MINOR__=8 -D __GNUC__=2 -D __GNUC_MINOR__=8 -D __GCC_NEW_VARARGS__ -D unix -D __unix -D __unix__ -D sparc -D __sparc -D __sparc__ -D sun -D __sun -D __sun__ -D svr4 -D __SVR4	No -include file mentioned

Note The use of the `OS-target` option is entirely equivalent to the following alternative approaches.

- Setting the same `-D` flags manually, or
 - Using the `-include` option on a copied and modified `pst-OS-target.h` file
-

My Target Application Runs on Linux

The minimum set of options is as follows:

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

If PolySpace runs on a Solaris™ machine:

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

My Target Application Runs on Vxworks

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

```
polyspace-c \  
-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-c \  
-OS-target no-predefined-OS \  
-I /your_path_to/MyTarget_include_directories
```

Address Alignment

PolySpace handles address alignment by calculating `sizeof` and alignments. This approach takes into account 3 constraints implied by the ANSI standard which guarantee that:

- that global `sizeof` and `offsetof` fields are optimum (i.e. as short as possible);
- the alignment of all addressable units is respected;
- global alignment is respected.

Consider the example:

```
struct foo {char a; int b;}
```

- Each field must be aligned; that is, the starting offset of a field must be a multiple of its own size⁹
- So in the example, `char a` begins at offset 0 and its size is 8 bits. `int b` cannot begin at 8 (the end of the previous field) because the starting offset must be a multiple of its own size (32 bits). Consequently, `int b` begins at offset=32. The size of the `struct foo` before global alignment is therefore 64 bits.
- The global alignment of a structure is the maximum of the individual alignments of each of its fields;
- In the example, `global_alignment = max (alignment char a, alignment int b) = max (8, 32) = 32`
- The size of a struct must be a multiple of its global alignment. In our case, `b` begins at 32 and is 32 long, and the size of the struct (64) is a multiple of the `global_alignment` (32), so `sizeof` is not adjusted.

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  
  
#####
```

9. except in the cases of “double” and “long” on some targets.

```

# 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/;
#
#####

```

Verifying Code That Uses KEIL or IAR Dialects

Typical embedded control applications frequently read and write port data, set timer registers and read input captures. To deal with this without using assembly language, some microprocessor compilers have specified special data types like `sfr` and `sbit`. Typical declarations are:

```

sfr A0 0x80
sfr A1 0x81
sfr ADCUP 0xDE
sbit EI 0x9F

```

These declarations reside in header files such as `regxx.h` for the basic 80Cxxx micro processor. The definition of `sfr` in these header files customizes the compiler to the target processor.

When accessing a register or a port, using `sfr` data is then simple, but is not part of standard ANSI C:

```

{
ADCUP = 0x08; /* Write data to register */
A1 = 0xFF; /* Write data to Port */
status = P0; /* Read data from Port */
EI = 1; /* Set a bit (enable all interrupts) */
}

```

You can verify this type of code using the **Kiel/IAR support** option (`-dialect`). This option allows the software to support the Keil or IAR C language extensions even if some structures, keywords, and syntax are not

ANSI standard. The following tables summarize what is supported when verifying code that is associated with the keil or iar dialects.

The following table summarizes the supported keil C language extensions:

Example: -dialect keil -sfr-types sfr=8

Type/Language	Description	Example	Restrictions
Type bit	<ul style="list-style-type: none"> An expression to type bit gives values in range [0,1]. Converting an expression in the type, gives 1 if it is not equal to 0, else 0. This behavior is similar to c++ bool type. 	<pre>bit x = 0, y = 1, z = 2; assert(x == 0); assert(y == 1); assert(z == 1); assert(sizeof(bit) == sizeof(int));</pre>	pointers to bits and arrays of bits are not allowed
Type sfr	<ul style="list-style-type: none"> The -sfr-types option defines unsigned types name and size in bits. The behavior of a variable follows a variable of type integral. A variable which overlaps another one (in term of address) will be considered as volatile. 	<pre>sfr x = 0xf0; // declaration of variable x at address 0xF0 sfr16 y = 0x4EEF;</pre> <p>For this example, options need to be:</p> <pre>-dialect keil sfr -types sfr=8, sfr16=16</pre>	sfr and sbit types are only allowed in declarations of external global variables.

Example: -dialect keil -sfr-types sfr=8 (Continued)

Type/Language	Description	Example	Restrictions
Type sbit	<ul style="list-style-type: none"> Each read/write access of a variable is replaced by an access of the corresponding sfr variable access. Only external global variables can be mapped with a sbit variable. Allowed expressions are integer variables, cells of array of int and struct/union integral fields. a variable can also be declared as extern bit in an another file. 	<pre>sfr x = 0xF0; sbit x1 = x ^ 1; // 1st bit of x sbit x2 = 0xF0 ^ 2; // 2nd bit of x sbit x3 = 0xF3; // 3rd bit of x sbit y0 = t[3] ^ 1; /* file1.c */ sbit x = P0 ^ 1; /* file2.c */ extern bit x; x = 1; // set the 1st bit of P0 to 1</pre>	
Absolute variable location	Allowed constants are integers, strings and identifiers.	<pre>int var _at_ 0xF0 int x @ 0xFE ; static const int y @ 0xA0 = 3;</pre>	Absolute variable locations are ignored (even if declared with a #pragma location).
Interrupt functions	A warnings in the log file is displayed when an interrupt function has been found: "interrupt handler detected : <name>" or "task entry point detected : <name>"	<pre>void foo1 (void) interrupt XX = YY using 99 { } void foo2 (void) _ task_ 99 _priority_ 2 { }</pre>	Entry points and interrupts are not taken into account as -entry-points.
Keywords ignored	alien, bdata, far, idata, ebddata, huge, sdata, small, compact, large, reentrant. Defining -D __C51__, keywords large code, data, xdata, pdata and xhuge are ignored.		

The following table summarize the iar dialect:

Example: -dialect iar -sfr-types sfr=8

Type/Language	Description	Example	Restrictions
Type bit	<ul style="list-style-type: none"> • An expression to type bit gives values in range [0,1]. • Converting an expression in the type, gives 1 if it is not equal to 0, else 0. This behavior is similar to c++ bool type. • If initialized with values 0 or 1, a variable of type bit is a simple variable (like a c++ bool). • A variable of type bit is a register bit variable (mapped with a bit or a sfr type) 	<pre>bit y1 = s.y.z.2; bit x4 = x . 4; bit x5 = 0xF0 . 5; y1 = 1; // 2nd bit of s.y.z is set to 1</pre>	pointers to bits and arrays of bits are not allowed
Type sfr	<ul style="list-style-type: none"> • The -sfr-types option defines unsigned types name and size. • The behavior of a variable follows a variable of type integral. • A variable which overlaps another one (in term of address) 	<pre>sfr x = 0xf0; // declaration of variable x at address 0xF0</pre>	sfr and sbit types are only allowed in declarations of external global variables.

Example: -dialect iar -sfr-types sfr=8 (Continued)

Type/Language	Description	Example	Restrictions
	will be considered as volatile.		
Individual bit access	<ul style="list-style-type: none"> Individual bit can be accessed without using sbit/bit variables. Type is allowed for integer variables, cells of integer array, and struct/union integral fields. 	<pre>int x[3], y; x[2].2 = x[0].3 + y.1;</pre>	
Absolute variable location	Allowed constants are integers, strings and identifiers.	<pre>int var _at_ 0xF0 int x @ 0xFE ; static const int y @ 0xA0 = 3;</pre>	Absolute variable locations are ignored (even if declared with a #pragma location).
Interrupt functions	A warnings in the log file is displayed when an interrupt function has been found: "interrupt handler detected : funcname"	<pre>interrupt [XX] using [99] void foo1 (void) { } monitor [YY] foo2 (void) { }</pre>	Entry points and interrupts are not taken into account as -entry-points.
Keywords ignored	saddr, reentrant, reentrant_idata, non_banked, plm, bdata, idata, pdata, code, data, xdata, xhuge, interrupt, __interrupt and __intrinsic		

Example: -dialect iar -sfr-types sfr=8 (Continued)

Type/Language	Description	Example	Restrictions
Unnamed struct/union	<ul style="list-style-type: none"> • Fields of unions/structs with no tag and no name can be accessed without naming their parent struct. • Option <code>-allow-unnamed-fields</code> need to be used to allow anonymous struct fields. • On a conflict between a field of an anonymous struct with other identifiers: <ul style="list-style-type: none"> ▪ with a variable name, field name is hidden ▪ with a field of another anonymous struct at different scope, closer scope is chosen ▪ with a field of another anonymous struct at same scope: an error "anonymous struct field name <name> conflict" is displayed in the log file. 	<pre>union { int x; }; union { int y; struct { int z; }; } @ 0xF0;</pre>	

Example: -dialect iar -sfr-types sfr=8 (Continued)

Type/Language	Description	Example	Restrictions
no_init attribute	<ul style="list-style-type: none"> a global variable declared with this attribute is handled like an external variable. It is handled like a type qualifier. 	<pre>no_init int x; no_init union { int y; } @ 0xFE;</pre>	#pragma no_init has no effect

The option `sfr-types` defines the size of a `sfr` type for the keil or iar dialect.

The syntax for an `sfr` element in the list is `type-name=typesize`.

For example:

```
sfr-types sfr=8,sfr16=16
```

defines two `sfr` types: `sfr` with a size of 8 bits, and `sfr16` with a size of 16-bits. A value `type-name` must be given only once. 8, 16 and 32 are the only supported values for `type-size`.

Note As soon as an `sfr` type is used in the code, you must specify its name and size, even if it is the keyword `sfr`.

Note Many IAR and Keil compilers currently exist that are associated to specific targets. It is difficult to maintain a complete list of those supported.

How to Gather Compilation Options Efficiently

The code is often tuned for the target (as discussed to “Verifying Code That Uses KEIL or IAR Dialects” on page 4-13). 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);
```

Verifying an Application Without a “Main”

In this section...
“Main Generator Overview” on page 4-22
“Automatically Generating a Main” on page 4-23
“Manually Generating a Main” on page 4-23

Main Generator Overview

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

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

PolySpace Client for C/C++ Software Default Behavior

The PolySpace Client for C/C++ product automatically checks whether the code for verification contains a "main" or not.

- If a main exists in the set of files, the verification proceeds with that main.
- If a main does not exist, the tool generates a main. You can specify the options: -main-generator-writes-variables and -main-generator-calls.

PolySpace Server for C/C++ Software Default Behavior

By default, the PolySpace Server for C/C++ product stops verification if it does not find a main. This behavior can help isolate files missing from the verification.

However, you can specify that the PolySpace Server for C/C++ product automatically generate a main. The tool then generates a main with

the assumption of verifying a library. You can specify the options `-main-generator-writes-variables` and `-main-generator-calls`.

Automatically Generating a Main

When you run a client verification, or a server verification using the **Generate a main** (`-main-generator`) option, the software automatically generates a main.

The generated main has three distinct default behaviors.

- It first initializes any variables identified by the option `-main-generator-writes-variables`. The default setting for this option is `public`.
- It then calls a function which could be considered an initialization function with the option `-function-called-before-main`.
- It then calls any functions identified by the option `-main-generator-calls`. The default setting for this option is `-main-generator-calls unused`.

For more information on the main generator, see “MAIN GENERATOR OPTIONS (`-main-generator`) for PolySpace Software”.

Manually Generating a Main

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

There are three steps involved in manually defining the main.

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

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

Consider the following example. Suppose that the API functions are:

```
int func1(void *ptr, int x);
void func2(int x, int y);
```

You should create the following main:

```
void main()
{
    volatile int random; /* We need an integer variable as a function
    parameter */
    volatile void * volatile ptr; /* We need a void pointer as a function
    parameter */
    while (random) {
        switch (random) {
            case 1:
                random = func1(ptr, random); break; /* One API function call */
            default:
                func2(random, random); /* Another API function call */
        }
    }
}
```

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

In this section...

“Overview of Data Range Specifications (DRS)” on page 4-25

“Specifying Data Ranges” on page 4-25

“File Format” on page 4-26

“Variable Scope” on page 4-28

“Performing Efficient Module Testing with DRS” on page 4-30

“Reducing Oranges with DRS” on page 4-31

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 4-28) 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 4-26.
- 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:

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

	init	permanent	globalassert	comments
Structure field in array	Ok	No effect	No effect	Only when leaves are int or real. Moreover the syntax is the following: <array_name>. <field_name>
Array	Ok	Ok	Ok	Only for int and real fields, including structures or arrays of integer or real fields (see below)
Pointer	No effect	No effect	No effect	
Union field	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	Reduces the number of oranges, (compared with a standard PolySpace verification)	Input data that were full range are set to a smaller range.	↓	↑
Outputs	globalassert	Increases the number of oranges, (compared with a standard PolySpace verification)	More verification is introduced into the code, resulting in both more orange checks and more green checks.	↑	→
Calibration	init	Increases the number of oranges, (compared with a standard PolySpace verification)	Data that were constant are set to a wider range.	↑	↓

Reducing Oranges with DRS

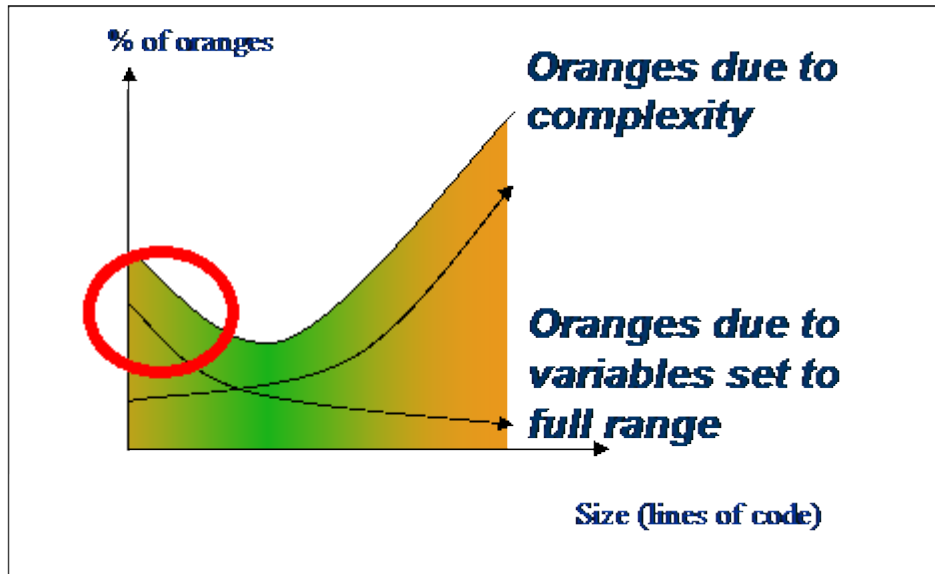
When performing robustness (worst case) verification, data inputs are always set to their full range. Therefore, every operation on these inputs, even a simple “one_input + 10” can produce an overflow, as the range of one_input varies between the min and the max of the type.

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

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

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

Using DRS removes the oranges located in the red circle below.



Why Is DRS Most Effective on Module Testing?

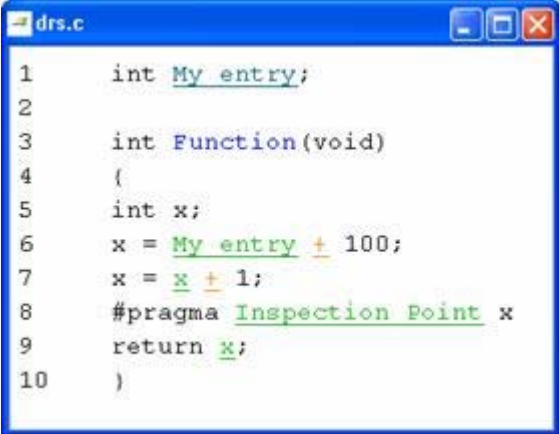
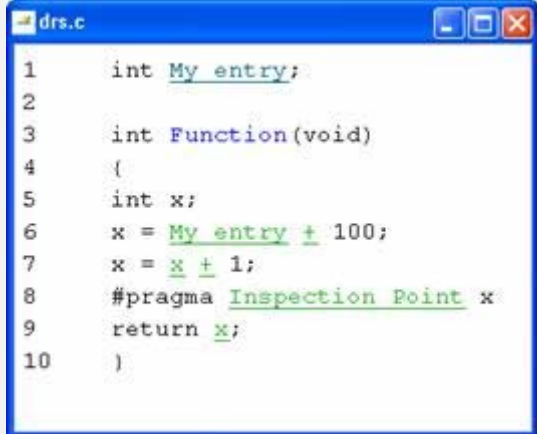
Removing oranges caused by full-range (worst-case) data can drastically reduce the total number of orange checks, especially when used on verifications of small files or modules. However, the number of orange checks caused by code complexity is not effected by DRS. For more information on oranges caused by code complexity, see “Considering the Effects of Application Code Size” on page 7-43, and “Why Should there be an Optimum Size?” on page 7-30.

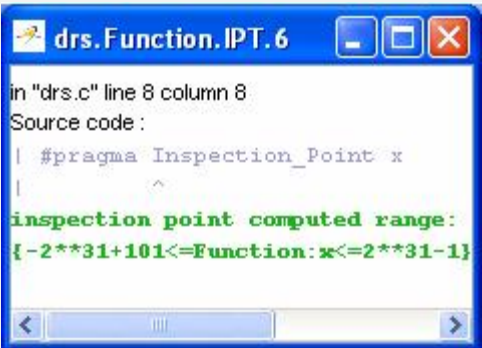
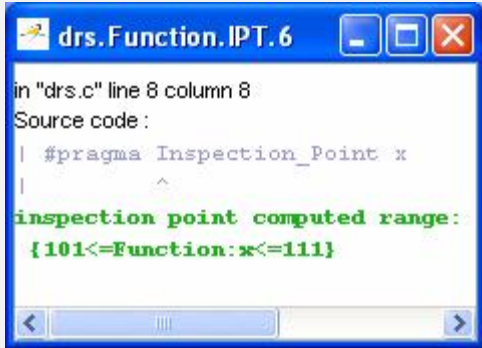
This section describes how DRS reduces oranges on files or modules only.

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"> • With “My_entry” being full range, the addition “+” is orange, • the result “x” is equal to all values between [min+100 max] • Due to previous computations, x+1 can here overflow too, making the addition “+”orange. 	<ul style="list-style-type: none"> • With “My_entry” being bounded to [0,10], the addition “+” is green • the result “x” is equal to [100,110] • Due to previous computations, x+1 can NOT overflow here, making the addition “+” green again.

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

Preparing Source Code for Verification

- “Stubbing” on page 5-2
- “Preparing Code for Variables” on page 5-11
- “Preparing Code for Built-in Functions” on page 5-19
- “Preparing Multitasking Code” on page 5-20
- “Verifying “Unsupported” Code” on page 5-37

Stubbing

In this section...
“Stubbing Overview” on page 5-2
“Manual vs. Automatic Stubbing” on page 5-2
“The Stubbing Options PURE and WORST” on page 5-6
“The Default and Alternative Behavior for Stubbing” on page 5-6
“Function Pointer Cases” on page 5-8
“Stubbing Functions with a Variable Argument Number” on page 5-8
“Finding Bugs in _polyspace_stdstubs.c” on page 5-9

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. (Please refer to “Ignoring Assembly Code” on page 5-37).

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 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.

Default Stubbing	Manual Stubbing	Function ignored
<pre>void main(void) { a = 1; b = 0; a_missing_function(&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(&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(&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.

The Stubbing Options PURE and WORST

External functions are assumed to have no effect (read, write) on global variables. Every external function for which these assumptions are not valid will need to be explicitly stubbed.

Stubbing has an effect on verification duration (“Reducing Verification Time” on page 7-27) and precision.

Consider the example `int f(char *)`. In the verification of this function there are three automatic stubbing approaches which may be considered, aside from manual stubbing.

Using this approach	<code>pragma POLYSPACE_WORST</code>	<code>pragma POLYSPACE_PURE</code>	Default automatic stubbing
implies the assumption of this worst case scenario in the stub	<pre>int f(char *x) { strcpy(x, "the quick brown fox, etc."); return &(x[2]); }</pre>	<pre>int f(char *x) { return strlen(x); }</pre>	<pre>int f(char *x) { *x = rand(); return 0; }</pre>
and then there is manual stubbing to consider.	If the function being modelled by the stub is not accurately represented by any of these approaches to automatic stubbing, then manual stubbing will yield more precise results.		

The Default and Alternative Behavior for Stubbing

Initial Prototype	With <code>pragma POLYSPACE_PURE</code>	With <code>pragma POLYSPACE_WORST</code>	PolySpace default automatic stubbing
<code>void f1(void);</code>	{do nothing}		

Initial Prototype	With pragma POLYSPACE_PURE	With pragma POLYSPACE_WORST	PolySpace default automatic stubbing
int f2 (int u);	Returns $[-2^{31}, 2^{31}-1]$	Returns $[-2^{31}, 2^{31}-1]$ and assumes the ability to write into (int *) u	Returns $[-2^{31}, 2^{31}-1]$
int f3 (int *u);			Assumes the ability to write into *u to any depth and returns $[-2^{31}, 2^{31}-1]$
int* f4 (int u);	Returns an absolute address (AA)	Returns AA or (int *) u and assumes the ability to write into (int *) u	Returns an absolute address
int* f5 (int *u);	Returns an absolute address	Returns $[-2^{31}, 2^{31}-1]$ and assumes the ability to write into *u, to any depth	Assumes the ability to write into *u, to any depth and returns an absolute address
void f6 (void (*ptr)(int), param2)	Does nothing	The function pointed to by ptr will be called with a full-range random value for the integer. Rules for param2 are as above.	
void f7 (void (*ptr)(param2)		Unless the option <code>-permissive-stubber</code> , is used, this function is not stubbed. The parameter (int *) associated with the function pointer is too complicated for PolySpace to stub it, and PolySpace stops. You must stub this function manually. Note If (*ptr) contains a pointer as a parameter, it won't be stubbed automatically and with <code>-permissive-stubber</code> , the function pointer ptr is called with random as a parameter.	

Function Pointer Cases

Function Prototype	Comments
<pre>int f(void (*ptr_ok)(int, char, float), other_type1 other_param1);</pre>	The -permissive-stubber option is not required.
<pre>int f(void (*ptr_ok)(int *, char, float), other_type1 other_param1);</pre>	The -permissive-stubber option is required because of the “int *” parameter of the function pointer passed as an argument
<pre>void _reg(int); int _seq(void *); unsigned char bar(void){ return 0; } void main(void){ unsigned char x=0; _reg(_seq(bar)); }</pre>	<p>Both functions “_reg” and “_seq” are automatically stubbed, but the call to the “bar” function is not exercised by the PolySpace software.</p> <p>The function that is a parameter is only called in stubbed functions if the stubbed function prototype contains a function pointer as parameter.</p> <p>Since here that is a “void *”, its not a function pointer</p>

Stubbing Functions with a Variable Argument Number

PolySpace is capable of stubbing most vararg functions. Nevertheless,

- This can generate imprecision in pointer verification;
- It causes a significant increase in complexity and hence in verification time.

There are two possible ways to deal with this.

- stub manually, or

- put a `#pragma POLYSPACE_PURE "function_1"` on every varargs function that you know to be pure. This can reduce the complexity of pointer verification tenfold.

Consider the following example.

Place this kind of line in any `.c` or `.h` file of the verification:

```
#ifdef POLYSPACE
#define example_of_function(format, args...)
#else
void example_of_function(char * format, ...)
#endif
void main(void)
{
    int i = 3;
    example_of_function("test1 %d", i);
}
```

```
polyspace-c -D POLYSPACE
```

Finding Bugs in `__polyspace_stdstubs.c`

By doing a selective review of oranges, the user can sometimes find bugs located in the `__polyspace_stdstubs.c` file. As for other oranges in the code, some are useless, others highlight real problems. How can we isolate the useful ones?

There are a number of practical ways to make it easy for the user to detect the useful oranges:

- Create the file using approaches with are sympathetic to PolySpace needs. This will yield up to 90% less useless oranges. For instance,
- Use functions that return random values instead of local volatile variables;
- Initialize char variables with a random char instead of a volatile int in order to reduce the number of overflow checks;
- Define an "APPLY_CONSTRAINT()" macro. Such a function will always create an orange check but it will be easy to filter.

- By checking oranges manually in the `__polyspace__stdstubs.c` file: many comments have been added to explain where an orange is expected and why.

Collectively, these features turn the chore of separating out the useful orange warnings into a fast and painless activity.

The user should start by reading IDP checks.

Example

The orange check in `fgets()` is one such check.

```
for (i=0; i < length; i++) /* write in s up to n-1 char */
    s[i] = _polyspace_random_char();
    ^
    IDP
```

This orange check is definitely a significant one. It means that PolySpace could not conclude that the buffer which is given as an argument to `fgets()` is always big enough to contain the specified character count. So, the severity of the problem highlighted depends on how the function is called in the application.

The check shouldn't generally be orange unless it is highlighting a real issue (unless `fgets()` is called very frequently. In that case, try using the `context-sensitivity` or `-inline` options).

Preparing Code for Variables

In this section...

“Assigning Ranges to Variables/Assert?” on page 5-11

“Checking Properties on Global Variables at Any Point: Global assert” on page 5-12

“Modeling Variable Values External to my Application” on page 5-15

“How are Variables Initialized?” on page 5-16

“Verifying Code with Undefined or Undeclared Variables and Functions” on page 5-17

Assigning Ranges to Variables/Assert?

Abstract

How can I use assert in PolySpace?

Explanation

Assert is a UNIX/linux/windows macro that aborts the program if the test performed inside the assertion proves to be false.

Assert failures are real RTEs because they lead to a processor halt. Because of this, PolySpace will produce checks for them. The behavior matches that exhibited during execution, because **all execution paths for unsatisfied conditions are truncated** (red and then gray). Thus it can be assumed that any verification performed downstream of the assert uses value ranges which satisfy the assert conditions.

Also refer to the use of volatile.

Solution

Assert can be used to constrain input variables to values within a particular range, for example:

```
#include <stdlib.h>
```

```
int return_between_bounds(int min, int max)
{
    int ret; // ret is not initialized
    ret = random(); // ret ~ [-2^31, 2^31-1]
    assert ((min<=ret) && (ret<=max));
    // assert is orange because the condition may or may not
    // be fulfilled
    // ret ~ [min, max] here because all execution paths that don't
    // meet the condition are stopped
    return ret;
}
```

Checking Properties on Global Variables at Any Point: Global assert

The global assert mechanism works by inserting a check on each write access to a global variable to ensure it is the range specified.

To use this feature:

- 1 Include the file "pst_gassert.h".
- 2 Create a list of Pst_Global_Assert statements for the variables you are interested in.

This header is located in `<PolySpaceInstallDir>/cinclude`.

The Pst_Global_Assert statement takes the form:

```
Pst_Global_Assert(identifier, test);
```

Where identifier has to be a unique reference for each global assert statement, and test is the logical test to perform on a variable. For example:

```
#include "pst_gassert.h"
int x;

Pst_Global_Assert(1,x>=0);

void main(void)
{
    x=12; // green global assert check on the variable x
```

```

x=0; // green global assert check on the variable x
x=-1; // red global assert check on the variable x
}

```

and associated results, using PolySpace Viewer:

The screenshot shows two windows from the PolySpace Viewer. The main window, titled 'to.c', displays the following C code:

```

1  #include "pst_gassert.h"
2
3  int x;
4
5  Pst Global Assert (1, x>=0);
6
7
8
9  void main(void)
10 {
11     x = 12;
12     x = 0;
13     x = -12;
14 }

```

An error message window titled 'to.main.COR.2' is overlaid on the code, showing the following text:

```

in "to.c" line 13 column 2
Source code :
| x = -12;
| ^
certain failure of global assertion condition [Pst_Global_Assert_1] (variable 'x')

```

The behavior of a global assertion is as follows:

- It defines the properties of global variables;
- At each new write access to a variable which had been the subject of a global assertion, PolySpace uses an extra check to indicate whether the global assert is true or not.

You can create a header file with extern references to the global variables of interest followed by the global assert statements.

Then, use the tools `-include` option to force inclusion of this file into every c file. e.g. "polyspace.h":

```
#ifndef _POLYSPACE_H_
```

```
#define _POLYSPACE_H_

#include "pst_gassert.h"
extern int x;
extern int y;
Pst_Global_Assert(1,x>=0);
Pst_Global_Assert(2,((y>=0) && (y<100)));

#endif /* _POLYSPACE_H */
```

The other activity you may want to do is to initialize the variables at the start of execution to these values. To do this you will need to create a hook into the applications main that you are analyzing or use the `-data-range-specifications` option.

Launching Command

```
polyspace-c -include "polyspace.h" ...
```

Variables Scope

Variables concern external linkage, const variables and not necessary a defined variable (i.e. could be extern with option `-allow-undef-variables`). Static variables are not concerned by this option.

The scalar type allows all modes: Variables of integral type signed or unsigned allow **any** mode (char, short, int, long and long long). It allows also structure fields and arrays cells (of integral type).

```
Pst_Global_Assert(1, x > 0);
Pst_Global_Assert(2, x < x1);
Pst_Global_Assert(3, x1 > 0 && x1 < 128);
Pst_Global_Assert(4, (s.b & 0x7f) == s.b);
Pst_Global_Assert(5, tab[1] != 0);
```

Limitations and Fatal Errors

The feature does not work for pointers, floats (float, double and long double) and struct/union variable:

```

extern int *p;
extern float f_var;
extern void change1(void);
Pst_Global_Assert(6, *p < 300);
Pst_Global_Assert(7, (change1(), 1 == 1));
Pst_Global_Assert(8, ((x = x + 3) > 10));
Pst_Global_Assert(9, x ++ < 100);
Pst_Global_Assert(10, f_var < 10.0f);

```

Modeling 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]  
}
```

How are Variables Initialized?

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

Extern

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

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

Refer to “Before You Review PolySpace Results” on page 8-2 for more information on color propagation.

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

Volatile

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

- 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 (NIV orange):

- `#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`

Verifying Code with Undefined or Undeclared Variables and Functions

The definition and declaration of a variable are two different but related operations that are frequently confused.

Definition

- **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, you must specify the option **Continue even with undefined global variables** (`-allow-undef-variable`) before you start a verification. When you specify this option, PolySpace software considers the variable to be initialized, and to potentially have any value in its full range (see “How are Variables Initialized?” on page 5-16).

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

Declaration

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

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

A declaration provides information about the type of the function or variable. If the function or variable is used in a file where it has not been declared, a compilation error will result.

Preparing Code for Built-in Functions

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

All these functions are declared in the standard list of headers, and can be redefined using their own definitions 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 `<polyspace>/cinclude/__polyspace__stdstubs.c`), `signal.h` ('`signal`' and '`raise`' functions are partially implemented – see `<polyspace>/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`.

Generally, 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()`.

Preparing Multitasking Code

In this section...
“PolySpace Software Assumptions” on page 5-20
“Modelling Synchronous Tasks” on page 5-21
“Modelling Interruptions and Asynchronous Events/Tasks/Threads” on page 5-23
“Are Interruptions Maskable or Preemptive by Default?” on page 5-25
“Shared Variables” on page 5-27
“Mailboxes” on page 5-31
“Atomicity (Can an Instruction be Interrupted by Another)” on page 5-34
“Priorities” on page 5-35

PolySpace Software Assumptions

This section describes the default behavior of the PolySpace software. If your code does not conform to these assumptions, you must make minor modifications to the code before starting verification.

The assumptions are as follows:

- The main procedure must terminate in order for entry-points (or tasks) to start.
- All tasks or entry-points start after the end of the main without any predefined basis regarding: the sequence, priority or preemption. If an entry-point is seen as dead code, it is because the main contains a red error and therefore does not terminate.
- PolySpace considers that there is no atomicity, nor timing constraints.
- Only entry points with `void any_name (void)` as prototype will be considered.

The MathWorks recommends that you read this entire section before applying the rules described below. Some rules are mandatory, and others allow you to gain selectivity.

Modelling Synchronous Tasks

In some circumstances, you must adapt your source code to allow synchronous tasks to be taken into account.

Suppose that an application has the following behavior:

- Once every 10 ms: `void tsk_10ms(void);`
- Once every 30 ms: ...
- Once every 50 ms

These tasks never interrupt each other. They include no infinite loops, and always return control to the calling context. For example:

```
void tsk_10ms(void)
{ do_things_and_exit();
  /* it's important it returns control*/
}
```

However, if you specify each entry-point at launch using the option:

```
polyspace-c -entry-points tsk_10ms,tsk_30ms,tsk_50ms
```

then the results are NOT valid, because each task is only called once.

To address this problem, you must specify that the tasks are purely sequential — that is, that they are functions to be called in a deterministic order. You can do this by writing a function to call each of the tasks in the correct sequence, and then declaring this new function as a single task entry point.

Solution 1

Write a function that calls the cyclic tasks in the right order: this is an **exact sequencer**. This sequencer is then specified at launch time as a single task entry point.

This solution:

- **is very precise;**
- requires knowledge of the exact sequence of events.

For example, the sequencer might be:

```
void one_sequential_C_function(void)
{
    while (1) {
        tsk_10ms();
        tsk_10ms();
        tsk_10ms();
        tsk_30ms ();
        tsk_10ms();
        tsk_10ms();
        tsk_50ms ();
    }
}
```

and the associated launching command:

```
polyspace-c -entry-points one_sequential_C_function
```

Solution 2

Make an **upper approximation sequencer**, taking into account every possible scheduling.

This solution:

- is less precise;
- **is quick to code**, especially for complicated scheduling

For example, the sequencer might be:

```
void upper_approx_C_sequencer(void)
{
    volatile int random;
    while (1) {
        if (random) tsk_10ms();
        if (random) tsk_30ms();
    }
}
```

```

        if (random) tsk_50ms();
        if (random) tsk_100ms();
        .....
    }
}

```

and the associated launching command:

```
polyspace-c -entry-points upper_approx_C_sequencer
```

Note If this is the only entry-point, then it can be added at the end of the main rather than specified as a task entry point.

Modelling Interruptions and Asynchronous Events/Tasks/Threads

You can adapt your source code to allow PolySpace software to consider both *asynchronous* tasks and *interruptions*. For example:

```

void interrupt isr_1(void)
{ ... }

```

Without such an adaptation, interrupt service routines will appear as gray (dead code) in the Viewer. The gray code indicates that this code is not executed and is not taken into account, and so all interruptions and tasks are ignored by PolySpace.

The standard execution model is such that the main is executed initially. Only if the main terminates and returns control (i.e. if it is not an infinite loop and has no red errors) will the entry points be started, with all potential starting sequences being modelled automatically. There are several different approaches which may be adopted to implement the required adaptations.

Solution 1: Where interrupts (ISRs) CANNOT preempt each other

If these 3 following conditions are fulfilled:

- the interrupt functions `it_1` and `it_2` (say) can never interrupt each other;

- each interrupt can be raised several times, at any time;
- they are returning functions, and not infinite loops.

Then these non preemptive interruptions may be grouped into a single function, and that function declared as a entry point.

```
void it_1(void);
void it_2(void);

void all_interruptions_and_events(void)
{ while (1) {
  if (random()) it_1();
  if (random()) it_2();
  ... }
}
```

The associated launching command would be:

```
polyspace-c -entry-points all_interruptions_and_events
```

Solution 2: Where interrupts CAN pre-empt each other

If two ISRs can be each be interrupted by the other, then:

- encapsulate each of them in a loop
- declare each loop as a entry point.

One way of approaching that is to replace the original file with a PolySpace version, as illustrated below.

```
original_file.c
void it_1(void)
{
  ... return;
}

void it_2(void)
{
  ... return;
}
```

```
void one_task(void)
{
    ... return;
}

polyspace.c
void polys_it_1(void)
{
    while (1)
    if (random())
        it_1();
}

void polys_it_2(void)
{
    while (1)
    if (random())
        it_2();
}

void polys_one_task(void)
{
    while (1)
    if (random())
        one_task();
}
```

The associated launching command would be

```
polyspace-c -entry-points polys_it_1,polys_it_2,polys_one_task
```

Are Interruptions Maskable or Preemptive by Default?

For user interruptions, no *implicit* critical section is defined: they all need to be written by hand.

Sometimes, an application which includes interrupts has a critical section written into its main entry point, but shared data is still flagged as unprotected.

This occurs because PolySpace does not distinguish between interrupt service routines and tasks. If you specify an interrupt to be a "-entry-point" entry point, it will have the same priority level as the other procedures declared as tasks ("-entry-points" option). So, because PolySpace makes an **upper approximation of all scheduling and all interleaving**, in this case that **includes the possibility that the ISR might be interrupted by any other task**. There are more paths modelled than could happen during execution, but this has no adverse effect on of the results obtained except that more scenarios are considered than could happen during "real life" execution - and the shared data is not seen as being protected.

To address this, the interrupt needs to be embedded in a specific procedure that uses the same critical section as the one used in the main task. Then, each time this function is called, the task will enter a critical section which will model the behavior of a nonmaskable interruption.

Original files

```
void my_main_task(void)
{
    ...
    MASK_IT;
    shared_x = 12;
    UMASK_IT;
    ...
}
int shared_x ;

void interrupt my_real_it(void)
{ /* which is by specification unmaskable */
    shared_x = 100;
}
```

Additional C files required by PolySpace:

```
#define MASK_IT pst_mask_it()
```



```
#define UMASK_IT pst_umask_it()
void other_task (void)
{
    MASK_IT;
    my_real_it();
    UMASK_IT;
}
```

The associated launch command:

```
polyspace-c \
-D interrupt= \
-entry-points my_main_task,other_task \
-critical-section-begin "pst_mask_it:table" \
-critical-section-end "pst_unmask_it:table"
```

Shared Variables

When PolySpace is launched without any options, all tasks are examined as though concurrent and with no assumptions about priorities, sequence order, or timing. Shared variables in this context will always be considered unprotected, and so will all be shown as orange in the variable dictionary.

The following explicit protection mechanisms can be used to protect the variables:

- critical section
- mutual exclusion

See details below:

- “Differences Between Dictionary and Concurrent Access Graph” on page 5-28
- “Critical Sections” on page 5-29
- “Mutual Exclusion” on page 5-30
- “Semaphores” on page 5-31

Differences Between Dictionary and Concurrent Access Graph

This section explains how the dictionary works, and how it differs to the concurrent access graph.

Consider the following code, which contains 3 tasks

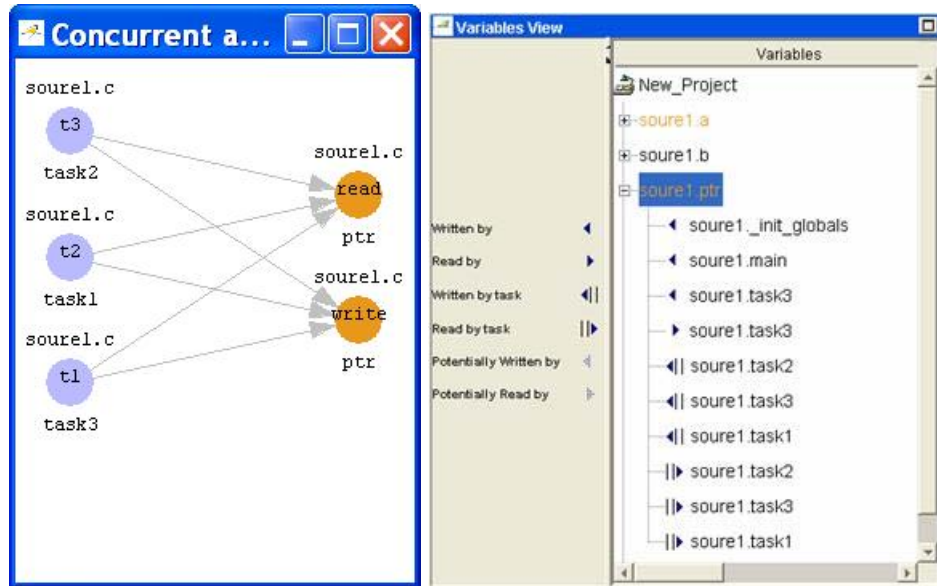
<pre>int *ptr; int a; int b; void main(void) { ptr = &a; }</pre>	<pre>void task1(void) {a ++;</pre>	<pre>void task2(void) { a = a + 10; }</pre>	<pre>void task3(void) { ptr = &b; *ptr = 0; }</pre>
------------------------------------------------------------------------	--------------------------------------	-----------------------------------------------	-------------------------------------------------------------

The variable “ptr” is a simple pointer. ptr itself is not a shared variable because it is only accessed by the main and task3. We can confirm this diagnostic by checking the dictionary which lists

- Writes accesses in the main and in task3
- Read access in task3

But it appears as shared in the dictionary because the concurrent access graph also gathers information regarding the variable “a”, which it points to. This highlights the difference between the dictionary and the concurrent access graph for pointer variables - the concurrent access graph includes both

- Read/write accesses to the pointer itself (ptr in the example below), and
- Read/write accesses to the variable pointed to (a in the example)



Critical Sections

This is the most common protection mechanism found in applications, and is simple to represent in PolySpace:

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

Consider the following example.

Original Code

```
void proc1(void)
{
```

```
    MASK_IT;
    x = 12; // X is protected
    y = 100;
    UMASK_IT;
}
void proc2(void)
{
    MASK_IT;
    x = 11; // X is protected
    UMASK_IT;
    y = 101; // Y is not protected
}
```

File Replacing the Original Include File

```
void begin_cs(void);
void end_cs(void);
#define MASK_IT begin_cs()
#define UMASK_IT end_cs()
```

Command line to launch PolySpace

```
polyspace-c \
-entry-point proc1,proc2 \
-critical-section-begin"begin_cs:label_1" \
-critical-section-end"end_cs:label_1"
```

Mutual Exclusion

Mutual exclusion between tasks or interrupts can be implemented while preparing PolySpace for launching.

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

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

These entry points cannot overlap:

- t1 and t3
- t2, t3 and t4

These entry-points can overlap:

- t1 and t2
- t1 and t4

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

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

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

```
t1 t3  
t2 t3 t4
```

Semaphores

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

Mailboxes

Suppose that an application has several tasks, some of which post messages in a mailbox while others read them asynchronously.

This communication mechanism is possible because the OS libraries provide send and receive procedures. It is likely that the source files will be unavailable because the procedures are part of the OS libraries, but the mechanism needs to be modelled if the verification is to be meaningful.

By default, PolySpace will automatically stub the missing OS send and receive procedures. Such a stub will exhibit the following behavior:

- for send (char *buffer, int length), the content of the buffer will be written only when the procedure is called;
- for receive (char *buffer, int *length), each element of the buffer will contain the full range of values appropriate to that data type.

This and other mechanisms are available, with different levels of precision.

Let PolySpace stub automatically

- quick and easy to code;
- **imprecise** because there is no direct connection between a mailbox sender and receiver. That means that even if the sender is only submitting data within a small range, the full data range appropriate for the type(s) will be for the receiver data.

Provide a **real mailbox** mechanism

- can be very costly (time consuming) to implement;
- can introduce errors in the stubs;
- provides little additional benefit when compared to the upper approximation solution

Provide an **upper approximation of the mailbox**

This models the mechanism such that new read from the mailbox reads **one** of the recently posted messages, but not necessarily the last one.

- quick and easy to code;
- **gives precise results;**

Consider the following detailed implementation of the upper approximation solution.

polyspace_mailboxes.h

```
typedef struct _r {
    int length;
    char content[100];
} MESSAGE;
extern MESSAGE mailbox;
void send(MESSAGE * msg);
void receive(MESSAGE *msg);
```

polyspace_mailboxes.c

```
#include "polyspace.h"
MESSAGE mailbox;
void send(MESSAGE * msg)
{
    volatile int test;
    if (test) mailbox = *msg;
    // a potential write to the mailbox
}
void receive(MESSAGE *msg)
{
    *msg = mailbox;
}
```

Original code

```
#include "polyspace_mailboxes.h"
void t1(void)
{
    MESSAGE msg_to_send;
    int i;
    for (i=0; i<100; i++)
        msg_to_send.content[i] = i;
    msg.length = 100;
    send(&msg);
}
void t2(void)
{
    MESSAGE msg_to_read;
    receive (&msg_to_read);
}
```

PolySpace then proceeds on the assumption that each new read from the mailbox reads a message, but not necessarily the last one.

The associated launching command is

```
polyspace-c -entry-points t1,t2
```

Atomicity (Can an Instruction be Interrupted by Another)

Atomic: In computer programming, atomic describes a unitary action or object that is essentially indivisible, unchangeable, whole, and irreducible

Atomicity: In a transaction involving two or more discrete pieces of information, either all of the pieces are committed or none are.

Instructional decomposition

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

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

Consider a 16 bit target that can manipulate a 32 bit type (an int, for example). In this case, the CPU needs at least two cycles to write to an integer.

Suppose that x is an integer in a multitasking system, with an initial value of 0x0000. Now suppose 0xFF55 is written it. If the operation was not atomic it could be interrupted by another instruction in the middle of the write operation.

- Task 1: Writes 0xFF55 to x.
- Task 2: Interrupts task 1. Depending on the timing, the value of x could be any of 0xFF00, 0x0055 or 0xFF55.

PolySpace considers write/read instructions atomic, so **task 2 can only read 0xFF55**, even if X is not protected (refer to “Shared Variables” on page 5-27).

Critical sections

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

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

More information is available in “Critical Sections” on page 5-29.

Priorities

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

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

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

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

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

Verifying “Unsupported” Code

In this section...
“Ignoring Assembly Code” on page 5-37
“Dealing with Backward “goto” Statements” on page 5-43
“Types Promotion” on page 5-46

Ignoring Assembly Code

You can ignore assembly code during verification using the **Discard assembly code** option (`-discard-asm`). Using this option can deal with many instances of assembly code within a C application, but it is not always a valid route to take.

Ignored assembly instructions will change the behavior of the code. For example, a write access to a shared variable can be written in assembly code. If this write access is ignored, the verification may produce inaccurate results.

In such cases, please refer to “Stubbing” on page 5-2, which applies to functions as well as to stubbed instructions.

PolySpace is designed for C code only. In most cases, the option `-discard-asm` combined with `-asm-begin` and `-asm-end` can be used to instruct PolySpace to discard a number of assembly code constructs:

- “Example: Ignore All Statements, the Rest of the Function Remains Unchanged” on page 5-38
- “Example: Automatic Stubbing” on page 5-40
- “Examples: Empty Body” on page 5-41
- “Example: #asm and #endasm Support” on page 5-42
- “Example: What to Do If `-discard-asm` Fails to Parse an asm Code Section” on page 5-42

Example: Ignore All Statements, the Rest of the Function Remains Unchanged

Discarding assembly code by using the `-discard-asm` is an acceptable approach where ignoring the assembly instructions will have no impact on the remainder of the function.

Also refer to the “Manual versus automatic stubbing”

```
int f(void)
{
    asm ("% reg val; mtmsr val;");
    asm ("\tmove.w #$2700,sr");
    asm ("\ttrap #7");
    asm(" stw r11,0(r3) ");
    assert (1); // is green
    return 1;
}

int other_ignored6(void)
{
#define A_MACRO(bus_controller_mode) \
    __asm__ volatile("nop"); \
    __asm__ volatile("nop"); \
    __asm__ volatile("nop"); \
    __asm__ volatile("nop"); \
    __asm__ volatile("nop"); \
    __asm__ volatile("nop")
    assert (1); // is green
    A_MACRO(x);
    assert (1); // is green
    return 1;
}

int pragma_ignored(void)
{
    #pragma asm
    SRST
    #pragma endasm
    assert (1); // is green
}
```

```

int other_ignored2(void)
{
    asm "% reg val; mtmsr val;";
    asm mtmsr val;
    assert (1); // is green
    asm ("px = pm(0,%2); \
        %0 = px1; \
        %1 = px2;"
        : "=d" (data_16), "=d" (data_32)
        : "y" ((UI_32 pm *)ram_address):
        "px");
    assert (1); // is green
}

int other_ignored1(void)
{
    __asm
    {MOV R8,R8
     MOV R8,R8
     MOV R8,R8
     MOV R8,R8
     MOV R8,R8}
    assert (1); // is green
}

int GNUC_include (void)
{
    extern int __P (char *__pattern, int __flags,
    int (*__errfunc) (char *, int),
    unsigned *__pglob) __asm__ ("glob64");
    __asm__ ("rorw $8, %w0" \
        : "=r" (__v) \
        : "0" ((guint16) (val)));
    __asm__ ("st g14,%0" : "=m" (*(AP)));
    __asm__(" \
        : "=r" (__t.c) \
        : "0" (((union { int i, j; } *) (AP))++->i));
    assert (1); // is green
    return (int) 3 __asm__("% reg val");
}

```

```
    }

    int other_ignored3(void)
    {
        __asm {ldab 0xffff,0;trapdis;};
        __asm {ldab 0xffff,1;trapdis;};
        assert (1); // is green
        __asm__ ("% reg val");
        __asm__ ("mtmsr val");
        assert (1); // is green
        return 2;
    }

    int other_ignored4(void)
    {
        asm {
            port_in: /* byte = port_in(port); */
            mov EAX, 0
            mov EDX, 4[ESP]
            in AL, DX
            ret
            port_out: /* port_out(byte,port); */
            mov EDX, 8[ESP]
            mov EAX, 4[ESP]
            out DX, AL
            ret }
        assert (1); // is green
    }
}
```

Example: Automatic Stubbing

You must use the `-discard-asm` option.

PolySpace detects that no body is defined, and automatically creates a stub.

```
asm int m(int tt);
```

Also refer to the “Manual versus Automatic stubbing” section

Examples: Empty Body

You must use the `-discard-asm` option.

```
#pragma inline_asm(ex1, ex2)

#pragma inline_asm(ex1, ex2)
int ex1(void)
{
    % reg val;
    mtmsr val;
    return 3;
};
int ex2(void)
{
    % reg val;
    mtmsr val;
    assert (1); // is dead code because the whole body is empty
    return 3;
};
```

```
#pragma inline_asm(ex3)

#pragma inline_asm(ex3)
int ex3(void)
{
    % reg val;
    mtmsr val;
    return 3;
};
```

Compiler specific implementation: an empty body is provided

```
asm int l(int tt){}
```

Compiler specific implementation: all statements in the function body are ignored.

```
asm
int h(int tt)
{
    % reg val; // is ignored
```

```
    mtmsr val; // is ignored
    return 3; // is ignored
};
```

Also refer to “Stubbing” on page 5-2.

Example: #asm and #endasm Support

Using `#asm` and `#endasm` allows fragments of (typically) assembly code to be disregarded by PolySpace, regardless of whether or not you use the `-discard-asm`.

Consider the following example.

```
void test(void)
{
    #asm
    mov _as:pe, reg
    jre _nop
    #endasm
    int r;
    r=0;
    r++;
}
```

Explanation

By default, the usage of `#asm` and `#endasm` requires the usage of the `-asm-begin` and `-asm-end` options in the following way. The syntax to use this facility when launching PolySpace in batch mode is:

```
polyspace-c -asm-begin asm -asm-end endasm
```

Example: What to Do If -discard-asm Fails to Parse an asm Code Section

Occasionally, the `-discard-asm` option does not deal with a particular assembly code construction, particularly when the code fragment is compiler specific

Note You could also consider using the `-asm-begin` and `-asm-end` options instead of the following approach).

Consider this example.

```

1 int x=12;
2
3 void f(void)
4 {
5 #pragma will_be_ignored
6 x =0;
7 x= 1/x;    // no colour is even displayed
8           // not even C code
9 #pragma was_ignored
10 x++;
11 x=15;
12 }
13
14 void main (void)
15 {
16 int y;
17 f();
18 y = 1/x + 1 / (x-15); // x is equal to 15
19
20 }
```

As shown in this example, any text or code placed between the two `#pragma` statements is ignored by the verification. This allows any unsupported construction to be ignored without changing the meaning of the original code. The options to enable this feature are accessible through the Graphical Interface PolySpace Launcher or in batch mode:

```
polyspace-c -asm-begin will_be_ignored -asm-end was_ignored
```

Dealing with Backward “goto” Statements

PolySpace is not designed to support backward “goto” statements. However, macros provide a solution in most cases. In general, verifications that includes

backward “goto” statements stop at an early stage, and a message appears saying that backward “goto” statements are not supported.

Macros provided with the PolySpace software can work around this limitation **as long as the “goto” labels and jump instructions are in the same code block (and in the same scope).**

To insert these macros into the code:

- 1 Edit the C file containing the “goto” statements;
- 2 Add `#include pstgoto.h` at the beginning of the file (located in `<PolySpaceInstallDir>/cinclude`).
- 3 Go to the beginning of the block containing the “goto” statements.
- 4 Insert the `USE_1_GOTO(<tag>)` macro call after the variable declarations (local to the block).
- 5 Insert the `EXIT_1_GOTO(<tag>)` macro call before the end of this same block (take care with the closing bracket `"}"`).
- 6 Replace `"goto <tag>"` with `"GOTO(<tag>)"`.

For example, the following code would cause a verification to terminate:

```
{
/* local variable declarations */
int x; ...
/* Instructions */
...
label1:
...
goto label1
...
}
```

You could address this problem as follows:

```
/* the pstgoto.h file is provided by PolySpace and its path */
```

```
{
/* local variable declarations */
int x; ...
USE_1_GOTO(label1);
/* Instructions */
...
label1:
...
GOTO(label1);
...
EXIT_1_GOTO(label1);
}
```

The code block may contain many instances of backward “goto” statements. Using matching `USE_n_GOTO()` and `EXIT_n_GOTO()` statements will address this (for example, `USE_2_GOTO()`, `USE_3_GOTO()`, etc.)

Note You must copy `pstgoto.h` from `<PolySpaceInstallDir>/cinclude`, and add it to the list of include directories (`-I`).

The code block may also use several different tags. You can use multiple “tag” parameters to address these situations. For example, use:

```
USE_n_GOTO (<tag 1>, <tag 2>, ..., <tag n>);
EXIT_n_GOTO(<tag 1>, <tag 2>, ..., <tag n>);
```

Consider the following example:

Original Code	Modified Code for Verification
<pre>{ . Reset: . { { if (X) goto Reset; } { if (Y) goto Reset; } }</pre>	<pre>{ USE_1_GOTO(Reset); Reset: { { if (X) GOTO(Reset); } { if (Y) GOTO(Reset); } } EXIT_1_GOTO(Reset);</pre>

Types Promotion

- “Unsigned Integers Promoted to Signed Integers” on page 5-46
- “What are the Promotions Rules in Operators?” on page 5-47
- “Example” on page 5-48

Unsigned Integers Promoted to Signed Integers

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

For example, the execution of the following 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 since y is unsigned */

```

A negative cast into unsigned gives a big value, which has to be bigger than 7. This big value can never be ≤ 7 , and so the assertion can never hold true.

What are the Promotions 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.
- Be careful when verifying any operation between variables of different types without an explicit cast.

Example

Consider the integral promotion aspect of the ANSI-C standard (see 6.2.1 in ISO/IEC 9899:1990). On arithmetic operators like +, -, *, % and /, an integral promotion is applied on both operands. From the PolySpace viewpoint, that can imply an OVFL or a UNFL orange check.

```
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 UOVFL: 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 6-2
- “Running Verifications on PolySpace Server” on page 6-3
- “Running Verifications on PolySpace Client” on page 6-19
- “Running Verifications from Command Line” on page 6-24

Types of Verification

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

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

Running Verifications on PolySpace Server

In this section...

“Starting Server Verification” on page 6-3

“What Happens When You Run Verification” on page 6-4

“Managing Verification Jobs Using the PolySpace Queue Manager” on page 6-5

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

“Viewing Verification Log File on Server” on page 6-9

“Stopping Server Verification Before It Completes” on page 6-11

“Removing Verification Jobs from Server Before They Run” on page 6-12

“Changing Order of Verification Jobs in Server Queue” on page 6-13

“Purging Server Queue” on page 6-13

“Changing Queue Manager Password” on page 6-15

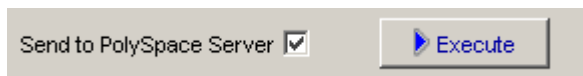
“Sharing Server Verifications Between Users” on page 6-15

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 3, “Setting Up a Verification Project”.
- 3 Select the **Send to PolySpace Server** check box next to the **Execute** button in the middle of the Launcher window.



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

4 Click **Execute**.

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

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

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

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

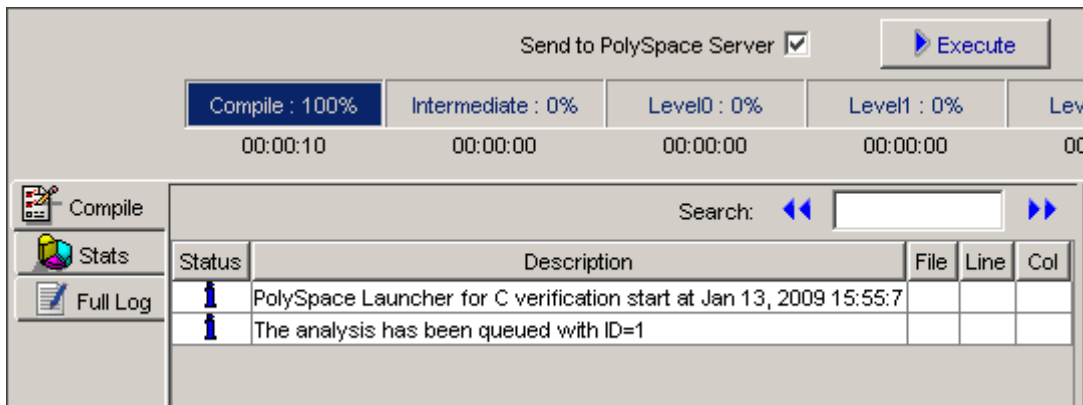
What Happens When You Run Verification

The verification has three main phases:

- 1 Checking syntax and semantics (the compile phase). Because PolySpace software is independent of any particular 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 “Main Generator Options (-main-generator) for PolySpace” 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

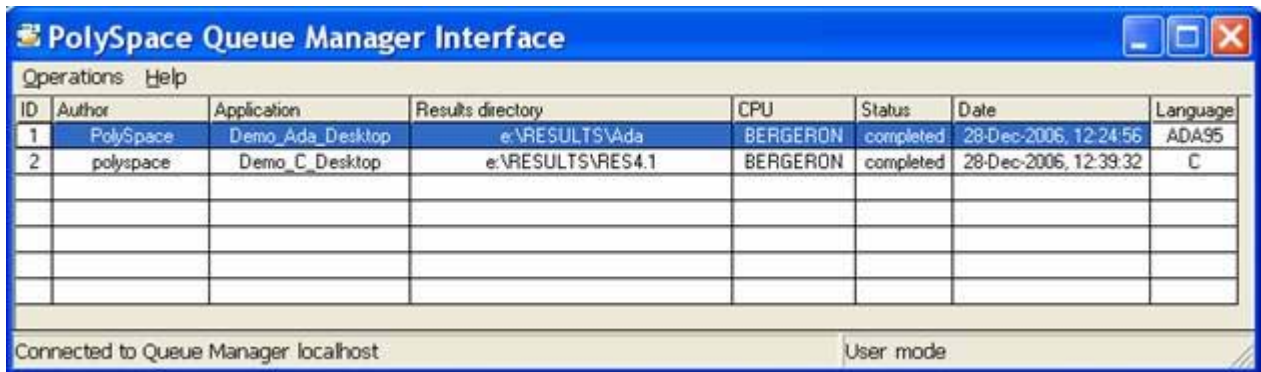
You manage all server verifications using the PolySpace Queue Manager (also called the PolySpace Spooler). The PolySpace Queue Manager allows you to move jobs within the queue, remove jobs, monitor the progress of individual verifications, and download results.

To manage verification jobs on the PolySpace Server:

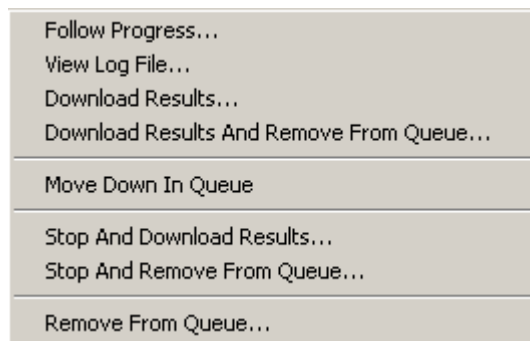
- 1 Double-click the **PolySpace Spooler** icon:




The **PolySpace Queue Manager Interface** opens.



- 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:

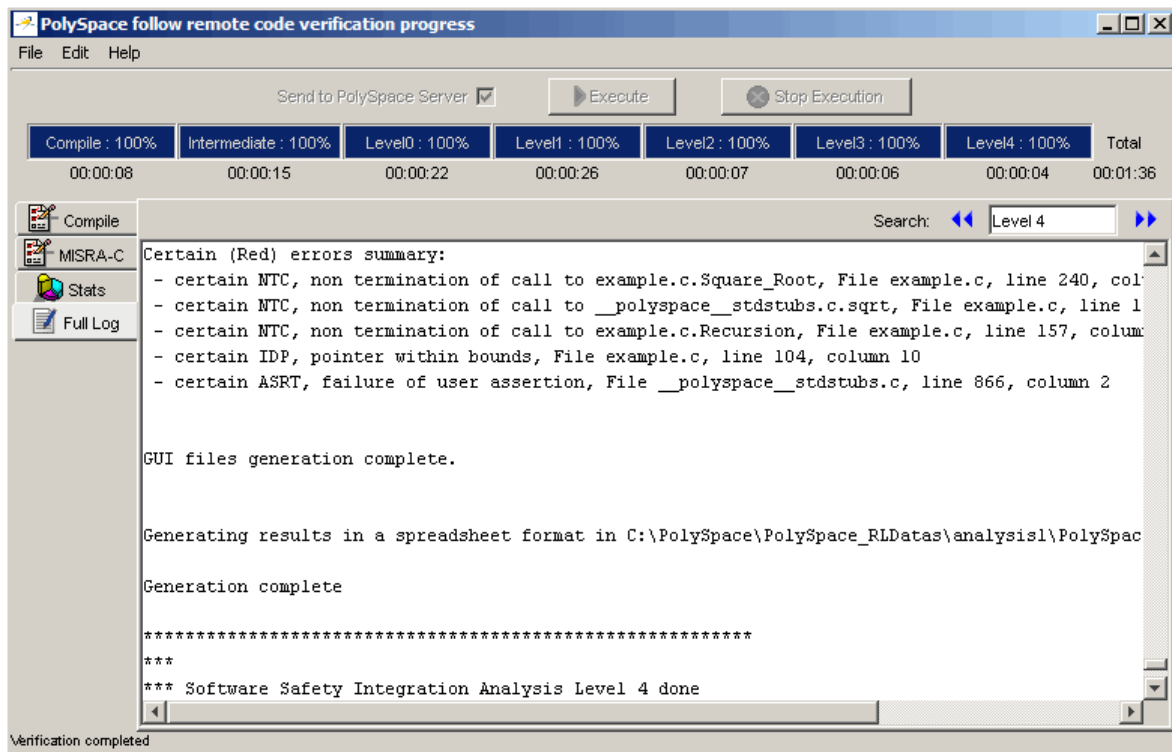


The **PolySpace Queue Manager Interface** opens.

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

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

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




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

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

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

arrows to search forward. Click on any message in the log to get details about the message.

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

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

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

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

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

Viewing Verification Log File on Server

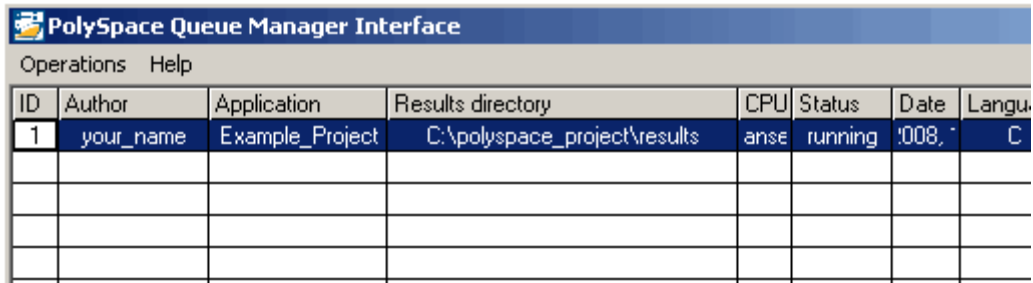
You can view the log file of a server verification using the PolySpace Queue Manager.

To view a log file on the server:

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



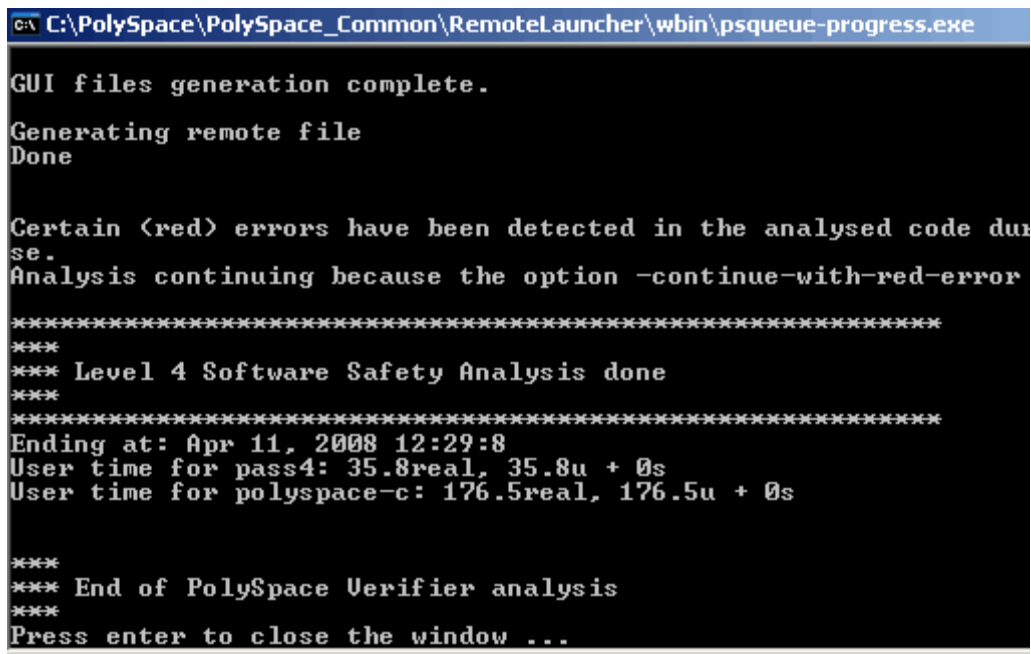
The **PolySpace Queue Manager Interface** opens.



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

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

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



```

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

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

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

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

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

Stopping Server Verification Before It Completes

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

To stop a server verification:

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



The **PolySpace Queue Manager Interface** opens.

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

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

Removing Verification Jobs from Server Before They Run

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

Note If the job has started running, you must stop the verification before you can remove the job (see “Stopping Server Verification Before It Completes” on page 6-11). Once you have aborted a verification, you can remove it from the queue.

To remove a job from the server queue:

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



The **PolySpace Queue Manager Interface** opens.

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

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

The job is removed from the queue.

Changing Order of Verification Jobs in Server Queue

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

To move a job within the server queue:

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



The **PolySpace Queue Manager Interface** opens.

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

- 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.

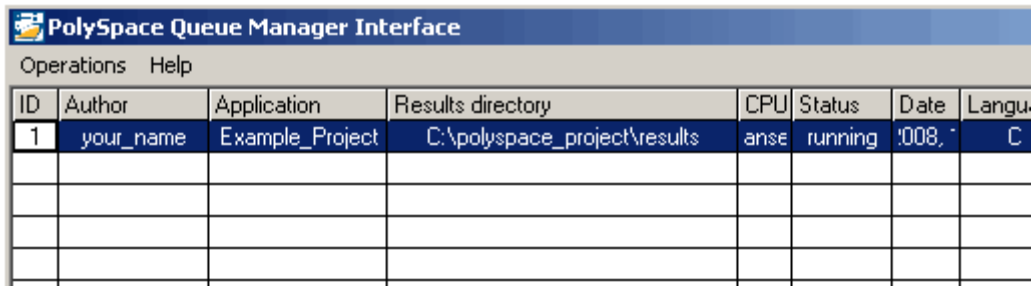
Note You must have the queue manager password to purge the server queue.

To purge the server queue:

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

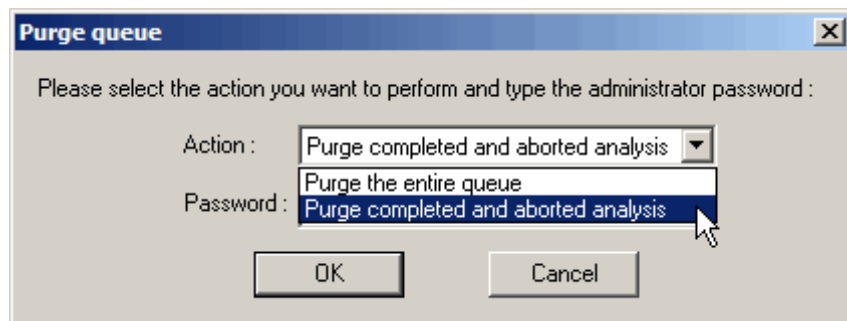


The **PolySpace Queue Manager Interface** opens.

The PolySpace Queue Manager Interface window, showing a table with columns for ID, Author, Application, Results directory, CPU, Status, Date, and Language. The first row contains the following data: ID: 1, Author: your_name, Application: Example_Project, Results directory: C:\polyspace_project\results, CPU: anse, Status: running, Date: '008, and Language: C.

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

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



- 3 Select one of the following options:

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

4 Enter the Queue Manager Password.

5 Click **OK**.

The server queue is purged.

Changing Queue Manager Password

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

Note The default password is administrator.

To set the Queue Manager password:

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

The PolySpace Queue Manager Interface opens.

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

The Change Administrator Password dialog box opens.

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

The password is changed.

Sharing Server Verifications Between Users

Security of Jobs in Server Queue

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

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

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

analysis-keys.txt File

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

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

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

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

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

The fields in the file are tab-separated.

The file cannot contain blank lines.

Example:

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

Sharing Verifications Between Accounts

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

To share a verification with another user:

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

The second user can then download or manage the verification.

Magic Key to Share Verifications

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

The format for a magic key is as follows:

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

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

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

If analysis-keys.txt File is Lost or Corrupted

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

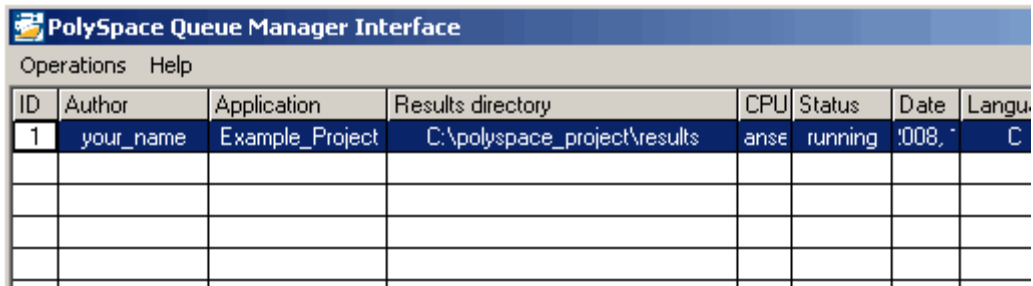
Note You must have the queue manager password to use Administrator Mode.

To use administrator mode:

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



The **PolySpace Queue Manager Interface** opens.

A screenshot of the PolySpace Queue Manager Interface window. The window title is "PolySpace Queue Manager Interface" and it has a menu bar with "Operations" and "Help". Below the menu bar is a table with the following columns: ID, Author, Application, Results directory, CPU, Status, Date, and Language. The first row of the table contains the following data: ID: 1, Author: your_name, Application: Example_Project, Results directory: C:\polyspace_project\results, CPU: anse, Status: running, Date: '008, and Language: C. The rest of the table is empty.

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

2 Select **Operations > Enter Administrator Mode**.

3 Enter the Queue Manager **Password**.

4 Click **OK**.

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

Running Verifications on PolySpace Client

In this section...

“Starting Verification on Client” on page 6-19

“What Happens When You Run Verification” on page 6-20

“Monitoring the Progress of the Verification” on page 6-21

“Stopping Client Verification Before It Completes” on page 6-22

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 2,000 assignments and calls, the verification will stop and you will receive an error message.

To start a verification that runs on a client:

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

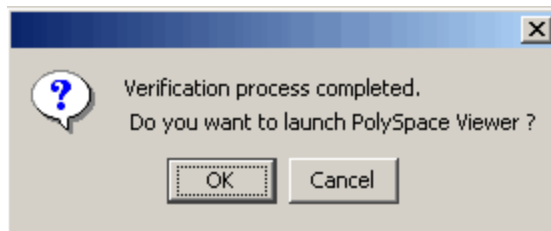


- 6 If you see a caution that PolySpace software will remove existing results from the results directory, click **Yes** to continue and close the message dialog box.

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

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

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



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

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

What Happens When You Run Verification

The verification has three main phases:

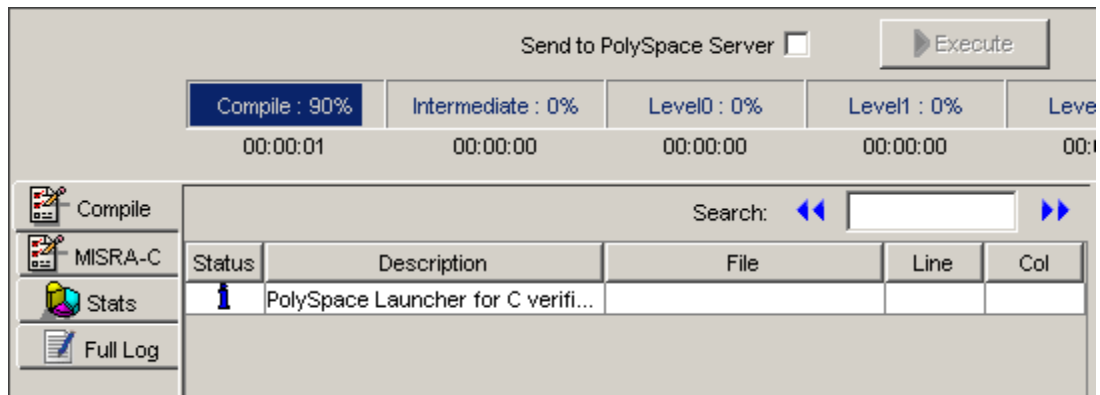
- 1 Checking syntax and semantics (the compile phase). Because PolySpace software is independent of any particular C compiler, it ensures that your code is portable, maintainable, and complies with ANSI standards.
- 2 Generating a main if it does not find a main and the **Generate a Main** option is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

“Main Generator Options (-main-generator) for PolySpace” 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.

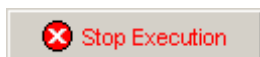
You can search the full log by entering a search term in the **Search in the log** box and clicking the left arrows to search backward or the right arrows to search forward.

Stopping Client Verification Before It Completes

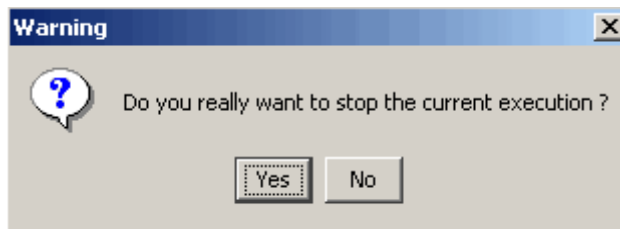
You can stop the verification before it completes. If you stop the verification, results will be incomplete, and if you start another verification, the verification starts over from the beginning.

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 6-24
“Managing Verifications in Batch” on page 6-24

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-c`
- **Client verification** —`polyspace-remote-desktop-c`

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

For example, `polyspace-remote-desktop-c -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-c.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 7-2
- “Compile Errors” on page 7-6
- “Link Messages” on page 7-12
- “Stubbing Errors” on page 7-17
- “Intermediate Language Errors” on page 7-25
- “Reducing Verification Time” on page 7-27

Verification Process Failed Errors

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

Overview

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

Hardware Does Not Meet Requirements

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

www.mathworks.com/products/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 3-8.

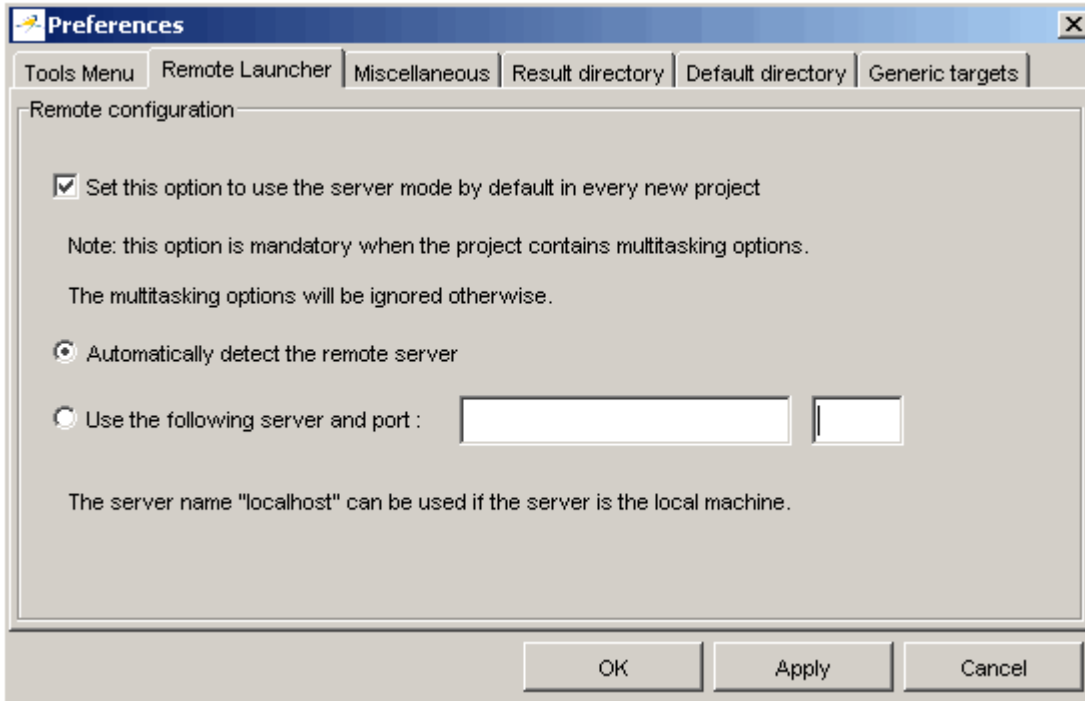
PolySpace Software Cannot Find the Server

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

Error: Unknown host :

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

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



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

Limit on Assignments and Function Calls

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

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

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

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

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

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

Compile Errors

In this section...
“Overview” on page 7-6
“Examining the Compile Log” on page 7-6
“Syntax error” on page 7-8
“Undeclared identifier” on page 7-8
“No such file or directory” on page 7-9
“Compilation errors with keywords: @interrupt, @address(0xABCDEF)” on page 7-9

Overview

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

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

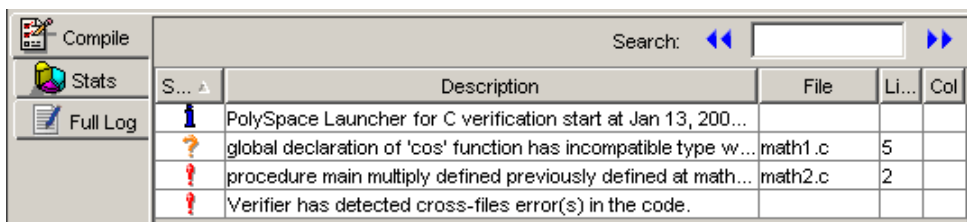
Examining the Compile Log

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

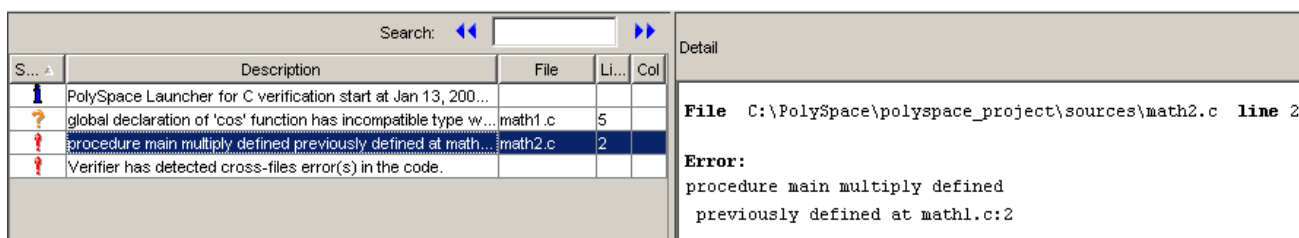
To examine errors in the Compile log:

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

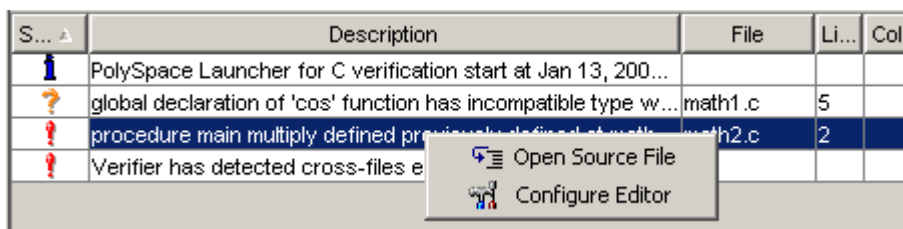
A list of compile phase messages appear in the log part of the window.



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



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



The file opens in your text editor.

Note You must configure a text editor before you can open source files. See “Configuring Text and XML Editors” on page 3-16.

- 4 Correct the error and run the verification again.

Syntax error

Log File	Code Used
<pre>Verifying compilation.c compilation.c:3: syntax error; found `x' expecting `;' compilation.c:3: undeclared identifier `x'</pre>	<pre>void main(void) { int far x; x = 0; x++; }</pre>

The “far” keyword is unknown in ANSI C. At “compilation” time, it therefore causes confusion - should it be a variable, or maybe a qualifier? The construction “int far x;” is illegal without any further information, and hence it is a syntax error. Here are some possible corrections:

- Remove *far* from the source code;
- Define *far* as a qualifier such as `const` or `volatile`;
- Remove *far* artificially by specifying a compilation flag like: “-D far= “ (with a space after the equal sign).

Note If you need to specify -D compilation flags which are generic to the project, then using the `-include` option may be the most efficient solution. Refer to “How to Gather Compilation Options Efficiently” on page 4-19.

Undeclared identifier

Log File	Code Used
<pre>compilation.c:3: undeclared identifier `x'</pre>	<pre>void main(void) { x = 0; x++; }</pre>

Should `x` be a float, an int or a char? The type is unknown, and therefore the compilation stops.

Sometimes variables are implicitly defined by certain cross compilers. They need to be declared before verification begins, as PolySpace software has no knowledge about implicit variables.

Similarly “__SP” can be interpreted as a reference to the stack pointer by some compilers, which may be dealt with by using the -D compilation flag.

Note If you need to specify -D compilation flags which are generic to the project, then using the -include option may be the most efficient solution. Refer to “How to Gather Compilation Options Efficiently” on page 4-19.

No such file or directory

Log File	Code Used
compilation.c:1: one_file.h: No such file or directory	#include "one_file.h"
compilation.c:1: catastrophic error: could not open source file "one_file.h"	#include "one_file.h"

The file called “one_file.h” is missing. The include directory holding this file must be made known to PolySpace. Refer to the -I option in the launcher.

These files are essential for PolySpace to complete the compilation. They will be used:

- for data coherency;
- for automatic stubbing.

Compilation errors with keywords: @interrupt, @address(0xABCDEF)

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 a first character. But in this case, the problem cannot be addressed by means of a compilation flag, nor with an -include file.

In this case, you need to use the -post-preprocessing option.

When this option is applied, the specified script file or command is run just after the preprocessing phase on each source file. The script executes on each preprocessed c file. The command should be designed to process the standard output from preprocessing and produce its results in accordance with that standard output.

Note You can find each preprocessed file in the results directory in the zipped file ci.zip located in <results>/ALL/SRC/MACROS. The extension of the preprocessed file is .ci.

It is important to preserve the number of lines in the preprocessed .ci file. Adding a line or removing one could result in some unpredictable behavior on the location of checks and MACROS in the PolySpace viewer.

Example:

Save the following script in a file named `myscript.pl`.

```
#!/usr/bin/perl
bin STDOUT;

# Process every line from STDIN until EOF
while ($line = <STDIN>)
{
# Replace keyword titi with toto
$line =~ s/titi/toto/g;
# Remove @interrupt (replace with nothing)
$line =~ s/@interrupt/ /g;

# DONT DELTE: Print the current processed line to STDOUT
print $line;
}
```

Now use the command `post-preprocessing-command`
`%POLYSPACE_C%\Verifier\tools\perl\win32\bin\perl.exe`
`<absolute path to myscript.pl>\myscript.pl` to run the above script
on each preprocessed c file.

Note If you are running PolySpace software version 5.1 (r2008a) or later on a Windows system, you cannot use Cygwin shell scripts. Since Cygwin is no longer included with PolySpace software, all files must be executable by Windows. To support scripting, the PolySpace installation now includes Perl. You can access Perl in

```
%POLYSPACE_C%\Verifier\tools\perl\win32\bin\perl.exe
```

Link Messages

In this section...

“Overview” on page 7-12

“Function: Wrong Argument Type” on page 7-12

“Function: Wrong Argument Number” on page 7-13

“Variable: Wrong Type” on page 7-14

“Variable: Signed/Unsigned” on page 7-14

“Variable: Different Qualifier” on page 7-15

“Variable: Array Against Variable” on page 7-15

“Variable: Wrong Array Size” on page 7-16

“Missing Required Prototype for varargs” on page 7-16

Overview

This section gives some examples of link errors.

Note Looking at the preprocessed code can help to find errors faster. They are located in the <<results directory>>/C-ALL/ or <<results directory>>/ALL/SRC/MACROS. These files have a .ci extension.

Function: Wrong Argument Type

PolySpace Output:

```
Verifying cross-files ANSI C compliance ...
Error: global declaration of 'f' function has incompatible type with its definition
      declared function type has 'arg 1' type incompatible with definition
      declared 'pointer' (32) type incompatible with defined 'float' (32) type
```

PolySpace Output:

```
int f(float y)
{
  int r;
  r=12;
}
```

```
int f(int *y);

void main(void)
{
  int r;
  r = f(&r);
}
```

Here, the first parameter for the “f” function is either a float or a pointer to an integer - but either way, the global declaration must match the definition. The error is explained in the textual output generated by PolySpace during the linking phase.

Function: Wrong Argument Number**PolySpace Output:**

Verifying cross-files ANSI C compliance ...

Error: global declaration of 'f' function has incompatible type with its definition
declared function type has incompatible args. number with definition

```
int f(int y, int z)
{
  int r;
  r=12;
}
```

```
int f(int y);

void main(void)
{
  int r;
  r = f(r);
}
```

These two functions haven't the same number of arguments, which would result in non determinism during execution. The error is explained in the textual output generated by PolySpace during the linking phase.

Variable: Wrong Type

PolySpace Output:

```
Verifying cross-files ANSI C compliance ...  
Error: global declaration of 'x' variable has incompatible type with its definition  
declared 'float' (32) type incompatible with defined 'int' (32) type
```

```
extern float x;
```

```
int x;  
void main(void)  
{}
```

The “x” variable must be declared in the same way in every file. If a variable x is as an integer equal to 1, which is 0x0001, what does this value mean when seen as a float? It could result in a NAN (Not A Number) during execution.

Variable: Signed/Unsigned

PolySpace Output:

```
Verifying cross-files ANSI C compliance ...  
Error: global declaration of 'x' variable has incompatible type with its definition  
declared 'unsigned' type incompatible with defined 'signed' type
```

```
extern unsigned char x;
```

```
char x;  
void main(void)  
{}
```

Consider the 8 bit binary value 10000010. Given that a char is coded in 8 bits, it is not clear how this should be considered in the code snippet shown; maybe 130 (unsigned), maybe -126 (signed). PolySpace highlights the ambiguity.

Variable: Different Qualifier

PolySpace Output:

```
Verifying cross-files ANSI C compliance ...
Warning: global declaration of 'x' variable has incompatible type with its definition
  declared 'non qualified' type incompatible with defined 'volatile' type
  'volatile' qualifier used
```

```
extern int x;
```

```
volatile int x;
```

```
void main(void)
{
}
```

The qualifier taken into account by PolySpace is the one with the most onerous implications for the verification. However, there is doubt regarding which statement is correct, and so PolySpace generates a warning.

Variable: Array Against Variable

PolySpace Output:

```
Verifying cross-files ANSI C compliance ...
Error: global declaration of 'x' variable has incompatible type with its definition
  declared 'array' (384) type incompatible with defined 'int' (32) type
```

```
extern int x[12];
```

```
int x;
```

```
void main(void)
{
}
}
```

The real allocated size for the x variable is one integer. Any function attempting to manipulate x[] would corrupt the memory. PolySpace textual output highlights this error.

Variable: Wrong Array Size

PolySpace Output:

```
Verifying cross-files ANSI C compliance ...
Error: global declaration of 'x' variable has incompatible type with its definition
      declared array type has 'upper bound' 12 out of range 5
```

```
extern int x[12];
```

```
int x[5];
```

```
void main(void)
{
}

```

The real allocated size for the x variable is five integers. Any function attempting to manipulate x[] between x[5] and x[11] will in fact corrupt the memory. PolySpace textual output highlights this error.

Missing Required Prototype for varargs

PolySpace Output:

```
Verifying cross-files ANSI C compliance ...
Error: missing required prototype for varargs. procedure 'g'.
```

```
void g(int, ...);
```

```
void f(void)
{
g(12, abcde ,40);
}
```

```
void main(void)
```

```
{
g(4);
}
```

The prototype for “g” must also be declared when the main is used.

To get rid of this error without modifying the main (by adding the line “void g(int, ...)”), you can include that line in a new file called (say) generic_for_example.h and then use the option –include “c:\PolySpace\generic_for_example.h” when your verification is launched.

Stubbing Errors

In this section...
“Errors when Compiling <code>_polyspace_stdstubs.c</code> ” on page 7-17
“Errors when Creating Automatic Stubs” on page 7-22

Errors when Compiling `_polyspace_stdstubs.c`

- “Standard Error Messages” on page 7-17
- “Troubleshooting” on page 7-19

Standard Error Messages

There may be occasions when a code set compiles for a target but when that same code is verified with PolySpace, an error message is generated during the compilation phase for `__polyspace_stdstubs.c`.

The following are example error messages. They highlight conflicts between a standard library function which is included as part of the application, and one of the standard stubs that PolySpace uses in place of that function.

Stubbing standard library functions ...

```
C-STUBS/_polyspace_stdstubs.c:1117: string.h: No such file  
or directory
```

```
Verifying C-STUBS/_polyspace_stdstubs.c
```

```
C-STUBS/_polyspace_stdstubs.c:1118: syntax error; found  
'strlen' expecting `;'
```

```
C-STUBS/_polyspace_stdstubs.c:1120: syntax error; found `i'  
expecting `;'
```

```
C-STUBS/_polyspace_stdstubs.c:1120: undeclared identifier `i'
```

Stubbing standard library functions ...

Verifying C-STUBS/___polyspace__stdstubs.c

```
Error: missing required prototype for varargs. procedure
'sprintf'.
```

Stubbing standard library functions ...

Verifying C-STUBS/___polyspace__stdstubs.c

```
C-STUBS/___polyspace__stdstubs.c:3027: missing parameter 4 type
```

```
C-STUBS/___polyspace__stdstubs.c:3027: syntax error; found '\n'
expecting `)'
```

```
C-STUBS/___polyspace__stdstubs.c:3027: skipping '\n'
```

```
C-STUBS/___polyspace__stdstubs.c:3037: undeclared identifier '\n'
```

The code uses standard library functions such as `sprintf` and `strcpy` and the examples above suggest problems with these functions.

Such problems can best be addressed by restarting the verification including the header file containing the prototype and the required definitions, as used during compilation for the target. The least invasive way of doing this is to use the `-I` option.

Failing that, a selection of files is provided that contain stubs for most standard library functions which can be used in place of automatic stubbing.

For this to work effectively, it is important for you to provide the **correct include file** for the function. In the following example, the standard library function is `strlen`. This assumes that `string.h` has been included. Because the `string.h` file may differ between targets, **there are no default include directories for PolySpace**.

Thus, if the compiler has implicit include files, they must be manually specified as illustrated in the following example.

```

(_polyspace_stdstubs.c located in <<results_dir>>/C-ALL/C-STUBS)

__polyspace_stdstubs.c
#if defined(_polyspace_strlen) || ... || defined(_polyspace_strtok)
#include <string.h>
size_t strlen(const char *s)
{
    size_t i=0;
    while (s[i] != 0)
        i++;
    return i;
}
#endif /* __polyspace_strlen */

```

If problems remain, refer to the solutions below.

Troubleshooting

There may be occasions where restarting the verification including the missing header file(s) using the -I option will not solve the problem. There are 3 potential solutions:

- “Where precision is important and preparation time is not a problem” on page 7-19
- “Where preparation time is short or problems remain after trying solution 1” on page 7-20
- “Where all other attempts have failed” on page 7-21

Where precision is important and preparation time is not a problem.

- 1 Copy <<results_dir>>/C-ALL/C-STUBS/ __polyspace_stdstubs.c to the source directory and rename it polyspace_stubs.c.
- 2 This file contains the whole list of stubbed functions, user functions and standard library functions. For example:

```

#define __polyspace_strlen
#define a_user_function

```

- 3 Find the problem function in the file.

```
#if defined(_polyspace_strlen) || ... || defined(_polyspace_strtok)
#include <string.h>
size_t strlen(const char *s)
{
    size_t i=0;
    while (s[i] != 0)
        i++;
    return i;
}
#endif /* __polyspace_strlen */
```

This is the stubbed definition for the function causing the problem, and hence the verification requires the applications own string.h include file.

- 4** EITHER extract the relevant part of that file for inclusion in the verification.

For example, for strlen:-

```
string.h
// put it in the /homemade_include directory
typedef int size_t;
size_t strlen(const char *s);
```

OR, preferably, provide the string.h file that contains the real prototype and type definitions for the stubbed function.

- 5** Specify the path for the include files and relaunch PolySpace:

```
polyspace-c -I /homemade_include
```

or

```
polyspace-c -I /our_target_include_path
```

Where preparation time is short or problems remain after trying solution 1.

- 1** Identify the function name causing the problem (sprintf, say);
- 2** If no prototype for this function can be found, provide a .c file containing the prototype for this function;

- 3 Restart the verification either with the PolySpace Launcher or from the command line.

Other `__polyspace_no_<function_name>` options can be found in `__polyspace__stdstubs.c` files, such as

```
__polyspace_no_vprintf
__polyspace_no_vsprintf
__polyspace_no_fprintf
__polyspace_no_fscanf
__polyspace_no_printf
__polyspace_no_scanf
__polyspace_no_sprintf
__polyspace_no_sscanf
__polyspace_no_fgetc
__polyspace_no_fgets
__polyspace_no_fputc
__polyspace_no_fputs
__polyspace_no_getc
```

Note If you are considering defining multiple project generic `-D` options, then using the `-include` option may provide a more efficient solution to this type of error. Refer to “How to Gather Compilation Options Efficiently” on page 4-19.

Where all other attempts have failed. To ignore `__polyspace_stdstubs.c` but still see which standard library functions are in use:

- 1 Deactivate all standard stubs using the option `-D POLYSPACE_NO_STANDARD_STUBS`. For example:

```
polyspace-c -D POLYSPACE_NO_STANDARD_STUBS
```

or

Deactivate all stubbed extensions to ANSI C standard by using `-D POLYSPACE_STRICT_ANSI_STANDARD_STUBS`. For example:

```
polyspace-c -D POLYSPACE_STRICT_ANSI_STANDARD_STUBS
```

This will present a list of functions PolySpace tries to stub, as well as the standard functions in use (**most probably** without any prototype). You will have the following type of message:

```
* Function strcpy may write to its arguments and may
return parts of them. Does not model pointer effects.
Returns an initialized value.
```

```
Fatal error: function 'strcpy' has unknown prototype
```

- 2 Add a “proper” include file in the C file that uses your standard library function. If PolySpace is restarted with the same options, the default behavior for these stubs for this particular function will result.

Consider the example `size_t strcpy(char *s, const char *i)`

- Stubbed to write anything in *s
- Stubbed to return any possible size_t.

Note If the problem remains after trying all 3 solutions, contact PolySpace support.

Errors when Creating Automatic Stubs

There are three different types of error messages which may be generated during the automatic creation of stubs.

Error 1

PolySpace output

```
Fatal error: function 'f' refers to a function pointer either
much too complex or in a too-complex data-structure, or with
unknown parameters.
```

```
It cannot be stubbed automatically.
```

Consider a prototype `f` which contains a function pointer as a parameter.

If the function pointer prototype only contains scalars and/or floats then “f” will be stubbed automatically.

For example, the following function will be stubbed automatically:

```
int f(
void (*ptr_ok)(int, char, float),
other_type1 other_param1);
```

If this function pointer prototype also contains pointers, the use will get the error message and will have to stub the “f” function manually

For example, the following function will need to be stubbed manually by default (unless the -permissive-stubber option is used):

```
int f(
void (*ptr_ok)(int *, char, float),
other_type1 other_param1);
```

Error 2

PolySpace output

```
Fatal error: function 'f' has unknown prototype
```

```
-----
```

```
Error message explanation:
```

- "function has wrong prototype" means that either the function has no prototype or its prototype is not ANSI compliant.
- "task is undefined" means that a function has been declared to be a task but has no known body

For any function to be automatically stubbed, PolySpace needs the prototype.

Error 3

PolySpace output

```
*** Verifier found an error in parameter -entry-points: task "w"
must be a userdef function
```

```
-----
```

```
---
```

```
--- Found some errors in launching command. ---
```

```
--- Please consult rte-kernel -h to correct them    ---  
--- and launch the verification again.             ---  
---                                         ---  
-----  
---
```

No function or procedure declared to be an `-entry-point` can be an automatically stubbed function.

Intermediate Language Errors

The verification log can sometimes indicate that a red error has been detected in the previous phase, and that the verification has therefore stopped. If no graphical result is provided, the errors and their locations are listed at the end of the log file. To find them, you can scroll through the verification log file starting at the end and working backwards.

Note This example only explains *where* to find the error list. Their *meaning* and the *error messages* themselves are detailed in the next section.

The log file may be similar to this one:

```

***** C to intermediate language translation 13.29 (P_SENUP) took
0.000773real, 0.000773u + 0.0s

-----
1 User Program Errors:
* failure of correctness condition [non-initialized variable]
"&" file intermediate.c line 5 column 0
Please correct the program and restart the verifier.
-----
***** C to intermediate language translation 13.30 (IL Partition)
0 empty package(s) removed
***** C to intermediate language translation 13.30 (IL Partition)
took 0.002252real, 0.002252u + 0.0s
**** C to intermediate language translation 13 (P_IL) took
1.069168real, 1.069168u + 0.0s
0 empty package(s) removed
**** C to intermediate language translation 14 (P_IPF)
96% init procedures removed
**** C to intermediate language translation 14 (P_IPF) took
0.002401real, 0.002401u + 0.0s
* terminating ../il-sources/a0.ads
* terminating ../il-sources/a0.adb
**** C to intermediate language translation 15 (P_TW)
**** C to intermediate language translation 15 (P_TW) took
0.003055real, 0.003055u + 0.0s

```

```
* assigns: 100% reduction
* asserts: 100% reduction
* total : 54% reduction
User time for command `iabc-c2if -input-file': 17 seconds on host
paris12
```

```
*****
***
*** C to intermediate language translation done
***
```

```
*****
Ending at: Oct 31, 2002 14:29:26
Certain (red) errors detected during previous phase.
You must correct them before continuing.
```

Reducing Verification Time

In this section...

“How Far has the Verification Progressed? How Can I Predict the Duration?” on page 7-27

“An Ideal Application Size” on page 7-29

“Why Should there be an Optimum Size?” on page 7-30

“Switch the Antivirus Off” on page 7-31

“Tuning PolySpace Parameters” on page 7-31

“Selecting a Subset of Code” on page 7-32

“A Decision Algorithm to Speed-Up a Verification: Hints and Troubleshooting” on page 7-37

“What are the Benefits of these Methods?” on page 7-42

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

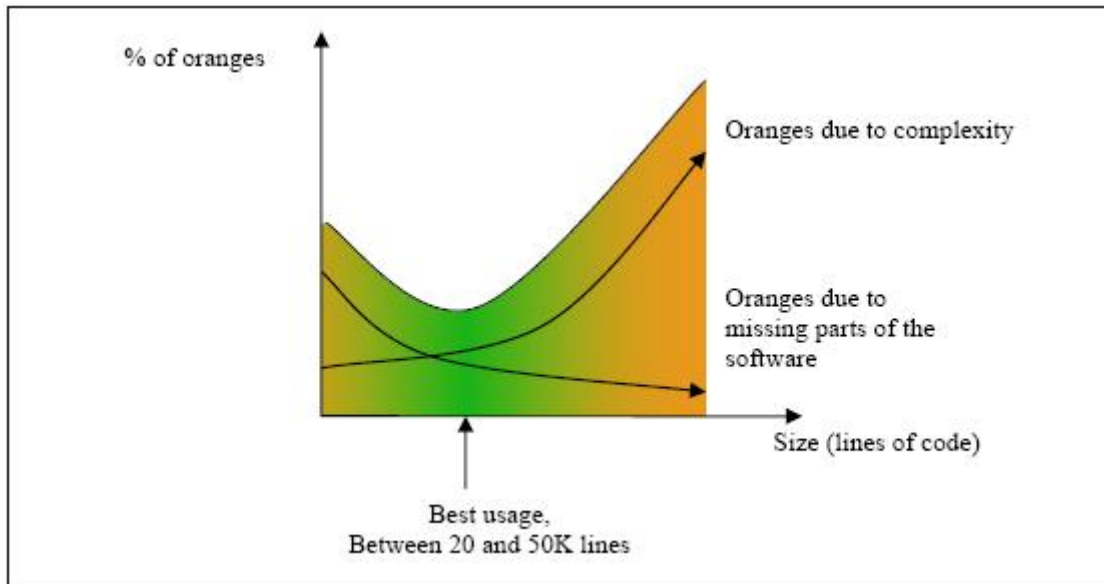
The remaining verification duration can be deduced by extrapolating from this data by considering the number of files and passes still to be completed.

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 5-2).

```
//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

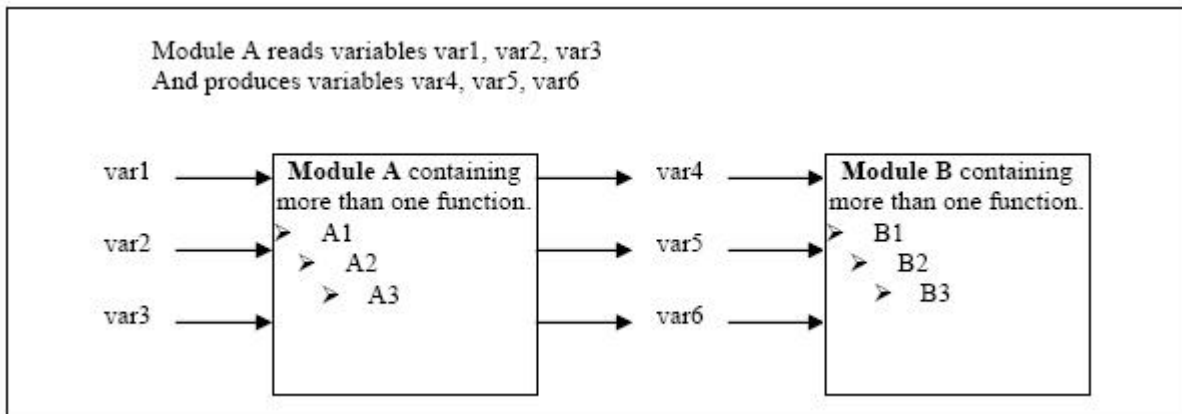
- **Error management modules.** These modules often contain a big array of structures that are accessed through an API, but return only a Boolean value. By removing the API code and retaining the prototype,

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"> • If var2 is equal to 0, than $\text{var4} > \text{var5} > 5$. • If var3 is greater than 4, than $\text{var4} < \text{var5} < 12$ • ...

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.

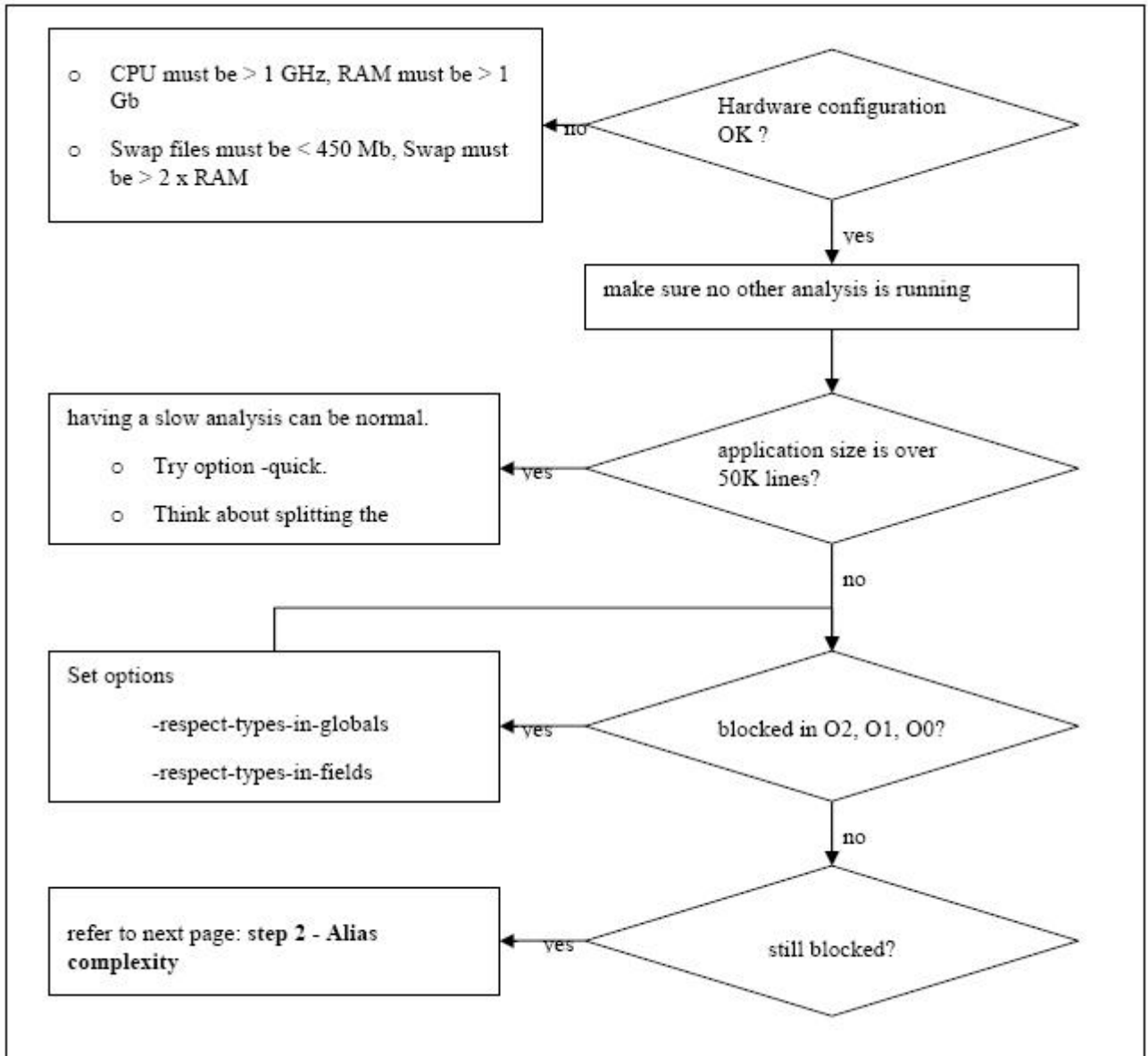
A Decision Algorithm to Speed-Up a Verification: Hints and Troubleshooting

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.

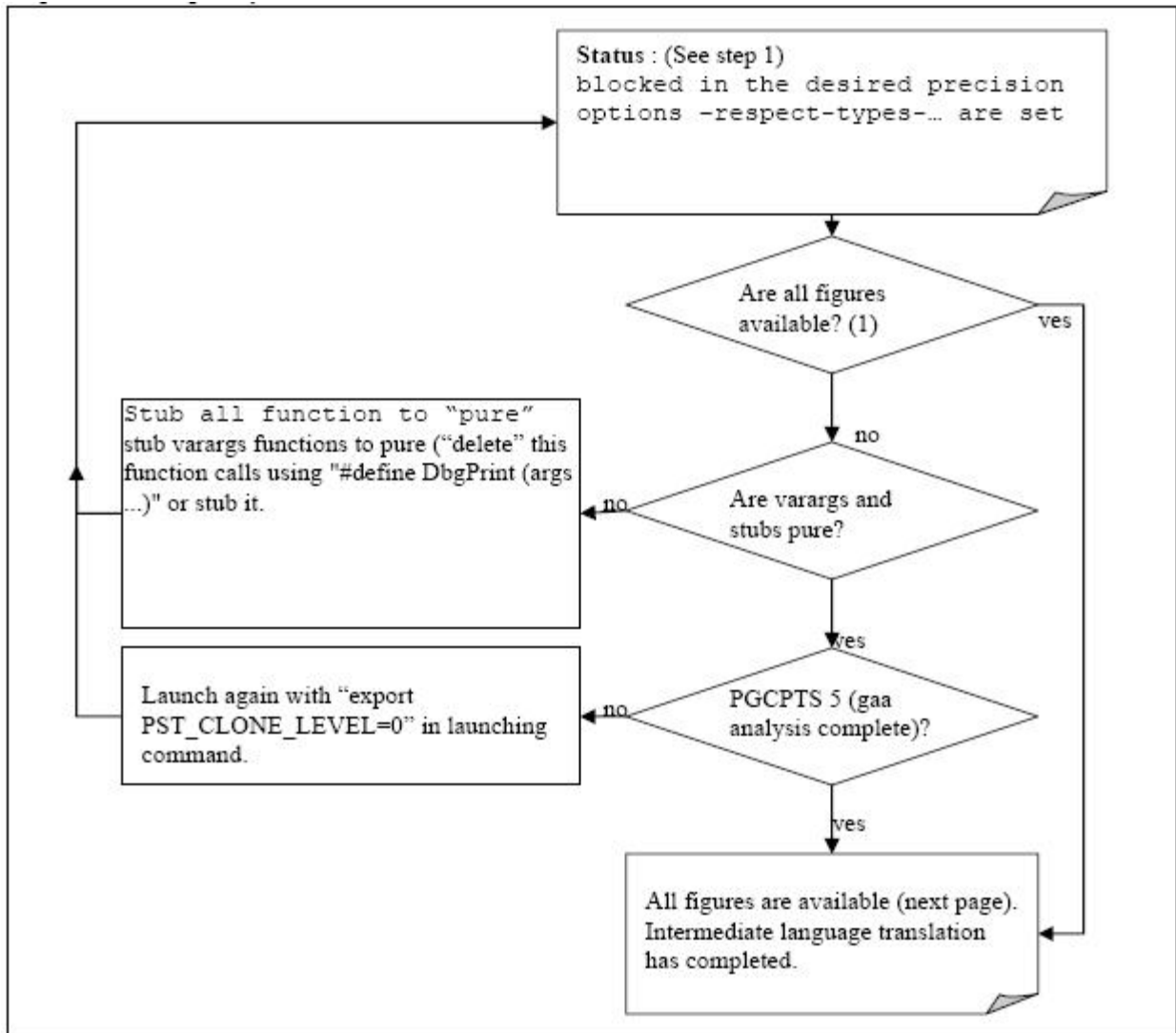
The size of a code sample which can be effectively analyzed can be increased by tuning the tool be optimized for that sample. Beyond that point, subdividing the code or choosing a lower precision level will bring better results (-O1, -O0).

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 :

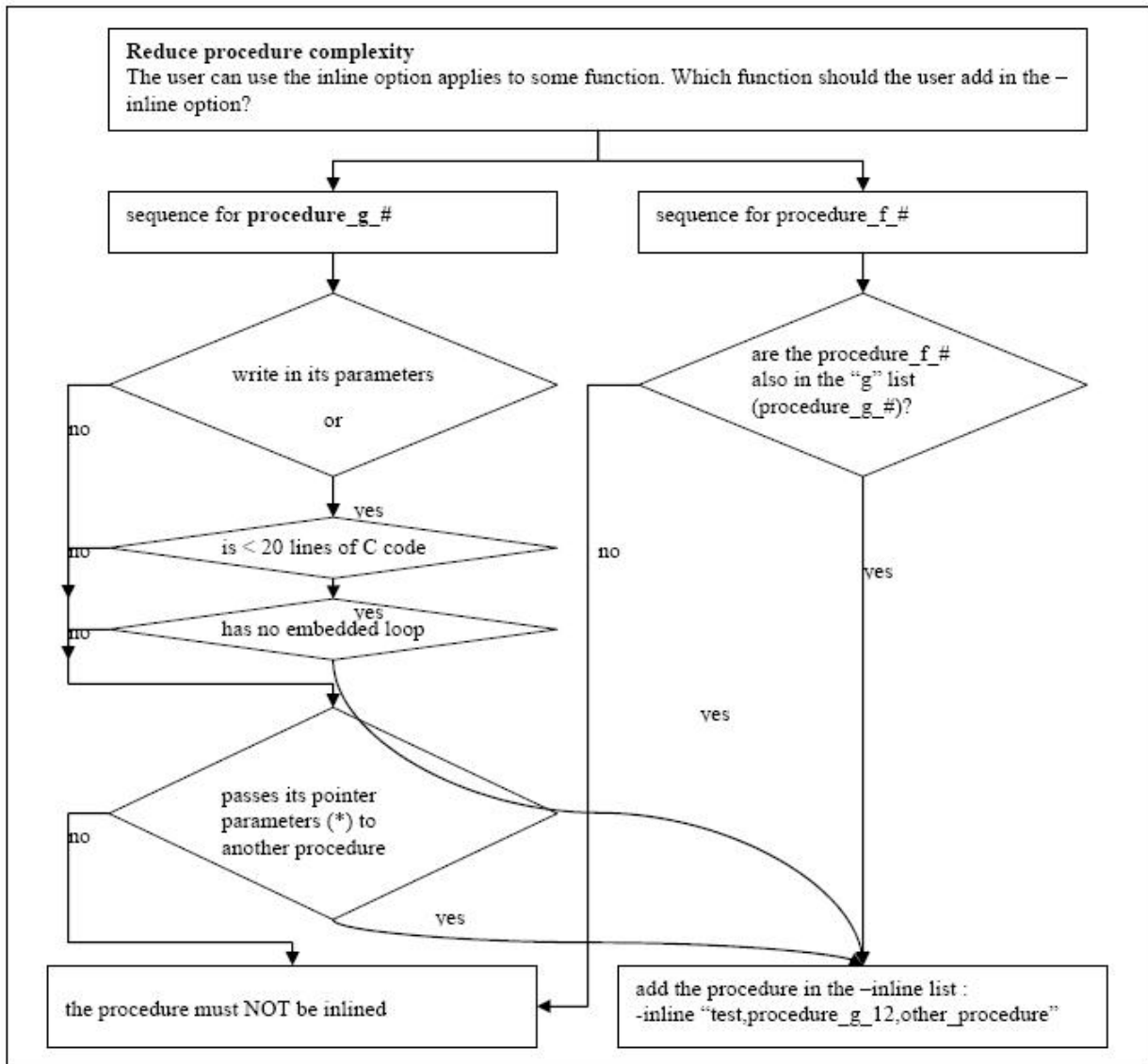
```
procedure_g_1      procedure_g_2
procedure_g_3
```

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 8-2
- “Opening Verification Results” on page 8-8
- “Reviewing Results in Assistant Mode” on page 8-17
- “Reviewing Results in Expert Mode” on page 8-25
- “Generating Reports of Verification Results” on page 8-37
- “Using PolySpace Results” on page 8-41

Before You Review PolySpace Results

In this section...

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

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

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

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

Overview: Understanding PolySpace Results

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

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

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

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

Why Gray Follows Red and Green Follows Orange

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

In the example below, consider why:

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

```

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 9-16.

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

You can see the category of each check by clicking on it in the Viewer. 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:

- " $x > 100$ " does **NOT** mean that PolySpace doesn't know anything about x .
- " $x > 100$ " **DOES** mean 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 ($y(x - 1)$).

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 8-8
“Opening Verification Results” on page 8-11
“Exploring the Viewer Window” on page 8-11
“Selecting Viewer Mode” on page 8-15
“Setting Character Encoding Preferences” on page 8-15

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.

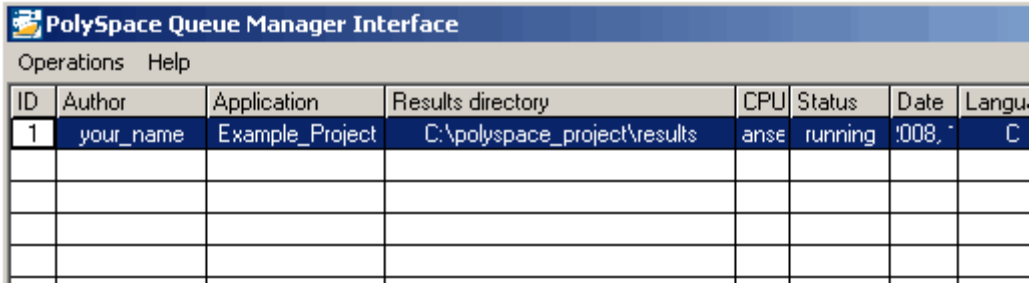
Note If you download results before the verification completes, you get partial results and the verification continues.

To download verification results to your client system:

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



The **PolySpace Queue Manager Interface** opens.



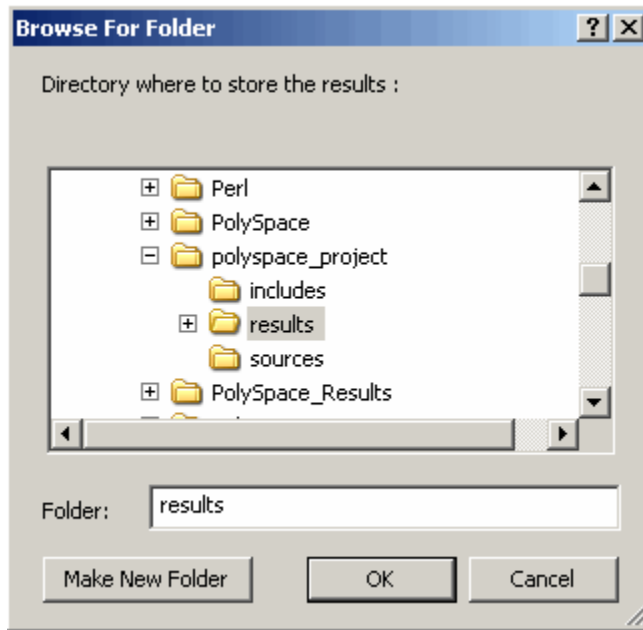
The screenshot shows the PolySpace Queue Manager Interface. At the top, there is a title bar with the PolySpace logo and the text "PolySpace Queue Manager Interface". Below the title bar is a menu bar with "Operations" and "Help". The main area contains a table with the following columns: ID, Author, 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: Example_Project, Results directory: C:\polyspace_project\results, CPU: anse, Status: running, Date: '008, and Language: C. There are four empty rows below the first row.

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

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

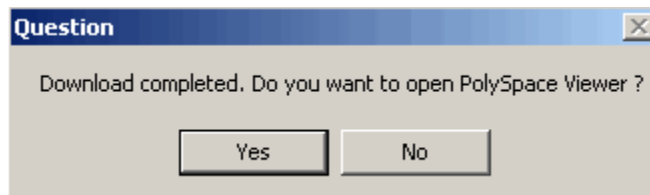
Note To remove the job from the queue after downloading your results, select **Download Results And Remove From Queue** from the context menu.

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



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

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



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

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

Opening Verification Results

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

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

To open the verification results:

- 1 Double-click the PolySpace Viewer icon:



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

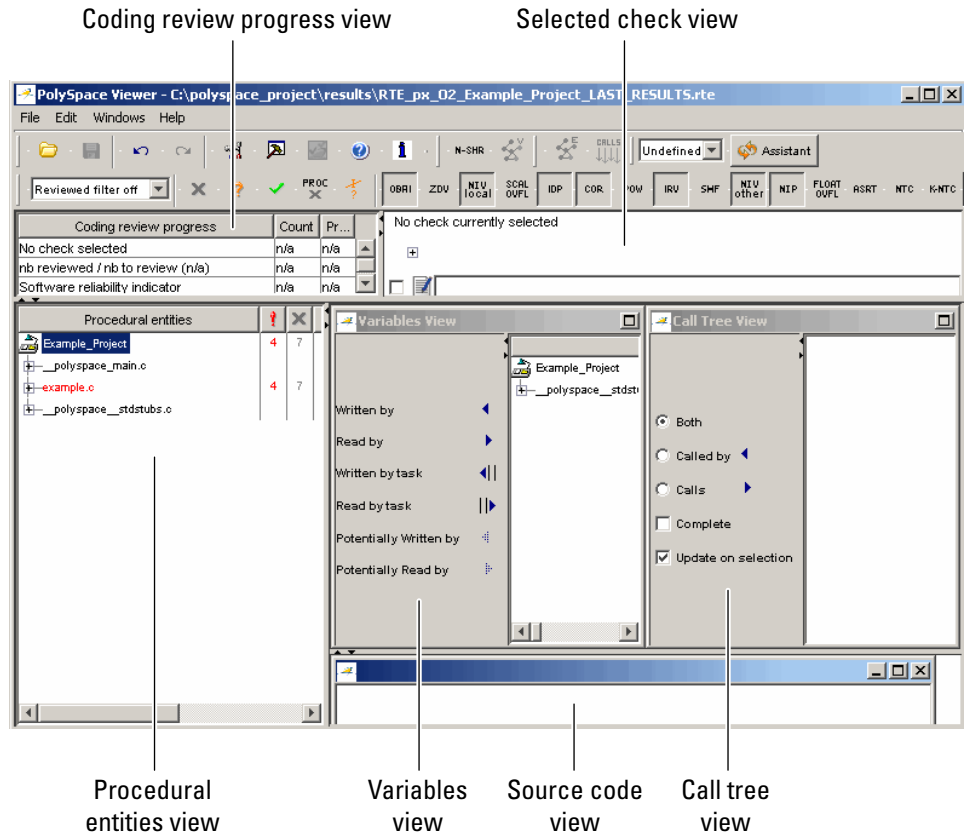
The results appear in the Viewer window.

Exploring the Viewer Window

- “Overview” on page 8-11
- “Procedural Entities View” on page 8-13

Overview

The PolySpace Viewer looks like:



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



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

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






You can resize or hide any of these sections.


Procedural Entities View

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

Procedural entities	?	X	?	✓	Line	...	%
Example_Project	4	7	7	21	1		82
+ __polyspace_main.c					1		0 __pol
+ example.c	4	7	7	21	1		82 exam
+ __polyspace_stdstubs.c					1		0 __pol

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 8-30.

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

Selecting Viewer Mode

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

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

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



Setting Character Encoding Preferences

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

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

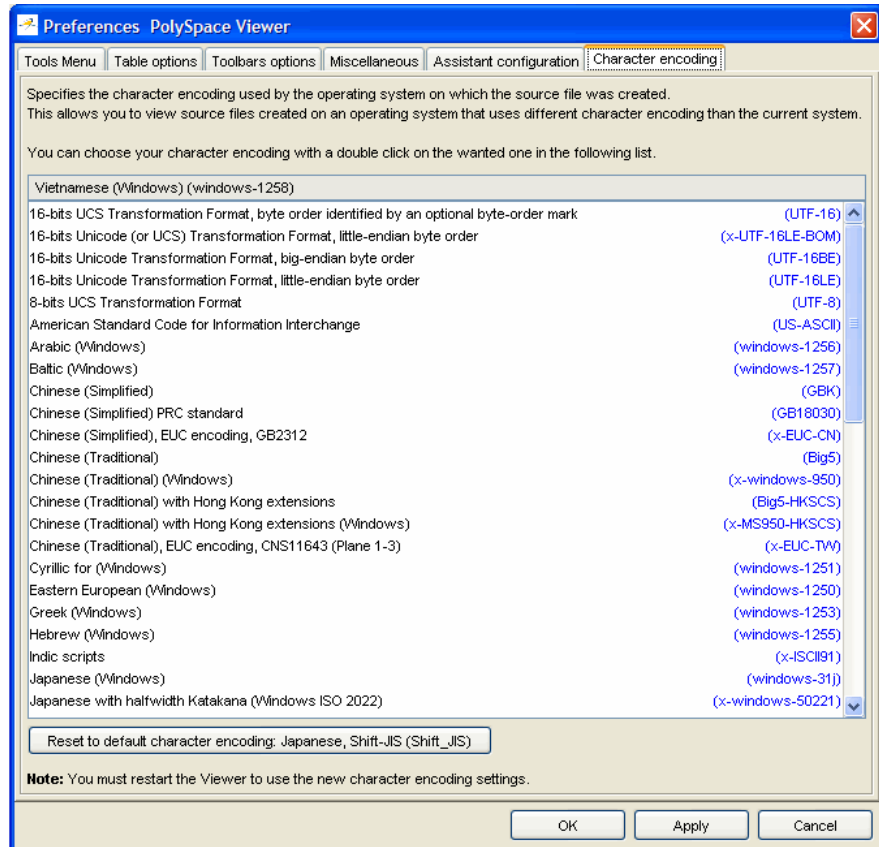
To set the character encoding for a source file:

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

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

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

The Character encoding tab appears.



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

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

- 5 Close and restart the Viewer.

Reviewing Results in Assistant Mode

In this section...

- “What Is Assistant Mode?” on page 8-17
- “Switching to Assistant Mode” on page 8-17
- “Selecting the Methodology and Criterion Level” on page 8-18
- “Exploring Methodology for C” on page 8-19
- “Defining a Custom Methodology” on page 8-21
- “Reviewing Checks” on page 8-22

What Is Assistant Mode?


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

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

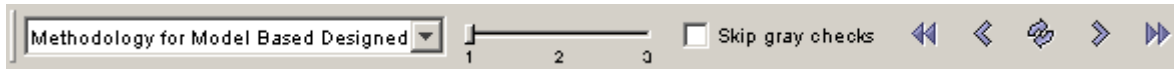
For more information about methodologies and criterion levels, see “Selecting the Methodology and Criterion Level” on page 8-18.

Switching to Assistant Mode

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

- Click the Viewer mode button .

The Viewer window toolbar displays controls specific to assistant mode.



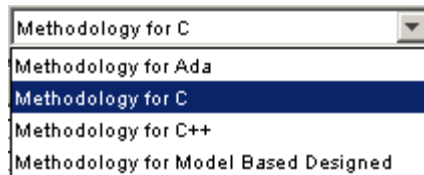
The controls for assistant mode include:

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

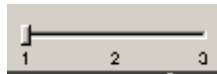
Selecting the Methodology and Criterion Level

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

- 1 Select **Methodology for 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.

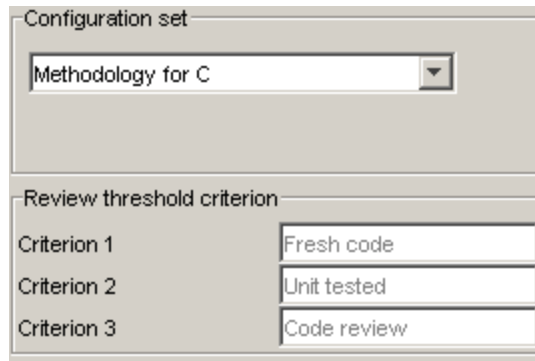
aneous Assistant configuration

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
C only			
IRV	5	20	ALL
C++ only			

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

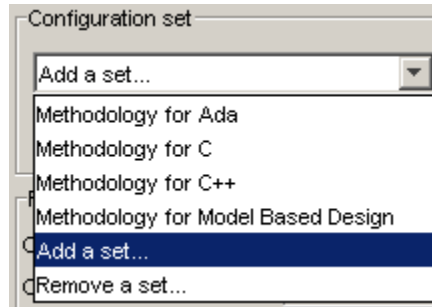
A methodology defines the number of orange checks that you review in assistant mode. You cannot change the predefined methodologies, such as Methodology for C, but you can define your own methodology.

To define a custom methodology:

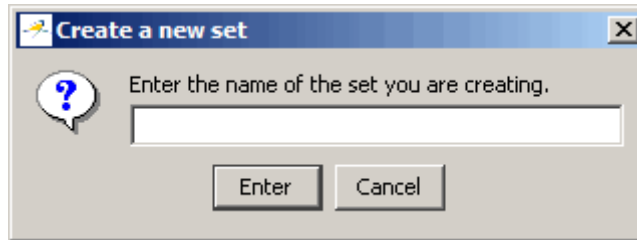
- 1 Select **Edit > Preferences**.

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)

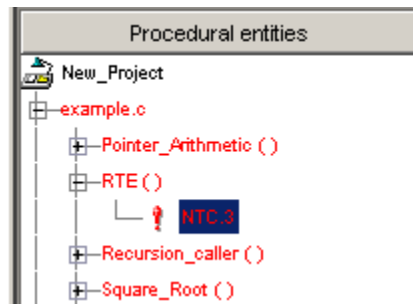
Note You can skip gray checks by selecting the **Skip gray checks** check box in the toolbar.

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.



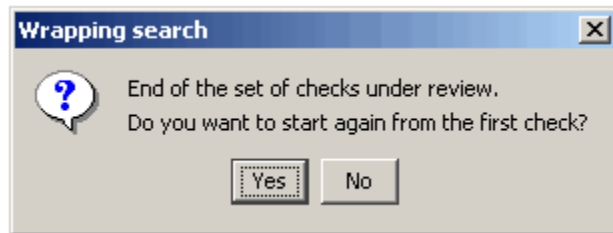
- The source code view (lower right) displays the source code for this check.
- The current check view (upper right) displays information about this check.

Note You can display the calling sequence and track review progress. See “Reviewing Results in Expert Mode” on page 8-25.

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 8-25
- “Switching to Expert Mode” on page 8-25
- “Selecting a Check to Review” on page 8-25
- “Displaying the Calling Sequence” on page 8-27
- “Tracking Review Progress” on page 8-28
- “Making the Reviewed Column Visible” on page 8-30
- “Filtering Checks” on page 8-33
- “Types of Filters” on page 8-33
- “Creating a Custom Filter” on page 8-35

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:



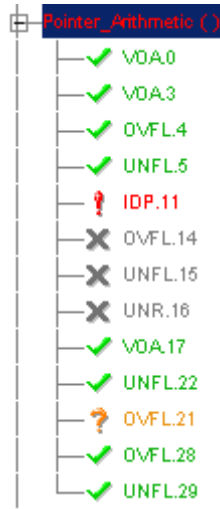
The Viewer window toolbar displays buttons and menus specific to expert mode.

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.


```

example.c
92     int i, *p = array;
93
94     for(i = 0; i < 100; i++)
95     {
96         *p = 0;
97         p++;
98     }
99
100    if(get_bus_status() > 0)
101    {
102        if(get_oil_pressure() > 0)
103        {
104            *p = 5; /* Out of bounds */
105        }
106        else
107        {
108            i++;
109        }
110    }

```

4 Click the colored check in the code.

An message box appears describing the error.

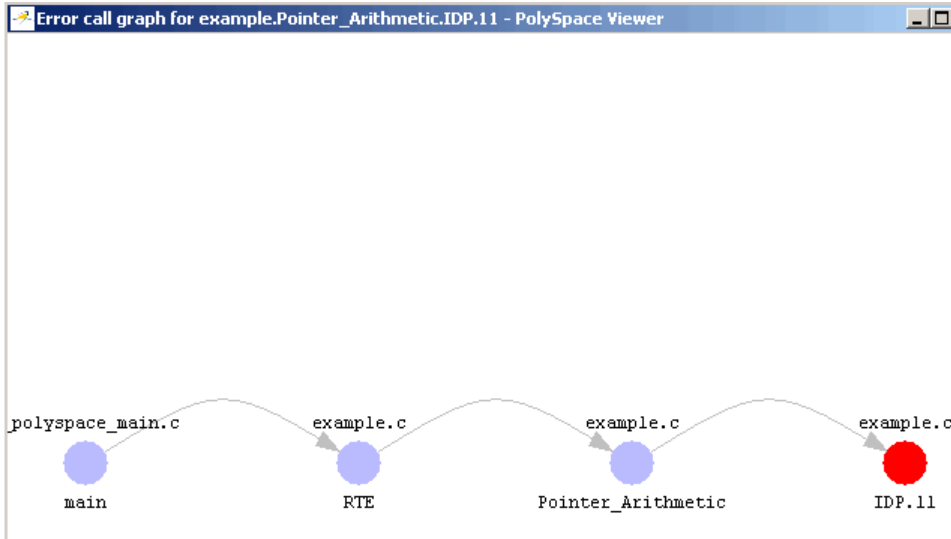
Displaying the Calling Sequence

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)	0/4	0
Software reliability indicator	93/115	80

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.

```
example.c / Pointer_Arithmetic / line 104 / column 10
 *p = 5; /* Out of bounds */
 
Error : pointer is outside its bounds
```

- 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.

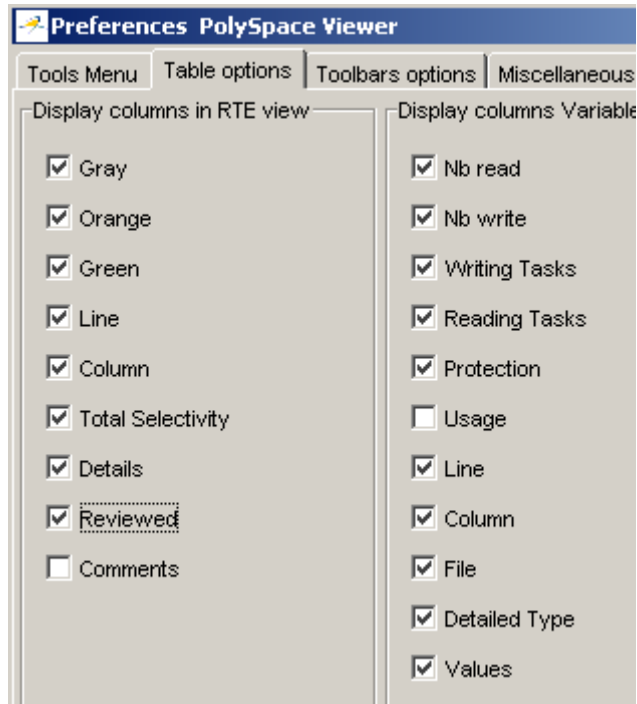
Coding review progress	Count	Progress
nb IDP reviewed / nb IDP to review (Red)	1/1	100
nb reviewed / nb to review (Red)	1/4	25
Software reliability indicator	93/115	80

Making the Reviewed Column Visible

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

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

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



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

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

Procedural entities	!	X	?	✓	Line	...	%	Details	Reviewed
Example_Project	4	7	7	21			82		<input type="checkbox"/>
+_polyspace_main.c					1		0	_poly...	<input type="checkbox"/>
example.c	4	7	7	21	1		82	exampl...	<input type="checkbox"/>
Close_To_Zero ()			3	2	37	12	40	exampl...	<input type="checkbox"/>
Non_Infinite_Loop ()				4	66	11	100	exampl...	<input type="checkbox"/>
Pointer_Arithmetic ()	1	3	1	5	89	12	90	exampl...	<input type="checkbox"/>
✓ VOA.0					94		6		<input type="checkbox"/>
✓ VOA.3					94		22		<input type="checkbox"/>
✓ OVFL.4				1	94		23	[+] ov...	<input type="checkbox"/>
✓ UNFL.5				1	94		23	[+] und...	<input type="checkbox"/>
! IOP.11	1				104		10	pointer ...	<input checked="" type="checkbox"/>
X OVFL.14		1			108		11	[+] ov...	<input type="checkbox"/>
X UNFL.15		1			108		11	[+] und...	<input type="checkbox"/>
X UNR.16		1			108		11		<input type="checkbox"/>
✓ VOA.17					112		2		<input type="checkbox"/>
✓ UNFL.22				1	114		16	[?] und...	<input type="checkbox"/>
? OVFL.21			1		114		16	[?] ove...	<input type="checkbox"/>

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

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

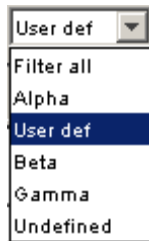
- Selects the check box for that check in the current check view (upper-right part of the window).
- Updates the counts in the coding review progress view (upper-left part of the window).

Filtering Checks

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

The default filter is User def.

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



Types of Filters

There are three types of filters:

- “Individual Filters” on page 8-33
- “Composite Filters” on page 8-34
- “Custom Filters” on page 8-34

Individual Filters

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

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

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

Composite Filters

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

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

Custom Filters

The custom filter is a composite filter that you define. It appears on the composite filter menu as `User def` and is the default composite filter. By default, the custom filter hides the 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 8-35.

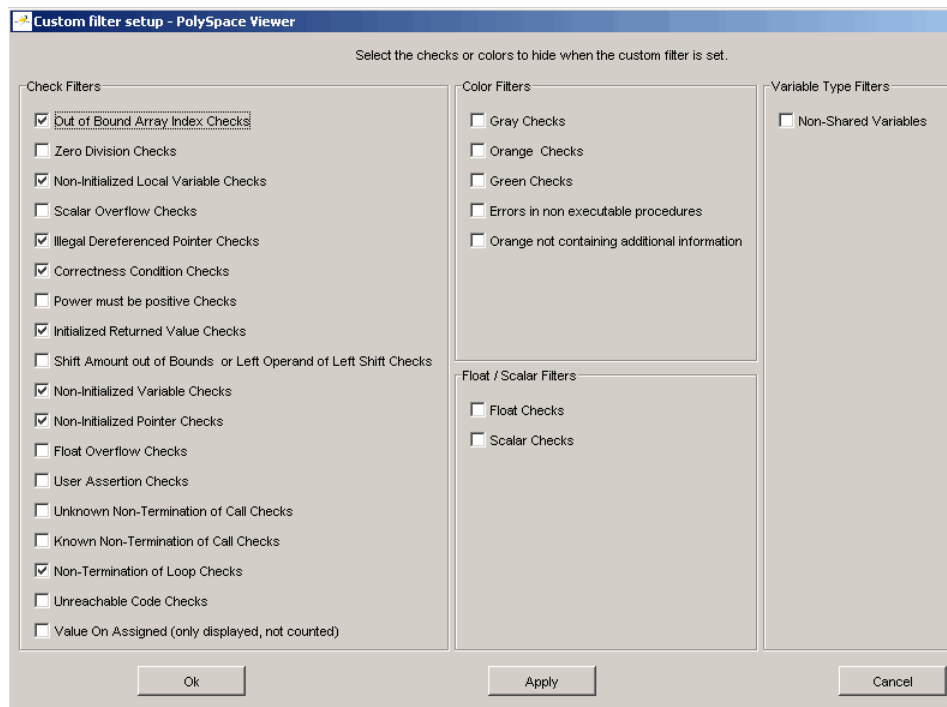
Creating a Custom Filter

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

To modify the custom filter:

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

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



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

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

4 Select the filters for the checks that you do not want to display.

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

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

Generating Reports of Verification Results

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

To generate an Excel report of your verification results:

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

The directory should have the following files:

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

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

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

A security warning dialog appears.

- 3 Click **Enable Macros**.

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

Apply filters? No filters Beta filters

Generate checks by file? yes no

Help Use this button to create the complete synthesis in one file. Select the RTE export view and a file in which to save results. If the other views are in the same directory as the RTE view then they will automatically be incorporated into the same file. Help

Generate PolySpace Results Synthesis

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

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

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

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

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

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

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

Example_Project-Synthesis.xls	
A	
1	Call Graph of ll tree
2	
3	all tree
4	__polyspace_main.main
5	- > example.RTE
6	- > example.Close_To_Zero
7	> pst_stubs_0.random_float
8	> pst_stubs_0.random_float
9	> pst_stubs_0.random_int
10	> example.Non_Infinite_Loop
11	- > example.Pointer_Arithmetic
12	> pst_stubs_0.get_bus_status
13	> example.get_oil_pressure
14	> pst_stubs_0.get_bus_status
15	- > example.Recursion_caller
16	> pst_stubs_0.random_int
17	- > example.Recursion
18	** RecursiveCall to example.Recursion:
19	> pst_stubs_0.random_int
20	- > example.Recursion
21	Already displayed above
22	> pst_stubs_0.random_int
23	- > example.Square_Root
24	> pst_stubs_0.random_float
25	- > example.Square_Root_conv
26	> ?extern.cos
27	> ?extern.sqrt
28	- > example.Unreachable_Code
29	> pst_stubs_0.random_int
30	> pst_stubs_0.random_int

Application Call Tree / Shared Globals / Global Data Dictionary / Checks by file

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

Example_Project-Synthesis.xls						
	A	B	C	D	E	F
1	RTE Statistics					
2	Check category	Check detail	R	O	Gy	Gr
3	OBAI	Out of Bounds Array Index	0	0	0	0
4	NIVL	Uninitialized Local Variable	0	1	2	32
5	IDP	Illegal Dereference of Pointer	1	1	0	7
6	NIP	Uninitialized Pointer	0	0	0	12
7	NIV	Uninitialized Variable	0	0	0	6
8	IRV	Initialized Value Returned	0	0	0	13
9	COR	Other Correctness Conditions	0	0	0	2
10	ASRT	User Assertion Failure	0	1	0	0
11	POW	Power Must Be Positive	0	0	0	0
12	ZDV	Division by Zero	0	1	0	4
13	SHF	Shift Amount Within Bounds	0	0	0	0
14	OVFL	Overflow	0	2	3	5
15	UNFL	Underflow	0	0	3	7
16	UOVFL	Underflow or Overflow	0	3	0	5
17	EXCP	Arithmetic Exceptions	0	0	0	0
18	NTC	Non Termination of Call	3	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
21	UNR	Unreachable Code	0	0	1	0
22	UNP	Uncalled Procedure	0	0	0	0
23	IPT	Inspection Point	0	0	0	0
24	OTH	other checks	0	0	0	0
25	EXC	Exception handling	0	0	0	0

Using PolySpace Results

In this section...

“Review Runtime Errors: Fix Red Errors” on page 8-41

“Review Dead Code Checks: Why Gray Code is Interesting” on page 8-42

“Selective Orange Review: Finding the Maximum Number of Bugs in One Hour” on page 8-44

“Exhaustive Orange Review at Unit Phase” on page 8-46

“Exhaustive Orange Review at Integration Phase” on page 8-47

“Integration Bug Tracking” on page 8-49

“How to Find Bugs in Unprotected Shared Data” on page 8-50

“Dataflow Verification” on page 8-51

“Data and Coding Rules” on page 8-51

“Potential Side Effect of a Red Error” on page 8-51

“PolySpace Remembers the Relationships Between Variables” on page 8-53

“Why There Might be 2 Distinct Colors in a while/for Statement.” on page 8-54

Review Runtime Errors: Fix Red Errors

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

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

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

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

PolySpace verification identifies two kinds of red checks

- Red errors which are compiler-dependant in a specific way. On some occasions a PolySpace option may be used to allow particular compiler specific behavior, and on others the code must be corrected in order to comply. An example of a PolySpace option to permit compiler specific behavior would be the option to force "IN/OUT" ADA function parameters to be initialized. Examples in C include options to deal with constant overflows, shift operation on negative values, etc.
- All other red errors must be fixed. They are bugs.

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

Review Dead Code Checks: Why Gray Code is Interesting

- "Functional Bugs Can Be Found in Gray Code" on page 8-42
- "Structural Coverage" on page 8-44

Functional Bugs Can Be Found in Gray Code

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

- Defensive code which is never reached
- Dead code due to a particular configuration
- Libraries which are not used to their full extent in a particular context
- Dead code resulting from bugs in the source code.

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

- A lack of parenthesis and operand priorities in the testing clause can change the meaning significantly.
- Consider a line of code such as

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

Now consider how misplaced parentheses might influence how that line behaves

```
IF NOT (a AND b OR c AND d)
IF (NOT (a) AND b) OR (c AND d)
IF NOT (a AND (b OR c) AND d)
```
- The test of variable inside a branch where the conditions are never met;
- An unreachable “else” clause where the wrong variable is tested in the “if” statement
- A variable that is supposed to be local to the file but instead is local to the function
- Wrong variable prototyping leading to a comparison which is always false (say)

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

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

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

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

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

Structural Coverage

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

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

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

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, 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 9-3 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.

How to Find Bugs in Unprotected Shared Data

Based on the list of entry points in a multi-task application, PolySpace verification identifies a list of shared data and provides several pieces of information about each entry:

- The data type;
- A list of reading and writing accesses to the data through functions and entry points;
- The type of any implemented protection against concurrent access.

A shared data item is a global data item that is read from or written to by two or more tasks. It is unprotected from concurrent accesses when one task can access it whilst another task is in the process of doing so. All the possible situations are considered below.

- If there is a possible scenario which would lead to such conflict for a particular variable, then a bug exists and protection is required.
- If there are no such scenarios, then one of the following explanations may apply:
 - The compilation environment guarantees an atomic read/write access on variable of type less than 1, 2 bytes, and therefore all conflicts concerning a particular variable type still guarantee the integrity of the variables content. But beware when porting the code!
 - The variable is protected by a critical section or a mutual temporal exclusion. You may wish to include this information in the PolySpace launching parameters and reverify.

It is also worth checking whether variables are modified which are supposed to be constant. Use the variables dictionary.

Dataflow Verification

Data flow verification is often performed within certification processes — typically in the avionic, aerospace or transport markets.

This activity makes heavy use of two features of PolySpace results, which are available any time after the Control and Data Flow verification phase.

- Call tree computation
- Dictionary containing read/write access to global variables. (This can also be used to build a database listing for each procedure, for its parameters, and for its variables.)

PolySpace software can help you to build these results by extracting information from both the call tree and the dictionary.

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.

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

global_ptr = &variable_it_points_to;

other_function();
}

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

```

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

PolySpace Remembers the Relationships Between Variables

Abstract

Understand that a red error can hide a bug which occurred on previous lines.

```

10 int main(void)
11 {
12   int x,old_x;
13
14   x = read_an_input();
15   old_x = x;
16
17   if (x<0 || x>10)
18     return 0;
19
20   f(x);
21
22   x = 1 / old_x; // division is red
23
24 }
1 double sqrt(double);
2 int read_an_input(void);
3
4 void f(int a)
5 {
6   int tmp;
7   tmp = sqrt(0-a);
8 }
9
```

Explanation 1

- When `old_x` is assigned to `x` (15 `old_x = x;`), PolySpace memorizes two pieces of information:
 - `x` and `old_x` are equivalent to the whole range of an integer: $[-2^{31}; 2^{31}-1]$;
 - and `x` and `old_x` are equal.
- After the "if" clause (17 `if (x<0 || x>10)`), `x` is equivalent to `[0; 10]`. Because `x` and `old_x` are equal, **`old_x` is equivalent to `[0;10]` as well**, because otherwise the return statement would have been executed;

- When X is passed to "F" (20 f(x);), the only possible valid conclusion for sqrt is that x is equal to 0. All other values lead to a runtime exception (7 tmp = sqrt(0-a));
- Back to line 22, because x and old_x are equal, old_x is also equal to 0.

Explanation 2

- Supposing that PolySpace **exits** immediately when encountering a runtime error, let's introduce a print statement that will write to the standard output after the "f" procedure has been called (20 f(x);), to show the current value of x and old_x;
- The only possibility of reaching the print statement is when X is equal to 0. So, if "x" is equal to 0, old_x must have been assigned to 0 too - which makes the division **red**.

Summary

PolySpace builds relationships between variables and propagates the consequence of these relationships backwards and forwards.

Why There Might be 2 Distinct Colors in a while/for Statement.

It is sometimes true that inside the condition of a loop, a check is **green** then **red**.

Consider the following example.

```
1 void main(void)
2 {
3   int tab[2] = { 1, 2 };
4   int index = 0;
5   while (tab[ index]) { index--; }
6   // the colour of "array index within bounds" is
7   // first green
8   // then red
9 }
```

Clicking on the tab variable (line 5) in the Viewer will reveal the following

```
Error : pointer is outside its bounds <= then red
variable is initialized
Pointer is initialized
Pointer is initialized
Pointer is initialized
Pointer is initialized
pointer is within its bounds <= first green
Unreachable check : NIV
```

Now, visualize the C loop as having been transformed into a label and a goto

```
if (not(tab[index]) goto end;
// first location of the check is green
loop_begin:
  index = index-1;
if (tab[index]) goto loop_begin;
// second location of the check is red
end:
```

So, the second color represents the second pass through the loop, and (in the example) should be investigated.

Managing Orange Checks

- “Understanding Orange Checks” on page 9-2
- “Reducing Orange Checks in Your Results” on page 9-6
- “Reviewing Orange Checks” on page 9-20
- “Automatically Testing Orange Code” on page 9-26

Understanding Orange Checks

In this section...

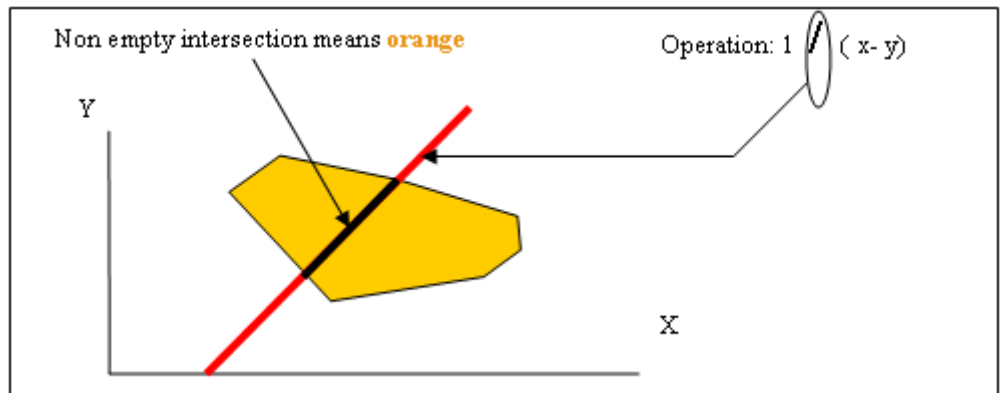
“What is an Orange Check?” on page 9-2

“Sources of Orange Checks” on page 9-3

“Determining Cause of Orange Checks” on page 9-5

What is an Orange Check?

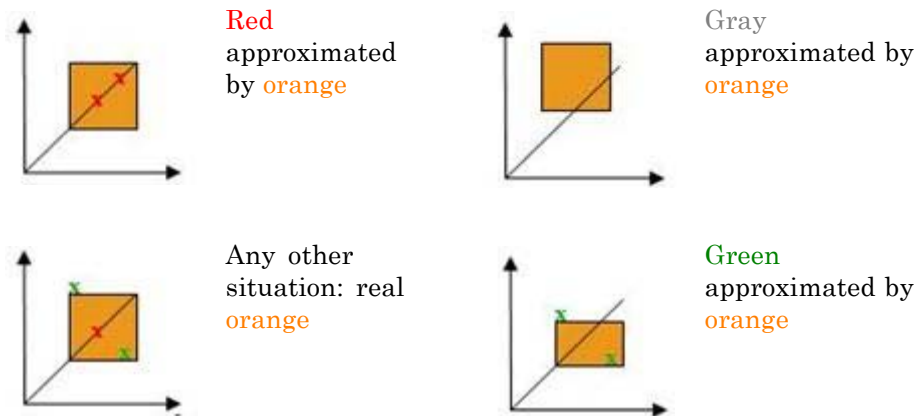
If a check is orange, it means that the approximate data set assumed by the verification to represent a variable intersects with the error zone.



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 9-6

“Generic Objectives: A Balance Between Precision and Verification Time” on page 9-7

“Applying Coding Rules to Reduce Orange Checks” on page 9-8

“Varying the Precision Level” on page 9-13

“Applying Software Safety Level Wisely” on page 9-14

“Adding Precision Constraints at the Periphery Via Stubs” on page 9-15

“Describing Multitasking Behavior Properly” on page 9-17

“Tuning Advanced Parameters” on page 9-18

“Applying Data Ranges” on page 9-19

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 the following subset of MISRA rules, the number of checks per file will typically decrease to 3 orange and 3 gray checks, containing at least one bug.

In addition, some constructions are known to produce a disproportionate number of orange checks. Avoiding these constructions at the design stage will improve your selectivity.

The following coding rules are recommended to reduce oranges:

- “Set of Coding Rules with a Direct Impact on Selectivity” on page 9-8
- “Set of Coding Rules with an Indirect Impact on Selectivity” on page 9-10

Set of Coding Rules with a Direct Impact on Selectivity

Following this set of coding rules will typically improve selectivity.

Rule #	Description
MISRA 8.	declarations of objects should be at function scope unless a wider scope is necessary

Rule #	Description
MISRA 8.11	all declaration at file scope should be static where possible
MISRA 8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
MISRA 10.4	mixed precision arithmetic should use explicit casting to generate the desired results
MISRA 10.5	Bitwise operations shall not be performed on signed integer types
MISRA 11.2	Implicit conversions which may result in a loss of information shall not be used
MISRA 11.5	Type casting from any type to or from pointers shall not be used.
MISRA 12.12	The underlying bit representations of floating-point values shall not be used.
MISRA 13.3	Floating-point expressions shall not be tested for equality or inequality.
MISRA 13.4	Floating point variables shall not be used as <i>loop</i> counters.
MISRA 13.5	Only expressions concerned with loop control should appear within a <i>for</i> statement
MISRA 16.1	Functions with variable numbers of arguments shall not be used.
MISRA 16.2	Functions shall not call themselves, either directly or indirectly.
MISRA 16.7	<i>const</i> qualification should be used on function parameters which are passed by reference, where it is intended that the function will not modify the parameter
MISRA 17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.

Rule #	Description
MISRA 17.3	Relational operators shall not be applied to pointer types except where both operands are of the same type and point to the same array, structure or union.
MISRA 17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
MISRA 18.3	overlapping variable storage shall not be used
MISRA 18.4	Unions shall not be used to access the subparts of larger data types
MISRA 20.4	Dynamic heap memory allocation shall not be used.

Note MISRA rules 16.7, 17.3 and 18.3 are not checked.

Set of Coding Rules with an Indirect Impact on Selectivity

Following good practice in designing and writing “clean” software tends to imply less complexity, and hence yields high selectivity from PolySpace verifications. The following rules are especially significant in this regard.

Rule #	Description
MISRA 5.1	Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.
MISRA 6.3	the basic types of char, int, short, long, float, and double should not be used, but specific-length equivalent should be “ <i>typedef</i> ” for the specific compiler, and these type names used in the code
MISRA 9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.

Rule #	Description
MISRA 9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.
MISRA 10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.
MISRA 11.1	Conversions shall not be performed between a pointer to a function and any type other than the integral type (All the functions pointed to by a pointer to function shall be identical in the number and type of parameters and the return type).
MISRA 12.1	no dependence should be placed on C's operator precedence rules in expressions.
MISRA 12.2	The value of an expression shall be the same under any order of evaluation that the standard permits.
MISRA 12.4	The right hand operand of a logical && or operator shall not contain side effects.
MISRA 12.5	The operands of a logical && or shall be primary-expressions.
MISRA 12.6	Logical operators should not be confused with bitwise operators.
MISRA 12.9	The unary minus operator shall not be applied to an unsigned expression.
MISRA 12.10	The comma operator shall not be used.
MISRA 13.1	Assignment operators shall not be used in expressions which return Boolean values.
MISRA 13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean
MISRA 14.8	The statement forming the body of a <i>if</i> , <i>else if</i> , <i>else</i> , <i>while</i> , <i>do ... while</i> or <i>for</i> statements shall always be enclosed in braces
MISRA 14.10	All <i>if else if</i> constructs should contain a final <i>else</i> clause.

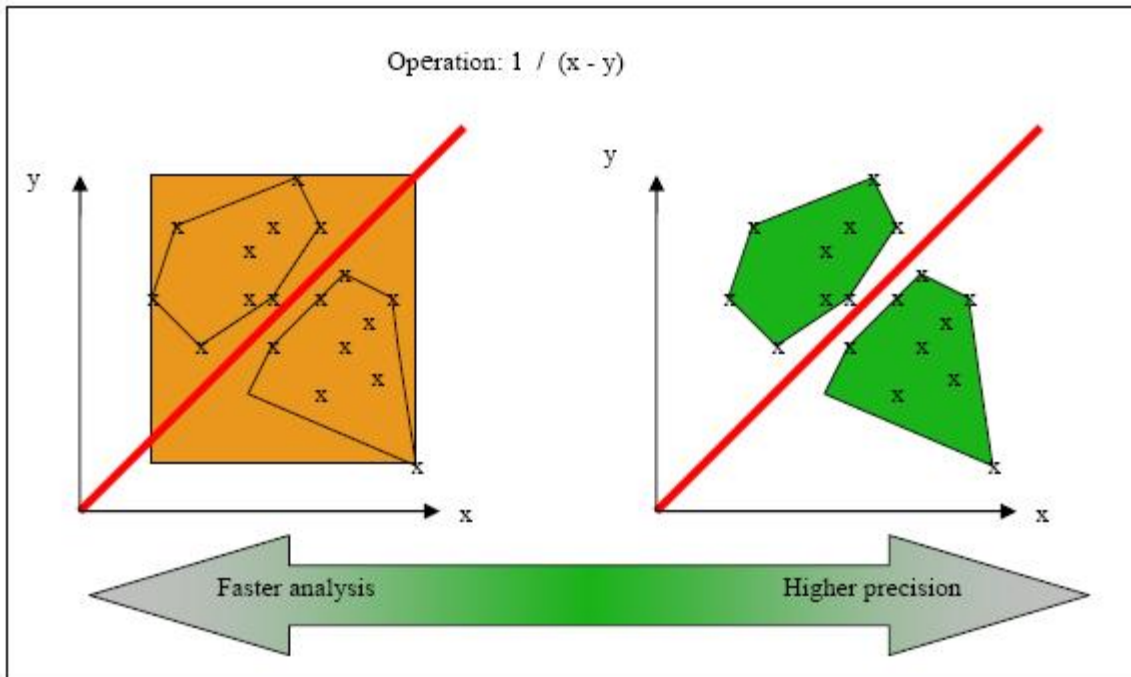
Rule #	Description
MISRA 15.3	All <i>switch</i> statements shall contain a final <i>default</i> clause
MISRA 13.6	Numeric variables being used within a “ <i>for</i> ” loop for iteration counting should not be modified in the body of the loop.
MISRA 16.3	Identifiers shall either be given for all of the parameters in a function prototype declaration, or for none.
MISRA 16.8	For functions with non-void return type: <ul style="list-style-type: none"> i) there shall be one <i>return</i> statement for every exit branch (including the end of the program), ii) each <i>return</i> shall have an expression iii) The <i>return</i> expression shall match the declared return type.
MISRA 16.9	Functions called with no parameters should have empty parentheses
MISRA 19.4	C macros shall only be used for symbolic constants, function-like macros, type qualifiers and storage class specifiers.
MISRA 19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
MISRA 19.10	In the definition of a function-like macro the whole definition, and each instance of a parameter, shall be enclosed in parentheses.
MISRA 19.11	Identifiers in preprocessor directives shall be defined before use.
MISRA 19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.
MISRA 20.3	The validity of values passed to library functions shall be checked.

Note MISRA rule 20.3 is not checked.

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, &y); *t = 1.0/(v - 2.0 * x); } float level2(float v) { float t; t = v; level1(0.0, 1.0, &t); return t; } void main(void) { float r,d; d= level2(1.0); r = 1.0 / (2.0 - d); } </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, &y); *t = 1.0/(v - 2.0 * x); } float level2(float v) { float t; t = v; level1(0.0, 1.0, &t); return t; } void main(void) { float r,d; d= level2(1.0); r = 1.0 / (2.0 - d); } </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, &y); *t = 1.0/(v - 2.0 * x); } float level2(float v) { float t; t = v; level1(0.0, 1.0, &t); return t; } void main(void) { float r,d; d= level2(1.0); r = 1.0 / (2.0 - d); } </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 <assert.h> int stub(void) { volatile int random; int tmp; tmp = random; assert(tmp>=1 && tmp<=10);</pre>	<pre>#include <assert.h> extern int other_func(void); int stub(void) { int tmp; tmp= other_func(); assert(tmp>=1 && tmp<=10);</pre>	<pre>extern int other_func(void); int stub(void) { int tmp; do {tmp= other_func();} while (tmp<1 tmp>10);</pre>

return	return }	return tmp; }
--------	-------------	------------------

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)

Describing Multitasking Behavior Properly

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

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

- T1 sets X to 12
- T2 **divides** by X

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

Refer to “*Preparing Multitasking Code*” on page 5-20 for a complete description of tasking facilities. These include:

- Shared variable protection:
 - Critical sections,
 - Mutual exclusion,
 - Tasks synchronization,
- Tasking:
 - Threads, interruptions,
 - Synchronous/asynchronous events,
 - Real-time OS.

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 4-25.

Reviewing Orange Checks

In this section...
“Selective Orange Review” on page 9-20
“Performing a Selective Orange Review” on page 9-21
“Exhaustive Orange Review” on page 9-22
“Performing an Exhaustive Orange Review” on page 9-23

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 9-3 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.

Automatically Testing Orange Code

In this section...
“Automatic Orange Tester Overview” on page 9-26
“Before Using the Automatic Orange Tester” on page 9-29
“Launching the Automatic Orange Tester” on page 9-31
“Reviewing the Test Results” on page 9-35
“Refining Data Ranges” on page 9-39
“Saving and Reusing Your Configuration” on page 9-43
“Exporting Data Ranges for PolySpace Verification” on page 9-44
“Configuring Compiler Options” on page 9-45
“Technical Limitations” on page 9-46

Automatic Orange Tester Overview

The PolySpace Automatic Orange Tester dynamically stresses unproven code (orange checks) to identify runtime errors, and provides information to help you identify the cause of these errors.

The Automatic Orange Tester complements the results review in the Viewer module of PolySpace Client for C/C++. Manually performing an exhaustive orange review can be time consuming. The Automatic Orange Tester saves time by automatically creating test cases for all input variables in orange code, and then dynamically testing the code to find actual runtime errors.

The Automatic Orange Tester also provides detailed information on why each test-case failed, including the actual values that caused the error. You can use this information to quickly identify the cause of the error, and determine if there is an actual bug in the code.

PolySpace Automatic Orange Tester - _testgen.tgf

File Options Help

Variable Name	Type	Values	Advanced
External Scope			
Function: random_float			
return	float32	min..max	Advanced
Function: random_int			
return	int32	min..max	Advanced
Function: get_bus_status			
return	int32	min..max	Advanced
Function: read_bus_status			
return	int32	min..max	Advanced

Test Campaign Configuration

Number of tests:

Number of iterations for infinite loops:

Per test timeout (in second):

Test Campaign Results

Completed tests: **1000**

No PolySpace run-time errors detected: **71**

Total failed: **929**

Number of checks/Tests with errors: **10/929**

Timeout: **0**

Stopped tests: **0**

Test Completed Time Remaining: 0:0:0 100%

Results	File	Line	Column	Error	# Testcases Failed
Log	example.c	26	2	ASRT (User Assertion Failed)	164
	example.c	114	16	IDP (Illegal Dereference of Pointer)	45
	example.c	49	16	UNFL (Float Underflow)	58
	example.c	104	10	IDP (Illegal Dereference of Pointer)	147
	example.c	193	17	NTC (Non Terminating Call)	73
	example.c	43	12	UNFL (Float Underflow)	137
	example.c	43	12	OVFL (Float Overflow)	114
	example.c	49	16	OVFL (Float Overflow)	71

PolySpace® Automatic Orange Tester

Note The version of the product used to verify the source code must be the same as the one used for analysis in the Automatic Orange Tester. If you open verification results created with an older version of the product in the Automatic Orange Tester, you may get a compilation error.

To avoid this problem, re-launch the code verification with the current version of the product.

How the Automatic Orange Tester Works

PolySpace verification mathematically analyzes the operations in the code to derive its dynamic properties without actually executing it (see “What is Static Verification” on page 1-4). While this verification can identify almost all runtime errors, some operations cannot be proved either true or false because the input values are unknown. These are reported as Orange checks in the Viewer (see “What is an Orange Check?” on page 9-2).

The Automatic Orange Tester takes the PolySpace verification results, and generates *instrumented code* around orange checks so the code can be run. It then generates test cases based on the input variables, and dynamically tests the code for runtime errors.

This dynamic testing approach allows the Automatic Orange Tester to separate actual runtime errors from theoretical problems. You can then focus on these errors to determine if an orange check is identifying an actual bug.

Limitations of Dynamic Testing

Because the Automatic Orange Tester uses a finite number of test cases to analyze the code, there is no guarantee that it will identify a problem in any individual test campaign. It is therefore possible that a particular variable value causes an error, but that value was never tested.

Similarly, since the Automatic Orange Tester builds test cases each time you run it, there is not guarantee that it will produce the same results with each test campaign.

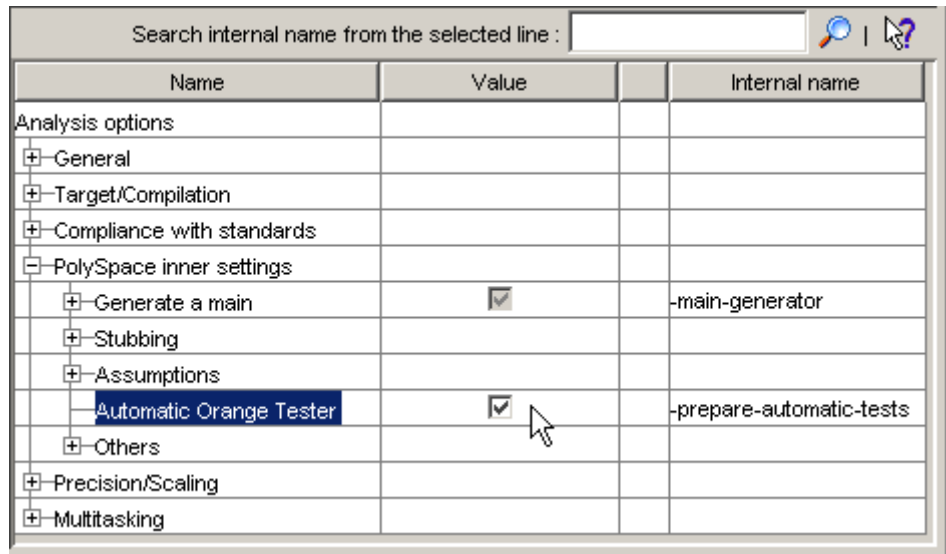
You can specify the number of tests to run in each test campaign. Running more tests increases the chances of finding a runtime error, but also takes more time to complete.

Before Using the Automatic Orange Tester

Before you can use the Automatic Orange Tester, you must run a PolySpace verification with the `-prepare-automatic-tests` option enabled. This option generates the data necessary to perform dynamic tests in the Automatic Orange Tester.

To run the verification:

- 1 Open the PolySpace Launcher for C.
- 2 Load the project Demo_C-without-MISRA-checker.cfg.
- 3 In the Analysis Options window, expand the **PolySpace inner settings** menu.
- 4 Select the **Automatic Orange Tester** check box.



The `-prepare-automatic-tests` option is enabled.

- 5 Deselect **Send to PolySpace Server**.
- 6 Click **Execute**.

The PolySpace verification starts. During the compilation phase, the software generates the data necessary to perform dynamic tests. The PolySpace verification then continues as usual.

When the verification process completes, the software asks if you want to launch PolySpace Viewer.

- 7 Click **OK** to launch the viewer.

Launching the Automatic Orange Tester

Once the PolySpace verification is complete, you can use the Automatic Orange Tester to perform dynamic tests of the unproven (orange) code.

To perform dynamic tests with the Automatic Orange Tester:

- 1 Open your results in the PolySpace Viewer.

PolySpace Viewer - C:\PolySpace\PolySpaceForCandCPP_R2008a\Examples\Demo_C\RTE_px_02_Demo_C_LAST_RESULTS...

File Edit Windows Help

Launch the PolySpace Automatic Oranger Tester.

Coding review progress		Count	Progress
nb UOVFL reviewed / nb UOVFL to review (Orange)		0/3	0
nb reviewed / nb to review (Orange)		0/21	0
Software reliability indicator		249/299	83

example.c / Close_To_Zero / line 43 / column 12

```

if ((xmax - xmin) < 1.0E-37f)

```

Warning : float variable may underflow/overflow on
[conversion from float(32) range -3.41E+38..3.4E+38
to float(32) range -3.41E+38..3.4E+38]

Procedural entities	?	X	O	✓	
Demo_C	8	21	21	249	
example.c	4	9	9	90	
Close_To_Zero ()			3	10	
VDA.0					
IRV.1				1	
VDA.2					
IRV.3				1	
NIVL.4				1	
UOVFL.5				1	
NIVL.8				1	
VDA.7					
VDA.8					
NIVL.9				1	
UOVFL.10				1	
NIVL.11				1	
ZDV.13				1	
UOVFL.12				1	
NIVL.14				1	
UOVFL.15				1	
NIVL.16				1	
-Non_Infinite_Loop ()				12	
-Pointer_Arithmetic ()	1	4	2	23	
-RTE ()	1			3	
-Recursion ()			1	15	
-Recursion_caller ()	1			4	
-Square_Root ()	1	2		4	

Variables View

Written by: Demo_C, initialisations.a, initialisations.c, initialisations.f, initialisations.s, initialisations.t, single_file_anz

Read by: Demo_C, initialisations.a, initialisations.c, initialisations.f, initialisations.s, initialisations.t, single_file_anz

Written by task: Demo_C, initialisations.a, initialisations.c, initialisations.f, initialisations.s, initialisations.t, single_file_anz

Read by task: Demo_C, initialisations.a, initialisations.c, initialisations.f, initialisations.s, initialisations.t, single_file_anz

Potentially Written by: Demo_C, initialisations.a, initialisations.c, initialisations.f, initialisations.s, initialisations.t, single_file_anz

Potentially Read by: Demo_C, initialisations.a, initialisations.c, initialisations.f, initialisations.s, initialisations.t, single_file_anz

Call Tree View

example.Close_To_Zero

- pst_stub
- pst_stub


example.c

```

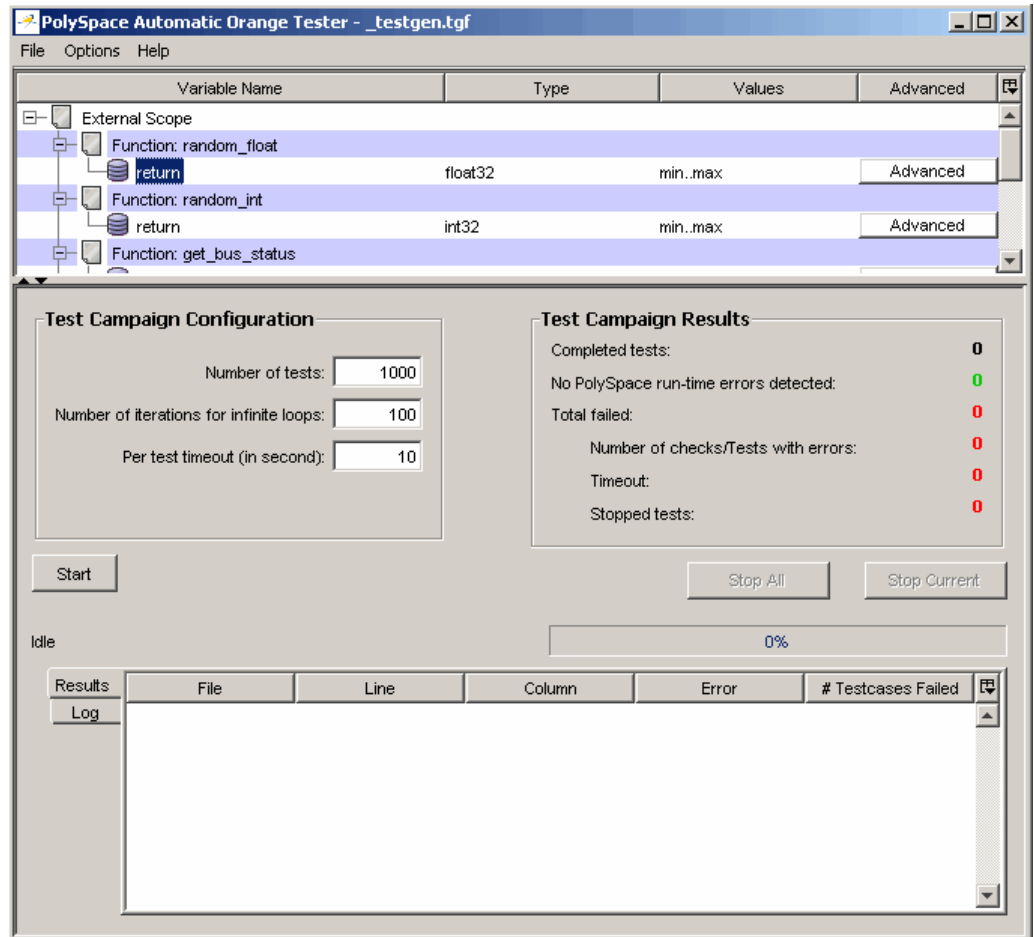
36  */
37  static void Close_To_Zero (void)
38  {
39      float xmin = random_float();
40      float xmax = random_float();
41      float y;
42
43      if ((xmax - xmin) < 1.0E-37f)
44          {
45              y = 1.0f;
46          }
47      else

```

Demo_C Source file: example.c example.c Line: 185 Column: 12

- 2 Click  (Launch the PolySpace Automatic Orange Tester) in the toolbar to open the Automatic Orange Tester.

The Automatic Orange Tester opens.



- 3 In the Test Campaign Configuration window, specify the following parameters:

- **Number of tests** – Specifies the total number of test cases you want to run. Running more tests increases the chances of finding a runtime error, but also takes more time to complete.
- **Number of iterations for infinite loops** – Specifies the maximum number of loop iterations to perform before the Automatic Orange Tester identifies an infinite loop. A larger number of iterations decreases the chances of incorrectly identifying an infinite loop, but also may take more time to complete.
- **Per test timeout** – Specifies the maximum time that an individual test can run (in seconds) before the Automatic Orange Tester moves on to the next test. Increasing the time limit reduces the number of tests that timeout, but can also increase the total verification time.

4 Click **Start** to begin testing.

The Automatic Orange Tester generates test cases and runs the dynamic tests.

The screenshot displays the PolySpace Automatic Orange Tester interface. The top section shows a tree view of the test campaign configuration with the following details:

Variable Name	Type	Values	Advanced
External Scope			
Function: random_float			
return	float32	min..max	Advanced
Function: random_int			
return	int32	min..max	Advanced
Function: get_bus_status			

The middle section contains the **Test Campaign Configuration** and **Test Campaign Results** panels.

Test Campaign Configuration:

- Number of tests: 1000
- Number of iterations for infinite loops: 100
- Per test timeout (in second): 10

Test Campaign Results:

- Completed tests: 616
- No PolySpace run-time errors detected: 44
- Total failed: 572
- Number of checks/Tests with errors: 10/572
- Timeout: 0
- Stopped tests: 0

Below the configuration and results panels, there are buttons for **Stop**, **Stop All**, and **Stop Current**. A progress bar indicates the test campaign is **Running...** with **Time Remaining: 0:0:6** and **61%** completion.

The bottom section shows a table of test results:

Results	File	Line	Column	Error	# Testcases Failed
Log	example.c	104	10	IDP (Illegal Dereferen...	99
	example.c	26	2	ASRT (User Asserti...	104
	example.c	43	12	OVFL (Float Overflo...	79
	example.c	43	12	UNFL (Float Underflo...	64
	example.c	114	16	OVFL (Scalar Overfl...	83
	example.c	193	17	NTC (Non Terminatin...	51
	example.c	114	16	IDP (Illegal Dereferen...	21
	example.c	40	16	OVFL (Float Overflo...	34

5 If you want to stop the testing before it completes:

- Click **Stop Current** to stop the current test and move on to the next one.
- Click **Stop All** to immediately stop all tests.

Reviewing the Test Results

When testing is complete, the Automatic Orange Tester displays an overview of the testing results, along with detailed information about each failed test.

The screenshot displays the Automatic Orange Tester interface, divided into two main sections: 'Test Campaign Configuration' and 'Test Campaign Results'.

Test Campaign Configuration:

- Number of tests: 1000
- Number of iterations for infinite loops: 100
- Per test timeout (in second): 10

Test Campaign Results:

- Completed tests: 1000
- No PolySpace run-time errors detected: 72
- Total failed: 928
- Number of checks/Tests with errors: 10/928
- Timeout: 0
- Stopped tests: 0

Buttons: Start, Stop All, Stop Current.

Test Completed: Time Remaining: 0:0:0. Progress bar: 100%.

Results	File	Line	Column	Error	# Testcases Failed
Log	example.c	104	10	IDP (Illegal Dereference of Pointer)	166
	example.c	26	2	ASRT (User Assertion Failed)	156
	example.c	43	12	OVFL (Float Overflow)	131
	example.c	43	12	UNFL (Float Underflow)	105
	example.c	114	16	OVFL (Scalar Overflow)	129
	example.c	193	17	NTC (Non Terminating Call)	74
	example.c	114	16	IDP (Illegal Dereference of Pointer)	37
	example.c	49	16	OVFL (Float Overflow)	62

Test Campaign Results

The Test Campaign Results window displays overview information about the results of your dynamic tests, including:

- **Completed tests** – Displays the total number of tests completed.
- **No PolySpace runtime errors detected** – Displays the number of tests that did not produce a runtime error.
- **Total failed** – Displays the number of tests that produced a runtime error.
- **Number of checks/Tests with errors** – Displays the number of PolySpace checks that produced at least one failed test, as well as the total number of tests that produced a runtime error.

- **Timeout** – Displays the number of tests that exceeded the specified **Per test timeout** limit.
- **Stopped tests** – The number of tests that were stopped manually.

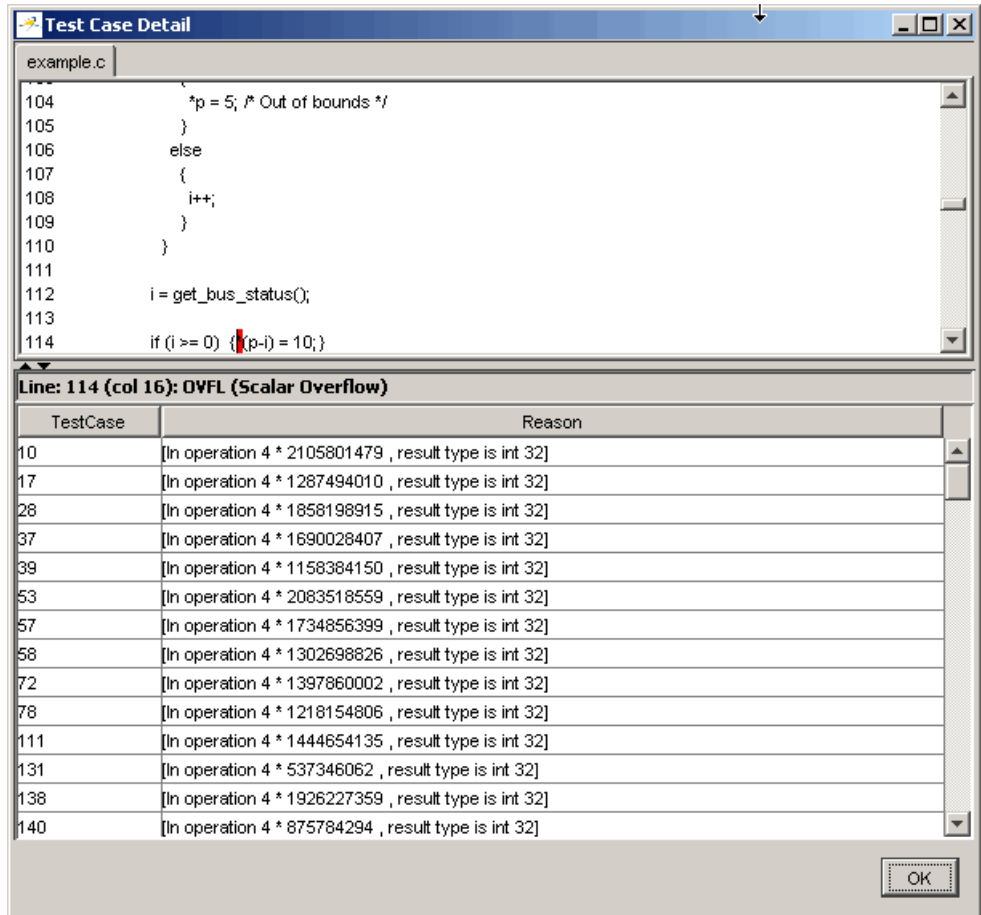
Use the Test Campaign Results Window to see an overall assessment of your test results, as well as to decide if you need to increase the **Per test timeout** value.

Results Table

The Results table displays detailed information about each failed test, to help you identify the cause of the runtime error. This information includes:

- The filename, line number, and column in which the error was found.
- The type of error that occurred.
- The number of test cases in which the error occurred.

In addition, You can view more details about any failed test by clicking on the appropriate row in the Results table. The Test Case Detail dialog box opens.



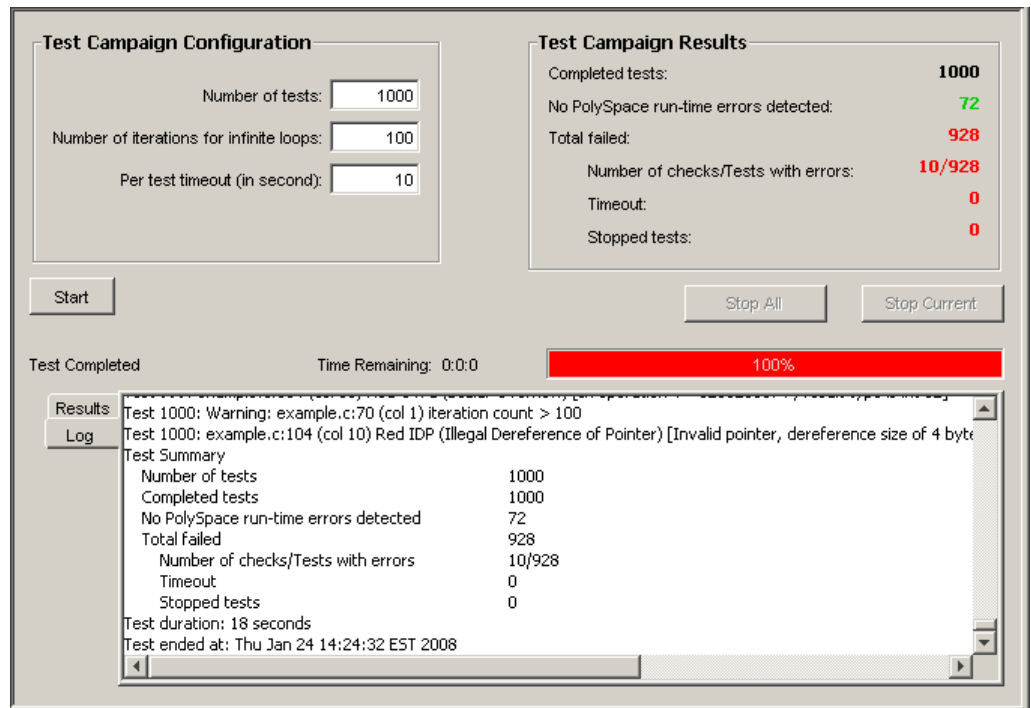
The Test Case Detail dialog box displays the portion of the code in which the error occurred, and gives detailed information about why each test case failed. Since the Automatic Orange Tester performs runtime tests, this information includes the actual values that caused the error.

You can use this information to quickly identify the cause of the error, and determine if there is an actual bug in the code.

Log

The Log window displays a complete list of all the tests which failed, as well as summary information.

You can copy information from the log window to paste into other applications, such as Microsoft® Excel®.



The log file is also saved in the PolySpace-Instrumented directory with the following filename:

`TestGenerator_day_month_year-time.out`

Refining Data Ranges

The Automatic Orange Tester allows you to specify ranges for external variables. This allows you to perform runtime tests using real-world values for your variables, rather than randomly selected values.

Setting ranges for your variables reduces the number of tests that fail due to unrealistic data values, allowing you to focus on actual problems, rather than purely theoretical problems.

To refine your data ranges:

- 1 In the Variables section at the top of the Automatic Orange Tester, identify the variable for which you want to set a data range.

The screenshot shows the PolySpace Automatic Orange Tester interface. At the top, there is a table of variables with columns for Variable Name, Type, Values, and Advanced. Below this is the Test Campaign Configuration section with input fields for Number of tests (1000), Number of iterations for infinite loops (100), and Per test timeout (10). To the right is the Test Campaign Results section showing 1000 completed tests, 80 no errors, 920 total failed, and 10/920 checks with errors. At the bottom, there is a progress bar at 100% and a table of results.

Variable Name	Type	Values	Advanced
External Scope			
Function: random_float	float32	min..max	Advanced
Function: random_int	int32	min..max	Advanced
Function: get_bus_status	int32	min..max	Advanced
Function: read_bus_status	int32	min..max	Advanced
Function: read_on_bus			

Test Campaign Configuration

Number of tests:

Number of iterations for infinite loops:

Per test timeout (in second):

Test Campaign Results

Completed tests: **1000**

No PolySpace run-time errors detected: **80**

Total failed: **920**

Number of checks/Tests with errors: **10/920**

Timeout: **0**

Stopped tests: **0**

Test Completed Time Remaining: 0:0:0 100%

Results	File	Line	Column	Error	# Testcases Failed
Log	example.c	114	16	OVFL (Scalar Overfl...	120
	example.c	104	10	IDP (Illegal Dereferen...	159
	example.c	26	2	ASRT (User Asserti...	150
	example.c	43	12	UNFL (Float Underflo...	114
	example.c	49	16	UNFL (Float Underflo...	64

2 Select **Advanced**. The Edit Values dialog box opens.

Edit Values

File: None - External Scope

Function: get_bus_status.return

Type: int32

Values: min..max

Writing mode

If the variable is a pointer this option allows the setting of the writing mode.

Write the pointed object

Writing mode : NO

SING : Only the object or first element in an array pointed to will be written.
 MULT : The complete object will be written. For example with an array all the elements will be written.

Variable Values

Single Value

Range of values min: min max

max: 0

Previous Next OK Cancel

3 Set the appropriate values for the variable:

Single Value – Specifies a constant value for the variable.

Range of values, – Specifies a minimum and maximum value for the variable.

Note For pointers, you can also specify the writing mode:

SING – The tests only write the object or first element in the array.

MULT – The tests write the complete object, or all elements in the array.

- 4 Click **Next** to edit the values for the next variable.
 - 5 When you have finished setting values, click **OK** to save your changes and close the Edit Values dialog box.
 - 6 Click **Start** to retest the code.
- The Automatic Orange Tester generates test cases, runs the tests, and displays the updated results.

The screenshot displays the PolySpace Automatic Orange Tester interface. The top section shows a tree view of the test campaign configuration with the following details:

Variable Name	Type	Values	Advanced
External Scope			
Function: random_float	float32	0..10000000	Advanced
Function: random_int	int32	min..0	Advanced
Function: get_bus_status	int32	-100..0	Advanced
Function: read_bus_status	int32	min..max	Advanced
Function: read_on_bus			

The bottom section shows the Test Campaign Configuration and Test Campaign Results:

Test Campaign Configuration:

- Number of tests: 1000
- Number of iterations for infinite loops: 100
- Per test timeout (in second): 10

Test Campaign Results:

- Completed tests: 1000
- No PolySpace run-time errors detected: 997
- Total failed: 3
- Number of checks/Tests with errors: 1/3
- Timeout: 0
- Stopped tests: 0

Buttons: Start, Stop All, Stop Current

Test Completed: Time Remaining: 0:0:0, 100%

Results	File	Line	Column	Error	# Testcases Failed
Log	example.c	114	16	IDP (Illegal Dereferen...	3

The updated results show fewer failed tests, allowing you to focus in on any actual code problems.

Saving and Reusing Your Configuration

You can save your Automatic Orange Tester preferences and variable ranges for use in future dynamic testing.

To save your configuration:

- 1 Select **File > Save**.
- 2 Enter an appropriate name and click **Save**.

Your configuration is saved in a `.tgf` file.

To open a configuration from a previous verification:

- 1 Select **File > Open**.
- 2 Select the appropriate `.tgf` file, then click **Open**.

The configuration is opened.

When you open a previously saved configuration, the **Log** window displays any differences in the configuration files. For example:

- If a variable does not exist in the new configuration, a warning is displayed.
- If the ranges for a variable are no longer valid (if the variable type changes, for example), a warning is displayed and the range is changed to the largest valid range for the new data type (if possible).

Exporting Data Ranges for PolySpace Verification

Once you have set the data ranges for your variables, you can export them to a Data Range Specifications (DRS) file for use in future PolySpace verifications. This allows you to reduce the number of orange checks identified in the PolySpace Viewer.

To export your data ranges:

- 1 Set the appropriate values for each variable you want to specify.
- 2 Select **File > Export DRS**.
- 3 Enter an appropriate name and click **Save**.

The DRS file is saved.

For information on using a DRS file for PolySpace verifications, see “Applying Data Ranges to External Variables and Stub Functions (DRS)” on page 4-25.

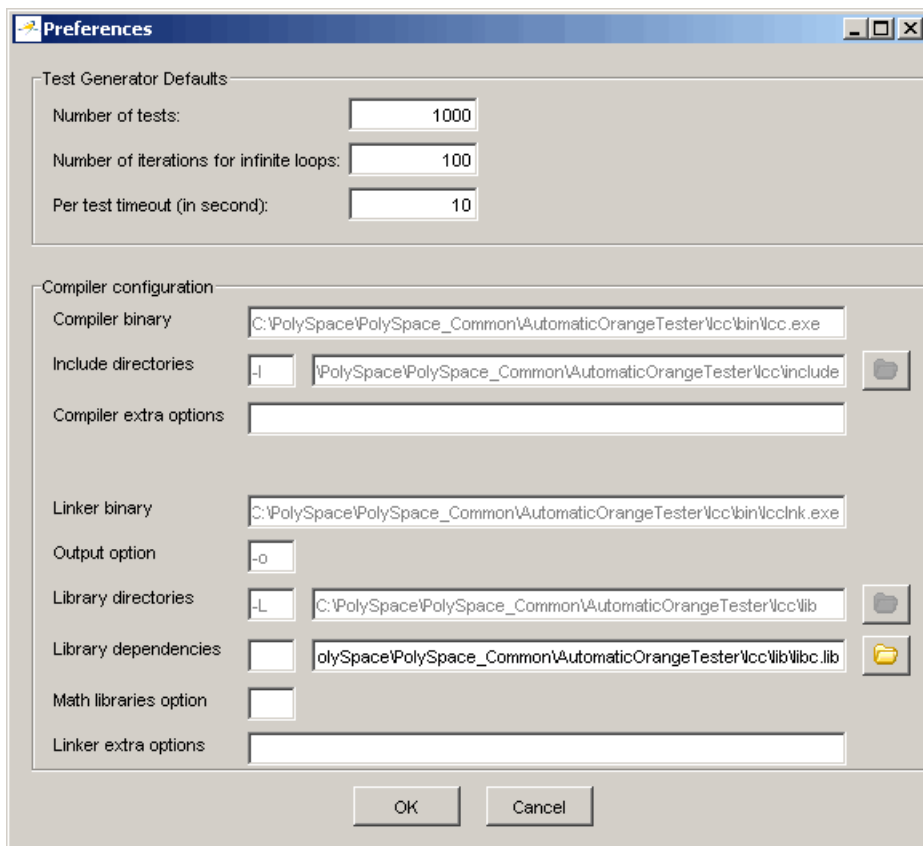
Configuring Compiler Options

On UNIX, Solaris, or Linux systems, you must configure your compiler and linker options before using the Automatic Orange Tester.

Note On Windows systems, the compiler options cannot be modified. You can only configure the library dependencies.

To set compiler and linker options:

- 1** Open the Automatic Orange Tester, as described above.
- 2** Select **Options > Configure**.
- 3** The Preferences dialog box opens.



4 Set the appropriate parameters for your compiler.

Technical Limitations

The Automatic Orange Tester has the following limitations:

- “Unsupported PolySpace Options” on page 9-47
- “Options with Limitations” on page 9-47
- “Unsupported C Language Constructions” on page 9-47

Unsupported PolySpace Options

The following options are not supported when you select `-prepare-automatic-tests`.

- `-entry-points`
- `-dialect`
- `-ignore-float-rounding`
- `-div-round-down`
- `-char-is-16its`
- `-short-is-8bits`
- `-respect-types-in-globals`
- `-respect-types-in-fields`

In addition, Global asserts in the code of the form `Pst_Global_Assert(A,B)` are not supported with the Automatic Orange Tester.

Options with Limitations

The following options cannot take specific values when you select `-prepare-automatic-tests`.

- `-target [tms320c3c | sharc21x61]`
- `-data-range-specification` (in global assert mode)

Unsupported C Language Constructions

The code verification stops when any of the following characteristics are met:

- ANSI C99 long long and long double types are unsupported for Windows systems
- Calls to following routines are unsupported:
 - va_start
 - va_arg
 - va_end
 - va_copy
 - setjmp
 - sigsetjmp
 - longjmp
 - siglongjmp

The following C language constructions are ignored:

- The endianness of the target is not managed. The tests are performed as if the user-defined target has the same endianness as the hardware on which the Automatic Orange Tester is running
- Calls to the following routines are ignored:
 - signal
 - sigset
 - sighold
 - sigrelse
 - sigpause
 - sigignore
 - sigaction
 - sigpending
 - sigsuspend
 - sigvec
 - sigblock

- sigsetmask
- sigprocmask
- siginterrupt
- srand
- srandom
- initstate
- setstate

Day to Day Use

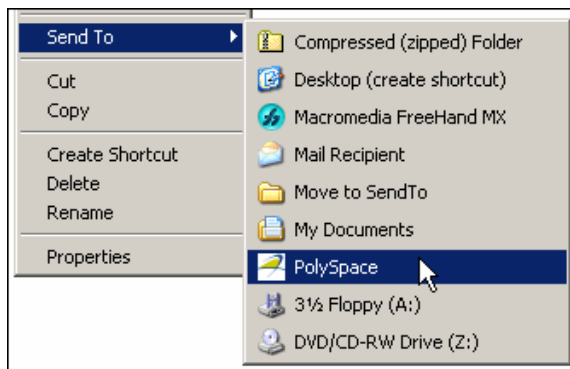
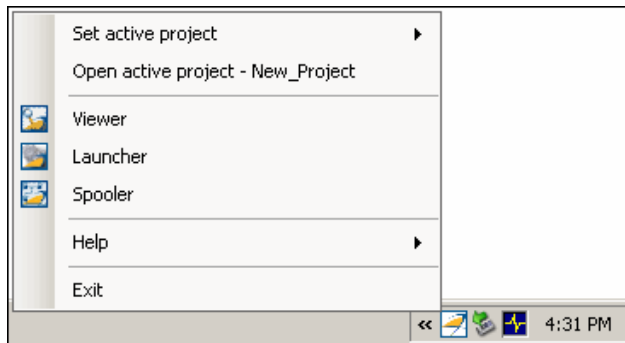
- “PolySpace In One Click Overview” on page 10-2
- “Using PolySpace In One Click” on page 10-3

PolySpace In One Click Overview

Most developers verify the same files multiple times (writing new code, unit testing, integration), and usually need to run verifications on multiple project files using the same set of options. In a Microsoft Windows environment, PolySpace In One Click provides a convenient way to streamline your work when verifying several files using the same set of options.

Once you have set up a project file with the options you want, you designate that project as the *active project*, and then send the source files to PolySpace software for verification. You do not have to update the project with source file information.

On a Windows systems, the plug-in provides a PolySpace Toolbar in the Windows Taskbar, and a **Send To** option on the desktop pop-up menu:



Using PolySpace In One Click

In this section...

“PolySpace In One Click Workflow” on page 10-3

“Setting the Active Project” on page 10-3

“Launching Verification” on page 10-5

“Using the Taskbar Icon” on page 10-9

PolySpace In One Click Workflow

Using PolySpace In One Click involves two steps:

- 1 Setting the active project.
- 2 Sending files to PolySpace software for verification.

Setting the Active Project

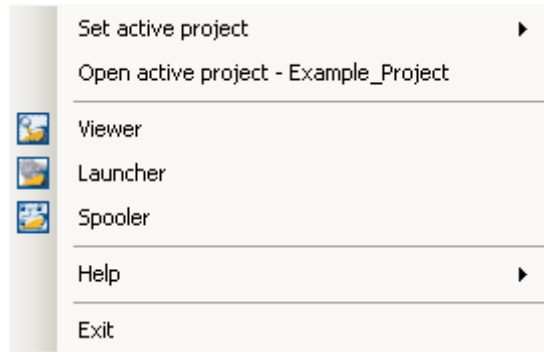
The active project is the project that PolySpace In One Click uses to verify the files that you select. Once you have set an active project, it remains active until you change the active project. PolySpace software uses the analysis options from the project; it does not use the source files or results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

To set the active project:

- 1 Right-click the PolySpace In One Click icon in the taskbar area of your Windows desktop:

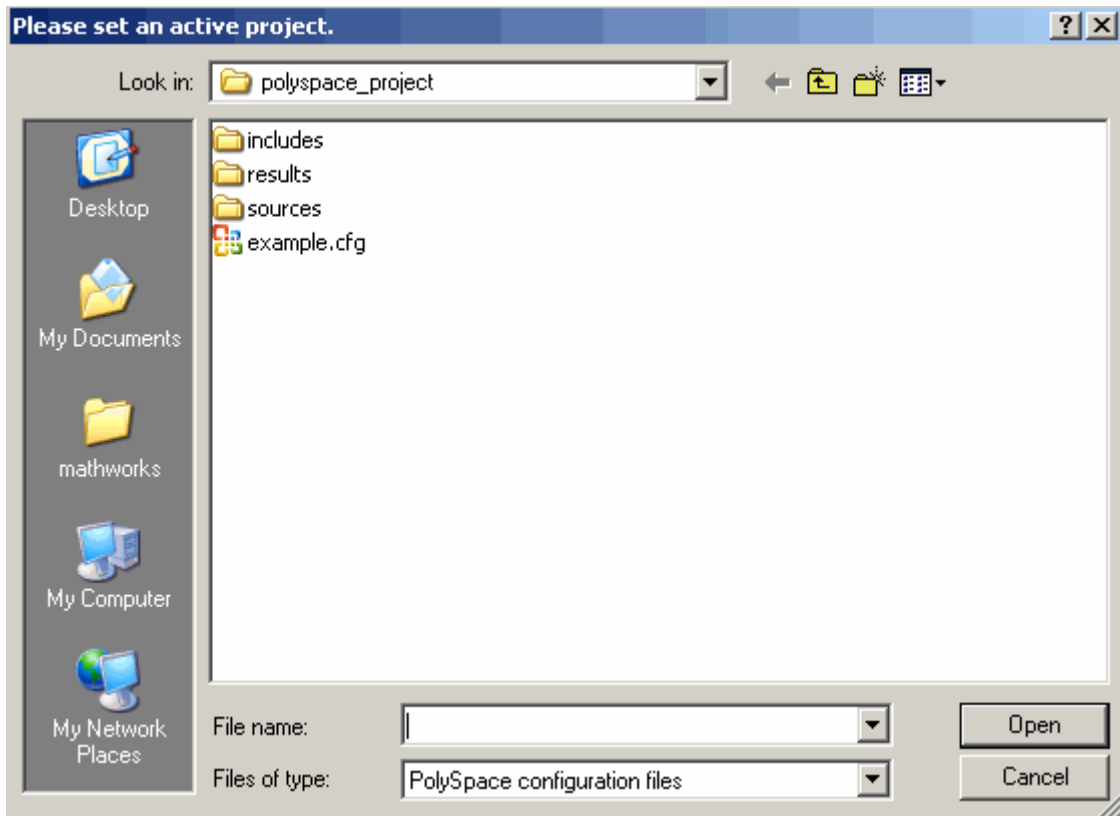


The context menu appears.



2 Select **Set active project > Browse** from the menu.

The **Please set an active project** dialog box appears:



- 3 Select the project you want to use as the active project.
- 4 Click **Open** to apply the changes and close the dialog box.

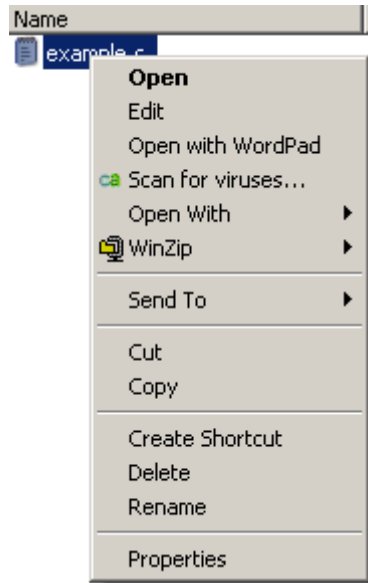
Launching Verification

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

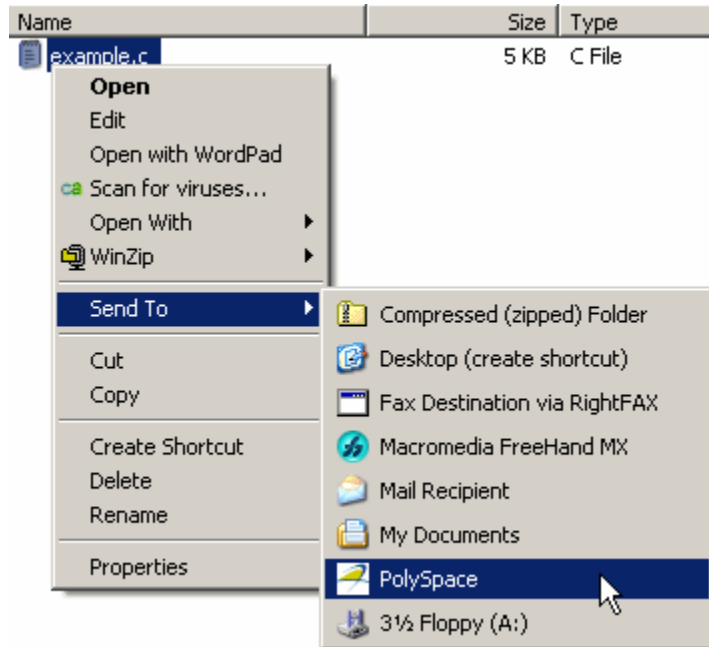
To send a file to PolySpace software for verification:

- 1 Navigate to the directory containing the source files you want to verify.
- 2 Right-click the file you want to verify.

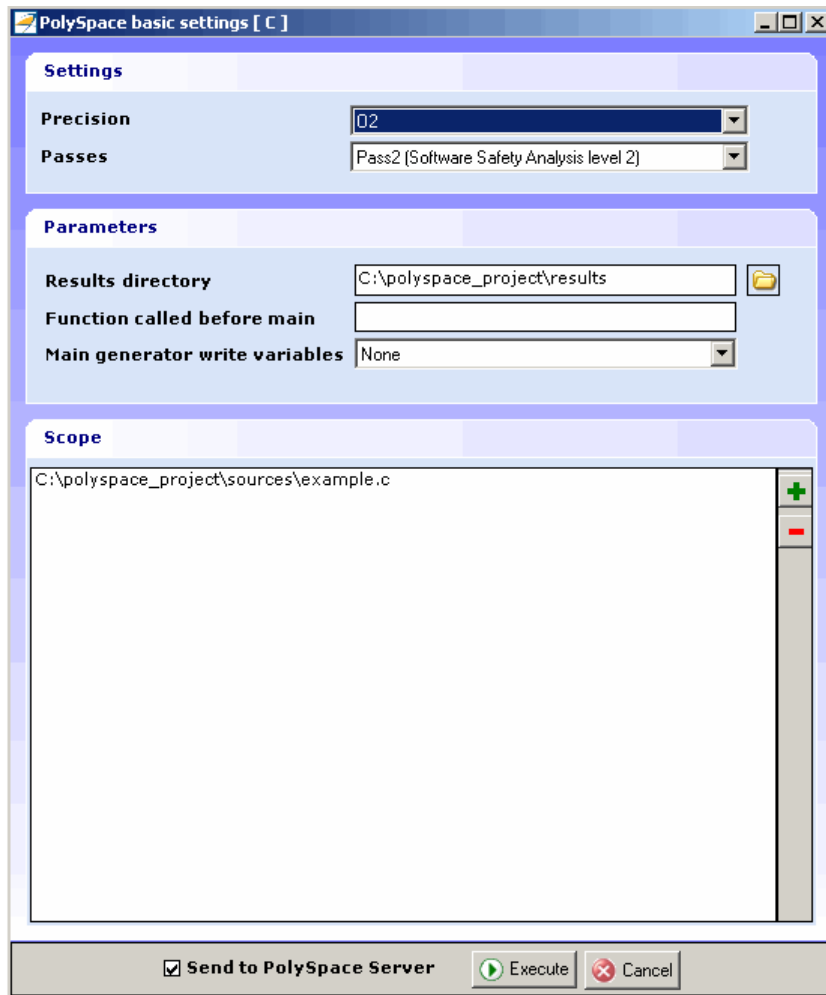
The context menu appears.



3 Select **Send To > PolySpace**.



The **PolySpace basic settings** dialog box appears.



Note The options you specify the basic settings dialog box override any options set in the configuration file. These options are also preserved between verifications.

- 4 Enter the appropriate parameters for your verification.

- 5 Leave the default values for the other parameters.
- 6 Click **Execute**.

The verification starts and the verification log appears.

```

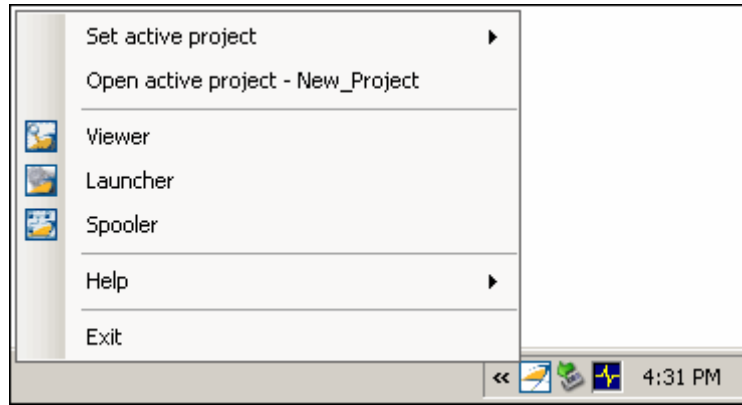
C:\polyspace_project\results\Example_Project.log
* Function random_float is pure. Returns an initialized value.
Generating the Main ...
Generating call to function: RTE
Doing code transformations ...
*****
***
*** C sources verification done
***
*****
Ending at: May 13, 2008 8:32:20
User time for suif: 5.4real, 5.4u + 0s
Generating remote file
Done
User time for polyspace-c: 5.8real, 5.8u + 0s
***
*** End of PolySpace Verifier analysis
***
Adding the analysis to the queue ...
Transferring the archive to the server ...
.....
Transfer completed.
Analysis ID : 1
The analysis has been queued. You may follow its progress using the spooler.

The analysis has been successfully done

```

Using the Taskbar Icon

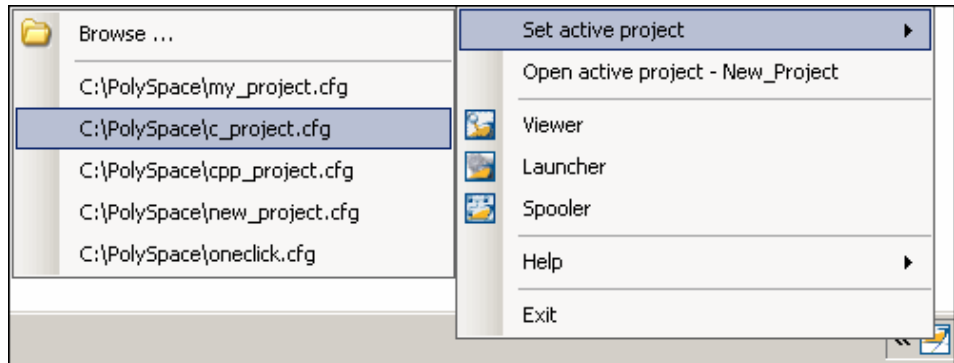
The PolySpace in One Click Taskbar icon allows you to access various software features.



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

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

A standard file browser allows you to choose the configuration file. If you have multiple configuration files, you can quickly switch between them using the browse history.



Note No configuration file is selected by default. You can create an empty file with a .cfg extension.

- **Open active project** — Opens the active configuration file. This allows you to update the project using the standard PolySpace Launcher graphical interface. It allows you to specify all PolySpace common options, including directives of compilation, options, and paths of standard and specific headers. It does not affect the precision of a verification or the results directory.
- **Viewer** — Opens the PolySpace viewer. This allows you to review verification results in the standard graphical interface. In order to load results into the viewer, you must choose a verification to review in the Verification Log window.
- **Launcher** — Opens the PolySpace Launcher. This allows you to launch a verification using the standard PolySpace graphical interface.
- **Spooler** — Opens the PolySpace Spooler. If you selected a server verification in the “PolySpace Preferences” dialog box, the spooler allows you to follow the status of the verification.

MISRA Checker

- “PolySpace MISRA Checker Overview” on page 11-2
- “Setting Up MISRA C Checking” on page 11-4
- “Running a Verification with MISRA C Checking” on page 11-10
- “Rules Supported” on page 11-14
- “Rules Partially Supported” on page 11-40
- “Rules Not Checked” on page 11-51

PolySpace MISRA Checker Overview

PolySpace software can check that C code complies with MISRA C 2004 standards.¹⁰

Note The PolySpace MISRA checker is based on MISRA C:2004 (<http://www.misra-c.com>).

The MISRA checker enables PolySpace software to provide messages when MISRA C rules are not respected. Most messages are reported during the compile phase of a verification. The MISRA checker can check nearly all of the 141 MISRA C:2004 rules.

These 142 rules are divided in three categories:

- **102** required and advisory rules fully supported. PolySpace software can check all these rules without any limitations. See “Rules Supported” on page 11-14.
- **20** required and advisory rules partially supported. PolySpace software can check all these rules with some limitations. These limitations are described in the associated “**Note**” paragraph for each rule. See “Rules Partially Supported” on page 11-40.
- **20** required and advisory rules which cannot be verified by PolySpace software. These rules cannot be verified because they are outside the scope of PolySpace verification. They may concern documentation, dynamic aspects or functional aspects of MISRA rules. These rules are not checked. The “**comment**” column details the reason. See “Rules Not Checked” on page 11-51.

10. MISRA and MISRA C are registered trademarks of MISRA Ltd., held on behalf of the MISRA Consortium.

Note Every violation, warning or error, will be written in the log file at compilation time of a PolySpace verification, except for rules 9.1 (NIV checks), 12.11 (OVFL check using `-detect-unsigned-overflows`), 13.7 (gray checks), 14.1 (gray checks), 16.2 (Call graph) and 21.1 (all runtime errors).

You will find a set of required and advisory MISRA rules in “Applying Coding Rules to Reduce Orange Checks” on page 9-8 which can have direct or indirect impact on the PolySpace selectivity (reliability percentage).

Setting Up MISRA C Checking

In this section...

“Checking Compliance with MISRA C Coding Rules” on page 11-4

“Creating a MISRA C Rules File” on page 11-5

“Excluding Files from the MISRA C Checking” on page 11-7

“Configuring Text and XML Editors” on page 11-8

Checking Compliance with MISRA C Coding Rules

To check MISRA C 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 MISRA C violations, you run the verification again.

To set the MISRA C checking option:

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

The Compliance with standards options appear.

- 2** Select the **Check MISRA-C:2004 rules** check box.
- 3** Expand the **Check MISRA-C:2004 rules** option.

Two options, **Rules configuration** and **Files and directories to ignore**, appear.

Name	Value		Internal name
Analysis options			
+ General			
+ Target/Compilation			
- Compliance with standards			
- Code from DOS or Windows filesystem	<input checked="" type="checkbox"/>		-dos
+ Embedded assembler			
+ Strict	<input type="checkbox"/>		-strict
+ Permissive	<input type="checkbox"/>		-permissive
- Check MISRA-C:2004 rules	<input checked="" type="checkbox"/>		
- Rules configuration		...	-misra2
- Files and directories to ignore		...	-includes-to-ignore
+ KeilMAR support	default	▼	-dialect
+ PolySpace inner settings			
+ Precision/Scaling			
+ Multitasking			


- 4** Specify which MISRA C rules to check and which, if any, files to exclude from the checking.

Creating a MISRA C Rules File

You must have a rules file to run a verification with MISRA C checking.

Opening a New Rules File

To open a new rules file:

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

A window for opening or creating a MISRA C rules file appears.

- 2** Select **File > New File**.

A table of rules appears.

Rules	Error	Warning	Off
MISRA C rules			
— Number of rules by mode :	7	1	134
+1 Environment			
+2 Language extensions			
+3 Documentation			
+4 Character sets			
+5 Identifiers			
+6 Types			
+7 Constants			
+8 Declarations and definitions			
+9 Initialisation			
+10 Arithmetic type conversions			
+11 Pointer type conversions			
+12 Expressions			
+13 Control statement expressions			
+14 Control flow			
+15 Switch statements			
-16 Functions			
—16.1 Functions shall not be defined with variable numbers of arguments.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
—16.2 Functions shall not call themselves, either directly or indirectly.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
—16.3 Identifiers shall be given for all of the parameters in a function prototype.	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
—16.4 The identifiers used in the declaration and definition of a function shall match.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
—16.5 Functions with no parameters shall be declared with parameter type void.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
—16.6 The number of arguments passed to a function shall match the number in the function prototype.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
—16.7 A pointer parameter in a function prototype should be declared as pointer to void.	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>
—16.8 All exit paths from a function with non-void return type shall have an explicit return statement.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
—16.9 A function identifier shall only be used with either a preceding &, or with a preceding *.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
—16.10 If a function returns error information, then that error information shall be returned via a pointer to a structure.	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>
-17 Pointer and arrays			
—17.1 Pointer arithmetic shall only be applied to pointers that address an array of the same type.	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>
—17.2 Pointer subtraction shall only be applied to pointers that address elements of the same type.	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>
—17.3 >, >=, <, <= shall not be applied to pointer types except where they point to the same object.	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>
—17.4 Array indexing shall be the only allowed form of pointer arithmetic.	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
—17.5 The declaration of objects should contain no more than 2 levels of pointer indirection.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
—17.6 The address of an object with automatic storage shall not be assigned to a pointer with static storage.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
+18 Structures and unions			
+19 Preprocessing directives			
+20 Standard libraries			
+21 Run-time failures			

3 For each rule, you 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).

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


The **Save as** dialog box opens.

5 In **File**, enter a name for the rules file.

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

Excluding Files from the MISRA C Checking

You can exclude files from MISRA C checking. You might want to exclude some included files. To exclude `math.h` from the MISRA C checking of the project `example.cfg`:

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

2 Click the folder icon.



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

3 Select the files or directories (such as include files) you want to ignore.

4 Click **OK**.

The selected files appear in the list of files to ignore.

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

Configuring Text and XML Editors

Before you check MISRA rules, 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 MISRA reports directly from the MISRA-C log in the viewer.

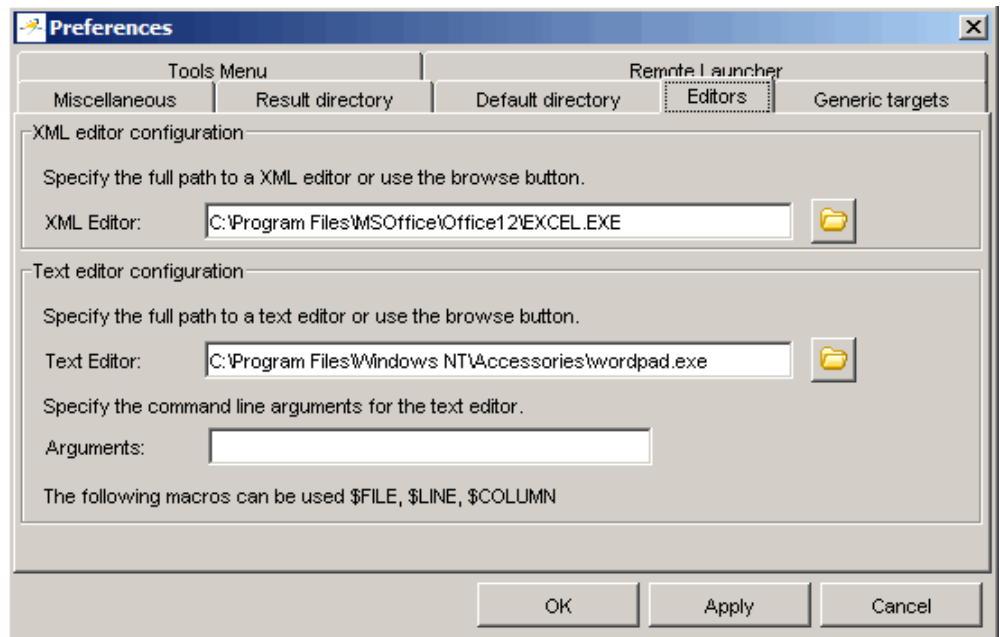
To configure your text and .XML editors:

1 Select **Edit > Preferences**.

The Preferences dialog box opens.

2 Select the **Editors** tab.

The Editors tab opens.



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

Running a Verification with MISRA C Checking

In this section...

“Starting the Verification” on page 11-10

“Examining the MISRA C Log” on page 11-11

“Opening MISRA-C Report” on page 11-12

Starting the Verification

When you run a verification with the MISRA C option selected, the verification checks most of the MISRA C rules during the compile phase.¹¹

Note Some rules address run-time errors.

The verification stops if there is a violation of a rule with state **Error**.

To start the verification:

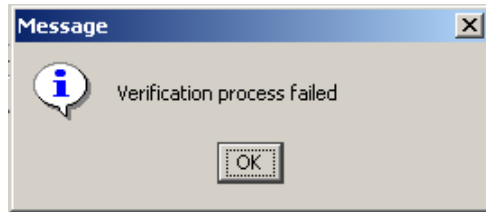
1 Click the **Execute** button



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.

If the verification fails because of MISRA C violations. A message dialog box appears.

11. MISRA and MISRA C are registered trademarks of MISRA Ltd., held on behalf of the MISRA Consortium.



3 Click **OK**.

Examining the MISRA C Log

To examine the MISRA C violations:

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

A list of MISRA C violations appear in the log part of the window.

```
ERROR : rule 16.3 (required) violated. At : C:\po:
| identifiers shall be given for all of the
WARNING : rule 17.4 (required) violated. At : exar
| array indexing shall be the only allowed
WARNING : rule 17.4 (required) violated. At : exar
| array indexing shall be the only allowed
```

2 Click on any of the violations to see a description of the violated rule, the full path of the file in which the violation was found, and the source code containing the violation.

Status	Rule	File	Line	Col
!	16.3	include.h	33	0
?	17.4	example.c	97	0
?	17.4	example.c	114	0
?	17.4	example.c	118	0

Search: << >>

Detail

Rule: 16.3 (Error): Identifiers shall be given for all of the parameters in a

File: C:\PolySpace\polyspace_project\includes\include.h line 33 (column 0)

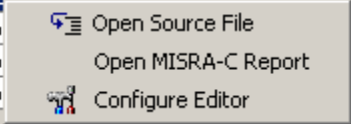
Source code

```
|extern void Exec_One_Cycle (int);
```

In this example, the log reports a violation of rule 16.3. A function prototype declaration in `include.h` is missing an identifier.

- 3 Right click the row containing the violation, then select Open Source File.

Status	Rule	File	Line	Col
!	16.3	include.h	33	0
?	17.4	examp		
?	17.4	examp		
?	17.4	examp		



The appropriate 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 11-8.

- 4 Correct the MISRA violation and run the verification again.

Opening MISRA-C Report

After you check MISRA rules, you can generate an XML report containing all the errors and warnings reported by the MISRA-C checker.

Note You must configure an XML editor before you can open a MISRA-C report. See “Configuring Text and XML Editors” on page 11-8.

To view the MISRA-C report:

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

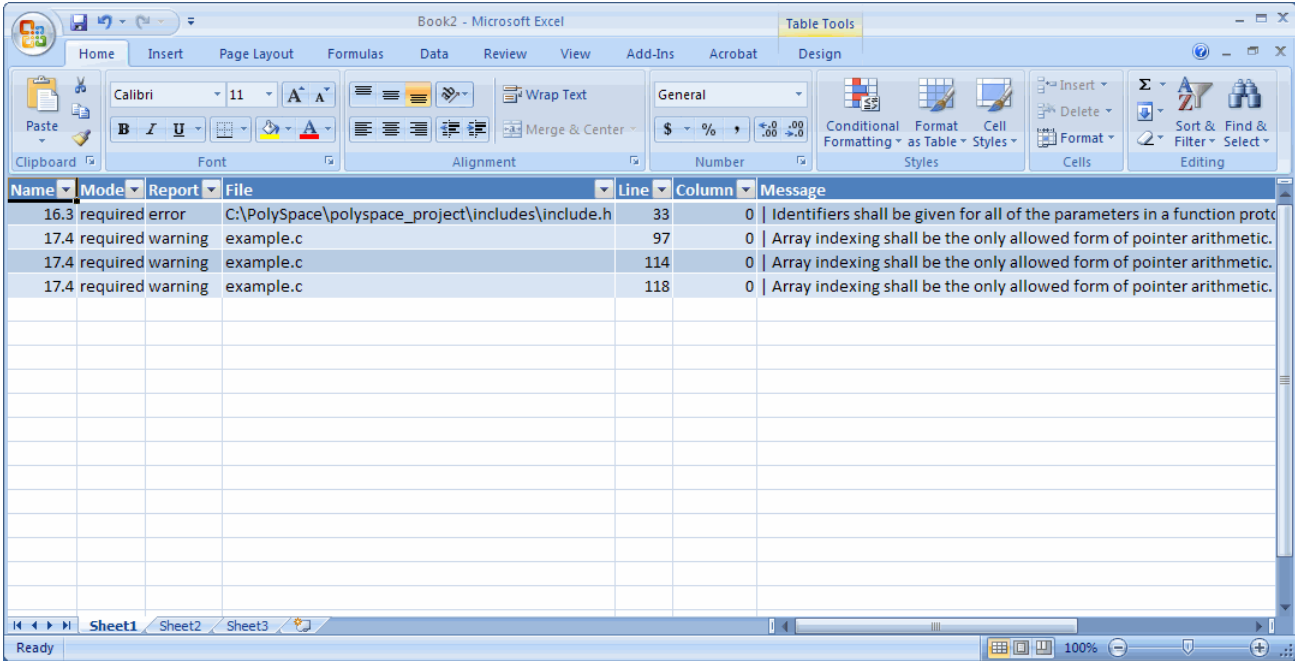
A list of MISRA C violations appear in the log part of the window.

- 2 Right click any row in the log, and select **Open MISRA-C Report**.

Status	Rule	File	Line	Col
!	16.3	include.h	33	0
?	17.4	examp		
?	17.4	examp		
?	17.4	examp		

- Open Source File
- Open MISRA-C Report
- Configure Editor

The report opens in your XML editor.



Rules Supported

In this section...
“Language Extensions” on page 11-15
“Character Sets” on page 11-15
“Identifiers” on page 11-16
“Types” on page 11-17
“Constants” on page 11-17
“Declarations and Definitions” on page 11-18
“Initialization” on page 11-20
“Arithmetic Type Conversion” on page 11-20
“Pointer Type Conversion” on page 11-24
“Expressions” on page 11-25
“Control Statement Expressions” on page 11-28
“Control Flow” on page 11-29
“Switch Statements” on page 11-31
“Functions” on page 11-32
“Pointers and Arrays” on page 11-33
“Structures and Unions” on page 11-33
“Preprocessing Directives” on page 11-34
“Standard Libraries” on page 11-37
“runtime Failures” on page 11-39

Language Extensions

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
2.2	source code shall only use /* */ style comments	C++ comments shall not be used.	C++ comments are handled as comments but lead to a violation of this MISRA rule
2.3	The character sequence /* shall not be used within a comment	The character sequence /* shall not appear within a comment.	This rule violation is also raised when the character sequence /* inside a C++ comment.

Character Sets

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
4.1	Only those escape sequences which are defined in the ISO® C standard shall be used.	\<character> is not an ISO C escape sequence Only those escape sequences which are defined in the ISO C standard shall be used.	
4.2	Trigraphs shall not be used.	Trigraphs shall not be used.	Trigraphs are handled and converted to the equivalent character but lead to a violation of the MISRA rule

Identifiers

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	<ul style="list-style-type: none"> • Local declaration of XX is hiding another identifier. • Declaration of parameter XX is hiding another identifier. 	Assumes that rule 8.1 is not violated.
5.3	A typedef name shall be a unique identifier	{ typedef name }'%s' should not be reused. (already used as { typedef name } at %s:%d)	Warning when a typedef name is reused as another identifier name.
5.4	A tag name shall be a unique identifier	{tag name }'%s' should not be reused. (already used as {tag name } at %s:%d)	warning when a tag name is reused as another identifier name
5.5	No object or function identifier with a static storage duration should be reused.	{ static identifier/parameter name }'%s' should not be reused. (already used as { static identifier/parameter name } at %s:%d)	warning when a static name is reused as another identifier name
5.6	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.	{member name }'%s' should not be reused. (already used as { member name } at %s:%d)	warning when a idf in a namespace is reused in another namespace
5.7	No identifier name should be reused.	{identifier}'%s' should not be reused. (already used as { identifier} at %s:%d)	warning on other conflicts (including member names)

Types

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
6.1	The plain char type shall be used only for the storage and use of character values	Only permissible operators on plain chars are '=', '==' or '!=' operators.	There is a warning when a plain char is used with an operator other than =, == or !=.
6.3	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types	typedefs that indicate size and signedness should be used in place of the basic types.	No warning is given in typedef definition. There is no exception on bitfields.
6.4	Bit fields shall only be defined to be of type <i>unsigned int</i> or <i>signed int</i> .	Bit fields shall only be defined to be of type unsigned int or signed int.	
6.5	Bit fields of type <i>signed int</i> shall be at least 2 bits long.	Bit fields of type signed int shall be at least 2 bits long.	No warning on anonymous signed int bitfields of width 0 - Extended to all signed bitfields of size <= 1 (if Rule 6.4 is violated).

Constants

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.	<ul style="list-style-type: none"> • Octal constants other than zero and octal escape sequences shall not be used. • Octal constants (other than zero) should not be used. • Octal escape sequences should not be used. 	

Declarations and Definitions

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	<ul style="list-style-type: none"> • Function XX has no complete prototype visible at call. • Function XX has no prototype visible at definition. 	Prototype visible at call must be complete.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated	Whenever an object or function is declared or defined, its type shall be explicitly stated.	
8.4	If objects or functions are declared more than once their types shall be compatible.	<ul style="list-style-type: none"> • If objects or functions are declared more than once their types shall be compatible. • Global declaration of 'XX' function has incompatible type with its definition. • Global declaration of 'XX' variable has incompatible type with its definition. 	During link phase, errors are converted into warnings with <code>-permissive-link</code> option. Cannot be turned Off.
8.5	There shall be no definitions of objects or functions in a header file	<ul style="list-style-type: none"> • Object 'XX' should not be defined in a header file. • Function 'XX' should not be defined in a header file. 	Tentative of definitions are considered as definitions.

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
8.6	Functions shall always be declared at file scope.	Function 'XX' should be declared at file scope.	
8.9	Definition: An identifier with external linkage shall have exactly one external definition.	<ul style="list-style-type: none"> • Procedure/Global variable XX multiply defined. • Forbidden multiple tentative of definition for object XX. • Global variable has multiples tentative of definitions 	Tentative of definitions are considered as definitions, No warning on undefined objects with <code>-allow-undef-variables</code> option, No warning on predefined symbols.
8.10	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required	Function/Variable XX should have internal linkage.	Not checked if <code>-main-generator</code> option is set. Assumes that 8.1 is not violated. No warning if 0 uses.
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage	static storage class specifier should be used on internal linkage symbol XX.	
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization	Array XX has unknown size.	

Initialization

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
9.1	All automatic variables shall have been assigned a value before being used.		Done by PolySpace (NIV Checks). Cannot be Off.
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	

Arithmetic Type Conversion

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
10.1	<p>The value of an expression of integer type shall not be implicitly converted to a different underlying type if:</p> <ul style="list-style-type: none"> it is not a conversion to a wider integer type of the same signedness, or the expression is complex, or the expression is not constant and is a function argument, or 	<ul style="list-style-type: none"> Implicit conversion of the expression of underlying type ?? to the type ?? that is not a wider integer type of the same signedness. Implicit conversion of one of the binary operands whose underlying types are ?? and ?? Implicit conversion of the binary right hand 	<ol style="list-style-type: none"> ANSI C base types order (signed char, short, int, long) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1. The same interpretation is applied on the unsigned version of base types. An expression of bool or enum types has int as underlying type.

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
	<ul style="list-style-type: none"> the expression is not constant and is a return expression 	<p>operand of underlying type ?? to ?? that is not an integer type.</p> <ul style="list-style-type: none"> Implicit conversion of the binary left hand operand of underlying type ?? to ?? that is not an integer type. Implicit conversion of the binary right hand operand of underlying type ?? to ?? that is not a wider integer type of the same signedness or Implicit conversion of the binary ? left hand operand of underlying type ?? to ??, but it is a complex expression. 	<p>3 Plain char may have signed or unsigned underlying type (depending on PolySpace target configuration or option setting).</p> <p>4 The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width is not taken into account and it assumes that only signed unsigned int are used for bitfield (Rule 6.4).</p>
10.1 (cont.)		<ul style="list-style-type: none"> Implicit conversion of complex integer expression of underlying type ?? to ??. Implicit conversion of non-constant integer expression of underlying type ?? in function return whose expected type is ??. Implicit conversion of non-constant integer expression of underlying type ?? as argument of function whose 	

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
		corresponding parameter type is ??.	
10.2	<p>The value of an expression of floating type shall not be implicitly converted to a different type if</p> <ul style="list-style-type: none"> • it is not a conversion to a wider floating type, or • the expression is complex, or • the expression is a function argument, or • the expression is a return expression 	<ul style="list-style-type: none"> • Implicit conversion of the expression from ?? to ?? that is not a wider floating type. • Implicit conversion of the binary ? right hand operand from ?? to ??, but it is a complex expression. • Implicit conversion of the binary ? right hand operand from ?? to ?? that is not a wider floating type or Implicit conversion of the binary ? left hand operand from ?? to ??, but it is a complex expression. • Implicit conversion of complex floating expression from ?? to ??. • Implicit conversion of floating expression of ?? type in function return whose expected type is ??. • Implicit conversion of floating expression of ?? type as argument of function whose 	ANSI C base types order (float, double) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1.

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
		corresponding parameter type is ??.	
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression	Complex ppliedta of underlying type ?? may only be cast to narrower integer type of same signedness, however the destination type is ??.	<ul style="list-style-type: none"> • ANSI C base types order (signed char, short, int, long) defines that T1 is narrower than T2 if T2 is on the right hand of T1 or T1 = T2. The same ppliedtation is applied on the unsigned version of base types. • An expression of bool or enum types has int as underlying type. • Plain char may have signed or unsigned underlying type (depending on target configuration or option setting). • The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width is not taken into account and it assumes that only signed, unsigned int are used for bitfield (Rule 6.4).

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
10.4	The value of a complex expression of float type may only be cast to narrower floating type	Complex expression of ?? type may only be cast to narrower floating type, however the destination type is ??.	ANSI C base types order (float, double) defines that T1 is narrower than T2 if T2 is on the right hand of T1 or T2 = T1.
10.5	If the bitwise operator ~ and << are applied to an operand of underlying type <i>unsigned char</i> or <i>unsigned short</i> , the result shall be immediately cast to the underlying type of the operand	Bitwise [<< ~] is applied to the operand of underlying type [unsigned char unsigned short], the result shall be immediately cast to the underlying type.	
10.6	The “U” suffix shall be applied to all constants of <i>unsigned</i> types	No explicit ‘U suffix on constants of an unsigned type.	

Pointer Type Conversion

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type	Conversion shall not be performed between a pointer to a function and any type other than an integral type.	Casts and implicit conversions involving a function pointer
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.	There is also a warning on qualifier loss

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
11.3	A cast should not be performed between a pointer type and an integral type	A cast should not be performed between a pointer type and an integral type.	Exception on zero constant. Extended to all conversions
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.	A cast should not be performed between a pointer to object type and a different pointer to object type.	Extended to all conversions
11.5	A cast shall not be performed that removes any <i>const</i> or <i>volatile</i> qualification from the type addressed by a pointer	A cast shall not be performed that removes any <i>const</i> or <i>volatile</i> qualification from the type addressed by a pointer	Extended to all conversions

Expressions

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
12.1	Limited dependence should be placed on C's operator precedence rules in expressions	Limited dependence should be placed on C's operator precedence rules in expressions	
12.3	The <i>sizeof</i> operator should not be used on expressions that contain side effects.	he size of operator should not be used on expressions that contain side effects.	No warning on volatile accesses and function calls
12.4	The right hand operand of a logical && or operator shall not contain side effects.	The right hand operand of a logical && or operator shall not contain side effects.	No warning on volatile accesses and function calls.

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
12.5	The operands of a logical && or shall be primary-expressions.	<ul style="list-style-type: none"> • operand of logical && is not a primary expression • operand of logical is not a primary expression • The operands of a logical && or shall be primary-expressions. 	<p>During preprocessing, violations of this rule are detected on the expressions in #if directives.</p> <p>Allowed exception on associatively (a && b && c), (a b c).</p>
12.6	Operands of logical operators (&&, and !) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (&&, or !).	<ul style="list-style-type: none"> • Operand of '!' logical operator should be effectively Boolean. Left operand of '%s' logical operator should be effectively Boolean. • Right operand of '%s' logical operator should be effectively Boolean. • Boolean should not be used as operands to operators other than '&&', ' ' or '!'. 	<p>"the operand of a logical operator should be a Boolean". As there are no Boolean in "C" but as the standard assumes it, some operator return Boolean like expression (var == 0).</p> <p>Example:</p> <pre>unsigned char flag; if (!flag) raises the rule: the operand of "!" is "flag". And "flag" is not a Boolean but an unsigned char. To be 12.6 MISRA compliant, the code need to be written like this:</pre> <pre>if (!(flag != 0)) or if (flag == 0)</pre>

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed	<ul style="list-style-type: none"> • [~/Left Shift/Right shift/&] operator applied on an expression whose underlying type is signed. • Bitwise ~ on operand of signed underlying type ??. • Bitwise [<< >>] on left hand operand of signed underlying type ??. • Bitwise [& ^] on two operands of s 	<p>The underlying type for an integer used in a re-processor expression is signed when :</p> <ul style="list-style-type: none"> • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number
12.8	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	<ul style="list-style-type: none"> • shift amount is negative • shift amount is bigger than 64 • Bitwise [<< >>] count out of range [0 ..X] (width of the underlying type ?? of the left hand operand - 1).. 	<p>The numbers that are manipulated in preprocessing directives are 64 bits wide so that valid shift range is between 0 and 63</p> <p>Check is also extended onto bitfields with the field width or the width of the base type when it is within a complex expression</p>

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	<ul style="list-style-type: none"> • Unary - on operand of unsigned underlying type ??. • Minus operator applied to an expression whose underlying type is unsigned 	The underlying type for an integer used in a re-processor expression is signed when: <ul style="list-style-type: none"> • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number
12.10	The comma operator shall not be used.	The comma operator shall not be used.	
12.13	The increment (++) and decrement (–) operators should not be mixed with other operators in an expression	The increment (++) and decrement (–) operators should not be mixed with other operators in an expression	warning when ++ or – operators are not used alone.

Control Statement Expressions

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
13.1	Assignment operators shall not be used in expressions that yield Boolean values.	Assignment operators shall not be used in expressions that yield Boolean values.	

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	No warning is given on integer constants. Example: if (2)
13.7	Boolean operations whose results are invariant shall not be permitted	Boolean operator '%s' should not have invariant result. (Result is always 'true/false').	Done by PolySpace (gray Checks). It is also checked during compilation on comparison between with a least one constant operand. Cannot be Off.

Control Flow

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
14.1	There shall be no unreachable code.		Done by PolySpace (gray checks). Cannot be Off.
14.2	All non-null statements shall either have at least one side effect however executed, or cause control flow to change	<ul style="list-style-type: none"> • All non-null statements shall either: • have at least one side effect however executed, or • cause control flow to change 	
14.4	The <i>goto</i> statement shall not be used.	The <i>goto</i> statement shall not be used.	
14.5	The <i>continue</i> statement shall not be used.	The <i>continue</i> statement shall not be used.	

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
14.6	For any iteration statement there shall be at most one <i>break</i> statement used for loop termination	For any iteration statement there shall be at most one break statement used for loop termination	
14.7	A function shall have a single point of exit at the end of the function	A function shall have a single point of exit at the end of the function	
14.8	The statement forming the body of a <i>switch</i> , <i>while</i> , <i>do while</i> or <i>for</i> statement shall be a compound statement	<ul style="list-style-type: none"> • The body of a do while statement shall be a compound statement. • The body of a for statement shall be a compound statement. • The body of a switch statement shall be a compound statement 	
14.9	An <i>if (expression)</i> construct shall be followed by a compound statement. The <i>else</i> keyword shall be followed by either a compound statement, or another <i>if</i> statement	<ul style="list-style-type: none"> • An if (expression) construct shall be followed by a compound statement. • The else keyword shall be followed by either a compound statement, or another if statement 	
14.10	All <i>if else if</i> constructs should contain a final <i>else</i> clause.	All if else if constructs should contain a final else clause.	

Switch Statements

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
15.0	<p>Unreachable code is detected between switch statement and first case.</p> <hr/> <p>Note this is not a MISRA C2004 rule.</p> <hr/>	switch statements syntax normative restrictions.	<p>On the following example, the rule is displayed in the log file at line 3:</p> <pre> 1 ... 2 switch(index) { 3 var = var + 1; // RULE 15.0 // violated 4 case 1: ...</pre> <p>The code between switch statement and first case is checked as gray by PolySpace verification. It follows ANSI standard behavior.</p>
15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a <i>switch</i> statement	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	
15.2	An unconditional <i>break</i> statement shall terminate every non-empty switch clause	An unconditional break statement shall terminate every non-empty switch clause	
15.3	The final clause of a <i>switch</i> statement shall be the <i>default</i> clause	The final clause of a switch statement shall be the default clause	

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
15.4	A <i>switch</i> expression should not represent a value that is effectively Boolean	A switch expression should not represent a value that is effectively Boolean	
15.5	Every <i>switch</i> statement shall have at least one <i>case</i> clause	Every switch statement shall have at least one case clause	

Functions

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
16.1	Functions shall not be defined with variable numbers of arguments.	Function XX should not be defined as varargs.	
16.2	Functions shall not call themselves, either directly or indirectly.	Function %s should not call itself.	Done by PolySpace software (Call graph in the viewer gives the information). PolySpace verification also checks that partially during compilation phase. Cannot be Off.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.	Identifiers shall be given for all of the parameters in a function prototype declaration.	Assumes Rule 8.6 is not violated.
16.5	Functions with no parameters shall be declared with parameter type <i>void</i> .	Functions with no parameters shall be declared with parameter type <i>void</i> .	Definitions are also checked.

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Missing return value for non-void function XX.	Warning when a non-void function is not terminated with an unconditional return with an expression.
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.	Function identifier XX should be preceded by a & or followed by a parameter list.	

Pointers and Arrays

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
17.5	A type should not contain more than 2 levels of pointer indirection	A type should not contain more than 2 levels of pointer indirection	

Structures and Unions

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
18.1	All structure or union types shall be complete at the end of a translation unit.	All structure or union types shall be complete at the end of a translation unit.	
18.4	Unions shall not be used	Unions shall not be used.	

Preprocessing Directives

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
19.1	#include statements in a file shall only be preceded by other preprocessors directives or comments	A message is displayed when a #include directive is preceded by other things than preprocessor directives, comments, spaces or “new lines”.	
19.2	Nonstandard characters should not occur in header file names in #include directives	<ul style="list-style-type: none"> • A message is displayed on characters ', \, " or /* between < and > in #include <filename> • A message is displayed on characters ', \ or /* between " and " in #include "filename" 	
19.3	The #include directive shall be followed by either a <filename> or "filename" sequence.	<ul style="list-style-type: none"> • '#include' expects "FILENAME" or <FILENAME> • '#include_next' expects "FILENAME" or <FILENAME> 	Cannot be Off.
19.5	Macros shall not be #defined and #undefd within a block.	<ul style="list-style-type: none"> • Macros shall not be #defined within a block. • Macros shall not be #undef'd within a block. 	
19.6	#undef shall not be used.	#undef shall not be used.	

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
19.7	A function should be used in preference to a function like-macro.	Message on all function-like macros expansions	
19.8	A function-like macro shall not be invoked without all of its arguments	<ul style="list-style-type: none"> • arguments given to macro '<name>' • macro '<name>' used without args. • macro '<name>' used with just one arg. • macro '<name>' used with too many (<number>) args. 	Cannot be Off.
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	Macro argument shall not look like a preprocessing directive.	This rule is detected as violated when the '#' character appears in a macro argument (outside a string or character constant)
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.	Parameter instance shall be enclosed in parentheses.	
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	'<name>' is not defined.	

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.	More than one occurrence of the # or ## preprocessor operators.	
19.13	The # and ## preprocessor operators should not be used	Message on definitions of macros using # or ## operators	
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.	'defined' without an identifier.	Cannot be Off.
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	directive is not syntactically meaningful.	
19.17	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	<ul style="list-style-type: none"> • #elif not within a conditional. • #else not within a conditional. • #elif not within a conditional. • #endif not within a conditional. • unbalanced #endif. • unterminated #if conditional. • unterminated #ifdef conditional. • unterminated #ifndef conditional. 	Cannot be Off.

Standard Libraries

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.	<ul style="list-style-type: none"> • The macro ‘<name> shall not be redefined. • The macro ‘<name> shall not be undefined. 	
20.2	The names of standard library macros, objects and functions shall not be reused.	Identifier XX should not be used.	In case a macro whose name corresponds to a standard library macro, object or function is defined, the rule that is detected as violated is 20.1 . Tentative of definitions are considered as definitions.
20.4	Dynamic heap memory allocation shall not be used.	<ul style="list-style-type: none"> • The macro ‘<name> shall not be used. • Identifier XX should not be used. 	In case the dynamic heap memory allocation functions are actually macros and the macro is expanded in the code, this rule is detected as violated. Assumes rule 20.2 is not violated.
20.5	The error indicator errno shall not be used	The error indicator errno shall not be used	Assumes that rule 20.2 is not violated
20.6	The macro <i>offsetof</i> , in library <stddef.h>, shall not be used.	<ul style="list-style-type: none"> • The macro ‘<name> shall not be used. • Identifier XX should not be used. 	Assumes that rule 20.2 is not violated

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
20.7	The <i>setjmp</i> macro and the <i>longjmp</i> function shall not be used.	<ul style="list-style-type: none"> • The macro ‘<name> shall not be used. • Identifier XX should not be used. 	In case the longjmp function is actually a macro and the macro is expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.8	The signal handling facilities of <signal.h> shall not be used.	<ul style="list-style-type: none"> • The macro ‘<name> shall not be used. • Identifier XX should not be used. 	In case some of the signal functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.9	The input/output library <stdio.h> shall not be used in production code.	<ul style="list-style-type: none"> • The macro ‘<name> shall not be used. • Identifier XX should not be used. 	In case the input/output library functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.10	The library functions atof, atoi and toll from library <stdlib.h> shall not be used.	<ul style="list-style-type: none"> • The macro ‘<name> shall not be used. • Identifier XX should not be used. 	In case the atof, atoi and atoll functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
20.11	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	<ul style="list-style-type: none"> • The macro '<name>' shall not be used. • Identifier XX should not be used. 	In case the abort, exit, getenv and system functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.12	The time handling functions of library <time.h> shall not be used.	<ul style="list-style-type: none"> • The macro '<name>' shall not be used. • Identifier XX should not be used. 	In case the time handling functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated

runtime Failures

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
21.1	Minimization of runtime failures shall be ensured by the use of at least one of: <ul style="list-style-type: none"> • static verification tools/techniques; • dynamic verification tools/techniques; • explicit coding of checks to handle runtime faults. 		Done by PolySpace (runtime error checks). Cannot be Off.

Rules Partially Supported

In this section...
“Environment” on page 11-40
“Language Extension” on page 11-41
“Identifier” on page 11-42
“Declarations and Definitions” on page 11-42
“Expressions” on page 11-43
“Control Statement Expressions” on page 11-45
“Control Flow” on page 11-46
“Functions” on page 11-47
“Pointers and Arrays” on page 11-48
“Preprocessing Directives” on page 11-49

Environment

Rule	Description
1.1 (Required)	All code shall conform to ISO 9899:1990 “Programming languages - C”, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.

Messages in log:

- ANSI C does not allow `#include_next`
- ANSI C does not allow macros with variable arguments list
- ANSI C does not allow `#assert`
- ANSI C does not allow `#unassert`
- ANSI C does not allow testing assertions
- ANSI C does not allow `#ident`
- ANSI C does not allow `#sccs`

Rule	Description
	<ul style="list-style-type: none"> • text following '#else' violates ANSI standard. • text following '#endif' violates ANSI standard. • text following '#else' or '#endif' violates ANSI standard. • ANSI C90 forbids 'long long int' type. • ANSI C90 forbids 'long double' type. • ANSI C90 forbids long long integer constants. • Keyword 'inline' should not be used. • Array of zero size should not be used. • Integer constant does not fit within unsigned long int. • Integer constant does not fit within long int.
	<hr/> <p>Note All the supported extensions lead to a violation of this MISRA rule. Standard compilation error messages do not lead to a violation of this MISRA rule and remain unchanged. Can be turned to Off (see -misra2 option).</p> <hr/>

Language Extension

Rule	Description
2.1 (Required)	Assembly language shall be encapsulated and isolated.

Rule	Description
<p>Message in log:</p> <ul style="list-style-type: none"> • Assembly language shall be encapsulated and isolated. 	
<p>Note no warnings if code is encapsulated in asm functions or in asm pragma (only warning is given on asm statements even if it is encapsulated by a MACRO). Can be turned to Off.</p>	

Identifier

Rule	Description
<p>5.1 (Required)</p>	<p>Identifiers (internal and external) shall not rely on the significance of more than 31 characters</p>
<p>Message in log:</p> <ul style="list-style-type: none"> • Identifier 'XX' should not rely on the significance of more than 31 characters. 	
<p>Note Only global variables (external linkage) are checked. Can be turned to Off</p>	

Declarations and Definitions

Rule	Description
<p>8.3 (Required)</p>	<p>For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.</p>

Rule	Description
	<p>Message in log:</p> <ul style="list-style-type: none"> • Definition of function 'XX' incompatible with its declaration.
	<p>Note Assumes that rule 8.1 is not violated. The rule is restricted to compatible types. Can be turned to Off</p>
8.7 (Required)	Objects shall be defined at block scope if they are only accessed from within a single function
	<p>Message in log:</p> <ul style="list-style-type: none"> • Object 'XX' should be declared at block scope.
	<p>Note Restricted to static objects. Can be turned to Off</p>
8.8 (Required)	An external object or function shall be declared in one file and only one file
	<p>Message in log:</p> <ul style="list-style-type: none"> • Function/Object 'XX' has external declarations in multiples files.
	<p>Note Restricted to explicit extern declarations (tentative of definitions are ignored). Can be turned to Off</p>

Expressions

Rule	Description
12.2 (Required)	The value of an expression shall be the same under any order of evaluation that the standard permits.

Rule	Description
	<p>Messages in log:</p> <ul style="list-style-type: none"> • The value of ‘sym’ depends on the order of evaluation. • The value of volatile ‘sym’ depends on the order of evaluation because of multiple accesses.
	<p>Note The expression is a simple expression of symbols (Unlike <code>i = i++;</code>; no detection on <code>tab[2] = tab[2]++;</code>). Rule 12.2 check assumes that no assignment in expressions that yield a Boolean values (rule 13.1) and the comma operator is not used (rule 12.10). Can be turned to Off.</p>
<p>12.11 (Advisory)</p>	<p>Evaluation of constant unsigned expression should not lead to wraparound.</p>
	<p>No message.</p> <p>Note This rule is partially implemented with the <code>-detect-unsigned-overflows</code> option in PolySpace software. Concerning possible preprocessing overflows, PolySpace preprocessor does not take into account target basic types and considers always 32-Bit long int. Cannot be ticked.</p>
<p>12.12 (Required)</p>	<p>The underlying bit representations of floating-point values shall not be used.</p>
	<p>Message in log:</p> <ul style="list-style-type: none"> • The underlying bit representations of floating-point values shall not be used.
	<p>Note Warning on casts with float pointers (excepted with <code>void *</code>). Can be turned to Off.</p>

Control Statement Expressions

Rule	Description
13.3 (Required)	Floating-point expressions shall not be tested for equality or inequality.
Message in log:	
<ul style="list-style-type: none"> Floating-point expressions shall not be tested for equality or inequality. 	
<hr/> <p>Note Warning on directs tests only. Can be turned to Off.</p> <hr/>	
13.4 (Required)	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type
Message in log:	
<ul style="list-style-type: none"> The controlling expression of a <i>for</i> statement shall not contain any objects of floating type 	
<hr/> <p>Note If <i>for</i> index is a variable symbol, checked that it is not a float. Can be turned to Off.</p> <hr/>	
13.5 (Required)	The three expressions of a <i>for</i> statement shall be concerned only with loop control
Messages in log:	
<ul style="list-style-type: none"> 1st expression should be an assignment. Bad type for loop counter (XX). 2nd expression should be a comparison. 2nd expression should be a comparison with loop counter (XX). 3rd expression should be an assignment of loop counter (XX). 3rd expression: assigned variable should be the loop counter (XX). 	

Rule	Description
<p>Note Checked if the for loop index (V) is a variable symbol; checked if V is the last assigned variable in the first expression (if present). Checked if, in first expression, if present, is assignment of V; checked if in 2nd expression, if present, must be a comparison of V; Checked if in 3rd expression, if present, must be an assignment of V. Can be turned to Off.</p>	
<p>13.6 (Required)</p>	<p>Numeric variables being used within a <i>for</i> loop for iteration counting should not be modified in the body of the loop.</p>
<p>Message in log:</p> <ul style="list-style-type: none"> • Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop. 	
<p>Note Detect only direct assignments if the for loop index is known and if it is a variable symbol. Can be turned to Off.</p>	

Control Flow

Rule	Description
<p>14.3 (Required)</p>	<p>All non-null statements shall either</p> <ul style="list-style-type: none"> • have at least one side effect however executed, or • cause control flow to change

Rule	Description
	<p>Message in log:</p> <ul style="list-style-type: none"> • A null statement shall appear on a line by itself
	<p>Note We assume that a ';' is a null statement when it is the first character on a line (excluding comments). The rule is violated when:</p> <ul style="list-style-type: none"> • there are some comments before it on the same line. • there is a comment immediately after it • there is something else than a comment after the ';' on the same line. <p>Can be turned to Off.</p>

Functions

Rule	Description
16.4 (Required)	The identifiers used in the declaration and definition of a function shall be identical.
	<p>Message in log:</p> <ul style="list-style-type: none"> • The identifiers used in the declaration and definition of a function shall be identical.
	<p>Note Assumes that rules 8.8, 8.1 and 16.3 are not violated. Can be turned to Off.</p>
16.6 (Required)	The number of arguments passed to a function shall match the number of parameters.

Rule	Description
	<p>Messages in log:</p> <ul style="list-style-type: none"> • Too many arguments to XX. • Insufficient number of arguments to XX.
<hr/> <p>Note Assumes that rule 8.1 is not violated. Can be turned to Off.</p> <hr/>	

Pointers and Arrays

Rule	Description
17.4 (Required)	Array indexing shall be the only allowed form of pointer arithmetic.
	<p>Message in log:</p> <ul style="list-style-type: none"> • Array indexing shall be the only allowed form of pointer arithmetic.
<hr/> <p>Note Warning on operations on pointers. (p+I, I+p and p-I, where p is a pointer and I an integer). Can be turned to Off.</p> <hr/>	
17.6 (Required)	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
	<p>Message in log:</p> <ul style="list-style-type: none"> • Pointer to a parameter is an illegal return value. Pointer to a local is an illegal return value.
<hr/> <p>Note Warning when returning a local variable address or a parameter address. Can be turned to Off.</p> <hr/>	

Preprocessing Directives

Rule	Description
19.4 (Required)	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.
<p>Message in log:</p> <ul style="list-style-type: none"> • Macro '<name>' does not expand to a compliant construct. <hr/> <p>Note We assume that a macro definition does not violate this rule when it expands to:</p> <ul style="list-style-type: none"> • a braced construct (not necessarily an initializer) • a parenthesized construct (not necessarily an expression) • a number • a character constant • a string constant (can be the result of the concatenation of string field arguments and literal strings) • the following keywords: typedef, extern, static, auto, register, const, volatile, __asm__ and __inline__ • a do-while-zero construct <p>Can be turned to Off.</p>	
19.15 (Required)	Precautions shall be taken in order to prevent the contents of a header file being included twice.

Rule	Description
	<p>Message in log:</p> <ul style="list-style-type: none">• Precautions shall be taken in order to prevent multiple inclusions.
	<p>Note When a header file is formatted as follows:</p> <pre data-bbox="427 499 771 624">#ifndef <control macro> #define <control macro> <contents> #endif</pre> <p>It is assumed that precautions have been taken to prevent multiple inclusions. Otherwise, a violation of this MISRA rule is detected.</p> <p>Can be turned to Off.</p>

Rules Not Checked

In this section...
“Environment” on page 11-51
“Language Extensions” on page 11-52
“Documentation” on page 11-52
“Types” on page 11-53
“Functions” on page 11-54
“Pointers and Arrays” on page 11-54
“Structures and Unions” on page 11-55
“Standard Libraries” on page 11-55

Environment

Rule	Description	Comments
1.2 (Required)	No reliance shall be placed on undefined or unspecified behavior	Not statically checkable unless the data dynamic properties is taken into account
1.3 (Required)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the language/compiler/assemblers conform.	It is a process rule method.

Rule	Description	Comments
1.4 (Required)	The compiler/linker/Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	The documentation of compiler must be checked.
1.5 (Advisory)	Floating point implementations should comply with a defined floating point standard.	The documentation of compiler must be checked as this implementation is done by the compiler

Language Extensions

Rule	Description	Comments
2.4 (Advisory)	Sections of code should not be “commented out”	It might be some pseudo code or code that does not compile inside a comment.

Documentation

Rule	Description	Comments
3.1 (Required)	All usage of implementation-defined behavior shall be documented.	The documentation of compiler must be checked. Error detection is based on undefined behavior, according to choices made for implementation-defined constructions.

Rule	Description	Comments
		Documentation can not be checked.
3.2 (Required)	The character set and the corresponding encoding shall be documented.	The documentation of compiler must be checked.
3.3 (Advisory)	The implementation of integer division in the chosen compiler should be determined, documented and taken into account.	The documentation of compiler must be checked.
3.4 (Required)	All uses of the <i>#pragma</i> directive shall be documented and explained.	The documentation of compiler must be checked.
3.5 (Required)	The implementation-defined behavior and packing of bitfields shall be documented if being relied upon.	The documentation of compiler must be checked.
3.6 (Required)	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.	The documentation of compiler must be checked.

Types

Rule	Description	Comments
6.2 (Required)	Signed and unsigned char type shall be used only for the storage and use of numeric values.	Consider an external function returning a char is been used and increased. There is no mean without the functional

Rule	Description	Comments
	<hr/> Note this rule is partially implemented in Rule 6.1. <hr/>	knowledge that this function stores a character value or not.

Functions

Rule	Description	Comments
16.7 (Advisory)	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.	Not statically checkable unless the pointer verification has been done.
16.10 (Required)	If a function returns error information, then that error information shall be tested.	Not statically checkable unless type defining error is standardized.

Pointers and Arrays

Rule	Description	Comments
17.1 (Required)	Pointer arithmetic shall only be applied to pointers that address an array or array element.	Not statically checkable unless the pointer verification has been done
17.2 (Required)	Pointer subtraction shall only be applied to pointers that address elements of the same array.	Not statically checkable unless the pointer verification has been done
17.3 (Required)	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	Not statically checkable unless the pointer verification has been done

Structures and Unions

Rule	Description	Comments
18.2 (Required)	An object shall not be assigned to an overlapping object.	Not statically checkable unless the data dynamic properties is taken into account
18.3 (Required)	An area of memory shall not be reused for unrelated purposes.	"purpose" is functional design issue.

Standard Libraries

Rule	Description	Comments
20.3 (Required)	The validity of values passed to library functions shall be checked.	Not statically checkable unless all library function are standardized

Code Verification for Eclipse IDE

- “Overview” on page 12-2
- “Using PolySpace Software Within Eclipse IDE” on page 12-3

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.

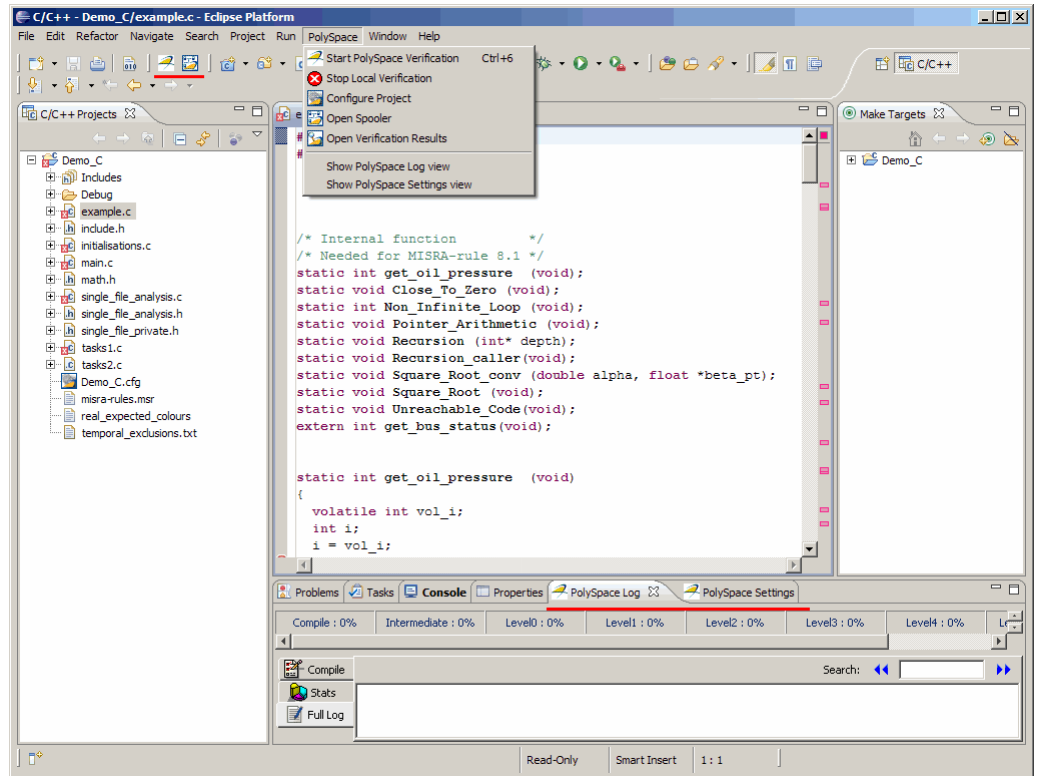
Using PolySpace Software Within Eclipse IDE

In this section...
“PolySpace Features in the Eclipse Editor” on page 12-3
“Verifying Files from Eclipse IDE” on page 12-5

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.

- **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_c`.
 - 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`

For information about these Target/Compilation options, see “Option Descriptions” 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 check."

Check

A test performed by PolySpace during a verification and subsequently colored red, orange, green or gray in the viewer.

Code verification

The PolySpace process through which code is tested to reveal definite and potential runtime errors and a set of results is generated for review.

Dead Code

Code which is inaccessible at execution time under all circumstances due to the logic of the software executed prior to it.

Development Process

The process used within a company to progress through the software development lifecycle.

Green check

Code has been proven to be free of runtime errors.

Gray check

Unreachable code; dead code.

Imprecision

Approximations are made during a PolySpace verification, so data values possible at execution time are represented by supersets including those values.

mcpu

Micro Controller/Processor Unit

Orange check

A warning that represents a possible error which may be revealed upon further investigation.

PolySpace Approach

The manner of use of PolySpace to achieve a particular goal, with reference to a collection of techniques and guiding principles.

Precision

An 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 check

Code has been proven to contain definite runtime errors (every execution will result in an error).

Review

Inspection of the results produced by a PolySpace verification.

Scaling option

Option applied when an application submitted to PolySpace proves to be bigger or more complex than is practical.

Selectivity

The ratio (green checks + gray checks + red checks) / (total amount of checks)

Unreachable code

Dead code.

Verification

The PolySpace process through which code is tested to reveal definite and potential runtime errors and a set of results is generated for review.

A

- active project
 - definition 10-3
 - setting 10-3
- analysis options 3-16
 - generic targets 3-27
 - MISRA C compliance 3-19 11-4
- ANSI compliance 6-3
- assistant mode
 - criterion 8-18
 - custom methodology 8-21
 - methodology 8-18
 - methodology for C 8-18 to 8-19
 - overview 8-17
 - reviewing checks 8-22
 - selection 8-17
 - use 8-17 8-22

C

- call graph 8-27
- call tree view 8-11
- calling sequence 8-27
- cfg. *See* configuration file
- client 1-6 6-2
 - installation 1-6
 - verification on 6-19
- Client
 - overview 1-6
- coding review progress view 8-11 8-28
- color-coding of verification results 1-3 8-13
- compile
 - log 7-6
- compile log
 - Launcher 6-21
 - Spooler 6-5
- compile phase 6-3
- compliance
 - ANSI 6-3
 - MISRA C 1-2 3-19 11-4

- composite filters 8-34
- configuration file
 - definition 3-2
- custom methodology
 - definition 8-21

D

- default directory
 - changing in preferences 3-6
- desktop file
 - definition 3-2
- directories
 - includes 3-10 3-13 3-15
 - results 3-10 3-13 3-15
 - sources 3-10 3-13 3-15
- downloading
 - results 8-8
- dsk. *See* desktop file

E

- expert mode
 - filters 8-33
 - composite 8-34
 - individual 8-33
 - overview 8-25
 - selection 8-25
 - use 8-25

F

- files
 - includes 3-10 3-13 3-15
 - results 3-10 3-13 3-15
 - source 3-10 3-13 3-15
- filters 8-33
 - alpha 8-34
 - beta 8-34
 - custom
 - modification 8-34 to 8-35

- use 8-34 to 8-35
- gamma 8-34
- individual 8-33
- user def 8-34

G

- generic target processors
 - adding 3-26
 - definition 3-27
 - deleting 3-30

H

- hardware requirements 7-2
- help
 - accessing 1-8

I

- installation
 - PolySpace Client for C/C++ 1-6
 - PolySpace products 1-6
 - PolySpace Server for C/C++ 1-6

L

- Launcher
 - monitoring verification progress 6-21
 - opening 3-3
 - starting verification on client 6-19
 - starting verification on server 6-3
 - viewing logs 6-21
 - window 3-3
 - overview 3-3
 - progress bar 6-21
- licenses
 - obtaining 1-6
- logs
 - compile
 - Launcher 6-21

- Spooler 6-5
- full
 - Launcher 6-21
 - Spooler 6-5
- stats
 - Launcher 6-21
 - Spooler 6-5
- viewing
 - Launcher 6-21
 - Spooler 6-5

M

- methodology for C 8-18 to 8-19
- MISRA C compliance 1-2
 - analysis option 3-19 11-4
 - checking 3-19 11-4
 - file exclusion 3-23 11-7
 - log 11-11
 - rules file 3-20 11-5

P

- PolySpace Client
 - overview 1-6
- PolySpace Client for C/C++
 - installation 1-6
 - license 1-6
- PolySpace In One Click
 - active project 10-3
 - overview 10-2
 - sending files to PolySpace software 10-5
 - starting verification 10-5
 - use 10-2
- PolySpace products for 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 3-26
 - definition 3-26
 - use 3-25
- 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 3-6
 - default server mode 6-3
 - generic targets 3-26
 - server detection 7-3
 - Viewer
 - assistant configuration 8-19
 - display columns in RTE view 8-30
- procedural entities view 8-11
 - reviewed column 8-30
- product overview 1-2
- progress bar
 - Launcher window 6-21
- project
 - creation 3-2
 - definition 3-2
 - directories
 - includes 3-3
 - results 3-3
 - sources 3-3
 - file types
 - configuration file 3-2
 - desktop file 3-2
 - PolySpace project model file 3-2
 - saving 3-17
- 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 8-37
- results
 - directory 3-10 3-13 3-15
 - downloading from server 8-8
 - opening 8-11
 - report generation 8-37
- reviewed column 8-30
- rte view. *See* procedural entities view

S

- selected check view 8-11
- server 1-6 6-2
 - detection 7-3
 - information in preferences 7-3
 - installation 1-6 7-3
 - verification on 6-3

Server

- overview 1-6
- source code view 8-11

Spooler

- monitoring verification progress 6-5
- removing verification from queue 8-8
- use 6-5
- viewing log 6-5

T

- troubleshooting failed verification 7-2

V

variables view 8-11

verification

 Ada code 1-7

 C code 1-2

 C++ code 1-7

 client 6-2

 compile phase 6-3

 failed 7-2

 monitoring progress

 Launcher 6-21

 Spooler 6-5

 phases 6-3

 results

 color-coding 1-3

 opening 8-11

 report generation 8-37

 reviewing 8-8

 running 6-2

 running on client 6-19

 running on server 6-3

 starting

 from Launcher 6-2 to 6-3 6-19

 from PolySpace In One Click 6-2 10-5

 stopping 6-22

 troubleshooting 7-2

 with MISRA C checking 11-10

Viewer

 modes

 selection 8-15

 opening 8-11

 window

 call tree view 8-11

 coding review progress view 8-11

 overview 8-11

 procedural entities view 8-11

 selected check view 8-11

 source code view 8-11

 variables view 8-11