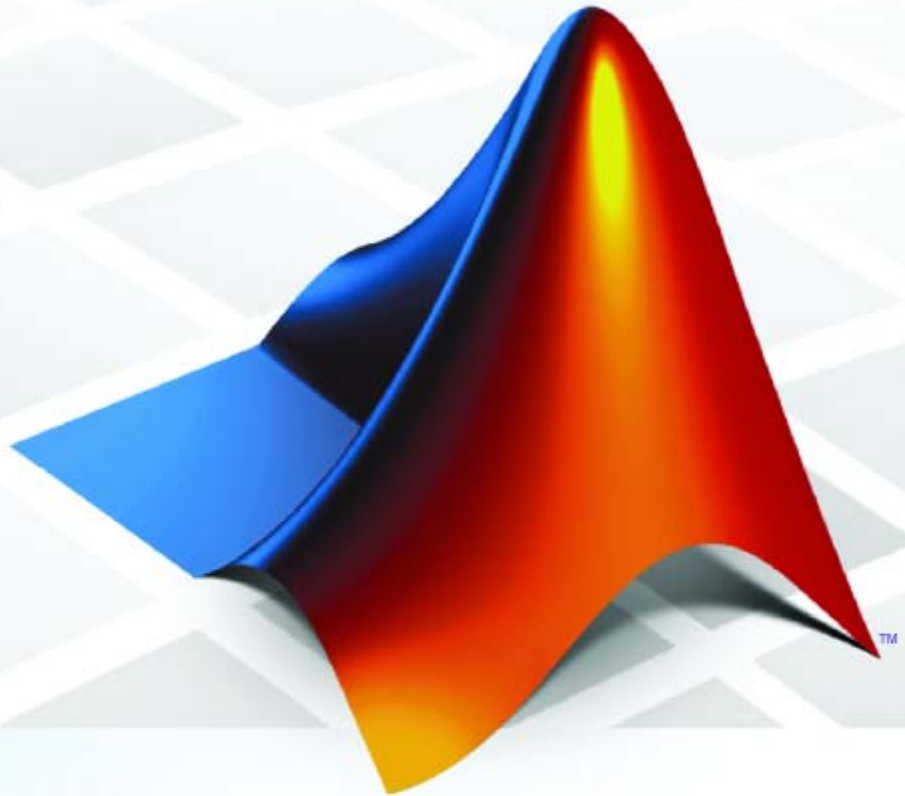


PolySpace™ Client/Server for C 5

User's Guide



How to Contact The MathWorks



www.mathworks.com
comp.soft-sys.matlab
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
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™ Client/Server for C User's Guide

© COPYRIGHT 1999–2008 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)

PolySpace™ Documentation Set

1

About this Guide	1-2
How to Use this Guide	1-3
Analyzing One File	1-3
Analyzing Code Generated from Simulink® Models	1-4
Analyzing Multiple Files	1-4
Detailed Contents	1-4

Getting Started

2

General Requirements	2-2
Computer Configuration	2-2
Timing Information	2-2
Installation Guide	2-2
Structure of this Document	2-3
PolySpace™ Client — Analyzing a Single C File	2-5
Overview	2-5
Analysis Prerequisites	2-5
Setting Up a PolySpace™ Client™ for C/C++ Analysis	2-6
PolySpace™ Client™ for C/C++: Running the Analysis ...	2-12
PolySpace™ Viewer — Exploring Results	2-20
Overview	2-20
Modes of Operation	2-20
Download Results into the Viewer	2-22
Reviewing PolySpace™ Results in “Expert” Mode (“example.c”)	2-24
Methodological Assistant	2-38
Report Generation	2-45

Setting Up and Launching the MISRA C® Checker	2-50
Before You Begin	2-50
Selecting MISRA C® Rules to Check	2-52
Running the MISRA® Checker	2-59
Launching PolySpace™ Analysis Remotely	2-62
Overview	2-62
Launching an Analysis	2-62
Management of PolySpace™ Analysis in Remote: the PolySpace™ Spooler	2-64
Batch Commands	2-67
Sharing Analyses Between Accounts	2-69
Summary	2-71

Analysis Setup

3

Compile Errors	3-2
Overview	3-2
Messages	3-2
Compiling Operating System Dependent Code (OS-target issues)	3-6
Target Specific Issues	3-9
Assembly Code	3-23
Dealing with Backward “goto” Statements	3-29
Link Messages	3-32
Overview	3-32
Function: Wrong Argument Type	3-32
Function: Wrong Argument Number	3-33
Variable: Wrong Type	3-34
Variable: Signed/Unsigned	3-34
Variable: Different Qualifier	3-35
Variable: Array Against Variable	3-36
Variable: Wrong Array Size	3-36
Missing Required Prototype for varargs	3-37
Can an Application without “main” be Analyzed? (For non Client mode only)	3-38

Stubbing Errors	3-40
Errors when Compiling _polyspace_stdstubs.c	3-40
Errors when Creating Automatic Stubs	3-45
How to Gather Compilations Options Efficiently	3-47
Stubbing	3-48
Intermediate Language Errors	3-57
Advanced Setup	3-59
Variables — Declaration and Definition	3-59
Types Promotion	3-60
Code Preparation for Variables	3-63
Code Preparation for Built-in Functions	3-69
My Code is Multitasking	3-69

PolySpace™ Software Day to Day Usage

4

PolySpace™ In One Click Overview	4-2
Using PolySpace™ In One Click	4-3
Overview	4-3
Creating an Active Configuration File Project	4-3
Using the TaskBar Icon	4-3
Using Right-Click to Launch PolySpace™	
Verification	4-6

MISRA® Checker

5

PolySpace™ MISRA® Checker Overview	5-2
Rules Supported	5-4
Language Extensions	5-5

Character Sets	5-5
Identifiers	5-6
Types	5-7
Constants	5-8
Declarations and Definitions	5-9
Initialization	5-11
Arithmetic Type Conversion	5-12
Pointer Type Conversion	5-16
Expressions	5-17
Control Statement Expressions	5-20
Control Flow	5-21
Switch Statements	5-23
Functions	5-24
Pointers and Arrays	5-25
Structures and Unions	5-25
Preprocessing Directives	5-26
Standard Libraries	5-30
Run-Time Failures	5-32
Rules Partially Supported	5-33
Environment	5-33
Language Extension	5-35
Identifier	5-35
Declarations and Definitions	5-36
Expressions	5-37
Control Statement Expressions	5-38
Control Flow	5-40
Switch Statements	5-40
Functions	5-41
Pointers and Arrays	5-42
Preprocessing Directives	5-43
Rules Not Checked	5-45
Environment	5-45
Language Extensions	5-46
Documentation	5-47
Types	5-48
Functions	5-48
Pointers and Arrays	5-48
Structures and Unions	5-49
Standard Libraries	5-49

Data Range Specifications

6

Overview	6-2
File Format	6-3
Variable Scope	6-5
Reduce Oranges with DRS	6-7
Perform Efficient Module Testing	6-7
Reduce Oranges with the —data-range-specification option	6-8

Using PolySpace™ Model Link Products

7

Overview of PolySpace™ Model Link Products	7-2
Getting Started	7-3
Overview	7-3
Creating a Simulink® Model and Generating Production Code	7-3
Starting the PolySpace™ Analysis	7-9
Fixing an Error in the Design and the Simulink® Model ..	7-13
Base Workspace vs. PolySpace™ Data Ranges	7-18
Advanced Setup	7-26
Hand-written Code	7-26
Target Production Environment	7-28
Creating a PolySpace™ Configuration File Template	7-30
Using the PolySpace™ Blocks Available in the Simulink® Library	7-33
PolySpace™ Utilities	7-35
Overview of PolySpace™ Utilities	7-35
Open PolySpace™ Results	7-36

PolySpace Enable COM Server	7-36
PolySpace™ Menu	7-37
PolySpace™ Project Configuration	7-38
Archives Files Produced for the PolySpace™ Analysis	7-39
PolySpace™ Commands Available in Batch Mode as M-Functions	7-41
Code Generator Specific Information	7-43
PolySpace™ Model Link™ SL Product	7-43
PolySpace™ Model Link™ TL Product	7-44

Results Review

8

Basics: Prerequisites to Reviewing PolySpace™	
Results	8-2
Overview	8-2
Grey Follows Red	8-3
What is the Message and What does it Mean?	8-4
What is the C Explanation	8-5
Specific Check Analysis	8-7
Colored Source Code for C	8-14
Illegal Pointer Access to Variable or Structure Field:	
IDP	8-15
Array Conversion Must Not Extend Range: COR	8-16
Array Index Within Bounds: OBAI	8-17
Initialized Return Value: IRV	8-18
Non-Initialized Variables: NIV/NIVL	8-19
Non-Initialized Pointer: NIP	8-20
Power Arithmetic: POW	8-20
User Assertion: ASRT	8-21
Scalar and Float Underflows: UNFL	8-23
Scalar and Float Overflows: OVFL	8-23
Float Underflows and Overflows: UOVFL	8-24
Scalar or Float Division by Zero: ZDV	8-28
Shift Amount in 0..31 (0..63):SHF	8-28
Left Operand or Left Shift is Negative: SHF	8-29
Function Pointer Must Point to a Valid Function: COR ...	8-30
Wrong Type for Argument: COR	8-31

Wrong Number of Arguments: COR	8-32
Wrong Return Type of a Function Pointer: COR	8-33
Wrong Return Type for Arithmetic Functions: COR	8-33
Pointer Within Bounds: IDP	8-34
Non Termination of Call or Loop	8-49
Unreachable Code: UNR	8-58
Value on Assignment: VOA	8-60
Inspection Points: IPT	8-62

PolySpace™ Methodological Guide

9

Overview	9-2
PolySpace™ Usage	9-5
Overview of the PolySpace™ Approach	9-5
Standard Development Process	9-10
Rigorous Development Process: Introducing Tools and Coding Rules	9-14
A Quality/Qualification Approach	9-17
Code Acceptance Criterion	9-18
PolySpace™ Activities	9-20
Review Run Time Errors: Fix Red Errors	9-20
Review Dead Code Checks: Why is Grey Code Interesting	9-21
How to Find a Maximum Number of Bugs Within an Hour Reviewing Oranges: Selective Orange Review	9-23
Cost and Benefits of an Exhaustive Orange Review at Integration Phase	9-27
Integration Bug Tracking	9-29
How to Find Bugs in Unprotected Shared Data	9-30
Dataflow Analysis	9-31
Data and Coding Rules	9-31
Automatically Testing Orange Code	9-33
PolySpace™ Automatic Orange Tester	9-33
Using the Automatic Orange Tester	9-35
Technical Limitations	9-52

How to Get the Best Results	9-56
Reduce Oranges Step by Step	9-56
Generic Objectives: A Balance Between Precision and Analysis Time	9-56
Options at Launching Time	9-58
How to Conclude an Orange Review	9-64
Duration of Analysis	9-68
Applying Coding Rules to Reduce Oranges	9-87
MISRA® Rules Which PolySpace™ Verification Can Help to Follow	9-87
Recommended Set of Coding Rules	9-87
Approximations Made by PolySpace™ Verification	9-92

Options Description

10

General	10-2
Overview	10-2
-prog Session identifier	10-2
-date Date	10-3
-author Author	10-3
-verif-version Version	10-3
-voa	10-4
-keep-all-files	10-4
-continue-with-red-error	10-5
-continue-with-existing-host	10-5
-allow-unsupported-linux	10-5
-results-dir Results Directory	10-6
-sources "files" or -sources-list-file file_name	10-7
-I directory	10-8
Target/Compiler	10-10
Overview	10-10
-target TargetProcessorType	10-10
GENERIC ADVANCED TARGET OPTIONS	10-11
-OS-target OperatingSystemTargetForPolySpaceStubs ...	10-17
-D compiler-flag	10-17
-U compiler-flag	10-18
-include file_name	10-18

-post-preprocessing-command <file_name> or "command"	10-19
-post-analysis-command <file_name> or "command"	10-20
Compliance with Standards	10-22
-dos	10-22
Embedded Assembler	10-23
Strictness during analysis launching	10-24
Permissiveness during analysis launching	10-25
MISRA-C 2004 Rules	10-28
-dialect [iar keil]	10-30
PolySpace™ Inner Settings	10-32
MAIN GENERATOR OPTIONS (-main-generator) for PolySpace™ Software	10-32
Stubbing	10-35
Assumptions	10-37
Automatic Orange Tester	10-44
Others	10-45
Precision/Scaling	10-47
-quick	10-47
-O(0-3)	10-48
-modules-precision mod1:O(0-3)[,mod2:O(0-3)[,...]]	10-49
-from verification-phase	10-49
-to verification-phase	10-50
-context-sensitivity "proc1[,proc2[,...]]"	10-51
-context-sensitivity-auto	10-51
-path-sensitivity-delta number	10-52
-retype-pointer	10-52
-retype-int-pointer	10-53
-k-limiting number	10-54
-no-fold	10-55
-respect-types-in-globals	10-55
-respect-types-in-fields	10-56
-inline "proc1[,proc2[,...]]"	10-57
-lightweight-thread-model	10-57
MultiTasking (PolySpace™ Server™ for C/C++ Product Only)	10-59
-entry-points str1[,str2[,...]]	10-59
-critical-section-[begin or end] "proc1:cs1[,proc2:cs2]"	10-59

-temporal-exclusions-file file_name	10-60
Batch Options	10-62
-server server_name_or_ip[:port_number]	10-62
-sources-list-file file_name	10-62
-v -version	10-63
-h[elp]	10-63
Complete Examples	10-64
Simple C Example	10-64
Apache Example	10-64
cxref Example	10-65
T31 Example	10-65
Dishwasher1 Example	10-65
Satellite Example	10-66

Static Verification

A

What is Static Verification	A-2
Exhaustiveness	A-4

Glossary

PolySpace™ Documentation Set

About this Guide (p. 1-2)

Describes the purpose of this manual

How to Use this Guide (p. 1-3)

Describes which sections to read for specific information

About this Guide

This document represents all the documentation required to use PolySpace™ tools, irrespective of whether you are a beginner or an experienced user. It covers both PolySpace™ Client™ for C/C++ and PolySpace™ Server™ for C/C++ products.

How to Use this Guide

In this section...
“Analyzing One File” on page 1-3
“Analyzing Code Generated from Simulink® Models” on page 1-4
“Analyzing Multiple Files” on page 1-4
“Detailed Contents” on page 1-4

Analyzing One File

If you are looking to analyze one file:

- Do you want to perform your first analysis and results review?
- Do you want to launch an analysis with a right click?
- Are you applying coding rules?
 - Reduce the number of orange checks - step by step
 - Apply chosen coding rules
- Is it possible for you to restrict data (functional) ranges in the file?
 - Using the Data Range Specification feature (DRS)
 - By replacing automatic stubs of functions manually
- Do you have issues with setting up or launching an analysis?
- When reviewing results, is your main concern
 - Productivity? Do you wish to focus on productivity by finding bugs quickly?
 - Reliability? Do you want to examine every result PolySpace™ verification provides?
 - Or do you want to find a compromise between productivity and reliability?
- Does your analysis take place:

- On a developer's machine, using the PolySpace™ Client™ for C/C++ product?
- Spooled on a distant server, using the PolySpace™ Server™ for C/C++ product?

Analyzing Code Generated from Simulink® Models

Do you want to Analyze code generated from Simulink® models using the PolySpace Model Link™ SL or PolySpace Model Link TL products?

Analyzing Multiple Files

If you are looking to analyze multiple packages:

- Do you have issues related to:
 - Analysis launching (setup)?
 - Common setup issues
 - Advanced setup
 - Multitasking issues?
 - Shared variables?
- Do you want to find bugs efficiently in the results?
- Does your analysis takes place on a server, and do you want access the queued analysis?

Detailed Contents

- PolySpace Installation. Please refer to the *PolySpace Installation Guide* and *PolySpace License Installation Guide* located on the CD-ROM (in <CD-ROM>\Docs\Install).
- Chapter 2, “Getting Started” explains how to get started with PolySpace products. It explains the principles of the tool, describes the installation procedure, and explains how to use the product with reference to some simple scenarios.
- Chapter 3, “Analysis Setup” details the features of PolySpace software which are relevant when preparing to analyze your code. It is a

comprehensive reference manual for the launching of analyses. It contains all information related to the launching of an analysis, error messages at different phases of an analysis, and means at setup-time to reduce ill founded warnings (oranges).

- Chapter 4, “PolySpace™ Software Day to Day Usage” allows configuring a project and launches analysis using PolySpace “Tool Bar” and right click in the “Send To” menu (Only on Windows® systems).
- Chapter 5, “MISRA® Checker” details all the MISRA C® 2004 rules that help developers achieves MISRA® compliance.
- Chapter 6, “Data Range Specifications” describes the PolySpace DRS, an easy to use module that helps developers achieves external constraints on global variables without intrusion.
- Chapter 7, “Using PolySpace™ Model Link Products” describes how to use the PolySpace Model Link SL and PolySpace Model Link TL products to analyze code generated from Simulink models.
- Chapter 8, “Results Review” details all features of PolySpace software which are relevant when reviewing your results. It is a comprehensive reference document, giving typical examples for each error category, offering advice on getting started with your first results, advising which colors to look at, and explaining how to find bugs efficiently.
- Chapter 9, “PolySpace™ Methodological Guide” gives guidance in the use of PolySpace software as an integral part of the development process. It is presented as a narrative, and will help proficient users of the tool to get the best possible use from it. It presents different development processes, and shows how PolySpace software might best be integrated in each case.
- “Advanced Setup” on page 3-59 includes multitasking information for PolySpace Verifier, hints and tips for quicker PolySpace Verifier analyses, and a complete description of those features which are used in order to launch a PolySpace analysis.

Getting Started

General Requirements (p. 2-2)	Describes requirements to consider before beginning the tutorial
PolySpace™ Client — Analyzing a Single C File (p. 2-5)	Describes how to analyze a single C file using PolySpace™ Client™ for C/C++ product
PolySpace™ Viewer — Exploring Results (p. 2-20)	Describes how to interpret the results of your analysis
Setting Up and Launching the MISRA C® Checker (p. 2-50)	Describes how to use the MISRA C® Checker during an analysis
Launching PolySpace™ Analysis Remotely (p. 2-62)	Describes how to perform an analysis remotely using the PolySpace™ Server™ for C/C++ product
Summary (p. 2-71)	Provides a summary of the information presented in this guide.

General Requirements

In this section...
“Computer Configuration” on page 2-2
“Timing Information” on page 2-2
“Installation Guide” on page 2-2
“Structure of this Document” on page 2-3

Computer Configuration

Please refer to PolySpace™ Installation Guide for the minimum hardware requirements.

Timing Information

The installation of PolySpace products takes around 5 minutes (see the complete installation guide is available from the PolySpace installation CD-ROM in \Docs\Install*PolySpace_Install_Guide.pdf*).

- The first step of this tutorial takes about 15 minutes.
- The second step of this tutorial takes about 15 minutes.
- The third step of this tutorial takes about 5 minutes.
- The fourth step of this tutorial takes about 5 minutes.

Installation Guide

Note If the PolySpace products are already installed on your computer, please go directly to step 1.

The PolySpace products are delivered on a CD-ROM. There are 4 modules:

- 1 *PolySpace™ Client™ for C/C++* for analyzing single files. Note that this module is available with the icon “*PolySpace Launcher*”.

- 2 *PolySpace™ Server™ for C/C++* for multi-file or composite analysis. Note that this module is available with the icon “*PolySpace Launcher*”.
- 3 *PolySpace Viewer* is the graphical user interface to explore the results computed by PolySpace Server for C/C++ or PolySpace Client for C/C++.
- 4 *PolySpace Spooler* is the graphical interface to manage analysis done remotely.

Please refer to PolySpace Installation Guide for installing the PolySpace products.

Structure of this Document

Once the installation is done, you can launch PolySpace software by using the following icons that were placed on your desktop:



Moreover, inside PolySpace Client for C/C++ and PolySpace Server for C/C++, a PolySpace MISRA® Checker is available, allowing at compilation time to verify some of the rules recommended by the MISRA Consortium (more about MISRA Consortium at <http://www.misra-c.com>).

This Getting Started will focus on the following four exercises using the Client, the Viewer, the PolySpace MISRA Checker and the Server:

- In Step 1: “PolySpace™ Client — Analyzing a Single C File” on page 2-5, you will analyze a simple file “example.c” by using the PolySpace Client for C/C++ product.
- In Step 2: “PolySpace™ Viewer — Exploring Results” on page 2-20, you will review the results obtained during Chapter 2 by using PolySpace Viewer.
- In Step 3: “Setting Up and Launching the MISRA C® Checker” on page 2-50, you will use PolySpace MISRA Checker during the compilation phase of a PolySpace analysis.
- In Step 4: “Launching PolySpace™ Analysis Remotely” on page 2-62, you will send an analysis remotely to a PolySpace Server for C/C++ server.

PolySpace™ Client – Analyzing a Single C File

In this section...

“Overview” on page 2-5

“Analysis Prerequisites” on page 2-5

“Setting Up a PolySpace™ Client™ for C/C++ Analysis” on page 2-6

“PolySpace™ Client™ for C/C++: Running the Analysis” on page 2-12

Overview

This section describes a basic file analysis. It focuses on the analysis of “example.c”, which is included in the PolySpace™ installation directory and located at:

```
<PolySpaceInstallDir>\Examples\Demo_C\sources\example.c.
```

The PolySpace analysis process is composed of three main phases:

- 1** First, PolySpace software checks the syntax and semantic of the analyzed file(s). However, as PolySpace products are not associated to a particular compiler, **benefits** of this phase are triple for the analyzed source code: **ANSI® compliance, portability** and **maintainability**.
- 2** Then, the client seeks the main procedure. If none is found, PolySpace™ Client™ for C/C++ will generate one automatically. By default, this function will call all public functions of the file.
- 3** Finally, the client proceeds with the code analysis phase, during which run time errors are detected and highlighted in the code.

Analysis Prerequisites

Any analysis requires the following:

- PolySpace products and their related license files are correctly installed;
- Source code files (in this case “example.c”) and all header files that it may directly or indirectly include. For this tutorial we will see later that we need two header files “math.h” and “include.h” in order to analyze “example.c”.

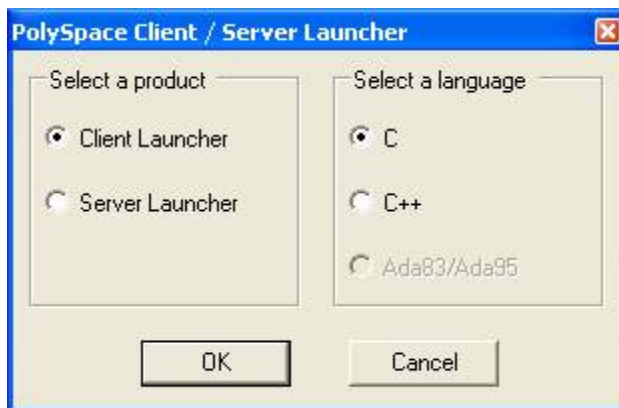
- All “-D” compilation switches necessary to compile the file are known. Please note that in this tutorial, no “-D” is necessary to compile “example.c”.

Setting Up a PolySpace™ Client™ for C/C++ Analysis

- 1 Double-click on the PolySpace Launcher icon (release number could not be same):

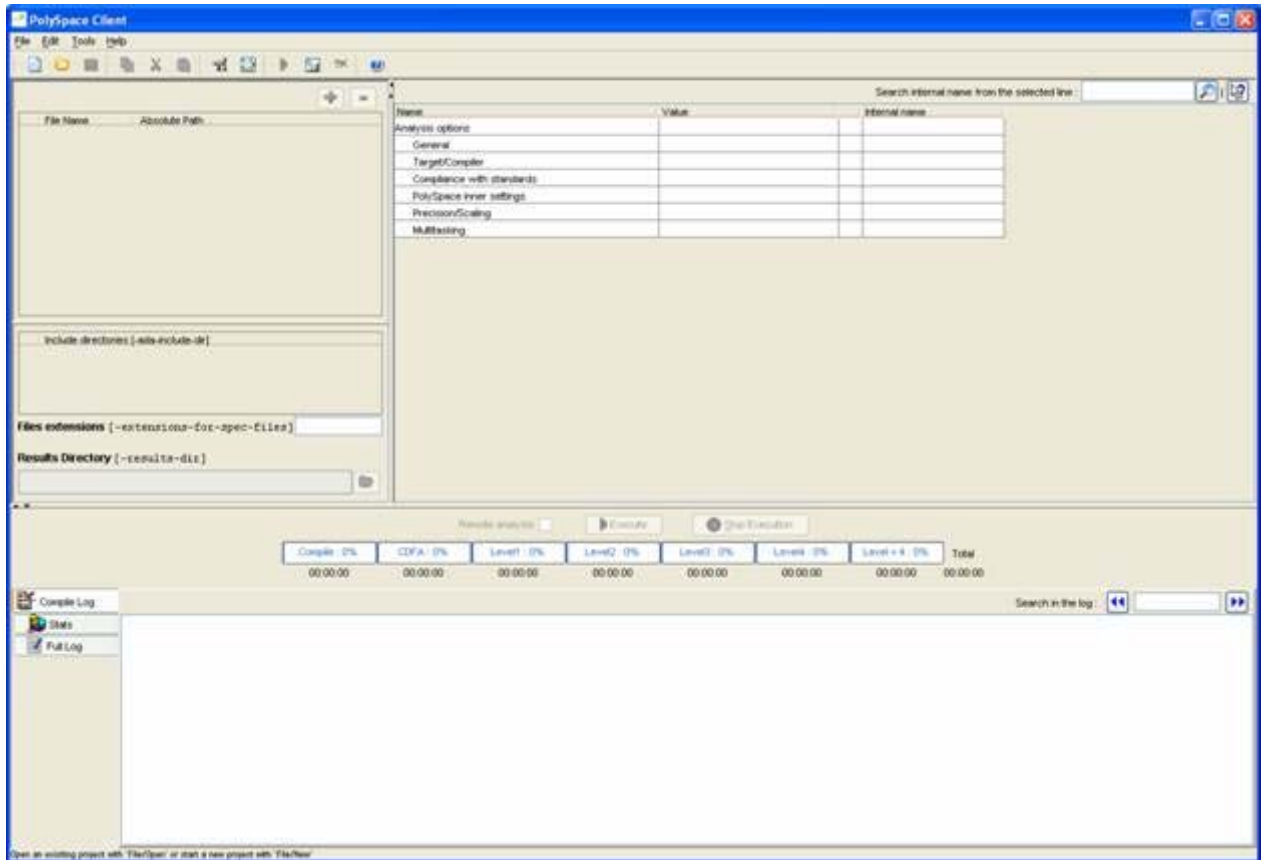


A dialog box window appears proposing to launch one of the following categories of analysis mixing the type of product and the language:

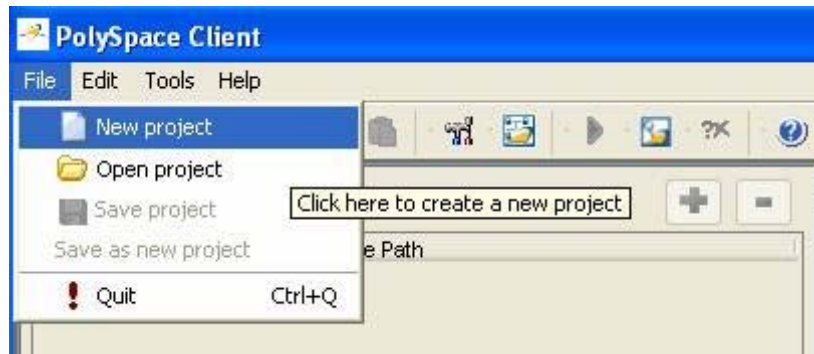


- 2 Select **Client Launcher**, and **C**, then click **OK**.

The Graphical Interface of PolySpace analysis Launcher is displayed as below:

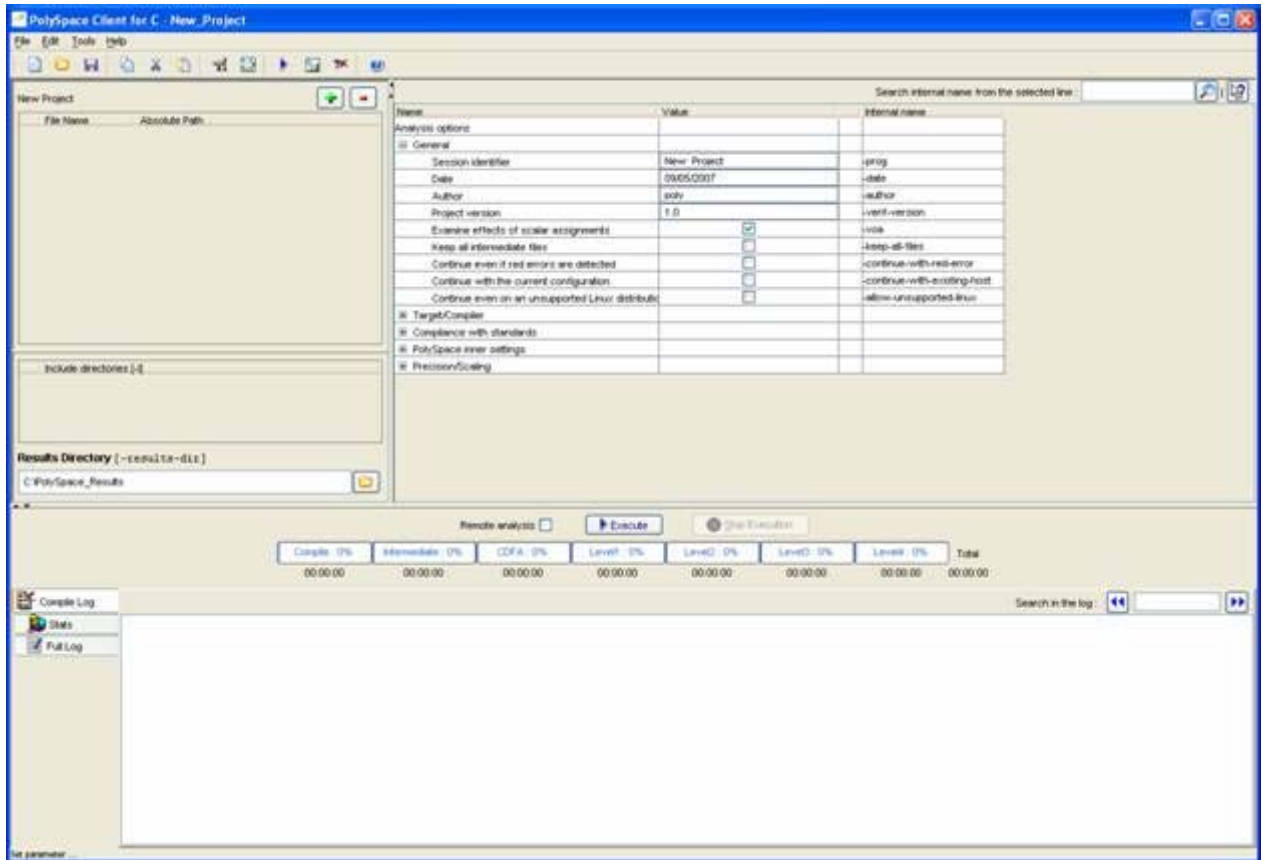



3 Click on File/New Project to start an analysis:

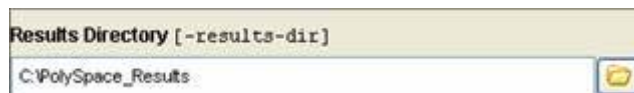


The PPolySpace Client for C/C++ New Project window opens (see figure below). It contains four sections:


- At the very top, the title bar, which contains usual icons and menus;
- Top left is the list of files to analyze, along with include and results directories;
- Top right is the set of options associated with the analysis that will be processed;
- Finally the bottom area allows following the execution and progress of the analysis.



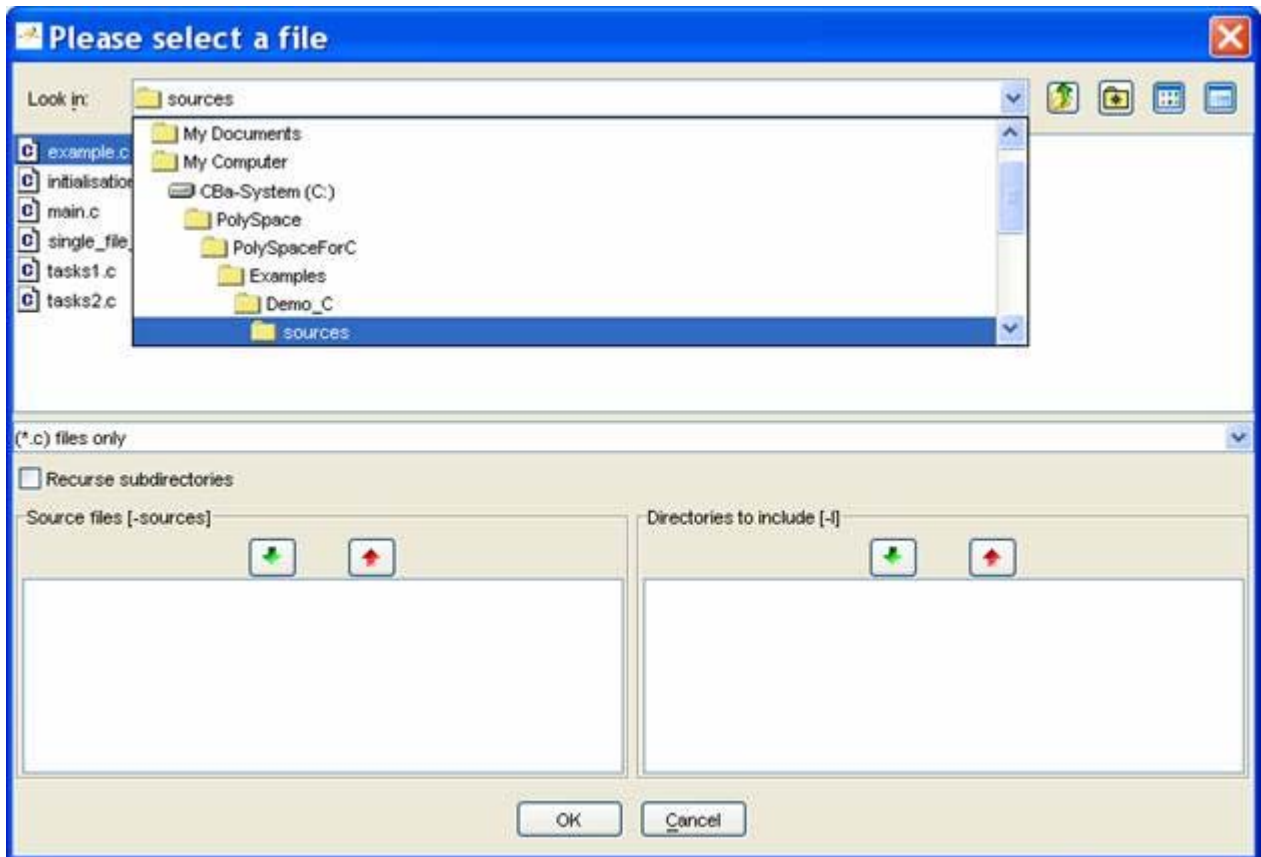
- 4 Start by updating the result directory name by clicking on the browse button .





This directory is the one where PolySpace Client for C/C++ product will store the results of the analysis. By default, the client will store results in “C:\PolySpace_Results”. This is the directory that we will choose for the analysis.

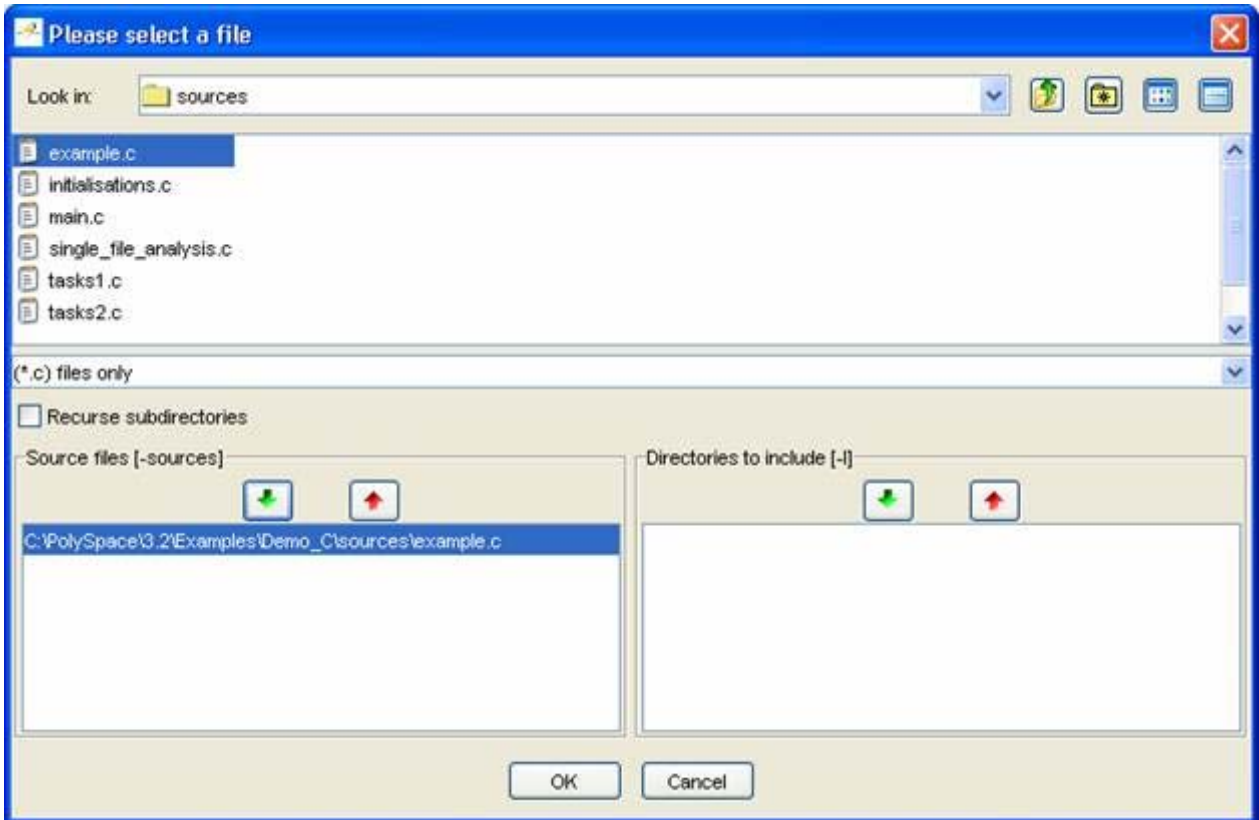
- 5 Now, Click on the  button (right of the “New Project” label).

It opens the “Please select a file” window, from which you can select one or several files to analyze.



- 6 In the “Look in” section, click on , and select “<PolySpaceInstallDir>\Examples\Demo_C\sources”. A list of files appears in the box (<PolySpaceInstallDir> corresponds to C:\PolySpace\PolySpaceForCandCPP in the figure above).

- 7 Select “example.c” and click on  in the “Source files [-sources]” section (bottom left) of the window. The file is now listed among the source files to be analyzed.




- 8 Click on OK to go back to the “PolySpace Client for C - New_Project” window.

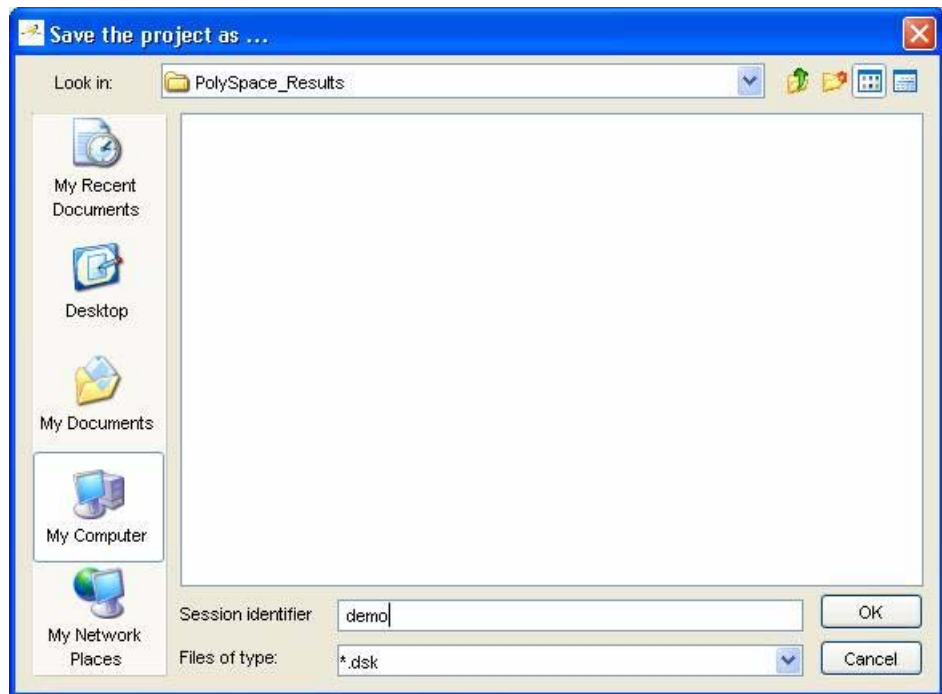
Note it is also possible to drag a directory or source files and drop it them directly in the “File Name/Absolute Path” part (top left of PolySpace Client) without using the “Please select a file” window.

PolySpace™ Client™ for C/C++: Running the Analysis

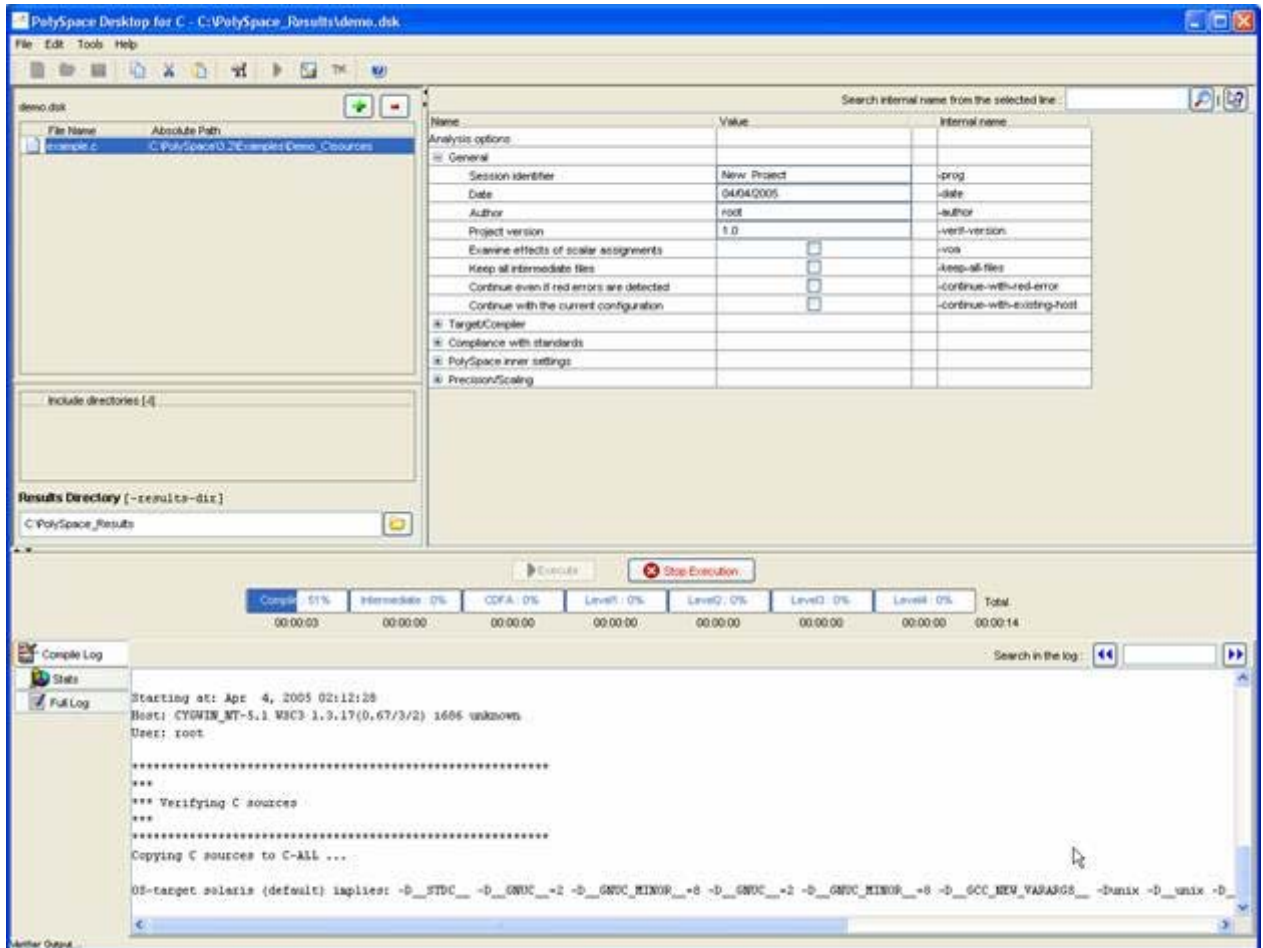
To run the analysis:


- 1 Click on  to start the analysis. Alternatively, you can click on the button in the title bar to run PolySpace Client for C/C++ analysis with the current setting.

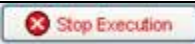
The window titled “Save the project as” opens. You can decide where to store the configuration information related to the analysis. Here, create a file called “demo” and save it under PolySpace result directory. The full name of that file will be “demo.dsk”.



- 2 Click on  to go back to the “PolySpace Desktop for C - New_Project” window and click again on  to proceed.



A progress report is displayed in the bottom part of the graphical interface, indicating that the analysis is being performed. The  button is also grayed out.

Note You may press the Stop Execution button -  to interrupt the analysis but it is not part of the current tutorial.

Parsing Errors During Preliminary Analysis Stages

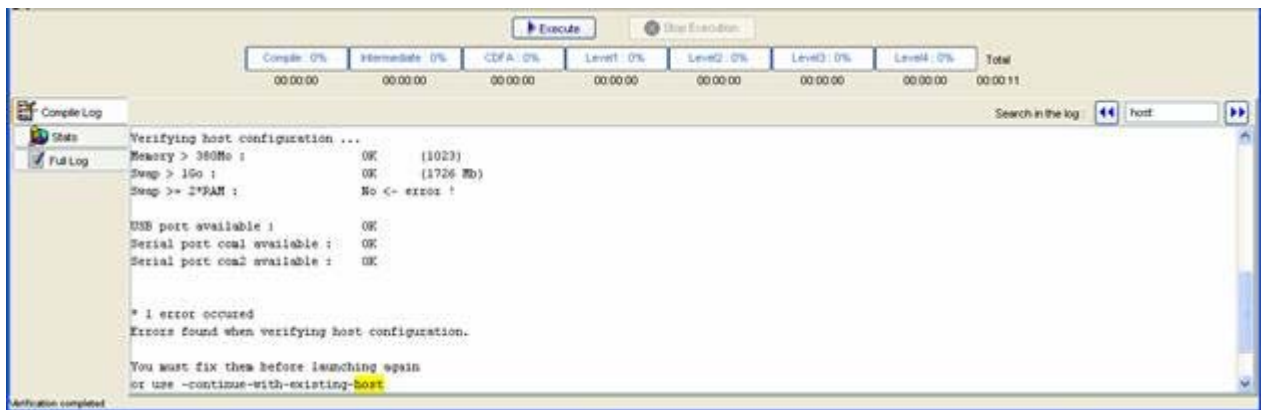
After some checks, the software will show an error message:




Lets try and understand why we get this error message.

First Possible Cause for the Error Message: Hardware Recommendation.

If this happens, please verify whether your computer fits the minimal hardware configuration requirements described in the general requirements. Moreover, a message like the following one is displayed in the bottom part of the graphical interface:



- 1 Type “host” in the “Search in the log:” box and click on  to search if the error corresponds to a hardware recommendation problem.

If the error message corresponds to the one shown above and in order to continue analysis, you can either:

- upgrade your computer to meet the minimal requirements, or

- use the `-continue-with-existing-host` option which overrides the initial check for minimal hardware configuration. To do so, please follow the following steps:
- 2 To set up the `-continue-with-existing-host` option, please type “continue” in the Search internal name from the selected line (top right box).

Search internal name from the selected line : continue 

- 3 Click .

It will show all options containing “continue” in the set of options part below:

Name	Value	Internal name
Analysis options		
[-] General		
Session identifier	New Project	-prog
Date	04/04/2005	-date
Author	root	-author
Project version	1.0	-verif-version
Examine effects of scalar assignments	<input type="checkbox"/>	-voa
Keep all intermediate files	<input type="checkbox"/>	-keep-all-files
Continue even if red errors are detected	<input type="checkbox"/>	-continue-with-red-error
Continue with the current configuration	<input type="checkbox"/>	-continue-with-existing-host
[+] Target/Compiler		
[+] Compliance with standards		
[+] PolySpace inner settings		
[+] Precision/Scaling		

- 4 Check the box

in the “Value” column that is associated to the “-continue-with-existing-host” line as shown below.

It is also recommended to select the -continue-with-red-error option. Indeed, “example.c” contains - on purpose - code with some definite errors, later called red errors. This option allows you to continue the analysis even if red errors are detected in previous passes.


Continue even if red errors are detected	<input checked="" type="checkbox"/>	-continue-with-red-error
Continue with the current configuration	<input checked="" type="checkbox"/>	-continue-with-existing-host

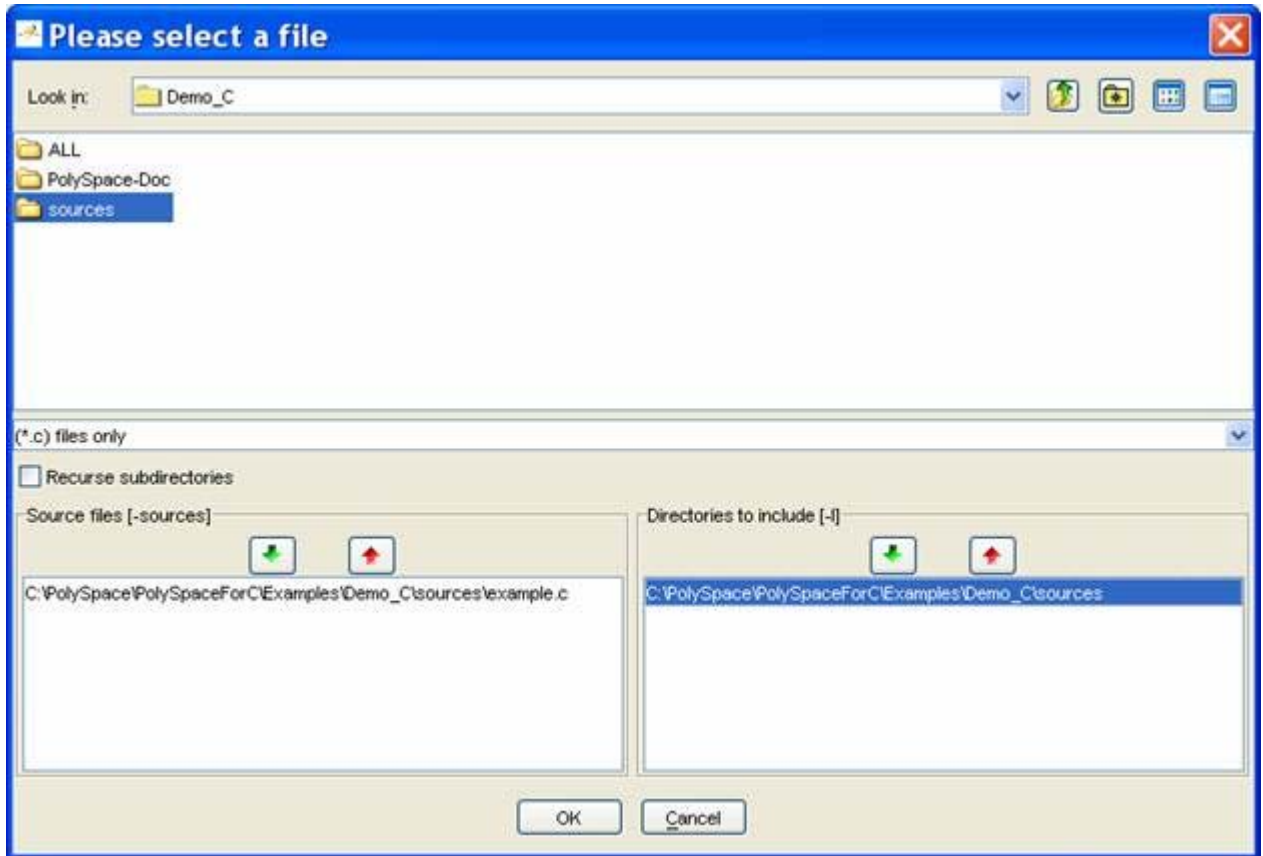
Second Possible Cause for the Error Message: Information About Header Files. Another cause of error may be that PolySpace Desktop misses some package specifications.




In the tutorial, as shown above, the file named “math.h” can not be found. To fix this problem, you need to indicate its location. As PolySpace products are not associated with one particular compiler, it is mandatory to indicate where library files are stored.

In our “example.c” file analysis, the related “math.h” file is located in the same directory as the C file: <PolySpaceInstallDir>\Examples\Demo_C.

- 1 Open the “Please select a file” window using the  button (right of the “demo.dsk” label in the top right of the interface):




- 2 Select “<PolySpaceInstallDir>\Examples\Demo_C\sources”, where “math.h” is located.
- 3 Click on  in the “Directories to include [-I]” section, then click **OK** to close the window.

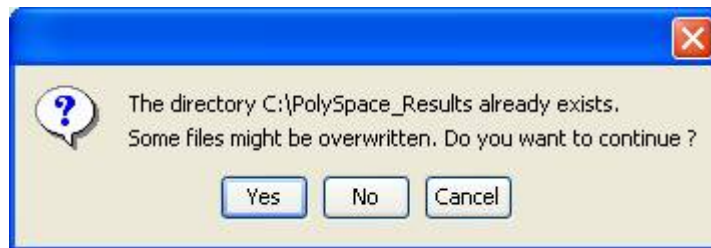
Note Other header file needed “include.h” is also located in same directory.

It is also possible to drag a directory and drop it directly in the “include directories [-I]” part (top left of PolySpace Client) without using the “Please select a file” window.


Progression of the Analysis

- 1 Click on  to restart the analysis. to restart the analysis.

If you previously clicked **Execute**, some results may have already been written in the “C:\PolySpace_Results” directory. Therefore a window opens to check whether you want to overwrite in this directory or not:






- 2 If this happens, click **Yes**.


Note Closing the PolySpace Desktop window will not stop the PolySpace analysis. If you wish to stop it, click  (a window of confirmation follows the click). If the window is closed without stopping the analysis, it continues in background. Opening again PolySpace Desktop with the same project automatically updates the analysis with its current status.

The progress bar allows to follow the progress of the analysis:

Compile : 100%	Intermediate : 71%	CDFA : 0%	Level1 : 0%	Level2 : 0%	Level3 : 0%	Level4 : 0%	Total
00:00:40	00:00:16	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:01:04

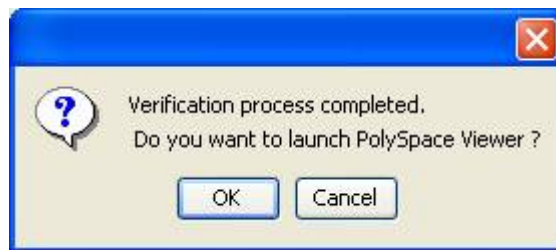
3 To obtain a progress report, click on  **Compile Log** for the compilation phase, or  **Full Log** for the full analysis in the low level window.

4 Click  **Stats** to get other pieces of information about current analysis (list of options, stubbed functions, functions used during main construction, checks found after each phase, etc.).


5 Click the  icon to refresh the summary.

End of the Analysis

When the analysis ends, the software proposes to review the results:



If you Click **OK**, go to the next section of the tutorial to view the results.

If you click **Cancel**, and no other analyses are running, you can access the results via the  icon in the title bar. , and if no other analyses are running, you can access the results via the icon in title bar.

PolySpace™ Viewer – Exploring Results

In this section...

“Overview” on page 2-20

“Modes of Operation” on page 2-20

“Download Results into the Viewer” on page 2-22

“Reviewing PolySpace™ Results in “Expert” Mode (“example.c”)” on page 2-24

“Methodological Assistant” on page 2-38

“Report Generation” on page 2-45

Overview

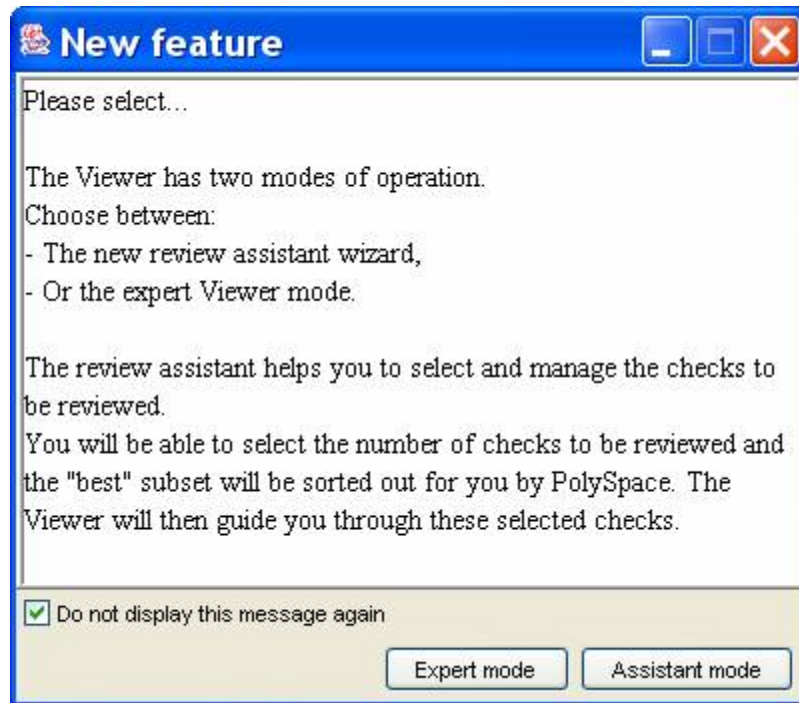
This section illustrates how to explore analysis results that were generated by either PolySpace™ Client™ for C/C++ or PolySpace™ Server™ for C/C++ products. We review the results of the analysis of “example.c” performed during step 1: “PolySpace™ Client — Analyzing a Single C File” on page 2-5.



If you clicked **OK** at the end of the previous analysis (see previous section), PolySpace™ Viewer automatically opens results.

Modes of Operation

The first time The PolySpace Viewer is opened, a sub-window will appear after the splash screen of the viewer. It is aimed to warn user about different modes of operation. User has to choose between launching the Viewer in an “expert” mode or in an “assistant” mode.



The mode will define the reviewing process of checks highlighted during an analysis:

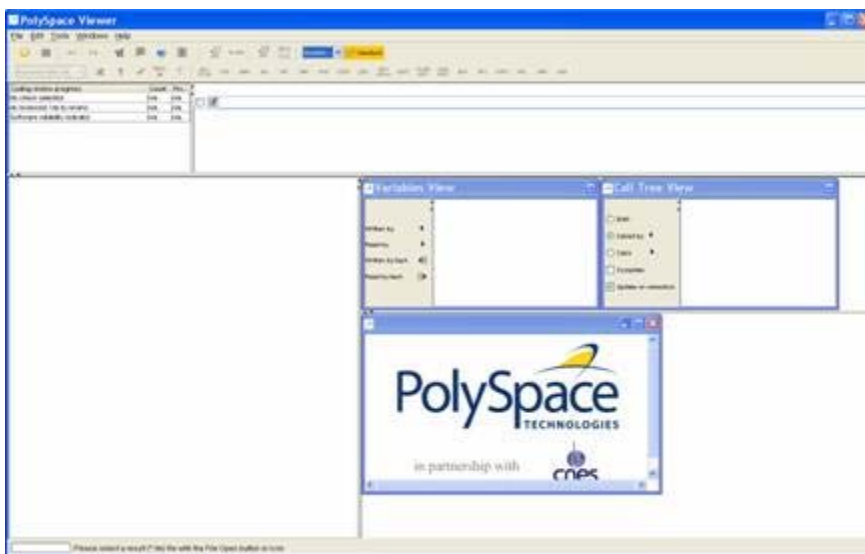
- **Expert mode** — The Viewer is opened in a mode where all checks can be seen. The number, the order and the categories of checks to be reviewed have to be selected by the user himself (See next section).
- **Assistant mode** — The reviewing rules for a C analysis results follows a methodology selected by PolySpace software. It concerns the “best” subset of checks sorted out for user. The PolySpace Viewer will then guide user through these selected checks.

For the need of this tutorial, please untick “Do not display this message again” and then click on “Expert mode”.

Note Even if the user has chosen one mode it is easy in one click to change the mode inside the PolySpace Viewer.

Download Results into the Viewer

After having clicked on “Expert mode” the PolySpace Viewer window looks like the figure below:

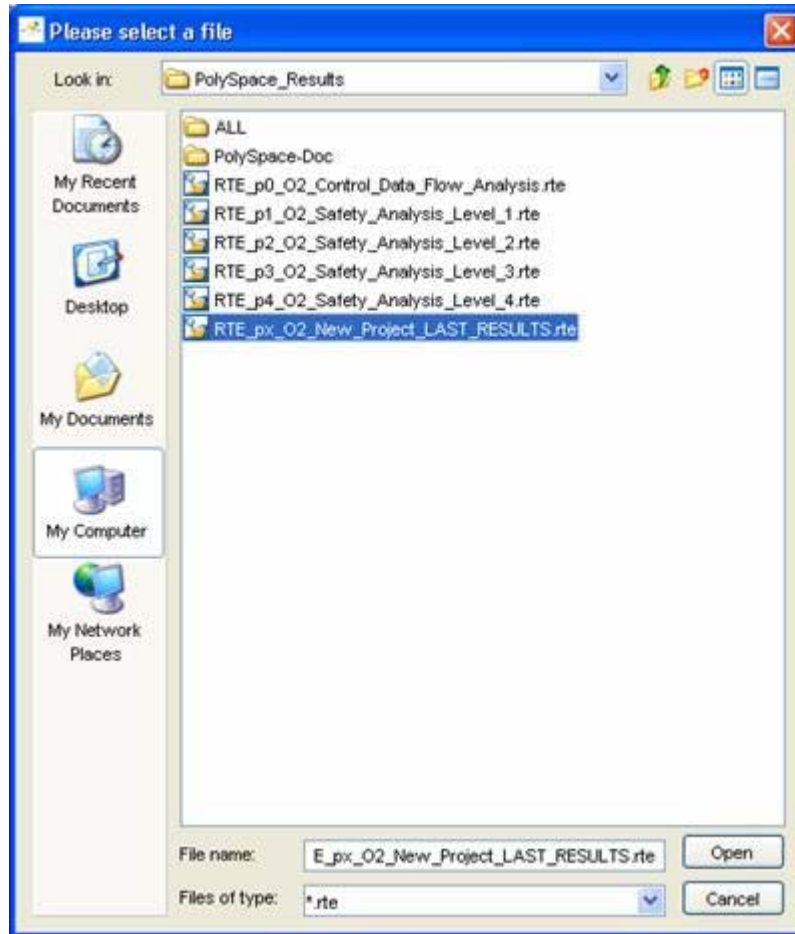


1 Click **File > Open** to load result files.

Note If you did not perform the analysis, you can still review the results by opening the following file:

```
<PolySpaceInstallDir>\  
Examples\Demo_C\RTE_px_O2_Demo_C_LAST_RESULTS.rte
```

2 Select the following file located in “C:\PolySpace_Results”.

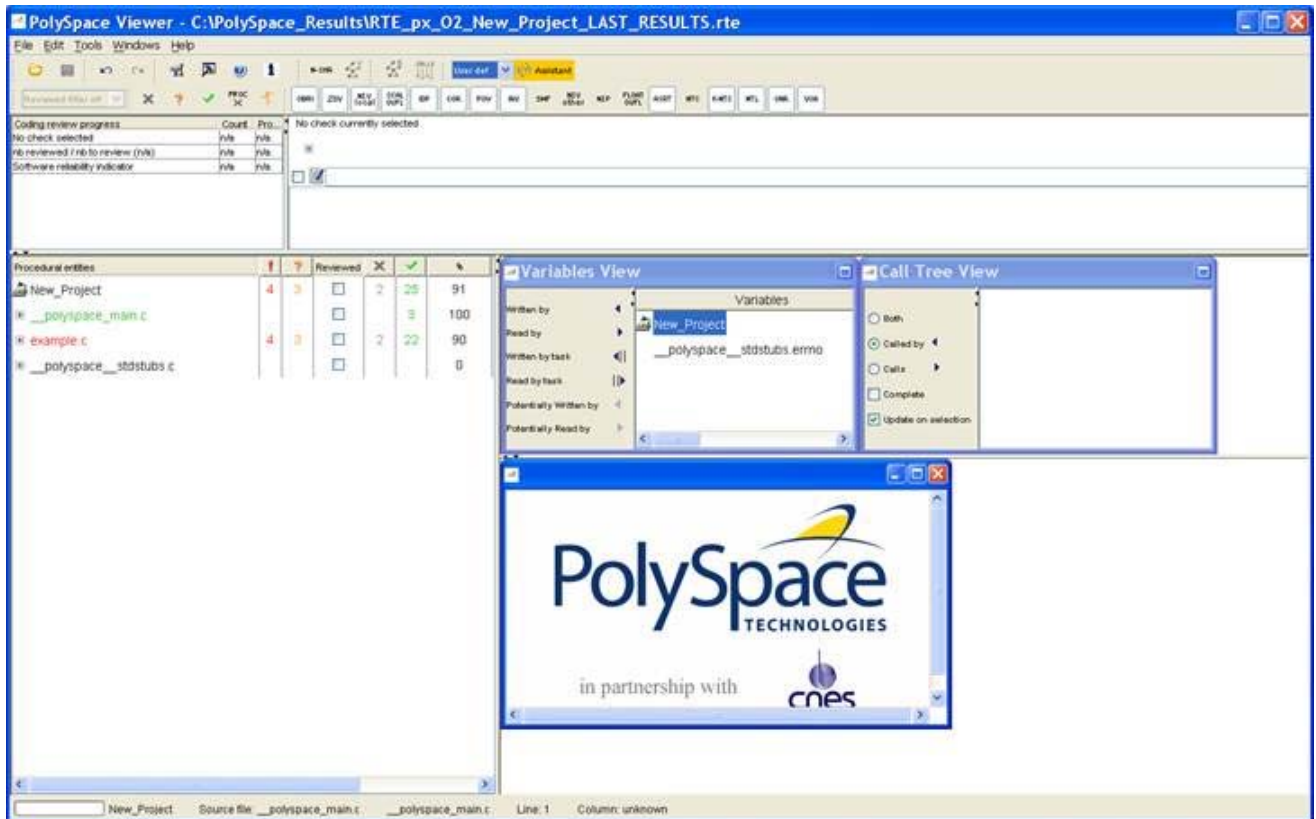


3 Click **Open** to proceed with further steps

Note The RTE_px_O2_Demo_C_LAST_RESULTS.rte is a sort of “link” on the best analysis in term of precision. This analysis is represented by RTE_p4_O2_Safety_Analysis_Level4.rte file. Lower level files represent lower precision analysis.

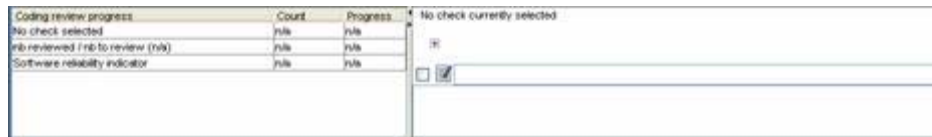
Reviewing PolySpace™ Results in “Expert” Mode (“example.c”)

After loading the results, and PolySpace Viewer window looks like below:

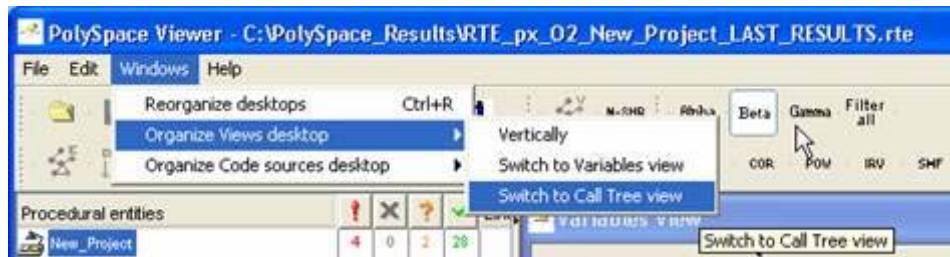


- On the left is the Procedural entities view (or RTE view). It displays the list of packages which have been analyzed or used during the analysis (specifications).
- In the bottom right area is the source code view with colored instructions. Each operation checked is displayed using meaningful color scheme and related diagnostic:
 - **Red** — Errors which occur at every execution.

- **Orange** — Warning - an error may occur sometimes.
- **Grey** — Shows unreachable code.
- **Green** — Error condition that will never occur.
- The two windows just below the tool bar concern details of a currently reviewed check (when the check has been selected):




- The top right area is used for displaying both control and data flow results. You can switch from one view to the other by using the “Windows” menu:



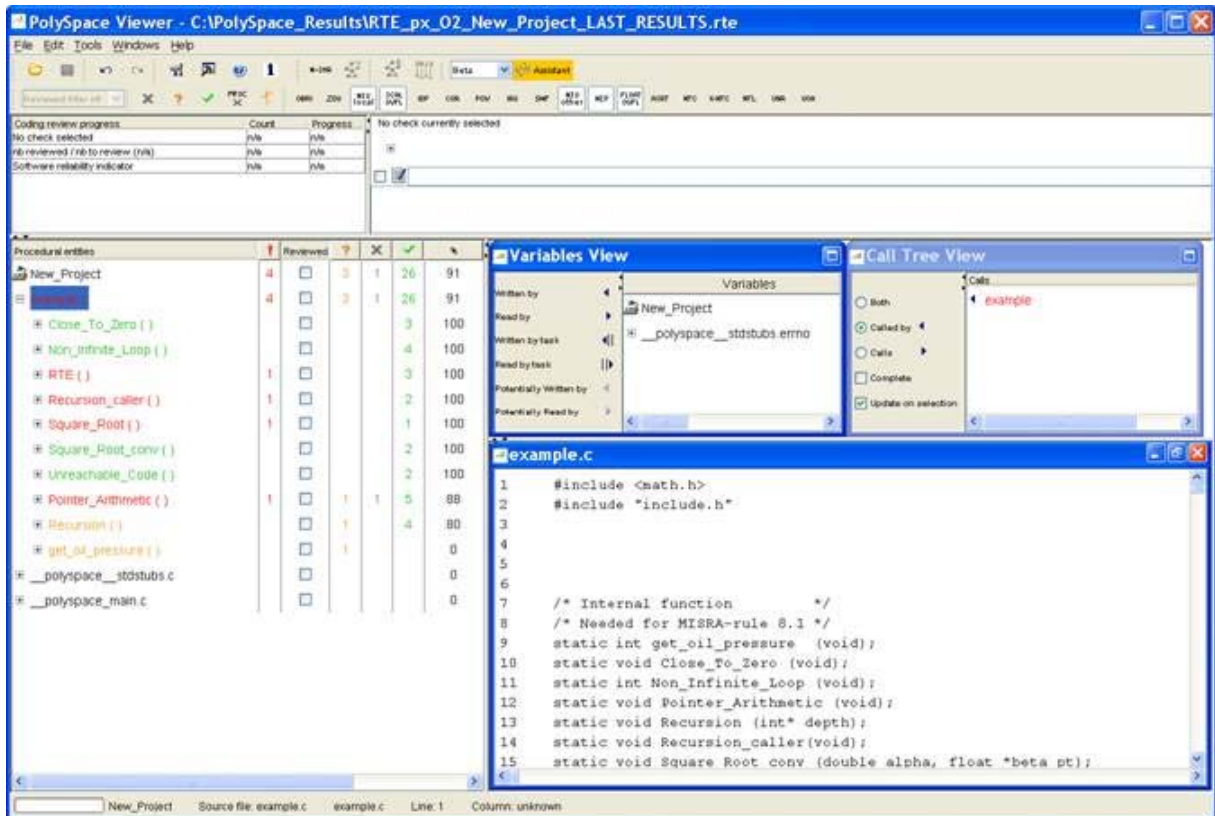
Procedural Entities View (RTE View)






Each file and underlying functions in the procedural entities view (or RTE view) is colored according to the most critical error found:






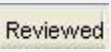
- `__polyspace_main.c` — This file contains the main which was automatically generated. All checks there are green: no run-time error (or RTE) has been found.
- `example.c` — This file is red: one or more *definite* run-time errors have been found in it.
- `__polyspace_stdstubs.c` — contains no checks. It contains stubs of standard functions part of libclibrary used in `example.c`.

Click once on the  left of “RUNTIME_ERROR” to find out more about this package.

“RUNTIME_ERROR” is expanded and the list of functions defined within “RUNTIME_ERROR” is displayed. The functions in red or grey have code sections that need to be inspected (PROCEDURE_ZDV, SQUARE_ROOT, etc.) first because they are definite diagnosis of PolySpace verification (either runtime errors or dead code).




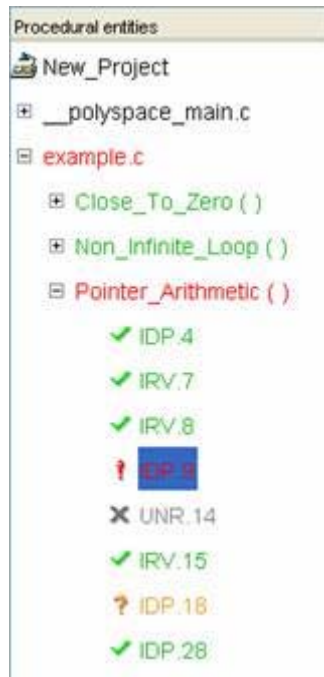
The columns (, , , , and ) provide information about run-time errors found in each function. The following table describes each of these columns.

Column	Indicates
	Reliability of the code (level of proof).
	Number of definite run-time errors or reds.
	Number of warnings or oranges (that may hide run-time errors that do not occur systematically).
	Number of safe operations or greens.
	Number of unreachable instructions or grey code sections.
	Allows marking reviewed checks.

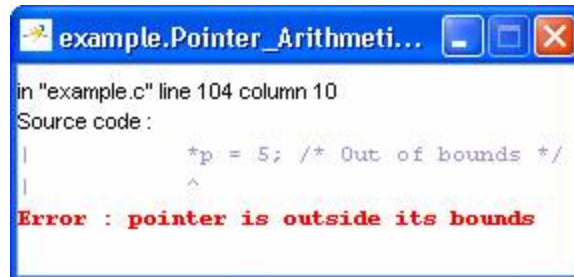
Lets have a look at some errors found by PolySpace verification in “example.c.”

First Example of Runtime Error: Memory Corruption. To investigate the first error:

- 1 Click on  to expand “Pointer_Arithmetic()“ to find out more about the red error. It displays a list of red, green, and orange symbols, featuring the complete list of code areas that PolySpace verification checked within the “Pointer_Arithmetic()” function.



- 2 Click on the red “IDP.9” item - which stands for Illegal De-referenced Pointer -, to precisely locate this error in the source code. The bottom right section is updated showing the location of the “IDP.9” item.
- 3 Click on red symbol in the source code at line 104. An error message is opened:



Pointer `p` is de-referenced outside of its bounds. Indeed, at the line 71 the instruction `*p = 5;` corrupts the memory as it puts the value “5” outside of the array “tab” pointed to by the pointer “p”.


Information about this red IDP is also accessible in the right windows below the toolbar line and the left one gives some statistic about all the IDP in the analysis:

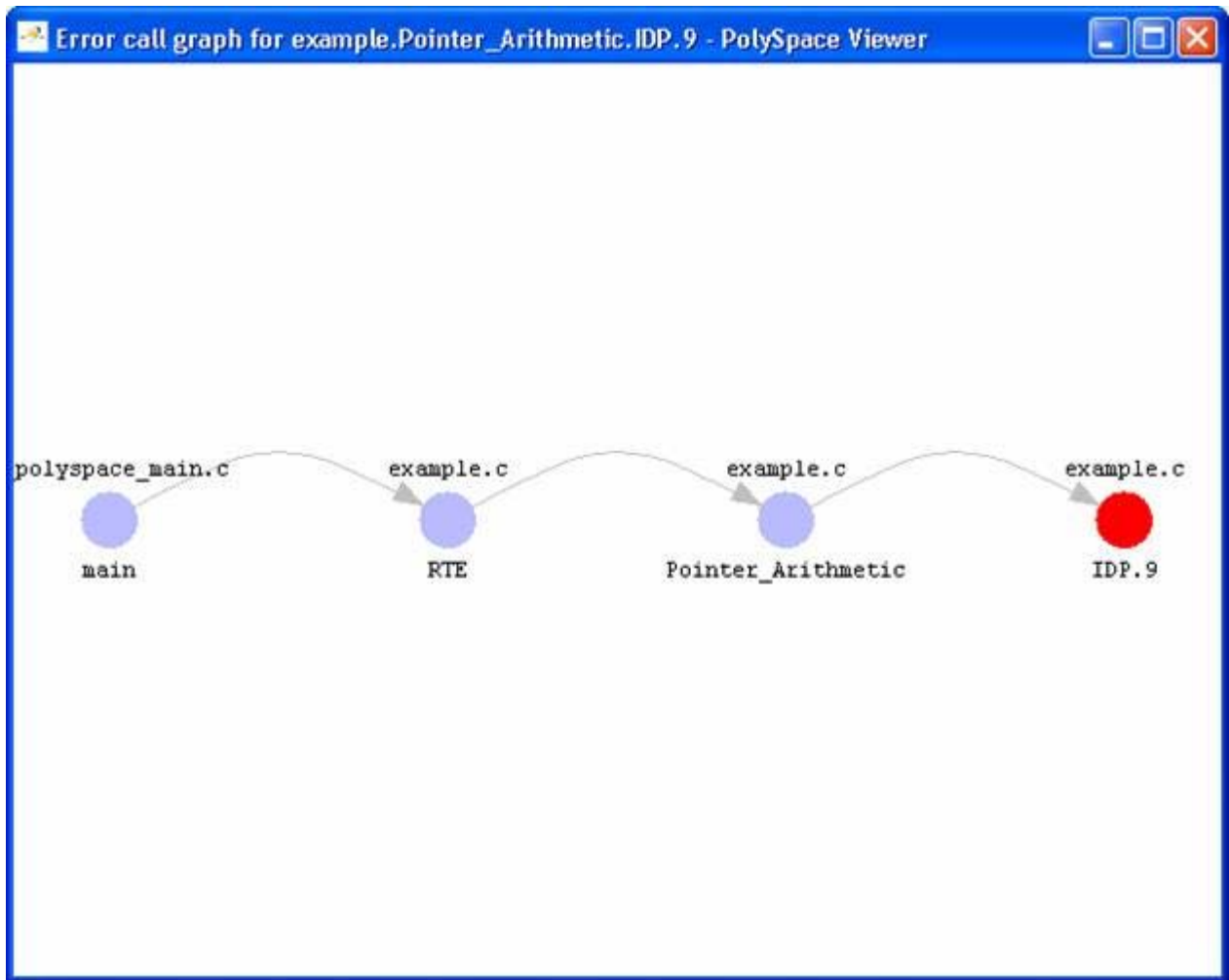
Coding review progress	Count	Progress
nb IDP reviewed / nb IDP to review (Red)	0/1	0
nb reviewed / nb to review (Red)	0/4	0
Software reliability indicator	93/115	80

example.c / Pointer_Arithmetic / line 104 / column 10

```
*p = 5; /* Out of bounds */
```

Error : pointer is outside its bounds

- 4 You can also see the calling sequence leading to that particular red code section. To do so, select “IDP.9” item in the “Procedural entities” column in the RTE View, and then click on the  icon (on the top left of the PolySpace Viewer window) to display the corresponding run-time error access graph:



Second Example of Runtime Error: Unreachable Code. Select "Unreachable_Code()" in the RTE View. You can see that "x = x + 1" is unreachable (gray color on each check) because of the non satisfied boolean condition: "x" is never negative when evaluating "x<0". PolySpace verification has detected some dead code.


```
199  static void Unreachable_Code(void)
200      /* Here we demonstrate PolySpace Verifier's ability to
201         identify unreachable sections of code due to the
202         value constraints placed on the variables.
203         */
204  {   int x = random_int();
205      int y = random_int();
206
207      if (x > y)
208      {
209          x = x - y;
210          if (x < 0)
211          {
212              x = x + 1;
213          }
214      }
215
216      x = y;
217  }
```

Colors in the Source Code View


Each operation checked is also displayed using meaningful color scheme and related diagnostic in the source code view as links:

- **Red** — A link to the error message associated to the error which occurs at every execution.
- **Orange** — A link to an unproven message - an error may occur sometimes.
- **Grey** — A link to a check shown as unreachable code. The error message is in grey.
- **Green** — A link to a VOA (Value on Assignment) or an error condition that will never occur.
- **Black** — Represents some comments, source code that does not contain any operation to be checked by PolySpace verification in terms of run time errors and optimized operations, e.g. `x := 0;`

- **Blue** — Text highlighting the keyword “procedure” and “function”
- **Blue Underlined** — A link to a global variable in the “Global variable View”.

More Examples of Run-Time Errors

Unlike most other testing techniques, PolySpace verification provides the benefit of finding the exact location of run-time errors in the source code. Below are some examples that you can review with PolySpace Viewer.

In a First Example of the Second Set: Arithmetic Error. Click  to expand “SQUARE_ROOT” function. You can see the source code view in the bottom right.

You can also display the call tree for that function by using the “Windows” menu (see previous paragraph).

“Square_Root()” is called by RTE function from “example.c”. It is displayed as “example.RTE” in the “*Call tree view*” window (right of the top right section).

“Square_Root” calls “random_float” (automatically stubbed function), “Square_Root_conv” (from example.c) and “sqrt” (standard library).

```

179 static void Square_Root_conv (double alpha, float *beta_pt)
180     /* Perform arithmetic conversion of alpha to beta */
181 {
182     *beta_pt = (float)((1.5 + cos(alpha))/5.0);
183 }
184
185 static void Square_Root (void)
186 {
187     double alpha = random_float();
188     float beta;
189     float gamma;
190
191     Square_Root_conv (alpha, &beta);
192
193     gamma = (float)sqrt(beta - 0.75); /* always sqrt(negative number) */
194 }

```

The green sections into the source code view are error-free but the red (**sqrt**) is an issue that needs to be fixed. Indeed, when the local float variable gamma is computed in the line “gamma=sqrt(beta - 0.75);”, the operation will cause a run-time error, as the parameter passed to “sqrt” is always negative.

Note Using -voa option at launching time, PolySpace software can help more suitably by giving information of range on scalar assignment

Second Example of the Second Set: Non-Infinite Loop. Select “Non_Infinite_Loop()” in the “Procedural entities” column in RTE View. The function is fully green: it means that the locale variable x never overflows, even if the exit condition of loop deals with y that is smaller than x. PolySpace verification confirms that the function always terminates.

```
66 static int Non_Infinite_Loop (void)
67 { const int big = 1073741821 ; /* 2**30-3 */
68   int x=0, y=0;
69
70   while (1)
71   {
72     if (y > big) { break;}
73     x = x + 2;
74     y = x / 2;
75   }
76
77   y = x / 100;
78   return y;
79 }
80 }
```

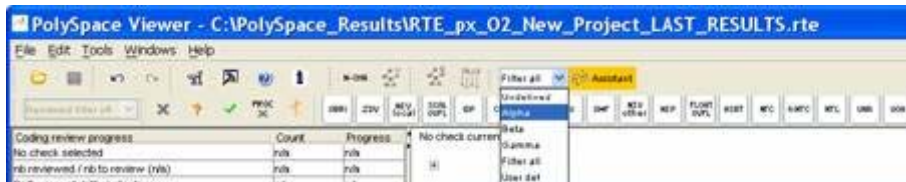
Third Example of the Second Set: Non-Infinite Loop. Select “Recursion_caller()”: The first call to Recursion is in red because when a negative parameter is passed, Recursion makes a division by zero (See the “Recursion” function). PolySpace verification also checks recursive constructs:

```
137 static void Recursion (int* depth)
138     /* if depth<0, recursion will lead to division by zero */
139 (   float advance;
140
141     *depth = *depth + 1;
142     advance = 1.0f/(float) (*depth); /* potential division by zero */
143
144
145     if (*depth < 50)
146     (
147         Recursion(depth);
148     )
149 )
150
151 static void Recursion_caller(void)
152 (   int x=random_int();
153
154
155     if ((x>-4) && (x < -1))
156     (
157         Recursion( &x ); // always encounters a division by zero
158     )
159
160
161     x = 10;
162     if (random_int() > 0)
163     (
164         Recursion( &x ); /* never encounters a division by zero */
165     )
166 )
```

Advanced Results Exploration


You can filter the information provided by PolySpace software to focus on the type of errors you wish to investigate.

There are pre-defined composite filters ( ,  and  that you can choose depending on your development process. These filters are accessible through a combo list:



To illustrate the use of these filters, we will focus on the Square Root function that we have examined in the previous section.

Gamma Mode. Gamma mode provides all the “red” and “grey” code sections. It is mainly used during the earliest development stages to focus quickly on critical bugs.

To select Gamma mode, click the  button.

The software reduces the information checks related to “SQUARE_ROOT”.




This list of acronyms - for type of operations checked - shows what PolySpace verification automatically analyzed for you.

Beta Mode. Beta mode highlights checks that could cause a processor halt, memory corruptions or overflows. Beta mode is the default mode.

To select Beta mode:

- 1 Click .

- 2 Select “Pointer_Arithmetic()” in the “Procedural entities”.
- 3 Click  to get the list of the checks.



Alpha Mode. Alpha Mode provides a comprehensive list of operations checked by PolySpace verification.



To switch to Alpha mode, click



You may also want to use filters to focus on particular categories of errors. Those filters are located at the top of the PolySpace Viewer window:




Note When the mouse pointer moves on the filter, a tool tips gives its definition.

- Click  (top of the window) to suppress all checks, then click  .

You will get list of checks containing only IDP (Illegal Dereference Pointers) reds, oranges or greens:





- Click  (top of the window) to suppress green code sections.

You will get a reduced list of checks reds, oranges and grays:



Miscellaneous

The  icon gives access to the PolySpace documentation. All views have a pop-up menu (right click on mouse).

Close the PolySpace Viewer window by clicking on the upper right  symbol (PolySpace Viewer can also be closed using **File > Close**).


Methodological Assistant

After a first navigation into the PolySpace Viewer, some simple questions remain:

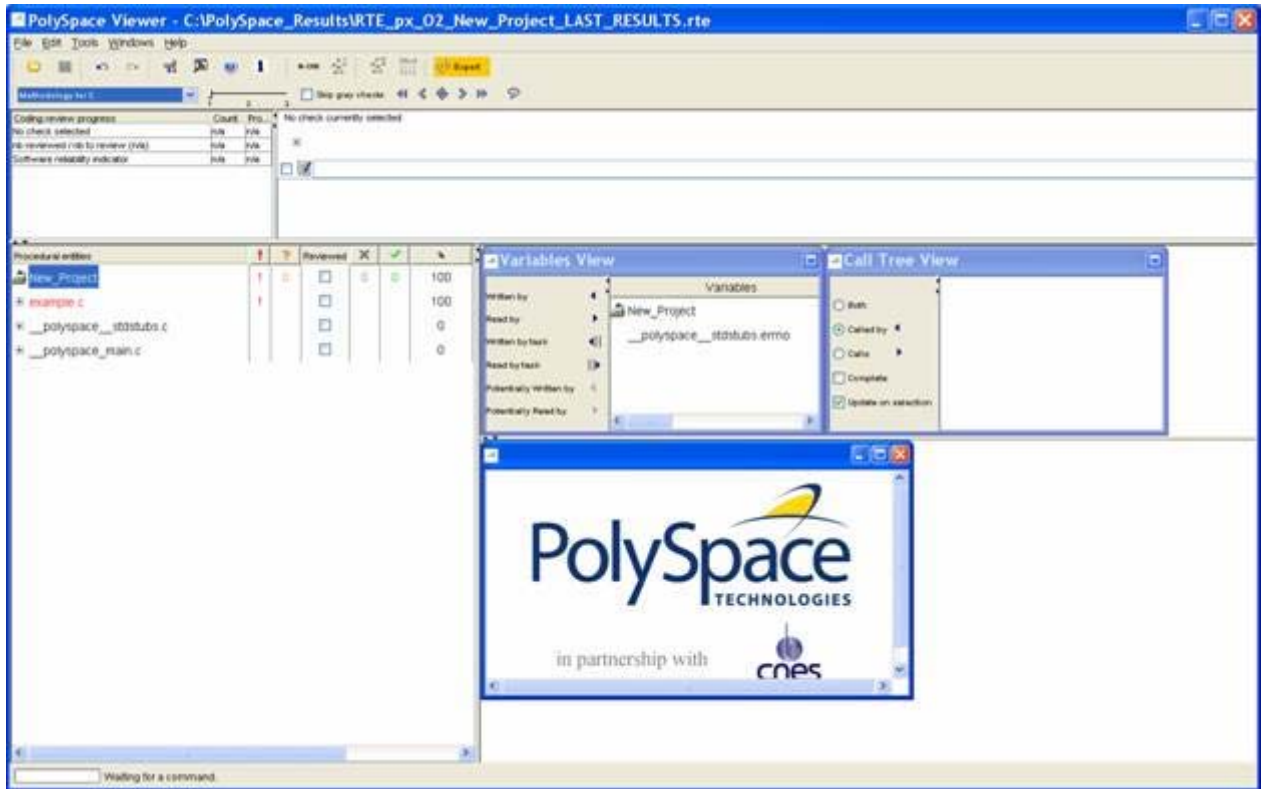
- Do all checks need to be reviewed?
- What are the checks to review?
- How many?
- What is the best order?

The Methodological assistant is here to answer to all these questions: It helps to select and manage the checks to be reviewed. It selects a “best” subset and sorts out them. The Assistant mode in the PolySpace Viewer will then guide through these selected checks.

To open the assistant:

- 1** If the PolySpace Viewer is still open, close it by clicking on the upper right  symbol.
- 2** Open the PolySpace Viewer again, then load the same results.
- 3** Choose “Assistant” mode.

After having loaded the results in “Assistant” mode, PolySpace Viewer window looks like below:




Assistant Dashboard

The second line of buttons on the toolbar and the two views just below are the navigation centre based on the methodological method used in the assistant mode:



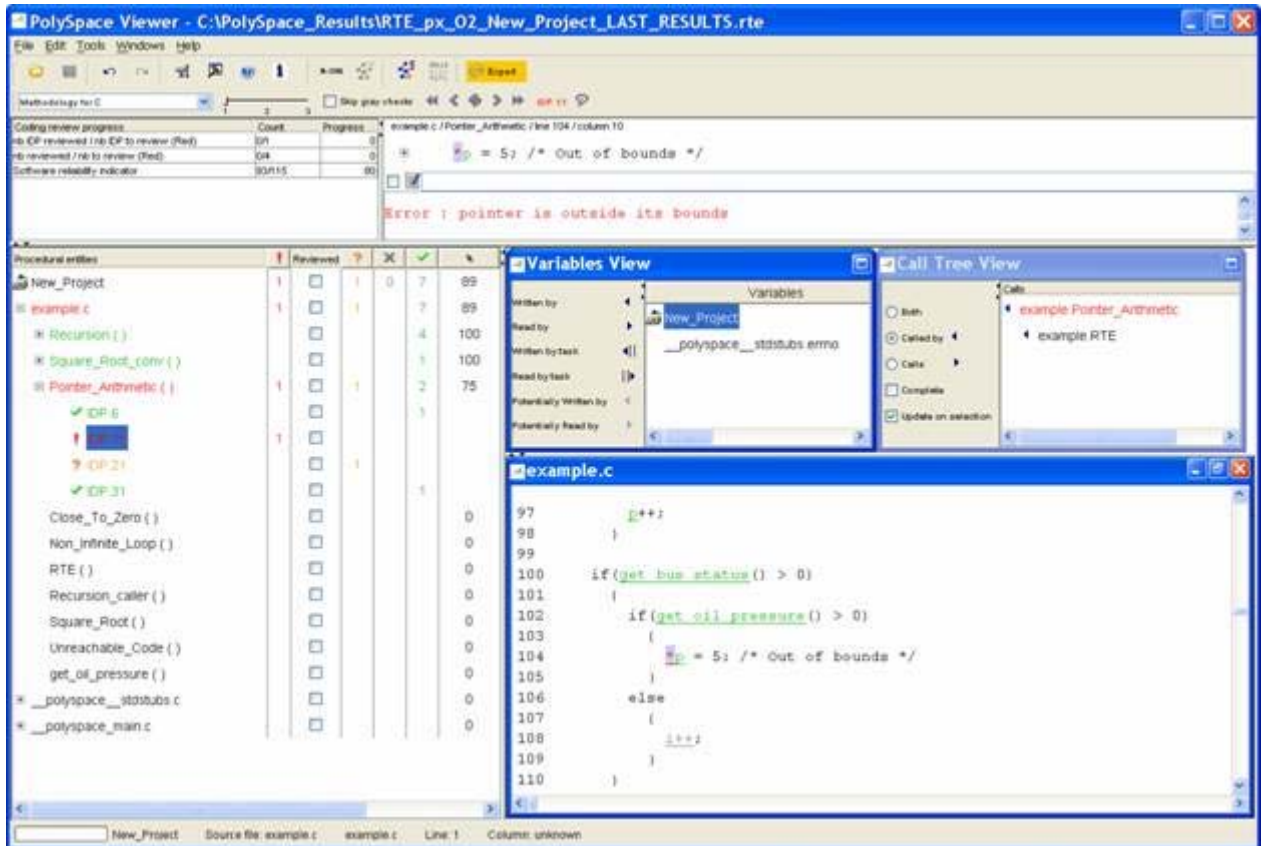
Some other changes can be seen in the viewer:

- Now, in the “Procedural Entities” view the list of files analyzed *is sorted by the methodological assistant used*.
- In the bottom right area is the source code view with colored instructions. Each operation will be checked and sorted by the methodological method using meaningful color scheme and related diagnostic and in the following order:
 - **Red** — Assistant browses all errors which occur at every execution.
 - **Gray** — Assistant browses each block of unreachable code depending if radio button “Skip gray checks” has been ticked or not.
 - **Orange** — Assistant chooses and reviews the “best” unproven operations -errors that may occur sometimes.

- 1 Click  to navigate to next check.

The PolySpace Viewer has been refreshed with the first check selected by the Methodology of review:

2 Getting Started



The Methodological dashboard gives details and allows reviewing the check. On the selected check, it is possible to mark the fact that it has been reviewed.

- 2 Select the radio button box.
- 3 Enter a comment in the associated edit box on the right.

After, it looks like:

The screenshot shows the PolySpace Viewer interface. On the left, a table displays coding review progress statistics. On the right, a code editor shows a snippet of C code with a red error message.

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

example.c / Pointer_Arithmetic / line 104 / column 10

```
*p = 5; /* Out of bounds */
```

I have reviewed this check and inserted a comment

Error : pointer is outside its bounds

The left part of the dashboard has been updated, and displays some statistics in three lines:

- The first line gives the number and percentage of remaining checks to review of the current category. In the previous example, it concerns red IDP checks.
- The second line gives values in the color category (red, grey and unproven).
- The Last line gives in permanence the Software reliability indicator.

Other buttons in the Methodological dash board allow navigating to previous check, coming back to current one



and going to next

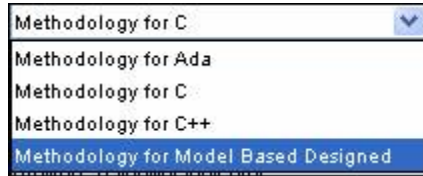


/ previous

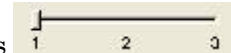


category of reviewed checks selected by the Methodology.

Choose a Methodological Assistant



and associated levels



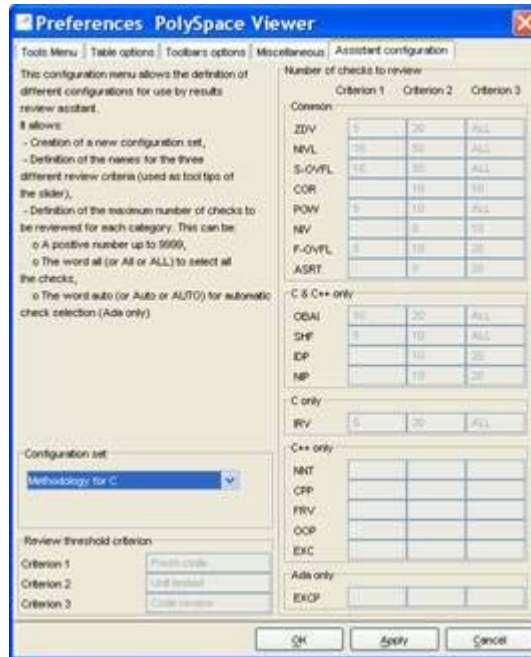
have been pre-selected by PolySpace software.

The methodology allows selecting the categories of checks to review, the number for each category and their order depending of a statistical algorithm.

The level (or criterion) defines the number of checks to review by category. Explicit name have been associated to each criterion like “Fresh code”, “Unit test” and “Code review”

It is possible to refine a self-created one or define its own Methodology.

- 1 Select **Edit > Preferences** in the PolySpace Viewer.
- 2 Select the Assistant Configuration tab.



3 Create a new configuration set

Define the categories of check to review for each criterion, how many in each one.

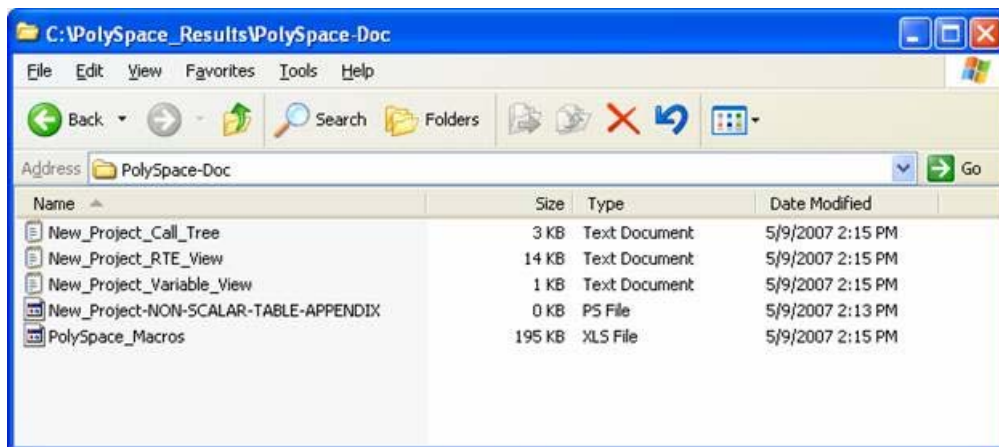
Note You cannot change an existing configuration except by duplication and refinement.

Report Generation

When PolySpace software performs an analysis, it generates textual files that can be used to generate Excel® reports. These files are located in the results directory (See "C:\PolySpace_Results\PolySpace-Doc" or "<PolySpaceInstallDir>\Examples\Demo_C\PolySpace-Doc").

All views (except source code) are printable and can be exported to textual or Excel format (protected by license).

The "C:\PolySpace_Results\PolySpace-Doc" directory should contain the following files:



To generate a report:

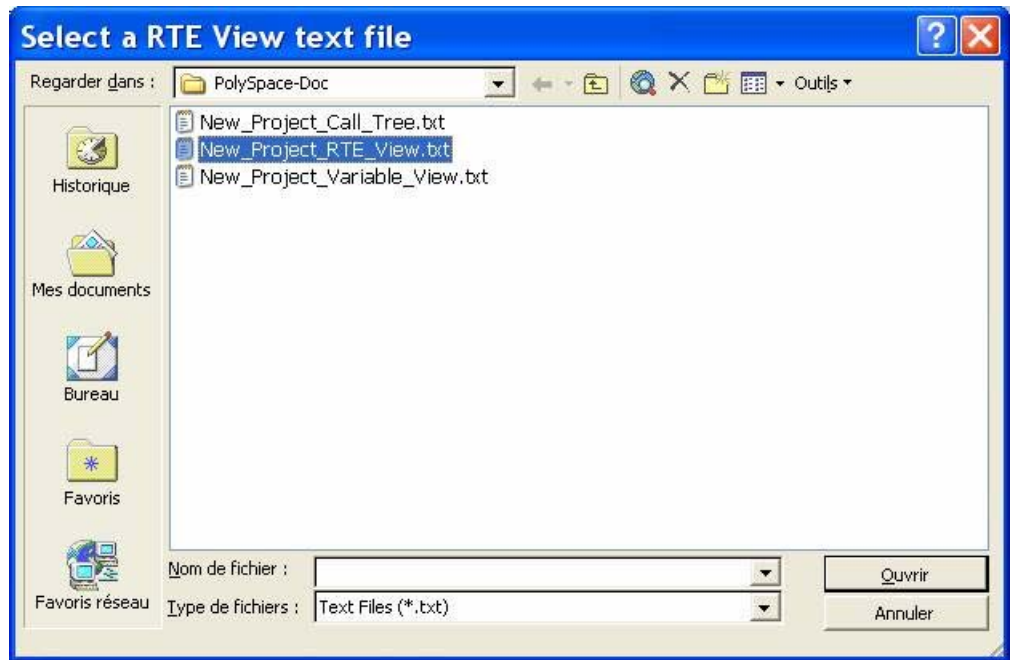
- 1 Open the file called "PolySpace_Macros.xls", enable macros when asked and then the following window opens:

	A	B	C	D	E	F	G	H
1								
2	Copyright © PolySpace Technologies, 1999-2006							
3								
4	<div style="border: 1px solid black; padding: 10px;"> <div style="display: flex; justify-content: space-between;"> <div style="border: 1px solid black; padding: 5px;"> <p>Apply filters?</p> <p><input checked="" type="radio"/> No filters</p> <p><input type="radio"/> Beta filters</p> </div> <div style="border: 1px solid black; padding: 5px;"> <p>Generate checks by file?</p> <p><input checked="" type="radio"/> yes</p> <p><input type="radio"/> no</p> </div> </div> <p style="text-align: center;"> 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 </p> <p style="text-align: center; color: red; font-weight: bold;"> Generate PolySpace Results Synthesis </p> </div>							
9								
10								
11								
12								
13								
14								
15								
16								
17	Reports can be generated from all PolySpace txt file format results. These are generated							
18	by the PolySpace Verifier during an analysis, the export option in the PolySpace Viewer,							
19	or from the command line using the "gen-excel-files" command.							
20								
21	Individual PolySpace text result files can be processed using the below macros:							
22								
23	<u>The macros are:</u>							
24	<div style="display: flex; justify-content: space-between; align-items: flex-start;"> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;">RTE</div> <div>Apply to RTE views exported from PolySpace Viewer</div> </div>							
25	<div style="display: flex; justify-content: space-between; align-items: flex-start;"> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;">Call Tree</div> <div>Apply to Call Tree views exported from PolySpace Viewer</div> </div>							
26	<div style="display: flex; justify-content: space-between; align-items: flex-start;"> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;">Variables</div> <div>Apply to Variable views exported from PolySpace Viewer</div> </div>							
27								
28								
29	Version 3.4.1D				RTE = Run Time Error			
30								

2 Click on

Generate PolySpace Results Synthesis

A file browser opens.



- 3 Select the file called “New_Project_RTE_View.txt”.

After a few seconds, an Excel file is generated. It contains several spreadsheets related to the application analyzed.

Application Call Tree / Shared Globals / Global Data Dictionary / Checks by file / Check Synthesis / Launching Options / RTE -> All checks location / Orange CI

For example, in “Checks Synthesis” all statistics about checks and colors are reported in a summary table.

	A	B	C	D	E	F	G
1	RTE Statistics						
2	Check category	Check detail	R	O	Gy	Gr	% proved
3	OBAI	Out of Bounds Array Index	0	0	0	0	0,00%
4	NIVL	Uninitialized Local Variable	0	0	1	28	100,00%
5	IDP	Illegal Dereference of Pointer	1	1	0	7	88,89%
6	NIP	Uninitialized Pointer	0	0	0	12	100,00%
7	NIV	Uninitialized Variable	0	0	0	8	100,00%
8	IRV	Initialized Value Returned	0	0	0	15	100,00%
9	COR	Other Correctness Conditions	0	0	0	2	100,00%
10	ASRT	User Assertion Failure	0	0	0	0	0,00%
11	POW	Power Must Be Positive	0	0	0	0	0,00%
12	ZDV	Division by Zero	0	1	0	4	80,00%
13	SHF	Shift Amount Within Bounds	0	0	0	0	0,00%
14	OVFL	Overflow	0	3	2	8	76,92%
15	UNFL	Underflow	0	1	2	9	91,67%
16	UOVFL	Underflow or Overflow	0	3	0	4	57,14%
17	EXCP	Arithmetic Exceptions	0	0	0	0	0,00%
18	NTC	Non Termination of Call	3	0	0	0	100,00%
19	k-NTC	Known Non Termination of Call	0	0	0	0	0,00%
20	NTL	Non Termination of Loop	0	0	0	0	0,00%
21	UNR	Unreachable Code	0	0	0	0	0,00%
22	UNP	Uncalled Procedure	0	0	0	0	0,00%
23	IPT	Inspection Point	0	0	0	0	0,00%
24	OTH	other checks	0	0	0	0	0,00%
25	Total :		4	9	5	97	92,17%

This ends ways of results review.

Setting Up and Launching the MISRA C® Checker

In this section...

- “Before You Begin” on page 2-50
- “Selecting MISRA C® Rules to Check” on page 2-52
- “Running the MISRA® Checker” on page 2-59

Before You Begin

- “Overview” on page 2-50
- “Activating the MISRA C® Checker” on page 2-50

Overview

This section describes the basic steps to add the MISRA C® Checker in the analysis of “example.c”. This operation takes place during ANSI® C compliance phase of the analysis.

Activating the MISRA C® Checker

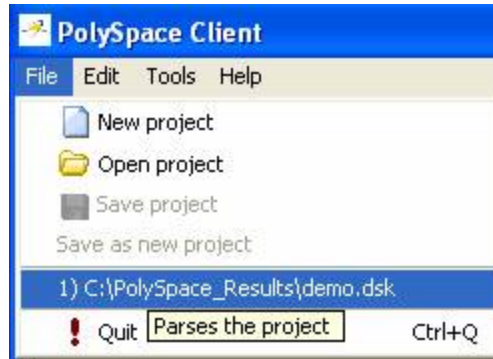
The project created during first step of this guide needs to be updated by activating the “**Check MISRA rules**” option while selecting the parameters.


Note If the PolySpace™ Client™ for C/C++ product is already opened with the project defined previously (refer to “Setting Up a PolySpace™ Client™ for C/C++ Analysis” on page 2-6), you can skip the first two steps.

- 1 If the PolySpace Client for C/C++ window has been closed, please open it again by double-clicking on the Client icon:



- 2 Select the saved project:



- 3 To set up the `-misra2` options, type “misra” in the **Search internal name from the selected line** (top right) box, then click .

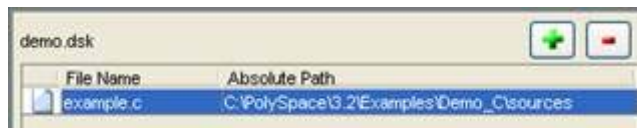
The software shows all options containing “misra” in the set of options part below.

- 4 Select the **Check MISRA-C: 2004 rules** option and expand it to see the two associated options `-misra2` and `-includes-to-ignore`:



We will detail these two options below.

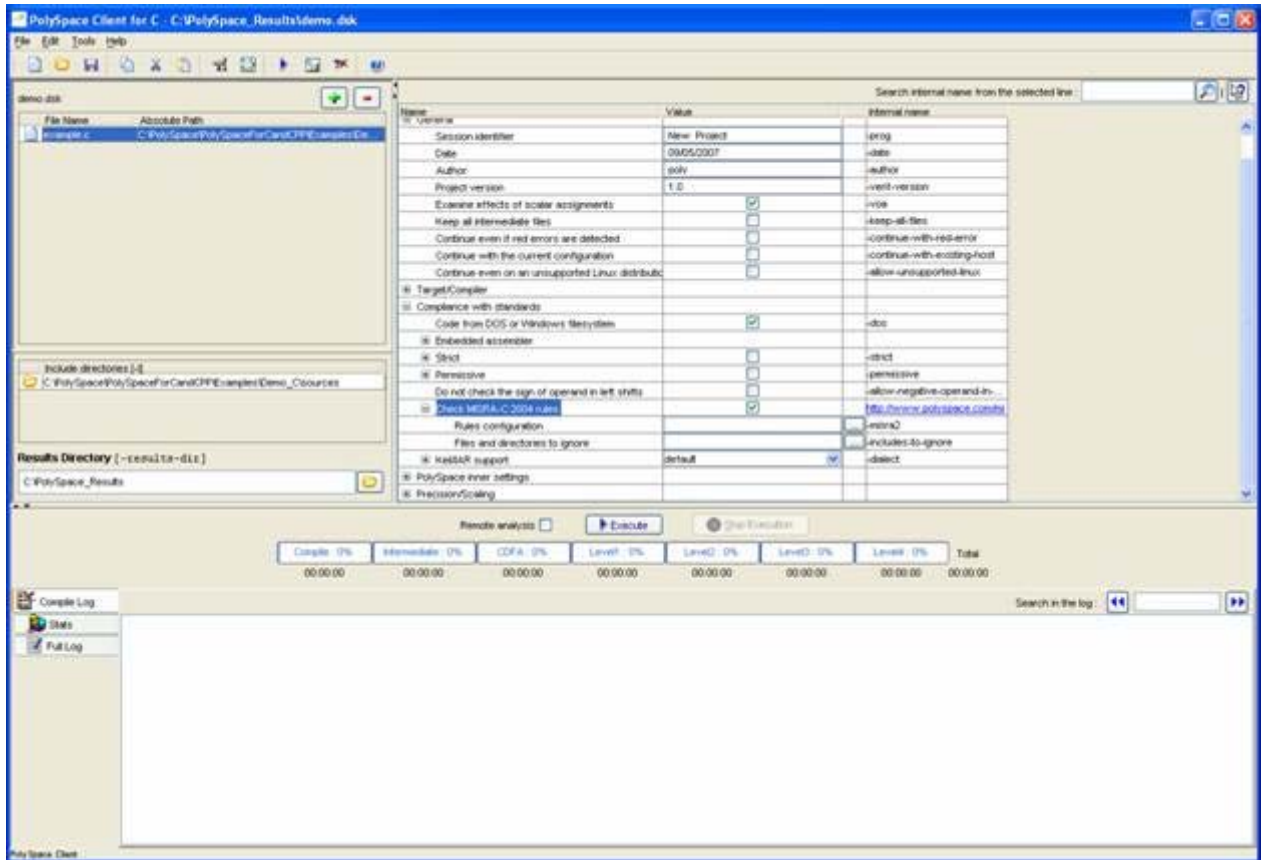
- 5 Make sure “example.c” is selected (location from `<PolySpaceInstallDir>\Examples\Demo_C\sources`):



- 6 Update the results directory:




The PolySpace™ Launcher should now look like this:



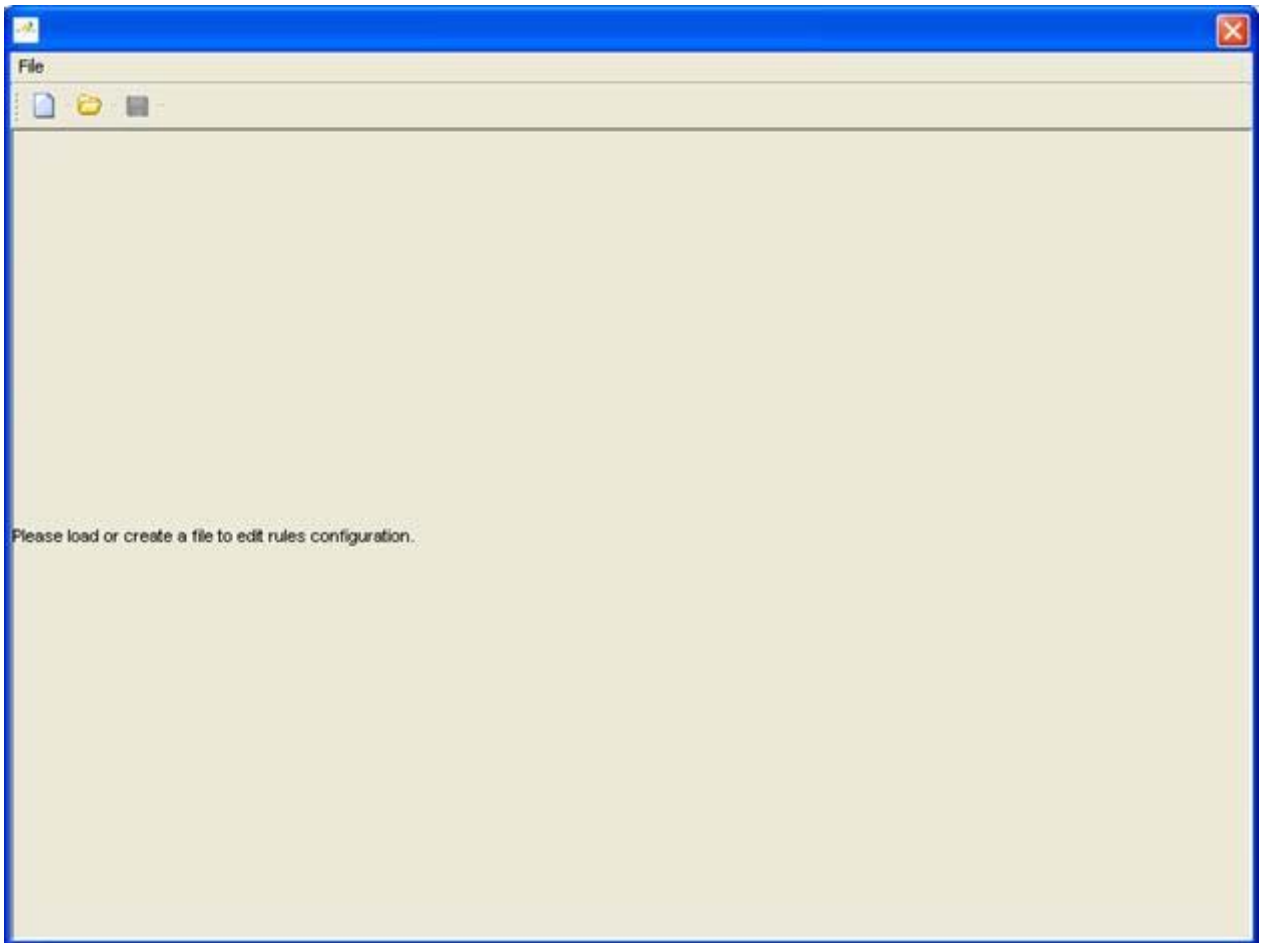
Selecting MISRA C® Rules to Check

- “File Configuration” on page 2-53
- “Discard Header Files from MISRA® Checking” on page 2-58

File Configuration

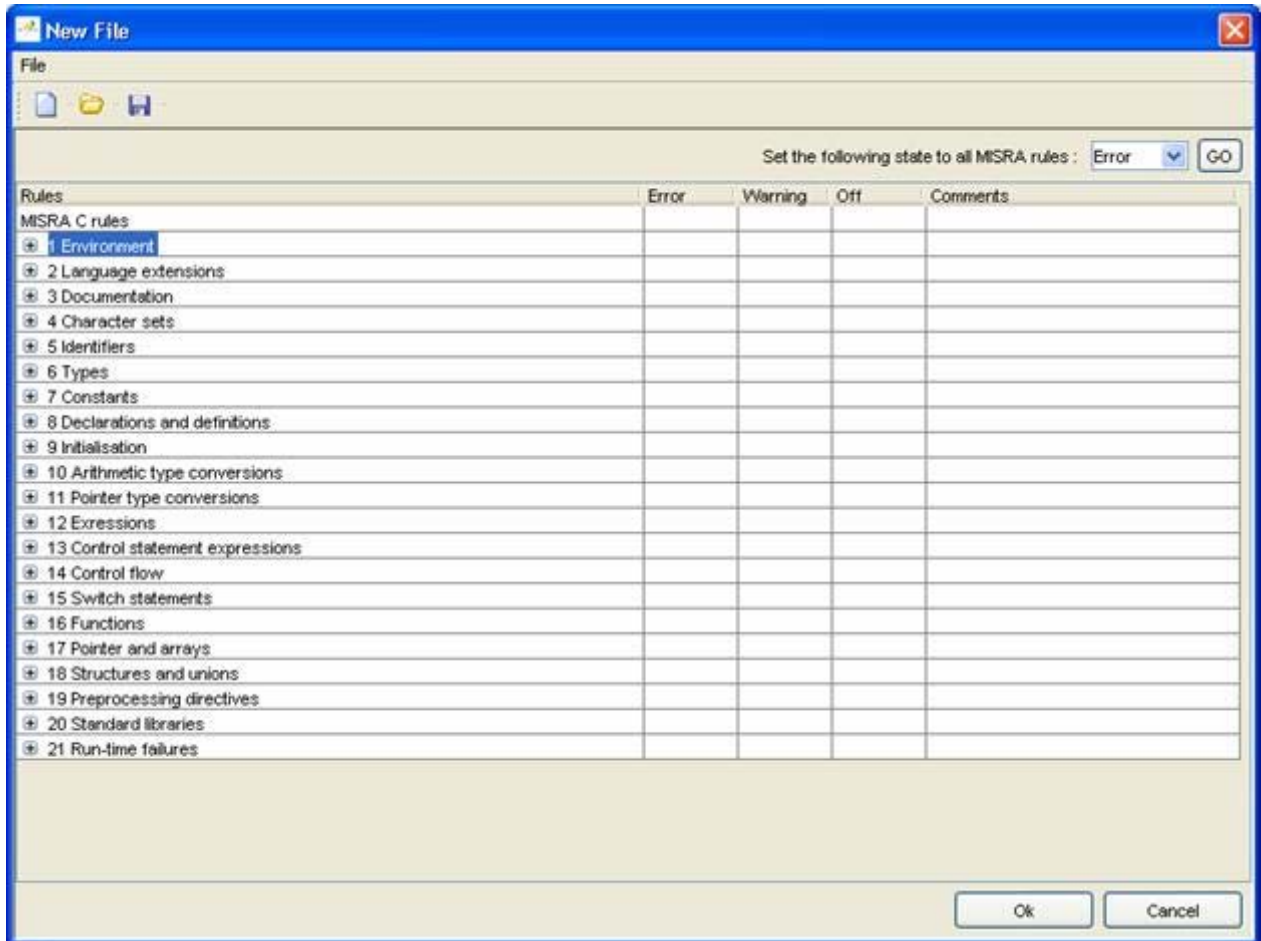
- 1 Click on  to invoke “Rules configuration.”

Note This button was enabled during activation of “Check MISRA-C: 2004 rules.”



- 2 Click  to create a new MISRA C configuration file.

The previous window is updated.

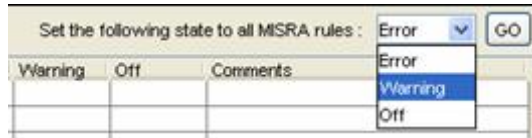




3 Set each rule as follows:

- Error — this MISRA C rule must be respected. If one or several errors are detected, the analysis will stop at the end of the compilation phase.
- Warning — if this MISRA-C rule is not respected, a warning will be displayed, but the analysis will continue.

- Off — the MISRA-C rule will not be verified by PolySpace MISRA® Checker module


Note The default setting for all rules is Warning.

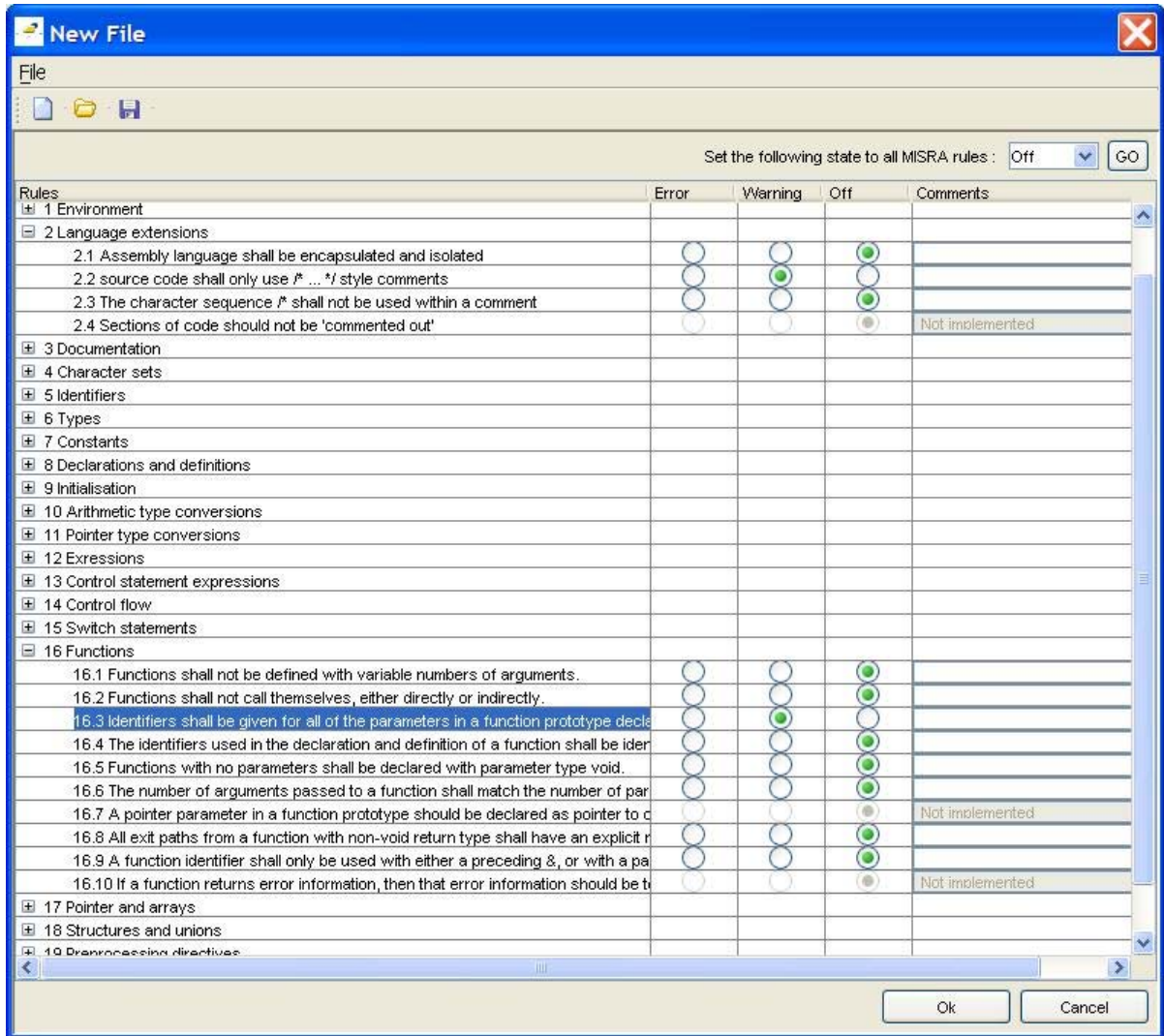


- 4 For the MISRA-C check of “example.c” file, please update the setting to Off for all rules and apply it using the  button
- 5 Click on  to expand the set of rules **16. - “Functions”**.

The status of some of the underlying rules cannot be modified (rules **16.7** and **16.10**), others are “Off”.

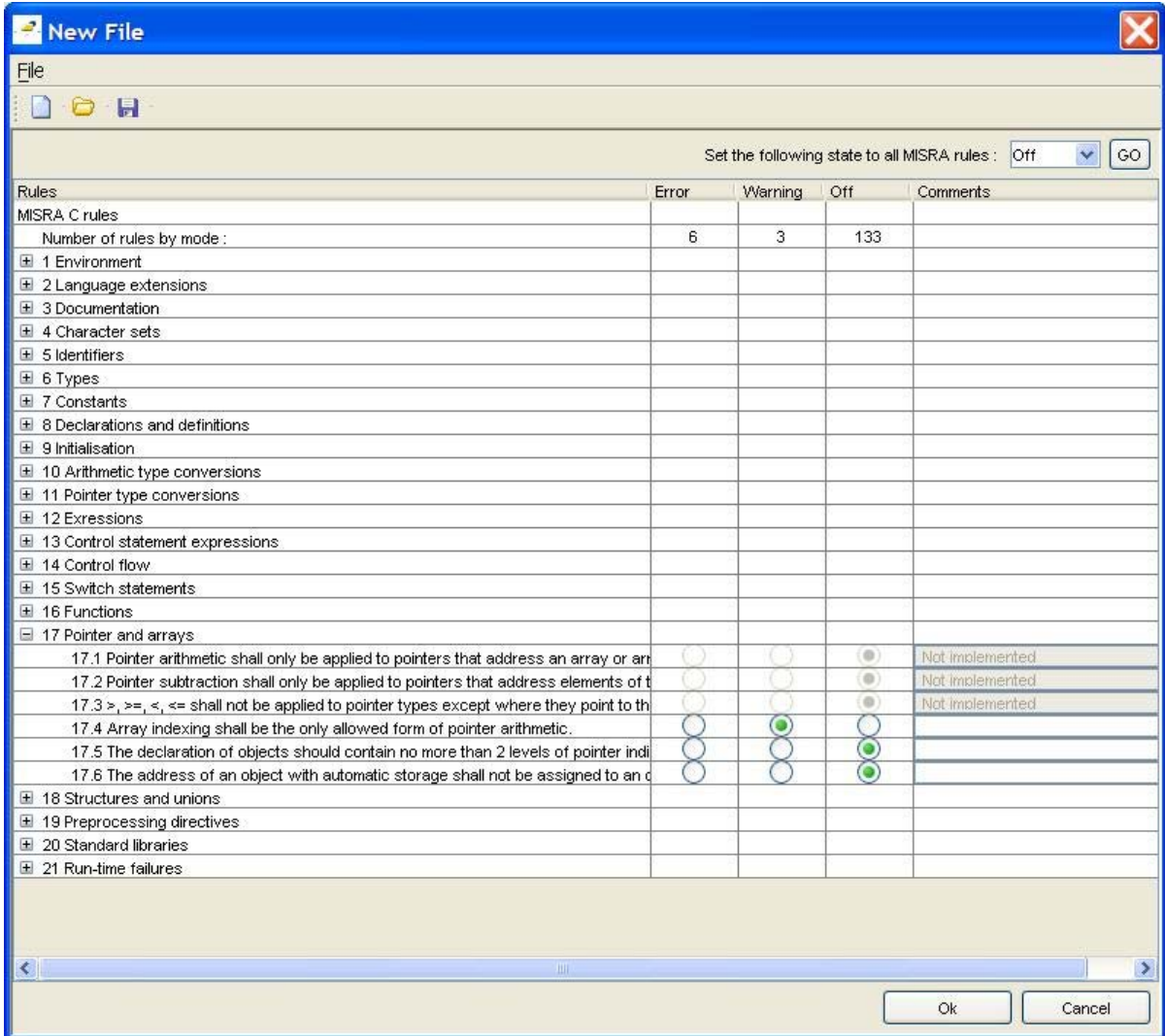
- 6 Click on the **Error** column for rule **16.3**.

The green dot  moves from column **Off** to column **Error**. This means that PolySpace MISRA Checker will verify whether the rule 16.3 (“Identifiers shall be given for all of the parameters in a function prototype”) is respected and will stop after the ANSI compliance checking phase if this is not the case.



- 7 Click on **+** to expand the set of rules 17- "Pointers and arrays," then select **Warning** for rule 17.4.

This means that PolySpace MISRA Checker will verify whether the rule 17.4 (“Array indexing shall be the only allowed form of pointer arithmetic”) is respected and display a warning message if this is not the case.



8 Click  .


A “Save As ” window opens, enabling to save the current configuration.

- 9 Type `misrarules.txt` in the `C:\MISRA_results` directory.

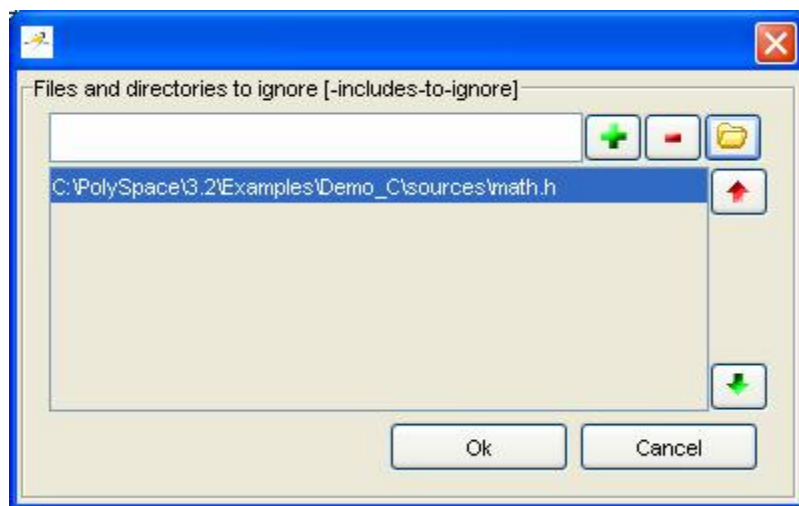
Discard Header Files from MISRA® Checking


You can disable MISRA verification on predefined files. For example, you may want to disable the MISRA C verification of `math.h`, included in `example.c`.

To discard header files:

- 1 Click  next to the `-includes-to-ignore` option.

The “Files and directories to ignore” window opens, enabling to disable the verification of MISRA-C rules on selected files and directories.



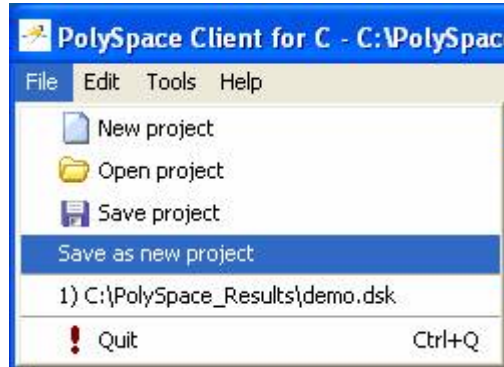
- 2 Select “`math.h`” using the browse button, then click  to close the window.

This file will not be checked.


This ends the setting up of the MISRA-C checking phase.

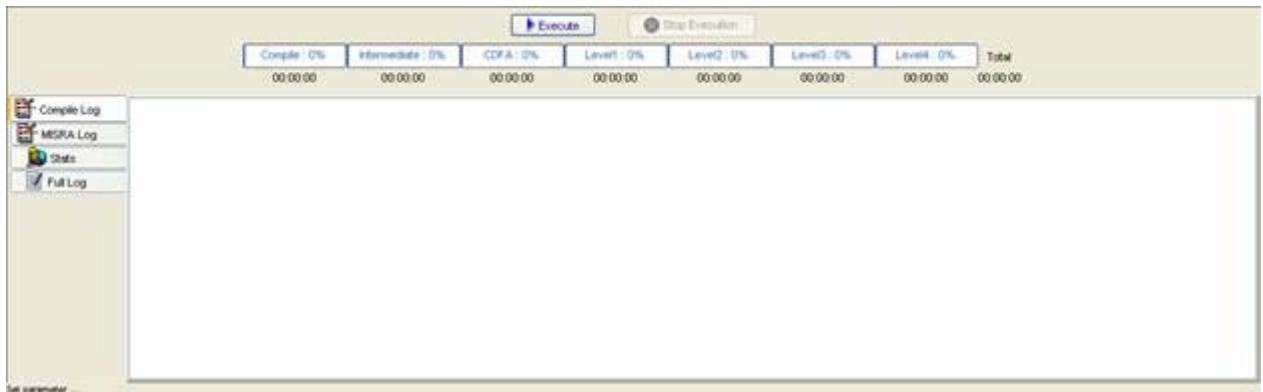
Running the MISRA® Checker

- 1 Before starting this analysis of “example.c” with MISRA C checker, Select “File>Save as new project” and choose another name for the current PolySpace project.



- 2 Click  to start the analysis.

During the new analysis performed on “example.c”, a new filter button is displayed -  - in the log view. This button enables the user to filter out messages associated to the MISRA-C Checker.




At the end of the compilation process, PolySpace Desktop shows the following error message:



- 3 Click  .

The analysis has been interrupted, because the MISRA-C checker found that rule **16.3**, marked as “Error”, has not been respected. A list of MISRA-C errors and warning messages appears in the bottom window.

- 4 If we focus on the MISRA log using appropriate filter  , only messages associated to MISRA-C checker are displayed.

Also, the “Search in log” box permits to navigate into log file searching for an appropriate key word: a particular rule for instance.

```
Verifying C files ...
Verifying example.c
Verifying sources ...
Verifying example.c
 /sources/include.h:34 : MISRA-C ERROR : rule 16.3 (required) violated.
 | Identifiers shall be given for all of the parameters in a function
 | prototype declaration.
example.c:97 : MISRA-C WARNING : rule 17.4 (required) violated.
 | Array indexing shall be the only allowed form of pointer arithmetic.
example.c:113 : MISRA-C WARNING : rule 17.4 (required) violated.
 | Array indexing shall be the only allowed form of pointer arithmetic.
example.c:117 : MISRA-C WARNING : rule 17.4 (required) violated.
 | Array indexing shall be the only allowed form of pointer arithmetic.
example.c:121 : MISRA-C WARNING : rule 17.4 (required) violated.
 | Array indexing shall be the only allowed form of pointer arithmetic.
```

- 5 Here, the recommendation is clear — an identifier is missing in a function prototype and must be added in “include.h” as required by MISRA-C rule **16.3**. Once this is done, you can re-launch the analysis.

Note You can also change the setting on rule **16.3** from “Error” to “Warning”, and launch the analysis again. The error message will change to “MISRA-C WARNING”, and the analysis will not stop after the ANSI checking phase.

- 6** When no error remains after the ANSI checking phase, the analysis continues as described in step 1, and will give results as described in step 2.

Note A log file, located at the root of the “C:\PolySpace_Misra_Results” directory with “.log” as suffix, contains all messages displayed in the bottom window, including MISRA C messages. The format of the log file name is the following: “PolySpace_4_2_X_Y_<Name-Project>_<date>_<time>.log”.

Launching PolySpace™ Analysis Remotely

In this section...
“Overview” on page 2-62
“Launching an Analysis” on page 2-62
“Management of PolySpace™ Analysis in Remote: the PolySpace™ Spooler” on page 2-64
“Batch Commands” on page 2-67
“Sharing Analyses Between Accounts” on page 2-69

Overview

This section describes the basic steps to launch an analysis in remote.

To do so you need:

- A Queue Manager server (QM) installed.
- Your desktop PC configured with the PolySpace™ Client™ for C/C++ product.
- A networked machine configured with the PolySpace™ Server™ for C/C++ product.

Please see the PolySpace™ Installation Guide (available on the PolySpace CD-ROM in \Docs\Install) to install and configure a Client and a Server.


Note Launching an analysis remotely requires a PolySpace Server for C/C++ product and associated license.

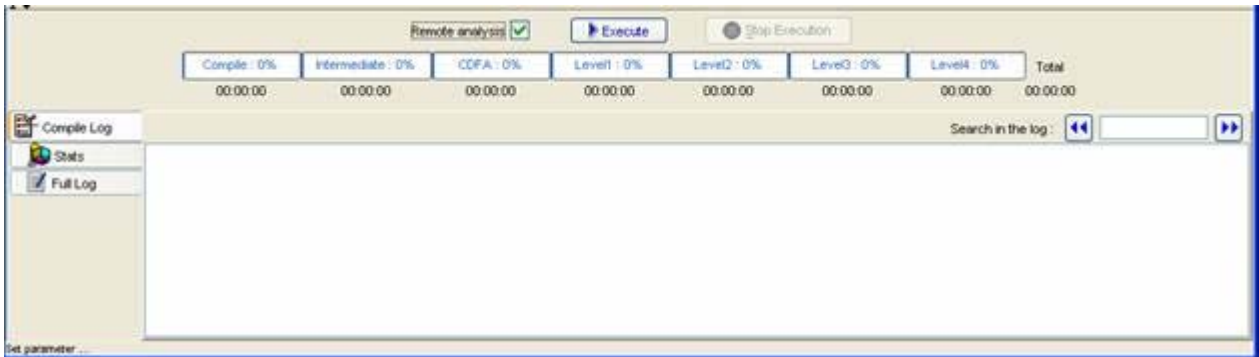
Launching an Analysis

To launch an analysis remotely:

- 1 Set up an analysis as described in “PolySpace™ Client — Analyzing a Single C File” on page 2-5, but do not launch it.

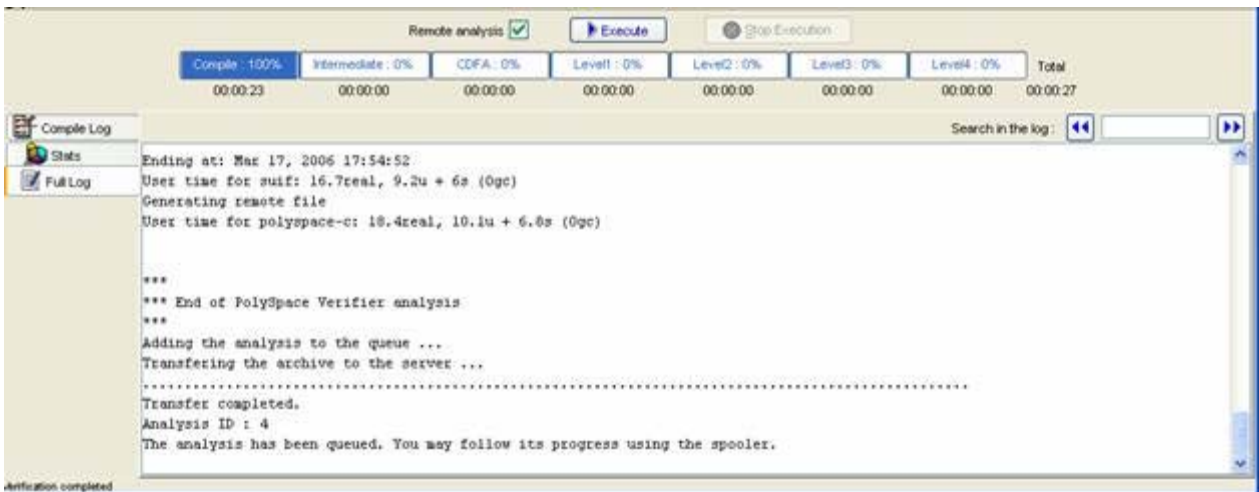
2 Select the **Remote analysis** checkbox (see next figure).

3 Click  to launch the analysis.



The analysis starts and the compilation phase is performed on the desktop PC. At the end of the “C source verification phase” the analysis is sent to the Queue Manager server.

4 Click on the **Full Log** tab. You will have a message like this:



The analysis has been queued with an ID number, and you can follow its progression using the PolySpace Spooler.

Note If you do not select the “Remote analysis” radio button, the analysis continues locally.

Management of PolySpace™ Analysis in Remote: the PolySpace™ Spooler

You can check the analysis processes in the queue using the PolySpace Spooler.

To manage an analysis in the queue:

1 Open the PolySpace Spooler by either:

- Clicking on the short cut on your desktop PC

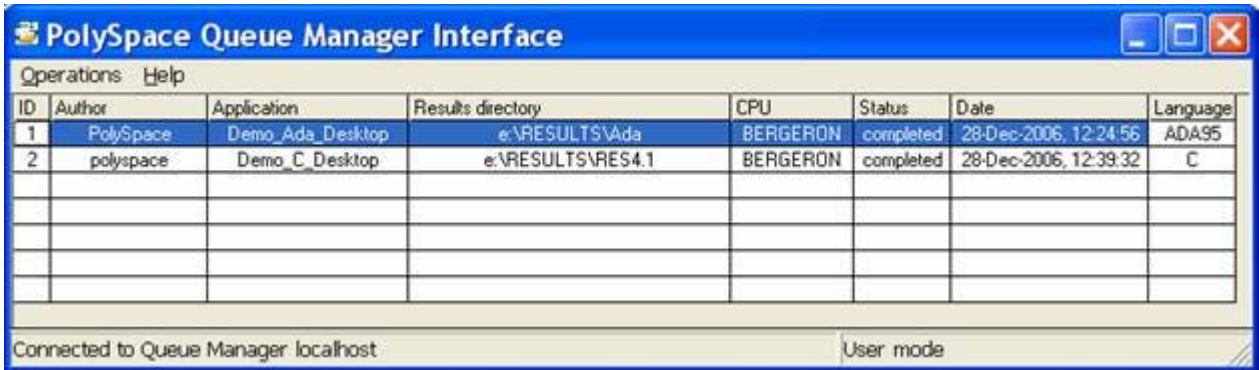


- Clicking on the icon



in the menu tab of the launcher.

The PolySpace Spooler appears.



2 Right-click on an analysis to manage it in the queue:



3 Select one of the following options:

- **Follow progress** — This action lists the associated log file in a Launcher window. If the analysis is running, you can follow the update of the log file and associated progress bar in real time on the Launcher window.
- **View log file** — This action lists the associated log file in a “Command prompt” window, in which you can the last 100 updated lines of the log file in real time. This option is only available when the analysis is running.
- **Download results** — This action downloads the results of an analysis onto the client. If the analysis is still running, available results are

downloaded on the client, without disturbing the analysis. The option is not possible for a “queued” analysis

- **Move down in queue** — This action reduces the priority of a “queued” analysis.
- **Kill and download results** — This action stops the analysis definitively and the results are downloaded. The status of the analysis changes from “running” to “aborted”. The analysis remains on the queue.
- **Kill and remove from queue** — This action stops the analysis definitively, and the analysis is removed from the queue.

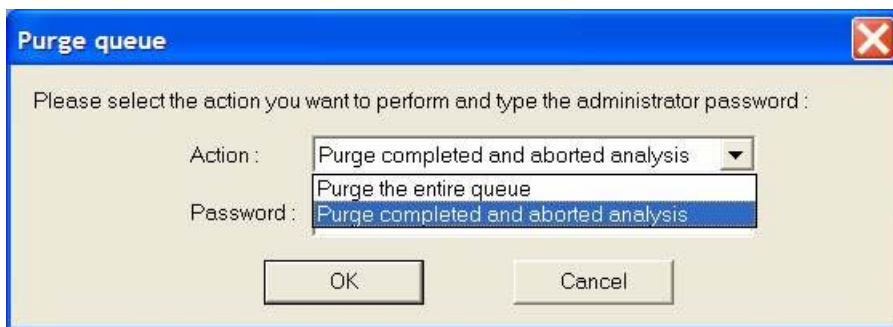
Note The results will be lost

- **Remove from queue** — This action removes a “queued”, “aborted” or a “completed” analysis.

Note The results will be lost

You can also manage the queue from an administrator point of view using the **Operations** menu:

- Select **Operations > Purge queue**, to purge the entire queue or purge only completed and aborted analysis (see next figure).



Note The queue manager password is required.

- Select **Operations > Change root password**, to change the administrator password of the queue manager or the default one.

Note By default the password is “administrator”.

Batch Commands

- “Launching an Analysis in Batch” on page 2-67
- “Managing an Analysis in Batch” on page 2-68

Launching an Analysis in Batch

A set of commands allow the launching of analysis in batch.

All these commands begin with the following prefixes:

- **Server analysis** —
`<PolySpaceInstallDir>/Verifier/bin/polyspace-remote-c`
- **Client analysis** —`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 analysis 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 an Analysis in Batch

In batch, a set of commands allow the management of analysis in the 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 analysis 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 analysis.
- `psqueue-purge all|ended` — remove all or finished analyses in the queue.
- `psqueue-dump` — gives the list of all analyses in the queue associated to default Queue Manager.
- `psqueue-move-down <id>` — move down an identified analysis in the Queue.
- `psqueue-remove <id>` — remove an identified analysis 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 analysis.
 - `[-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.

Sharing Analyses Between Accounts

- “Analysis-key.text File” on page 2-69
- “Magic Key or Shared Analysis Between Projects” on page 2-70

Analysis-key.text File

From a security point of view, all analysis spooled on a same Queue Manager are owned by the user who sent the analysis from a specific account. Each analysis has a unique cryptic key.

The public part of the key is stored in a file `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 the ASCII file is the following (spaces are tabulation):

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

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

Example:

```
1 m120 27CB36A9D656F0C3F84F959304ACF81BF229827C58BE1A15C8123786
```

```
2 m120 2860F820320CDD8317C51E4455E3D1A48DCE576F5C66BEEF391A9962
8 m120 2D51FF34D7B319121D221272585C7E79501FBCC8973CF287F6C12FCA
```

When we make an attempt of management (download, kill and remove, etc.) on a particular analysis, the Queue Manager will examine this file and find the associated public key to authenticate the analysis on the server.

If the key does not exist, an error message appears: “key for analysis <ID> not found”. So sharing an analysis with another user account necessitates the public key.

Sharing an analysis is quite simple, ask to the owner of the analysis the line in `analysis-key.txt` which containing the associated <ID> and put it the line in your own file. After, it will be able to download the analysis.

Magic Key or Shared Analysis Between Projects

A magic key allows sharing analyses without taking into account the <ID>. It allows same key for all analysis launched by a user account. The format is the following:

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

All analyses spooled will have this key instead of random one. In the same way, if this kind of key is available in an `analysis-key.txt` file of another user, it allows to authorize any operation on any analyses pushed with this key.

Note It only works for all analysis launched after having put the magic key in the file. If the analysis has been launched before, the allowed key associated to the ID will be used for the authentication.

Summary

After having followed each steps of this tutorial, you are now able to launch an analysis using the PolySpace™ Client™ for C/C++ product, enabling or not the MISRA® Checker phase, and explore some results with PolySpace™ Viewer. All these command can be performed locally on your desktop PC or in Client/Server architecture.

You will find more information on advanced options available with our tools in the following chapters.

Analysis Setup

Compile Errors (p. 3-2)	Describes how to use PolySpace™ verification to detect compile errors
Link Messages (p. 3-32)	Provides examples of link errors
Stubbing Errors (p. 3-40)	Describes how to use PolySpace to detect stubbing errors
Intermediate Language Errors (p. 3-57)	Describes the log file containing error messages
Advanced Setup (p. 3-59)	Describes how to prepare your code to streamline orange checks

Compile Errors

In this section...
“Overview” on page 3-2
“Messages” on page 3-2
“Compiling Operating System Dependent Code (OS-target issues)” on page 3-6
“Target Specific Issues” on page 3-9
“Assembly Code” on page 3-23
“Dealing with Backward “goto” Statements” on page 3-29

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).

Messages

Some examples of compilation errors are detailed below:

- “Syntax error” on page 3-3
- “Undeclared identifier” on page 3-3
- “No such file or directory” on page 3-4
- “Compilation errors with key words: @interrupt, @address(0xABCDEF)” on page 3-4

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 Compilations Options Efficiently” on page 3-47.

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 analysis 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 Compilations Options Efficiently” on page 3-47.

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 key words: @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 first character. But in this case, the problem cannot be addressed by means of a compilation flag, nor a -include file - that is, the “gather compilation options.”

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

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

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

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

Note In Linux, you must:

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

Launch an analysis on the example “my_file.cpp” below, and confirm that the compilation phase generates no errors.

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

int titi; // will be replaced by int toto

int r=0; r++;    toto++;
}
```

To confirm that the right transformation has been performed, open the expanded file “my_file.ci” which is located in the directory “<results_folder>/C-ALL/my_file.ci”

Compiling Operating System Dependent Code (OS-target issues)

- “List of already predefined compilation flags” on page 3-6
- “My target application runs on a Linux® OS” on page 3-8
- “My target application runs on Solaris™” on page 3-8
- “My target application runs on Vxworks” on page 3-8
- “My target application runs neither on Linux®, vxworks nor Solaris™” on page 3-9

List of already 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>/cininclude/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>/cininclude/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 a Linux® OS

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`.

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 runs neither 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
```

Target Specific Issues

- “Target Specification (size of char, int, float, double...)” on page 3-9
- “Generic/Custom Target” on page 3-11
- “Address Alignment” on page 3-12
- “”KEIL” and “IAR” Dialects” on page 3-13
- “Keywords to Automatically Ignore or Replace, Before Compilation” on page 3-20

Target Specification (size of char, int, float, double...)

The type of CPU to be used at run time determines various characteristics of data representation such as data sizes, addressing, etc. These factors determine whether some types of error (such as overflows) will occur or not.

Consequently, PolySpace must take the type of CPU used in the target environment into account.

PolySpace supports some of the most commonly used processors as listed in the table below. Even if the processor used in a target environment is not explicitly mentioned, it is safe to specify one from the table which shares the characteristics listed.

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

Target	char	short	int	long	long long	float	double	long double	ptr	char is	Endian	ptr diff type
sparc	8	16	32	32	64	32	64	128	32	signed	Big	int, long
i386	8	16	32	32	64	32	64	96	32	signed	Little	int, long
c-167	8	16	16	32	32	32	64	64	16	signed	Little	int
m68k / ColdFire ¹	8	16	32	32	64	32	64	96	32	signed	Big	int, long
powerpc	8	16	32	32	64	32	64	128	32	unsigned	Big	int, long
tms320c3x	32	32	32	32	64	32	32	40 ²	32	signed	Little	int, long
sharc21x61	32	32	32	32	64	32	32 ³	64	32	signed	Little	int, long
NEC-V850	8	16	32	32	32	32	32	64	32	signed	Little	int
hc08 ⁴	8	16	16	32	32	32	32	32	16 5	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 none of the characteristics described above match, please contact PolySpace Technical Support for advice.

1. The M68k family (68000, 68020, etc.) includes the “ColdFire” processor
2. All operations on long double values will be imprecise (that is, shown as orange).
3. On this target, a double may be 32 or 64 bits long. Only 32 bits double are supported.
4. Non ANSI C specified key-words and compiler implementation-dependent pragmas and interrupt facilities are not taken into account by this support
5. 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, but for which an analysis is possible. In the cases listed below, the target processor mentioned in the “Nearest Processor” column can be selected as a near equivalent. Where the characteristics are not identical between the target processor and its near equivalent, it is highlighted in red below. These mismatches need to be taken into account during results review.

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

Generic/Custom Target

The size of some basic types is configurable (-int-is-32bits option, compiler memory model option, near/far pointer syntax)

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

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

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

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

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

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 size⁷
- 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, $\text{global_alignment} = \max(\text{alignment char a, alignment int b}) = \max(8, 32) = 32$
- 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.

“KEIL” and “IAR” Dialects

In the typical embedded control application, reading and writing port data, setting timer registers and reading input captures etc. are commonplace. To cope with this without recourse to assembler, some compilers associated to micro processor have specified special data types like sfr and sbit. Typical declarations are:

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

and so on. 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.

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

When accessing a register or a port, using sfr data is then a simple matter but 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) */
}
```

Analyzing previous code with PolySpace is possible using the `-dialect` option. This option allows the Keil or IAR C language extensions to be supported even if some structures, keyword and syntax are not ANSI standard. The following tables summarize what is supported when analyzing a code which has been considered as associated to a dialect keilor iar.

The following table summarizes the keil C language extensions to be supported:

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

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

Type/Language	Description	Example	Restrictions
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.
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;</pre> <pre>/* file1.c */ sbit x = P0 ^ 1; /* file2.c */ extern bit x; x = 1; // set the 1st bit of P0 to 1</pre>	

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

Type/Language	Description	Example	Restrictions
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, epdata, huge, sdata, small, compact, large, reentrant. Defining -D __C51__, keywords large code, data, xdata, pdata and xhuge are ignored.		

The following table summarize the dialect iar:

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) will be considered as volatile. 	<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
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

Associated option `-sfr-types`, for keil or iar dialect, defines the size of a sfr type. The syntax for an sfr element in the list is `type-name=typesize`. For example `-sfr-types sfr=8,sfr16=16` define two sfr types: sfr with a 8 bit size and sfr16 with a 16-bits size. A value `type-name` must be given only once and 8, 16 and 32 are the only available values for `type-size`. Note that as soon as a 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 associated to specific targets currently exist. It is difficult to maintain a complete list of those supported.

Keywords to Automatically Ignore or Replace, Before Compilation

If you want to ignore non-compliant key words such as “far” or 0x followed by an absolute address, you can use the template described below to deal with them. Save it under `c:\PolySpace\myTpl.pl`, and select `myTpl.plin` the PolySpace Launcher using browse button associated to `-post-preprocessing-command`.

Content of the myTpl.pl file.

```
#!/usr/bin/perl

#####
```

```
# Post Processing template script
# Copyright 1999-2005 PolySpace Technologies.
#
#####
# Usage from Launcher GUI:
#
# 1) Linux: /usr/bin/perl PostProcessingTemplate.pl
# 2) Solaris: /usr/local/bin/perl PostProcessingTemplate.pl
# 3) Windows: /usr/bin/perl 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/;
#
#####

```

Assembly Code

Ignoring assembly code by using the option “-discard-asm” 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 (they can be ignored manually or by option) 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 analysis may produce inaccurate results. In such cases, please refer to the “Manual vs. Automatic Stubbing” on page 3-49, 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:

All statements are ignored: 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
}
```

Following example is automatically stubbed

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

All following examples have an empty body

The user must use the `-discard-asm` option.

```
#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 3-48.

#asm and #endasm support

The use of #asm and #endasm allows fragments of (typically) assembly code to be disregarded by PolySpace, irrespective of whether the -discard-asm option is used.

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
```

What to do if -discard-asm failes parsing an asm code section

There will be occasions when the -discard-asm option does not deal with a particular assembly code construction, particularly when the code fragment is compiler specific (Note that you could also consider using the -asm-begin and -asm-end options instead of the following approach).

Consider the 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 the example above, any text or code placed between the two `#pragma` statements is ignored by PolySpace. This allows any unsupported construction to be ignored by PolySpace 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 deal with backward “goto” statements, but MACROS provide a solution in most cases. In general, the analysis of C code which includes (a) backward “goto” statement(s) will stop at an early stage, and a message will appear saying that backward “goto” statements are not allowed by PolySpace.

Macros provided by PolySpace will 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:

- Edit the C file containing the “goto” statements;
- Add `#include pstgoto.h` at the beginning of the file (located in `<PolySpaceInstallDir>/cinclude`);
- Go to the the beginning of the block containing the “goto” statements;
- Just after the variable declarations (local to the block), insert the `USE_1_GOTO(<tag>)` macro call;
- Just before the end of this same block, insert the `EXIT_1_GOTO(<tag>)` macro call (taking care with the closing bracket `"}"`);
- Finally, replace `"goto <tag>"` with `"GOTO(<tag>)"`.

The following example would cause an analysis to terminate:

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

And this could be dealt with in the following way:

```
/* 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 concerned may contain many different uses of backward “goto” statements. The use of `USE_n_GOTO()` and matching `EXIT_n_GOTO()` statements will deal with this (that is, using `USE_2_GOTO()`, `USE_3_GOTO()` etc). Note that `pstgoto.h` need to be copied from `<PolySpaceInstallDir>/include` directory and location added in the list of `-I`.

The code block may also use several different tags and/or multiple “tag” parameters can be used to deal with these situations. So in the generic case, 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 Analysis
<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>

Link Messages

In this section...

“Overview” on page 3-32

“Function: Wrong Argument Type” on page 3-32

“Function: Wrong Argument Number” on page 3-33

“Variable: Wrong Type” on page 3-34

“Variable: Signed/Unsigned” on page 3-34

“Variable: Different Qualifier” on page 3-35

“Variable: Array Against Variable” on page 3-36

“Variable: Wrong Array Size” on page 3-36

“Missing Required Prototype for varargs” on page 3-37

“Can an Application without “main” be Analyzed? (For non Client mode only)” on page 3-38

Overview

This section gives some examples of link errors.

Note Looking at the pre-processed 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.

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 Compilations Options Efficiently” on page 3-47.

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.

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 Compilations Options Efficiently” on page 3-47.

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.

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 Compilations Options Efficiently” on page 3-47.

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
```

PolySpace Output:

```
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.

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 Compilations Options Efficiently” on page 3-47.

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 analysis. However, there is doubt regarding which statement is correct, and so PolySpace generates a warning.

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 Compilations Options Efficiently” on page 3-47.

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.

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 Compilations Options Efficiently” on page 3-47.

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
```

PolySpace Output:

```
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.

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 Compilations Options Efficiently” on page 3-47.

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 analysis is launched.

Can an Application without “main” be Analyzed? (For non Client mode only)

This section only concerns PolySpace Server for C. PolySpace Client for C will automatically generate a main if needed.

When your application is a function library (API) or a single module, you will have to provide a main that calls each function because of the execution model that is used by PolySpace. This manual technique is recommended in preference to the automated approach adopted by Client, because it allows a much more accurate model of the calling sequence to be generated.

There are three steps involved in manually defining the main.

- Identify the API functions and extract their declaration;
- Create a main to contain the declaration of a volatile variable for each type that is mentioned in the function prototypes;
- Create a loop with a volatile end condition. Inside this loop, create a switch block with a volatile condition and, for each API function, create a case branch that calls the function using the volatile variable parameters created previously.

Alternatively, refer to the “main generator” option section to automatically generate a main.

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

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

The main to be created manually is as follows:

```
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 */
}
}
```

Stubbing Errors

In this section...
“Errors when Compiling <code>_polyspace_stdstubs.c</code> ” on page 3-40
“Errors when Creating Automatic Stubs” on page 3-45
“How to Gather Compilations Options Efficiently” on page 3-47
“Stubbing” on page 3-48

Errors when Compiling `_polyspace_stdstubs.c`

- “Standard Error Messages” on page 3-40
- “Troubleshooting” on page 3-42

Standard Error Messages

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

Examples of such error messages follow. They highlight conflicts between a standard library function which is included as part of the application, and one of the standard stubs which is used by PolySpace 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 suggests problems with such functions.

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

Failing that, a selection of files is provided which contains stubs for most standard library functions which can be used instead of having them automatically stubbed.

For this to work effectively, it is important for you to include 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 target `string.h` file may be differ between targets **there are no default include directories for PolySpace**.

So, if the compiler has implicit include files, they must be specified by hand in the verification script, as illustrated in the following example.

```
(__polyspace_stdstubs.c is 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 analysis 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 3-42
- “Where preparation time is short or problems remain after trying solution 1” on page 3-43
- “Where all other attempts have failed” on page 3-44

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 analysis requires the applications own string.h include file.

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

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 re-launch 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 analysis by using a -D option (-D __polyspace_no_sprintf, say)

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 Compilations Options Efficiently” on page 3-47.

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 analysis again.                    ---
---                                                    ---
-----
---
```

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

How to Gather Compilations Options Efficiently

The code is often tuned for the target (as discussed to “KEIL” and “IAR” Dialects” on page 3-13). 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 analysis.

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. Doing so will allow the 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);
void fgetc(void);
```

Stubbing

- “Manual vs. Automatic Stubbing” on page 3-49
- “The Stubbing Options PURE and WORST” on page 3-51
- “The Default and Alternative Behavior for Stubbing” on page 3-52
- “Function Pointer Cases” on page 3-53
- “Stubbing Functions with a Variable Argument Number” on page 3-54
- “Finding Bugs in `_polyspace_stdstubs.c`” on page 3-55

Manual vs. Automatic Stubbing

The adopted approach to stubbing can have a significant influence on the speed and precision of your analysis, and there are occasions when automatic stubbing will not provide an adequate representation of the code it represents -with regards to both missing functions and assembly instructions.

Example

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

By relying on Verifiers 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 paragraph, *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 “Assembly Code” on page 3-23).

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;
}
```

By relying on Verifiers 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>

By relying on Verifiers default stub, the assembly code is ignored and the division `/` is green. The red division `/` could only be achieved with a manual stub.

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

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

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 analysis duration (“Reducing Analysis Time” on page 9-70) and precision.

Consider the example `int f(char *)`. In the analysis 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}		
<code>int f2 (int u);</code>	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]
<code>int f3 (int *u);</code>			Assumes the ability to write into *u to any depth and returns [-2 ³¹ , 2 ³¹ -1]

Initial Prototype	With pragma POLYSPACE_PURE	With pragma POLYSPACE_WORST	PolySpace default automatic stubbing
<pre>int* f4 (int u);</pre>	Returns an absolute address (AA)	Returns AA or (int *) u and assumes the ability to write into (int *) u	Returns an absolute address (refer to “Understanding Addressing” on page 8-35)
<pre>int* f5 (int *u);</pre>	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
<pre>void f6 (void (*ptr)(int), param2)</pre>	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.	
<pre>void f7 (void (*ptr)(param2)</pre>		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.	
		<p>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.</p>	

Function Pointer Cases

Function Prototype	Comments
<pre>int f(void (*ptr_ok)(int, char, float), other_type1 other_param1);</pre>	The <code>-permissive-stubber</code> option is not required.

Function Prototype	Comments
<pre>int f(void (*ptr_ok)(int *, char, float), other_type1 other_param1);</pre>	<p>The -permissive-stubber option is required because of the “int *” parameter of the function pointer passed as an argument</p>
<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 analysis;
- It causes a significant increase in complexity and hence in analysis 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 analysis tenfold.

Consider the following example.

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

```
#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;
- Initialise 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 Verifier 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).

Intermediate Language Errors

The analysis log can sometimes indicate that a red error has been detected in the previous phase, and that the analysis 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 analysis 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:
* certain 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.
```

Advanced Setup

In this section...

“Variables — Declaration and Definition” on page 3-59

“Types Promotion” on page 3-60

“Code Preparation for Variables” on page 3-63

“Code Preparation for Built-in Functions” on page 3-69

“My Code is Multitasking” on page 3-69

Variables – Declaration and Definition

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

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.

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, the `-allow-undef-variable` is required to start the analysis. Where that option is used, PolySpace will consider the variable to be initialized, and to potentially take any value in its full range (see “How are variables initialized?” on page 3-68).

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

Types Promotion

- “An Example of an Unsigned Promoted to Signed” on page 3-60
- “What are the Promotions Rules in Operators?” on page 3-61
- “Example” on page 3-61

An Example of an Unsigned Promoted to Signed

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

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

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

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

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

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

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

Implicit promotion explains this behavior.

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

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

A negative cast into unsigned gives a big value, which has to be bigger than 7. And this big value can never be ≤ 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 (refer to “What is the Type of Constants/What is a Constant Overflow?” on page 8-26);
- Be careful when analyzing 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.

Code Preparation for Variables

- “How can I assign ranges to variables/assert?” on page 3-63
- “Checking properties on global variables at any point: Global assert” on page 3-64
- “How can I model variable values external to my application?” on page 3-67
- “How are variables initialized?” on page 3-68

How can I assign 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, Verifier will produce checks for them. The behavior matches that exhibited during execution, because **all execution paths for unsatisfied conditions are truncated** (red and then grey). Thus it can be assumed that any analysis 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.

In order to use this feature you need to firstly include the file "pst_gassert.h", then create a list Pst_Global_Assert statements for the variables you are interested in.

This header is located in <PolySpaceInstallDir>/include folder.

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 a code editor window titled 'to.c' with the following 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 dialog box 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;
|   ^
certian 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.

For your case 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 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);
```

How can I model variable values external to my application?

There are three main considerations.

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

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

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

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

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

Note that although the volatile characteristic of a variable is also commonly used by programmers to avoid compiler optimisation, 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 “Basics: Prerequisites to Reviewing PolySpace™ Results” on page 8-2 for more information on color propagation.

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

Volatile.

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

Code Preparation for Built-in Functions

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

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

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

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

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()`.

My Code is Multitasking

We strongly recommend to read the different section contained here before applying the rules described below. Some rules are mandatory; some rules allow the user to gain selectivity.

The following describes the default behavior of PolySpace Verifier. If the code to be analyzed does not conform to these assumptions, then some minor modifications to the code will be required.

- 1 The main procedure must terminate in order for entry-points (or tasks) to start.
- 2 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 has dead code, it is because the main contains a red error and therefore does not terminate.

PolySpace Verifier considers that there is no atomicity, nor timing constraints.

At last, only entry point with `void any_name (void)` as prototype will be considered.

This section contains the following topics:

- “Modelling Tasks, Interruptions and Events” on page 3-70
- “Shared Variables” on page 3-77
- “Miscellaneous” on page 3-81

Modelling Tasks, Interruptions and Events

- “Modelling Synchronous Tasks” on page 3-70
- “Interruptions and Asynchronous Events/Tasks/Threads” on page 3-73
- “Are Interruptions Maskable or Preemptive by Default?” on page 3-75

Modelling Synchronous Tasks. It will sometimes be necessary to adapt the 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*/
}
```

Now, if each entry-point was to be specified at Verifier launch by using the option

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

then the results would NOT be valid, because each task would only be called once.

To address this problem, PolySpace Verifier needs to be informed that the tasks are purely sequential - that is, that they are functions to be called in a deterministic order. This can be achieved by writing a function to call each of the tasks in the correct sequence, and then declaring this new function as a single task entry point.

Solution 1

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 Verifier 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.

Interruptions and Asynchronous Events/Tasks/Threads. Source code may be adapted to allow *asynchronoustasks* and *interruptions* to be taken into account; for example:

```
void interrupt isr_1(void)
{ ... }
```

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

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 pre-empt 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 pre-emptive 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 Verifier 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 Verifier 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 non-maskable 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 Verifier

```
#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 launching 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™ Verifier 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 mechanisms can be used to protect the variables:

- two explicit protection mechanisms (critical section and mutual exclusion);
- implicit protection (access pattern).

See details below:

- “Differences Between Dictionary and Concurrent Access Graph” on page 3-77
- “Critical Sections” on page 3-78
- “Mutual Exclusion” on page 3-80
- “Access Pattern” on page 3-80
- “Semaphores” on page 3-81

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

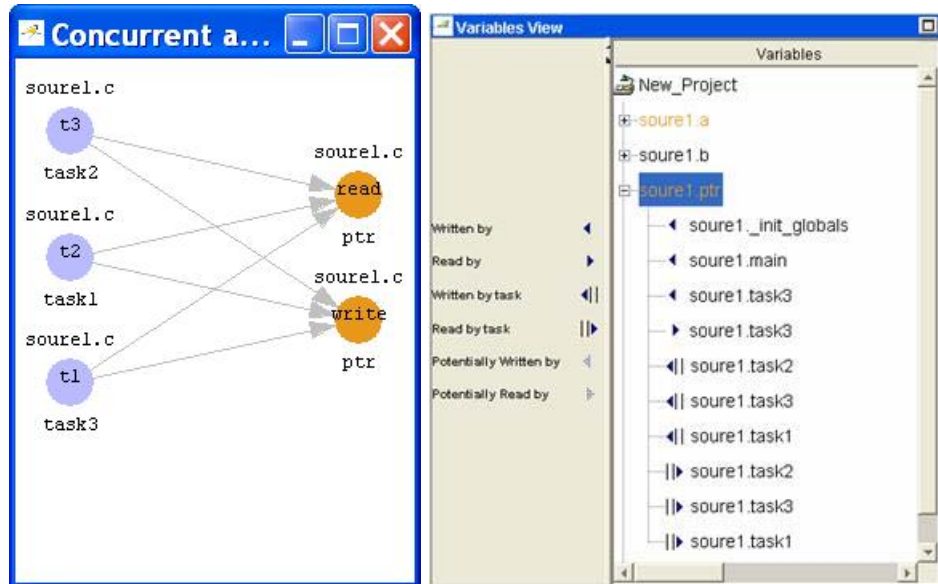
<pre>int *ptr; int a; int b; void main(void) { ptr = &a; }</pre>	<pre>void task1(void) {a ++;</pre>	<pre>void task2(void) { a = a + 10; }</pre>	<pre>void task3(void) { ptr = &b; *ptr = 0; }</pre>
--	--------------------------------------	---	---

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 Verifier:

- 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 Verifier

```
polyspace-c \
  -entry-point proc1,proc2 \
  -critical-section-begin"begin_cs:label1_1" \
```

```
-critical-section-end"end_cs:label1_1"
```

Mutual Exclusion. Mutual exclusion between tasks or interrupts can be implemented while preparing PolySpace Verifier for launch setting.

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

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

These entry-points cannot overlap:

- t1 and t3
- t2, t3 and t4

These entry-points can overlap:

- t1 and t2
- t1 and t4

Before launching Verifier, 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 myExclusions.txt is also required in the current directory. This will contain:

```
t1 t3  
t2 t3 t4
```

Access Pattern. If a variable is a structure, then provided the same fields aren't being accessed, by its nature the variable is protected even if different tasks are accessing it. In PolySpace, this is regarded as protection by "access pattern" which will be shown in the Shared Variables section of the Viewer.

Consider the following example.

If a variable `x`, is a structure containing two fields, `A` and `B`, and

- `task_1` only reads/writes field `A`
- `task_2` only reads/writes field `B`

Then `x` is shown as being protected by access pattern in PolySpace Viewer.

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

Miscellaneous

- “Mailboxes” on page 3-81
- “Atomicity (Can an Instruction be Interrupted by Another)” on page 3-84
- “Priorities” on page 3-85

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 analysis is to be meaningful.

By default, PolySpace Verifier 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 Verifier 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 Verifier behavior 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 Verifier 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 Verifier 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 Verifier considers write/read instructions atomic, so **task 2 can only read 0xFF55**, even if X is not protected (refer to “Shared Variables” on page 3-77).

Critical sections

In terms of critical sections, PolySpace Verifier 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 Verifiers analysis of Run Time Errors (RTE) supposes that there was no conflict when writing the shared variables. Hence even if a shared variable is not protected, the RTE analysis is complete and correct.

More information is available in “Critical Sections” on page 3-78.

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

In practice, while there is no facility to specify differing task priorities, all priorities **are** taken into account because of the default behavior of PolySpace Verifier 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 Verifier 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.

PolySpace™ Software Day to Day Usage

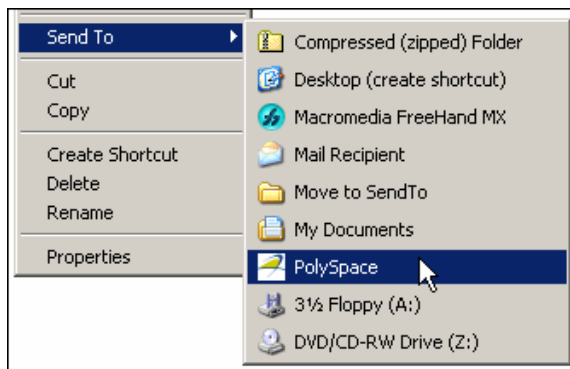
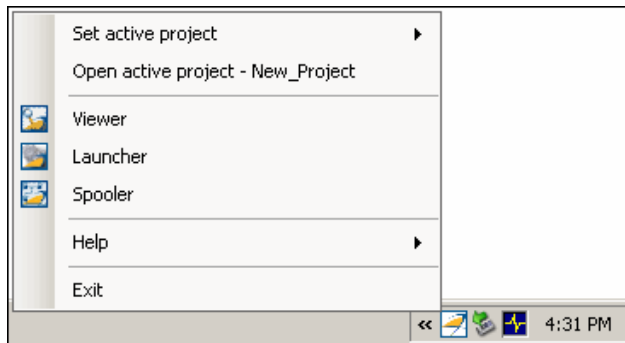
PolySpace™ In One Click Overview (p. 4-2)	Provides an overview of the PolySpace™ In One Click plug-in
Using PolySpace™ In One Click (p. 4-3)	Describes how to use PolySpace In One Click
Using Right-Click to Launch PolySpace™ Verification (p. 4-6)	Describes how to start a PolySpace verification using right-click

PolySpace™ In One Click Overview

The PolySpace™ In One Click plug-in has been specifically designed for developers. Since developers tend to work on the same project over time (new code, unit tests, integration), they often need the same options for multiple files in their project. PolySpace In One Click allows you to easily launch verification of multiple files in the same project (or other files of the same application) using the same set of options.

Once you set up your preferred options, PolySpace In One Click uses them for as many files as the project holds, without you having to continuously update them. Launching an analysis is then just a matter of clicking.

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...
“Overview” on page 4-3
“Creating an Active Configuration File Project” on page 4-3
“Using the TaskBar Icon” on page 4-3

Overview

Usage consists in launching analysis through out an active PolySpace™ project.

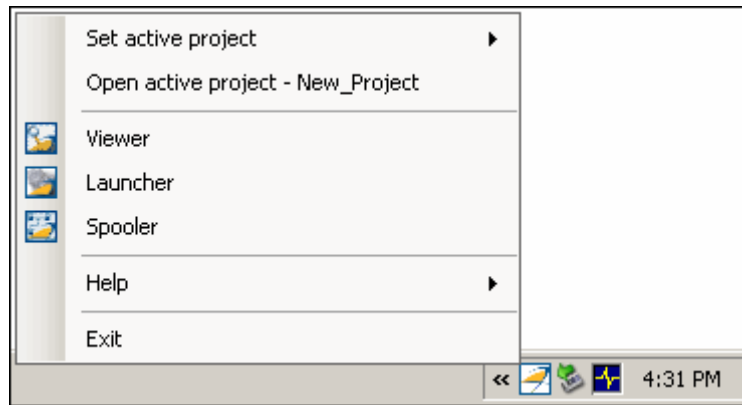
Creating an Active Configuration File Project

In each PolySpace product you can find a PolySpace configuration file (associated to each language) which can be copied in your own directory and can serve as a basis for your own active configuration project file. One configuration file can found for each language located in <PolySpace Install Products>\Examples\Demo_<Language> folder.

You can also use the PolySpace Launcher added on your desktop windows during installation to create a new active project.

Using the TaskBar Icon

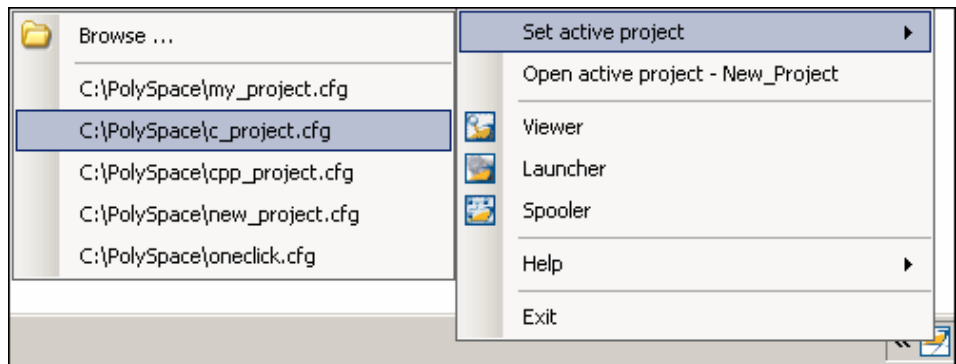
For a new project, choose an active PolySpace configuration project file, with a *.cfg* (a Verifier Configuration file) or *.dsk* (a Desktop Configuration file) extension. Some common options will be set up in this file, and all further launching analysis will use this active set of options.



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 By default there is no selected configuration file. You can create an empty file with a .cfg or .dsk extension.

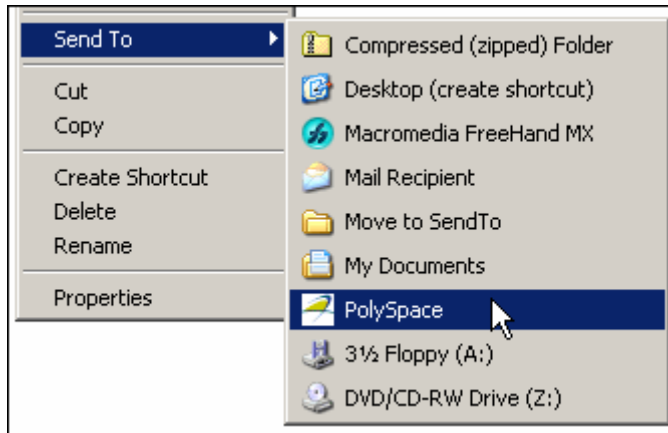
- **Open active project** — Opens the active configuration file. This allows you to update the project using the standard PolySpace Launcher graphical interface. It allows you to specify all PolySpace common options, including directives of compilation, options, and path of standard and specific headers. It does not affect the precision of an analysis and the results directory.

The active configuration file can be updated every time you consider that a coding file is part of the whole analysis.

- **Viewer** — Opens the PolySpace viewer. This allows you to see verification results in the standard graphical interface. By default, the viewer opens results in the results directory, which is specified in the “Analysis Parameters” dialog box.
- **Launcher** — Opens the PolySpace Launcher. This allows you to launch an analysis using the standard PolySpace graphical interface.
- **Spooler** — Opens the PolySpace Spooler. If you selected a server analysis in the “PolySpace Preferences” dialog box, the spooler allows you to follow the status of the analysis.

Using Right-Click to Launch PolySpace™ Verification

Once you have set your active configuration file, you can also right click on one or more files to start an analysis using the configuration file settings.



Note The **Send To** shortcut calls
<PolySpaceCommon>\PolySpaceInOneClick\PolySpaceInOneClick.exe -f.

MISRA[®] Checker

PolySpace [™] MISRA [®] Checker Overview (p. 5-2)	Describes the PolySpace MISRA [®] checker
Rules Supported (p. 5-4)	Describes the MISRA rules supported by PolySpace [™] software
Rules Partially Supported (p. 5-33)	Describes MISRA rules partially supported by PolySpace software
Rules Not Checked (p. 5-45)	Describes MISRA rules not checked by PolySpace software

PolySpace™ MISRA® Checker Overview

The PolySpace™ MISRA® checker helps developers achieves MISRA compliance. The PolySpace MISRA checker is based on MISRA C®:2004, (<http://www.misra-c.com>) enabling PolySpace software to provide messages at compile phase (mainly) when rules are not respected.

Only two options `-misra2` and `-includes-to-ignore`, permit you to enable and verify C code sources on nearly all of the **141** rules, part of MISRA C:2004 and one rule (named 15.0) implemented by PolySpace software (this rule is described in the MISRA C:2004 manual about switch statements). “Setting Up and Launching the MISRA C® Checker” on page 2-50 explains how to set up the MISRA checker from a graphical point of view using PolySpace Launcher.

Theses 142 rules are divided in three categories:

- **102** required and advisory rules fully supported. PolySpace software can check all theses rules without any limitations. See “Rules Supported” on page 5-4.
- **20** required and advisory rules partially supported. PolySpace software can check all theses rules with some limitations. Theses limitations are described in the associated “**Note**” paragraph for each rule. See “Rules Partially Supported” on page 5-33.
- **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. Theses rules are not checked. The “**comment**” column details the reason. See “Rules Not Checked” on page 5-45.

Note Every violation, warning or error, will be written in the log file at compilation time of a PolySpace analysis, except for rules 9.1 (NIV checks), 12.11 (OVFL check using `-detect-unsigned-overflows`), 13.7 (grey checks), 14.1 (grey 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 Oranges” on page 9-87 which can have direct or indirect impact on the PolySpace selectivity (reliability percentage).

Rules Supported

In this section...
“Language Extensions” on page 5-5
“Character Sets” on page 5-5
“Identifiers” on page 5-6
“Types” on page 5-7
“Constants” on page 5-8
“Declarations and Definitions” on page 5-9
“Initialization” on page 5-11
“Arithmetic Type Conversion” on page 5-12
“Pointer Type Conversion” on page 5-16
“Expressions” on page 5-17
“Control Statement Expressions” on page 5-20
“Control Flow” on page 5-21
“Switch Statements” on page 5-23
“Functions” on page 5-24
“Pointers and Arrays” on page 5-25
“Structures and Unions” on page 5-25
“Preprocessing Directives” on page 5-26
“Standard Libraries” on page 5-30
“Run-Time Failures” on page 5-32

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

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
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.

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
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 -permissive-link 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 -allow-undef-variables 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 -main-generator 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 Verifier (NIV Checks). Cannot be Off.
9.2	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	Braces shall be used to indicate and match the structure in the non-zero 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 • the expression is not constant and is a return expression 	<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 operand of underlying type ?? to ?? that is not an integer type. • 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. 	<ol style="list-style-type: none"> 1 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. 2 An expression of bool or enum types has int as underlying type. 3 Plain char may have signed or unsigned underlying type (depending on PolySpace target configuration or option setting). 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 token 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
10.1 (cont.)		<ul style="list-style-type: none"><li data-bbox="614 336 954 458">• Implicit conversion of complex integer expression of underlying type ?? to ??.<li data-bbox="614 479 954 666">• Implicit conversion of non-constant integer expression of underlying type ?? in function return whose expected type is ??.<li data-bbox="614 687 954 904">• Implicit conversion of non-constant integer expression of underlying type ?? as argument of function whose corresponding parameter type is ??.	

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
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 corresponding parameter type is ??. 	<p>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.</p>

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
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 Polyspace Verifier 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 token 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 pplied 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 Cs operator precedence rules in expressions	Limited dependence should be placed on Cs 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 Verifier (grey 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 (grey 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 C[®]2004 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 grey 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 parenthesised 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 pre-processors directives or comments	A message is displayed when a #include directive is preceded by other things than pre-processor directives, comments, spaces or “newlines”.	
19.2	Non-standard 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 <i>#include</i> 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 #define'd 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 pre-processing 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 ## pre-processor operators in a single macro definition.	More than one occurrence of the # or ## preprocessor operators.	
19.13	The # and ## pre-processor operators should not be used	Message on definitions of macros using # or ## operators	
19.14	The defined pre-processor operator shall only be used in one of the two standard forms.	'defined' without an identifier.	Cannot be Off.

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
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 '<code><name></code>' shall not be redefined. • The macro '<code><name></code>' 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 '<code><name></code>' 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 <code>errno</code> shall not be used	The error indicator <code>errno</code> shall not be used	Assumes that rule 20.2 is not violated
20.6	The macro <code>offsetof</code> , in library <code><stddef.h></code> , shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' 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 '<i><name></i>' shall not be used. • Identifier <i>XX</i> should not be used. 	In case the <i>longjmp</i> 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 <i><signal.h></i> shall not be used.	<ul style="list-style-type: none"> • The macro '<i><name></i>' shall not be used. • Identifier <i>XX</i> 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 <i><stdio.h></i> shall not be used in production code.	<ul style="list-style-type: none"> • The macro '<i><name></i>' shall not be used. • Identifier <i>XX</i> 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 <i>atof</i> , <i>atoi</i> and <i>toll</i> from library <i><stdlib.h></i> shall not be used.	<ul style="list-style-type: none"> • The macro '<i><name></i>' shall not be used. • Identifier <i>XX</i> should not be used. 	In case the <i>atof</i> , <i>atoi</i> and <i>atoll</i> 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

Run-Time Failures

N.	MISRA Definition	Messages in log file	Detailed PolySpace Specification
21.1	Minimisation of run-time failures shall be ensured by the use of at least one of: <ul style="list-style-type: none"> • static analysis tools/techniques; • dynamic analysis tools/techniques; • explicit coding of checks to handle run-time faults. 		Done by PolySpace Verifier (Run-time error checks). Cannot be Off.

Rules Partially Supported

In this section...
“Environment” on page 5-33
“Language Extension” on page 5-35
“Identifier” on page 5-35
“Declarations and Definitions” on page 5-36
“Expressions” on page 5-37
“Control Statement Expressions” on page 5-38
“Control Flow” on page 5-40
“Switch Statements” on page 5-40
“Functions” on page 5-41
“Pointers and Arrays” on page 5-42
“Preprocessing Directives” on page 5-43

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.

Rule	Description
	<p data-bbox="393 302 598 331">Messages in log:</p> <ul data-bbox="393 366 1161 1164" style="list-style-type: none"><li data-bbox="393 366 911 395">• ANSI® C does not allow ‘#include_next’<li data-bbox="393 413 1161 442">• ANSI C does not allow macros with variable arguments list<li data-bbox="393 460 817 489">• ANSI C does not allow ‘#assert’<li data-bbox="393 506 839 536">• ANSI C does not allow ‘#unassert’<li data-bbox="393 553 933 583">• ANSI C does not allow testing assertions<li data-bbox="393 600 802 630">• ANSI C does not allow ‘#ident’<li data-bbox="393 647 788 677">• ANSI C does not allow ‘#sccs’<li data-bbox="393 694 980 723">• text following ‘#else’ violates ANSI standard.<li data-bbox="393 741 998 770">• text following ‘#endif’ violates ANSI standard.<li data-bbox="393 788 1110 817">• text following ‘#else’ or ‘#endif’ violates ANSI standard.<li data-bbox="393 835 887 864">• ANSI C90 forbids ‘long long int’ type.<li data-bbox="393 881 874 911">• ANSI C90 forbids ‘long double’ type.<li data-bbox="393 928 991 958">• ANSI C90 forbids long long integer constants.<li data-bbox="393 975 872 1005">• Keyword ‘inline’ should not be used.<li data-bbox="393 1022 888 1052">• Array of zero size should not be used.<li data-bbox="393 1069 1092 1098">• Integer constant does not fit within unsigned long int.<li data-bbox="393 1116 973 1145">• Integer constant does not fit within long int.
	<hr/> <p data-bbox="393 1246 1319 1373">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.
Message in log: <ul style="list-style-type: none"> • Assembly language shall be encapsulated and isolated. 	
<hr/> <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> <hr/>	

Identifier

Rule	Description
5.1 (Required)	Identifiers (internal and external) shall not rely on the significance of more than 31 characters
Message in log: <ul style="list-style-type: none"> • Identifier 'XX' should not rely on the significance of more than 31 characters. 	
<hr/> <p>Note Only global variables (external linkage) are checked. Can be turned to Off</p> <hr/>	

Declarations and Definitions

Rule	Description
8.3 (Required)	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
Message in log: <ul style="list-style-type: none"> • Definition of function 'XX' incompatible with its declaration. 	
Note Assumes that rule 8.1 is not violated. The rule is restricted to compatible types. Can be turned to Off	
8.7 (Required)	Objects shall be defined at block scope if they are only accessed from within a single function
Message in log: <ul style="list-style-type: none"> • Object 'XX' should be declared at block scope. 	
Note Restricted to static objects. Can be turned to Off	
8.8 (Required)	An external object or function shall be declared in one file and only one file
Message in log: <ul style="list-style-type: none"> • Function/Object 'XX' has external declarations in multiples files. 	
Note Restricted to explicit extern declarations (tentative of definitions are ignored). Can be turned to Off	

Expressions

Rule	Description
12.2 (Required)	The value of an expression shall be the same under any order of evaluation that the standard permits.
<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>	
12.11 (Advisory)	Evaluation of constant unsigned expression should not lead to wrap-around.
<p>No message.</p>	
<p>Note This rule is partially implemented with <code>-detect-unsigned-overflow</code> option in PolySpace™ software. Concerning possible pre-processing overflows, PolySpace pre-processor does not take into account target basic types and considers always 32-Bit long int. Cannot be ticked.</p>	
12.12 (Required)	The underlying bit representations of floating-point values shall not be used.

Rule	Description
	<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 void *). Can be turned to Off.</p>

Control Statement Expressions

Rule	Description
<p>13.3 (Required)</p>	<p>Floating-point expressions shall not be tested for equality or inequality.</p>
	<p>Message in log:</p> <ul style="list-style-type: none"> Floating-point expressions shall not be tested for equality or inequality.
	<p>Note Warning on direct tests only. Can be turned to Off.</p>
<p>13.4 (Required)</p>	<p>The controlling expression of a <i>for</i> statement shall not contain any objects of floating type</p>
	<p>Message in log:</p> <ul style="list-style-type: none"> The controlling expression of a <i>for</i> statement shall not contain any objects of floating type
	<p>Note If <i>for</i> index is a variable symbol, checked that it is not a float. Can be turned to Off.</p>

Rule	Description
13.5 (Required)	The three expressions of a <i>for</i> statement shall be concerned only with loop control
<p>Messages in log:</p> <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). 	
<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>	
13.6 (Required)	Numeric variables being used within a <i>for</i> loop for iteration counting should not be modified in the body of the loop.
<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
14.3 (Required)	<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
<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>	

Switch Statements

Rule	Description
15.0 (Advisory)	Misra Switch syntax rules

Rule	Description
	<p>Message in log:</p> <ul style="list-style-type: none"> • switch statements syntax normative restriction
	<p>Note Warning on declarations or instructions before the first switch case. PolySpace checks that if declarations or statements are put between the switch() and first case keyword.</p> <p>This rule is a clearly advisory made by MISRA C® consortium.</p> <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 initialiser, a constant, a parenthesised 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 parenthesised 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="427 505 768 626">#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 5-45
“Language Extensions” on page 5-46
“Documentation” on page 5-47
“Types” on page 5-48
“Functions” on page 5-48
“Pointers and Arrays” on page 5-48
“Structures and Unions” on page 5-49
“Standard Libraries” on page 5-49

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. 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)	<p>Signed and unsigned char type shall be used only for the storage and use of numeric values.</p> <hr/> <p>Note this rule is partially implemented in Rule 6.1.</p> <hr/>	<p>Consider an external function returning a char is been used and increased. There is no mean without the functional knowledge that this function stores a character value or not.</p>

Functions

Rule	Description	Comments
16.7 (Advisory)	<p>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.</p>	<p>Not statically checkable unless the pointer analysis has been done.</p>
16.10 (Required)	<p>If a function returns error information, then that error information shall be tested.</p>	<p>Not statically checkable unless type defining error is standardized.</p>

Pointers and Arrays

Rule	Description	Comments
17.1 (Required)	<p>Pointer arithmetic shall only be applied to pointers that address an array or array element.</p>	<p>Not statically checkable unless the pointer analysis has been done</p>

Rule	Description	Comments
17.2 (Required)	Pointer subtraction shall only be applied to pointers that address elements of the same array.	Not statically checkable unless the pointer analysis 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 analysis 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

Data Range Specifications

Overview (p. 6-2)	Provides an overview of data range specifications (DRS)
File Format (p. 6-3)	Describes the file format for data range specifications
Variable Scope (p. 6-5)	Describes the scope of data range variables
Reduce Oranges with DRS (p. 6-7)	Describes how to use data range specification to reduce orange checks

Overview

The PolySpace™ Data Range Specifications (DRS) is an easy to use module that helps developers achieves external constraints on global variables without intrusion.

The associated option `-data-range-specification` option permits the setting of specific data ranges for a list of given global variables. The point during the analysis at which the range is applied to a variable is controlled by one of the following mode keyword: `init`, `permanentand` `globalassert`. The option is protected by a license.

- `<filename>` specifies the list global variables involved in the setting of specific data ranges (See next section: “File format”).
- Only variables concerned by external linkage can benefit from the data range setting (See next section: Variables scope).

File Format

Added to `-data-range-specification` option the file filename contains a list of global variables with the below format:

```
variable_name val_min val_max <init|permanent|globalassert>
```

- Keyword `init`: variable is assigned to specific range, only at initialization and keeps it until first write.
- Keyword `permanent`: variable is permanently assigned to specific range. If the variable is assigned outside the specified range during the program no warning is provided. Use the `globalassert` mode if you need a warning.
- Keyword `globalassert`: after each assignment an assert check is performed, controlling the specified range. The assert check is also performed at global initialization.
- Values `val_min` and `val_max` could be replaced by the keywords "min" or "max" to denote the minimum and maximum values of the variable type. Example for the long type: min and max correspond to -2^{31} and $2^{31}-1$ respectively.
- Hexadecimal values can be used: `x 0x12 0x100 init`
- Allowed column separators are: tab, comma, space or semi-column.
- To insert comments use shell style "#".

Example (x, y, z, w, array and v are the name of global variables):

```
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
```

Variable Scope

Variables concern external linkage, const variables and defined variables. It could be extern variables with option `-allow-undef-variables`.

Static variables are not concerned by this option. The following table summarizes possible uses:

	init	permanent	globalassert	comments
Integer	Ok	Ok	Ok	char, short, int, enum, long and long long
Reals	Ok	Ok	Ok	float, double and long double
Volatile	No effect	OK	Full range	Only for integer and reals
Structure field	Ok	No effect	Ok	Only for integer and reals fields.
Structure field in array	Ok	No effect	No effect	Only when leaves are "integer" or reals. Moreover the syntax is the following: <array_name>. <field_name>
Array	Ok	Ok	Ok	Only for integer and reals
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] ...

Note Every variable and associated data range will be written in the log file at compilation time of a PolySpace™ analysis. If PolySpace software does not support the variable, a warning message is displayed.

Reduce Oranges with DRS

In this section...
“Perform Efficient Module Testing” on page 6-7
“Reduce Oranges with the —data-range-specification option” on page 6-8

Perform Efficient Module Testing

The data-range-specification add-on can be used 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 characteristic

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

The PolySpace™ feature enables the user to define

- What is the nominal range for input data
- What is the expected range for output data
- What is the generic specified range for calibrations

It allows making one unique static analysis and performing two simultaneous tasks

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

In that context, several options have to be selected according to the type of data, whether they are input, outputs, or calibrations.

Type of Data	DRS Mode	Effect on Results	Why?	Oranges	Selectivity
input (entries)	permanent	Reduces the number of oranges, (compared with a standard Desktop analysis)	Input data which are full range with Desktop are now set to a smaller range with this option	↓	↑
Outputs	globalassert	Increases the number of oranges	More verification are introduced into the code, which means more checks orange and more green ones	↑	→
Calibration	init	Increases the number of oranges, (compared with a standard Desktop analysis)	Data which are constant with Desktop are now set to a wider range with this option	↑	↓

Now there is a derivate and specific usage of DRS which is to only focus on reducing oranges. A detailed explanation on how that can be accomplished is given in the next section

Reduce Oranges with the `–data-range-specification` option

When verifying worst case robustness with PolySpace Desktop, data inputs are set to their full range. Therefore, every operation on these inputs, even a simple “one_input + 10” might produce an overflow, as the range of one_input varies between the min and the max of the type.

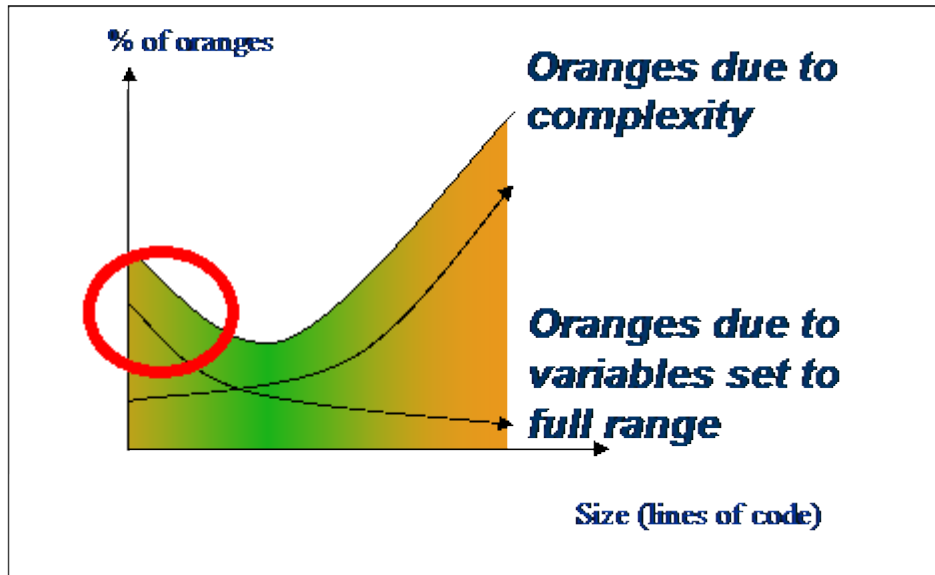
By the use of DRS to restrict the range of “one-input” to its real functional constraints found in the specification, design documents or models - lets say “one-input” can vary between 0 and 10 - PolySpace software will definitely know that

- “one_input + 100” will never overflow

- the results will be between 100 and 110

This not only suppresses the local overflow orange, but also injects more accuracy in the data which is propagated through the rest of the code.

It removes the oranges located in the red circle below.



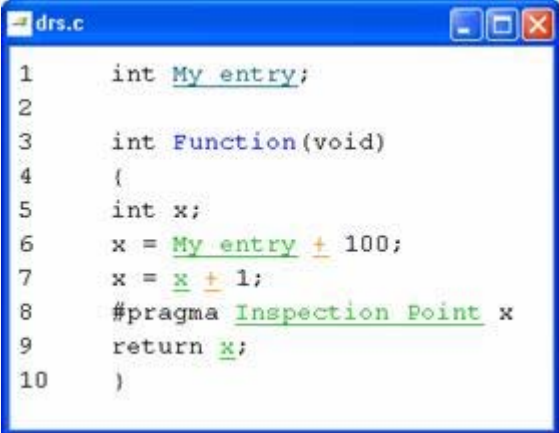
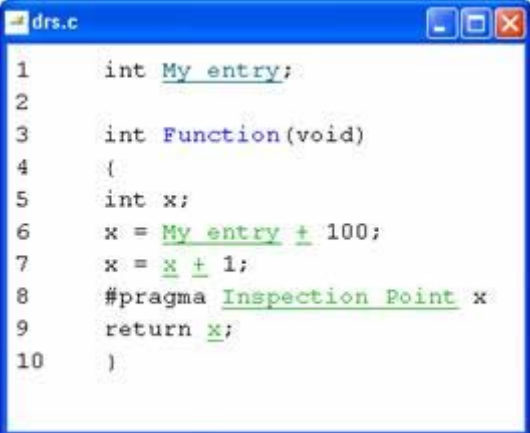
Why Only on Modules

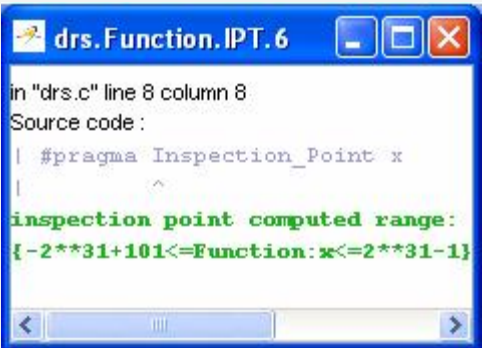
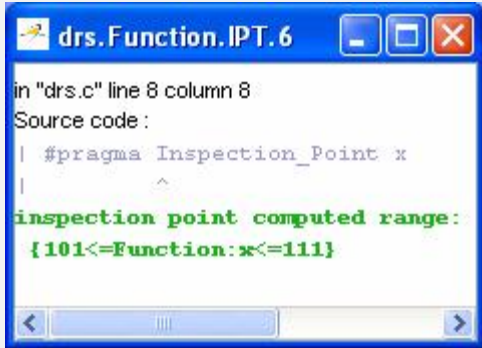
By removing the oranges introduced because data is set to it is worst case, the orange decreases drastically, especially when used on units constituted of small files or modules. We would not explain in this section why the number of orange due to complexity can largely and negatively destroy the reduction of the number of oranges introduced thanks to DRS, as this is covered in the documentation, both in “Why Should there be an Optimum Size?” on page 9-71 and “Considering the effects of application code size.”

We will only explain here how DRS can reduce oranges on file or modules only.

Example

Lets prove it by considering a simple example. Well suppose that the input called “My_entry” can vary - in the real world - vary between 0 and 10. Two analysis can be performed; One with data-range-specification (DRS), 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 [010], the addition “+” is green • the result “x” is equal to [100110] • 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]
 <p>The screenshot shows a debugger window titled "drs.Function.IPT.6". It displays the source code for a function with an inspection point. The output indicates the computed range for the function is $\{-2^{31}+101 \leq \text{Function}:x \leq 2^{31}-1\}$.</p>	 <p>The screenshot shows a debugger window titled "drs.Function.IPT.6". It displays the source code for a function with an inspection point. The output indicates the computed range for the function is $\{101 \leq \text{Function}:x \leq 111\}$.</p>

Using PolySpace™ Model Link Products

Overview of PolySpace™ Model Link Products (p. 7-2)	Provides an overview of the PolySpace™ Model Link™ SL and PolySpace Model Link TL products
Getting Started (p. 7-3)	Describes how to use PolySpace Model Link SL or PolySpace Model Link TL products
Advanced Setup (p. 7-26)	Describes how to configure advance options
PolySpace™ Utilities (p. 7-35)	Describes the blocks in the PolySpace Utilities library
Code Generator Specific Information (p. 7-43)	Provides information specific to the PolySpace Model Link SL or PolySpace Model Link TL code generators

Overview of PolySpace™ Model Link Products

This chapter describes how to use PolySpace™ for Model-Based Design. The PolySpace Model Link™ SL and PolySpace Model Link TL products allow you to launch a PolySpace C verification from a Simulink® model associated with Real-Time Workshop® Embedded Coder™ software, or dSPACE® TargetLink® software.

PolySpace Model Link SL and PolySpace Model Link TL products provide automatic error detection for code generated from Simulink models. It consists of two principal components:

- A Simulink PolySpace library with associated blocks.
- A “Back to model” extension in the PolySpace Viewer that allows direct navigation from a run-time error in the auto-generated code to the corresponding block in the Simulink model.

Getting Started

In this section...
“Overview” on page 7-3
“Creating a Simulink® Model and Generating Production Code” on page 7-3
“Starting the PolySpace™ Analysis” on page 7-9
“Fixing an Error in the Design and the Simulink® Model” on page 7-13
“Base Workspace vs. PolySpace™ Data Ranges” on page 7-18

Overview

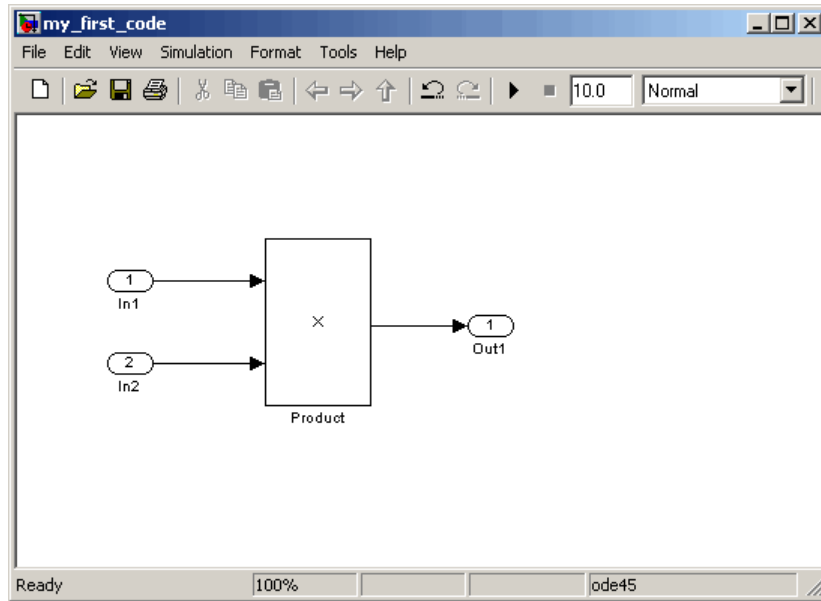
This consists of several steps, all detailed in this section

- Create a Simulink® model and generate production code (For more information, see the *Real-Time Workshop® Embedded Coder™ Getting Started Guide*)
- Start the PolySpace™ analysis

Creating a Simulink® Model and Generating Production Code

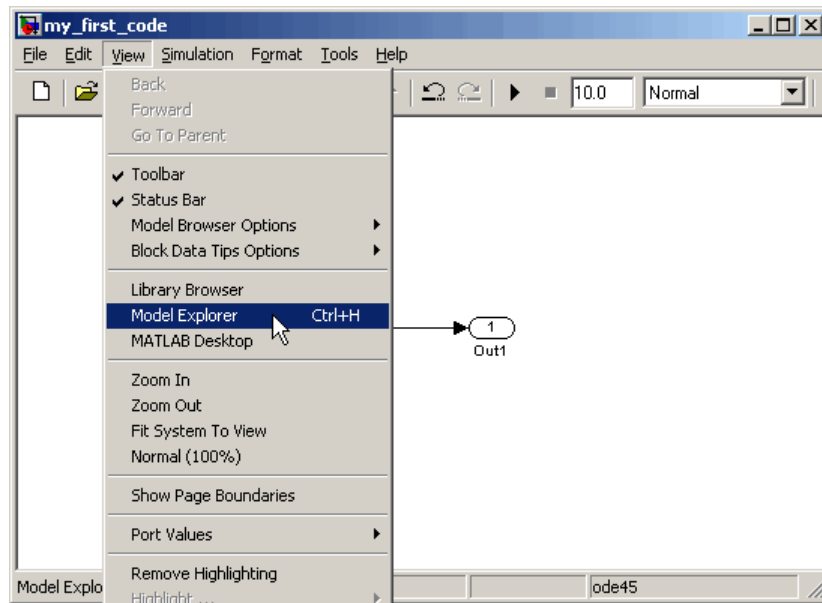
To create a Simulink model and generate production code:

- 1 Open MATLAB®, then start Simulink software.
- 2 Create a Simulink model, similar to the one below.



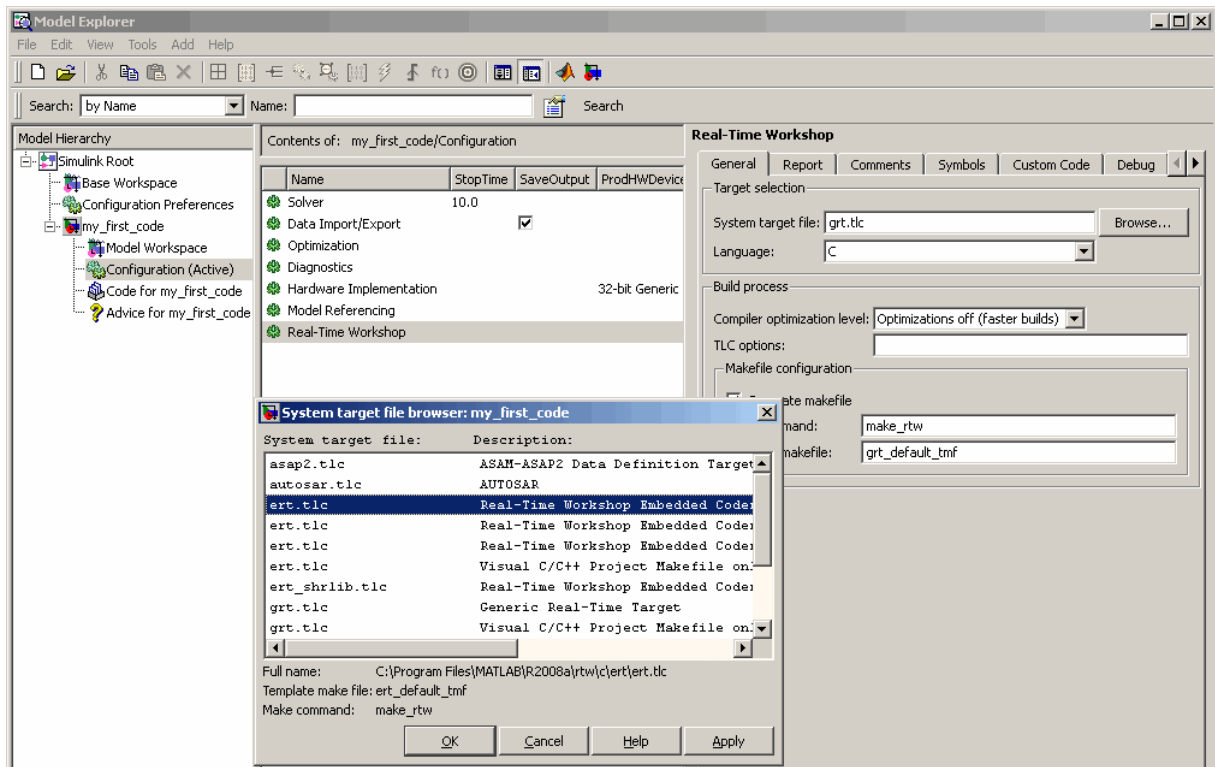
Create the my_first_code model

- 3 Use the model explorer to edit the current configuration and set it, for example, to Real-Time Workshop Embedded Coder software.



Open the model explorer

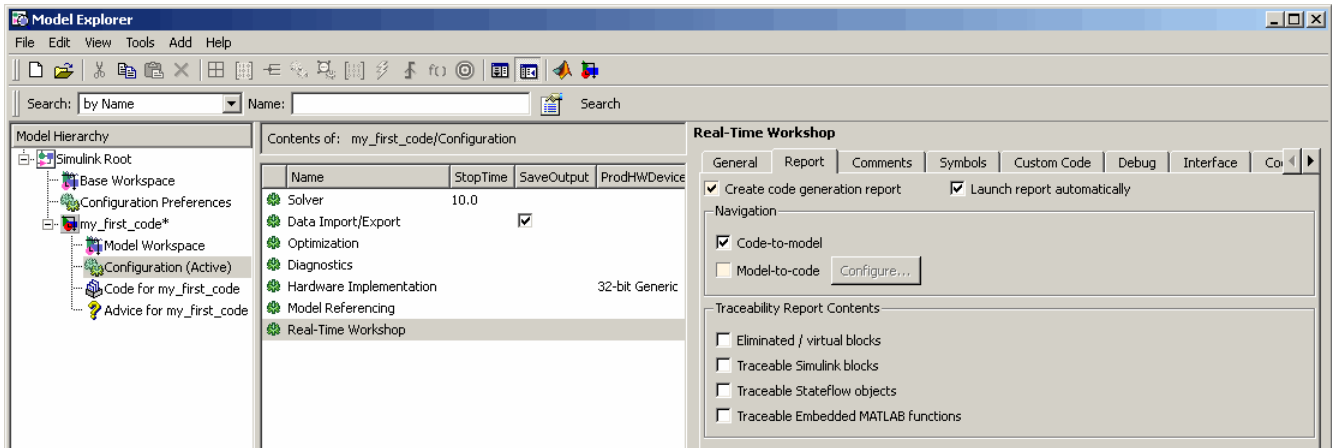
- 4 Set the System target file to Real Time Workshop Embedded Coder: ert.tlc.



Change the code generator to Real-Time Workshop® Embedded Coder™ software

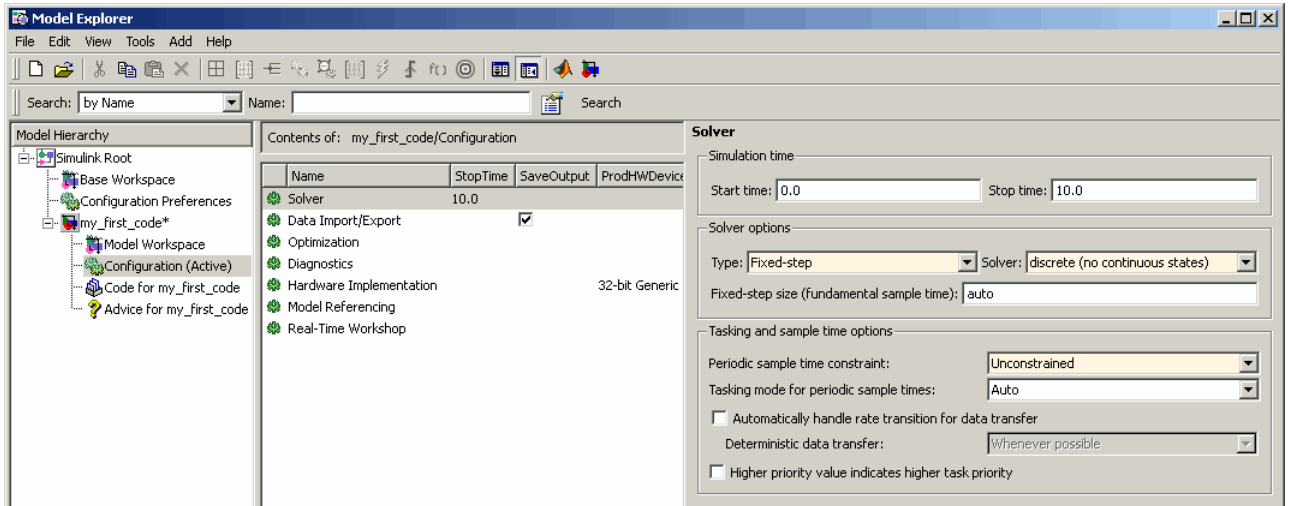
5 Select the **Report** tab.

6 Select **Create code-generation report**, and select **Code-to-model Navigation**.



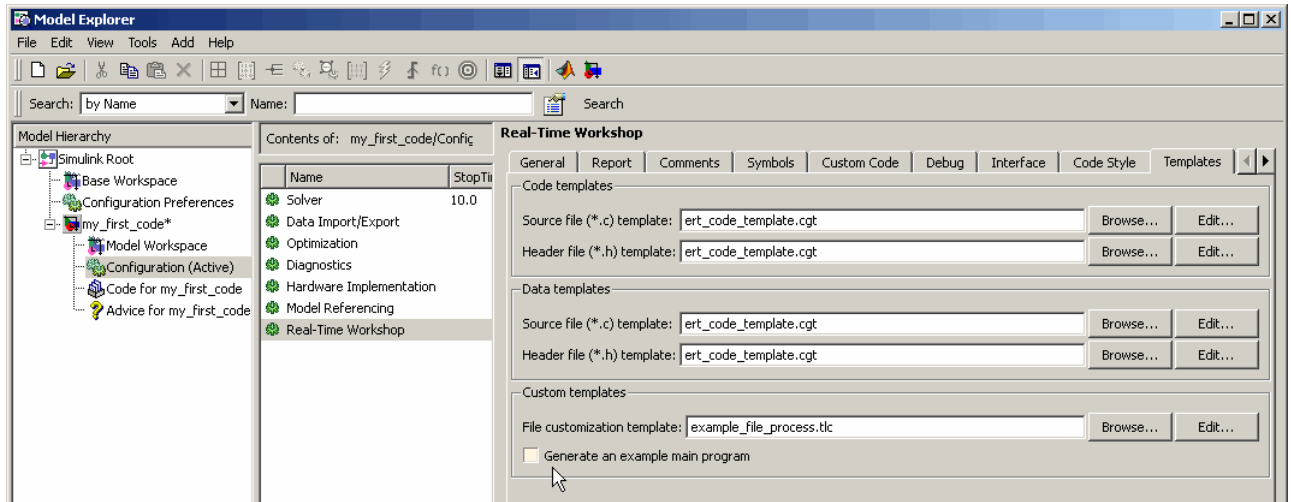
Detail of the model explorer, enabling html generation

- 7 Select the **Solver** tab, then set the solver **Type** to Fixed-step, and the **Solver** to discrete (no continuous states).



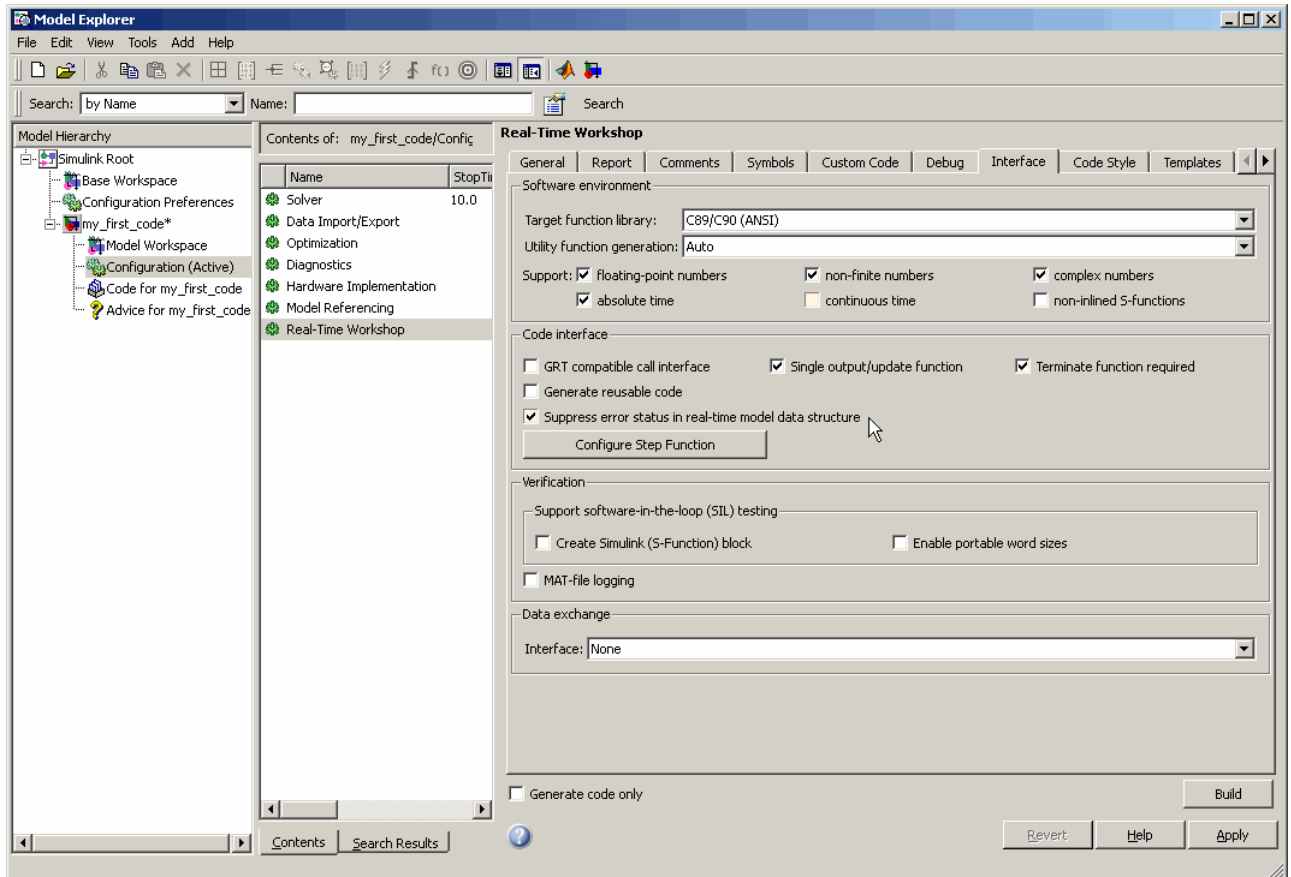
Choose fixed-step type

- 8 Select the **Templates** tab, then disable **Generate an example main program**.



Templates tab in the model explorer

- 9 Select the **Interface** tab, then enable **suppress error status in real-time model data structure**.



Interface tab in the model explorer

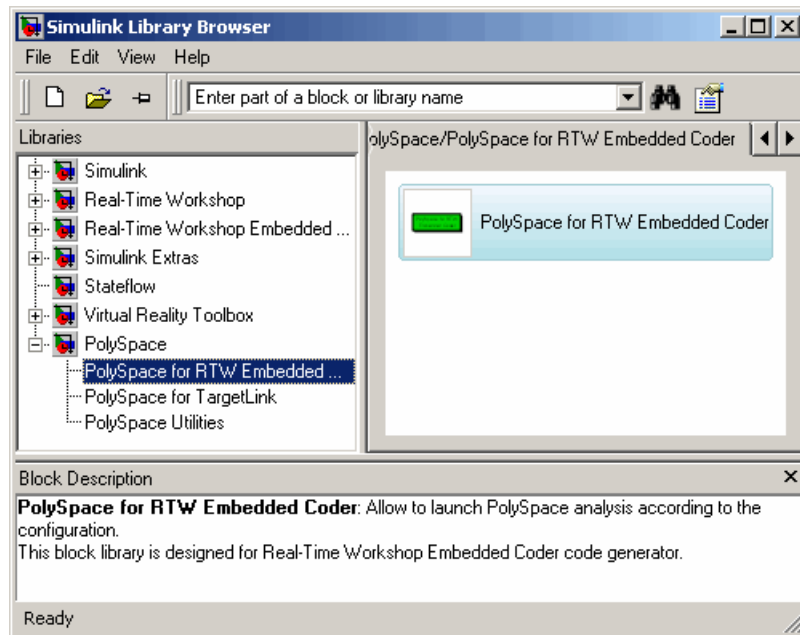
- 10 Click **Apply**.
- 11 Generate the production code.

Starting the PolySpace™ Analysis

To Start the PolySpace analysis:

- 1 Open the Simulink library browser.
- 2 Locate the PolySpace library and expand it.

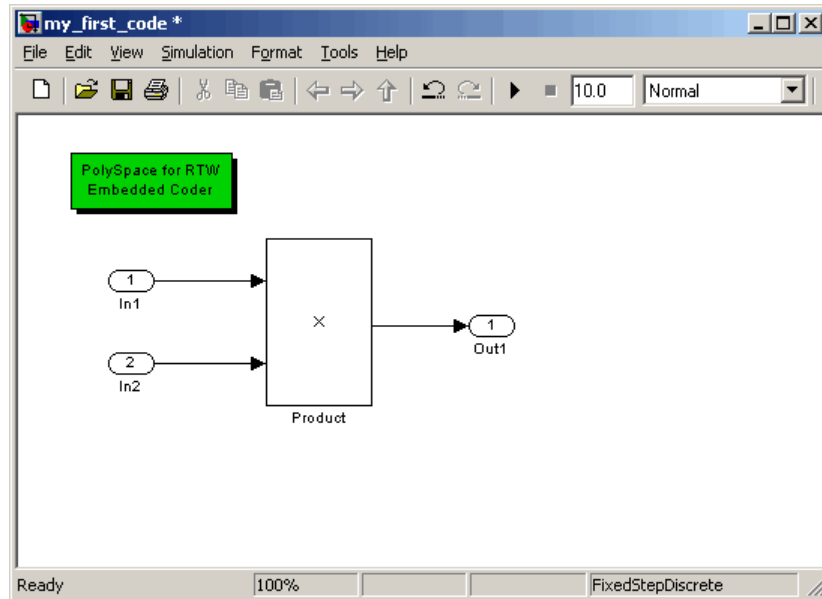
The Simulink Library Browser is updated with the PolySpace Library during the installation process. The library contains a common part called PolySpace Utilities, and sections for each of the installed code generators called PolySpace For “Code Generator” (see figure).



Simulink® Library Browser

- 3 Drag and drop the “PolySpace for RTW Embedded Coder” block.

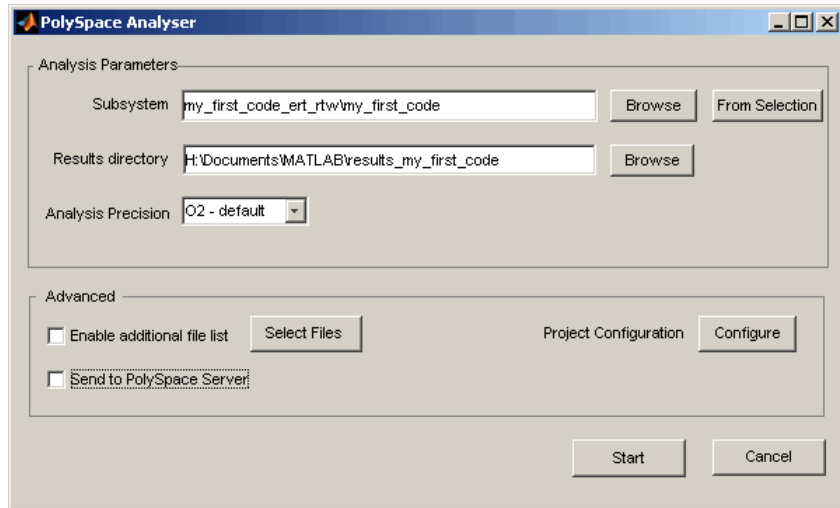
To perform an analysis of your code using the default settings drag the PolySpace for “Code Generator” block into the subsystem which is going to be analyzed. If the code for the subsystem has not been generated already, generate the code first. Then double click on the PolySpace for “Code Generator” block.



Simulink® model with PolySpace™ launching block

4 Double-click on it

The PolySpace Analyzer Panel will then be displayed. Click on the “Start” button to start the analysis. Note the subsystem field is automatically populated with the name of the current subsystem and the results directory is automatically set to results ”subsystem_name”. If more than one subsystem is present in the model a subsystem selection dialog will be presented.



Analysis Parameters dialog

A few messages will be displayed in the main MATLAB Command window:

```
### PolySpace Technologies RTW Embedded Coder integration
### Version 1.4
### Preparing analysis
### Locating generated source files:
    ert_main.c ok (c:\MatLAB704\toolbox\rtw\rtwdemos
\rtwdemo_exemplemain_ert_rtw)
    rtwdemo_exemplemain.c ok (c:\MatLAB704\toolbox\rtw\rtwdemos
\rtwdemo_exemplemain_ert_rtw)
### Generating DRS table
### Get Parameters
### Get Signals
### Starting analysis
```

The exact messages that appear depend on the code generator being used. However all the integrations follow same format:

- First, the name of code generator is displayed and then, the version of the plug-in.
- Following this, is a list of source files, and finally the DRS (Data Range Specification) information.

- 5 Click on Execute to proceed. The progress of the analysis can be followed in the MATLAB Command window and later using the PolySpace Spooler if remote launching has been enabled.

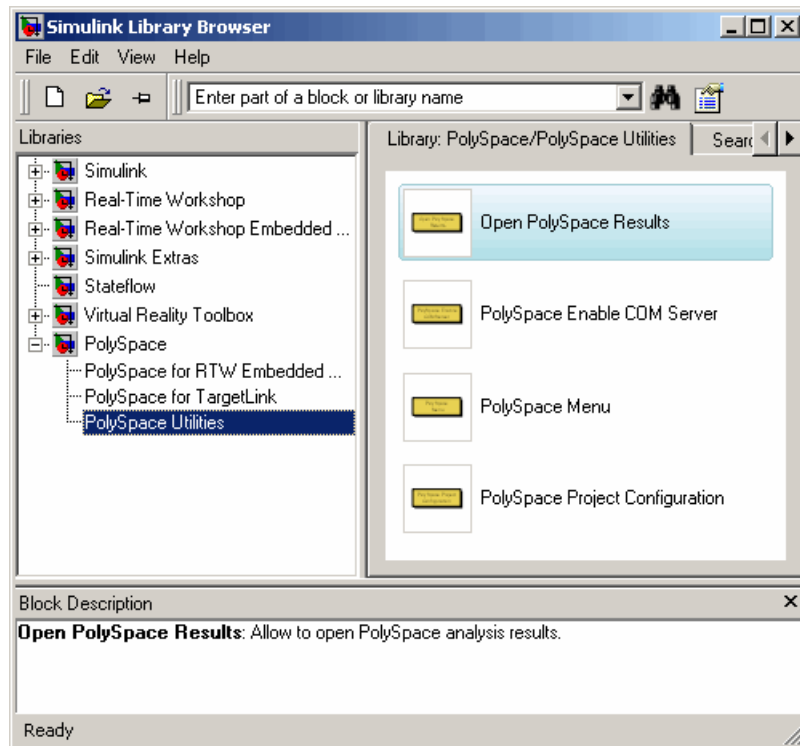
Note You can expect around 7 minutes for this model, i.e. for 4 lines of generated C code. You can also count one hour for a 3000 block model, and 15 minutes for 2000 lines of generated code. Its not proportional, and these seven minutes is the entry ticket to almost any analysis.

Fixing an Error in the Design and the Simulink® Model

After approximately 10 minutes, we have now some results to look at. We will browse the results thanks to the PolySpace Viewer.

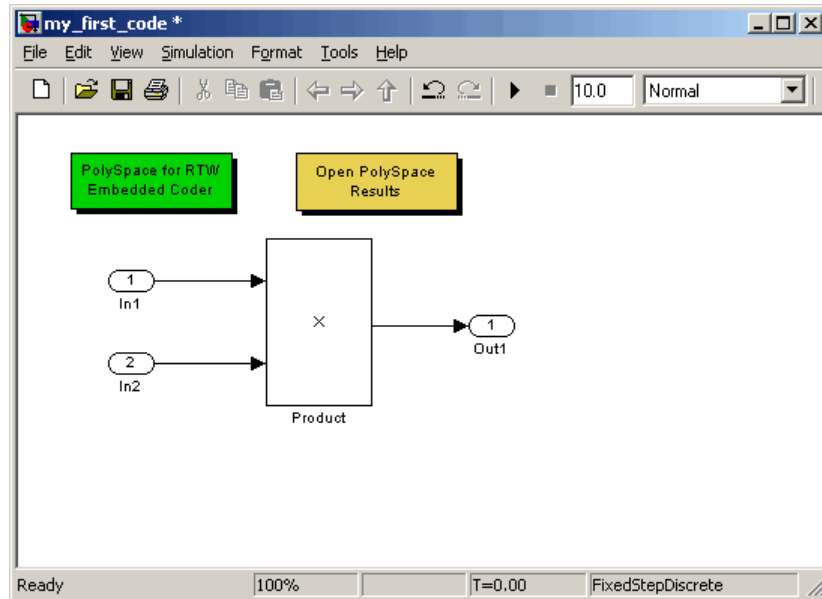
The PolySpace Viewer allows easy navigation with a right click to block in the Simulink model. To browse your results:

- 1 Drag the “Open PolySpace Results” block from the Simulink PolySpace Utilities library into the model.



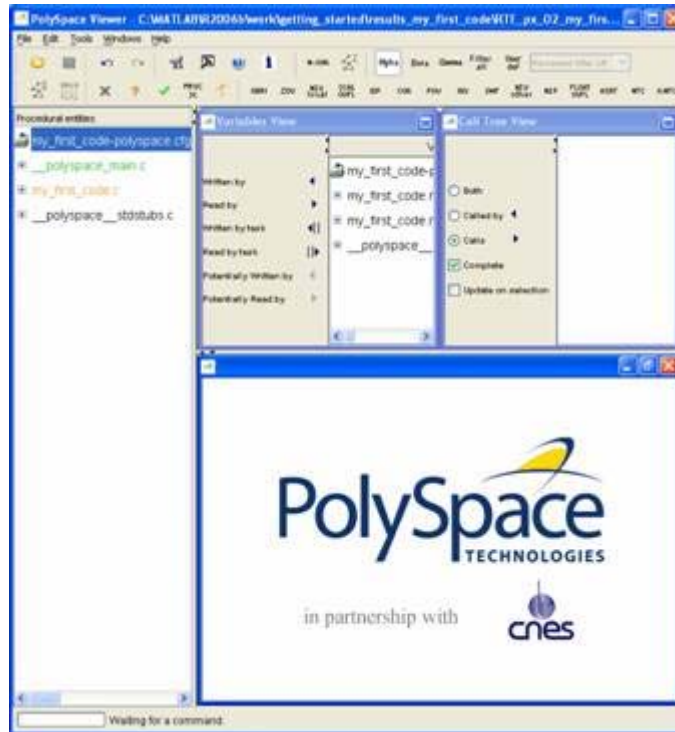
Simulink® Library Browser, section PolySpace utilities

- 2 Double-click the block. This will start the PolySpace viewer with the appropriate results.



Details of the PolySpace™ Viewer icon in Simulink® Model

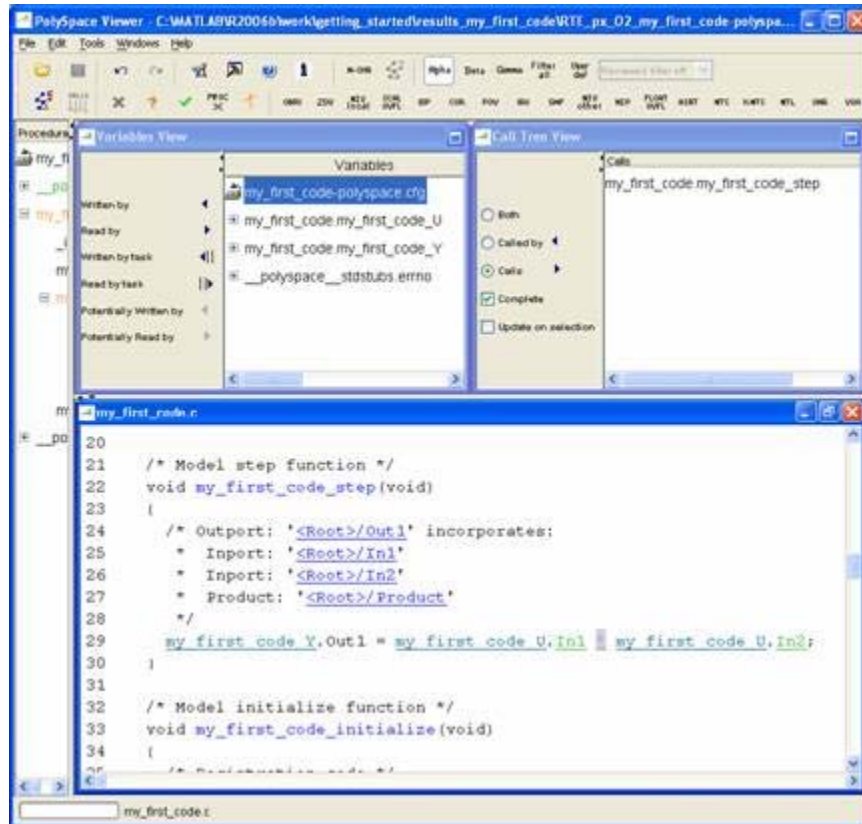
After 10-30 seconds, depending on the PC, the PolySpace Viewer will open, as shown below.



PolySpace™ Viewer

Note The mode of operation chosen in the case of this example is not important. Indeed there are few orange that we will all review.

- 3 If you have enabled Remote Launching of the analysis you will need to download your results from the server first using the PolySpace Spooler. The tool will prompt you to do this if this has not already been done. Click on CTRL-N to go to the next error.



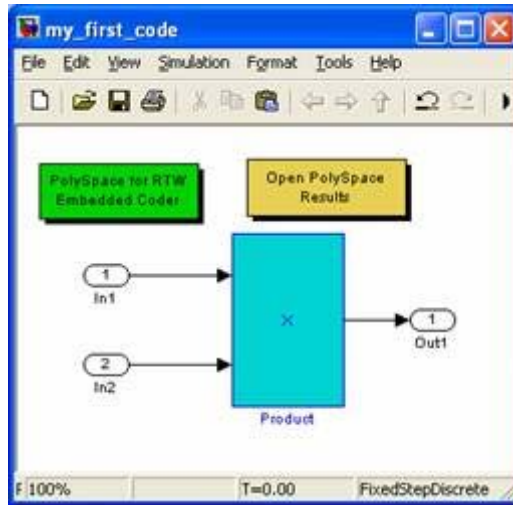
Detail of an orange check in PolySpace™ Viewer

- 4 Click on the orange PolySpace diagnostic: we have here an overflow of the two entries. PolySpace software assumes that the values for entries are full range, and their multiplication can overflow.



Overflow in the code

- 5 It is now time to get back to the model to understand what should be fixed. Searching and clicking on the first underlined blue HTML link near the check in the Source Code View will open the Simulink model and highlight the block with the error. It looks like something equivalent to “`/* <Root>/Product`”.



Model with a highlighted block

- 6 It is now up to the developer to fix his defect in his model. For instance, he may come to one of the following conclusions:
- It is a bug in the design. The developer should saturate the output, providing this functionally makes sense bound the entries in the model, by adding blocks which will test the input values, and bound them accordingly.
 - Its a bug in the specifications. The developer should bound the entries, by giving them a range in Simulink software that PolySpace verification can take the ranges into account and turns the code green.

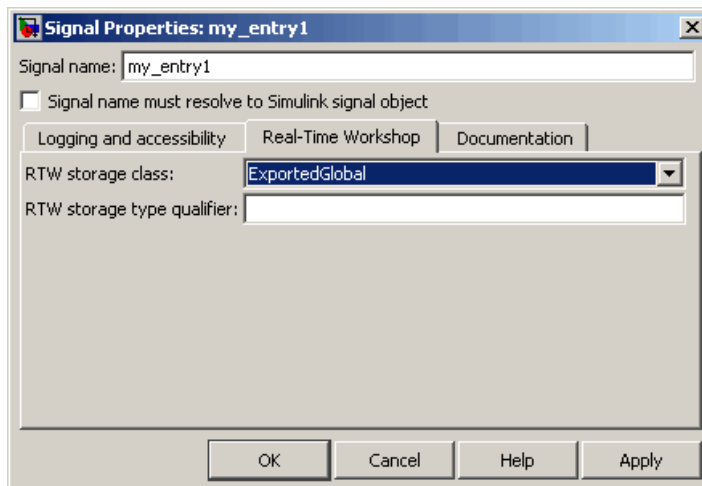
Base Workspace vs. PolySpace™ Data Ranges

After having browsed a model, the developer has identified a block whose signal ranges is not the expected one.

- If its block is supposed to be robust against this range, it is a design bug. Should the previous block be saturated? Should the signal be bounded with a “switch” block? It is up to the developer to decide the appropriate change in the model
- If the range is an input range of the model, the developer may wish to give this information to the Simulink model, so that PolySpace tools can use that range as an entry.my

Prerequisites

Have signals as ExportedGlobal.

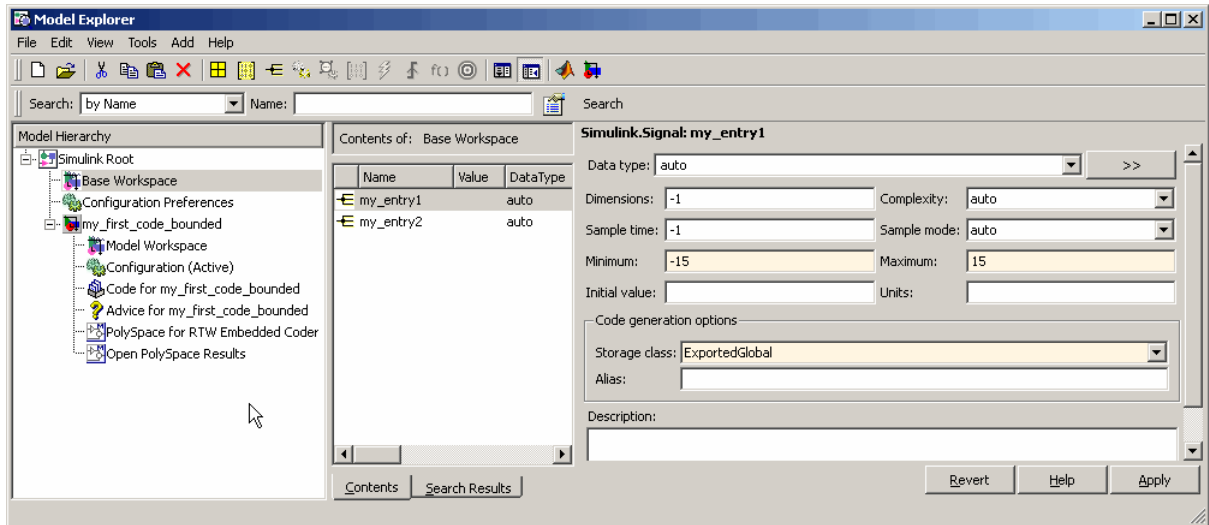


Details of a signal

Update Range of Signals

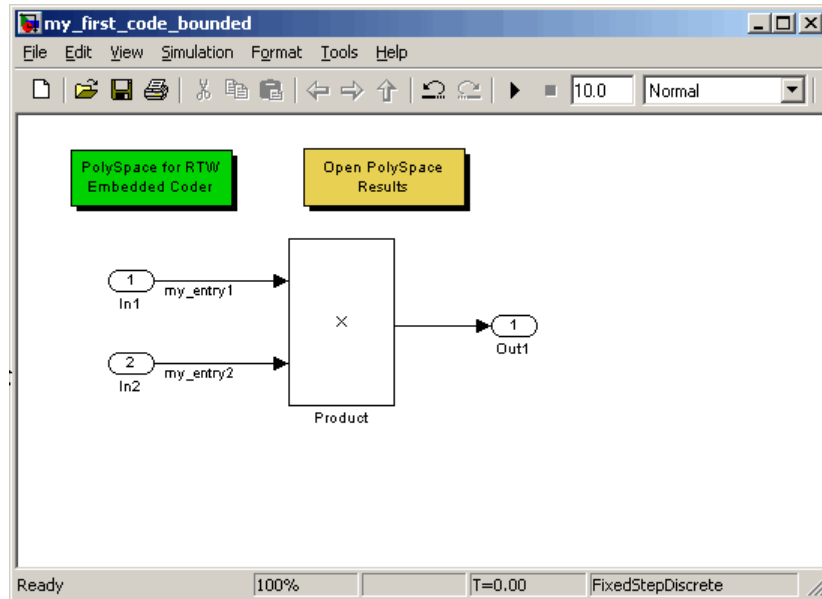
To update your signal ranges:

- 1 Open the “model explorer”, and go into the “Base Workspace” tab
- 2 Create a signal “my_entry1” & “my_entry2.”
- 3 Bound it to -15 to 15. Specify its storage class to ExportedGlobal



Signal in the “Base Workspace”

4 Model with signals on entries:



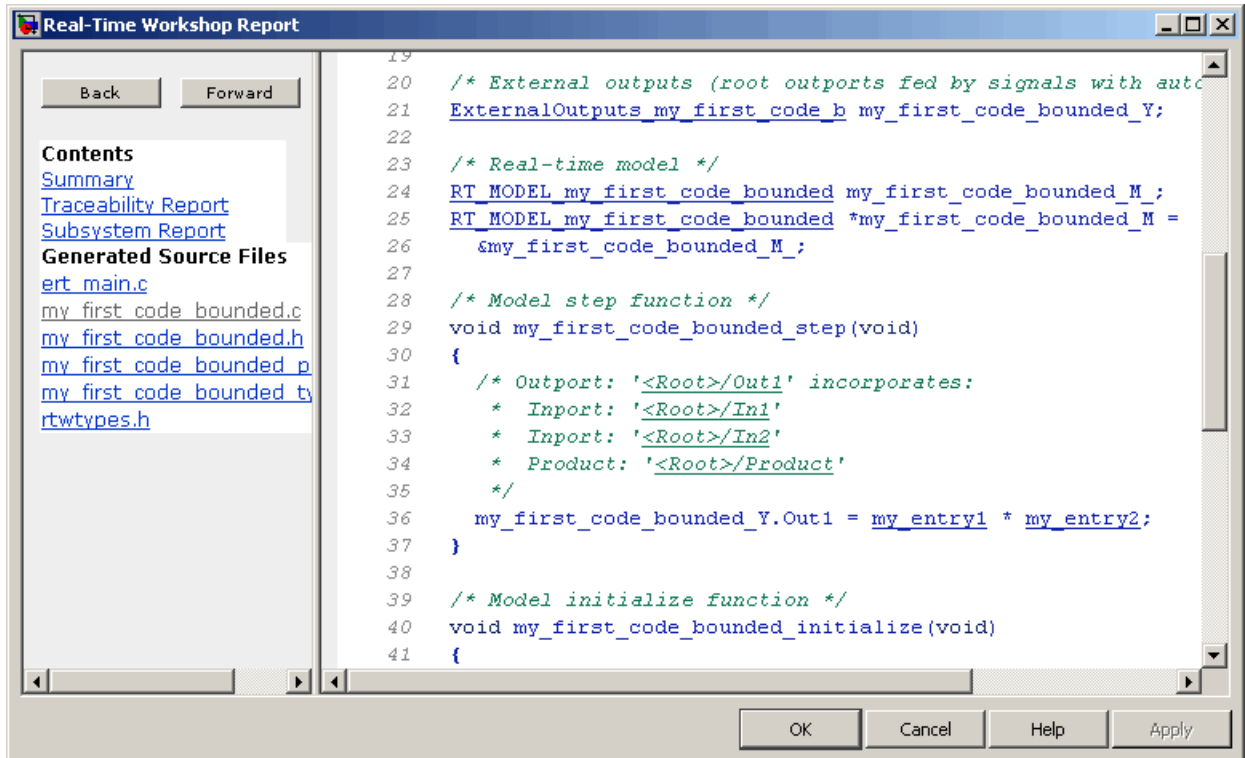
Model with signals `my_entry1` and `my_entry2` as “ExportedGlobal”.

Re-Generate Code and Launch the PolySpace™ Analysis Again

To re-generate the code and relaunch the PolySpace analysis:

- 1 Re-generate the code

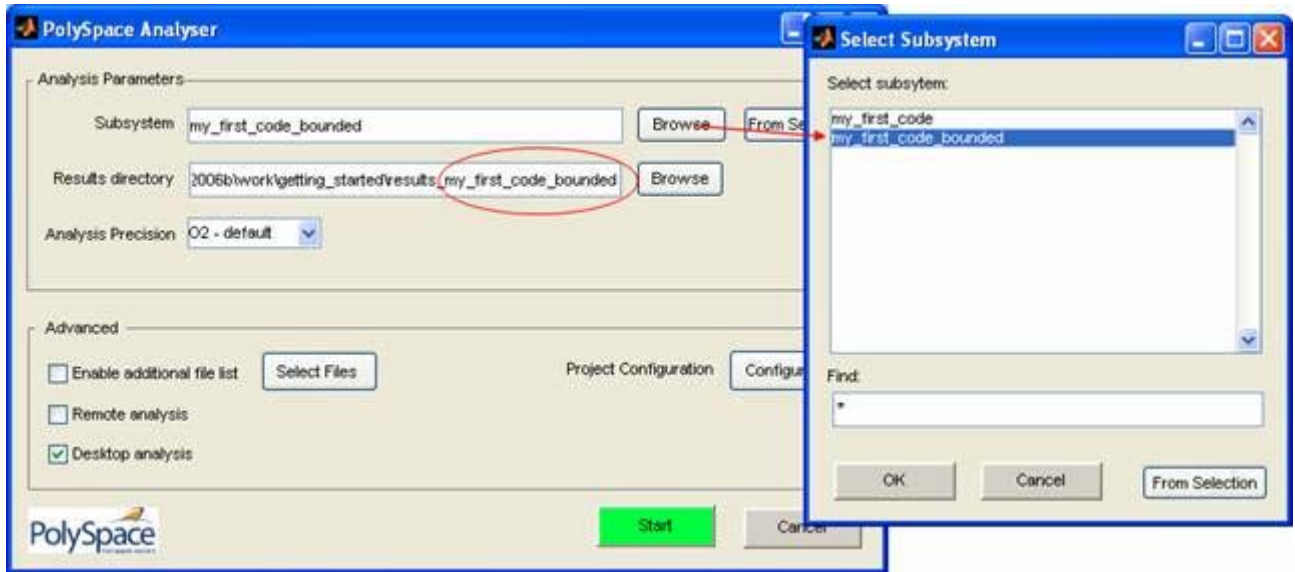
The entries are no longer part of a structure, they are separated global each.



Html report generator from Real-Time Workshop® Embedded Coder™ Software

PolySpace for RTW
Embedded Coder

- 2 Double-click on the PolySpace box:
- 3 Update the results folder name, and set it to "results_my_first_code_bounded."
- 4 Update the subsystem name, and set it to my_first_code_bounded.



Details of the results folder

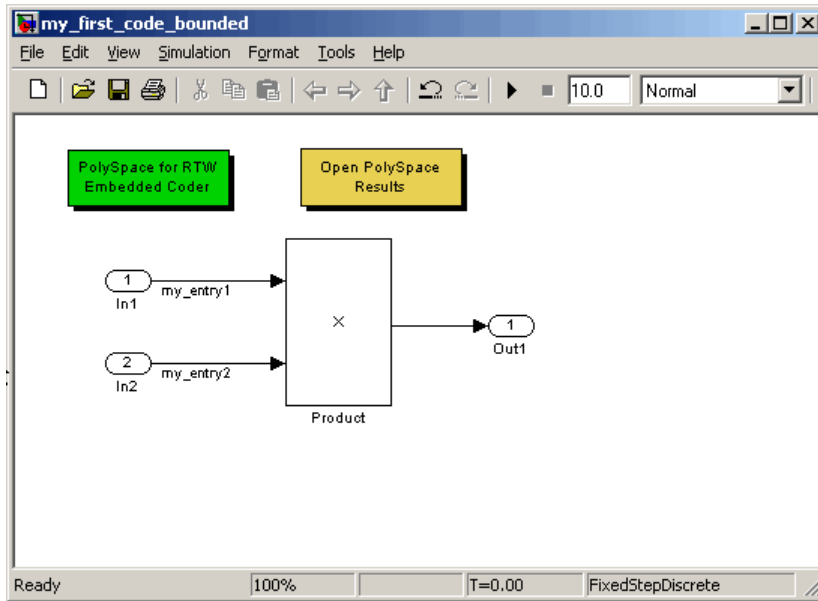
5 Start the analysis:

Start

6 Check the obtained reliability of the model using the PolySpace Viewer:

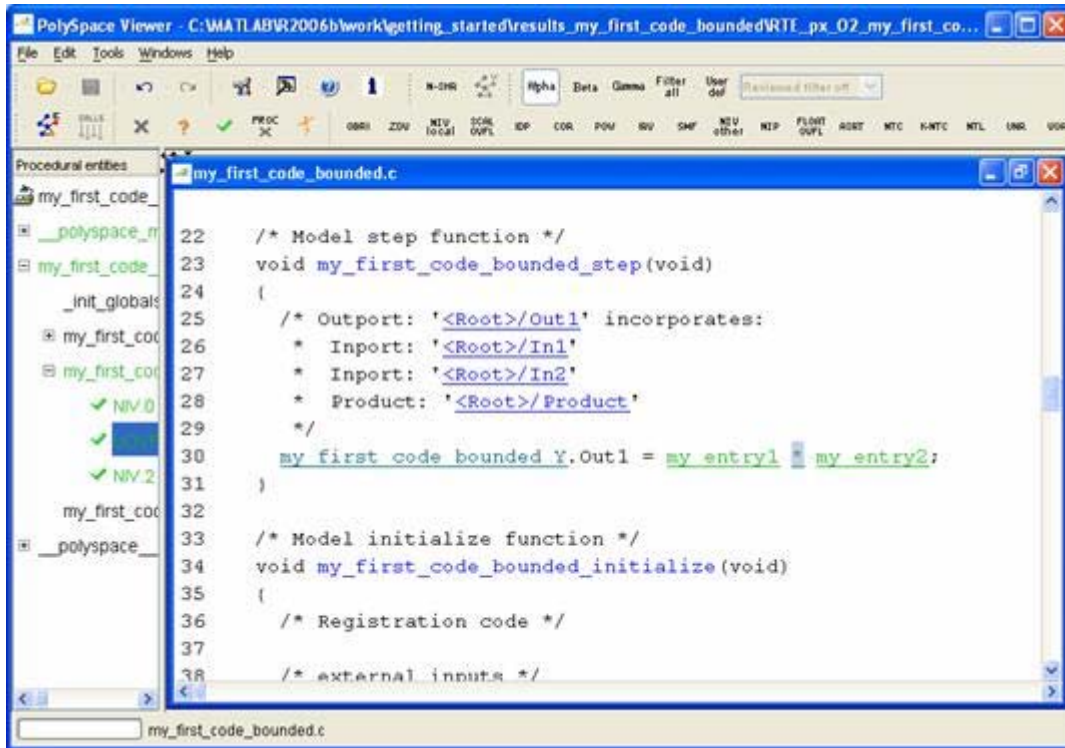
- Open the PolySpace Viewer by double-clicking on the icon

Open PolySpace
Results



Model with a PolySpace™ viewer menu

- b** Examine the generated files in the PolySpace Viewer:



Detail of generated files viewed in PolySpace™ Viewer

It is all green. The code confirms that no Run Time Error is present in the model.

Can we find more bugs in that Model?

- To answer this question, we need to now more about the tool
- Which windows of PolySpace Viewer contain which information?
- Which Colors hide which messages?
- How to find bugs thanks to PolySpace Viewer?
- Please refer to the Chapter 8, “Results Review” for more information.

Advanced Setup

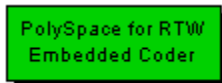
In this section...
“Hand-written Code” on page 7-26
“Target Production Environment” on page 7-28
“Creating a PolySpace™ Configuration File Template” on page 7-30
“Using the PolySpace™ Blocks Available in the Simulink® Library” on page 7-33

Hand-written Code

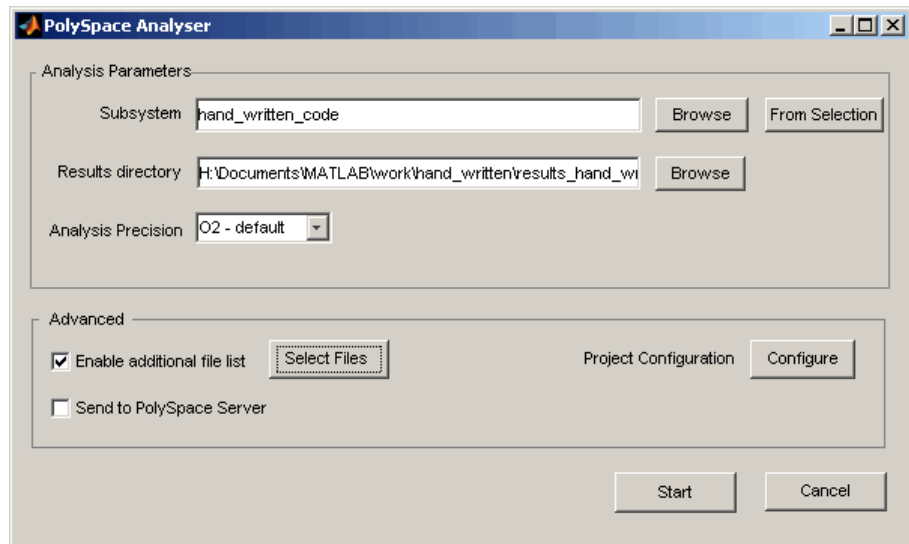
Files such as S-function wrappers are, by default, not part of the PolySpace™ analysis. They should be added manually.

To add a file manually:

- 1 When starting the PolySpace analysis

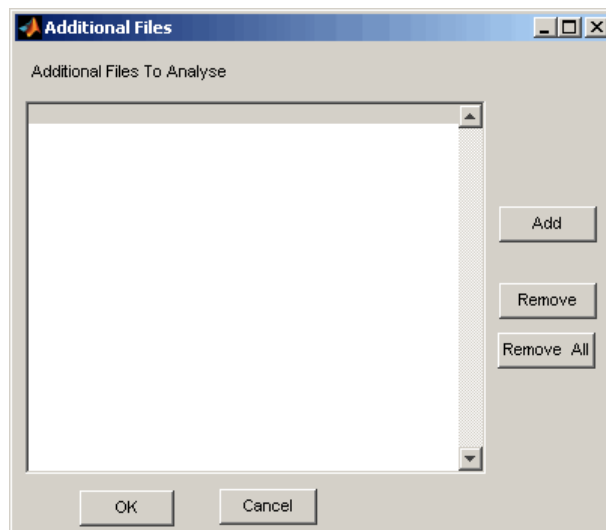


, browse and add c-files to your analysis:



- 2 Select additional files by ticking “Enable additional file list,” then click on “Select Files”.

A C File browser appears to add files to the PolySpace analysis.



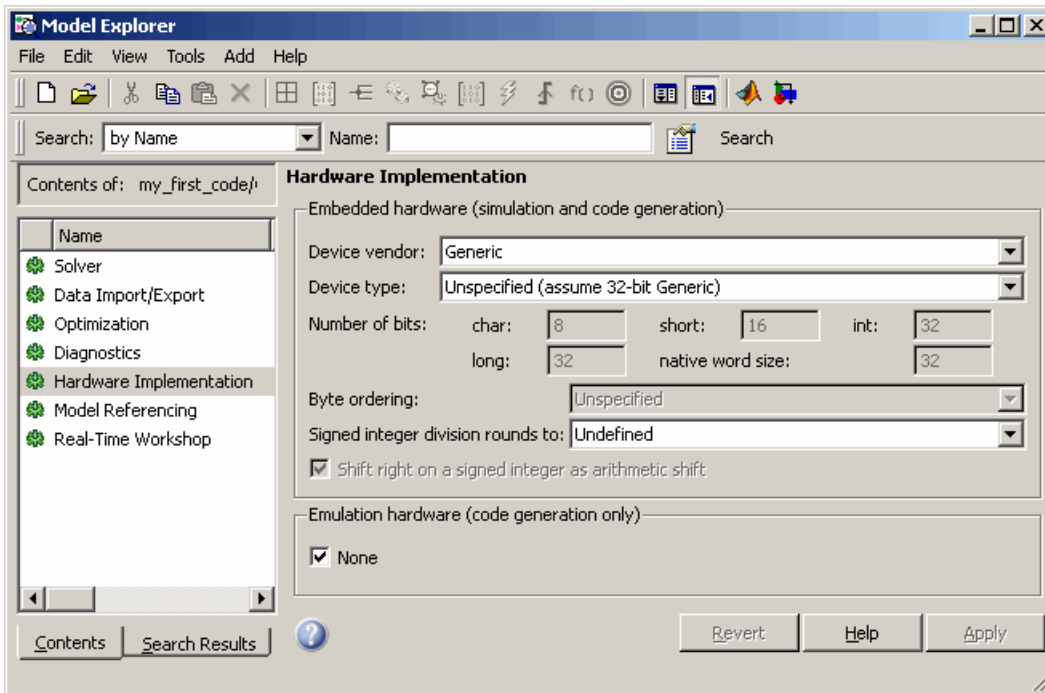
- 3 Select the appropriate c file and then start the analysis.

Target Production Environment

In Simulink® software, you need to configure the target and cross-compiler specificities.

These parameters include:

- Size of the types for char, short, int (see Hardware implementation of the model explorer)



Target selection in Simulink® Configuration Parameters

- Cross compiler flag (-D), and library include (-I), implicitly defined when - for instance - the cross compiler is setup via the “mex -setup” command.


```

Command Window
>> mex -setup
Please choose your compiler for building external interface (MEX) files:

Would you like mex to locate installed compilers [y]/n?

Select a compiler:
[1] Lcc C version 2.4.1 in C:\MATLAB\R2006b\sys\lcc
[2] Microsoft Visual C/C++ version 6.0 in C:\Program Files\Microsoft Visual S
[0] None

Compiler: 1

Please verify your choices:

Compiler: Lcc C 2.4.1
Location: C:\MATLAB\R2006b\sys\lcc

Are these correct?([y]/n):

Trying to update options file: C:\Documents and Settings\Marc Lalo\Applicatio
From template: C:\MATLAB\R2006b\bin\win32\mexopts\lccopts.bat

Done . . .

>> |

```

Cross compiler settings in MATLAB® Command Window

PolySpace settings work exactly the same way, you will need to perform the following tasks (they will be detailed step by step in the next sections).

- 1 define the same parameters for your cross compiler and target.
- 2 save this in a template PolySpace configuration file and set this template to be the default configuration file for every PolySpace analysis.

Why does this matter?

- For the PolySpace verification, an overflow on an integer type does not mean the same when the size of an integer is 16 bits or 32 bits.

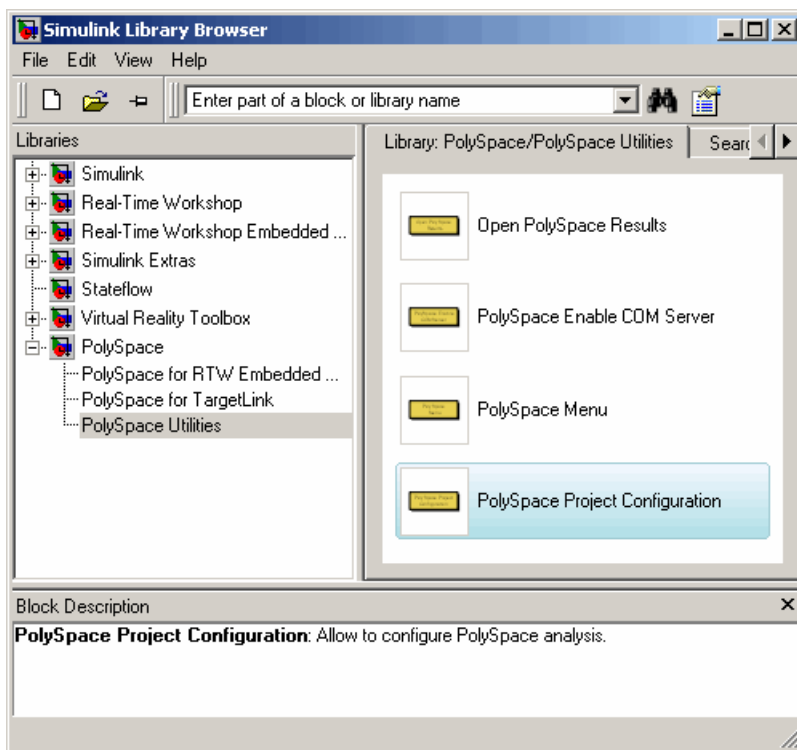
- PolySpace software needs the cross compiler header files, as they contain definitions of types, macros, used by the application, whether the application made of generated code or hand written code.

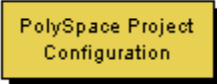
For more information, please refer to Chapter 3, “Analysis Setup”, and Chapter 10, “Options Description” in this guide.

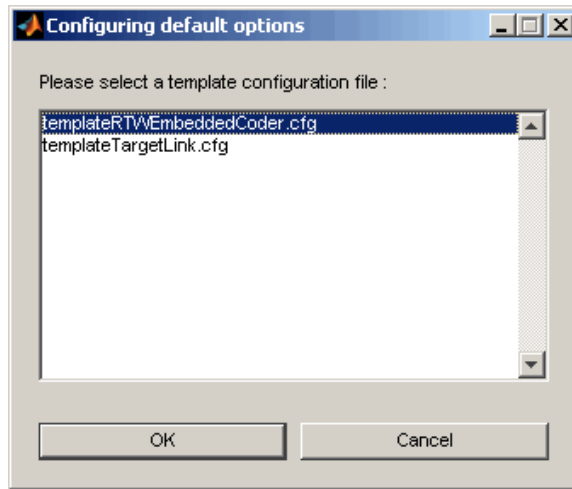
Creating a PolySpace™ Configuration File Template

To Create a configuration file template:

- 1 In the Simulink library browser, locate the PolySpace library, and expand it.



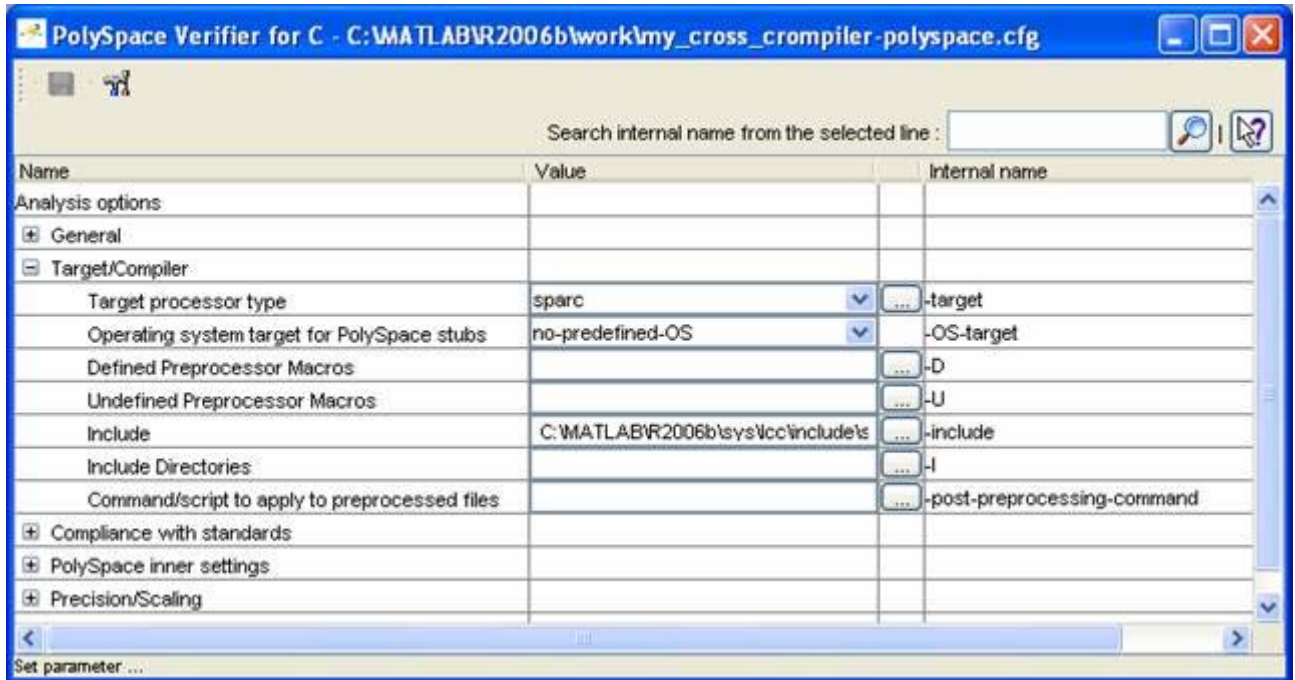
- 2 Drag the PolySpace project configuration block  to your Simulink model.
- 3 Double-click the block. This will bring a pop-up window.



Template selection

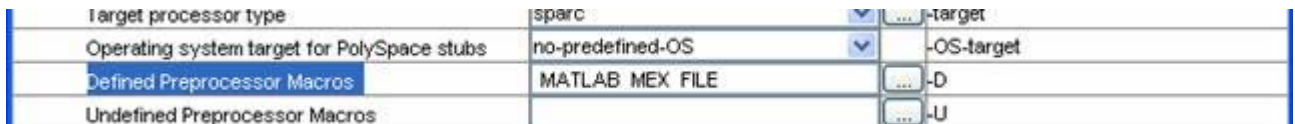
- 4 Select the first template in the list.

This will open the PolySpace interface to customize the target and cross compiler.



Target and cross compiler settings in PolySpace™ tools

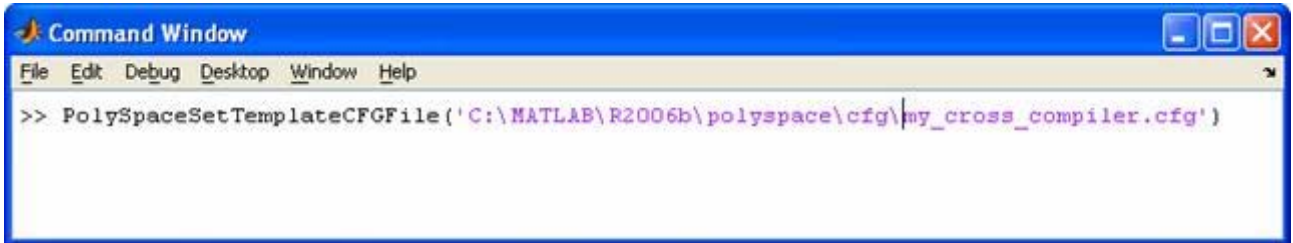
- 5 The “target” option defined the size of types. You can configure a custom target by selecting “mcpu (advanced)” at the bottom of the drop-down list
- 6 You can configure cross compiler settings by clicking on the “-D” options.



Note MATLAB_MEX_FILE is a directive option that is needed when the lcc cross-compiler is specified. Defining templates can be use in all subsequent analysis.

- 7 Save the configuration file and close the interface.

- 8 Copy the file in <<matlabroot>>/polyspace/cfg directory.
- 9 Rename it in my_cross_compiler.cfg (It could be any other name).
- 10 Type in the MATLAB® command window:
PolySpaceSetTemplateCFGFile('C:\MATLAB\R2006b\polyspace\cfg\my_cross_compiler.cfg')



Create a template configuration file

This configuration file can now be used as a template for all subsequent analysis.

Using the PolySpace™ Blocks Available in the Simulink® Library

The PolySpace Viewer allows easy navigation with a right click to block in the Simulink model.

Drag the “Open PolySpace Results” block from the Simulink PolySpace Utilities library into the model and then double click block. This will start a new MATLAB session in automation mode, open the model and start the viewer.

You can chose the Methodological assistant to review all colors in the Viewer by selecting the “Assistant” mode. The “Back to model” consists in searching the above close relative HTML link which will open the Simulink model and highlight the block with the error.

Note If you have enabled Remote Launching of the analysis you will need to download your results from the server first using the PolySpace Spooler. The tool will prompt you to do this if this has not already been done.

PolySpace™ Utilities

In this section...

“Overview of PolySpace™ Utilities” on page 7-35

“Open PolySpace™ Results” on page 7-36

“PolySpace Enable COM Server” on page 7-36

“PolySpace™ Menu” on page 7-37

“PolySpace™ Project Configuration” on page 7-38

“Archives Files Produced for the PolySpace™ Analysis” on page 7-39

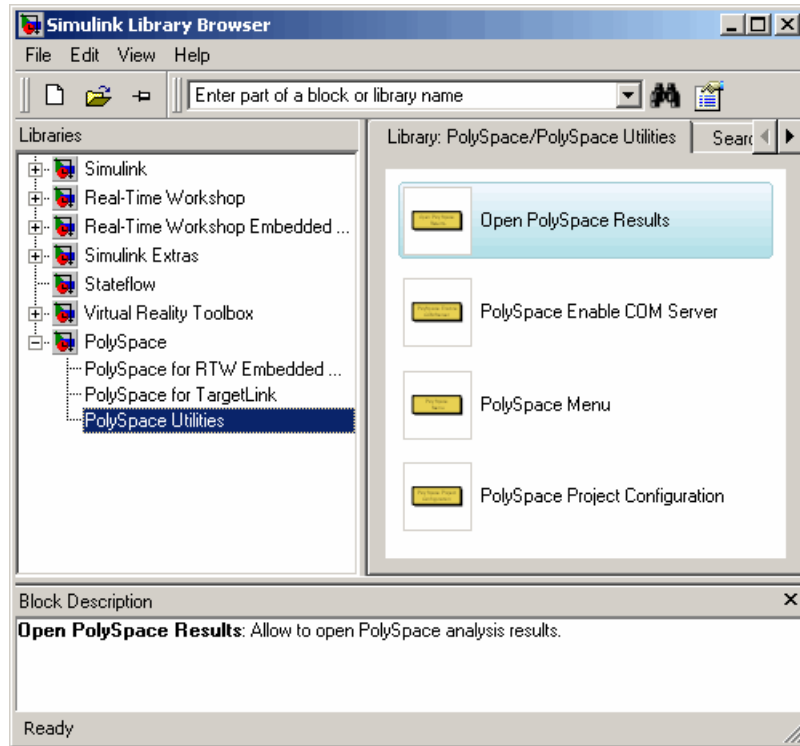
“PolySpace™ Commands Available in Batch Mode as M-Functions” on page 7-41

Overview of PolySpace™ Utilities

The PolySpace™ Utilities section consists of four blocks:

- **Open PolySpace Results**
- **PolySpace Enable COM Server**
- **PolySpace Menu**
- **PolySpace Project Configuration**

They can either be run directly from the Simulink® library browser or dragged into a Simulink model (see next figure).



PolySpace™ Utilities

Open PolySpace™ Results

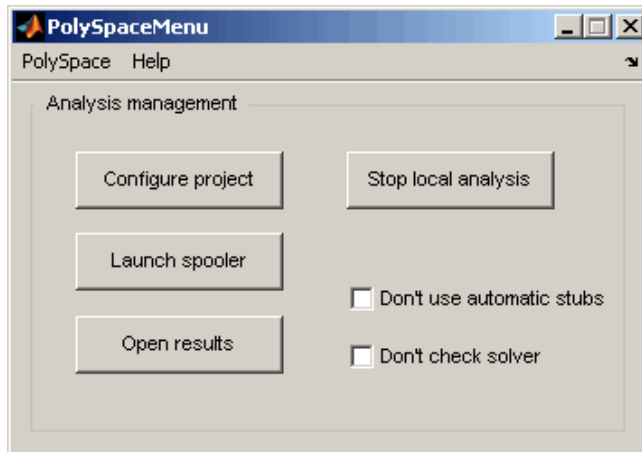
This option allows the results of the PolySpace analysis to be viewed and easy navigation with a right click from the PolySpace results to an element in the Simulink model.

PolySpace Enable COM Server

This block is called by default with the “Open PolySpace Results” block. This block is mandatory when The PolySpace Viewer has been opened outside a Simulink session to enable the feature “Back To Model” inside the Viewer.

PolySpace™ Menu

The menu consists of two sections, the first for managing the analysis and the second for configuring the tools and documentation.



PolySpace™ Menu

Analysis Management

Analysis management contains the following options:

- **Configure project** – Opens the PolySpace configuration dialog, for more information see next section.
- **Launch spooler** – Opens the PolySpace spooler. For more information, see Chapter 2, “Getting Started”.
- **Open results** – Opens the PolySpace Viewer with the last available results. If the analysis has been done on the server, downloading them first is required before clicking on this button. It is recommended to not change the proposed directory during download.
- **Stop local analysis** – Stops an analysis running on the local machine. If the analysis has been remotely spooled this option will only work during the compilation phase before the analysis is sent to the server. However, you can click the **Launch spooler** button and stop the analysis from the spooler dialog.

General Options

Tools & Documentation contains the following options:

- **Don't use automatic stubs** – Enables/disables PolySpace automatic stubbing of certain blocks behavior. This behavior depends on the code generator being used and is described in the documentation specific to your code generator below.
- **Don't check solver** – Disables the check of the solver used with Real-Time Workshop® Embedded Coder™ software.

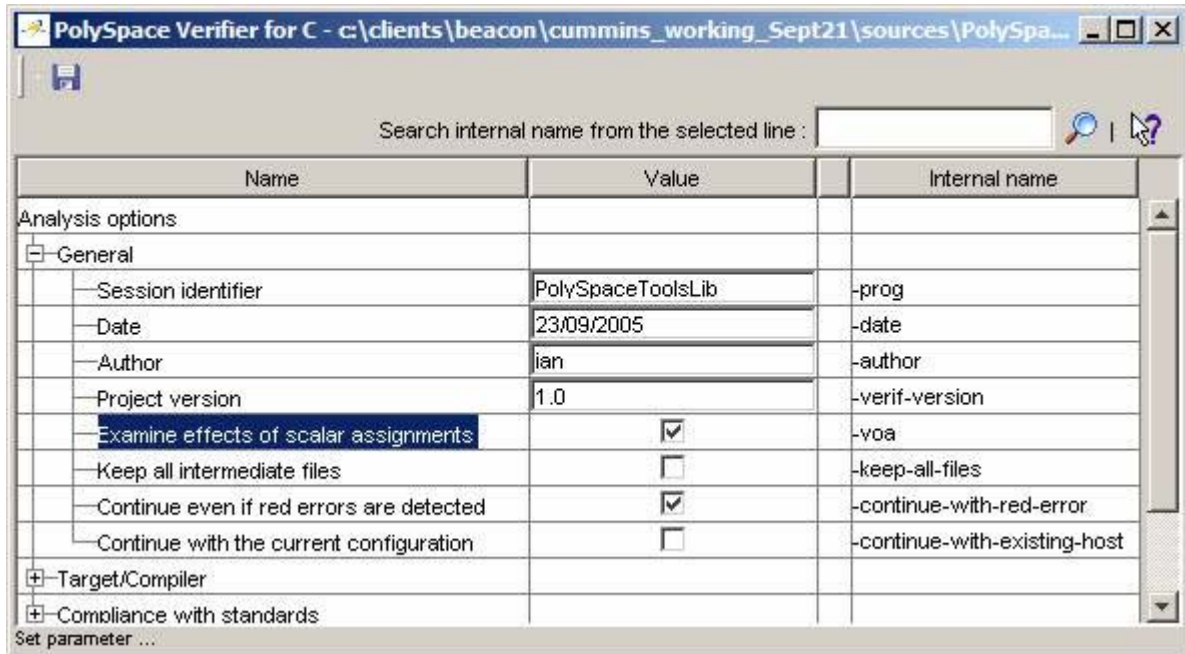
PolySpace™ Project Configuration

Clicking on “Project configuration” starts a cut-down PolySpace launcher (see next figure).

Next figure allows the configuration of the PolySpace project. For example setting items such as the processor type the code has been generated for, Compilation flags etc. The first time the tool is run a template configuration is created with the following options set:

```
-voa  
-continue-with-red-error  
-continue-with-existing-host  
-ignore-float-rounding  
-OS-target no-predefined-OS  
-allow-ptr-arith-on-struct  
-results-dir results
```

Other options are automatically set depending on the code generator being used. See the documentation for specific code generators below for more information.



Project Configuration Interface

Archives Files Produced for the PolySpace™ Analysis

For further information, here is a list of files used during a PolySpace analysis:

Template files located in MATLAB® installation directory\polyspace\

When an analysis is first performed the tool copies the following two files into the local model directory. On subsequent analyses the files are not copied again meaning it is OK to model the copies in the model directory.

- `cfg\templateEmbeddedCoder.cfg` — This file is copied to the `model_directory/model_name-polyspace.cfg` at the start of the first analysis of the model. It contains the template PolySpace configuration settings to support the TargetLink® code generator. The `templateTargetLink.cfg` file can be updated with site specific settings, to ease analysis of new models.

A MATLAB® command exists to change the name/location of the file which contains the template configuration:

```
PolySpaceSetTemplateCFGFile(config_filename)
```

This is most useful when the PolySpace analysis is started as part of an automated process. Here the process would set the template configuration file to use, erase the local copy in the model directory and then start the PolySpace analysis.

- `stub\ppcom_ec.sh` — This file is copied to the `model_directory/ppcom_ec.sh` at the start of the first analysis of a model. The file is not recopied for subsequent analyses. It is used to stubbing of lookup tables (only of interpolation, not extrapolation) types to improve the accuracy of analysis results.

Files used in the model directory

- `model-name-polyspace.cfg` — As mentioned above this file is copied from the MATLAB installation directory `\polyspace\cfg\templateEmbeddedCoder.cfg` file the first time an analysis is run on a model. It is subsequently modified by the Project Configuration block, or the Configure button in the option in the PolySpace Analyzer dialog. It contains the PolySpace Verifier settings for analyzing the current model.
- `ppcom_ec.sh` — The PolySpace Embedded Coder post pre-processing command.
- `polyspace_additional_file_list.txt` — This file is created if the Advanced option, Select Files is used in the PolySpace Analyzer dialog box. This option allows files that are not part of the model to be analyzed together with the model. For example these files could contain custom lookup table code, custom stubs, device driver code etc. The Enable additional file list option needs to be set together with configuring the list of extra files to analyze.



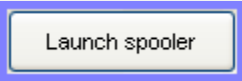
Auto-generated files in the model directory


These files are generated from the model for each analysis when it is started, and do not need archiving:

- `model_name_drs.txt` — The drs information extracted automatically from the model.
- `polyspace_include_dir_list.txt` — List of compilation include directories extracted from the mode.
- `polyspace_file_list.txt` — List of file contained in the model to analyze
- `model_name_last_parameter.txt` — The last set of parameters used in the PolySpace Analyzer dialog box.

PolySpace™ Commands Available in Batch Mode as M-Functions

You can also run the following commands from the command line.

Command	Description	Icon
<code>PolySpaceForEmbeddedCoder</code>	Launch PolySpace verification on code generated by Real-Time Workshop Embedded Coder software	
<code>PolySpaceForTargetLink</code>	Launch PolySpace on code generated by TargetLink	
<code>PolySpaceSpooler</code>	Inspect the queue of the remotely sent analysis over the server	

Command	Description	Icon
PolySpaceViewer	Launch PolySpace Viewer	
PolySpaceSetTemplateCFGFile	Select a template file in batch mode	
PolySpaceGetTemplateCFGFile	Get the currently selected template file (empty by default)	
PolySpaceReconfigure	In case of a PolySpace release update without enabling the MATLAB plug-in	

Example with EmbeddedCoder:

Suppose that you open a Simulink model with the name `example.mdl`.

Enter `PolySpaceForEmbeddedCoder('example')` in the MATLAB Command window.

The analysis starts.

Code Generator Specific Information

In this section...
“PolySpace™ Model Link™ SL Product” on page 7-43
“PolySpace™ Model Link™ TL Product” on page 7-44

PolySpace™ Model Link™ SL Product

The PolySpace™ Model Link™ SL product has been tested with Real-Time Workshop® Embedded Coder™ software — see the Installation Guide for more information.

Subsystems

A dialog will be presented after clicking on the PolySpace for Embedded Coder block if multiple subsystems are present in a diagram. Simply select the subsystem to analyze from the list. The subsystem list is generated from the directory structure from the code that has been generated.

Default Options

The following default options are set by the tool:

```
-I path to source code
-desktop
-D PST_ERRNO
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
```

Note *matlabroot* is the MATLAB® tool installation directory.

Data Range Specification

The tool automatically creates PolySpace Data RangeSpecification (DRS) information using the MATLAB workspace. This DRS information is used to initialize each global variable to the range of valid values as defined by the min-max information in the workspace.

Main sources of information are Simulink.signals and Simulink.parameters.

Code Generation Options

The Real-Time Workshop® configuration parameters settings must be configured as follows for optimum use of the tool.

Note These are the options recommended by The MathWorks for generating target code.

- **Real Time Workshop** tab:
 - a** Select “Generate HTML report” and set “Include hyperlinks to model”. **Note** that if this is not set navigation from PolySpace results to the model will not work.
 - b** Set the system target file to be an appropriate ert.tlc (use the browse button to locate). This is an indication that the code generator is Real-Time Workshop Embedded Coder software (and not just Real-Time Workshop software, used for rapid prototyping).
 - c** Set the Solver parameters “Type” to *Fixed-step*, and “Solver to *discrete* (no continuous state). It illustrates that the code has been generated for a target, and not for a simulation based on continuous timing.
- Optionally, on “*Interface panel*” tab, make sure that “Generate reusable code” is unselected. Setting this option will generate more warnings in the PolySpace results.

PolySpace™ Model Link™ TL Product

The PolySpace Model Link TL product has been tested with the some release of the dSPACE® Data Dictionary version and TargetLink® Code Generator - see the Installation Guide for more information.

As the PolySpace Model Link TL product extracts information from the dSPACE Data Dictionary remember to regenerate the code before performing a PolySpace analysis. This ensures that the Data Dictionary has been correctly updated.

Subsystems

A dialog will be presented after clicking on the PolySpace for TargetLink block if multiple subsystems are present in a diagram. Simply select the subsystem to analyze from the list.

Data Range Specification

The tool automatically creates PolySpace Data RangeSpecification (DRS) information using the dSPACE Data Dictionary for each global variable. This DRS information is used to initialize each global variable to the range of valid values as defined by the min-max information in the data dictionary. This allows PolySpace software to model every value that is legal for the system during its analysis. Further the Boolean types are modeled having a minimum value of 0 and a maximum of 1. Defining the min-max information carefully in the model can help PolySpace verification to be more precise significantly because only range of reels values are analyzed.

DRS cannot be applied to static variables. Therefore, the compilation flags `-Dstatic=` is set automatically. It has the effect of removing the static keyword from the code. If you have a problem with name clashes in the global name space you may need to either rename one of or variables or disable this option in PolySpace configuration.

Lookup Tables

The tool by default provides stubs for the lookup table functions. This behavior can be disabled from the PolySpace menu — see “PolySpace™ Menu” on page 7-37 for more information. The dSPACE data dictionary is used to define the range of their return values. Note that a lookup table that uses extrapolation will return full range for the type of variable that it returns.

Default Options. The following default options are set by the tool:

- I path to source code
- desktop

```
-D PST_ERRNO
-I dspaceroot\matlab\TL\SimFiles\Generic
-I dspaceroot\matlab\TL\srcfiles\Generic
-I dspaceroot\matlab\TL\srcfiles\i86\LCC
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
```

Note *dspaceroot* and *matlabroot* are the dSPACE and MATLAB tool installation directories respectively.

Code Generation Options

From the TargetLink Main Dialog, it is recommended to set the option “Clean code” and deselect the option “Enable sections/pragmas/inline/ISR/user attributes”.

When installing the PolySpace Model Link TL product, the `tlcgOptions` variable has been updated with 'PolyspaceSupport', 'on' (see variable in 'C:\dSPACE\Matlab\TL\config\codegen\tl_pre_codegen_hook.m' file).

Results Review

Basics: Prerequisites to Reviewing
PolySpace™ Results (p. 8-2)

Provides an overview of PolySpace™
results

Colored Source Code for C (p. 8-14)

Provides specific examples of C
checks

Basics: Prerequisites to Reviewing PolySpace™ Results

In this section...
“Overview” on page 8-2
“Grey Follows Red” 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
“Specific Check Analysis” on page 8-7

Overview

Once PolySpace™ software has completed an analysis and there are graphical results available, there will be colored entries shown in the source code. This section explains how to understand the implications of the four colors:

- Red shows run-time errors which will occur every time that piece of code is executed;
- Grey shows code which is unreachable (dead code);
- Orange is a warning;
- Green shows safe instructions: these are code sections which can never lead to a run time error.

This section explains the steps necessary to analyze a result of any color. There are four core rules to bear in mind throughout this section, viz.

- The next instruction is reached providing no Run Time Error was met at the previous one.
- Each Run Time Error implies a “core dump” for PolySpace. The corresponding execution is considered to have stopped, even if the run time execution of the code might not. SO - red checks will be followed by grey checks, and orange checks only propagate the green parts through to subsequent checks.

- You should focus on the message given by PolySpace, and try not to jump to false conclusions. You must explain the color of a check step by step, until you find the root cause.
- You should focus on an explanation by examining the code, and try not to be influenced by knowledge of what the code actually does.

Grey Follows Red

This section explains grey checks follow **red** ones, and hence how **green** checks are propagated out of **orange** ones. In the example below, consider the explanation of

- the grey checks after the **red** in the red function;
- and the **green** checks relating to the array.

```

void red(void)                extern int Read_An_Input(void);
{                               void propagate(void)
int x;                         {
x = 1 / x ;                    int X;
x = x + 1;                    int y[100];
}                               X = Read_An_Input();
                               y[X] = 0; // [array index within bounds]
                               y[X] = 0;
                               }

```

Consider each line of code for:

The red function:

- When PolySpace divides by X, it has not been initialized. Therefore the corresponding check (Non Initialized Variable) on X is red;
- As a result all possible execution paths are stopped, because they all produce an RTE.

The propagate function:

- X is assigned the return value of Read_An_Input. After this assignment, $X \sim [-2^{31}, 2^{31}-1]$.

- At the first array access, an “out of bounds” error is possible since X can be equal to (say) -3 as well as 3;
- Subsequently, all conditions leading to an RTE are assumed to have been truncated - they are no longer considered in the analysis. So on the following line, the executions for which $X \sim [-2^{31}, -1]$ and $[100, 2^{31}-1]$ are stopped;
- Consequently, at the next instruction, $X \sim [0, 99]$;
- Hence at the second array access, the check is green because $X \sim [0, 99]$.

Summary

Green checks are propagated out of orange ones.

Note When writing manual stubs, you can use this property of PolySpace software to restrict data input values: See how to assign ranges of variables in “Reduce the cloud of points” on page 9-62.

What is the Message and What does it Mean?

PolySpace numbers the checks it makes using the same sequence as that followed during the execution of the code.

Consider the instruction `x++;`

PolySpace first checks for a potential NIV (Non Initialized Variable) for x, then checks the potential OVFL (overflow) - which mimics the execution sequence. An awareness of such sequences will help you to understand the message presented by PolySpace, and will help you to work out what that message implies.

Consider the orange NIV on x in the test:

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

You might conclude that the PolySpace analysis has not kept track of the value of x. However, studying the context in which the check is made will help you to understand it better.

```
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++; } // grey code
}
```

Explanation

Using the viewer, you can see the category of each check by clicking on it. When an orange check is considered, any value of a variable which would result in an RTE is not considered further. However, as the example NIV (Non Initialized Variable) shows, any value which does not cause the RTE is considered for analysis on subsequent lines.

The correct interpretation of these analysis results might be that if `x` has been initialized, the only possible value for it is 100. Thus `x` can never be initialized and greater than 101, so the rest of the code is grey. Such a conclusion might be different from that reached in haste!

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 first rules of reviewing results are:

- Focus on the message given by PolySpace Verifier,
- and try not to jump to conclusions.

What is the C Explanation

Results can only be explained based on the code analyzed, so be wary of considering:

- a physical action from the environment in which the code is intended to operate;
- a particular configuration which is not part of the analysis;
- or any reason other than the code itself.

Remember, all the tool deals with is the C code submitted to it!

Consider the example below, paying particular attention to the dead (grey) 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; // grey code on this line
}
```

You can see that

- the line containing the access to the y array is unreachable;
- so the test to assess whether x is equal to 0 is always false;
- **the initial conclusion is that "the test is always false"**. Now, it would be easy to jump to the conclusion that this results from input data which is always different from 0. However, Read_An_Input can be any value in the full integer range, so this is not the right explanation.

So consider the execution path leading to the grey code

- The orange check on the array access (y[x]) will truncate any execution path leading to a run time error, meaning that subsequent lines will be dealing with $x \sim [0, 99]$

- The orange check on the division will also truncate all executions paths that lead to a run time error so that in our example, all instances where x is equal to 0 are stopped. For the code execution path after the orange division sign, $x \sim [1; 99]$;
- Thus x is never equal to 0 **at this line** - and hence, the array access is green ($y(x-1)$).

Summary

In this example, all results are located in the same procedure. But by using the call tree, the same process can easily be followed 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!**

Specific Check Analysis

- “PolySpace Memorizes the Relationships Between Variables” on page 8-7
- “The Purpose of the -continue-with-red-error Option.” on page 8-9
- “Default Settings, -continue-with-red-error and Side Effects” on page 8-11
- “Why There Might be 2 Distinct Colors in a while/for Statement. ” on page 8-12

PolySpace Memorizes 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 run time 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 Verifier exits immediately when encountering a run time 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.

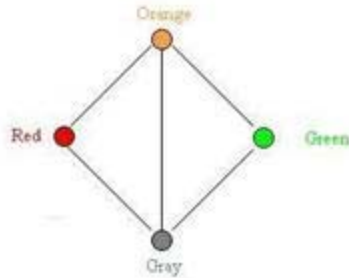
The Purpose of the -continue-with-red-error Option.

This option is used to deal with two primary circumstances.

- A red error appears in code which was expected to be dead code.
- A red error appears which was expected, but the analysis is required to continue.

PolySpace performs an upper approximation of variables. Consequently, it may be true that PolySpace analyses a particular branch of code as though it was accessible, despite the fact that it could never be reached during "real life" execution. In the example below, there is an attempt to compare elements in an array, and PolySpace is not able to conclude that the branch was unreachable. PolySpace may conclude that an error is present in a line of code, even when that code cannot be reached.

Consider the following figure. As a result of imprecision, each color shown can be approximated by a color immediately above it in the grid. It is clear that green or red checks can be approximated by orange ones, but the approximation of grey checks is less obvious.



During PolySpace analysis, data values possible at execution time are represented by supersets including those values - and possibly more besides.

Grey code represents a situation where no valid data values exist. Imprecision means that such situation can be approximated

- by an empty super set;
- by a non-empty super set, members of which may generate checks of any color.

And hence PolySpace cannot be guaranteed to find all dead code in an analysis.

However, there is no problem in having grey checks approximated by red ones. Where any red error is encountered, all instructions which follow it in the relevant branch of execution are aborted as usual. At execution time, it is also true that those instructions would not be executed.

Consider the following example.

```
if
  (condition)
then
  action_producing_a_red;
```

After the "if" statement, the only way execution can continue is if the condition is false; otherwise a **red check** would be produced. Therefore, after this branch the condition is always false. For that reason, the

-continue-with-red-error option is provided to allow code analysis to continue, even with a specific error.

Remember that this propagates values throughout your application. None of the execution paths leading to the RTE will continue after the error and if the **red check** is a real problem rather than an approximation of a grey check, then the analysis will not be representative of how the code will behave when the red error has been addressed.

The -continue-with-red-error option is applicable in this example case.

```
1 int a[] = { 1,2,3,4,5,7,8,9,10 };
2
3
4 void main(void)
5 {
6   int x=0;
7   int tmp;
8
9
10  if (a[5] > a[6])
11    tmp = 1 /x;
12 }
```

Default Settings, -continue-with-red-error and Side Effects

This section explains why when a red error has been found the analysis 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 analysis, 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 grey 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 grey. This red NTC is the result of an imprecision, and should be grey.

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 analysis
- 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 analysis

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--; }
   // the colour of "array index within bounds" is
   // first green
   // than red
6 }
```

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.

Colored Source Code for C

In this section...

“Illegal Pointer Access to Variable or Structure Field: IDP” on page 8-15

“Array Conversion Must Not Extend Range: COR” on page 8-16

“Array Index Within Bounds: OBAI” on page 8-17

“Initialized Return Value: IRV” on page 8-18

“Non-Initialized Variables: NIV/NIVL” on page 8-19

“Non-Initialized Pointer: NIP” on page 8-20

“Power Arithmetic: POW” on page 8-20

“User Assertion: ASRT” on page 8-21

“Scalar and Float Underflows: UNFL” on page 8-23

“Scalar and Float Overflows: OVFL” on page 8-23

“Float Underflows and Overflows: UOVFL” on page 8-24

“Scalar or Float Division by Zero: ZDV” on page 8-28

“Shift Amount in 0..31 (0..63):SHF” on page 8-28

“Left Operand or Left Shift is Negative: SHF” on page 8-29

“Function Pointer Must Point to a Valid Function: COR” on page 8-30

“Wrong Type for Argument: COR” on page 8-31

“Wrong Number of Arguments: COR” on page 8-32

“Wrong Return Type of a Function Pointer: COR” on page 8-33

“Wrong Return Type for Arithmetic Functions: COR” on page 8-33

“Pointer Within Bounds: IDP” on page 8-34

“Non Termination of Call or Loop” on page 8-49

“Unreachable Code: UNR” on page 8-58

“Value on Assignment: VOA” on page 8-60

“Inspection Points: IPT” on page 8-62

Illegal Pointer Access to Variable or Structure Field: IDP

This is a check to establish whether in the dereferencing of an expression of the form $ptr+i$, the variable/structure field initially pointed to by ptr is still the one accessed. See ANSI C standard ISO/IEC 9899 section 6.3.6.

Consider the following example.

```
1 int a;
2
3 struct {
4   int f1;
5   int f2;
6   int f3;
7 } S;
8
9 void main(void)
10 {
11   volatile int x;
12
13   if (x)
14     *(&a+1) = 2; // IDP ERROR: &a +1 doesn't point to a any longer
15   if (x)
16     *(&S.f1 +1) = 2; // IDP ERROR: you are not allowed to access f2
like this
17 }
```

According to the ANSI C standard, it is not permissible to access a variable (or a structure field) from a pointer to another variable. That is, $ptr+i$ may only be dereferenced if $ptr+i$ is the address of a subpart of the object pointed to by ptr (such as an element of the array pointed to by ptr , or a field of the structure pointed to by ptr).

For instance, the following code is correct because the length of the entity pointed to by ptr_s reflects the full structure length of My_struct (at line 11):

```
1 typedef struct {
2   int f1;
3   int f2;
4   int f3;
```

```
5 } My_Struct;
6
7 My_Struct s = {1,2,3};
8
9 int main(void)
10 {
11 My_Struct *ptr_s = &s;
12
13 // change to f2 field
14 *((int *)&s +1) = 2; // Correct evaluation
15
16 return 0;
17 }
```

Array Conversion Must Not Extend Range: COR

This is a check to establish whether a small array is mapped onto a bigger one through a pointer cast. Consider the following example.

```
1
2 typedef int Big[100];
3 typedef int Small[10];
4 typedef short EquivBig[200];
5
6 Small smalltab;
7 Big bigtab;
8
9 void main(void)
10 {
11 volatile int random;
12
13 Big * ptr_big = &bigtab;
14 Small * ptr_small = &smalltab;
15
16 if (random) {
17     Big *new_ptr_big = (Big*)ptr_small; // COR ERROR: array
conversion must not extend range
18 }
19 if (random) {
20     EquivBig *ptr_equivbig = (EquivBig*)ptr_big;
21     Small *ptr_new_small = (Small*)ptr_big; // Conversion
```

```

verified
22 }
23 }

```

In the example above, a pointer is initialized to the *Big* array with the address of the *Small* array. This is not legal since it would be possible to dereference this pointer outside the *Small* array. Line 20 shows that the mapping of arrays with same size and different prototypes is acceptable.

Array Index Within Bounds: OBAI

This is a check to establish whether an index accessing an array is compatible with the length of that array. Consider the following example.

```

1
2 #define TAILLE_TAB 1024
3 int tab[TAILLE_TAB];
4
5 void main(void)
6 {
7     int index;
8
9     for (index = 0; index < TAILLE_TAB ; index++)
10    {
11        tab[index] = 0;
12    }
13    tab[index] = 1; // OBAI ERROR: Array index out of bounds [0..1023]
14 }

```

Just after the loop, *index* equals *SIZE_TAB*. Thus *tab[index] = 1* overwrites the memory cell just after the last array element.

An OBAI check can also be localized on a + operator, as another example illustrates.

```

1 int tab[10];
2
3 void main(void)
4 {
5     int index;
6     for (index = 0; index < 10 ; index++)

```

```
7 *(tab + index) = 0;
8
9 *(tab + index) = 1; // OBAI ERROR: Array index out of bounds
10 }
```

Note that the message associated with the check OBAI gives always the range of the array: Array index out of bounds [0..1023]

Initialized Return Value: IRV

This is a check to establish whether a function returns an initialized value. Consider the following example.

```
1
2 extern int random_int(void);
3
4 int reply(int msg)
5 {
6     int rep = 0;
7     if (msg > 0) return rep;
8 }
9
10 void main(void)
11 {
12     int ans;
13
14     if (random_int())
15         ans = reply(1); // IRV verified: function returns an
initialised value
16     else if (random_int())
17         ans = reply(0); // IRV ERROR: function does not return an
initialised value
18     else
19         reply(0); // No IRV checks because the return value
is not used
20
21 }
22
23
```

Variables are often initialized using the return value of functions. However, in the above example the return value is not initialized for all input parameter values. In this case, the target variable will not be always be properly initialized with a valid return value.

Non-Initialized Variables: NIV/NIVL

This is a check to establish whether a variable is initialized before being read. Consider the following example.

```
1
2 extern int random_int(void);
3
4 void main(void)
5 {
6   int x,i;
7   double twentyFloat[20];
8   int y = 0;
9
10  if (random_int()) {
11    y += x; // NIV ERROR: Non
Initialized Variable (type: int 32)
12  }
13  if (random_int()) {
14    for (i = 1; i < 20; i++) {
15      if (i % 2) twentyFloat[i] = 0.0;
16    }
17    twentyFloat[2] = twentyFloat[4] + 5.0; // NIV Warning. only odd
indexes are initialized.
18  }
19 }
```

The result of the addition is unknown at line 11 because x is not initialized (UNR unreachable code on "+" operator).

In addition, line 17 shows how PolySpace™ software prompts the user to investigate further (by means of an orange check) when all cells have not been initialized.

Note Associated to each message which concerns a NIV check, PolySpace software gives the type of the variable like the following examples: (type: volatile int32), (type: int 16), (type: unsigned int 8), etc.

Non-Initialized Pointer: NIP

This is a check to establish whether a pointer is initialized before being dereferenced. Consider the following example.

```
2
3 void main(void)
4 {
5   int* p;
6   *p = 0; // NIP ERROR: pointer not initialized
7 }
```

As p is not initialized, an undefined memory cell would be overwritten at line 6 ($*p = 0$) (also leading to the unreachable grey check on " $*$ ").

Power Arithmetic: POW

Check to establish whether the standard **pow** function from *math.h* library is used with an acceptable (positive) argument.

```
1 #include <math.h>
2
3 extern double pst_randd(void);
4 extern int pst_intr(void);
5 int main(void)
6 {
7   int IRes, ILeft, IRight;
8   double Res;
9
10  if (pst_intr()) {
11    ILeft = 0;
12    IRight = -1;
13    Res = pow(ILeft,IRight); // POW ERROR: Power must be positive
14  }
15 }
```

```
16 ILeft = 2e8;
17 IRight = 2;
18 Res = pow(ILeft, IRight); // OVFL Warning
19 Res = pow (pst_randd(), pst_randd()); // POW Warning :
Power may be not positive
20
21 return(0);
22 }
```

An error should occur on the **pow** function on integer or float values with respect to the values of the left and right parameters for some couple of parameters (*left = 0 and right < 0*) or (*left < 0 and right > 0*) : (0, -2), (-2,0.5), etc. Otherwise, PolySpace software prompts the user to investigate further by means of an orange check.

User Assertion: ASRT

This is a check to establish whether a user assertion is valid. If the assumption implied by an assertion is invalid, then the standard behavior of the assert macro is to abort the program. PolySpace therefore considers a failed assertion to be a runtime error. Consider the following example.

```
1 #include <assert.h>
2
3 typedef enum
4 {
5     monday=1, tuesday,
6     wensday, thursday,
7     friday, saturday,
8     sunday
9 } dayofweek ;
10
11 // stubbed function
12 dayofweek random_day(void);
13 int random_value(void);
14
15 void main(void)
16 {
17     unsigned int var_flip;
18     unsigned int flip_flop;
```

```
19 dayofweek curDay;
20 unsigned int constant = 1;
21
22 if (random_value()) flip_flop=1; else flip_flop=0; // flip_flop
randomly be 1 or 0
23 var_flip = (constant | random_value()); // var_flip is
always > 0
24
25 if(random_value()) {
26   assert(flip_flop==0 || flip_flop==1); // User Assertion is
verified
27   assert(var_flip>0); // User Assertion is
verified
28   assert(var_flip==0); // ASRT ERROR: Failure User
Assert
29 }
30
31 if (random_value()) {
32   curDay = random_day(); // Random day of the week
33   assert( curDay > thursday); // ASRT Warning: User
assertion may fails
34   assert( curDay > thursday); // User assertion is
verified
35   assert( curDay <= thursday); // ASRT ERROR: Failure User
Assertion
36 }
37 }
```

In the *main*, the *assert* function is used in two different manners:

- 1** To establish whether the values *flip_flop* and *var_flip* in the program are inside the domain which the program is designed to handle. If the values were outside the range implied by the *assert* (see line 28), then the program would not be able to run properly. Thus they are flagged as run-time errors.
- 2** To redefine the range of variables as shown at line 34 where *curDay* is restricted to just a few days. Indeed, PolySpace makes the assumption that if the program is executed without run time error at line 33, *curDay* can only have a value greater than *thursday* after this line.

Scalar and Float Underflows: UNFL

These are checks to establish whether arithmetic expressions underflow. A scalar check is used with integer type, and a float check for floating point expressions. Consider the following example.

```

1 #include <float.h>
2 extern int random_int(void);
3
4 void main(void)
5 {
6     int i = 1;
7     float fval = FLT_MAX;
8
9     i = -2 * (i << 30); // i = -2**31
10    if (random_int()) i = i - 1; // UNFL ERROR: scalar
    variable is underflow
11    if (random_int()) fval = -2 * fval; // UNFL ERROR: float
    variable is underflow
12 }
13

```

The minimum integer value on a 32 bit architecture platform is represented by -2^{31} , thus adding (-1) will raise an underflow.

Scalar and Float Overflows: OVFL

These are checks to establish whether arithmetic expressions overflow. This is a scalar check with integer type and float check for floating point expression. Consider the following example.

```

1 #include <float.h>
2 extern int random_int(void);
3
4 void main(void)
5 {
6     int i = 1;
7     float fvalue = FLT_MAX;
8
9     i = i << 30; // i = 2**30
10    if (random_int())

```

```
11  i = 2 * (i - 1) + 2; // OVFL ERROR: 2**31 is an overflow
    value for int32
12  if (random_int())
13  fvalue = 2 * fvalue + 1.0; // OVFL ERROR: float variable is
    overflow
14 }
```

On a 32 bit architecture platform, the maximum integer value is $2^{31}-1$, thus 2^{31} will raise an overflow.

In the same manner, if *fvalue* represents the biggest float its double cannot be represented with same type and raises an overflow.

Left shift overflow on signed variables: OVFL

Overflows can be also be encountered in the case of left shifts on signed variables. In the following example, the higher order bit of *0x41021011* (hexadecimal value of *1090654225*) has been lost, highlighting an overflow (integer promotion).

```
1
2 void main(void)
3 {
4  int i;
5
6  i = 1090654225 << 1; // OVFL ERROR: on left shift range
7 }
```

Float Underflows and Overflows: UOVFL

The check UOVFL only concerns float variables. PolySpace shows an UOVFL when both overflow and underflow can occur on the same operation.

```
1 #include <math.h>
2 extern int random(void);
3 #define FLT_MAX 3.40282347e+38F
4
5 int toto(void)
6 {
7  float x;
8  if (random())
```

```
9 {
10  x = -FLT_MAX;
11 }
12 else if (random())
13 {
14  x = FLT_MAX;
15 }
16 else
17 {
18  x = 0;
19 }
20 x = 2.0F * x; // UOVFL unproven: possible overflow and
underflow
21 return 1;
22 }
```

According to the branch in use, the results of the operation $2.0F * x$ could overflow or underflow.

How Much is the Biggest Float in C?

There are occasions when it is important to understand when overflow may occur on a float value approaching its maximum value. Consider the following example.

```
void main(void)
{
  float x, y;
  x = 3.40282347e+38f; // is green
  y = (float) 3.40282347e+38; // OVFL red
}
```

There is a red error on the second assignment, but not the first. The real "biggest" value for a float is: 340282346638528859811704183484516925440.0 - MAXFLOAT - .

Now, rounding is not the same when casting a constant to a float, or a constant to a double:

- floats are rounded to the nearest lower value;

- doubles are rounded to the nearest higher value;
- 3.40282347e+38 is strictly bigger than 340282346638528859811704183484516925440 (named MAXFLOAT).
- In the case of the second assignment, the value is cast to a double first - by your compiler, using a temporary variable D1 -, then into a float - another temporary variable -, because of the cast. Float value is greater than MAXFLOAT, so the check is red.
- In the case of the first assignment, 3.40282347e+38f is directly cast into a float, which is less than MAXFLOAT

The solution to this problem is to use the "f" suffix to specify the variable directly as a float, rather than casting.

What is the Type of Constants/What is a Constant Overflow?

Consider the following example, which would cause an overflow.

```
int x = 0xFFFF; /* OVFL */
```

The type given to a constant is the first type which can accommodate its value, from the appropriate sequence shown below. (See “Target Specification (size of char, int, float, double...)” on page 3-9 for information about the size of a type depending on the target.)

Decimals	int , long , unsigned long
Hexadecimals	Int, unsigned int, long, unsigned long
Floats	double

For examples (assuming 16-bits target):

5.8	double
6	int
65536	long
0x6	int

0xFFFF	unsigned int
5.8F	float
65536U	unsigned int

The options `-ignore-constant-overflows` allow the user to bypass this limitation and consider the line:

```
int x = 0xFFFF; /* OVFL */ as int x = -1; instead of 65535, which does not fit
into a 16-bit integer (from -32768 to 32767).
```

Float Underflow Versus Values Near Zero: UNFL

The definition of the word "underflow" differs between the ANSI standard and the ANSI/IEEE 754-1985 standard. According to the former definition, underflow occurs when a number is sufficiently negative for its type not to be capable of representing it. According to the latter, underflow describes the erroneous representation of a value close to zero due to the limits of its representation.

PolySpace analyses apply the former definition. The latter definition does not impose the raising of an exception as a result of an underflow. By default, processors supporting this standard permit the deactivation of such exceptions.

Consider the following example.

```
2 #define FLT_MAX 3.40282347e+38F // maximum representable
float found in <float.h>
3 #define FLT_MIN 1.17549435e-38F // minimum normalised
float found in <float.h>
4
5 void main(void)
6 {
7   float zer_float = FLT_MIN;
8   float min_float = -(FLT_MAX);
9
10  zer_float = zer_float * zer_float; // No check underflow
near zero. VOA says {[expr] =
0.0}
```

```
11 min_float = min_float * min_float; // UNFL ERROR:
underflow checked by verifier
12
13 }
```

Scalar or Float Division by Zero: ZDV

This is a check to establish whether the right operand of a division (that is, the denominator) is different from 0[.0]. Consider the following example.

```
1 extern int random_value(void);
2
3 void zdvs(int p)
4 {
5     int i, j = 1;
6     i = 1024 / (j-p); // ZDV ERROR: Scalar Division by Zero
7 }
8
9 void zdvf(float p)
10 {
11     float i,j = 1.0;
12     i = 1024.0 / (j-p); // ZDV ERROR: float Division by Zero
13 }
14
15 int main(void)
16 {
17     volatile int random;
18     if (random_value()) zdvs(1); // NTC ERROR: because of ZDV ERROR
in ZDVS.
19     if (random_value()) zdvf(1.0); // NTC ERROR: because of ZDV ERROR
in ZDVF.
20 }
```

Shift Amount in 0..31 (0..63):SHF

This is a check to establish whether a shift (left or right) is bigger than the size of the integral type operated upon (int or long int). The range of allowed shift depends on the target processor: 16 bits on *c-167*, 32 bits on *i386* for int, etc. Consider the following example.

```
1 extern int random_value(void);
2
```

```
3 void main(void)
4 {
5     volatile int x;
6     int k, l = 1024; // 32 bits on i386
7     unsigned int v, u = 1024;
8
9     if (x) k = l << 16;
10    if (x) k = l >> 16;
11
12    if (x) k = l << 32; // SHF ERROR
13    if (x) k = l >> 32; // SHF ERROR
14
15    if (x) v = u >> 32; // SHF ERROR
16    if (x) k = u << 32; // SHF ERROR
17
18 }
```

In this example, it is shown that the shift amount is greater than the integer size.

Left Operand or Left Shift is Negative: SHF

This is a check to establish whether the operand of a left shift is a signed number. Consider the following example.

```
1
2
3 void main(void)
4 {
5     int x = -200;
6     int y;
7
8     y = x << 1; // SHF ERROR: left operand must be positive
9
10 }
```

As an aside, note that the `-allow-negative-operand-in-shift` option used at launching time instructs PolySpace software to permit explicitly signed numbers on shift operations. Using the option in the example above would see the red check at line 8 transformed in a green one. Similarly, if the analysis

had included the expression `-2 << 2` at line 9, then that line would have been given a green check and `y` would assume a values of `-8`.

Function Pointer Must Point to a Valid Function: COR

This is a check to establish whether a function pointer points to a valid function, or to function with a valid prototype. Consider the following example.

```
1
2 typedef void (*Callback)(float *data);
3
4 struct {
5     int ID;
6     char name[20];
7     Callback func;
8 } funcS;
9
10 float fval;
11
12 void main(void)
13 {
14     Callback cb = (Callback)((char*)&funcS + 24*sizeof(char));
15
16     cb(&fval); // COR ERROR: function pointer must point
17               to a valid function
18 }
```

In the example above, *func* has a prototype in conformance with *Callback*'s declaration. Therefore *func* is initialized to point to the NULL function through the global declaration of *funcS*.

Consider a second example.

```
1
2 #define MAX_MEMSEG 32764
3 typedef void (*ptrFunc)(int memseg);
4 ptrFunc initFlash = (ptrFunc)(0x003c);
5
6 void main(void)
7 {
8     int i;
```



```
9
10 for (i = 0 ; i < MAX_MEMSEG; i++) // NTL propagation
11 {
12   initFlash(i); // COR ERROR: function pointer must point to a
    valid function
13 }
14
15 }
```

As PolySpace verification does not take the memory mapping of programs into account, it cannot ascertain whether *0x003* is the address of a function code segment or not (for instance, as far as PolySpace is concerned it could be a data segment). Thus a certain (red) error is raised.

Wrong Type for Argument: COR

This is a check to establish whether each argument passed to a function matches the prototype of that function. Consider the following example.

```
1
2 typedef struct {
3   float r;
4   float i;
5 } complex;
6
7 typedef int (*t_func)(complex*);
8
9 int foo_type(int *x)
10 {
11   if (*x%2 == 0) return 0;
12   else return 1;
13 }
14
15 void main(void)
16 {
17   t_func ptr_func;
18   int j,i = 0;
19
20   ptr_func = foo_type;
21   j = ptr_func(&i); // COR ERROR: wrong type of argument for #1
```

```
22 }  
23
```

In this example, *ptr_func* is a pointer to a function which expects a pointer to a *complex* as input argument. However, the parameter used is a pointer to an *int*.

Wrong Number of Arguments: COR

This is a check to establish whether the number of arguments passed to a function matches the number of arguments in its prototype. Consider the following example.

```
1  
2 typedef int (*t_func_2)(int);  
3 typedef int (*t_func_2b)(int,int);  
4  
5 int foo_nb(int x)  
6 {  
7     if (x%2 == 0)  
8         return 0;  
9     else  
10        return 1;  
11 }  
12  
13  
14 void main(void)  
15 {  
16     t_func_2b ptr_func;  
17     int i = 0;  
18  
19     ptr_func = (t_func_2b)foo_nb;  
20     i = ptr_func(1,2); // COR ERROR: the wrong number of arguments  
21 }  
22
```

In this example, *ptr_func* is a pointer to a function that takes two arguments but it has been initialized to point to a function that only takes one.

Wrong Return Type of a Function Pointer: COR

This is a check to establish whether the return type passed to a function pointer matches the declaration in its prototype. Consider the following example.

```
1
2 typedef int (*t_func_2)(int);
3 typedef double (*t_func_2b)(int);
4
5 int foo_nb(int x)
6 {
7     if (x%2 == 0)
8         return 0;
9     else
10        return 1;
11 }
12
13
14 void main(void)
15 {
16     t_func_2b ptr_func;
17     int i = 0;
18
19     ptr_func = (t_func_2b)foo_nb;
20     i = ptr_func(1,2); // COR ERROR: function pointer must
                        // point on a valid function
21                       // COR Warning: return type of function
                        // is INT but a FLOAT was expected
22 }
23
```

In this example, *ptr_func* is a pointer to a function that return a double but it has been initialized to point to a function that returns an int. The understanding of the red error is given in the orange associated COR message.

Wrong Return Type for Arithmetic Functions: COR

This is a check to establish whether that a wrong return type is used for an arithmetic function.

Using arithmetic functions without including `<math.h>` is compiler dependent in the real world because compiler could associate a integral return type to an implicit function.

However, as arithmetic functions are built-in in PolySpace software, you can face an inconsistency problem `<math.h>` is not explicitly included in the code file where an arithmetic function is used. All arithmetic function declared in `<math.h>` are concerned.

Consider the following examples:

Results without `<math.h>`:

```
1
2 int main(void) {
3
4 double x;
5 x = cos(2*3.1415); // COR ERROR: return type of
function cos is INT 32 but a float 64 was expected
6 }
```

Results with `<math.h>`:

```
1 #include <math.h>
2 int main(void) {
3
4 double x;
5 x = cos(2*3.1415);
6 }
```

In the previous example without the definition of `<math.h>`, `cos` is declared without prototype and default return type is an `int32`.

Pointer Within Bounds: IDP

This is a check to establish whether a dereference pointer is still within the bounds of the object it intended to point to.

Consider the following example.

```
1
```

```
2 #define TAILLE_TAB 1024
3 int tab[TAILLE_TAB];
4 int *p = tab;
5
6 void main(void)
7 {
8
9     int index;
10
11     for (index = 0; index < TAILLE_TAB ; index++, p++)
12     {
13         *p = 0;
14     }
15
16     *p = 1; // IDP ERROR: pointer is outside its bound
17 }
```

In the example, the pointer *p* is initialized to point to the first element of the *tab* array at line 4. When the loop is exited, *p* points beyond the last element of the array.

Thus line 16 overwrites memory illegally.

Understanding Addressing

- “I Systematically Have an Orange Out of Bounds Access On My Hardware Register” on page 8-35
- “The NULL Pointer Case” on page 8-37
- “Comparing Address” on page 8-40

I Systematically Have an Orange Out of Bounds Access On My Hardware Register. Many code analyses exhibit **orange** out of bound checks with respect to accesses to absolute addresses and/or hardware registers.

(Also refer to the discussion on Absolute Addressing)

Here is an example of what such code might look like:

```
#define X (* ((int *)0x20000))
X = 100;
y = 1 / X; // ZDV check is orange because X ~ [-2^31, 2^31-1] permanently.
// The pointer out of bounds check is orange because 0x20000
// may address anything of any length
// NIV check is orange on X as a consequence
```

```
Expanded source code - PolySpace Viewer
Expanded source code :
|8      y = 1 / (* ((int *)0x20000)) ;
```

```
Expanded source code - PolySpace Viewer
Expanded source code :
|7      (* ((int *)0x20000)) = 100;
```

```
3 void main (void)
4 {
5 int y;
6
7 X = 100;
8 y = 1 / X;
9
10 }
```

```
int *p = (int *)0x20000;
*p = 100;
y = 1 / *p; // ZDV check is orange because *p ~ [-2^31, 2^31-1] permanently
// The pointer out of bounds is orange because 0x20000
// may address anything of any length
// NIV check on *p is orange as a consequence
```

This can be addressed by defining registers as regular variables:

Replace	By
#define X	int X;

Replace	By
<code>int *p;</code>	<code>int _p;#define p (&_p)</code> <hr/> <p>Note Check that the chosen variable name (p in this example) doesn't already exist</p> <hr/>
<code>int *p;</code>	<code>volatile int _p;int *p = &_p;</code>

See “Volatile” on page 3-68 for a discussion of an approach which will help avoid the orange check on the pointer dereference, but retains the representation of a “full range” variable.

The NULL Pointer Case. Consider the NULL address, viz.

```
#define NULL ((void *)0)
```

- It is illegal to dereference this NULL address;
- **0 is not** treated as an absolute address.

```
*NULL = 100; //produces a red - Illegal Dereference Pointer (IDP)
```

Assuming these declarations:-

```
int *p = 0x5;  
volatile int y;
```

and these definitions:-

```
#define NULL ((void *) 0)  
#define RAM_MAX ((int *)0xffffffff)
```

consider the code snippets below.

```
While (p != (void *)0x1)  
p--; // terminates
```

0x1 is an absolute address, it can be reached and the loop terminates


```
for (p = NULL; p <= RAM_MAX; p++)
{
    *p = 0; // illegal dereference of pointer
}
```

At the first iteration of the loop `p` is a `NULL` pointer. Dereferencing a `NULL` pointer is forbidden.

```
While (p != NULL)
{
    p--;
    *p = 0; // Orange dereference of a pointer
}
```

When `p` reaches the address `0x0`, there is an attempt to considered it as an absolute address

In effect, it is an attempt to dereference a `NULL` pointer - which is forbidden.

Note In this case, the **check is orange** because the execution of the code here is OK (green) until `0x0` is reached (red)

The best way to address this issue depends on the purpose of the function.

- Thanks to the default behavior of PolySpace software, it is easy to automatically stub a function whose purpose is to copy data from/to RAM or to compute a checksum on RAM.
- If a function is supposed to copy calibration data, it should also be stubbed automatically.
- If the purpose of a function is to map EEPROM data to global variables, then a manually written stub is essential to ensure the assignment of the correct initialization values to them.

Comparing Address. PolySpace software only deals with the information referred to by a pointer, and not the physical location of a variable. Consequently it does not compare addresses of variables, and makes no assumption regarding where they are located in memory.

Consider the following two examples of PolySpace verification behavior:

```
int a,b;
if (&a > &b) // condition can be true and/or false
{ } // both branches are reachable
else
{ } // both branches are reachable
```

and

```
int x,z;
void main(void)
{ int i;
  x = 12;
  for (i=1; i<= 0xffffffff; i++)
  {
    *((int *)i) = 0;
  }
  z = 1 / x; // ZDV green check because PolySpace doesn't consider
            // any relationship between x and its address
}
```

“x” is aliased by no other variable. No pointer points to “x” in this example, so as far as the PolySpace analysis is concerned, “x” remains constantly equal to 12.

Understanding Pointers

PolySpace software does not analyze anything which would require the physical address of a variable to be taken into account.

- Consider two variables x and y. PolySpace analysis will not make a meaningful comparison of “&x” (address of x) and “&y”

- So, the Boolean ($\&x < \&y$) can be true or false as far as PolySpace analysis is concerned.

However, PolySpace analysis does keep track of the pointers that point to a particular variable.

- So, if `ptr` points to `X`, `*ptr` and `X` will be synonyms.

Address Alignment: the bitfield Example. Structure size depends on bit alignment.

Consider the following example, where an attempt is made to map a character to a bitfield.

```
struct reg {
    unsigned int a: 5;
    unsigned int b: 3;
};
int main()
{
    volatile unsigned char c;
    struct reg *r;
    r = (struct reg *) &c;
    if (r-> a == 10)
        return 1;
    return 0;
}
```

Consider a 32 bit target architecture (so `int` are 32 bits, i.e. 4 bytes). The size of a bit field is the size of the type of its elements. In the example above, the elements in the bit field are `unsigned int`, hence the size is 4 bytes. Since this is greater than 1, the structure `reg` cannot be contained in the `char c`.

This can be solved by using the `unsigned char` type for the elements in the bit field. The size of the bit field is then 1 byte and there is therefore no red error.

```
struct reg {
    unsigned char a: 5;
    unsigned char b: 3;
};
int main()
```

```
{
  volatile unsigned char c;
  struct reg *r;
  r = (struct reg *) &c;
  if (r-> a == 10)
    return 1;
  return 0;
}
```

Note You must also use the option `-allow-non-int-bitfield` to implement this solution, since this is an extension to the ANSI® standard.

How Does malloc Work for PolySpace Verification? PolySpace analysis accurately models malloc, such that both the possible return values of a null pointer and the requested amount of memory are taken into account.

Consider the following example.

```
void main(void)
{
  char *p;
  char *q;
  p = malloc(120);
  q = p;
  *q = 'a'; // results in an orange dereference check
}
```

This code will avoid the orange dereference:

```
void main(void)
{
  char *p;
  char *q;
  p = malloc(120);
  q = p;
  if (p!= NULL)
    *q = 'a'; // results in a green dereference check
}
```

Data Mapping into a Structure . It often happens that structured data are read as a char array. Before manipulating them it might be desirable to map those data into a structure that reflects their organization. In the following example an IDP warning (orange check) at line 22 suggests that the correctness of the code needs to be confirmed.

```
1
2
3 typedef struct
4 {
5     unsigned int MsgId;
6     union {
7         float fltv;
8         unsigned int intv;
9     } Msgbody;
10 } Message;
11
12 int random_int(void);
13 Message *get_msg(void);
14 void wait_idl(void);
15
16 void treatment_msg(char *msg)
17 {
18     Message *ptrMsg;
19
20     ptrMsg = (Message *)msg;
21     if (ptrMsg != NULL) {
22         if (ptrMsg->MsgId) { // IDP Warning: pointer may be
outside its bounds
23             // ...
24         }
25     }
26 }
27
28 int main (void) {
29
30     Message *msg;
31
32     while(random_int()) {
33         msg = get_msg();
```

```
34  if (msg) treatment_msg((char *)msg);
35  wait_idl();
36  }
37  return 0;
38  }
```

Mapping of a small structure into a bigger one. For example, suppose that p is a pointer to an object of type t_struct and it is initialized to point to an object of type t_struct_bis .

Now suppose that the size of t_struct_bis is less than the size of t_struct . Under these circumstances, it would be illegal to dereference p because it would be possible to access memory outside of t_struct_bis .

Consider the following example.

```
1 #include <malloc.h>
2
3 typedef struct {
4   int a;
5   union {
6     char c;
7     float f;
8   } b;
9 } t_struct;
10
11 void main(void)
12 {
13   t_struct *p;
14
15   // optimize memory usage
16   p = (t_struct *)malloc(sizeof(int)+sizeof(char));
17
18   p->a = 1; // IDP ERROR: not allowed to dereference p
19
20 }
```

Partially allocated pointer (-size-in-bytes). According to the ANSI standard, the whole of a structure must be populated for that structure to be valid. In this case, the pointer is said to be fully allocated. A pointer is said to be partly allocated when only the first part of a structure is populated. In some development environments, that approach is tolerated despite the ANSI stance.

By default, PolySpace verification strictly conforms to the standard and checks for adherence to it. A more tolerant approach can be specified by using the `-size-in-bytes` option. So, depending on the `-size-in-bytes` option, when a partially allocated pointer is encountered during a PolySpace analysis, the first elements of the allocated object may or may not be considered as valid.

First consider the following example. (A second example follows it to illustrate how this might apply to pointer arithmetic within a structure)

```

1 typedef struct _little { int a; int b; } LITTLE;
2 typedef struct _big { int a; int b; int c; } BIG;
3
4 int main(void)
5 {
6     BIG *p = malloc(sizeof(LITTLE));
7     volatile int y;

```

With `-size-in-bytes` option

```

9     if (p==((void *)0)) return 0;
10    if(y) { p->a = 0; } // green
11    if(y) { p->b = 0; } // green
12    if(y) { p->c = 0; } // red
    }

```

Default launching option

```

9     if(y) { p->a = 0 ; } // red
10    if(y) { p->b = 0 ; } // red
11    if(y) { p->c = 0 ; } // red
12
13    if (p==((void *)0))
14        return 0;
15    else

```

```
16     return 1; // dead code
17     return 1;
18 }
```

With the standard launching option, a pointer that has not been allocated to a complete structure is considered invalid, or NULL (as shown in the dead code).

This second example illustrates how this might apply to pointer arithmetic within a structure

```
1 typedef struct _inside { int a; int b; } INSIDE;
2 typedef struct _outside { int a; INSIDE x; } OUTSIDE;
3
4 OUTSIDE out;
5
6 void main(void)
7 {
8     unsigned char *ptr = (unsigned char *) &out;
9     INSIDE *p = (INSIDE *) (ptr + sizeof(int));
```

With `-size-in-bytes` option

```
11 p->b = 100; // green
```

Default launching option

```
11 p->b = 100; // red
```

With the default launching option and in accordance with the ANSI standard, the size of the `INSIDE` structure function implies that there is only one such structure within the `OUTSIDE` structure. Therefore, `p` has passed that one, and is out of bounds. With the `-size-in-bytes` option, the dereference check is green because since the pointer remains within the structure.

Pointer to a structure field. According to the ANSI C standard, pointer arithmetic is to be independent of the size of the object (structure or array) to which the pointer points. By default, PolySpace verification strictly conforms to the standard and checks for adherence to it.

In some development environments an approach that does not recognize that requirement is tolerated, despite the ANSI stance. Under those circumstances, results are likely to include **red** pointer out of bounds **checks** unexpectedly.

A more tolerant approach can be specified at launch time. Consider the following examples.

```
char *p; // the size of the object pointed to is unknown,
// but arithmetic on this pointer is well defined.
// p = p + 5; will increment the location pointed to by
5 bytes (if the
size of a char is 1 byte)
int x; // assuming that an int is 4 bytes
p = &x; *p = 0; // the first byte of x
p++; *p = 0; // the second byte of x
p++; *p = 0; // the third byte of x
p++; *p = 0; // the fourth byte of x
p++; *p = 0; // an out of bound access
```

For structures, the same behavior can be applied.

```
struct { int a; int b; } x;
char *p = &x.a; // the pointed object is not the structure
but the field
*p = 0; // it is the first byte of x.a
p++; *p = 0; // it is the second byte of x.a
p++; *p = 0; // it is the third byte of x.a
p++; *p = 0; // it is the fourth byte of x.a
p++; *p = 0; // here is an out of bound access because
we are out of the field
```

If you wish to tolerate an approach which allows a pointer to go from one field to another, you can do so by using the `-size-in-bytes` option **together with** the `-allow-ptr-arith-on-struct` option. When a pointer points to a field in a structure, you will then be allowed to access other fields from this pointer. Note that as a consequence, any other "out of bound" accesses in the code will be ignored.

An alternative solution is to make your variable point to the structure rather than to the field, as follows:

```

struct { int a; int b; } x;
char *p = &x; // the pointed object is the structure
*p = 0; // we are modifying x.a (first byte)
p++; *p = 0; // we are modifying x.a (second byte)
p++; *p = 0; // we are modifying x.a (third byte)
p++; *p = 0; // we are modifying x.a (fourth byte)
p++; *p = 0; // we are modifying x.b (fifth byte of the structure)

```

A further alternative is to follow the ANSI C recommendation to use the “`offsetof()`” function, which jumps to the corresponding offset within the structure:-

```

#include <stddef.h>
typedef struct _m { int a; int b; } S;
S x;
char *p = (char *) &x + offsetof(S,b); // points to field b

```

I have a red when reading a field of one structure . Consider the following example.

```

5 typedef struct {
6   unsigned char c1;
7   unsigned char c2;
8 } my_struct;
9
10 int main(void)
11 {
12   my_struct v;
13   unsigned short x=0,y=0;
14
15   v.c1=9;
16   v.c2=15;
17   x = *((unsigned short *)&v.c1);

```

Just like the example in “Pointer to a structure field” on page 8-46, the object pointed to is the field in the structure, not the structure itself. Therefore, it is only possible to navigate inside this field. A short variable occupies more memory than a char, so it is a red pointer out of bounds.

This can be addressed by replacing

```
x = *((unsigned short *)&v.c1);
```

with

```
y = (v.c1 <<sizeof(v.c2)*8 ) | v.c2;
```

This solution also ensures that the code is no longer target dependent.

Non Termination of Call or Loop

NTC and NTL are informative red (or orange) checks.

- They are the only red checks which can be filtered out as shown below
- They don't stop the analysis
- As for other red checks, code found after them are grey (unreachable)
- These checks may only be red. There are no "orange" NTL or NTC checks.
- They can reveal a bug, or can simply just be informative

NTL	<p>In a Non Terminating Loop, the break condition is never met. Here are some examples.</p> <pre>while(1) { function_call(); } // informative NTL</pre> <p>while(x>=0) {x++; } // where x is an unsigned int. This may reveal a bug?</p> <p>for(i=0; i<=10; i++) my_array[i] = 10; // where “int my_array[10];” applies. This red NTL reveals a bug in the array access, flagged in orange</p> <p>ptr = NULL; for(i=0; i<=100; i++) *ptr=0; // the first iteration of the loop is red, and therefore it is flagged as an NTL. The “i++” will be grey, because the first iteration crashed.</p>
NTC	<p>Suppose that a function calls f(), and that function call is flagged with a red NTC check. There could be five distinct explanations:</p> <ul style="list-style-type: none"> • “f” contains a red error; • “f” contains an NTL ; • “f” contains an NTC; • “f” contains an orange which is context dependant; that is, it is either red or green. For this particular call, it makes the function “f” crash. • “f” is a mathematical function, such as sqrt, acos which has always an invalid input parameter <p>Remember, additional information can be found when clicking on the NTC</p>

Note A sqrt check is only colored if the input parameter is **never** valid. For instance, if the variable x may take any value between -5 and 5, then sqrt(x) has no color.

The list of constraints which cannot be satisfied (found by clicking on the NTC check) represents the variables that cause the red error inside the function.

The (potentially) long list of variables can help to understand the cause of the red NTC, as it shows each condition causing the NTC

- where the variable has a given value; and
- where the variable is not initialized. (Perhaps the variable is initialized outside the set of files under analysis?).

If a function is identified which is not expected to terminate (such as a loop or an exit procedure) then the `-known-NTC` function is an option. You will find all the NTCs and their consequences in the `k-NTC` facility in the Viewer, allowing you to filter them.

Non Termination of a Call: NTC

This is a check to establish whether a procedure call returns. It is not the case when the procedure contains an endless loop or a certain error, or if the procedure calls another procedure which does not terminate. In the latter instance, the status of this check is propagated to caller.

```
1
2
3 void foo(int x)
4 {
5   int y;
6   y = 1 / x; // Warning ZDV: its depends of the context
7   while(1) { // NTL ERROR: loop never terminates
8     if ( y != x) {
9       y = 1 / (y-x);
10    }
11  }
12 }
13
14 void main(void) {
15   volatile int _x;
16
17   if (_x)
18     foo(0); // NTC ERROR: Zero DiVision (ZDV) in foo
19   if (_x)
20     foo(2); // NTC ERROR: Non Termination Loop (NTL) in foo
21
```

```
22 }  
23
```

In this example, the function *foo* is called twice in *main* and neither of these 2 calls ever terminates.

- 1 The first never returns because a division by zero occurs at line 6 (bad argument value),
- 2 The second never terminates because of an infinite loop (red NTL) at line 7.

Also with reference to the example and as an aside, note that by using either the `-context-sensitivity "foo"` option or the `-context-sensitivity-auto` option at launch time it would be possible for PolySpace verification to show explicitly that a ZDV error comes from the **first** call of *foo* in *main*.

Note An NTC check can only be red or uncolored, unless you use the `-context-sensitivity` option. If you use the `-context-sensitivity` option, NTC checks can also be orange.

Known Non-Termination of a Call: k-NTC

By using the `-known-NTC` option with a specified function at launch time it is possible to transform an NTC check to a k-NTC check. Like NTC checks, k-NTC checks are propagated to their callers. Functions designed not to terminate can then be filtered out through the use of the appropriate filter in the viewer.

Consider the following example, supposing that `-know-NTC "SysHalt"` option has been applied at launch time.

```
1  
2 /* external get data function */  
3 extern int get_data(int *ptr,void *data);  
4 extern int printf (const char *, ...);  
5  
6 // known NTC function  
7 void SysHalt(int value)  
8 {
```

```
9  printf("Halt value %d",value);
10 while (1) ; // NTL ERROR: Loop Never Terminate
11 }
12
13 #define OK 1
14 int main(void)
15 {
16  int data, *ptr = NULL;
17  int status = OK;
18
19  // get next store
20  status = get_data(ptr,(void *)&data);
21  if (status != OK)
22  SysHalt(status); // k-NTC check: Call never
terminate
23
24  return(0);
25 }
```

In the example, the relevant NTC check is converted to a k-NTC one.

Non Termination of Loop: NTL

This is a check to establish whether a loop (for, do-while or while) terminates. Consider the following example:

```
1
2 // Function prototypes
3 void send_data(double data);
4 void update_alpha(double *a);
5
6 void main(void)
7 {
8  volatile double _acq;
9  double acq, filtered_acq, alpha;
10
11  // Init
12  filtered_acq = 0.0;
13  alpha = 0.85;
14
15  while (1) { //NTL ERROR: Non Termination Loop
```

```
16 // Acquisition
17 acq = _acq;
18 // Treatment
19 filtered_acq = acq + (1.0 - alpha) * filtered_acq;
20 // Action
21 send_data(filtered_acq);
22 update_alpha(&alpha);
23 }
24 }
```

In the example, the continuation condition is always true and the loop will never exit. PolySpace verification will raise an error in trivial examples such as this, and in much more complex circumstances.

Consider this second analysis. When an error is found inside a for, do-while, or while loop, PolySpace will not continue to propagate it.

```
1
2 void main(void)
3 {
4   int i;
5   double twentyFloat[20];
6
7   for (i = 0; i <= 20; i++) { // NTL ERROR: propagation of OBAI ERROR
8     twentyFloat[i] = 0.0; // OBAI Warning: 20 verification with i
9     in [0,19] and one ERROR with i = 20
10  }
```

At line 8 in this second example, the **red** OBAI related to the **21th** execution of the loop has yielded the **orange check**. The 20 first executions would be no problem, so this **orange** warning represents a combination of **red** and **green** checks.

Note An NTL check can only be **red** or uncolored, unless you use the `-context-sensitivity` option. If you use the `-context-sensitivity` option, NTL checks can also be orange.

Arithmetic Expressions: NTC

This is a check to establish whether standard arithmetic functions are used with valid arguments, as defined in the following:

- Argument of *sqrt* must be positive (ISO®/IEC 9899 section 7.5.5.2)
- Argument of *tan* must be different from $\pi/2$ modulo π (ISO/IEC 9899 section 7.5.2.7)
- Argument of *log* must be strictly positive (ISO/IEC 9899 section 7.5.4.4)
- Argument of *acos* and *asin* must be within $[-1..1]$ (ISO/IEC 9899 sections 7.5.2.1 and 7.5.2.2)
- Argument of *exp* must be less than or equal to 709 (ISO/IEC 9899 section 7.5.4.1)
- Argument of *atanh* must be within $] -1..1[$ (ISO/IEC 9899 section 7.12.5.3)
- Argument of *acosh* must be greater or equal to 1 (ISO/IEC 9899 section 7.12.5.1)

A domain error (such that *errno* returns *EDOM*) occurs if an input argument is outside the domain over which the mathematical function is defined. A range error occurs (such that *errno* returns *ERANGE*) if the result cannot be represented as a double value. In the latter case, the function returns 0 if the result is too small, or *HUGE_VAL* with the appropriate sign if it is too big.

Consider the following example

```
1
2 #include <math.h>
3 #include <assert.h>
4
5 extern int random_int(void);
6
7 int main(void)
8 {
9
10  volatile double dbl_random;
11  const double dbl_one = 1.0;
12  const double dbl_mone = -1.0;
13
```

```
14 double sp      = dbl_random;
15 double p       = dbl_random;
16 double sn      = dbl_random;
17 double n       = dbl_random;
18 double no_trig_val_neg = dbl_random;
19 double no_trig_val_pos = dbl_random;
20 double pun     = dbl_random;
21 double res;
22
23 // assert is used here to redefine range values of variables
24 assert(sp > 0.0);
25 assert(p >= 0.0);
26 assert(sn < 0.0);
27 assert(n <= 0.0);
28 assert(pun < 1.0);
29 assert(no_trig_val_neg < -1.0); assert(no_trig_val_pos > 1.0);
30
31 if (random_int()) res = sqrt(sn);          // NTC ERROR:
need argument positive
32 if (random_int()) res = asin(no_trig_val_neg); // NTC ERROR:
need argument in range [-1..1]
33 if (random_int()) res = asin(no_trig_val_pos); // NTC ERROR:
need argument in range [-1..1]
34 if (random_int()) res = acos(no_trig_val_pos); // NTC ERROR:
need argument in range [-1..1]
35 if (random_int()) res = acos(no_trig_val_neg); // NTC ERROR:
need argument in range [-1..1]
36 if (random_int()) res = tan(1.5707963267948966); // NTC ERROR:
need argument in range ]-pi/2..pi/2[
37 if (random_int()) res = log(n);           // NTC ERROR:
need argument strictly positive
38 if (random_int()) res = exp(710);        // NTC ERROR:
need argument less or equal to 709
39
40 // No information about asin or acos because of random value
41 if (random_int()) {
42   res = asin(dbl_random);
43   res = acos(dbl_random);
44 }
45
```

```

46 // hyperbolic functions are available in the float range
47 if (random_int()) {
48   res = cosh(710);
49   res = cosh(10.0);
50   assert (res < 1.0);
51 }
52 if (random_int()) res = sinh(710);
53 if (random_int()) {
54   res = tanh(1.0);
55   assert (res > -1.0 && res < 1.0);
56 }
57
58 // inverted hyperbolic functions
59 if (random_int()) res = acosh(pun);      // NTC ERROR:
Need argument >= 1
60 else res = acosh(1.0);
61 if (random_int()) res = atanh(no_trig_val_neg); // NTC ERROR:
Need argument in ]-1..1[
62 if (random_int()) res = atanh(no_trig_val_pos); // NTC ERROR:
Need argument in ]-1..1[
63 if (random_int()) res = atanh(dbl_mone);    // NTC ERROR:
Need argument in ]-1..1[
64 if (random_int()) res = atanh(dbl_one);    // NTC ERROR:
Need argument in ]-1..1[
65
66 return 0;
67 }
68

```

sqrt, *tan*, *asin*, *acos*, *exp* and *log* errors are derived directly from the mathematical definition of functions. PolySpace verification highlights any definite problems by means of an NTC to show that this is where execution would terminate. No NTC information is delivered when Verifier cannot determine the exact value of the argument, (for *asin* and *acos* at lines 42 and 43). No range restriction is currently made for hyperbolic functions.

The *pow* function benefits from a specific check POW.

Caution Due to a lack of precision in some areas, PolySpace verification is not always able to indicate a red NTC check on mathematical functions even where a problem exists. In the following example involving a *sqrt* function, neither an orange nor a red check is shown on line16 even though the variable *val2* is negative.

By default it is important to consider each call to any mathematical functions as though it had been highlighted by an **orange check**, and could therefore lead to a runtime error.

```
1
2 #include <math.h>
3
4 extern int random_int(void);
5
6 int main(void)
7 {
8
9  double val1, val2;
10
11  int i;
12  val2 = 5.0;
13  for (i = 0 ; i < 10 ; i++) {
14    val2 = val2 - 1.0;
15  }
16  val1 = sqrt(val2); // No check on sqrt
17  return ((int)val1);
18 }
19
```

Unreachable Code: UNR

This is a check to establish whether different code snippets (assignments, returns, conditional branches and function calls) are dead, such that they can never be accessed during the normal execution of the software. Dead, or Unreachable, code is represented by means of a grey coding on every check, with supplementary UNR checks also being added.

Consider the following example.

```
1
2 #define True 1
3 #define False 0
4
5 typedef enum {
6   Intermediate, End, Wait, Init
7 } enumState;
8
9 // pure stub
10 int intermediate_state(int);
11 int random_int(void);
12
13 int State (enumState stateval)
14 {
15   volatile int random;
16   int i;
17   if (stateval == Init) return False;
18   return True;
19 }
20
21 int main (void)
22 {
23   int i, res_end;
24   enumState inter;
25
26   res_end = State(Init);
27   if (res_end == False) {
28     res_end = State(End);
29     inter = (enumState)intermediate_state(0);
30     if (res_end || inter == Wait) { // UNR code on inter
31       == Wait
32       inter = End;
33     }
34     // use of I not initialized
35     if (random_int()) {
36       inter = (enumState)intermediate_state(i); // NIV ERROR
37       if (inter == Intermediate) { // UNR code because
38         of NIV ERROR
39         inter = End;
40       }
41     }
42   }
43 }
```

```
39  }
40  } else {
41    i = 1; // UNR code
42    inter = (enumState)intermediate_state(i); // UNR code
43  }
44  return res_end;
45  }
46
```

The example illustrates three possible reasons why code might be unreachable, and hence be colored **grey**:

- At line 30 the first part of a two part test is always true. The other part is never evaluated, following the standard definition of logical operator "||".
- The piece of code after a red error is never evaluated by PolySpace software. The call to the function on line 35 and the line following it are considered to be dead code. Correcting the red error and re-launching would allow the color to be revised.
- At line 27, the test is always true (*if*-*if* part), and the first branch is always executed. Consequently there is dead code in the other branch (i.e. in the *else* part at lines 41 to 42).

Value on Assignment: VOA

This is a check to establish the range or values which a variable may take, each time an assignment is made to it. Such checks are only available when the **-voa** option is used at launch time. VOA checks are only available on scalar variables.

Consider the following example.

```
1
2
3 typedef enum {
4   dOff=0, dOn
5 } t_digital ;
6
7 #define MAX_ANA (9.999)
8 #define MIN_ANA (-10.0)
9 #define ZERO_ANA ((MAX_ANA - MIN_ANA)/2.0 - MAX_ANA)
```

```
10
11 float get_analogic (int);
12 int get_digit (int);
13
14 typedef enum {Red, Green, Orange, Black} VerifierColor;
15
16 typedef struct {
17     float a;
18     VerifierColor b;
19     int c;
20 } Record;
21
22 int main(void)
23 {
24     volatile int var_int;
25     volatile float volatile_float;
26     t_digital var_digit;
27     Record var_rec;
28     int i;
29     float var_sensor;
30     VerifierColor var_color = Green;    // Currently no
VOA on enum
31
32     var_digit = dOff; // no VOA
33     var_sensor = (float)(ZERO_ANA);    // VOA: {[expr] <=
FLT_MAX} and {FLT_MIN <= [expr]}
34     for (i = 0 /* VOA: {[expr]=0} */ ; i < 8 ; i++) { // VOA: {1<=[expr]
<=8}
35         var_sensor = get_analogic(i);    // VOA: currently
not concise
36         var_digit = (t_digital)get_digit(i); // no VOA
37     }
38
39     // Float examples
40     var_sensor = volatile_float;    // VOA: currently
not concise
41     var_sensor = MAX_ANA;           // VOA: {[expr]
=9.9989}
42
43     var_rec.a = var_sensor;         // Curently no VOA
```

```
on structures
44 var_rec.b = var_color;
45 var_rec.c = 5;
46 }
```

Note that inspection points (IPT) can be used to discover the possible range of a variable at any point in the code – not just where a value is assigned.

Inspection Points: IPT

The use of **#pragma Inspection_Point <var>** in code submitted for an analysis (where *<var>* is a scalar variable), instructs PolySpace verification to reveal the possible range of a variable at that point in the code.

Consider the example below.

```
1
2 typedef struct {
3   unsigned char msb;
4   unsigned char lsb;
5 } int16;
6
7 int main(void)
8 {
9   volatile unsigned char var_uc;
10  float var_float;
11  int i;
12  int16 val;
13
14  #pragma Inspection_Point var_uc // IPT: {main:var_uc=0..256}
15  i = 3;
16  #pragma Inspection_Point i // IPT: {main:i=3}
17  val.msb = 12;
18  val.lsb = var_uc;
19  #pragma Inspection_Point val // IPT currently ignored
20  var_float = 10.0;
21  #pragma Inspection_Point var_float // IPT currently ignored
22
23 }
24
```


25

Note Inspection points at lines 19 and 21 are ignored.

PolySpace™ Methodological Guide

Overview (p. 9-2)	Describes how PolySpace™ verification can be used during the project development cycle
PolySpace™ Usage (p. 9-5)	Describes how PolySpace software can be used
PolySpace™ Activities (p. 9-20)	Describes regular activities you can do to maximize your results
Automatically Testing Orange Code (p. 9-33)	Describes how to use the Automatic Orange Tester feature
How to Get the Best Results (p. 9-56)	Describes how to use PolySpace software efficiently
Applying Coding Rules to Reduce Oranges (p. 9-87)	Describes how to utilize MISRA® rules to reduce orange checks

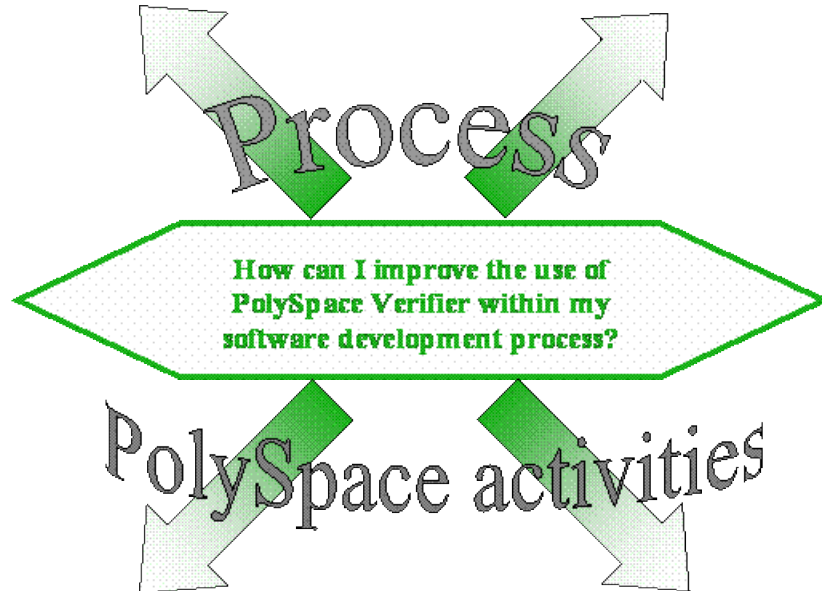
Overview

This chapter will be of interest to **Project managers, quality managers and developers** who are looking to understand PolySpace™ results, and are looking to optimize the timing of its use during the project development cycle. The document suggests how PolySpace verification might best be applied at each phase of a typical project lifecycle. The twin goals of productivity and quality are considered, and it is acknowledged that the criticality of the application will affect the balance between them.

However, the following assumes that the primary goal is to achieve maximum productivity with no quality defects. The document explains how to use PolySpace tools at each phase of the development cycle to aim for such a goal, with the financial implications of implementing each recommendation is left for assessment by the user.

How can I use PolySpace in my current process?

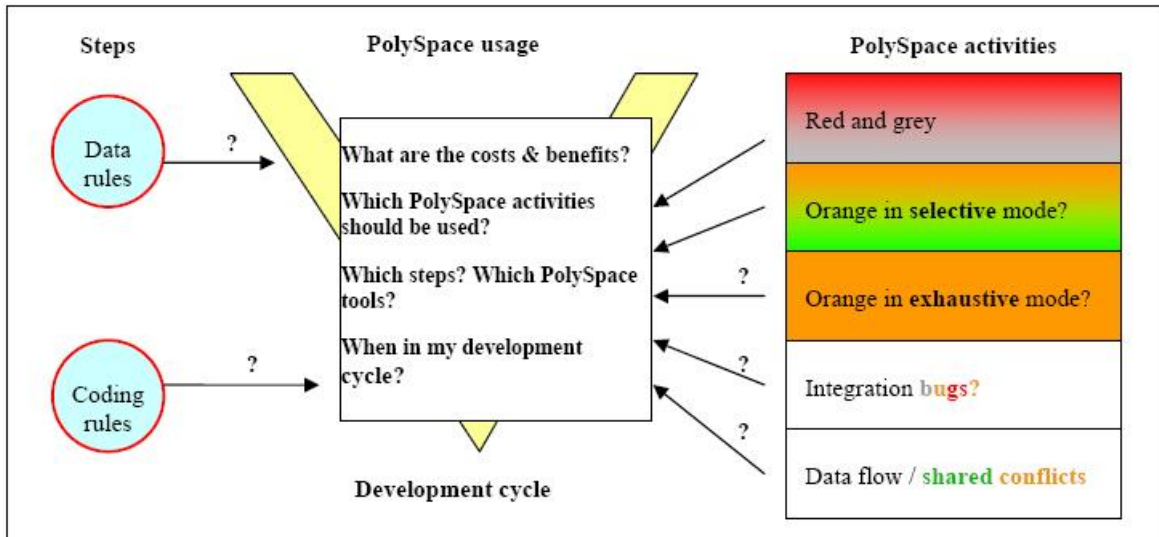
How can I change the process to get more out of PolySpace?



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.



It answers those questions by means of the following topics: an explanation of each PolySpace approach. A “PolySpace approach” or “Approach” is defined in this context as the manner of use of PolySpace verification to achieve a particular goal, with reference to a collection of techniques and guiding principles. These include suggestions of different activities which might be completed before functional unit test or integration tests, depending on the development process:

- An explanation of the collection of techniques and guiding principles going to form each Approach.
- Fixing red and grey — review run time errors and dead code checks only
- Selective orange review — review warnings and find bugs quickly and efficiently. Suitable when time is short, and the aim is to maximize the number of bugs discovered.
- Exhaustive orange review — how much it costs and the value it brings at the unit phase and at the integration phase
- Shared data conflict detection — and the problems it can highlight

- Data flow analysis
- Integration bugs tracking

An explanation of the steps required to progress seamlessly from one approach to the next:

- Coding rules to allow an efficient exhaustive warning review.
- Data rules to allow efficient integration bug tracking

PolySpace™ Usage

In this section...
“Overview of the PolySpace™ Approach” on page 9-5
“Standard Development Process” on page 9-10
“Rigorous Development Process: Introducing Tools and Coding Rules” on page 9-14
“A Quality/Qualification Approach” on page 9-17
“Code Acceptance Criterion” on page 9-18

Overview of the PolySpace™ Approach

PolySpace™ tools can support two main objectives concurrently.

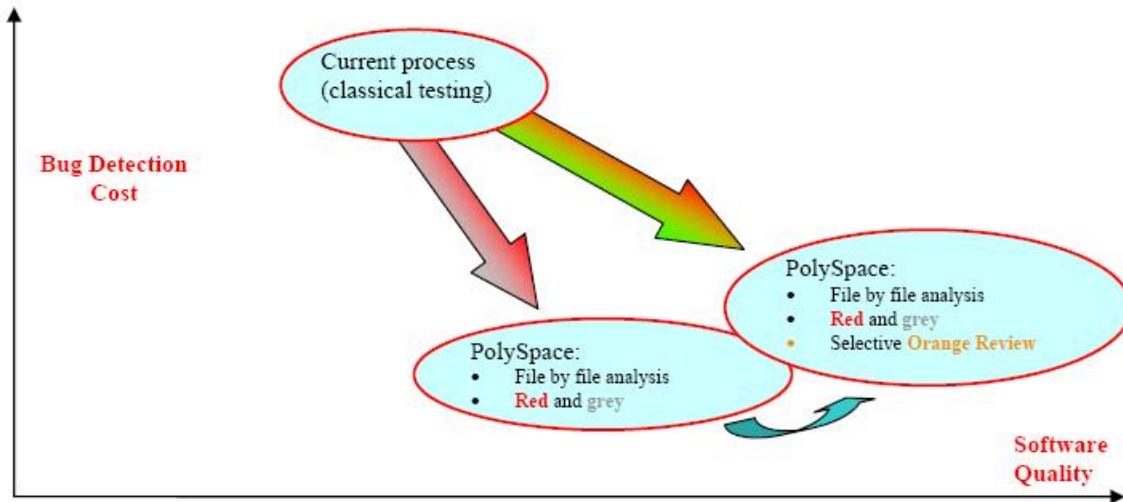
- Reduction of testing and validations costs
- Improvement of the software quality

PolySpace tools can be used in different ways depending on the context, the primary difference being in the approach used to exploit the results generated. The following diagrams summarize the different approaches.

Note The aim here is not to compare the cost of certification processes, or of development processes with or without coding rules. The graphs aim to compare the costs of typical processes with and without PolySpace software.

When No Coding Rules Are Adopted

During the coding activity, there are two recommended approaches:



The first approach is to use only the red and **grey** results: fix the red bugs, and check the dead code for abnormalities.

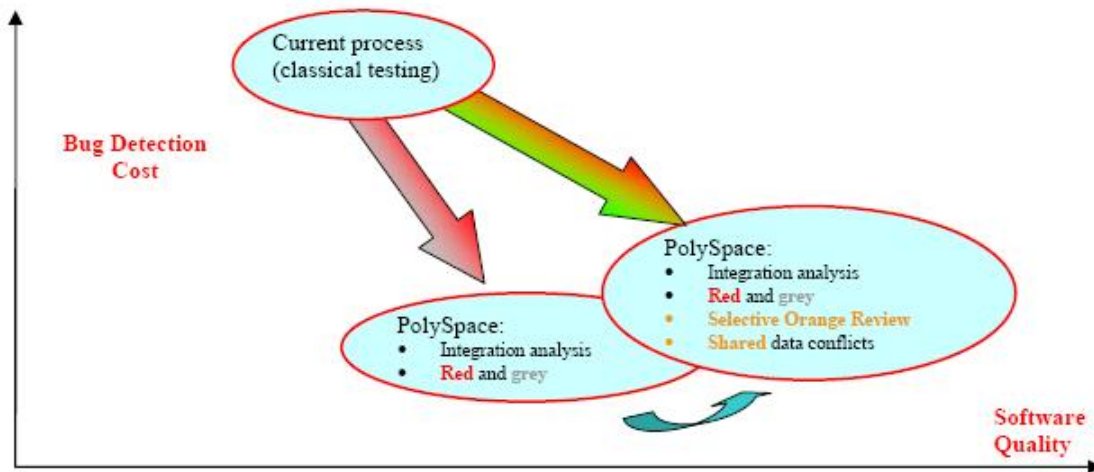
The second approach involves the same activities, and adds a partial review of the orange warnings. The aim is to find as many bugs as possible, with very limited efforts. This approach finds more bugs and therefore improves the quality. It does involve more effort, but the amount of time spent to find each bug remains very small.

Note Using PolySpace verification on one single file is efficient: even though there is no knowledge of the file context, experience shows that 50% of the bugs detected by PolySpace verification can be found locally.



This symbol is used to indicate that when level of usage of PolySpace verification has been successfully implemented the development team can migrate to a more demanding (and more fruitful) one. This migration is not always desirable; it of course depends on the projects context.

Then, after coding, before the testing activity:



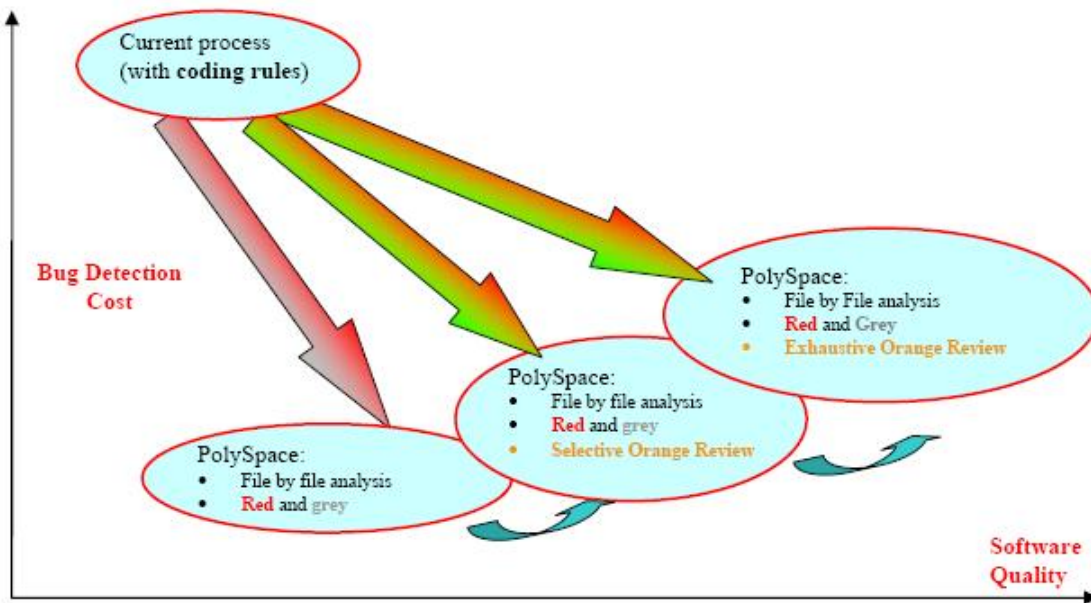
Again, the first approach is to use only the red and grey results: fix the red bugs, and check the dead code.

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

When Coding Rules Have Been Adopted

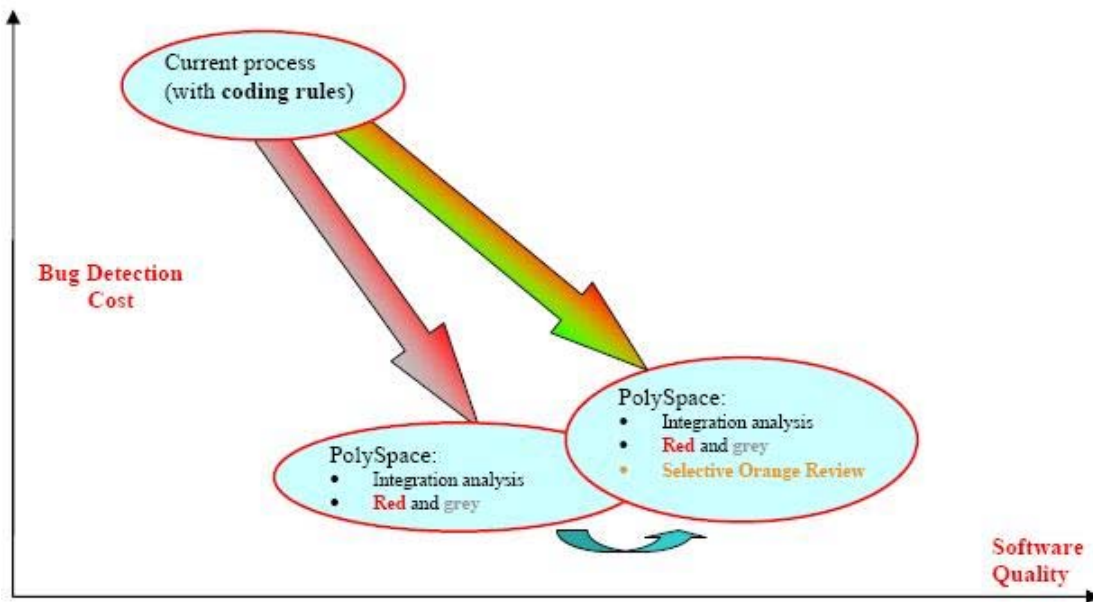
The main difference here by comparison with the previous processes is with respect to the cost of bug detection. When PolySpace verification is used in accordance with a set of coding rules, the bug detection cost is much lower.

There are three recommended ways to use PolySpace verification, **during the coding activity**:



Compared to the previous situation (where no coding rules are in place), an additional possibility exists. Instead of reviewing only certain orange warnings in a file, all of them are systematically checked. This is possible as when the **right coding rules** are respected (see the end of this section for recommendations). That leads to there being only a few orange checks in a file, and therefore checking all of them is potentially 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 analysis, 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 analysis, 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” on page 9-10
- “The Software Development Process” on page 9-10
- “The Objective of Using PolySpace™ Verification” on page 9-11
- “The PolySpace™ Approach” on page 9-11
- “A Complementary Approach” on page 9-12
- “Integration with Configuration Management Tools” on page 9-12
- “Costs and Benefits” on page 9-13

Overview

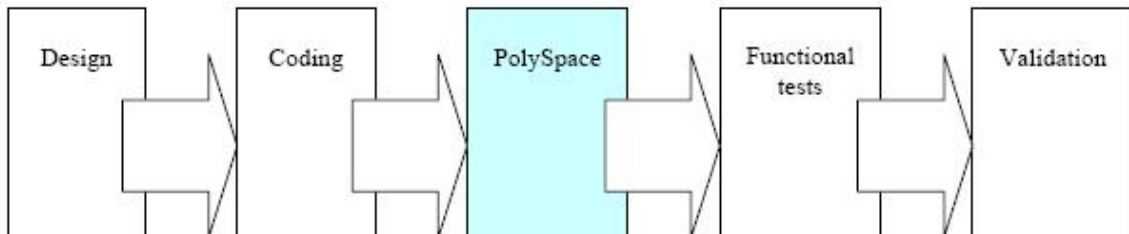
This approach is mainly for consideration by a project manager rather than a quality manager. It aims to improve productivity rather than to prove the quality of the application being analyzed.

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.



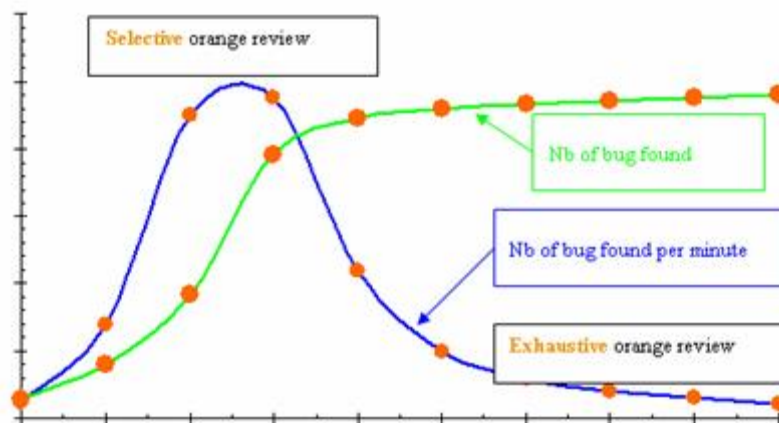
The Objective of Using PolySpace™ Verification

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 timeframe 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 Desktop to be applied by developers or testers on a file-by-file/package-by-package analysis basis. The users will use the **default PolySpace Desktop 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 **grey** 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 Run Time 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 analysis will be performed by PolySpace software on larger modules, and the orange review will be focused on orange Run Time errors **which were not examined** after the file-by-file/package-by-package analysis.

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

- The user will ignore orange overflows with PolySpace Desktop when performing file-by-file analysis,
- He will examine them with PolySpace Verifier.

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 Desktop is applied to the file(s) currently under development. Compilation must complete without the permissive option.
- **Pre-unit test check-in** — PolySpace Desktop is applied to the file(s) currently under development.
- **Pre-integration test check-in** — PolySpace Verifier 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 Verifier is applied to the whole project, with all multi-tasking aspects accounted for as appropriate.
- **Pre-peer review check-in** — PolySpace Verifier is applied to the whole project, with all multi-tasking 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 Desktop 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 grey checks. The number of bugs found thanks to these colors can vary from one user to another, but experience shows that on average, around of the analyses will reveal a red error(s) and/or will reveal bugs in grey 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 analysis 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 contain more oranges when analyzed with PolySpace Verifier than

with PolySpace Desktop (about twice as many). These oranges are likely to be anomalies, and will be responsible for the orange check review becoming more time consuming.

- **A more stable software version implies a later analysis** — If PolySpace Verifier is used **instead of** PolySpace Desktop, bugs might be revealed much later because a more *complete* version of the software can only be provided at a later phase in the project.
- **An exhaustive orange review can take 25 men-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” on page 9-14
- “The Software Development Process” on page 9-14
- “The PolySpace™ Approach” on page 9-15
- “A Complementary Approach” on page 9-16
- “Costs and Benefits” on page 9-16

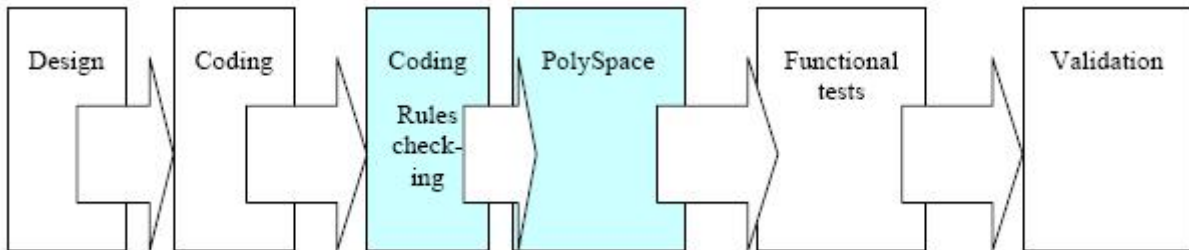
Overview

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 process which has the following characteristics. In Ada, unit testing tools or coverage tools are used.

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 Desktop on a file by file analysis basis.

- The “main” used to analyze each file is very often **automatically generated by the project**, and not by PolySpace Desktop (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 analysis.
- **[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 Desktop can flag instances where that range of values might be breached.
- Red errors will be fixed and grey 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 Verifier at integration phase to track integration bugs, and review:-

- Red and grey integration checks;
- Orange checks on code which produced green checks when analyzed by Desktop.
- The remaining orange checks with a selective review: *Integration bug tracking*.

Costs and Benefits

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

- 3-5 orange checks per file, 3 grey 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 analyses by PolySpace Desktop per file is typical before the file can be checked-in to the configuration management system.
- The average analysis 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 Verifier 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” on page 9-17
- “The Software Development Process” on page 9-17
- “The Objective of Using PolySpace™ Verification” on page 9-17
- “The PolySpace™ Approach” on page 9-18
- “Costs and Benefits” on page 9-18

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 analysis
- Shared data detection
- Robustness unit tests.

The PolySpace™ Approach

Depending on the activity replaced, both PolySpace Verifier and/or Desktop may be useful.

- For data and control flow analysis and shared data detection. PolySpace Verifier can be used on the whole application or on a sub-section of the application.
- For robustness unit tests (as opposed to functional unit tests). PolySpace Desktop might be used in the same way as the one 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 analysis 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” on page 9-18
- “The Software Development Process” on page 9-18
- “The Objective of Using PolySpace™ Verification” on page 9-19
- “The PolySpace™ Approach” on page 9-19

Overview

This is likely to be of interest for a quality manager in a company which is out-sourcing 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

Either PolySpace Desktop or Verifier can be used at this stage, depending on the project size. 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 grey 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

PolySpace™ Activities

In this section...

“Review Run Time Errors: Fix Red Errors” on page 9-20

“Review Dead Code Checks: Why is Grey Code Interesting” on page 9-21

“How to Find a Maximum Number of Bugs Within an Hour Reviewing Oranges: Selective Orange Review” on page 9-23

“Cost and Benefits of an Exhaustive Orange Review at Integration Phase” on page 9-27

“Integration Bug Tracking” on page 9-29

“How to Find Bugs in Unprotected Shared Data” on page 9-30

“Dataflow Analysis” on page 9-31

“Data and Coding Rules” on page 9-31

Review Run Time Errors: Fix Red Errors

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

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

This result is 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 irrespective of their consequence, or of the ease with which they can be corrected.

Review Dead Code Checks: Why is Grey Code Interesting

- “Functional Bugs Can Be Found in Grey Code” on page 9-21
- “Structural Coverage” on page 9-22

Functional Bugs Can Be Found in Grey 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 Verifier 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 grey 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 run time 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.

How to Find a Maximum Number of Bugs Within an Hour Reviewing Oranges: Selective Orange Review

- “Overview” on page 9-23
- “How” on page 9-24
- “Why” on page 9-24
- “In Practice” on page 9-24
- “Step by Step” on page 9-24
- “Which Category of Checks Should I Choose First” on page 9-25
- “Exhaustive Orange Review at Unit Phase” on page 9-26

Overview

Note Before reading this section, it is necessary to understand how the user might conclude the status of an orange check. This is explained in a later section.

Suppose, for example, that the user wishes to spend the first hour of the day reviewing an analysis which was performed overnight. This is an approach which can be adopted to enhance the quality of code under development, perhaps supported by more extensive analysis as the project nears completion.

Experience suggests that such an approach can highlight 5 bugs in orange checks in such a timescale: “finding 5 bugs an hour”

How

Focus on modules which have the highest selectivity in the application, where selectivity is the ratio of (green + grey + red) / (total number of checks)

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

Why

If PolySpace verification finds only one or two orange checks in a module or function, there is a very good possibility that they are not caused by “basic imprecision”. Consequently, the concentration of bugs in orange checks here will be higher than in those found elsewhere in the code.

If you come across an orange check which takes more than a few minutes to understand, it might well be the result of inconclusive PolySpace analysis. To optimize the number of bugs found in a limited time, you should move on to another check. A good rule of thumb is to spend no more than 5 minutes on each check, remembering that the goal is to review at least 50 checks per hour to maximize the number of bugs found.


In Practice

For any particular function, PolySpace verification may be better at detecting some kinds of Run Time Errors than others. For instance, the analysis of one function may yield imprecise results from the analysis of Non Initialized Variables (NIV) but very precise results from the analysis of overflows (OVFL). In the analysis of another function, the precise opposite may be true.

So, the “high selectivity focus” should be applied to each Run Time Error category **separately**.

Step by Step

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

2 Click on 

3 Click on the check type of interest (ZDV in the example).



4 Choose files/packages containing only 1 or 2 orange checks of the selected kind.

5 Proceed with a quick code review on each orange check, spending no more than 5 minutes on each. The goal is to identify the orange check as a *potential bug*, *inconclusive check* or *data set issue*, navigating the code using the call tree and the dictionary. If the check proves too complicated to explain, it may well be the result of *basic imprecision*.

6 Once this job done, the user can select the “Verified” checkbox in the PolySpace Viewer, and put an explanation of the check in the comment field (for instance, “inconclusive”, or “data set issue” when calibration of <x> is set greater than 100”,)

7 Select another type of RTE and repeat the procedure.

Which Category of Checks Should I Choose First

The following sequence is recommended.

- 1 Start with the four categories found to be the most likely to yield bugs, which are described in the following sections.
- 2 Next, use the Beta filter which will highlight the remaining categories most likely to include any remaining critical Run Time Errors.
- 3 Finally, complete the remaining checks as time permits.

The impact made by the use of C coding rules is huge, because they reduce complexity - a key factor in limiting **orange checks** due to basic imprecision. The C constructions impacting each of the four are listed below.

- Potential bug or data set issue. These are **orange checks** representing genuine problems.
- Inconclusive check. These are **orange checks** which mostly highlight design issues, not addressed by this section.
- Basic imprecision.
 - Unspecified Standard behavior
 - Complexity
 - Approximations made by the tool on specific constructions

MISRA® rules have a huge impact on complexity and all unspecified ANSI® behavior. Some details of approximations made by the tool are discussed following the section discussing MISRA rules.

Exhaustive Orange Review at Unit Phase

- “Without Coding Rules” on page 9-26
- “With Coding Rules” on page 9-27

Without Coding Rules. An exhaustive orange review progresses at a typical rate of **50 orange checks per hour**. An hour spent on an exhaustive check review is different to an hour spent on a selective orange review in several significant ways.

Time:

- 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.

Cost:

- 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 **orange check** would be about 1 minute.

With Coding Rules. The number of spurious **orange checks** per file strongly depends on coding styles within the project. The following coding rules are recommended, as are a subset of MISRA rules.

If the code follows the recommended MISRA subset, the count of checks per file will typically decrease to 3 **orange** and 3 **grey** checks, hiding at least one bug between them.

The review of the PolySpace results generated by a unit analysis would normally take no more than 15 minutes.

Cost and Benefits of an Exhaustive Orange Review at Integration Phase

- “Benefits” on page 9-27
- “Costs” on page 9-27
- “Method” on page 9-28

Benefits

The purpose of this activity is to assess the probability of missing an orange containing a bug when performing a “selective orange review”. This needs to be balanced with the cost of a bug left in the code.

Costs

Experience suggests that an average of **4-5 minutes reviewing time per orange check** is typical. Four hundred (400) such checks will require 4 days of code review whereas a three thousands (3000) orange review will require 25 days.

If the checks are reviewed in the sequence suggested by the selective review approach, then the first 80% of these checks will take a disproportionately small amount of time.

Method

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

Consider the three possible reasons for an orange check:

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

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

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

Consider an example below. Suppose that

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

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

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

In this case, PolySpace 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:

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

In this second case, an exhaustive orange review will already have been performed package by package at a unit level. 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 Verifier launching parameters and reanalyze.

It is also worth checking whether variables are modified which are supposed to be constant. Use the variables dictionary.

Dataflow Analysis

Data flow analysis 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 analysis 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.

Automatically Testing Orange Code

In this section...
“PolySpace™ Automatic Orange Tester” on page 9-33
“Using the Automatic Orange Tester” on page 9-35
“Technical Limitations” on page 9-52

PolySpace™ Automatic Orange Tester

The PolySpace™ Automatic Orange Tester dynamically analyzes 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.

The screenshot displays the PolySpace Automatic Orange Tester interface. At the top, a tree view shows the test campaign structure under 'External Scope', including functions like 'random_float', 'random_int', 'get_bus_status', and 'read_bus_status', each with a 'return' variable. Below this, the 'Test Campaign Configuration' section allows setting the number of tests (1000), iterations for infinite loops (100), and per test timeout (10 seconds). The 'Test Campaign Results' section shows 1000 completed tests, 71 no errors detected, and 929 total failures, with 10/929 checks/tests having errors. A progress bar indicates 100% completion. At the bottom, a table lists the results of failed test cases.

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:	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:	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

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 Appendix A, “Static Verification”). 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?” on page 9-65).

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.

Using the Automatic Orange Tester

This section describes how to use the Automatic Orange Tester. It describes:

- “Before Using the Automatic Orange Tester” on page 9-36
- “Launching the Automatic Orange Tester” on page 9-37
- “Reviewing the Test Results” on page 9-41
- “Refining Data Ranges” on page 9-45
- “Saving and Reusing Your Configuration” on page 9-50
- “Exporting Data Ranges for PolySpace™ Verification” on page 9-50
- “Configuring Compiler Options” on page 9-51

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.

Search internal name from the selected line : <input type="text"/>			
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.

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.

The screenshot displays the PolySpace Viewer interface with the following components:

- Top Panel:** Contains a menu bar (File, Edit, Windows, Help), a toolbar with icons for file operations and analysis, and a status bar with a tooltip that reads "Launch the PolySpace Automatic Oranger Tester."
- Coding review progress table:**

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
- Code Editor:** Shows a warning message: "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]". The code snippet includes:

```


example.c / Close_To_Zero / line 43 / column 12
if ((xmax - xmin) < 1.0E-37f)

```
- Procedural entities tree:** A hierarchical tree view showing entities like Demo_C, example.c, and Close_To_Zero(). Under Close_To_Zero(), various NIVL (Non-Infinite Loop) and UOVFL (Underflow/Overflow) entities are listed with their respective counts. UOVFL.6 is highlighted in orange.
- Variables View:** A panel showing the scope of variables, listing "Written by", "Read by", and "Potentially Written by" for the current context.
- Call Tree View:** A panel showing the call graph, with options to filter by "Both", "Called by", "Calls", "Complete", and "Update on selection".
- Source Code Editor:** Displays the source code for example.c, showing the Close_To_Zero function definition:

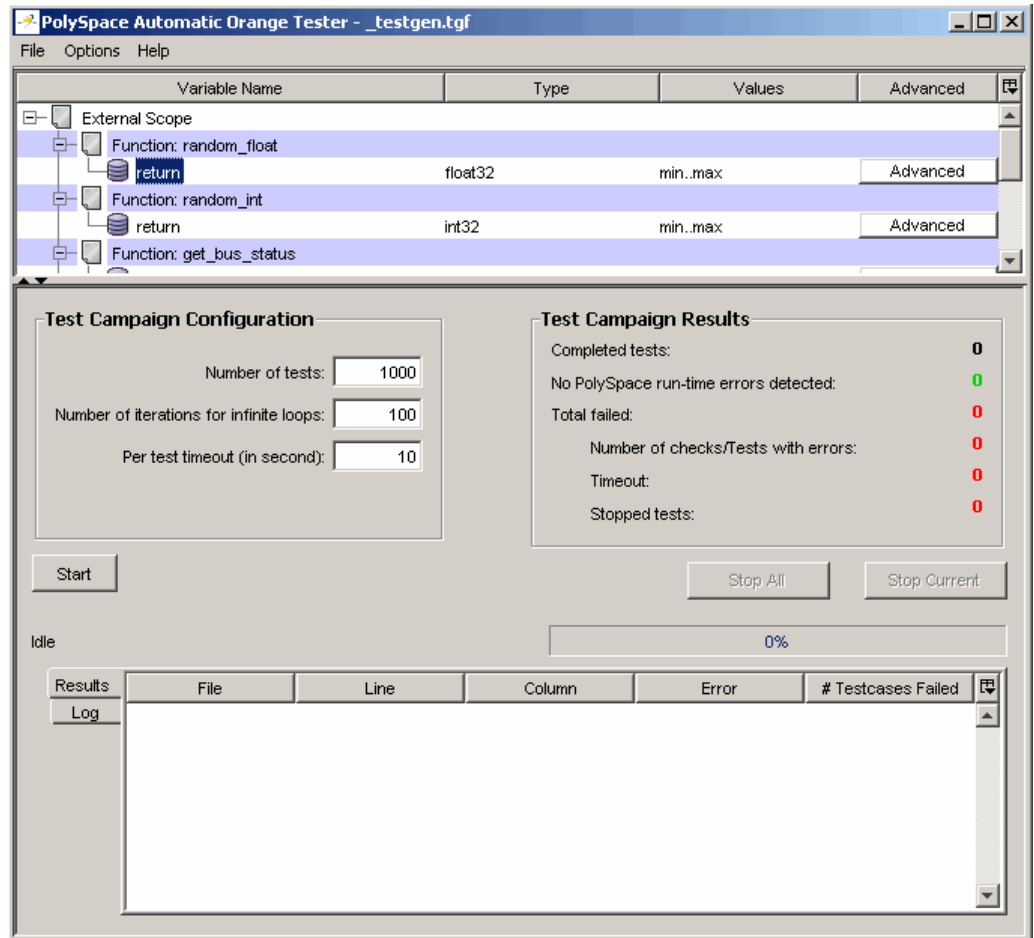
```

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

```
- Bottom Panel:** Shows the current file path: 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 analysis time.

4 Click **Start** to begin testing.

The Automatic Orange Tester generates test cases and runs the dynamic tests.

The screenshot shows the PolySpace Automatic Orange Tester interface. The top section displays 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:

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

Buttons: Stop, Stop All, Stop Current.

Running... Time Remaining: 0:0:6 61%

Results Table:

File	Line	Column	Error	# Testcases Failed
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.

File	Line	Column	Error	# Testcases Failed
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 run-time 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.

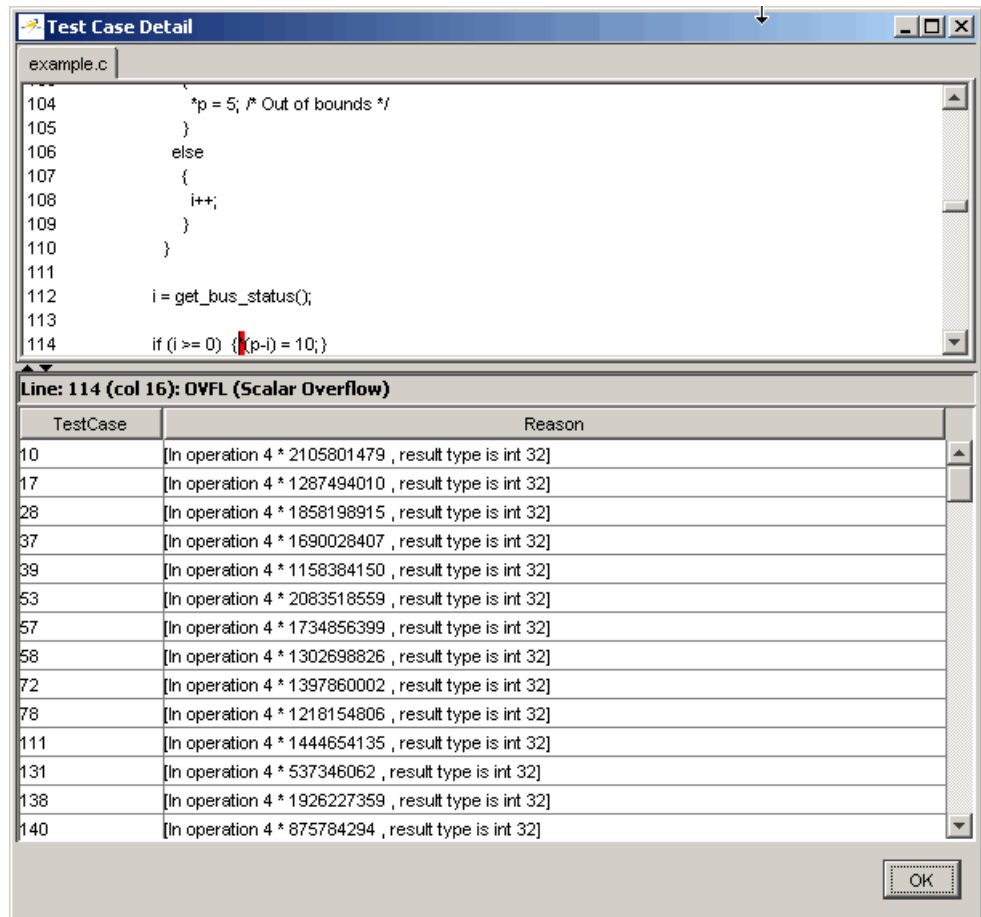
- **Stopped tests** – The number of tests that were stopped manually.

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.

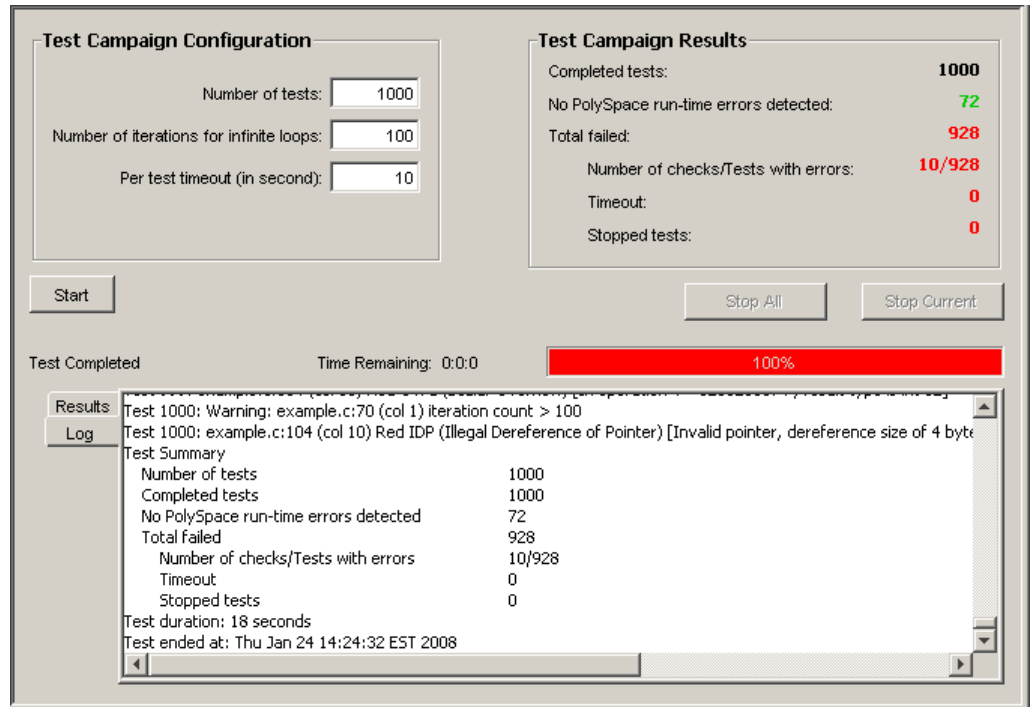


The Test Case Detail dialog box displays the portion of the code in which the error occurred, and gives detailed information about why each test case failed. Since the Automatic Orange Tester performs runtime tests, this information includes the actual values that caused the error.

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

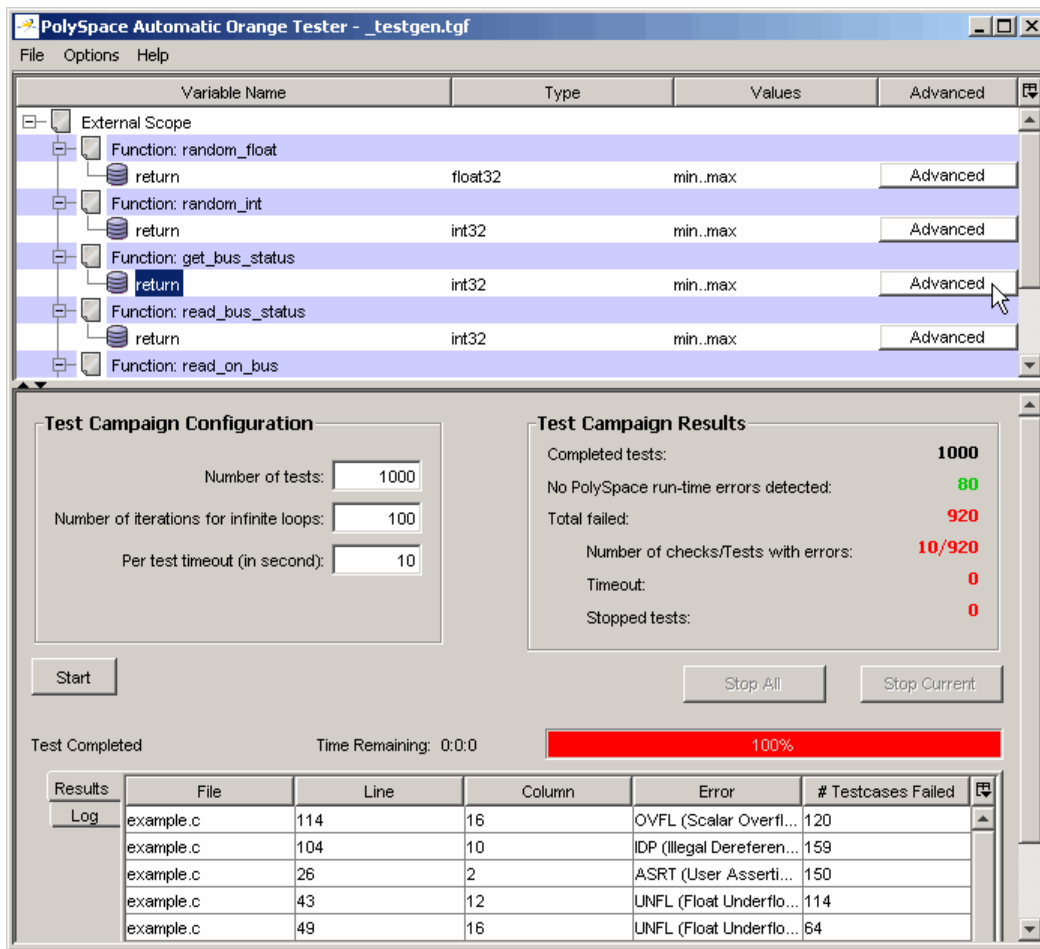
Refining Data Ranges

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.



2 Select **Advanced**. The Edit Values dialog box opens.

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 re-test 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			
return	float32	0..10000000	Advanced
Function: random_int			
return	int32	min..0	Advanced
Function: get_bus_status			
return	int32	-100..0	Advanced
Function: read_bus_status			
return	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 analysis:

- 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 Analysis, see Chapter 6, “Data Range Specifications”.

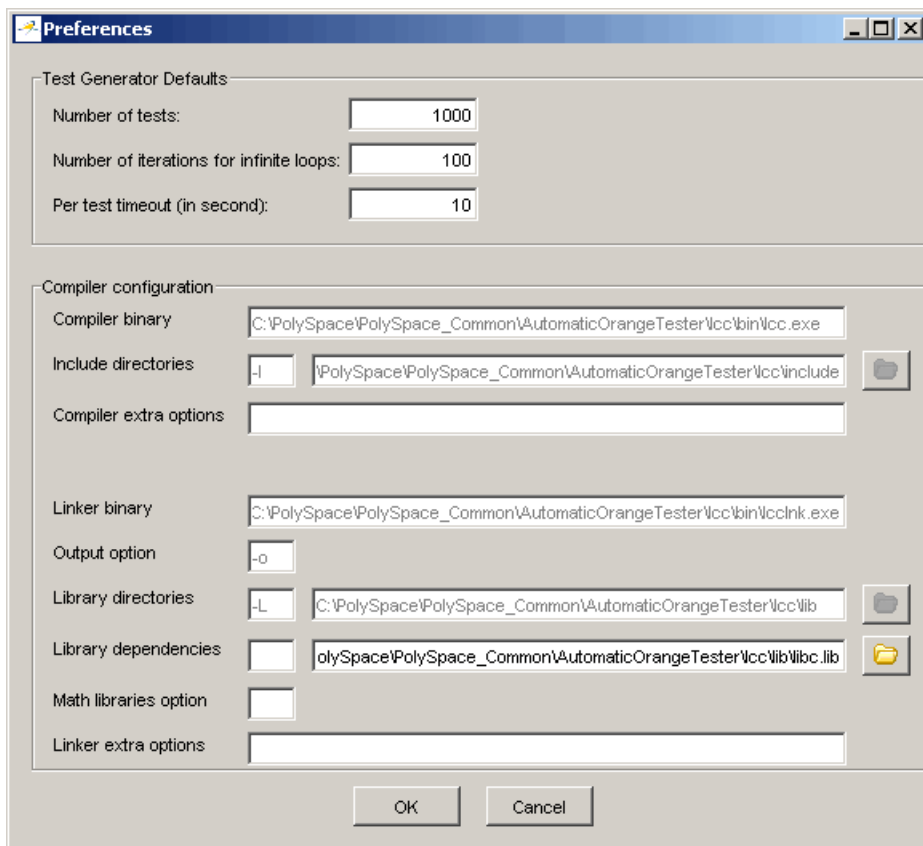
Configuring Compiler Options

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 9-53
- “Options with Limitations” on page 9-53
- “Unsupported C Language Constructions” on page 9-54

Unsupported PolySpace™ Options

The following options are not supported when you select `-prepare-automatic-tests`.

- `-entry-points`
- `-dialect`
- `-ignore-float-rounding`
- `-div-round-down`
- `-entry-points`
- `-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`.

- `-align [8|16]`
- `-target [c-167 | tms320c3c | hc08 | sharc21x61]`
- `-data-range-specification` (in global assert mode)

The endianness of a target is not managed. The analysis is performed as if the user-defined target has the same endianness as the operating system.

In addition, when using the Automatic Orange Tester, the `-target mcpu` option must be used together with `-pointer-is-32bits`.

Unsupported C Language Constructions

The code verification stops when any of the following characteristics are met:

- Referenced absolute address is not supported
- 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

How to Get the Best Results

In this section...

“Reduce Oranges Step by Step” on page 9-56

“Generic Objectives: A Balance Between Precision and Analysis Time” on page 9-56

“Options at Launching Time” on page 9-58

“How to Conclude an Orange Review” on page 9-64

“Duration of Analysis” on page 9-68

Reduce Oranges Step by Step

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 some recommended 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 analyses 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 Analysis 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 an analysis takes an eternity and the results contain the maximum possible number of grey, red and green checks, this analysis is not useful because of the time spent waiting for the results.
- If an analysis is very quick but contains only oranges, the analysis won't be very useful because of the large number of manual checks to be performed.

Using PolySpace verification is a compromise between analysis 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 an analysis. 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 an analysis and will spend **the next morning** analyzing the results.
- Validation/acceptance testing: the developer leaves the office on Friday evening and starts an analysis. The developer will spend the following **week** analyzing the results.

Note So analysis 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 an analysis never ends. The user might need to split his application.

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

Most of the time, the first analysis should be in “-quick” mode.

Note All activities and methods relating to results analysis remain unchanged irrespective of the precision selected (-O0, -O1,-O2 or -O3 in Ada and C, and -quick in C).

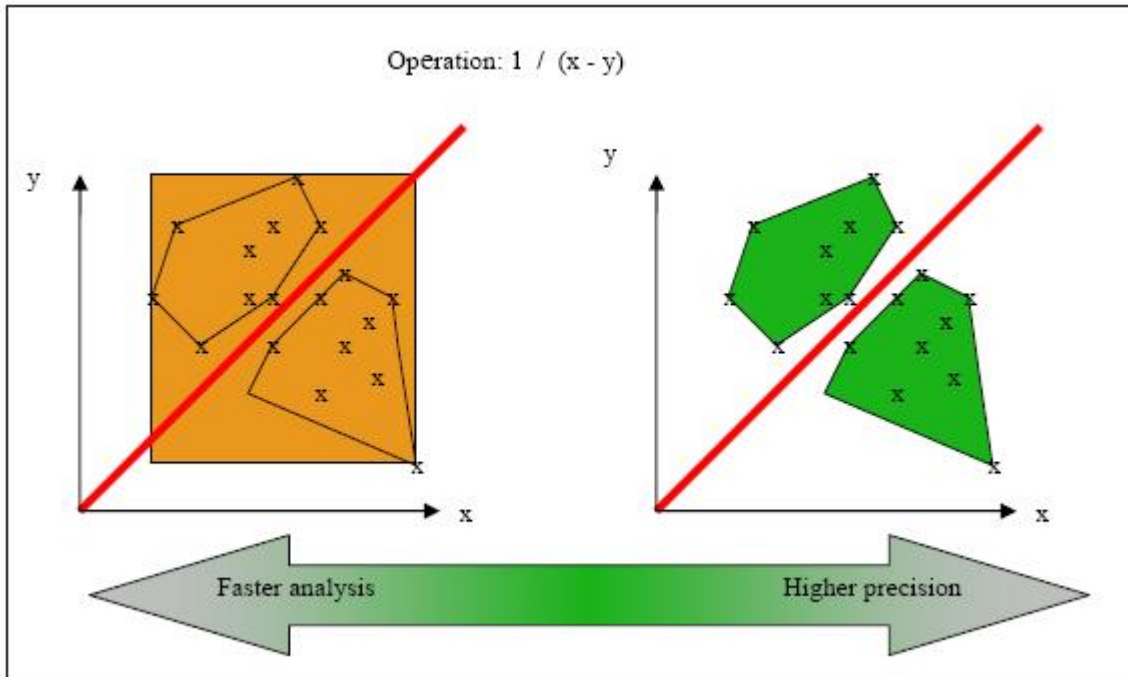
Options at Launching Time

- “Vary the Precision Level” on page 9-58
- “Apply Software Safety Level Wisely” on page 9-59
- “Add Precision Constraints at the Periphery Via Stubs” on page 9-61
- “Describe Multitasking Behavior Properly” on page 9-63
- “Tuning Advanced Parameters” on page 9-64

Vary 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 -quick (only in C or C++ language), -O0, -O1, -O2 or -O3 precision level. You can also select a particular precision for a specific body (in Ada) or a C file (in C).

The methods used by Verifier 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 analysis will become green when analyzed at a higher precision.



Vary the Precision Rate

Apply Software Safety Level Wisely

Abstract. What are the differences between analysis levels

Explanation. There follows an example of the distinction between Safety Analysis levels 1, 2 and 3. The deeper the analysis 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 an analysis is the depth of analysis of PolySpace Verification.** It starts with Safety Analysis 1 (which approximates to unit analysis) 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 analysis goes, the more precise it is. Depending on the backward/forward dependencies, oranges will be resolved into red, green or grey at the Safety Analysis level 1 or later in level 2, 3 or 4.

The level of an analysis 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 analysis 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 analysis 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>

Add Precision Constraints at the Periphery Via Stubs

Another mean to increase the selectivity is to indicate PolySpace Desktop that some variables (detailed here after) might vary between some functional ranges instead of the full range of the considered type.

It concerns mainly two items from the language

- Parameters passed to functions.

- Variables content, mostly globals, which might change from one execution to another: typically, calibration data, 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 Verifier models data ranges throughout the code it analyses, 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); return</pre>	<pre>#include <assert.h> extern int other_func(void); int stub(void) { int tmp; tmp= other_func(); assert(tmp>=1 && tmp<=10); return }</pre>	<pre>extern int other_func(void); int stub(void) { int tmp; do {tmp= other_func();} while (tmp<1 tmp>10); return tmp; }</pre>

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

```
int a,b;
int *ptr;
void a_missing_function(int *dest, int src);
/* should copy src into dest */
void main(void)
{
    a = 1;
    b = 0;
    a_missing_function(&a, b);
    b = 1 / a;
}
```

- By relying on the PolySpace default stub, the division is shown with an orange warning because a is assumed to be anywhere in the full permissible integer range (including 0)
- If the function was commented out, then the division would be green.
- A red division could only be achieved with a manual stub.

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

Describe 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 PolySpace Verifier.

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.

You can refer to “*My Code is Multitasking*” on page 3-69 for a complete description of tasking facilities. These include

- Shared variable protection:
 - Critical sections,
 - Mutual exclusion,
 - Access pattern,
 - Tasks synchronization,
 - Rendez-vous (for Ada only),
- 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 analysis 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*” on page 10-47.

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

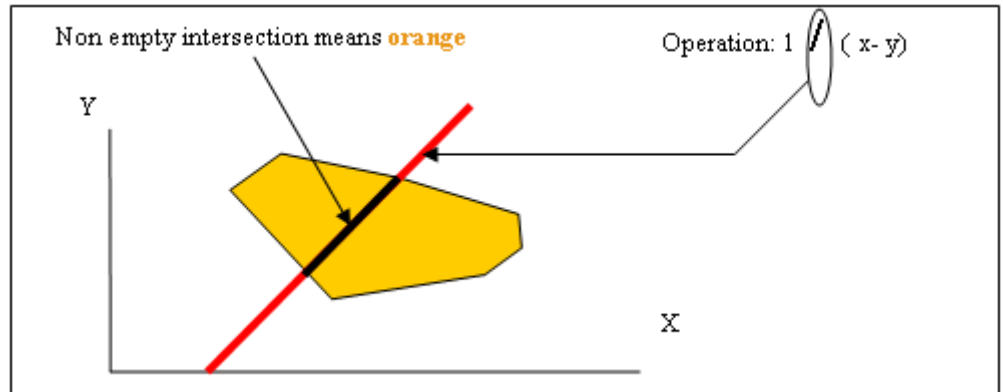
How to Conclude an Orange Review

- “What is an Orange?” on page 9-65
- “What are the Different Sources of Oranges?” on page 9-66

- “How to Determine the Cause of an Orange?” on page 9-67

What is an Orange?

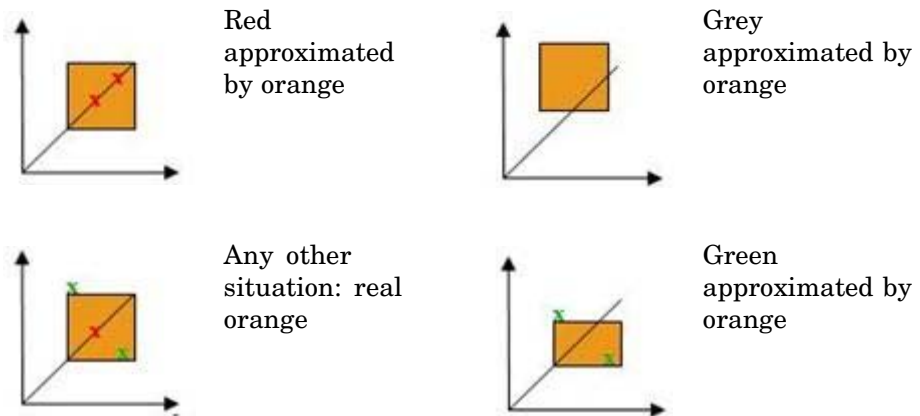
If a check is orange, it means that the approximate data set assumed by the analysis to represent a variable intersects with the error zone.



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 analysis 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.)

What are the Different Sources of Oranges?

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 X, a signed integer between -2^{31} and $2^{31}-1$.
 - Suppose a function is called which performs the assignment $x=1/x$
 - The parameters passed to the function imply that x must be equal to -5, -3, 8 or [10..20]. It is clear from inspection that there is no problem here, but in this case PolySpace verification has made an imprecise approximation.

How to Determine the Cause of an Orange?

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 analysis** — 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.

Duration of Analysis

- “How Far has the Analysis Gone? How Can I Predict the Analysis Duration?” on page 9-68
- “Reducing Analysis Time” on page 9-70

How Far has the Analysis Gone? How Can I Predict the Analysis Duration?

The duration of an analysis 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 analysis 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_New_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_Project_03_25_2004-11h50.log for further information.

$

```

Consider the area displaying:

```
Currently in Level 1 Software Safety Analysis
```

4 .atz files out of a total 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 analysis duration can be deduced by extrapolating from this data by considering the number of files and passes still to be completed.

Reducing Analysis Time

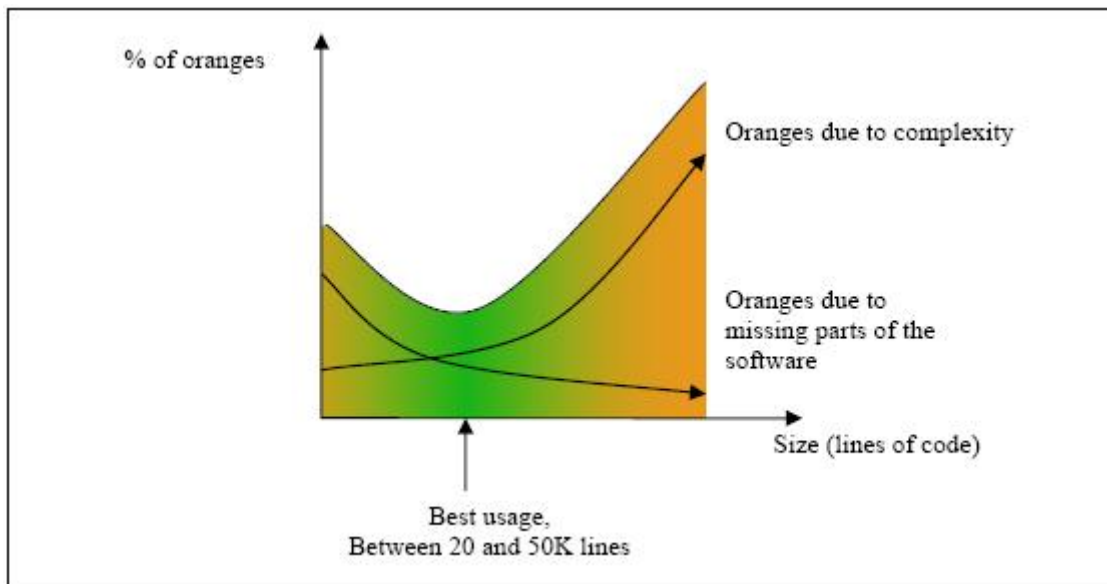
- “An Ideal Application Size” on page 9-70
- “Why Should there be an Optimum Size?” on page 9-71
- “Switch the Anti-virus Off” on page 9-72
- “Tuning PolySpace™ Parameters” on page 9-72
- “By Selecting a Subset of Code” on page 9-73
- “A Decision Algorithm to Speed-Up an Analysis: Hints and Troubleshooting” on page 9-78
- “What are the Benefits of these Methods?” on page 9-84

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 analysis further and to solve traditionally incomputable problems. However, they also mean that the benefits derived from analyzing 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 analyses (when dealing with C language), package by package analyses (when dealing with Ada language) and class by class analyses (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 analysis time should **not be reasonable**.

Experience suggests that subdividing an application prior to analysis will normally have a **beneficial impact on selectivity** — that is, more red, green and grey 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 Verifier

- 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, whatever) is analyzed using PolySpace software, the focus should be on the red and grey 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, grey 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 analysis 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 Anti-virus Off. Disabling or switching off any third party anti-virus software for the duration of an analysis can reduce the analysis time by up to forty percent.

Tuning PolySpace Parameters. here is a compromise to be made to balance the time required to perform an analysis, and the time required to review the results. Launching PolySpace verification with the following options will allow the time taken for analysis 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 analysis 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, appropriate launching commands might be

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

or

```
polyspace-c -quick
```

By Selecting a Subset of Code. If a project is subdivided for analysis purposes, then the total analysis time will be considerably shorter for the sum of the parts than for the whole project considered in one pass. (See also: “Understanding Addressing” on page 8-35 , “Checking properties on global variables at any point: Global assert” on page 3-64, and “Stubbing” on page 3-48). 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 3-48).

```
//removed definition of complete_treatment_based_on_x
void main(void)
{
  x = ... // what ever
  y = complete_treatment_based_on_x(x); // now stubbed!
}
```

Some Consequences

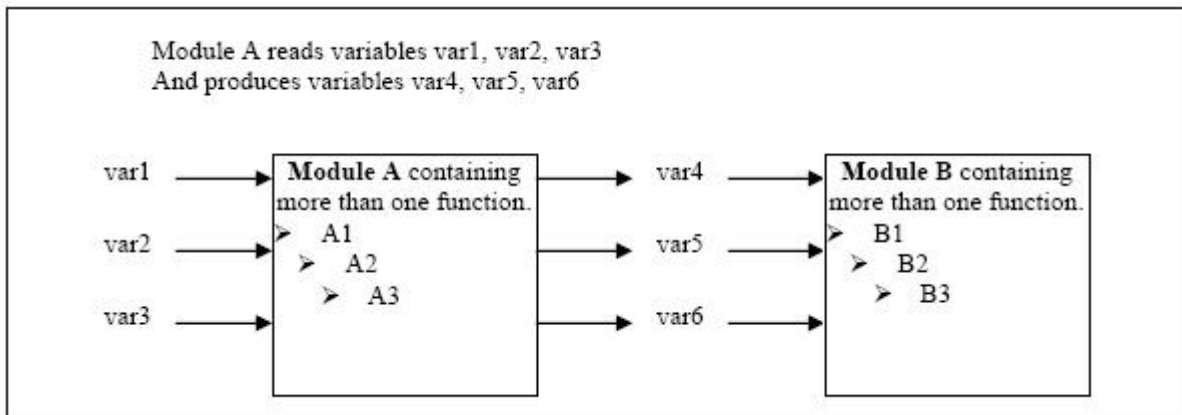
- (-) Some loss of precision on y. Verifier will now consider all possible values for y, including those specified for the first analysis;
- (+) A huge investigation of the code is not necessary to isolate a meaningful subset. Any application can be split logically in this way;
- (+) No functional modules 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 analysis and the arrays are initialized with random values instead, then the analysis 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 $var4 > var5 > 5$. • If var3 is greater than 4, than $var4 < var5 < 12$ • ...

Subdivision in accordance with data flow allows modules A and B to be 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 analysis, 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 analysis.

For example, If the B module included the test

“If var2 is equal to 0, than var4>var5>5”

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

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

Typical examples of removable components:

- Error management modules. A function `has_an_error_already_occurred` might return TRUE or FALSE. Such a module may contain a big array of structures which are accessed through an API. The removal of the API code with the retention of the prototype will result in the Verifier analysis 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 Verifier analysis, 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 an analysis is initiated using only a few tasks, PolySpace Verifier 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 analysis 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 analysis 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 an analysis 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.

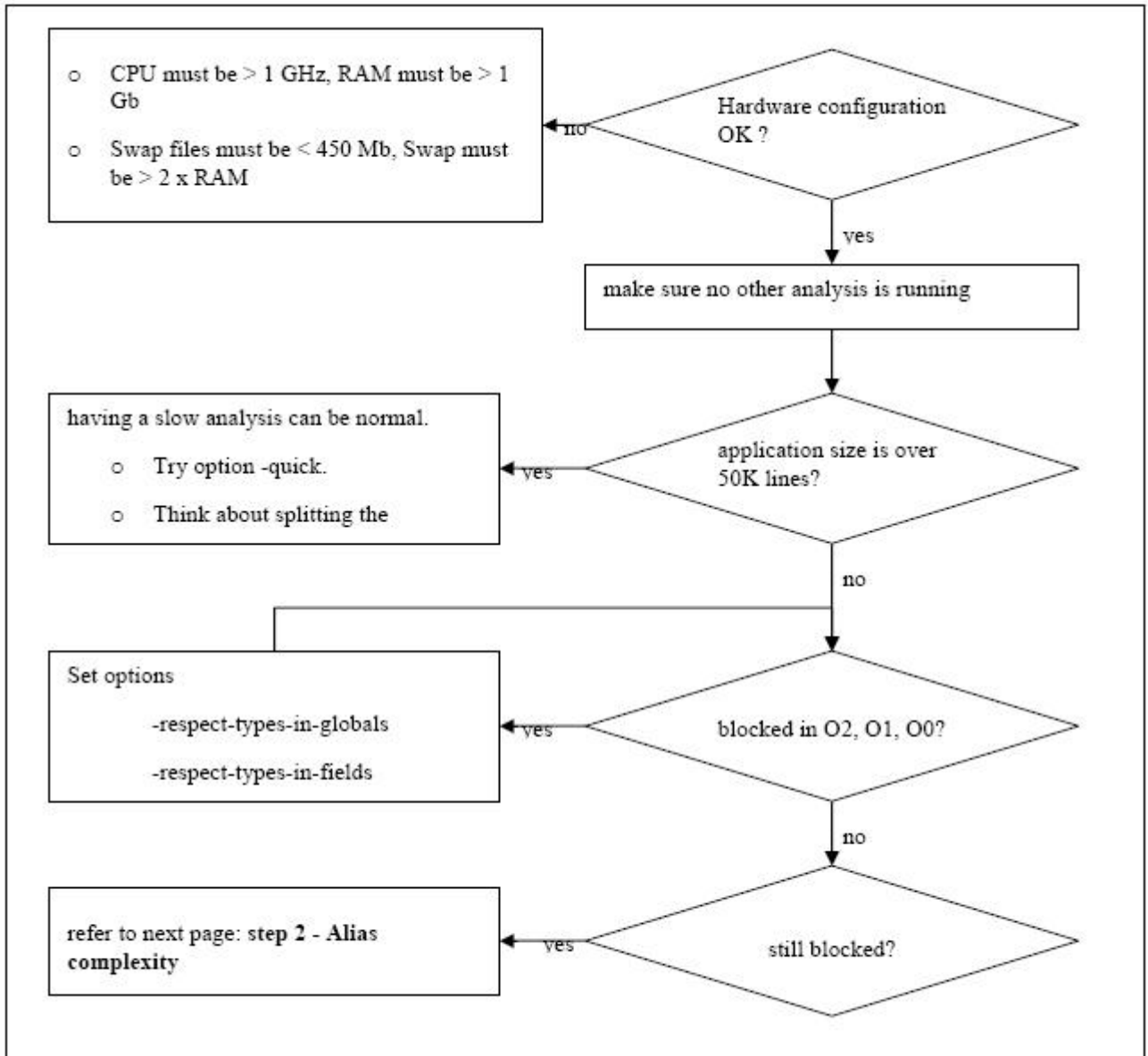
This method may look too simple to be efficient but it can produce good results when the aim is to find red errors and bugs in grey code.

A Decision Algorithm to Speed-Up an Analysis: Hints and Troubleshooting. This chapter suggests methods to reduce the duration of a particular analysis, while minimizing the need to compromise the launch parameters or the precision of the results.

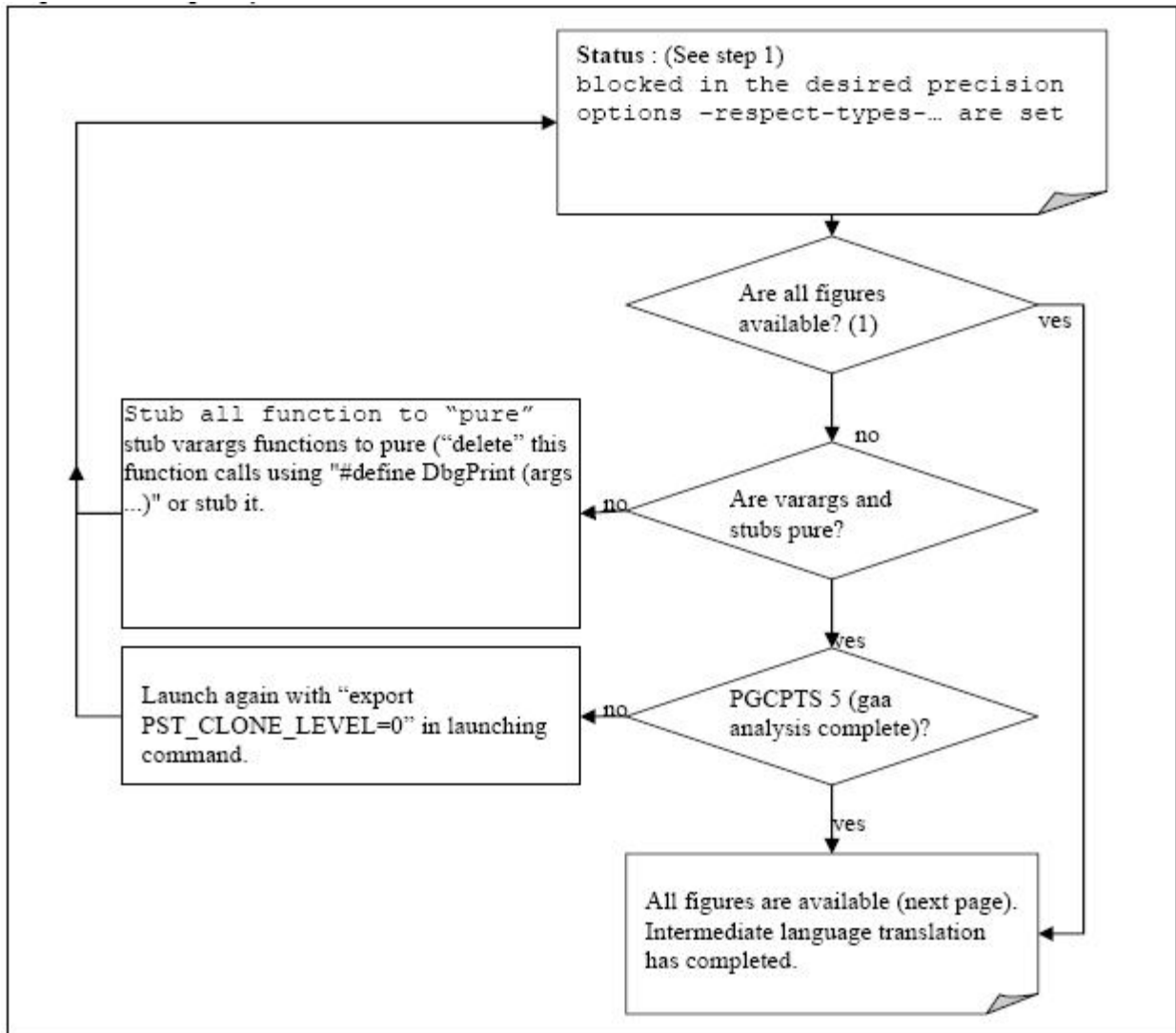
The size of a code sample which can be effectively analyzed can be increased by tuning the tool be optimized for that sample. Beyond that point, subdividing the code or choosing a lower precision level will bring better results (-O1, -O0).

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 :

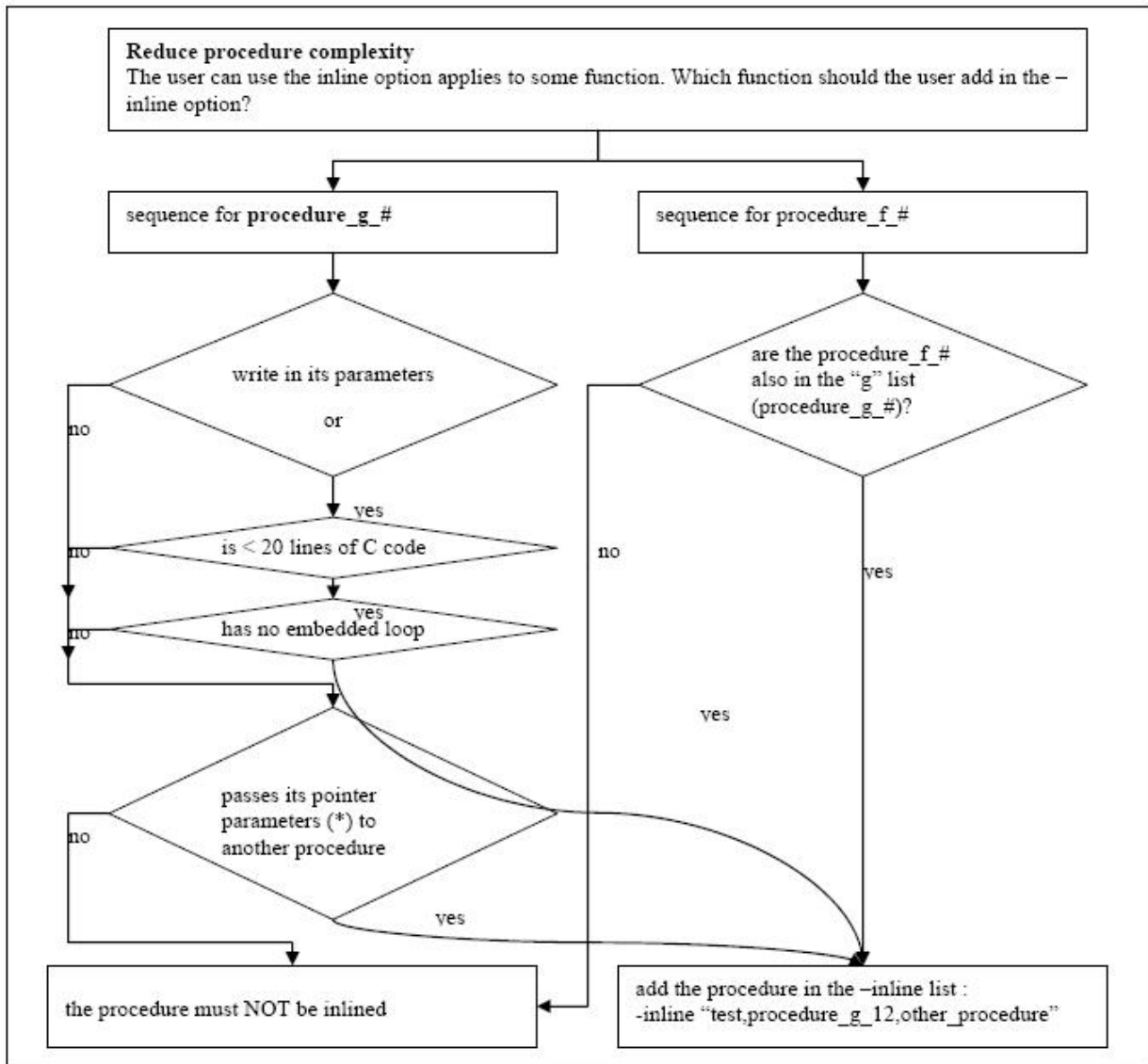
```
procedure_g_1      procedure_g_2
procedure_g_3
```

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 analysis.

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 analysis 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 analysis 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, analyzing 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 Verifier 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 Verifier 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 Verifier would simplify the internal data representation by using a less precise approximation, such as [-2 ; 2] U {10} U [15 ; 17] U {25} . Here, the same division appears as an orange check.

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

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

Applying Coding Rules to Reduce Oranges

In this section...
“MISRA® Rules Which PolySpace™ Verification Can Help to Follow” on page 9-87
“Recommended Set of Coding Rules” on page 9-87
“Approximations Made by PolySpace™ Verification” on page 9-92

MISRA® Rules Which PolySpace™ Verification Can Help to Follow

Rule #	Adv/Rec	Description
21.1	required	Provision should be made for appropriate run-time checking.
9.1	required	All automatic variables shall have been assigned a value before being used.
12.8	required	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the left hand operand.
12.11	advisory	Evaluation of constant unsigned integer expressions should not lead to wrap-around.
16.6	required	The number of arguments passed to a function shall match the number of parameters.
11.3	advisory	A cast should not be performed between a pointer type and an integral type (the null pointer shall not be de-referenced).

Recommended Set of Coding Rules

It is recommended that a subset of MISRA rules should be applied.

In addition, some constructions are known to produce a disproportionate number of orange checks. It will help to improve selectivity if these constructions are avoided at the design stage.

- “Set of Coding Rules with a Direct Impact on Selectivity” on page 9-88
- “Set of Coding Rules with an Indirect Impact on Selectivity” on page 9-89

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
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.

Rule #	Description
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.
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 sub-parts 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 coding rules 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™ analyses. 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 non-zero initialization of arrays and structures.
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 Cs 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.

Rule #	Description
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.
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: 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 pre-processing directives.

Rule #	Description
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 pre-processor directives shall be defined before use.
MISRA 19.12	There shall be at most one occurrence of the # or ## pre-processor 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 a coding rule not checked.

Approximations Made by PolySpace™ Verification

- “Volatile Variables” on page 9-93
- “Structures with Volatile Fields” on page 9-93
- “Absolute Addresses” on page 9-93
- “Pointer Comparison” on page 9-93
- “Left Shift on Negative Variables” on page 9-94
- “Some Bitwise Operators” on page 9-94
- “Float Loops” on page 9-95
- “Shared Variables” on page 9-96
- “Array of Function Pointers” on page 9-96
- “Trigonometric Functions” on page 9-97
- “Unions” on page 9-97
- “Loop Exit Conditions” on page 9-98
- “Constant Pointer” on page 9-99

Volatile Variables

Volatile variables are potentially uninitialized and their content is always full range.

```

2 int volatile_test (void)
3 {
4   volatile int tmp;
5   return(tmp); // NIV orange: the variable content is full range
6   [-2^31;2^31-1]
6 }

```

In the case of a global variable the content would also be full range, but the NIV check would be green.

Structures with Volatile Fields

In this example, although only the b field is declared as volatile, in practice any read access to the “a” field will be full range and orange.

```

2 typedef struct {
3   int a;
4   volatile int b;
5 } Vol_Struct;

```

Absolute Addresses

Both reading from, and writing to, an absolute address leads to warning checks on the pointer dereference. An absolute address is considered as a volatile variable.

```

Val = *((char *) 0x0F00); // NIV and IDP orange: access to an
absolute address

```

Pointer Comparison

PolySpace verification is a static tool analyzing source code. Memory management concerns dynamic considerations, and the characteristics of particular compilers and targets. PolySpace verification therefore doesn't consider where objects are actually implanted in memory

```

5 int *i, *j, k;

```

```
6 i = (int *) 0x0F00;
7 j = (int *) 0x0FF0;
8
9 if ( i < j ) // the condition can be true or false
10  k = 12; // this line is reachable
11 else
12  k = 23; // this line is reachable too.
```

Its the same situation if “i” and “j” points to real variable

```
6 i = & one_variable;
7 j = & another_one;
9 if ( i < j ) // the condition can still be true or false
```

Left Shift on Negative Variables

Consider the example below.

- When the option **-allow-negative-operand-in-shift** is not used, PolySpace verification gives a red error on the SHF check because behavior is compiler-dependant.
- When the option **-allow-negative-operand-in-shift** is used, y is always full range even if the signed value of x is known.

```
4 char x, y;
5 x = 0x8F;
6 y = x << 3 ; // OVFL and UNFL Warnings
```

Some Bitwise Operators

PolySpace results are not equally precise with all bitwise operators - AND, OR, XOR, and NOT (resp. `&`, `|`, `^`, `)`)

```
1 int random_uint(void);
2
3 void test (void)
4 {   unsigned int var1, var2, var3;
5   var1=0; var2=0;
6
7   // precision with zero on values with AND bitwise operator
8   var3= 0x01 & var2;
```



```

9  if (random_uint()) assert(var3==0);    // ASRT Checked
10 var3= 0x02 & 0xF3;
11  if (random_uint()) assert(var3==0x02); // ASRT checked
12  // Full range with other values
13  var3 = random_uint();
14  var3 = var3 & 0x02;
15  if (random_uint()) assert(var3==0x02 || var3==0); // ASRT
Warning
16
17  // Full range on values with OR bitwise operator
18  var3=var1|var2;
19  if (random_uint()) assert(var3==0);    // ASRT Warning
20  if (random_uint()) assert(var3!=0);    // ASRT Warning
21
22  // Full range on values with XOR bitwise operator
23  var3=var1^var2;
24  if (random_uint()) assert(var3==0);    // ASRT Warning
25  if (random_uint()) assert(var3!=0);    // ASRT Warning
26
27  // precision with zero values on NEGATIVE bitwise operator
28  var3 = ~var1;
29  if (random_uint()) assert(var3==0xFFFFFFFF); // ASRT checked
30  // precision on values with NEGATIVE bitwise operator
31  var3 = ~0xAE;
32  if (random_uint()) assert(var3==0xFFFFFFFF51); // ASRT
checked
33 }

```

Float Loops

Values on constructions are less precise when floats are used in loops.

```

5  int i;
6  double X = 0.0;
7
8  // less precision on float evaluation in loops
9  for (i = 0 ; i < 6; i++)
10 X = X + 10.56;    // OVFL warning
11 // VOA says 10.561 >= EXPR >= 10.559 OR EXP >= 21.119

```

Shared Variables

At the minimum, a shared variable contains a union of all ranges it can contain among the application. At the maximum, the variable will be full range.

```
12 void p_task1(void)
13 {
14   begin_cs();
15   X = 0;
16   if (X) {
17     Y = X; // Verified NIV, even it should be grey
18     assert (Y == 12); // Warning assert, even it should be grey
19   }
20   end_cs();
21 }
22
23 void p_task2(void)
24 {
25   begin_cs();
26   X = 12;
27   Y = X + 1; // Verifier considers [X==1] or [X==13]
28   if (Y == 13)
29     Y = 14;
30   else
31     Y = X - 1 ; // Verified checks even it should be grey
32   end_cs();
33 }
```

Array of Function Pointers

In the following example, PolySpace results show an orange check despite the test for a NULL function pointer test. However, it does accurately track the functions being called.

```
18 ptr_func array_func[] = {
19   f1,
20   f2,
21   NULL,
22 };
23
```

```
24 void main(void)
25 {
26     int i;
27
28     i = 0;
29     while (i < 3) {
30         if (array_func[i] != NULL)
31             array_func[i](); // function must point to a valid
function
32         i++; }

```

Trigonometric Functions

With all trigonometric functions such as cosines, sines etc., PolySpace verification always assumes that the return value is bound between the limits of that function - irrespective of the parameter passed to it. Consider the following example, which uses `acos`, `sin` and `asin` functions.

```
7 double res;
8
9 res = sin(3.141592654);
10 assert(res == 0.0); // VOA says [-1..1]
11
12 res = asin(0.0);
13 assert(res == 0.0); // VOA Always in [-pi/2..pi/2]
14
15 res = acos(0.0);
16 assert(res == 0.0); // VOA always in [0..pi]

```

Unions

It is recognized nonetheless that there are situations in which the careful use of unions is desirable in constructing an efficient implementation. Nevertheless, the kinds of implementation behavior that might be relevant are:

- **Padding:** padding could be inserted at the end of an union.
- **Alignment:** members of any structures within union could have different alignments.
- **Endianness:** whether the most significant byte of a word could be stored at the lowest or highest memory address.

- Bit-order: bits within bytes could have both different numbering and allocation to bit fields.

This why PolySpace verification can lose precision when structure unions are considered. Indeed this kind of implementation is compiler dependant. Conversions from one type a union to another will cause a loss of precision on two checks:

- Is the other field initialized? Orange NIV
- What is the content of the other field? Full range

```
typedef union _u {
  int a;
  char b[4]; } my_union;
my_union X;

X.b[0] = 1; X.b[1] = 1; X.b[2] = 1; X.b[3] = 1;
if (X.A == 0x1111)
  else // both branches are reachable
```

Loop Exit Conditions

PolySpace verification is more precise in loops where a test other than “does not equal” is used. Consider the loop index exit values in the following examples.

The orange check in this example

```
4 x = 0;
5 While (x != value)
6 {
7   ;
8   x++;
9 }
```

is not evident here:

```
5 While (x <= value)

8   x++;
```

Constant Pointer

To increase PolySpace precision where pointers are analyzed, replace

```
const int *p = &y;
```

with:

```
#define p (&y)
```


Options Description

General (p. 10-2)	Describes general options
Target/Compiler (p. 10-10)	Describes compiler options
Compliance with Standards (p. 10-22)	Describes compliance options
PolySpace™ Inner Settings (p. 10-32)	Describes options for PolySpace™ software settings
Precision/Scaling (p. 10-47)	Describes precision options
MultiTasking (PolySpace™ Server™ for C/C++ Product Only) (p. 10-59)	Describes multitasking options
Batch Options (p. 10-62)	Describes batch options
Complete Examples (p. 10-64)	Provides several examples of using PolySpace™ Client™ for C/C++ Verification

General

In this section...
“Overview” on page 10-2
“-prog Session identifier” on page 10-2
“-date Date” on page 10-3
“-author Author” on page 10-3
“-verif-version Version” on page 10-3
“-voa” on page 10-4
“-keep-all-files” on page 10-4
“-continue-with-red-error” on page 10-5
“-continue-with-existing-host” on page 10-5
“-allow-unsupported-linux” on page 10-5
“-results-dir Results Directory” on page 10-6
“-sources "files" or -sources-list-file file_name” on page 10-7
“-I directory ” on page 10-8

Overview

This section collates all options relating to the identification of the analysis, including the destination directory for the results and sources.

-prog Session identifier

This option specifies the application name, using only the characters which are valid for Unix file names. This information is labelled in the GUI as the *Session Identifier*.

Default:

Shell Script: polyspace

GUI: New_Project

Example shell script entry:

```
polyspace-c -prog myApp ...
```

-date Date

This option specifies a date stamp for the analysis in dd/mm/yyyy format. This information is labelled in the GUI as the *Date*. The GUI also allows alternative default date formats, via the Edit/Preferences window.

Default:

Day of launching the analysis

Example shell script entry:

```
polyspace-c -date "02/01/2002"...
```

-author Author

This option is used to specify the name of the author of the verification.

Default:

the name of the author is the result of the *whoami* command

Example shell script entry:

```
polyspace-c -author "John Tester"
```

-verif-version Version

Specifies the version identifier of the verification. This option can be used to identify different analyses. This information is identified in the GUI as the *Version*.

Default:

1.0.

Example shell script entry:

```
polyspace-c -verif-version 1.3 ...
```

-voa

When applied at launch time, this option enables the inspection of calculated domains for simple type assignments (scalar or float).

A new category of checks — named VOA — is generated on "=" of some scalar assignments to give the ranges. VOA checks are not available for volatile variables.

Default:

Disabled by default

Note Depending on code optimization, this check may not be present at all assignment locations

Example Shell Script Entry:

```
polyspace-c -voa ...
```

-keep-all-files

When this option is set, all intermediate results and associated working files are retained. Consequently, it is possible to restart Verifier from the end of any complete pass (provided the source code remains entirely unchanged). If this option is not used, it is only possible to restart Verifier from scratch.

By default, intermediate results and associated working files are erased when they are no longer needed by the Verifier.

-continue-with-red-error

Note This option may yield invalid results when used improperly.

Ordinarily, red errors (other than NTC) prevent PolySpace™ verification from continuing to the next integration pass. This option allows PolySpace verification to continue even if one of these red errors is encountered. In most cases, this will mean that the dynamic behavior of the code beyond the point where red errors are identified will be undefined, unless the red code is actually inaccessible.

Default:

Verifier stops upon finding red errors.

Example shell script entry :

```
polyspace-c -continue-with-red-error ...
```

-continue-with-existing-host

When this option is set, the analysis will continue even if the system is under specified or its configuration is not as preferred by PolySpace software. Verified system parameters include the amount of RAM, the amount of swap space, and the ratio of RAM to swap.

Default:

Verifier stops when the host configuration is incorrect or the system is under specified.

Example Shell Script Entry:

```
polyspace-c -continue-with-existing-host ...
```

-allow-unsupported-linux

This option specifies that PolySpace verification will be launched on an unsupported OS Linux® distribution.

In such case a warning is displayed in the log file against possible incorrect behaviors:

```
*****  
***  
***          WARNING          ***  
***  
*** You are running PolySpace Verifier on an ***  
*** unsupported Linux distribution. It may lead ***  
*** to incorrect behaviour of the product. Please ***  
*** note that no support will be available for ***  
*** this operating system.          ***  
***  
***** ** **
```

Default:

Disabled

Example Shell Script Entry:

```
polyspace-c allow-unsupported-linux ...
```

-results-dir Results Directory

This option specifies the directory in which Verifier will write the results of the analysis. Note that although relative directories may be specified, particular care should be taken with their use especially where the tool is to be launched remotely over a network, and/or where a project configuration file is to be copied using the "Save as" option.

Default:

Shell Script: The directory in which tool is launched.

From Graphical User Interface: C:\PolySpace_Results

Example Shell Script Entry:

```
polyspace-c -results-dir RESULTS ...  
export RESULTS=results_`date +%d%B_%HH%M_%A`  
polyspace-c -results-dir `pwd`/$RESULTS ...
```

-sources "files" or -sources-list-file file_name

-sources "file1[file2[...]]" (Linux and Solaris™)

or

-sources "file1[,file2[, ...]]" (windows, Linux and Solaris)

or

-sources-list-file file_name (not a graphical option)

List of source files to be analyzed, double-quoted and separated by commas.

Note UNIX® standard wild cards are available to specify a number of files.

Note The specified files must have valid extensions:
*.c|C|cc|cpp|CPP|cxx|CXX

Defaults:

sources/*.(c|C|cc|cpp|CPP|cxx|CXX)

Example Shell Script Entry under linux or solaris (*files are separated with a white space*):

```
polyspace-c -sources "my_directory/*.cpp" ...
polyspace-c -sources "my_directory/file1.cc other_dir/file2.cpp"
...
```

Example Shell Script Entry under windows (*files are separated with a comma*):

```
polyspace-c -sources "my_directory/file1.cpp,other_dir/file2.cc"
...
```

Using `-sources-list-file`, each file *name* need to be given with an absolute path. Moreover, the syntax of the file is the following:

- One file by line.
- Each file name is given with its absolute path.

Note This option is only available in batch mode

Example Shell Script Entry for `-sources-list-file`:

```
polyspace-c -sources-list-file "C:\Analysis\files.txt"  
polyspace-c -sources-list-file "/home/poly/files.txt"
```

-I directory

This option is used to specify the name of a directory to be included when compiling C sources. Only one directory may be specified for each `-I`, but the option can be used multiple times.

Default:

- When no directory is specified using this option, the `./sources` directory (if it exists) is automatically included
- If several `include-dir` are mentioned, the `./sources` directory (if it exists), is implicitly added at the end of the `"-I"` list

Example Shell Script Entry-1:

```
polyspace-c -I /com1/inc -I /com1/sys/inc
```

is equivalent to

```
polyspace-c -I /com1/inc -I /com1/sys/inc -I ./sources
```

Example Shell Script Entry-2:

```
polyspace-c
```

is equivalent to

```
polyspace-c -I ./sources
```

Target/Compiler

In this section...

“Overview” on page 10-10

“-target TargetProcessorType” on page 10-10

“GENERIC ADVANCED TARGET OPTIONS” on page 10-11

“-OS-target OperatingSystemTargetForPolySpaceStubs” on page 10-17

“-D compiler-flag” on page 10-17

“-U compiler-flag ” on page 10-18

“-include file_name” on page 10-18

“-post-preprocessing-command <file_name> or "command"” on page 10-19

“-post-analysis-command <file_name> or "command"” on page 10-20

Overview

This section allows details of the target processor and operating system to be specified. Header files should not be entered here; instead, include directories should be added using the relevant field under the Compile flag options.

-target TargetProcessorType

This option specifies the target processor type, and by doing so informs Verifier of the size of fundamental data types and of the endianness of the target machine.

Possible values are: sparc, m68k, powerpc, i386, c-167, tms320c3x, sharc21x61, necv850, mcpu, or generic target.

mcpu is a reconfigurable Micro Controller/Processor Unit target. One or more generic target can also be specified and saved. Also code which is to be run on an unlisted processor type can be analyzed using one of the other processor types listed, if the data properties which are relevant to Verifier are common. Refer to the “target specific issues” section for more details.

Instructions on the specification of a generic target and on the modification of the *mcpu* target are available in “GENERIC ADVANCED TARGET OPTIONS” on page 10-11.

Default:

sparc

Example shell script entry:

```
polyspace-c -target m68k ...
```

GENERIC ADVANCED TARGET OPTIONS

The previous *Generic target options* dialog box is only available when a *mcpu* target is selected. (*Enter the target name* in PolySpace™ Launcher)

Allows the specification of a generic "*Micro Controller/Processor Unit*" or *mcpu* target name. Initially, it is necessary to use the GUI to specify the name of a new *mcputarget* – say, “*MyTarget*”.

That new target is added to the *-target* options list. The new target’s default characteristics are as follows, using the *type [size, alignment]* format.

- *char [8, 8, char [16,16]]*
- *short [8,8], short [16, 16]*
- *int [16, 16]*
- *long [32, 32], long long [32, 32]*
- *float [32, 32], double [32, 32], long double [32, 32]*
- *pointer [16, 16]*
- *char is signed*

When using the command line, *MyTarget* is specified with all the options for modification:

```
polyspace-c -target MyTarget
```

For example, a specific target uses 8 bit alignment (see also `-align`), for which the command line would read:

```
polyspace-c -target mcpu -align 8
```

-default-sign-of-char [signed|unsigned]

This option is available for all targets. It allows a char to be defined as "signed", "unsigned", or left to assume the mcpu target's default behavior

- **default mode** – The sign of char is left to assume the target's default behavior. By default all targets are considered as signed except for hc08 and powerpc targets.
- **signed** – Disregards the target's default char definition, and specifies that a "signed char" should be used.
- **unsigned** – Disregards the target's default char definition, and specifies that a "unsigned char" should be used.

Example Shell Script Entry

```
polyspace-c -default-sign-of-char unsigned -target mcpu ...
```

-char-is-16bits

This option is only available when a *-mcpu* generic target has been chosen.

The default configuration of a generic target defines a char as 16 bits. This option changes it to 16 bits, irrespective of sign.

the minimum alignment of objects is also set to 16 bits and so, incompatible with the options `-short-is-8bits` and `-align 8`.

Setting the char type to 16 bits has consequences on the following:

- computation of size of for objects
- detection of underflow and overflow on chars

Without the option `char` for *mcpu* are 8 bits

Example shell script entry:

```
polyspace-c -target mcpu -char-is-16bits
```

-short-is-8bits

This option is only available when a *mcpu* generic target has been chosen.

The default configuration of a generic target defines a short as 16 bits. This option changes it to 8 bits, irrespective of sign.

It sets a short type as 8-bit without specific alignment. That has consequences for the following:

- computation of size of objects referencing short type
- detection of short underflow/overflow

Example shell script entry

```
polyspace-c -target mcpu -short-is-8bits
```

-int-is-32bits

This option is available with a *mcpu* generic target, hc08, hc12 and mpc5xx target has been chosen.

The default configuration of a generic target defines an int as 16 bits. This option changes it to 32 bits, irrespective of sign. Its alignment, when an int is used as struct member or array component, is also set to 32 bits. See also `-align` option.

Example shell script entry

```
polyspace-c -target mcpu -int-is-32bits
```

-long-long-is-64bits

This option is only available when a *mcpu* generic target has been chosen.

The default configuration of a generic target defines a long long as 32 bits. This option changes it to 64 bits, irrespective of sign. When a long long is used as struct member or array component, its alignment is also set to 64 bits. See also `-align` option.

Example shell script entry

```
polyspace-c -target mcpu -long-long-is-64bits
```

-double-is-64bits

This option is available when either a *mcpu* generic target or a *sharc21x61* target has been chosen.

The default configuration of a generic target defines a double as 32 bits. This option, changes both double and *long double* to 64 bits. When a double or long double is used as a struct member or array component, its alignment is set to 4 bytes.

See also `-align` option.

Defining the double type as a 64 bit double precision float impacts the following:

- Computation of sizeofobjects referencing double type
- Detection of floating point underflow/overflow

Example

```
int main(void)
{
  struct S {char x; double f;};
  double x;
  unsigned s1, s2;
  s1 = sizeof (double);
  s2 = sizeof(struct S);
  x = 3.402823466E+38; /* IEEE 32 bits float point maximum value */
  x = x * 2;
  return 0;
}
```

```
}

```

Using the default configuration of sharc21x62, C Verifier assumes that a value of 1 is assigned to s1, 2 is assigned to s2, and there is a consequential float overflow in the multiplication $x * 2$. Using the `-double-is-64bits` option, a value of 2 is assigned to s1, and no overflow occurs in the multiplication (because the result is in the range of the 64-bit floating point type)

Example shell script entry

```
polyspace-c -target mcpu -double-is-64bits
```

-pointer-is-32bits

This option is only available when a *mcpu* generic target has been chosen.

The default configuration of a generic target defines a pointer as 16 bits. This option changes it to 32 bits. When a pointer is used as struct member or array component, its alignment is also set also to 32 bits (see `-align` option).

Example shell script entry

```
polyspace-c -target mcpu -pointer-is-32bits
```

-align [8|16|32]

This option is available with a *mcpu* generic target and some other specific targets (with *hc08*, *hc12* or *mpc5xx* available values are 16 and 32). It is used to set the largest alignment of all data objects to 4/2/1 byte(s), meaning a 32, 16 or 8 bit boundary respectively.

The default alignment of a generic target is 32 bits. This means that when objects with a size of more than 4 bytes are used as struct members or array components, they are aligned at 4 byte boundaries.

Example shell script entry with a 32 bits default alignment

```
polyspace-c -target mcpu
```

-align 16. If the `-align 16` option is used, when objects with a size of more than 2 bytes are used as struct members or array components, they are aligned at 2 bytes boundaries.

Example shell script entry with a 16 bits specific alignment:

```
polyspace-c -target mcpu -align 16
```

-align 8. If the `-align 8` option is used, when objects with a size of more than 1 byte are used as struct members or array components, are aligned at 1 byte boundaries. Consequently the storage assigned to the arrays and structures is strictly determined by the size of the individual data objects without member and end padding.

Example shell script entry with a 8 bits specific alignment:

```
polyspace-c -target mcpu -align 8
```

-logical-signed-right-shift

In the Graphical User Interface, the user can choose between arithmetical and logical computation.

- **- Arithmetic:** the sign bit remains:

```
(-4) >> 1 = -2  
(-7) >> 1 = -4  
7 >> 1 = 3
```

- **- Logical:** 0 replaces the sign bit

```
(-4) >> 1 = (-4U) >> 1 = 2147483646  
(-7) >> 1 = (-7U) >> 1 = 2147483644  
7 >> 1 = 3
```

Example shell script entry

When using the command line, arithmetic is the default computation mode. When this option is set, logical computation will be performed.

```
polyspace-c -logical-signed-right-shift
```

-OS-target **OperatingSystemTargetForPolySpaceStubs**

This option specifies the operating system target for PolySpace stubs.

Possible values are 'Solaris', 'Linux', 'VxWorks', 'Visual' and 'no-predefined-OS'.

This information allows the appropriate system definitions to be used during preprocessing in order to analyze the included files properly. *-OS-target* no-predefined-OS may be used in conjunction with *-include* or/and *-D* to give all of the system preprocessor flags to be used at execution time. Details of these may be found by executing the compiler for the project in verbose mode. They are also listed in this document - search for keyword "OS-target option"

Default:

Solaris

Note Only the Linux® include files are provided with PolySpace software (see the include folder in the installation directory). Projects developed for use with other operating systems may be analyzed by using the corresponding include files for that OS. For instance, in order to analyze a VxWorks® project it is necessary to use the option *-I <<path_to_the_VxWorks_include_folder>>*

Example shell script entry:

```
polyspace-c -OS-target linux
polyspace-c -OS-target no-predefined-OS -D GCC_MAJOR=2 /
    -include /complete_path/inc/gn.h ...
```

-D compiler-flag

This option is used to define macro compiler flags to be used during compilation phase.

Only one flag can be used with each *-D* as for compilers, but the option can be used several times as shown in the example below.

Default:

Some defines are applied by default, depending on your `-OS-target` option.

Example Shell Script Entry:

```
polyspace-c -D HAVE_MYLIB -D USE_COM1 ...
```

-U compiler-flag

This option is used to undefine a macro compiler flags

As for compilers, only one flag can be used with each `-U`, but the option can be used several times as shown in the example below.

Default:

Some undefines may be set by default, depending on your `-OS-target` option.

Example Shell Script Entry:

```
polyspace-c -U HAVE_MYLIB -U USE_COM1 ...
```

-include file_name

This option is used to specify files to be included by each C file involved in the analysis.

Default:

No file is universally included by default, but directives such as `"#include <include_file.h>"` are acted upon.

Example Shell Script Entry:

```
polyspace-c -include `pwd`/sources/a_file.h -include  
/inc/inc_file.h ...
```

```
polyspace-c -include /the_complete_path/my_defines.h ...
```

-post-preprocessing-command <file_name> or "command"

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

Note We can have find each pre-processed 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 also keep the number of lines of the preprocessed file ci file. Adding a line or removing one could have some unpredictable behavior on the location of checks and MACROS in the PolySpace viewer.

Default:

No command.

Example Shell Script Entry – file name:

To remove the key word interrupt or @near, you can type the following command

```
polyspace-c -post-preprocessing-command  
`pwd`/remove_bad_keywords.sh
```

where remove_bad_keywords.sh is the following script:

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

Example Shell Command Entry:

This example performs the same function as that illustrated above, but specifies the command line directly:

```
polyspace-c -post-preprocessing-command "sed s/@near//g"
```

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

```
PolySpaceInstallDir\Verifier\tools\perl
```

-post-analysis-command <file_name> or "command"

When this option is used, the specified script file or command is executed once the analysis has completed.

The script or command is executed in the results directory of the analysis.

Execution occurs after the last part of the analysis. The last part of is determined by the `-to` option.

Note Depending of the architecture used, notably when using remote launcher, the script can be executed on the client side or the server side.

Default:

No command.

Example Shell Script Entry – file name:

This example shows how to send an email to tip the client side off that his analysis has been ended. This example supposes that the `mailx` command is available on the machine. So the command looks like:

```
polyspace-c -post-analysis-command `pwd`/end_email.sh
```

where `end_emails.sh` is the following script:

```
#!/bin/sh
echo analysis finished | mailx s PolySpace Analysis
ended name@domain.com
```

Example Shell Command Entry:

This example performs the same function as that illustrated above, but specifies the command line directly:

```
polyspace-c -post-analysis-command "mailx s \ PolySpace Analysis
ended\ \ name@domain.com\ "
```

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

```
PolySpaceInstallDir\Verifier\tools\perl
```

Compliance with Standards

In this section...
“-dos” on page 10-22
“Embedded Assembler” on page 10-23
“Strictness during analysis launching” on page 10-24
“Permissiveness during analysis launching” on page 10-25
“MISRA-C 2004 Rules” on page 10-28
“-dialect [iar keil]” on page 10-30

-dos

This option must be used when the contents of the **include** or **source** directory comes from a DOS or Windows® file system. It deals with upper/lower case sensitivity and control characters issues.

Concerned files are:

- header files: all include dir specified (-I option)
- source files: all sources files selected for the analysis (-sources option)

```
#include "..\mY_TEst.h"^M
```

```
#include "..\mY_other_FILE.H"^M
```

```
into
```

```
#include "../my_test.h"
```

```
#include "../my_other_file.h"
```

Default:

disabled by default

Example Shell Script Entry:

```
polyspace-c -I /usr/include -dos -I ./my_copied_include_dir -D test=1
```

Embedded Assembler

- “-discard-asm ” on page 10-23
- “Pragmas asm” on page 10-23

-discard-asm

This option instructs the PolySpace™ analysis to discard assembler code. If this option is used, the assembler code should be modelled in `c`.

This option is not compatible with `-asm-begin` and `-asm-end` options.

Default:

Embedded assembler is treated as an error.

Example Shell Script Entry:

```
polyspace-c -discard-asm ...
```

Pragmas asm

```
-asm-begin "mark1[mark2[...]] "
```

and

```
-asm-end "mark1[mark2[...]]"
```

This option is used to allow compiler specific asm functions to be excluded from the analysis, with the offending code block delimited by two `#pragma` directives.

Consider the following example.

```
#pragma asm_begin_1
```

```
int foo_1(void) { /* asm code to be ignored by PolySpace */ }
#pragma asm_end_1
#pragma asm_begin_2
void foo_2(void) { /* asm code to be ignored by PolySpace */ }
#pragma asm_end_2
```

Where "asm_begin_1" and "asm_begin_2" marks the beginning of asm sections which will be discarded and "asm_end_1", respectively "asm_end_2" mark the end of those sections.

Also refer to the -discard-asm option with regards to the following code:

```
asm int foo_1(void) { /* asm code to be ignored by PolySpace */ }
asm void foo_2(void) { /* asmcode to be ignored by PolySpace */ }
```

Example Shell Script Entry:

```
polyspace-c -discard-asm -asm-begin "asm_begin_1,asm_begin_2"
-asm-end "asm_end_1,asm_end_2" ...
```

Strictness during analysis launching

- “-strict” on page 10-24
- “-wall” on page 10-24

-strict

This option selects the Strict mode of PolySpace verification. It is equivalent to using the -Wall and -no-automatic-stubbingoptions simultaneously.

This option is not compatible with -asm-begin and -asm-end options.

-wall

When this option is used, the C compliance phase will print all warnings. For example, with this option, a warning will raise in the log file during

compilation phase when trying to write into a const variable: “warning: assignment of read-only member <var>”

Default:

By default, only warnings about compliance across different files are printed.

Example Shell Script Entry:

```
polyspace-c -Wall ...
```

Permissiveness during analysis launching

- “-permissive ” on page 10-25
- “-permissive-link” on page 10-25
- “-allow-non-int-bitfield” on page 10-26
- “-allow-undef-variables” on page 10-26
- “-ignore-constant-overflows ” on page 10-27
- “-allow-unnamed-fields” on page 10-27
- “-allow-negative-operand-in-shift ” on page 10-28

-permissive

This option selects the PolySpace permissive mode, which is equivalent to the simultaneous use of -allow-non-int-bitfield, -allow-undef-variables, -ignore-constant-overflows, -discard-asm, -permissive-stubber, -continue-with-red-error, and -permissive-link.

-permissive-link

When this option is used, PolySpace verification accepts integral type conflicts between declarations and definitions on arguments or/and returning functions.

It has an effect only

- when the size of a conflicting integral type is not greater than int, or
- conflicts occur between a pointer type and an integral type of same size.

Default:

By default, PolySpace verification does not accept any conflicts between declarations and definitions.

-allow-non-int-bitfield

This option allows the user to define types of bitfields other than those specified by ANSI® C. The standard accepts bitfields of signed and unsigned int types only.

Default:

Bitfields must be signed or unsigned int.

Example Shell Script Entry :

```
polyspace-c -allow-non-int-bitfield ...
```

-allow-undef-variables

When this option is used, PolySpace verification will continue in case of linkage errors due to undefined global variables. For instance when this option is used, PolySpace verification will tolerate a variable always being declared as extern

Default:

Undefined variables causes PolySpace verification to stop.

Example Shell Script Entry:

```
polyspace-c -allow-undef-variables ...
```


-ignore-constant-overflows

This option specifies that the analysis should be permissive with regards to overflowing computations on constants. Note that it deviates from the ANSI C standard.

For example,

```
char x = 0xff;
```

causes an overflow according to the standard, but if it is analyzed using this option it becomes effectively the same as

```
char x = -1;
```

With this second example, a red overflow will result irrespective of the use of the option.

```
char x = (rnd?0xFF:0xFE);
```

Default:

```
char x = 0xff; causes an overflow
```

Example Shell Script Entry:

```
polyspace-c -ignore-constant-overflows ...
```

-allow-unnamed-fields

When this option is used, PolySpace verification will continue in case of compilation errors due to un-named fields in structures. For instance when this option is used, PolySpace verification will tolerate a structure where fields are un-named since there are no duplicate names. With the option, the following source code is tolerate:

```
struct {  
    union { int x; int y;}  
    union {int z; int w;}  
} s;  
s.x = 2; s.z = 2;
```

Default:

Un-named fields cause PolySpace to stop.

Example Shell Script Entry:

```
polyspace-c -allow-unnamed-fields ...
```

-allow-negative-operand-in-shift

This option permits a shift operation on a negative number.

According to the ANSI C standard, such a shift operation on a negative number is illegal – for example,

```
-2 << 2
```

With this option in use, PolySpace verification considers the operation to be valid. In the example, the result would be

```
-2 << 2 = -8
```

Default:

A shift operation on a negative number causes a red error.

Example Shell Script Entry:

```
polyspace-c -allow-negative-operand-in-shift ...
```

MISRA-C 2004 Rules

- “-misra2 [all-rules | file_name]” on page 10-29
- “-includes-to-ignore "dir_or_file_path1[,dir_or_file_path2[,...]]” on page 10-29

-misra2 [all-rules | file_name]

This option permits to check set of coding rules in conformity to MISRA-C:2004. All MISRA checks are included in the log file of the analysis.

- Keyword *all-rules*: It checks all available MISRA C® rules. It implies the use of the default configuration: any violation of MISRA C rules is considered as a warning.
- Option *filename*: it is the name of an absolute ASCII file containing a list of MISRA® rules to check.

Format of the file:

```
<rule number> off|error|warning
# is considered as comments.
Example:
# MISRA configuration file for project C89
10.5 off # disable misra rule number 10.5
17.2 error # violation misra rule 17.2 as an error
17.3 warning # non-respect to misra rule 17.3 is a only a warning
```

Default:

disable

Example shell script entry:

```
polyspace-c -misra2 all-rules ...

polyspace-c -misra2 misra.txt
```

-includes-to-ignore "dir_or_file_path1[,dir_or_file_path2[,...]]"

This option prevents MISRA rules checking in a given list of files or directories (all files and sub-directories under selected directory). This option is useful when non-MISRA C conforming include headers are used. A warning is displayed if one of the parameter does not exist.

This option is authorized only when -misra2 is used.

Example shell script entry :

```
polyspace-c -misra2 misra.txt includes-to-ignore  
"c:\usr\include"
```

-dialect [iar|keil]

When this option is used, PolySpace verification will take into account some non Target Support Package™ TC6 syntax and semantic associated to a chosen dialect between iar and keil. It refers to the well known compilers of the company IAR (www.iar.com) and Keil (www.keil.com).

Using this option, PolySpace verification will tolerate some new structure types as keyword of the language such sfr, sbit, bit etc. These structures and associated semantic are part of compiler which have integrated it to the ANSI C language as an extension.

Example of source code with keil dialect:

```
unsigned char bdata Status[4];  
sfr AU = 0xF0;  
sbit OCmd = Status[0]^2;  
s^2 = 1; s^6 = 0;
```

Example with iar dialect:

```
unsigned char bdata Status[4];  
sfr OCmd @ 0x4FFE;  
OCmd.2 = 1; s.6 = 0;
```

Example Shell Script Entry:

```
polyspace-c dialect keil
```

-sfr-types <sfr_name>=<size_in_bits>,<sfr_name1>=<size_in_bits1>,...

Associated to the option -dialect, if the code uses specific sfr type keyword, it is **mandatory** to declare using **-sfr-types** option. It gives the name of the sfr type and its size in bits. The syntax is:

```
-sfr-types <sfr_name>=<size_in_bits>,
```

where <sfr_name> could be any name, but most of the time we encounter sfr, sfr16 and sfr32 . <size in bits> could be one of the values 8, 16 and 32.

Default:

No dialect used.

Example Shell Script Entry:

```
polyspace-c dialect iar sfr-types sfr=8,sfr32=32,sfrb=16
```

PolySpace™ Inner Settings

In this section...

“MAIN GENERATOR OPTIONS (-main-generator) for PolySpace™ Software” on page 10-32

“Stubbing” on page 10-35

“Assumptions” on page 10-37

“Automatic Orange Tester” on page 10-44

“Others” on page 10-45

MAIN GENERATOR OPTIONS (-main-generator) for PolySpace™ Software

This same option can be used for both PolySpace™ Client™ for C/C++ and PolySpace™ Server™ for C/C++, but the default behavior differs between the two:

- **Using PolySpace Server** the user has the choice as to whether to activate the option.
- **Using PolySpace Client** the option is activated by default.

This section describes:

- “PolySpace™ Client™ for C/C++ default behavior” on page 10-33
- “PolySpace™ Server™ for C/C++ default behavior” on page 10-33
- “-main-generator (detailed options)” on page 10-33
- “-main-generator-writes-variables [none | public | all | custom=v1,v2,] ” on page 10-34
- “-function-called-before-main function_name” on page 10-34
- “-main-generator-calls [none | unused | all | custom=f1,f2,...]” on page 10-35

PolySpace™ Client™ for C/C++ default behavior

There is no need to ascertain whether the code for analysis contains a "main" or not. That is automatically checked by the PolySpace Client for C/C++ product:

- If a main exists in the set of file(s), then the analysis continues with this main
- Otherwise the tool generates a main with default options:
 - main-generator-writes-variables public and call all unused functions
 - main-generator-calls unused.

PolySpace™ Server™ for C/C++ default behavior

By default, if no main found in a PolySpace Server for C/C++ analysis, then it will stop. This behavior can help isolate files missing from the analysis.

It is also possible to allow the PolySpace Server for C/C++ product to ascertain whether a main is available.

- if a main is found, the analysis continues as normal.
- if not, the tool generates a main with assumption of analyzing a library. Option used are -main-generator-writes-variables none and -main-generator-calls none.

-main-generator (detailed options)

This option initiates the default behavior for PolySpace™ Verifier. The generated main has two distinct purposes.

- It first initializes any variables identified in the first part of the option (-main-generator-writes-variables)
- It then calls a function which could be considered as a initialize function (-function-called-before-main)
- It then calls any functions identified in the second part of the option (-main-generator-calls) in a "while (random)" loop.

Each option is described separately in the following.

-main-generator-writes-variables [none | public | all | custom=v1,v2,..]

This option is used with the `-main-generator` option to dictate how the generated main will initialize global variables.

Settings available:

- `-none` — no global variable will be written by the main.
- `-public` — every variable except static and const variables are assigned a “random” value, representing the full range of possible values
- `-all` — every variable except const variables are assigned a “random” value, representing the full range of possible values
- `-custom` — only variables present in the list are assigned a “random” value, representing the full range of possible values

Example

```
polyspace-c -main-generator -main-generator-writes-variables none
polyspace-c -main-generator -main-generator-writes-variables
custom=variable_a,variable_b
```

-function-called-before-main function_name

It is possible to specify an initialization function that will be called on startup after the initialization of the global variables and before the main loop when using the `-main-generator` option.

The skeleton of the generated main looks like:

- 1 Initialization of global variables
- 2 Call the specified function frame
- 3 main loop with a call to all the specified function depending of option `-main-generator-calls`

Example shell script entry:

```
polyspace-c -main-generator function-called-before-main
MyInitFunction
```


-main-generator-calls [none | unused | all | custom=f1,f2,...]

The generated main will call functions according to this option. It is used with the `-main-generator` option, to specify the functions to be called.

Possible values:

- `none` — no function is called. This can be used with a multitasking application without a main (PolySpace Verifier only).
- `unused` (default) — every function is called by the generated main unless it is called elsewhere by the code under analysis.
- `all` — every function is called by the generated main except inlined.
- `custom` — only functions present in the list are called from the main. Inlined functions can be specified in the list.

An inline (static or extern) function is not called by the generated main program with values `all` or `unused`. An inline function can only be called with custom value: `-main-generator-calls custom=my_inlined_func`.

Example:

```
polyspace-c -main-generator -main-generator-calls public
```

```
polyspace-c -main-generator -main-generator-calls  
custom=function_1,function_2
```

Stubbing

- “`-data-range-specifications file_name`” on page 10-35
- “`-permissive-stubber`” on page 10-36
- “`-no-automatic-stubbing`” on page 10-36

-data-range-specifications file_name

This option permits the setting of specific data ranges for a list of given global variables. This option is protected by a license.

File format:

The file filename contains a list of global variables with the below format:

```
variable_name val_min val_max <init|permanent|globalassert>
```

Variables scope:

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

Note Only one mode can be applied to a global variable.

No checks are added with this option except for `globalassert` mode.

Some warning can be displayed in log file concerning variables when format or type is not in the scope.

Default:

Disable.

Example shell script entry:

```
polyspace-c -data-range-specifications range.txt ...
```

-permissive-stubber

By default, the stubber rejects functions:

- with complex function pointers as parameters
- with function pointers as return type

With this option, all functions are stubbed, at any cost (results may be wrong).

-no-automatic-stubbing

By default, PolySpace verification automatically stubs all functions. When this option is used, the list of functions to be stubbed is displayed and the analysis is stopped.

Benefits:

This option may be used where

- The entire code is to be provided, which may be the case when analyzing a large piece of code. When the analysis stops, it means the code is not complete.
- Manual stubbing is preferred to improve the selectivity and speed of the analysis.

Default:

All functions are stubbed automatically

Assumptions

- “-div-round-down ” on page 10-37
- “-no-def-init-glob ” on page 10-38
- “-size-in-bytes ” on page 10-39
- “-allow-ptr-arith-on-struct ” on page 10-39
- “-ignore-float-rounding” on page 10-41
- “-detect-unsigned-overflows ” on page 10-43
- “-known-NTC proc1[,proc2[,...]]” on page 10-44

-div-round-down

This option concerns the division and modulus of a negative number.

The ANSI® standard stipulates that *"if either operand of / or % is negative, whether the result of the / operator, is the largest integer less or equal than the algebraic quotient or the smallest integer greater or equal than the quotient, is implementation defined, same for the sign of the % operator"*.

Note $a = (a / b) * b + a \% b$ is always true.

Default:

Without the option (default mode), if either operand of `/` or `%` is negative, the result of the `/` operator is the smallest integer greater or equal than the algebraic quotient. The result of the `%` operator is deduced from $a \% b = a - (a / b) * b$

Example:

```
assert(-5/3 == -1 && -5%3 == -2); is true .
```

With the *-div-round-down* option:

If either operand `/` or `%` is negative, the result of the `/` operator is the largest integer less or equal than the algebraic quotient. The result of the `%` operator is deduced from $a \% b = a - (a / b) * b$.

Example:

```
assert(-5/3 == -2 && -5%3 == 1); is true .
```

Example Shell Script Entry:

```
polyspace-c -div-round-down ...
```

-no-def-init-glob

This option specifies that PolySpace verification should not take into account default initialization defined by ANSI C. When this option is not used, default initialization are

- 0 for integers
- 0 for characters
- 0.0 for floats

With the option in use, all global variable will be treated as non initialized - and therefore cause a red error - if they are read before being written to.

Example Shell Script Entry :

```
polyspace-c -no-def-init-glob ...
```

-size-in-bytes

This option allows incomplete or partial allocation of structures. This allocation can be made by malloc or cast .

The example below shows an example using malloc. Further explanation can be found in the section describing the partial and incomplete allocation of structures. Also refer to the -allow-ptr-arith-on-struct section.

```
typedef struct _little { int a; int b; } LITTLE;
typedef struct _big { int a; int b; int c; } BIG;
BIG *p = malloc(sizeof(LITTLE));
```

Default results

```
p->a = 0 ; // red pointer out of its bounds
or p->b = 0 ; // red pointer out of its bounds
or p->c = 0 ; // red pointer out of its bounds
```

Results using this option

```
if (p!= ((void *) 0) ) {
  p->a = 0 ; // green pointer within bounds
or p->b = 0 ; // green pointer within bounds
or p->c = 0 ; // red pointer out of its bounds
}
```

-allow-ptr-arith-on-struct

This option enables navigation within a structure or union from one field to another, within the rules defined below. It automatically sets the -size-in-bytes option.

Default

By default, when a pointer points to a variable then the size of the object pointed to is that of that variable - irrespective of whether it is contained

within a bigger object, like a structure. Therefore, going out of the scope of this variable leads to a red IDP check (Illegal Dereference Pointer). This is illustrated below.

```
struct S {char a; char b; int c;} x;
char *ptr = &x.b;
ptr ++;
*ptr = 1; // red on the dereference, because the pointed
object was "b"
```

Using this option

When this option is used in the above option, PolySpace verification considers that the object pointed to is now the host object "x". The "ptr" pointer is in fact pointing to &x, with the correct offset to the field "b" within the structure of type S (inter-fields and end-padding included). Therefore, the dereference becomes green

Consider a second example:

```
struct S {
  char a;
  /* 3 bytes of padding between 'a', 'b' */
  int b;
  int c;
  char d[3];
  unsigned char e:7;
  char f;
  /* 3 bytes of end padding */
} x;
char *ptr;
struct Nesting_S {
  struct S s;
  int c;
} z
ptr = (char *)&x.a; ptr++; *ptr = 10; // ptr points to the
padding between a and b
ptr = (char *)&x.b; ptr += 4; *ptr = 10; // ptr points to the
first byte of c
ptr = (char *)&x.d; ptr += 3; *ptr = 10; // ptr points to the
ptr = (char *)&x.f; ptr++; *ptr = 10; // ptr points to the
```

first byte of end-padding

Note For nested structures, for instance with `ptr = (char *)&x.d.a`, the dereference of `*ptr` is green if `ptr` remains within `x.d`. However, even with this option in use, a red check is generated if the pointer navigates above `x.d.a`. That is, if this pointer is incremented or decremented such that it now points to `x.a`, `x.b`, or `x.c`, it causes a red IDP.

In the third example below, the `*ptr` access is red irrespective of whether the option is set or not.

With the option set, the `ptr` pointer points to the structure+offset `z.s`, and `ptr` can safely navigate within this structure `z.s`, but `z.c` is outside it.

Without the option, the `ptr` pointer points to `z.s.f`, which is only 1 byte long. So no navigation is allowed, not even within `z.s`.

```
ptr = (char *)z.s.f; ptr += 4; *ptr = 10; // ptr points to the
first byte of c:
```

-ignore-float-rounding

Without this option, PolySpace verification rounds floats according to the IEEE® 754 standard: simple precision on 32-bits targets and double precision on target which define double as 64-bits.

With the option, **exact** computation is performed.

Example:

```
1
2 void ifr(float f)
3 {
4   double a = 1.27;
5   if ((double)1.27F == a) {
6     assert (1);
7     f = 1.0F * f;
8     // reached when -ignore-float-rounding is used or not
9   }
```

```
10 else {
11   assert (1);
12   f = 1.0F * f;
13   // reached when compiled when -ignore-float-rounding is not used
14 }
15 }
```

Using this option can lead to different results compared to the "real life" (compiler and target dependent): Some paths will be reachable or not for PolySpace verification while they are not (or are) depending of the compiler and target. So it can potentially give approximate results (green should be unproven). This option has an impact on OVFL/UNFL checks on floats.

However, this option allows reducing the number of unproven checks because of the "delta" approximation.

For example:

- FLT_MAX (with option set) = 3.40282347e+38F
- FLT_MAX (following IEEE 754 standard) = 3.40282347e+38F ± Δ

```
1
2 void ifr(float f)
3 {
4   double a = 1.27;
5   if ((double)1.27F == a) {
6     assert (1);
7     f = 1.0F * f; // Overflow never occurs because f <= FLT_MAX.
8                 // reached when -ignore-float-rounding is used
9   }
10  else {
11    assert (1);
12    f = 1.0F * f; // OVFL could occur when f = (FLT_MAX + D)
13                // reached when -ignore-float-rounding is not used
14  }
15 }
```

Default:

IEEE 754 rounding under 32 bits and 64 bits.

Example Shell Script Entry:

```
polyspace-c -ignore-float-rounding ...
```

-detect-unsigned-overflows

When this option is selected, PolySpace verification becomes more pedantic than the ANSI C standard requires, with regards overflowing computations on unsigned integers

Consider the examples below, which apply when the option is in use.

Example 1

```
unsigned char x;  
  
x = 255;  
  
x = x+1;//causes an overflow according to this option.
```

Example 2

```
unsigned char y=1;  
  
y &= ~y; //causes an overflow because of type promotion
```

Default:

Without this option in place, Example 1 would generate no error.

```
unsigned char x;  
  
x = 255;  
  
x = x+1;// turns x into 0 (wrap around)
```

Example Shell Script Entry:

```
polyspace-c -detect-unsigned-overflows ...
```

-known-NTC proc1[,proc2[,...]]

After a few analyses, you may discover that a few functions "never terminate". Some functions such as tasks and threads contain infinite loops by design, while functions that exit the program such as *kill_task* , *exit* or *Terminate_Thread* are often stubbed by means of an infinite loop. If these functions are used very often or if the results are for presentation to a third party, it may be desirable to filter all NTC of that kind in the Viewer.

This option is provided to allow that filtering to be applied. All NTC specified at launch will appear in the viewer in the known-NTC category, and filtering will be possible.

Default :

All checks for deliberate Non Terminating Calls appear as red errors, listed in the same category as any problem NTC checks.

Example Shell Script Entry :

```
polyspace-c -known-NTC "kill_task,exit"
```

```
polyspace-c -known-NTC "Exit,Terminate_Thread"
```

Automatic Orange Tester**-prepare-automatic-tests**

This option activates the PolySpace Automatic Orange Tester. The Automatic Orange Tester finds run-time errors in the orange (and red) checks remaining at the end of the PolySpace verification.

The Automatic Orange Tester results contain precise information to help you identify the cause of a run-time error. This complements the results review in the Viewer module of PolySpace Client for C/C++.

For more information, see "Automatically Testing Orange Code " on page 9-33.

The following options are not compatible with `-prepare-automatic-tests`.

- `-entry-points`
- `-dialect`
- `-ignore-float-rounding`
- `-div-round-down`
- `-entry-points`
- `-char-is-16bits`
- `-short-is-8bits`
- `-respect-types-in-globals`
- `-respect-types-in-fields`

The following options cannot take specific values when you select `-prepare-automatic-tests`.

- `-align [8|16]`
- `-target [c-167 | tms320c3c | hc08 | sharc21x61]`
- `-data-range-specification` (in global assert mode)

In addition, when using the Automatic Orange Tester, the `-target mcpu` option must be used together with `-pointer-is-32bits`.

Default :

Disabled

Example Shell Script Entry :

```
polyspace-c -prepare-automatic-tests ...
```

Others

- “`-extra-flags option-extra-flag`” on page 10-46
- “`-c-extra-flags flag`” on page 10-46

-extra-flags option-extra-flag

This option specifies an expert option to be added to the analyzer. Each word of the option (even the parameters) must be preceded by *-extra-flags*.

These flags will be given to you by Tehnical Support as necessary for your analyses.

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-c -extra-flags -param1 -extra-flags -param2 \  
-extra-flags 10 ...
```

-c-extra-flags flag

This option is used to specify an expert option to be added to an analysis. Each word of the option (even the parameters) must be preceded by *-c-extra-flags*.

These flags will be given to you by The Mathworks as necessary for your analyses.

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-c -c-extra-flags -param1 -c-extra-flags -param2  
-c-extra-flags 10
```

Precision/Scaling

In this section...

“-quick” on page 10-47
 “-O(0-3)” on page 10-48
 “-modules-precision mod1:O(0-3)[,mod2:O(0-3)[,...]] ” on page 10-49
 “-from verification-phase” on page 10-49
 “-to verification-phase” on page 10-50
 “-context-sensitivity "proc1[,proc2[,...]]” on page 10-51
 “-context-sensitivity-auto” on page 10-51
 “-path-sensitivity-delta number” on page 10-52
 “-retype-pointer ” on page 10-52
 “-retype-int-pointer” on page 10-53
 “-k-limiting number ” on page 10-54
 “-no-fold ” on page 10-55
 “-respect-types-in-globals” on page 10-55
 “-respect-types-in-fields” on page 10-56
 “-inline "proc1[,proc2[,...]]” on page 10-57
 “-lightweight-thread-model” on page 10-57

-quick

This option is used to select a very fast mode for PolySpace™ C Verifier. This option cannot be used with the -O(0-3), -from, -to and -modules-precision options.

Benefits

This option allows results to be generated very quickly. These are suitable for initial analysis of red and grey errors only, as orange checks are too plentiful to be relevant using this option.

Quick mode is up to 25 times faster than a typical analysis using a specified combination of precision level and integration level.

Limitations

- No NTL or NTC are displayed (non termination of loop/call)
- The variable dictionary is not available
- No check is performed on floats
- The call tree is available but navigation is not possible
- Orange checks are too plentiful to be relevant

-O(0-3)

This option specifies the precision level to be used. It provides higher selectivity in exchange for more analysis time, therefore making results review more efficient and hence making bugs in the code easier to isolate. It does so by specifying the algorithms used to model the program state space during analysis.

It is recommended that analyses should begin with the -quick option. Red errors and grey code can then be addressed before re-launching Verifier using this option, applying a precision level as described below.

Benefits:

- A higher precision level contributes to a higher selectivity rate, making results review more efficient and hence making bugs in the code easier to isolate.
- A higher precision level also means higher analysis time
 - -O0 corresponds to static interval analysis.
 - -O1 corresponds to complex polyhedron model of domain values.
 - -O2 corresponds to more complex algorithms to closely model domain values (a mixed approach with integer lattices and complex polyhedrons).
 - -O3 is only suitable for code smaller than 1000 lines of code. For such codes, the resulting selectivity might reach high values such as 98%,

resulting in a very long analysis time, such as an hour per 1000 lines of code.

Default:

-O2

Example Shell Script Entry:

```
polyspace-c -O1 -to pass4 ...
```

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

This option is used to specify the list of .c files to be analyzed with a different precision from that specified generally -O(0-3) for this analysis.

In batch mode, each specified module is followed by a colon and the desired precision level for it. Any number of modules can be specified in this way, to form a comma-separated list with no spaces.

Default:

All modules are treated with the same precision.

Example Shell Script Entry:

```
polyspace-c -O1 \  
-modules-precision myMath:O2,myText:O1, ...
```

-from verification-phase

This option specifies the verification phase to start from. It can only be used on an existing analysis, possibly to elaborate on the results that you have already obtained.

For example, if an analysis has been completed -to pass1, PolySpace verification can be restarted *-from* pass1 and hence save on analysis time.

The option is usually used in an analysis after one run with the -to option, although it can also be used to recover after power failure.

Possible values are as described in the *-to verification-phase* section, with the addition of the *scratch* option.

Note

- Unless the *scratch* option is used, this option can be used only if the previous analysis was launched using the option *-keep-all-files* .
 - This option cannot be used if you modify the source code between analyses.
-

Default :

scratch

Example Shell Script Entry :

```
polyspace-c -from c-to-il ...
```

-to verification-phase

This option specifies the verification phase after which the Verifier will stop.

Benefits:

This option provides improved selectivity, making results review more efficient and making bugs in the code easier to isolate.

- A higher integration level contributes to a higher selectivity rate, leading to "finding more bugs" with the code.
- A higher integration level also means higher analysis time

Possible values:

- *c-compile* or "C Source Compliance Checking"
- *c-to-il* or *normalize* or "C to Intermediate Language"
- *pass0* or *CDFA* or "Control and Data Flow Analysis"

- pass1 or "Software Safety Analysis level 1"
- pass2 or "Software Safety Analysis level 2"
- pass3 or "Software Safety Analysis level 3"
- pass4 or "Software Safety Analysis level 4"
- other

Note If you use *-to other* then PolySpace verification will continue until you stop it manually (via `kill -rte -kernel`) or stops until it has reached *pass20*.

Default:

pass4

Example Shell Script Entry:

```
polyspace-c -to "Software Safety Analysis level 3"...
```

```
polyspace-c -to pass0 ...
```

-context-sensitivity "proc1[,proc2[,...]]"

This option allows the precise analysis of a procedure with regards to the discrete calls to it in the analyzed code.

Each check inside the procedure is split into several sub-checks depending on the context of call. Therefore if a check is red for one call to the procedure and green for another, both colors will be revealed.

This option is especially useful if a problem function is called from a multitude of places.

-context-sensitivity-auto

This option is similar to the `-context-sensitivity` option, except that the system automatically chooses the procedures to be considered.

-path-sensitivity-delta number

This option is used to improve interprocedural analysis precision within a particular pass (see *-to pass1, pass2, pass3* or *pass4*). The propagation of information within procedures is done earlier than usual when this option is specified. That results in improved selectivity and a longer analysis time.

Consider two analyses, one with this option set to 1 (with), and one without this option (without)

- a level 1 analysis in (with) (pass1) will provide results equivalent to level 1 or 2 in the (without) analysis
- a level 1 analysis in (with) can last x times more than a cumulated level 1+2 analysis from (without). "x" might be exponential.
- the same applies to level 2 in (with) equivalent to level 3 or 4 in (without), with potentially exponential analysis time for (a)

Gains using the option

- (+) highest selectivity obtained in level 2. no need to wait until level 4
- (-) This parameter increases exponentially the analysis time and might be even bigger than a cumulated analysis in level 1+2+3+4
- (-) This option can only be used with less than 1000 lines of code

Default:

0

Example Shell Script Entry:

```
polyspace-c -path-sensitivity-delta 1 ...
```

-retype-pointer

This option can be used to retype variables of pointer types in order to improve precision of pointer conversions chain.

The principle consists in replacing original type by the aliased object type when a symbol of pointer type aliases to a single type of objects.

For example, following assert can be proved using `-retype-pointer` option:

```
struct A {int a; char b;} s = {1,2};
char *tmp = (char *)&s;
struct A *pa = (struct A*)tmp;
assert((pa->a == 1) && (pa->b == 2));
```

This principle can be applied to fields of struct/unions of a pointer type. However, this option set `-size-in-bytes` option and it does not have expected precision with `-allow-ptr-arith-on-struct`.

Moreover, this option is forbidden when using `-retype-int-pointer` option.

Default:

disable by default

Example Shell ScriptEntry:

```
polyspace-c -retype-pointer ...
```

-retype-int-pointer

This option can be used to retype variables of pointer to signed or unsigned integer types in order to improve precision of pointer conversions chain.

The principle consists in replacing original type by the aliased object type when a symbol of pointer type aliases to a single type of objects. It applies only on symbols of signed or unsigned integer types.

For example, following assert can be proved using `-retype-int-pointer` option:

```
void function(void)
{
  struct S1 {
    int x;
    int y;
    int z;
    char t;
  } s1 = {1,2,3,4};
```

```
struct S2 {
  int first;
  void *p;
} s2;
int addr;
addr = (int)&s1;
assert(((struct S1 *)addr)->y == 2); // ASRT is verified
s2.first = (int)&s1;
assert(((struct S1 *)s2.first)->y == 2); // ASRT is verified
}
```

However, this option set `-size-in-bytes` and has no effect when set `-respect-types-in-globals` on global symbols of integer types and when set `-respect-types-in-fields` on fields of struct/union of integer types.

Some sides effects can be noticed on PolySpace checks concerning initialization on variables which can be stated as initialization on pointer check (NIP).

Moreover, this option implies `-retype-pointer` option.

Default:

Disable by default

Example Shell ScriptEntry:

```
polyspace-c -retype-int-pointer...
```

-k-limiting number

This is a scaling option to limits the depth of analysis into nested structures during pointer analysis.

This option is only available for C and C++.

Default:

There is no fixed limit.

Example Shell Script Entry:

```
polyspace-c -k-limiting 1 ...
```

In this example above, analysis will be precise to only one level of nesting.

-no-fold

When variables are defined with huge static initialization, scaling problems may occur during the compilation phase. This option approximates the initialization of array types of integer, floating point, and char types (included string) if needed.

It can speed up the analysis, but may decrease precision for some applications

Default:

Option not set.

Example Shell Script Entry:

```
polyspace-c -no-fold ...
```

-respect-types-in-globals

This is a scaling option, designed to help process complex code. When it is applied, PolySpace verification assumes that global variables not declared as containing pointers are never used for holding pointer values. This option should only be used with Type-safe code, when it does not cause a loss of precision. See also `-respect-types-in-fields`.

In the following example, we will lose precision using option `-respect-types-in-globals` option:

```
int x;
void t1(void) {
  int y;
  int *tmp = &x;
  *tmp = (int)&y;
  y=0;
  *(int*)x = 1; // x contains address of y
  assert (y == 0); // green with the option
```

```
}
```

PolySpace verification will not take care that `x` contains the address of `y` resulting a green assert.

Default:

PolySpace verification assumes that global variables may contain pointer values.

Example Shell Script Entry:

```
polyspace-c -respect-types-in-globals ...
```

-respect-types-in-fields

This is a scaling option, designed to help process complex code. When it is applied, PolySpace verification assumes that structure fields not declared as containing pointers are never used for holding pointer values. This option should only be used with Type-safe code, when it does not cause a loss of precision. See also `-respect-types-in-globals`.

In the following example, we will lose precision using option `-respect-types-in-fields` option:

```
struct {
    unsigned x;
    int f1;
    int *z[2];
} S1;

void funct2(void) {
    int *tmp;
    int y;
    ((int**)&S1)[0] = &y; /* S1.x points on y */
    tmp = (int*)S1.x;
    y=0;
    *tmp = 1; /* write 1 into y */
    assert(y==0);
}
```

PolySpace verification will not take care that S1.x contains the address of y resulting a green assert.

Default:

PolySpace verification assumes that structure fields may contain pointer values.

Example Shell Script Entry:

```
polyspace-c -respect-types-in-fields ...
```

-inline "proc 1[,proc2[,...]]"

A scaling option that creates a clone of a each specified procedure for each call to it.

Cloned procedures follow a naming convention viz:

```
procedure1_pst_cloned_nb,
```

where nb is a unique number giving the total number of cloned procedures.

Such an inlining allows the number of aliases in a given procedure to be reduced, and may also improve precision.

Restrictions :

- Extensive use of this option may duplicate too much code and may lead to other scaling problems. Carefully choose procedures to inline.
- This option should be used in response to the inlining hints provided by the alias analysis
- This option should not be used on main, task entry points and critical section entry points

-lightweight-thread-model

This scaling option can be used to reduce task complexity (see also -entry-points).

It uses a slightly less precise model of pointer/thread interaction compared to that used by default, and is likely to prove helpful when there are a lot of pointers in an application. See Chapter 9, “PolySpace™ Methodological Guide” for more explanation of when to use it.

It causes a loss of precision:

- It causes a slight loss of precision when shared variables are reads via pointers.
- Some read/write accesses may not appear in the Global Variable Dictionary.

Default:

disabled by default.

Example Shell Script Entry :

```
polyspace-c -lightweight-thread-model ...  
polyspace-c -lwtm ...
```


MultiTasking (PolySpace™ Server™ for C/C++ Product Only)

In this section...

“-entry-points str1[,str2[,...]]” on page 10-59

“-critical-section-[begin or end] "proc1:cs1[,proc2:cs2]"” on page 10-59

“-temporal-exclusions-file file_name” on page 10-60

Note Concurrency options are not compatible with -main-generator options.

-entry-points str1[,str2[,...]]

This option is used to specify the tasks/entry points to be analyzed by PolySpace™ server, using a Comma-separated list with no spaces.

These entry points must not take parameters. If the task entry points are functions with parameters they should be encapsulated in functions with no parameters, with parameters passed through global variables instead.

Using PolySpace verification, c tasks must have the prototype "void *task_name*(void);".

Example Shell Script Entry:

```
polyspace-c -entry-points proc1,proc2,proc3 ...
```

-critical-section-[begin or end] "proc1:cs1[,proc2:cs2]"

```
-critical-section-begin "proc1:cs1[,proc2:cs2]"
```

and

```
-critical-section-end "proc3:cs1[,proc4:cs2]"
```

These options specify the procedures beginning and ending critical sections, respectively. Each uses a list enclosed within double speech marks, with list

entries separated by commas, and no spaces. Entries in the lists take the form of the procedure name followed by the name of the critical section, with a colon separating them.

These critical sections can be used to model protection of shared resources, or to model interruption enabling and disabling.

Default:

no critical sections.

Example Shell Script Entry:

```
polyspace-c -critical-section-begin "start_my_semaphore:cs" \  
-critical-section-end "end_my_semaphore:cs"
```

-temporal-exclusions-file file_name

This option specifies the name of a file. That file lists the sets of tasks which never execute at the same time (temporal exclusion).

The format of this file is :

- one line for each group of temporally excluded tasks,
- on each line, tasks are separated by spaces.

Default:

No temporal exclusions.

Example Task Specification file

File named 'exclusions' (say) in the 'sources' directory and containing:

```
task1_group1 task2_group1  
task1_group2 task2_group2 task3_group2
```

Example Shell Script Entry :

```
polyspace-c -temporal-exclusions-file sources/exclusions \  
-entry-points task1_group1,task2_group1,task1_group2,\  
task2_group2,task3_group2 ...
```

Batch Options

In this section...

“-server server_name_or_ip[:port_number]” on page 10-62

“-sources-list-file file_name” on page 10-62

“-v | -version” on page 10-63

“-h[elp]” on page 10-63

-server server_name_or_ip[:port_number]

Using `polyspace-remote[-desktop]-[ada] [-server [name or IP address][:<port number>]]` allows to send analysis to a specific or referenced PolySpace Queue manager server.

Note If the option `-server` is not specified, the default server referenced in the `PolySpace-Launcher.prf` configuration file will be used as server.

When a `-server` option is associated to the batch launching command, the name or IP address and a port number need to be specified. If the port number does not exist, the 12427 value will be used by default.

Note also that `polyspace-remote-` accepts all other options.

Option Example Shell Script Entry:

```
polyspace-remote-desktop-c server 192.168.1.124:12400
```

```
polyspace-remote-c
```

```
polyspace-remote-c server Bergeron
```

-sources-list-file file_name

This option is only available in batch mode. The syntax of *file_name* is the following:

- One file per line.
- Each file name includes its absolute or relative path.

Example Shell Script Entry for -sources-list-file:

```
polyspace-c -sources-list-file "C:\Analysis\files.txt"  
  
polyspace-c -sources-list-file "files.txt"
```

-v | -version

Display the PolySpace™ version number.

Example Shell Script Entry:

```
polyspace-c v
```

It will show a result similar to:

```
PolySpace r2007a+
```

```
Copyright (c) 1999-2008 The Mathworks, Inc.
```

-h[elp]

Display in the shell window a simple help in a textual format giving information on all options.

Example Shell Script Entry:

```
polyspace-c h
```

Complete Examples

In this section...
“Simple C Example” on page 10-64
“Apache Example” on page 10-64
“cxref Example” on page 10-65
“T31 Example” on page 10-65
“Dishwasher1 Example” on page 10-65
“Satellite Example” on page 10-66

Simple C Example

```
polyspace-c \  
-prog myCproject \  
-O1 \  
-I /home/user/includes \  
-D SUN4 -D USE_FILES \  

```

Apache Example

Here is a script for verifying the code for Apache (after proper formatting). The source code is in C and the compilation is for a Sun™.

Note The use of O0 to reduce analysis time.

```
polyspace-c \ \  
-target sparc \  
-prog Apache \  
-keep-all-files \  
-allow-undef-variables \  
-continue-with-red-error \  
-O0 \  
-D PST \  
-D __GNUC_MINOR__=6 -D SOLARIS2=270 -D USE_EXPAT \  
-D NO_DL_NEEDED \  

```

```

-I sources \
-I /usr/local/pst/include.sparc \
-I /usr/include \
-results-dir RESULTS

```

cxref Example

Here is another C launch command. The compilation is for Linux®. Note the escape characters, allowing quoted strings to be used as compiler defines.

```

polyspace-c \
-OS-target linux \
-prog cxref \
-OO \
-I `pwd` \
-I sources \
-I <<PolySpace_Verifer_Installation_Path>>/include/include.linux \
-D CXREF_CPP="'"/usr/local/gcc/bin/cpp"' \
-D PAGE="'A4"' \
-results-dir RESULTS

```

T31 Example

Another simple C launcher. There are a couple of tasks and compilation is for an m68k.

```

polyspace-c \
-target m68k \
-entry-points task_callback_main,task_tcp_main,cdtask_depmain,
task_receiver \
-to pass1 \
-prog T31 \
-OO \
-results-dir `pwd`/RESULTS_31 \
-keep-all-files

```

Dishwasher1 Example

Another C example. This one is for the c-167 and has tasks protected by critical section.

```

polyspace-c \
-target c-167 \

```

```
-entry-points periodic,pst_main \  
-D PST -D const= -D water= \  
-from scratch \  
-to pass4 \  
-critical-section-begin "critical_enter:cs1" \  
-critical-section-end "critical_exit:cs1" \  
-prog dishwasher1 \  
-I `pwd`/sources \  
-O0 \  
-keep-all-files \  
-results-dir RESULTS
```

Satellite Example

A C example with tasks and critical sections.

```
polyspace-c  
-target c-167 \  
-entry-points ctask0,ctask1,ctask2,ctask3,interrupts \  
-O2 \  
-keep-all-files \  
-from scratch \  
-critical-section-begin "DisableInterrupts:sc1" \  
-critical-section-end "EnableInterrupts:sc1" \  
-ignore-constant-overflows \  
-include `pwd`/sources/options.h \  
-to pass4 \  
-prog satellite \  
-I `pwd`/sources \  
-results-dir RESULTS
```


Static Verification

What is Static Verification (p. A-2)

Describes static verification

Exhaustiveness (p. A-4)

Describes the thoroughness of static verification

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. Static Verification differs significantly from other techniques, such as run-time debugging, in that the analysis it provides is not based on a given test case or set of test cases. The dynamic properties obtained in the PolySpace analysis are true for all executions of the software.

Most Static Verification tools only provide an analysis of the complexity of the software, in a search for constructs which may be potentially dangerous.

PolySpace provides deep-level analysis identifying almost all run-time errors and possible access conflicts on global shared data.

The idea is to use an approximation of the software under analysis, using safe and representative approximations of software operations and data.

An example is given below:

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

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

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

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

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

Exhaustiveness

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

Analysis

In order to use a PolySpace tool, the code is prepared and an analysis is launched which in turn produces results for review.

Atomic

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

Atomicity

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

Batch mode

Execution of PolySpace from the command line, rather than via the launcher Graphical User Interface.

Category

One of four types of orange check: *potential bug*, *inconclusive check*, *data set issue* and *basic imprecision*

Certain error

See red error

Check

Test performed by PolySpace during analysis, colored red, orange, green or grey in the viewer

Dead Code

Code which is inaccessible at execution time under all circumstances, due to the logic of the software executed before it.

Development Process

Development process used within a company to progress through the software development lifecycle.

Green check

Check found to be confirmed as error free.

Grey code

Dead code.

Imprecision

Approximations made during PolySpace analysis, so that data values possible at execution time are represented by supersets including those values

mcpu

Micro Controller/Processor Unit

Orange warning

Check found to represent a possible error, which may be revealed on further investigation.

PolySpace Approach

The manner of use of PolySpace to achieve a particular goal, with reference to a collection of techniques and guiding principles.

Precision

An analysis which includes few inconclusive orange checks is said to be precise

Progress text

Output from PolySpace during analysis to indicate what proportion of the analysis has been completed. Could be considered as a “textual progress bar”.

Red error

Check found to represent a definite error

Review

Inspection of the results produced by a PolySpace analysis, using the Viewer.

Scaling option

Option applied when an application submitted to PolySpace Verifier proves to be bigger or more complex than is practical.

Selectivity

The ratio of (green + grey + red) / (total amount of checks)

Unreachable code

Dead code