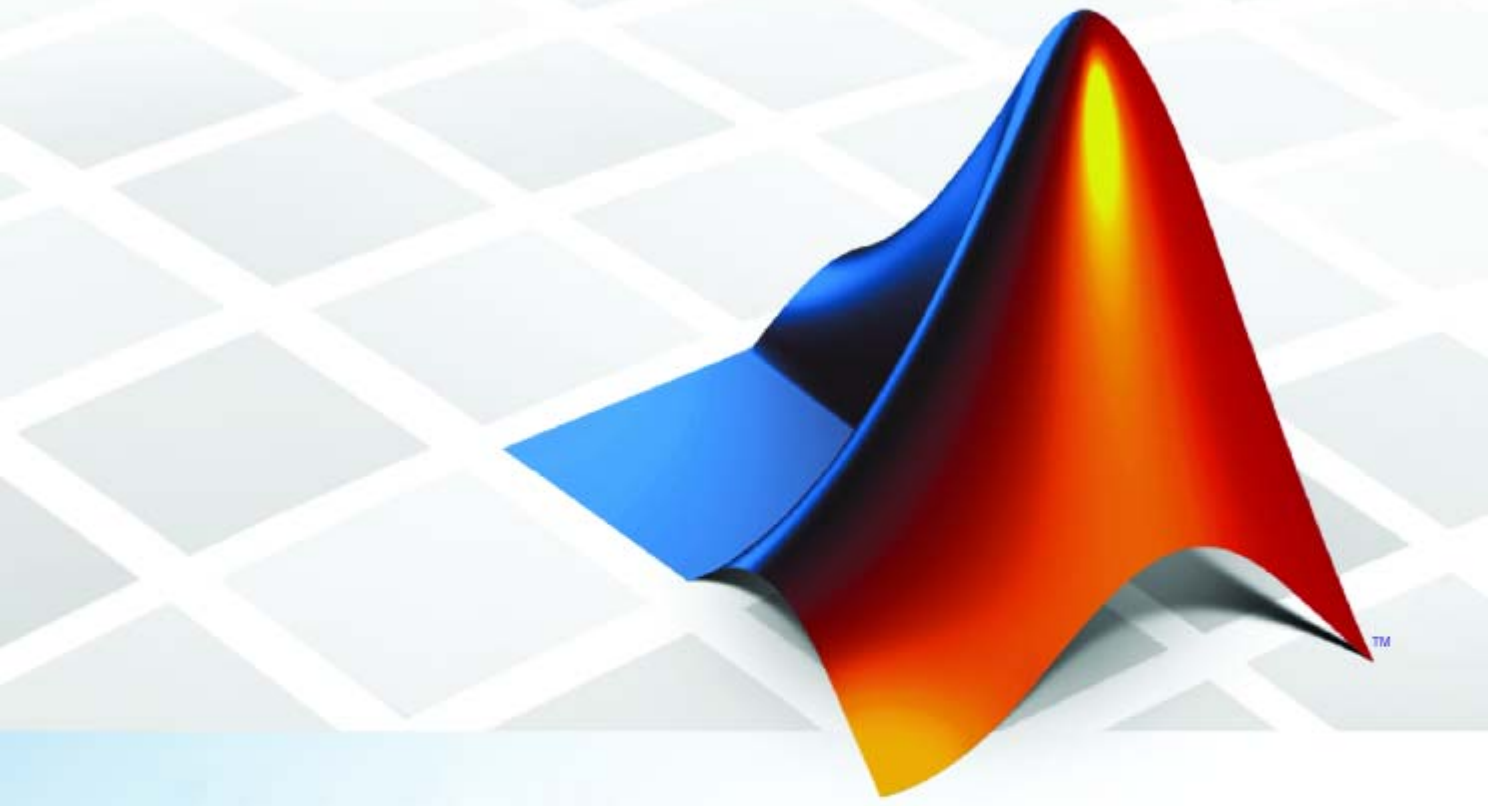
PolySpace™ Client/Server for C++ 5

User's Guide



The MathWorks™

Accelerating the pace of engineering and science

How to Contact The MathWorks

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

suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.c om	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.c om	Sales, pricing, an d 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 acquirin g 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)

Contents

1 PolySpace™ Documentation Set

About this Guide 1-2

How to Use this Guide 1-3
Analyzing One Class 1-3
Analyzing a Class from Microsoft ® Visual Studio ® 1-3
Analyzing a Class on a Server 1-3
Analyzing Code from UML Models 1-3
Detailed Contents 1-4

2 Getting Started

General Requirements 2-2
Computer Configuration 2-2
Timing Information 2-2
Installation Guide 2-2
Structure of this Document 2-3

Step 1: PolySpace™ Client – Setting Up and Launching an Analysis on a Single Class 2-5
Overview 2-5
Analysis Prerequisites 2-5
Setting Up a PolySpace™ Client Analysis 2-6
PolySpace™ Client: Running the Analysis 2-13

Step 2 : Class Analyzer 2-24
PolySpace™ Class Analyzer Overview 2-24
Log File 2-26
Behavior of Global variables and members 2-27
Methods and Classes Specificities 2-30

Step 3: PolySpace™ Viewer – Exploring Results 2-33
Overview 2-33
Modes of Operation 2-33
Download Results into the Viewer 2-35
Reviewing PolySpace™ Results in “Expert” Mode (“training.cpp”) 2-37
Methodological Assistant 2-54
Report Generation 2-61

Step 4: Launch PolySpace™ Remotely 2-65
Overview 2-65
Launching an Analysis 2-65
Managing Your Remote Analysis: the PolySpace™ Spooler 2-67
Batch Commands 2-70
Sharing Analyses Between Accounts 2-72

Summary 2-74

3 Analysis Setup

Common Compile errors 3-2
Includes 3-2
Specific Keyword or Extended Keyword 3-2
Initialization of Global Variables 3-7

Dialect Issues 3-8
ISO® versus Default Dialects 3-8
CFront2 and CFront3 Dialects 3-10
Visual Dialects 3-11
GNU Dialect 3-13

Link Messages 3-16
STL Library C++ Stubbing Errors 3-16
Lib C Stubbing Errors 3-17

Methodology Using the Pre-processed .ci Files 3-21

Overview 3-21
Example of ci File 3-21
Methodology Guide 3-23

OS and Target Specifications 3-26
List of Predefined Compilation Flags 3-26
Target Specifications 3-29
Generic/Custom Target 3-30

Intermediate Language Errors 3-32

Advanced Setup 3-34
Reduce Oranges Step by Step 3-34
Approximations Made by PolySpace 3-47
Variables 3-54
Types Promotion 3-58
Built-in Functions 3-61

4 PolySpace™ Class Analyzer Process

Analyzin g C++ Classes 4-2
Overview 4-2
Why Provi de a Class Analyzer 4-2

Types of Classes 4-3
Simple C lass 4-3
Simple Inheritance 4-5
Multip le Inheritance 4-6
Abstra ct Class 4-7
Virtu al Inheritance 4-8
Other Types 4-9

5 PolySpace™ C++ add-in for Visual Studio®

Overview 5-2

Using PolySpace™ Software Within Visual Studio® . . . 5-3
- Overview 5-3
- PolySpace™ Parameters Within Visual Studio® 5-3
- Performing a File Verification from Visual Studio® 5-7
- Analyzing Classes 5-11
- The Configuration File and Default Options 5-14

Analyzing an Entire Project 5-16

6 PolySpace™ UML Link™ RH Product

Getting Started 6-2
- Overview 6-2
- Step 1 – Opening the Example Airbag Model 6-2
- Step 2 – Starting an Analysis 6-3
- Step 3 – The Start Analysis Panel 6-4
- Step 4 – Navigating from the PolySpace™ Results to the Rhapsody® Model 6-8

PolySpace™ Panel 6-10
- Overview 6-10
- Start Button 6-11
- Stop Button 6-14
- Compilation Log Button 6-14
- Configure Button 6-14
- Manage Analyses Button 6-15
- View Results Button 6-16
- Browse for Results Button 6-16
- Help Button 6-16

Installing the Integration into an Existing Model 6-17
- Overview 6-17

Merging the PolySpace™ Code with Rhapsody ® 6-17

Other Topics 6-19
Analysis Information 6-19
Default Template CFG Options 6-19
Back to Model Viewer Link 6-20

7 Working with Results Review

Basics: Prere quisites to Reviewing PolySpace Results 7-2
Overview 7-2
Grey Follows Red 7-3
What is the Me ssage and What does it Mean? 7-4
What is the C ++ Explanation 7-5
Review Run T ime Errors: Fix Red Errors 7-7
Review Dead Code Checks: Why is Grey Code Interesti ng 7-8
How to Conc lude an Orange Review 7-10

Methodol ogy: Selective Orange Review 7-14
Overview 7-14
The Basic Principles 7-14
The Rati onale Behind the Approach 7-15
In Pract ice 7-16
Step by Step 7-16
Considering the Effects of Application Code Size 7-17

Colore d Source Code for C++ 7-19
Category of Checks 7-20
Funct ion Returns a value: FRV 7-28
Non Nu ll this-pointer: NNT 7-29
Posit ive Array Size: CPP 7-31
Incor rect typeid Argument: CPP 7-32
Incorrect dynamic_cast on Pointer: CPP 7-33
Incorrect dynamic_cast on Reference: CPP 7-35
Inva lid Pointer to Member: OOP 7-36
Call of Pure Virtual Function: OOP 7-37

Incorrect Type for this-pointer: OOP 7-39
Potential Call to: INF 7-41
Non-Initialized Variable: NIV/NIVL 7-43
Non-Initialized Pointer: NIP 7-45
User Assertion Failure: ASRT 7-45
Overflows and Underflows 7-47
Scalar or Float Division by zero: ZDV 7-53
Shift Amount is Outside its Bounds: SHF 7-54
Left Operand of Left Shift is Negative: SHF 7-55
Power Must be Positive: POW 7-56
Array Index is Outside its Bounds: OBAI 7-58
Function Pointer Must Point to a Valid Function: COR . . . 7-59
Wrong Number of Arguments: COR 7-60
Wrong Type of Argument: COR 7-61
Pointer is Outside its Bounds: IDP 7-62
logic_error is Thrown: EXC 7-70
runtime_error is Thrown: EXC 7-73
Function throws: EXC 7-74
Call to Throws: EXC 7-77
Destructor or Delete Throws: EXC 7-78
Main, Tasks or C Library Function Throws: EXC 7-80
Exception Raised is Not Specified in the Throw List: EXC 7-82
Throw During Catch Parameter Construction: EXC 7-84
Continue Execution in __except: EXC 7-86
Unreachable Code: UNR 7-88
Values on Assignment: VOA 7-89
Non Terminations: Calls and Loops 7-91

Advanced Results Review 7-97
Red Checks Where Grey Checks were Expected 7-97
Potential Side Effect of a Red Error 7-99

8 Options Description

Overview 8-2

Sources/Includes 8-3
-results-dir Results_Directory 8-3

-sources files or -sources-list-file file_name 8-3
-I directory 8-5

General 8-6
Overview 8-6
-prog Session identifier 8-6
-date Date 8-7
-author Author 8-7
-verif-version Version 8-7
-voa 8-8
-keep-all-files 8-8
-continue-with-existing-host 8-8
-allow-unsupported-linux 8-9

Targets/Compilers 8-10
-target TargetProcessorType 8-10
GENERIC ADVANCED TARGET OPTIONS 8-11
-OS-target OperatingSystemTarget 8-15
-D compiler-flag 8-16
-U compiler-flag 8-16
-include file1[,file2[,...]] 8-17
-post-preprocessing-command "command" 8-17
-post-analysis-command <file_name> or "command" 8-18

Compliance with Standards 8-20
-dos 8-20
Embedded Assembler 8-21
-wchar-t-is-unsigned-long 8-22
-size-t-is-unsigned-long 8-22
-no-extern-C 8-22
-no-stl-stubs 8-22
-dialect DialectName 8-23
-wchar-t-is 8-23
-for-loop-index-scope 8-24
Visual Specific Options 8-25
-ignore-constant-overflows 8-27
-allow-undef-variables 8-27
-allow-negative-operand-in-shift 8-28
-Wall 8-28

PolySpace Inner Settings 8-29
-main sub_program_name 8-29

Generate a Main Using a Given Class 8-30
-main-generator-calls 8-32
General options for the generation of mains 8-33
-no-automatic-stubbing 8-35
-ignore-float-rounding 8-35
-detect-unsigned-overflows 8-37
-extra-flags option-extra-flag 8-37
-cpp-extra-flags flag 8-38

Precision/Scaling 8-39
-quick 8-39
-O(0-3) 8-40
-from verification-phase 8-41
-to verification-phase 8-42
-path-sensitivity-delta number 8-43
-context-sensitivity "proc1[,proc2[,...]]" 8-44
-context-sensitivity-auto 8-44
-respect-types-in-globals 8-44
-k-limiting number 8-45
-respect-types-in-fields 8-45
-inline "proc1[,proc2[,...]]" 8-46
Tuning Precision and Scaling parameters 8-47

MultiTasking (PolySpace Server for C/C++ Only) 8-49
-entry-points str1[,str2[,...]] 8-49
-critical-section-[begin or en d] "proc1:cs1[,proc2:cs2]" 8-49
-temporal-exclusions-file file_name 8-50

Specific Batch Options 8-52
-server server_name_or_ip[:port_number] 8-52
-sources-list-file file_name 8-53
-v | -version 8-53
-h[elp] 8-53

A Static Verification

What is Static Verification A-2

Exhaustiveness A-4

Glossary

1

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 docu mentation required to use PolySpace™ Products, 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 Class” on page 1-3
“Analyzing a Class from Microsoft ® Visual Studio ®” on page 1-3
“Analyzing a Class on a Server” on page 1-3
“Analyzing Code from UML Models” on page 1-3
“Detailed Contents” on page 1-4

Analyzing One Class

- Do you want to perform your firs t analysis and results review?
- Is it possible for you to restrict data (functional) ranges in the file?
- 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™ analysis provides?

Analyzing a Class from Microsoft ® Visual Studio ®

Do you want to analyze a class comi ng from a project using Microsoft ® Visual Studio ®.NET IDE?

Analyzing a Class on a Server

Do you want to perform a class analysis taking place on a server, and do you want access the queued analysis?

Analyzing Code from UML Models

Do you want to analyze code generated from UML models using PolySpace UML Link™ RH ?

Detailed Contents

- PolySpace Installation. Please refer to the PolySpace Installation Guide and the 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 software. 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 all 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™ Class Analyzer Process” gives a strategy for analyzing C++ classes. This allows the developer to identify, and possibly remove most of the runtime errors present in a class depending of the type of the class to analyze.
- Chapter 5, “PolySpace™ C++ add-in for Visual Studio®” provides automatic source code verification and bug detection in source code developed inside the Visual IDE.
- Chapter 6, “PolySpace™ UML Link™ RH Product” – While using Collaborative Model-Driven Development, run-time errors can be caused either by design issues in the model itself or faulty hand written code. These reliability flaws can sometimes be found using code reviews and intensive testing – but these techniques are time-consuming and costly. PolySpace UML Link RH performs an exhaustive verification of the C++ code and automatically flagging flaws directly in the original Rhapsody® model, enabling engineers to fix these issues quickly and early during the design process.
- Chapter 7, “Working with 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.

- “Advanced Setup” on page 3-34 includes options description for PolySpace software, hints and tips for quicker P olySpace 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
Step 1: PolySpace™ Client – Setting Up and Launching an Analysis on a Single Class (p. 2-5)	Describes how to analyze a simple C++ class using PolySpace™ Client™ for C/C++
Step 2: Class Analyzer (p. 2-24)	Describes how to use the PolySpace™ class analyzer
Step 3: PolySpace™ Viewer – Exploring Results (p. 2-33)	Describes how to interpret the results of your analysis
Step 4: Launch PolySpace™ Remotely (p. 2-65)	Describes how to perform an analysis remotely using PolySpace ™ Server™ for C/C++
Summary (p. 2-74)	Provides a summary of the information presented in this chapter

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

Minimum hardware requirements to follow step by step this tutorial on a Windows ® PC are described in the installation guide available from the PolySpace™ installation CD-ROM (\Docs\Install \PolySpace_Install_Guide.pdf).

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.

Installation Guide

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

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

- PolySpace™ Client™ for C/C++ product – analyzing single class.

- PolySpace™ Server™ for C/C++ product – analyzing classes or composite analysis.
- PolySpace Viewer is the graphical user interface to explore the results computed by PolySpace Client or PolySpace Server.
- PolySpace Spooler is the graphical interface to manage analysis sent in remote.

Please refer to the 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:

This Getting Started chapter will focus on the following steps using PolySpace products:

- In “Step 1: PolySpace™ Client – Setting Up and Launching an Analysis on a Single Class” on page 2-5 we will analyze a simple class in “training.cpp” by using the class analyzer available in PolySpace Client for C/C++.
- In “Step 2: Class Analyzer” on page 2 -24 we will describe the capabilities of the class analyzer.

- In “Step 3: PolySpace™ Viewer – Expl oring Results” on page 2-33 we will review the results obtained during Step 1 by using PolySpace Viewer
- In the “Step 4: Launch PolySpace™ Remotely” on page 2-65, instead of performing a PolySpace analysis locall y, we will send it remotely to a server.

Step 1: PolySpace™ Client – Setting Up and Launching an Analysis on a Sin gle Class

In this section...

"Overview" on page 2-5

"Analysis Prerequisites" on page 2-5

"Setting Up a PolySpace™ Client Analysis" on page 2-6

"PolySpace™ Client: Running the Analysis" on page 2-13

Overview

This chapter de scribes a basic class analysis. It focuses on the analysis of MathUtils cla ss of "training.cpp ", which is included in the PolySpace™ installation directory and located at:

<PolySpaceIn stallDir> \Examples\Demo_Cpp_Long\sources\training.cpp .

The PolySpace analysis process is composed of three main phases:

1 First, PolySp ace checks the syntax and semantic of the analyzed file(s). Howe ver, as PolySpace is not associated to a particular compiler, benefits of this phase are triple for th e analyzed source code: ANSI C++ compliance , portability and maintainability .

2 Then, PolySpace seeks the main procedure. If none is found, PolySpace Client for C/C++ will generate one automatically. By default, the main will build an instance of the class using the constructor and call all its public and protected function methods.

3 Finally, PolySpace 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:

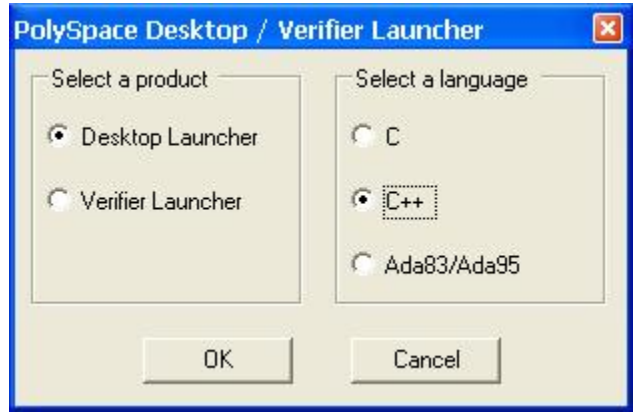
- PolySpace Client For C/C++ product an d its related license file correctly installed;

- Source code files (in this case “training.cpp”) and all header files that it may directly or indirectly include. For this tutorial we will see later that we need three header files, “training.h”, “zz_utils.h” and “math.h” in order to analyze the class MathUtilsin “training.cpp”.
- All “-D” compilation switches necessary to compile the file are known. Please note that in this tutorial, no “-D” is necessary to compile “training.cpp”.

Setting Up a PolySpace™ Client Analysis

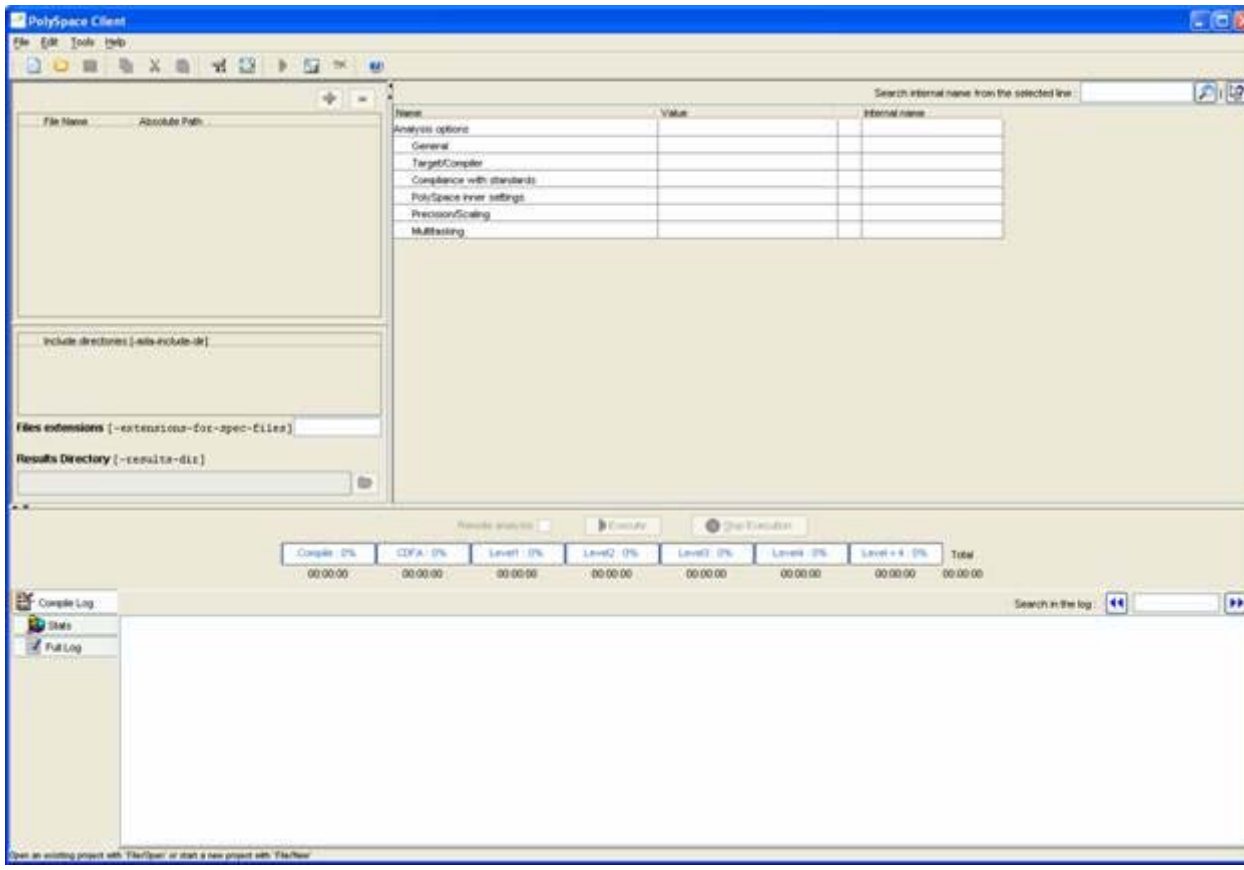
1 Double-click on the PolySpace Launcher icon:

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 Desktop Launcher, and C++, then click OK.

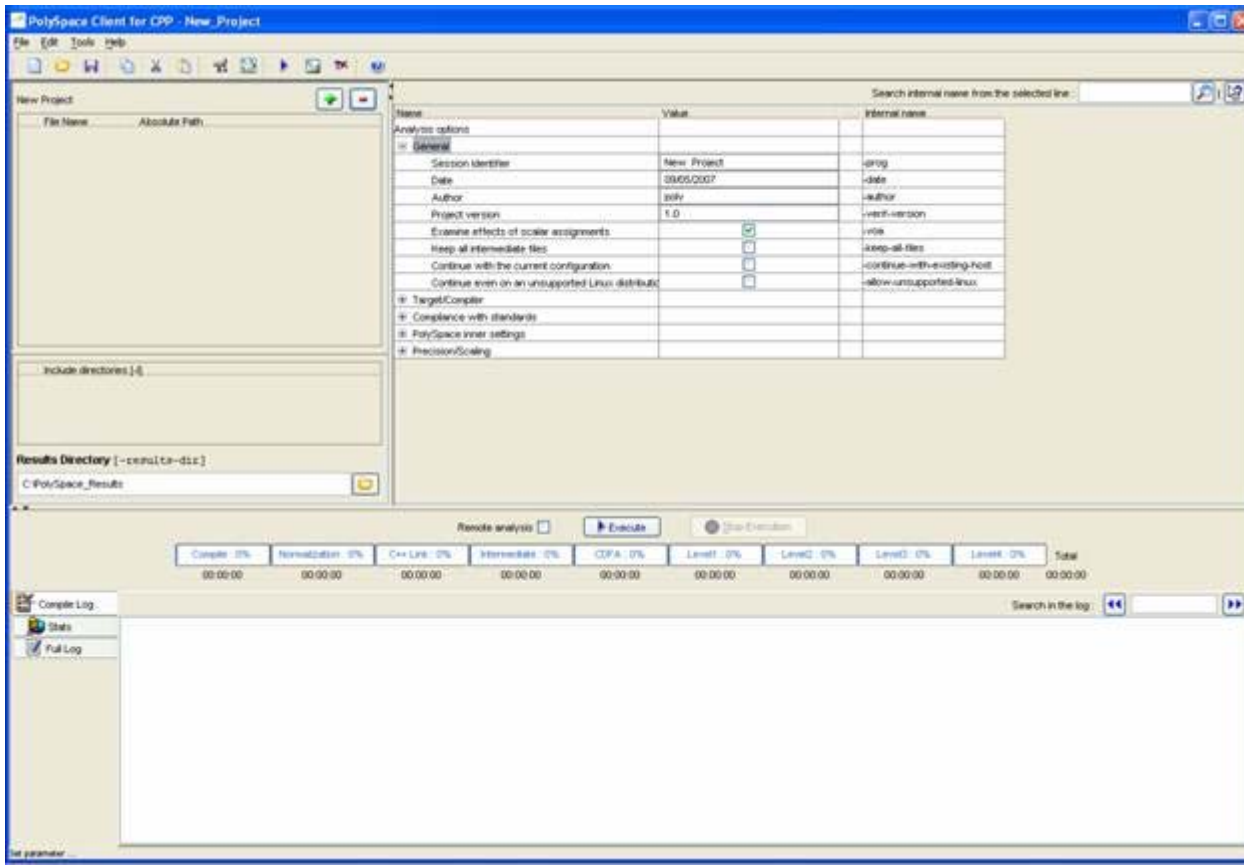
The Graphical Interface of PolySpace analysis Launcher is displayed as below:



3 Select File > New Project to start an analysis:

The PolySpace Client New Project window ope ns. It contains four sections:

- At the very to p, the title bar, which contains usual icons and menus;
- Top left is th e list of files to analyze, along with include and results directories ;
- Top right is t he set of options associated with the analysis that will be processed;
- Finally the b ottom area allows following the execution and progress of the analy sis.



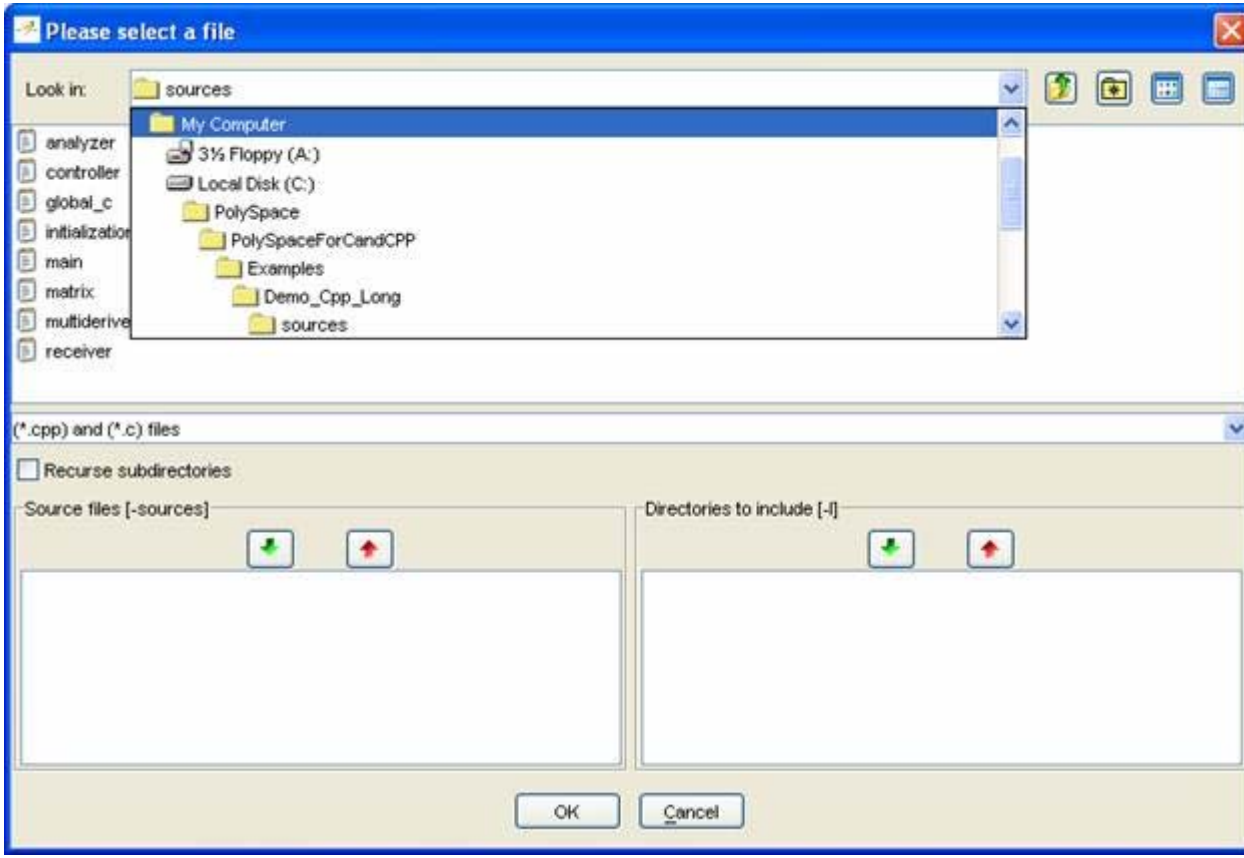
Note it is also possible to drag a direct ory or source files and drop it them directly in the “File Name/Absolute Path” part (top left of PolySpace Client) without using the “Please select a file” window.

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

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

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

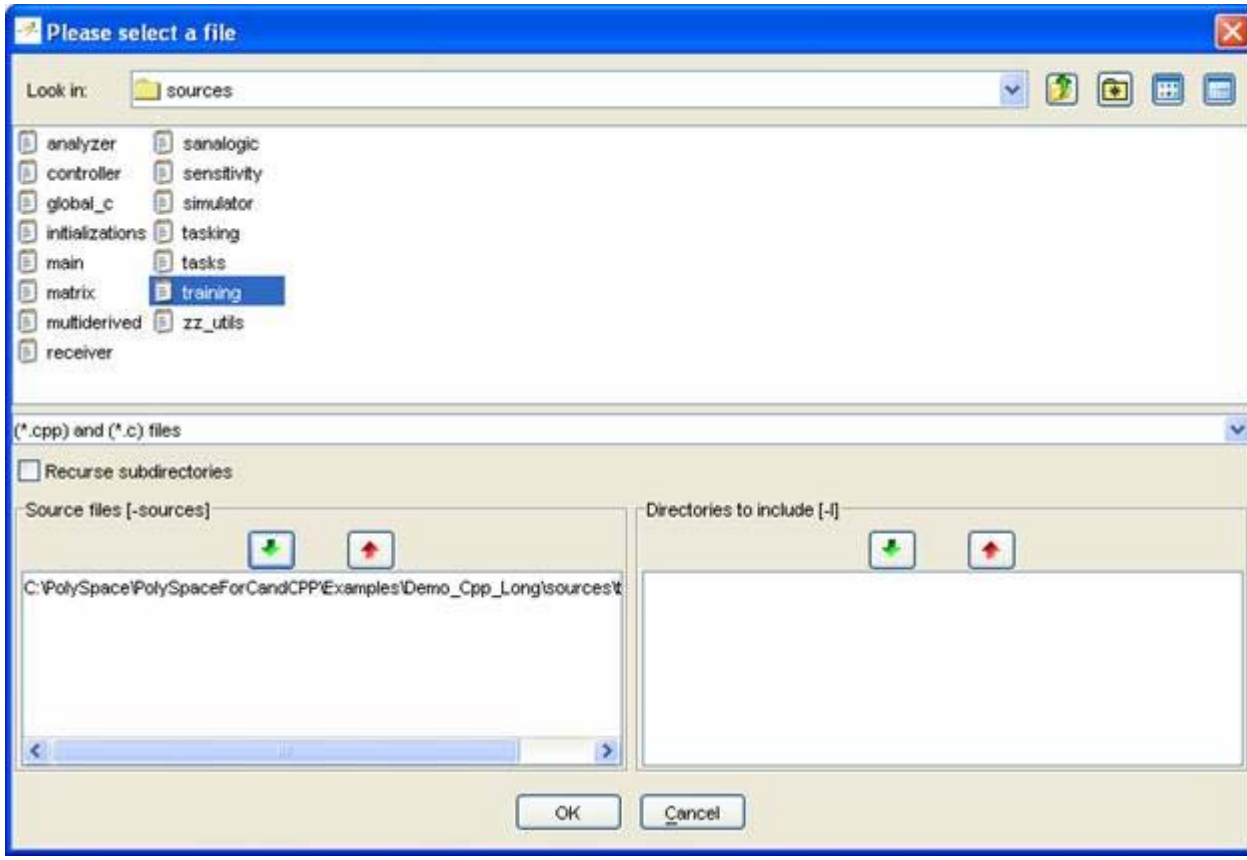
It opens the “Please select a file” window, from which you can select one or several files to analyze.



6 In the “Look in” section, click on , and select “<PolySpaceInstallDir>\Examples\Demo_Cpp_Long\sources”. A list

of files appears in the box (<PolySpaceInstallDir> corresponds to C:\PolySpace\PolySpaceForCandCPP in the figure above).

7 Select “training.cpp” 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 CPP - New_Project” window.

Note it is also possible to drag a direct ory 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.

9 Now, click on

and expand the “PolySpace inner settings” group.

10 Check the box in the “Generate a main” column that is associated to the “-main-generator” line as shown below. It enables the “-class-analyzer” option allowing to give the name of the class to analyze (see also step 2). For the ne eded of this tutorial, please type “MathUtils” in the column at the centr e as shown in the figure below. When the class is surrounded by a name space, use the standard C++ syntax <namespace>::<classname>.

Name	Value	Internal name
Analysis options		
⊟ General		
Session identifier	New Project	-prog
Date	10/05/2006	-date
Author	bard	-author
Project version	1.0	-verif-version
Examine effects of scalar assignments	☑	-voa
Keep all intermediate files	☐	-keep-all-files
Continue with the current configuration	☐	-continue-with-existing-host
Continue even on an unsupported Linux distribution	☐	-allow-unsupported-linux
⊞ Target/Compiler		
⊞ Compliance with standards		
⊟ PolySpace inner settings		
⊞ Specify a Visual kind of main	☐	
⊟ Generate a main using a given class	☑	
Class name	MathUtils	-class-analyzer
Analyze only the given class	☐	-class-only
Functions called by the generated main	default	-class-analyzer-calls
⊞ Generate a main using given functions	☐	
⊞ Stubbing		
⊞ Assumptions		
⊞ Others		
⊞ Precision/Scaling		
⊞ Multitasking		

11 It is also recommended to select the -voa option. This option allows you to give some information on each scalar assignment with possible range of values. It can help understanding PolySpace messages.

Note When you want to analyze classes alone, the -class-only option can been checked. It means that even if you add any other classes and function members definitions, PolySpace will stubs them. This option accelerates analysis and allows to check robustness issues only on the class. For the need of this tutorial, it is not necessary to check this option: the “MathUtils” class does not depend on other classes.

PolySpace™ Client: Running the Analysis

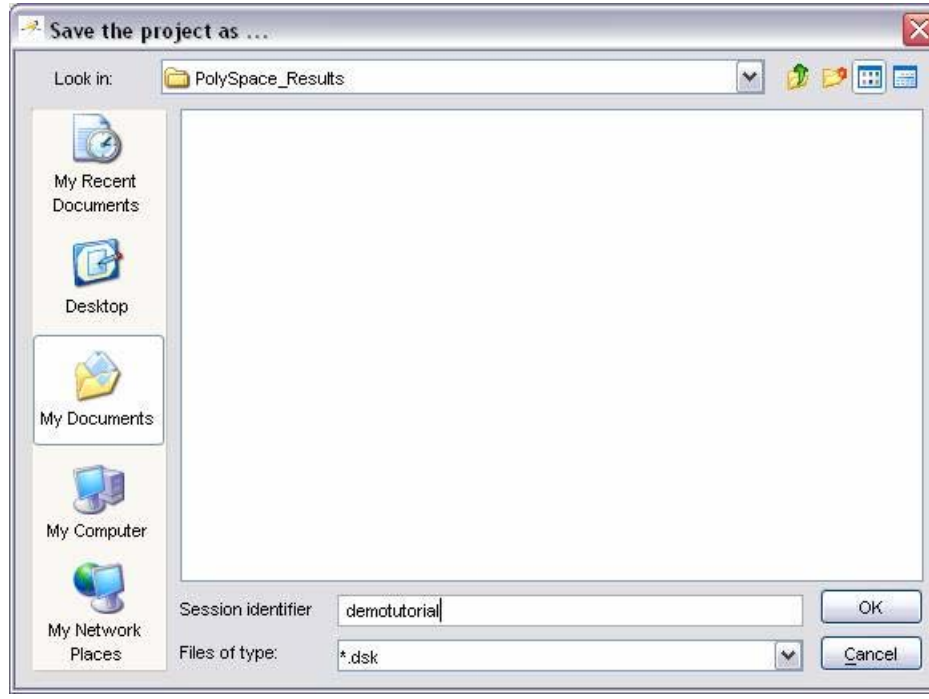
- “Starting the Analysis” on page 2-13
- “Parsing Errors During Preliminary Analysis Stages” on page 2-15
- “Progression of the Analysis” on page 2-21
- “End of the Analysis” on page 2-23

Starting the Analysis

To run the analysis:

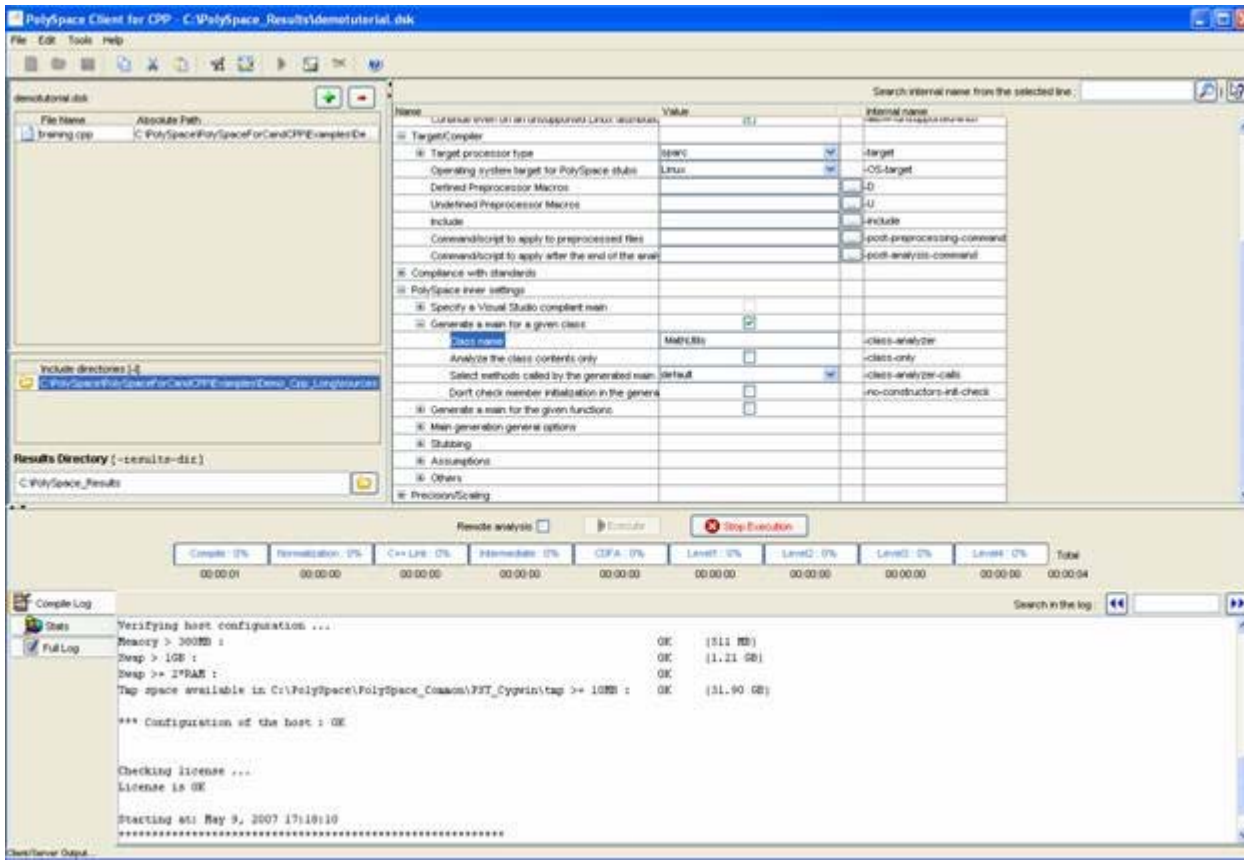
1 Click on to start the analysis. Alternatively, you can click on the button in the title bar to run PolySpace Client with the current setting.

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 “demotutorial” and save it under PolySpace result directory. The full name of that file will be “demotutorial.dsk”.



2 Click OK to go back to the “PolySpace Client for CPP - New_Project” window and click again on

to proceed.

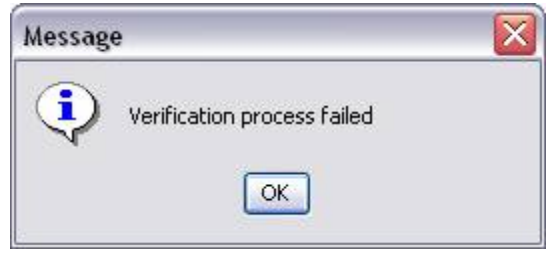


A progress report is displayed in the bottom part of the graphical interface, indicating that the analysis is being performed. The Execute button is also grayed out.

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

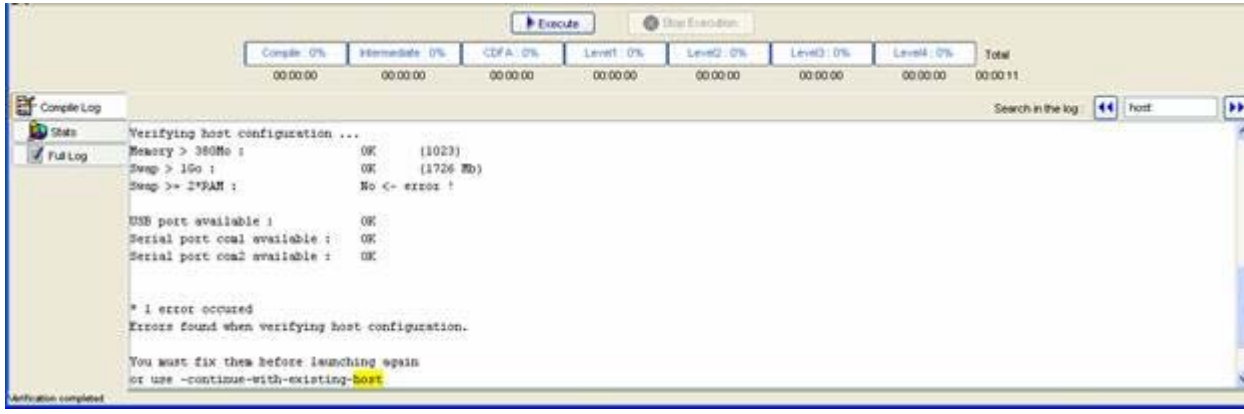
Parsi ng Errors During Preliminary Analysis Stages

After some checks, PolySpace 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 r equirements 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 hardwar e recommendation problem.

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

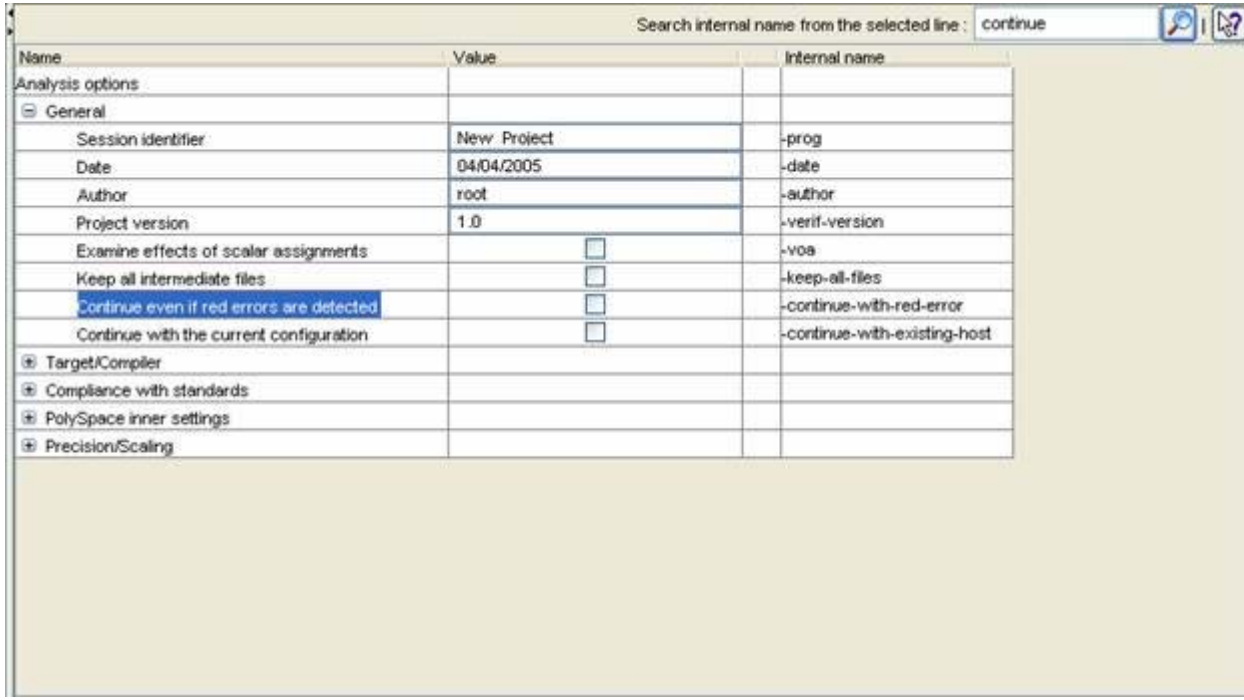
- upgrade your computer to meet the minimal requirements, or

- use the -continue-with-existing-host option which overrides the initial check for minimal hardware co nfiguration. 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 f rom the selected line (top right box).

3 Click .

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



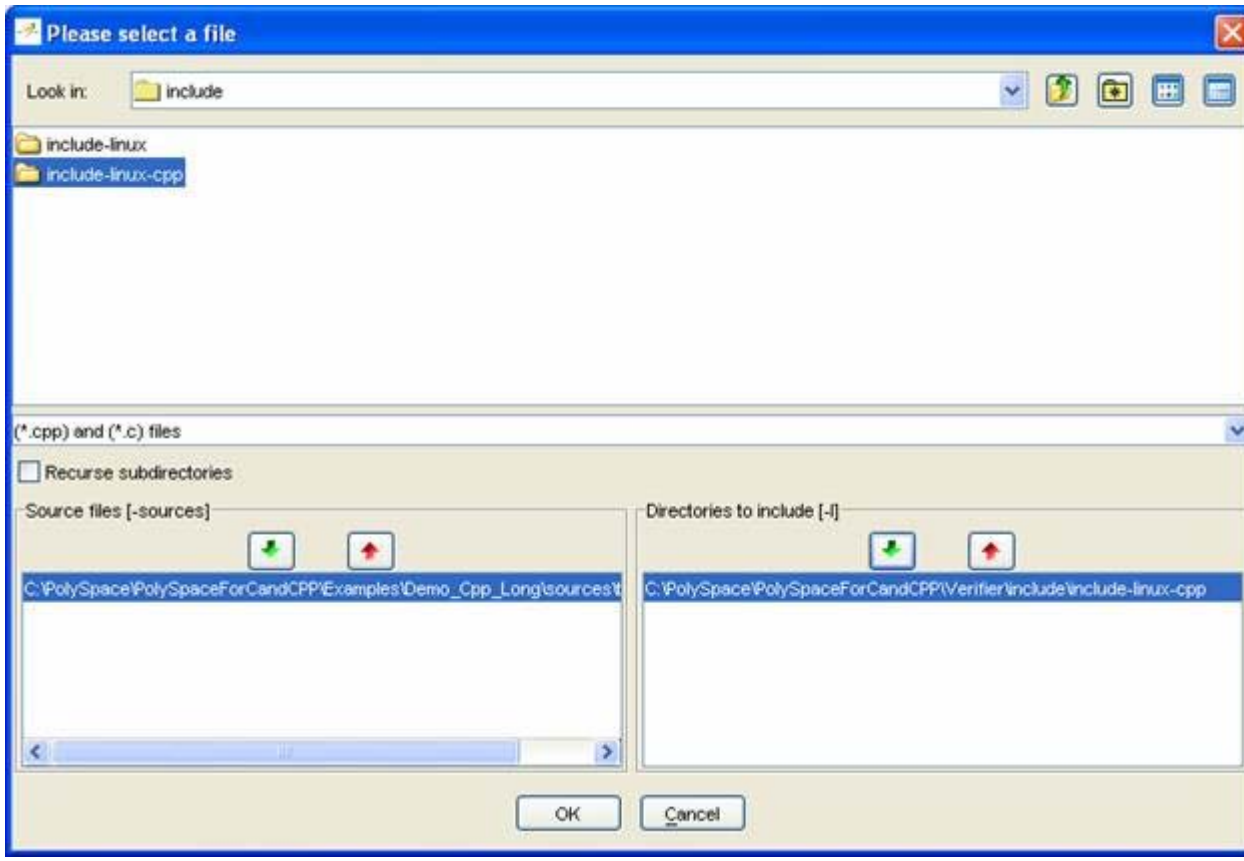
4 Check the box in the “Value” column that is associated to the “-conti nue-with-existing-host” line as shown below.

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 tuto rial, as shown above, the file named “math.h” can not be found. To fix thi s problem, you need to indicate its location. As PolySpace is not associated with one particular compiler, it is mandatory to indicate where library f iles are stored.

In our “tr aining.cpp” file analysis, the related “math.h” file is one of inclu des distributed with PolySpace C++ product located in <PolySpaceInstallDir>\include\include-linux-cpp. This distribution concerns a Linux O S target and is only given as material of help. For analyzing your code, it is recommended to indicate the path to the standard headers dedicated to your own compiler.

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



2 Select “<PolySpaceInstallDir>\Verifie r\include\include-linux-cpp”, where an exemplary of <“math.h”> is located for the Linux OS target.

3 Click on in the “Directories to include [-I]” section.

Select “<PolySpaceInstallDir>\Examp les\Demo_Cpp_Long\sources”, where “training.h” is located.

4 Click on in the “Directories to include [-I]” section, then close the window using OK .

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

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

In this tutorial, as we have chosen includes of the OS Linux distribution, we have to select a Linux OS target. It defines a set of predefined compilation flags, known to be default or implicit compile options from cross-compiler for these platforms:

5 To set up the -OS-target Linux option, enter “OS-target” in the Search internal name from the selected line — top right box.

6 Then click on . It will show all options containing “OS-target” in the set of options part below:

Name	Value		Internal name
Analysis options			
⊟ General			
Session identifier	New Project		-prog
Date	10/05/2006		-date
Author	bard		-author
Project version	1.0		-verif-version
Examine effects of scalar assignments	☑		-voa
Keep all intermediate files	☐		-keep-all-files
Continue with the current configuration	☐		-continue-with-existing-host
Continue even on an unsupported Linux distribution	☐		-allow-unsupported-linux
⊟ Target/Compiler			
Target processor type	sparc		-target
Operating system target for PolySpace stubs	Linux		-OS-target
Defined Preprocessor Macros		...	-D
Undefined Preprocessor Macros		...	-U
Include		...	-include
Command/script to apply to preprocessed files		...	-post-preprocessing-command
⊞ Compliance with standards			
⊟ PolySpace inner settings			
⊞ Specify a Visual kind of main	☐		
⊟ Generate a main using a given class	☑		
Class name	MathUtils		-class-analyzer
Analyze only the given class	☐		-class-only
Functions called by the generated main	default		-class-analyzer-calls
⊞ Generate a main using given functions	☐		
⊞ Stubbing			
⊞ Assumptions			
⊞ Others			
⊞ Precision/Scaling			
⊞ Multitasking			

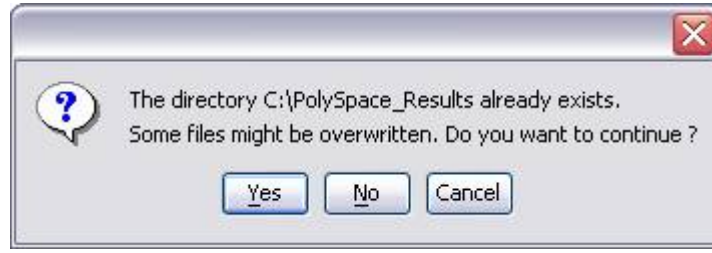
7 Then, click on the , allowing to chose linux OS target out of some predefined Operating System targets in the list solaris, linux, vxworks, no-predefined-OS and visual.

Note Associated to chosen Operating System (except no-predefined-OS), PolySpace dedicates a set of accurate stubs concerning standard templates and C libraries.

Progression of the Analysis

1 Click on to restart the analysis.

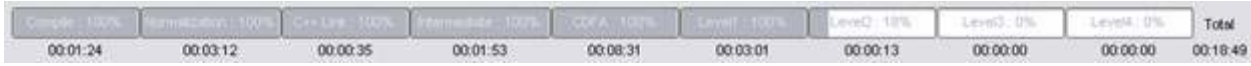
If you previously clicked Execute , some results may have already been written in the “C:\PolySpace_Results” directory. Therefore a window opens to check whether you want to overwrite in this directory or not:



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 Stop Execution (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 u pdates the analysis with its current status.

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



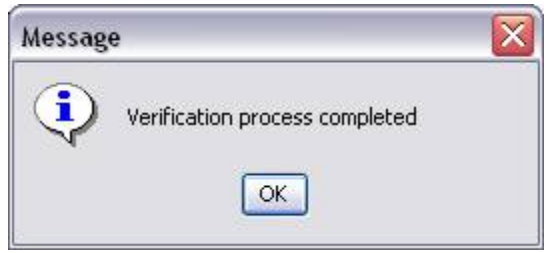
3 To obtain a progress report, click on Compile Log for the compilation phase, or Full Log for the full analysis in the low level window.

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, PolySpace 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.

Step 2: Class Analyzer

In this section...

"PolySpace™ Class Analyzer Overview" on page 2-24

"Log File" on page 2-26

"Behavior of Global variables and members" on page 2-27

"Methods and Classes Specificities" on page 2-30

PolySpace™ Class Analyzer Overview

PolySpace Class Analyzer analyses applications class by class, even if theses classes are only partially developed.

Benefits : detecting errors at a very early stage, even if the class is not fully developed, without any test cases to w rite. The process is very simple: give the class name (see step 1), and the PolySpace Class Analyzer will analyze its robustness:

- PolySpace will generate a "pseudo" main;
- It will call each constructor of the class;
- Then it will call each public function from the constructors;
- Each parameter will be initialized with full range (i.e. with a random value);
- External variables are also defined to random value.

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

As a result, a class will be analyzed by exploring every branch of the methods through all its constructors (see som e restrictions in the next section).

Sources to be Analyzed

The functions that are analyzed include all non-inherited constructors and destructors, and all non-inherited pub lic or protected methods of the class. However, sources can also come from i nherited classes (fathers) or be the sources of other classes used by the class that is being analyzed (friend, etc.).

Architecture of the Generated main

PolySpace generates the call to each con structor and method of the class. Each method will be analyzed with all constru ctors. Each parameter is initialized to random. However, even if you can have an idea of the architecture of the generated main in PolySpace Viewer, t he main is not real. You can not reuse and compile it with your analysis or PolySpace.

If we come back to the class “MathUtils”, analyzed in the first step, it contains one constructor, a destructor and sev en public methods. The architecture of the generated main is as follows:

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

Note An ASCII file representing the “pseudo” main can be seen in C:\PolySpace_Results\ALL\SRC__polyspace_main.cpp

If the class contains more than one con structor, they are called before the while statement in an if then else stat ement. From a PolySpace point of view, this architecture ensures that the anal ysis will evaluate each function method with every constructor.

Log File

- “Log File Overvi ew” on page 2-26
- “Characteristi cs of a Class and Messages of the Log File” on page 2-27

Log File Overview

When analyzing a class, the list of methods used for the main is also given in the log file duri ng the normalization phase of the C++ analysis.

You can have the d etails of what will be analyzed in the log. Here is the example concerning the MathUtils class and associated log file which can be found at root of the C:\PolySpace_Results:

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

It may happen that a m ain is already defined in the files you are analyzing. In this case, no oth er main will be generated, and this one will be analyzed. You will receive th is warning:

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

* Warning: a main procedure already exists.
* No main will be generated: the existing one will be used
```

Note The main will be analyzed even if it does not concern the class given to the -class-analyzer option.

Characteristics of a Class and Messages of the Log File

The log file may contain some error messages concerning the class to analyze. Theses messages appear when characteristics of class are not respected:

- It is not possible to analyze a class which does not exist in the given sources. The analysis will stop with the following message:

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

- It is not possible to analyze a class which only contains declarations without code. The analysis will stop with the following message:

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

Behavior of Global variables and members

- “Global Variables” on page 2-28
- “Data Members of Other Classes” on page 2-29

Global Variables

In a class analysis, global variables are not considered as following ANSI Standard anymore. if they are defined and but not initialized. Remember that ANSI Standard considers, by default, that global variables are initialized to zero.

In a class analysis, global variables do not follow standard behavior:

- Defined variables: they are initialized to random. Then they follow the data flow of the code to analyze.
- Initialized variables: they are used with the initialized value. Then they follow the data flow of the code to analyze.
- External variables: the analysis will stop. To continue the analysis, it is mandatory to use the -allow-undef-variable option. In doing so, external variables follow the behavior of a defined variable.

An example below shows behavior of two global variables:

```
1
2  extern  int  fround(float  fx);
3
4  //  global  variables
5  int  globvar1;
6  int  globvar2  =  100;
7
8  class  Location
9  {
10  private:
11    void  calculate_new(void);
12    int  x;
13
14  public:
15    //  constructor  1
16    Location(int  intx  =  0)  {  x  =  intx;  };
17    //  constructor  2
18    Location(float  fx)  {  x  =  fround(fx);  };
19
20    void  setx(int  intx)  {  x  =  intx;  calculate_new();  };
21    void  fsetx(float  fx)  {
```

```
22      int tx = fround(fx);
23      if (tx / globvar1 != 0) // ZDV check is orange
24      {
25        tx = tx / globvar2; // ZDV check is green
26        setx(tx);
27      }
28    };
29  };
```

In this example, globavar1 is defined but not initialized (see line 5): the check ZDV is orange at line 23. On the other hand, globvar2 is initialized to 100 (see line 6). The ZDV check is green at line 25.

Data Members of Other Classes

When analyzing a specific class, variable members of other classes, even members of parent classes, are considered as initialized. They follow the following behavior:

1 They are considered as may be not initialized (unproven check NIV), if constructor of the class is not defined. So they are assigned to full range and then, they follow the data flow of the code to analyze.

2 They are considered as initialized to th e value defined in the constructor, if the constructor of the class is defined in the class and given to the analysis. If -class-only option is used, it just like definition of constructor is missing (see item 1). Then they follow the data flow of the code to analyze.

3 They could be checked as run-time erro r, if and only if, the constructor is defined and does not initialize the considered member.

An example below shows the result of th e analysis of the class MyClass. It shows behavior of a variable member of the class OtherClass given without definition of its constructor:

```
class  OtherClass
{
protected:
  int  x;
  OtherClass  (int  intx);          // code  is  missing
public:
```

```
  int getMember(void) {return x;};   // NIV is warning
};
class MyClass
{
  OtherClass m_loc;
public:
  MyClass(int intx) : m_loc(0) {};
  void show(void) {
    int wx, wl;
    wx = m_loc.getMember();
    wl = wx*wx + 2;     // Possible overflows because OtherClass
                // member is assigned to full range
};
};
```

In the example above, variable member of OtherClass is checked initialized to random: the check is orange at line 7 and there are possible overflows at line 17 because range of the return value wx is full range in the type definition.

Methods and Classes Specificities

- “Template” on page 2-30
- “Abstract Classes” on page 2-31
- “Static Classes” on page 2-31
- “Inherited Classes” on page 2-31

Template

A template class can be not analyzed alone. Only instance of a template will be considered as the class that can be analyzed with the PolySpace Class Analyzer.

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

In the example above, we want to analyze template class A with two class parameters T and Z. For that, we have to define a “typedef” to create a specialization of the template, with a specific specialization for T and Z. In the example below, T represents a int and Z a double:

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

my_tempate is used as parameter of -cl ass-analyzer opti on, to analyze the this instance of template A.

Abstract Classes

In the real world an instance of an abstract class can not be created, so it can not be analyzed. However, it is easy to analyze by “removing” the pure declarations. For example, in an abstract class definition change:

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

If an abstract class is given to analyze, the class analyzer will make the change automatically and the virtual pure function (in the example above abstract_func) will then be ignored in the analysis of the abstract class.

This means that no call will be made from the generated main, so function is purely ignored. Moreover, if the function is called by another one, the pure virtual function will be stubbed and an orange check will be put on the call: “call of virtual function [f] may be pure”.

Static Classes

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

Inherited Classes

When a function is not defined in a derived class, even if it is visible because inherited from a fathers class, it is not called in the generated class. In the example below, the class Point derives from the class Location:

```
class Location
{
protected:
  int x;
  int y;
  Location (int intx, int inty);
public:
```

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

Since the two methods Location::getx and Location::gety are “visible” for derivated classes, the generated main does not include theses methods when analyzing the class Point. No matter because, we have to analyze the Location class

However, inherited members are con sidered as volatile if there are not explicitly initialized in the fathers con structors. In the example above, the use of the two members Location::x and Locat ion::y will be considered as volatile. Indeed, if we analyze the above example in the current state, the method Location:: Locati on(constructor) will be stubbed.

Step 3: PolySpace™ Viewer – Exploring Results

In this section...

“Overview” on page 2-33

“Modes of Operation” on page 2-33

“Download Results into the Viewer” on page 2-35

“Reviewing Poly Space™ Results in “Expert” Mode (“training.cpp”)” on page 2-37

“Methodologica l Assistant” on page 2-54

“Report Genera tion” on page 2-61

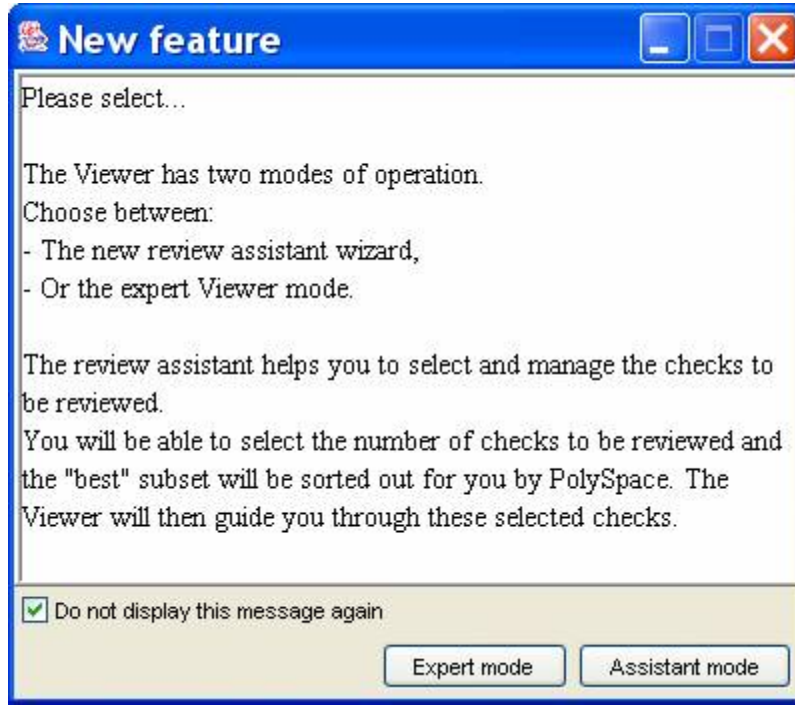
Overview

This step illustrates how to explore analysis results that were generated by either PolySpace Client or PolySpace Server. We review the results of the analysis of “training.cpp” performed during “Step 1: PolySpace™ Client – Setting Up and Launching an Analysis on a Single Class” on page 2-5.

If you clicked OK at the “End of the Analysis” on page 2-23 in Step 1, 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 pro cess of checks highlighted during an analysis:

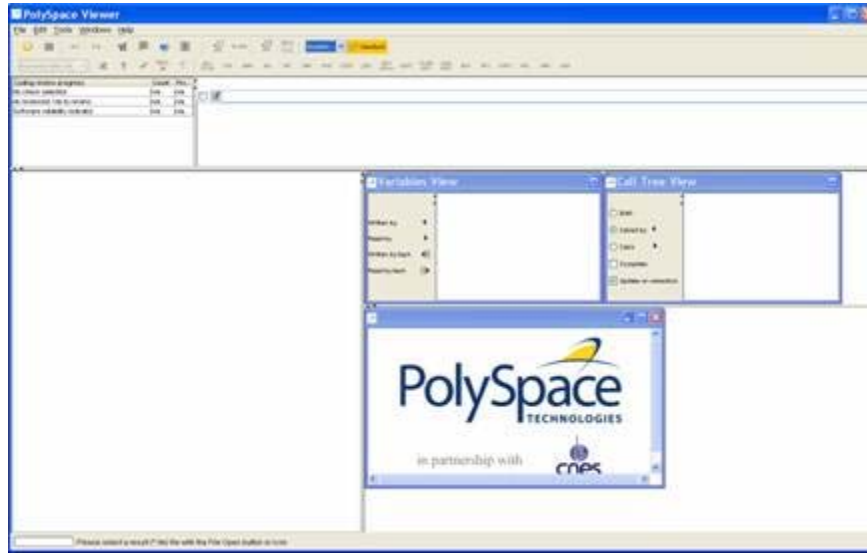
- Expert mode —The Viewer is opened in a mode where all checks can be seen. The number, the order and the ca tegories of checks to be reviewed have to be selected by the user himself (See next section).
- Assistant mode — The reviewing rules for a C++ analysis results follows a methodology selected by PolySpac e. 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 unt ick “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:

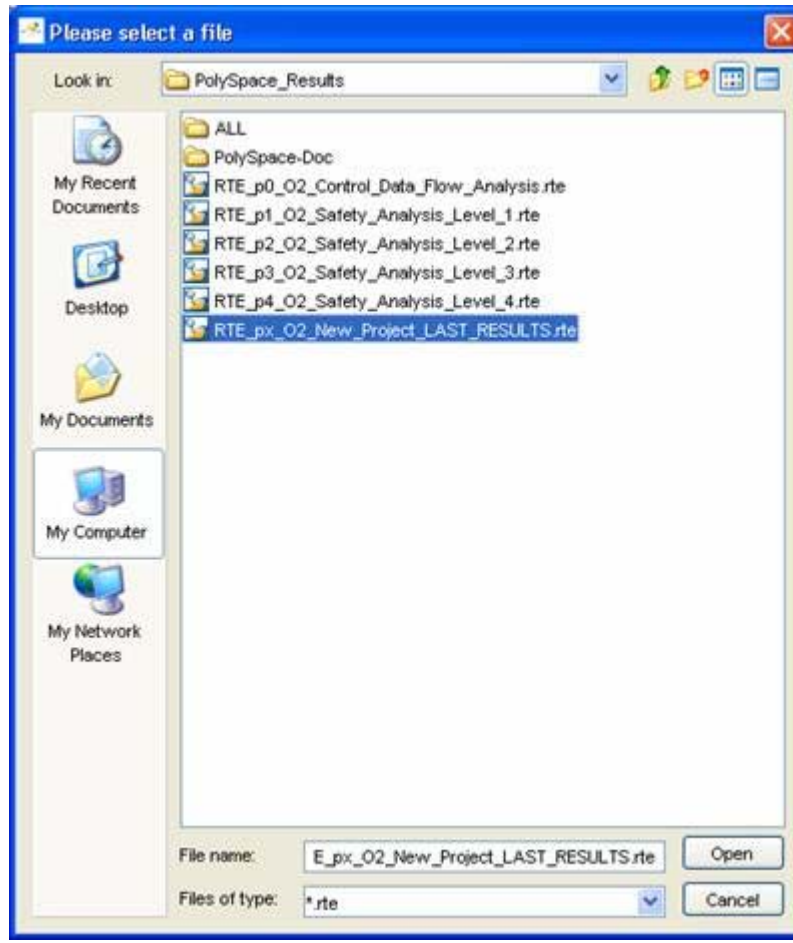


1 Click File > Open to load result files.

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

<Install Directory> \Examples\Demo_CPP_Long\ RTE_px_O2_Demo_Cpp_Long_LAST_RESULTS.rte . We will focus on “training.cpp” Procedural entity.

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



3 Click Open to proceed with further steps

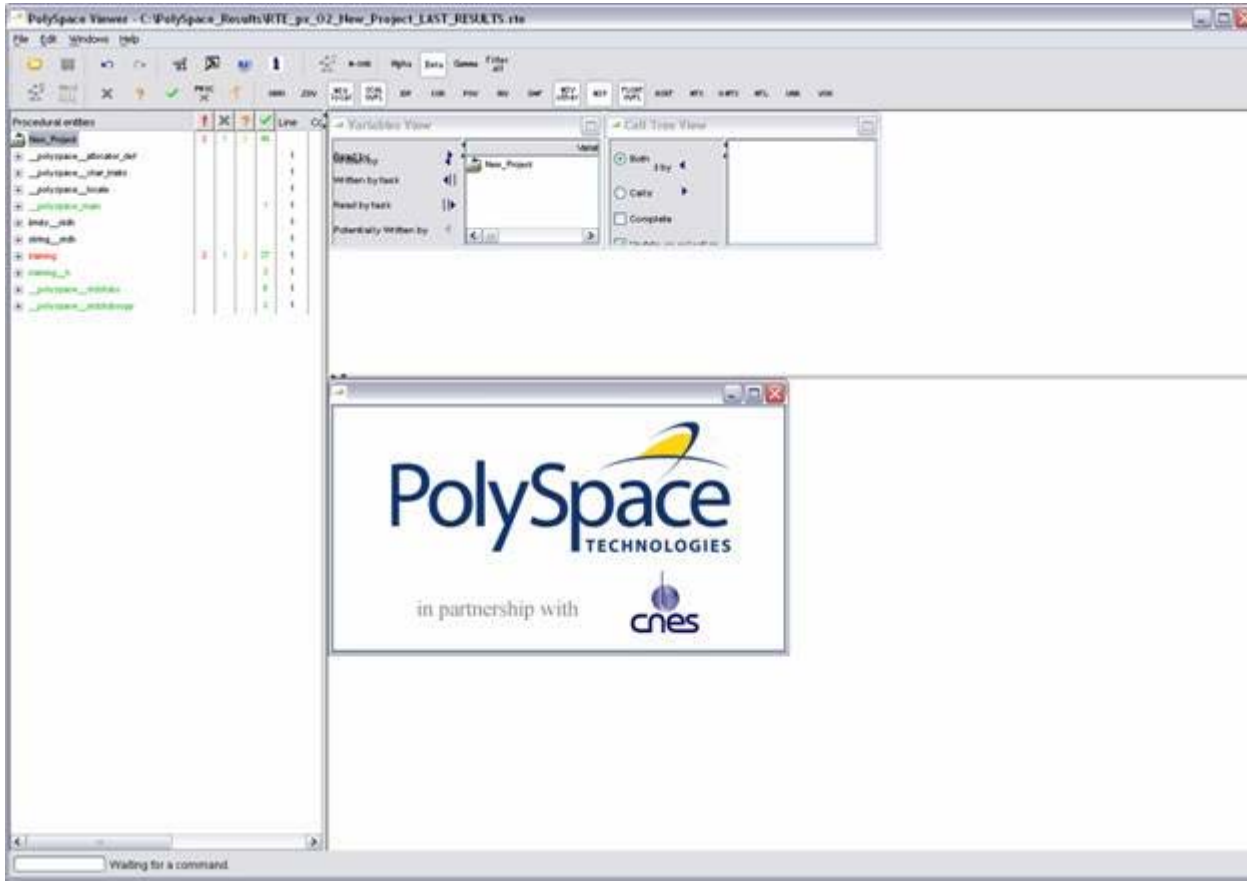
Note The RTE_px_O2_<Project Name>_LAST_RESULTS.rte is a sort of "link" on the best analysis in term of precision. This analysis is represented by RTE_p4_O2_Safety_Analysis_Level4 .rte file. Lower level files represent lower precision analysis.

Reviewing PolySpace™ Results in “Expert” Mode (“training.cpp”)

- “Overview of Expert Mode” on page 2-37
- “Procedural Entities View (RTE View)” on page 2-39
- “Colors in the Source Code View” on page 2-45
- “More Examples of Run-Time Errors” on page 2-46
- “Advanced Results Exploration” on page 2-49
- “C++ specific checks” on page 2-53
- “Miscellaneous” on page 2-53

Overview of Expert Mode

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

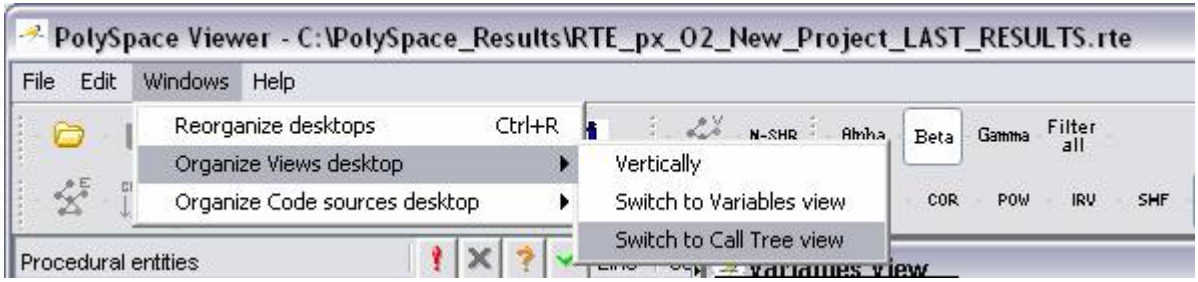


- On the left is the Procedural entities view (or RTE view). It displays the list of packages which have been ana lyzed or used during the analysis (specifications).
- In the bottom right area is the source co de view with colored instructions. Each operation checked is displayed using meaningful color scheme and related diagnostic:
  - Red — Errors which occur at every execution.
  - Orange — Warning - an error may occurs 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 displayi ng 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 colorized according t o the most critical error found:

- exception.stdh — This file contains no check. This file contains stubs of the <exception> template part of the s tandard stl library. This template stubs is an accurate representation of the initial template provided by PolySpace. All templates of standard library have been stubbed to speed up analyses.

- new.stdh — This file contains no check. This file contains implemented stubs of <new> part of stl library template.

  __polyspace_main.cpp — This file contains the main which was automatically generated . All checks there are green: no run-time error (or RTE) has been found. Please note that the pseudo code in this file is only

here to give information about the generated main. It must not be analyzed with PolySpace.

training.cpp — This file is red. This is the famous “training.cpp” containing the analyzed Class “MathUtils”. One or more definite run-time errors have been found in it.

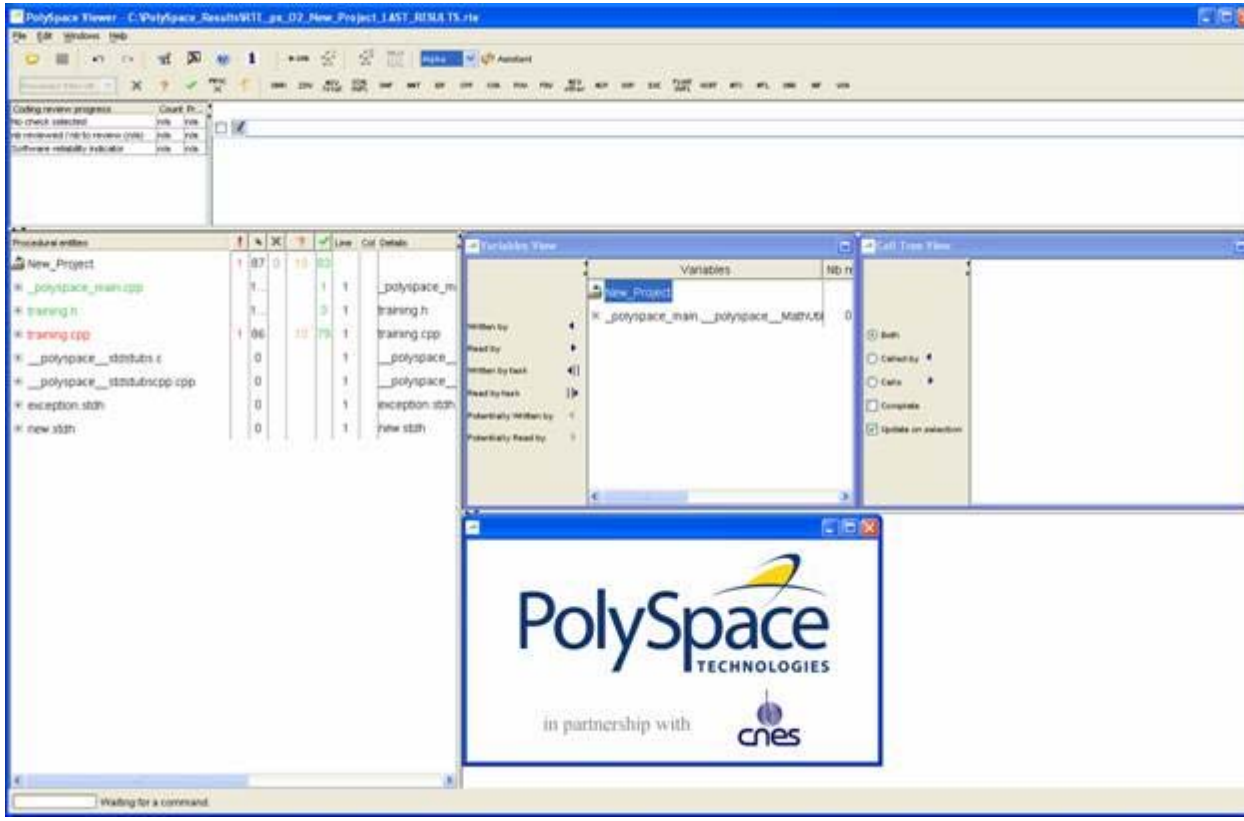
training.h — This file is the famous “training.h”, locale header included in “training.cpp”. All checks are green: no run-time error (or RTE) has been found.

__polyspace_stdstubs.c — It contains stubs of standard functions part of libc library used in training.cpp. This file contains no check.

__polyspace__stdstubscpp.cpp — It contains stubs of some standard functions part of the stl library used in training.cpp. This file contains no check.

Click once on the left of “training.cpp” to find out more about this file.

“training.cpp” is expanded and the list of function members defined within “MathUtils” of “training.cpp” is displayed. The function members in red or grey have code sections that need to be inspected (MathUtils::Pointer_Arithmetic(), MathUtils::Recursion_caller(), etc.) first because they are definite diagnosis of PolySpace (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.

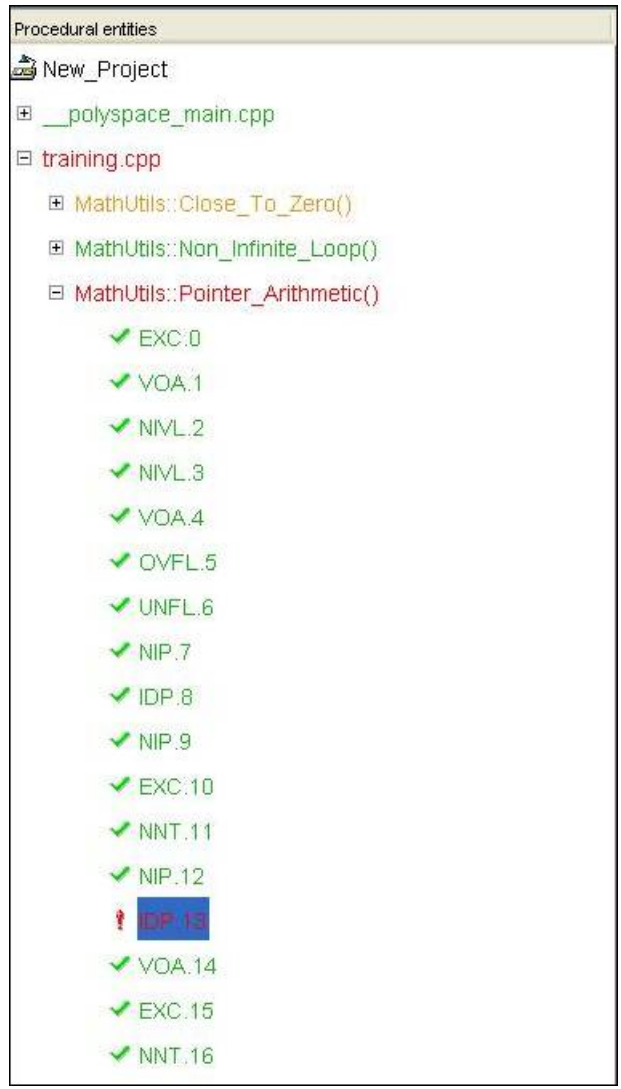
Column	Indicates
	Reliability of the code (level of proof).
	Number of definite run- time errors or reds.

Column	Indicates
	Number of warnings or oranges (that may hide run-time errors that do not occur sys tematically).
	Number of safe operations or greens.
	Number of unreachable instructions or grey code sections.

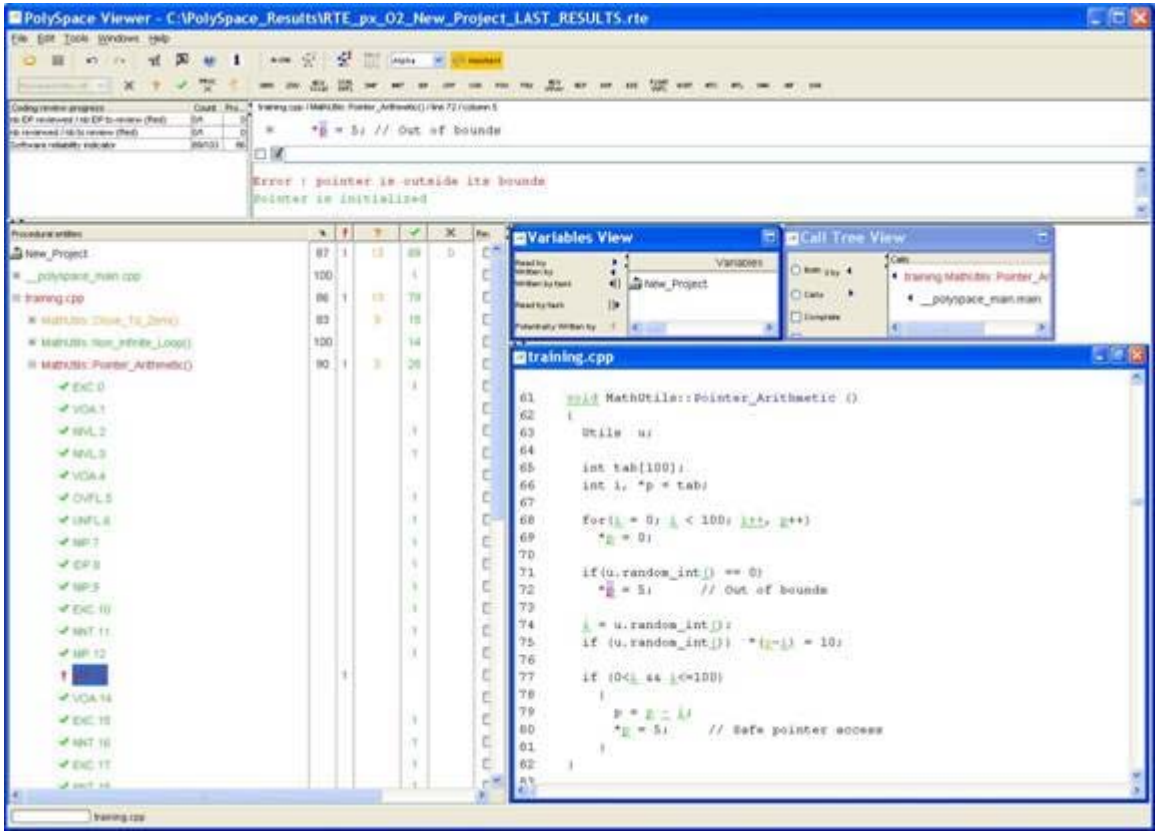
Lets have a look at some errors found by PolySpace in “training.cpp” .

First Example of Runtime Error Found by PolySpace: Memory Corruption.

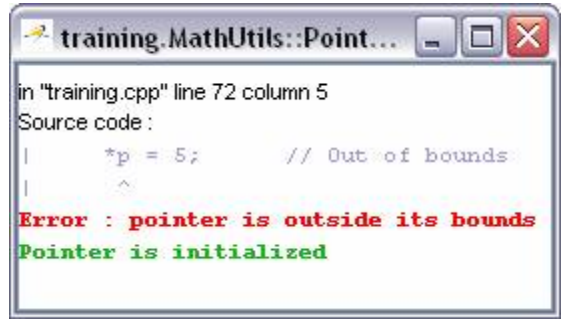
1 Click on to expand “MathUtils::Pointer_Arithmetic()“ to find out more about the red error. It displays a list of red, green, and orange symbols, featuring the complete list of code areas that PolySpace checked within the “MathUtils::Pointer_Ari thmetic()” function.



2 Click on the red “IDP.13” item - which stands for Illegal De-referenced Pointer -, to precisely locate this e rror in the source code. The bottom right section is updated showing the location of the “IDP.13” item.

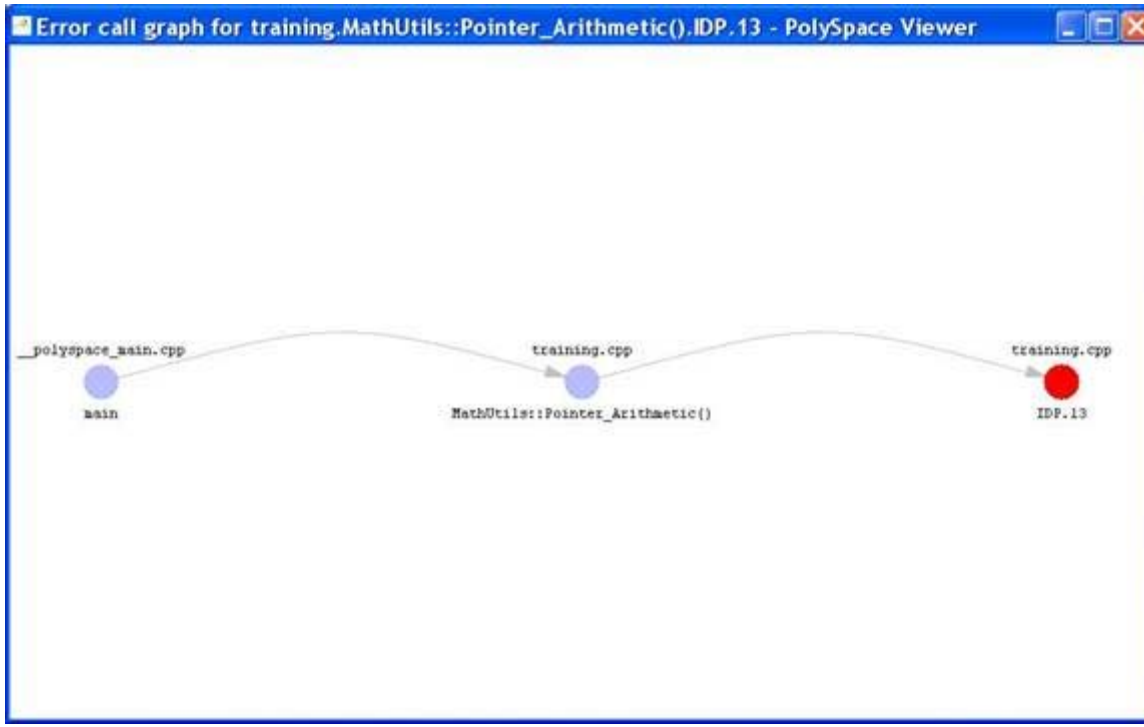


3 Click on red symbol in the source code at line 72. An error message is opened with the exact location:



Pointer p is de-referenced outside of its bounds. Indeed, at line 72 the instruction “*p = 5;” corrupts the memory as it puts the value “5” outside of the array “tab” pointed to by the pointer “p”.

4 You can also see the calling sequence leading to that particular red code section. To do so, select “IDP.13” 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 displa y the corresponding run-time error access graph:



Colors in the Source Code View

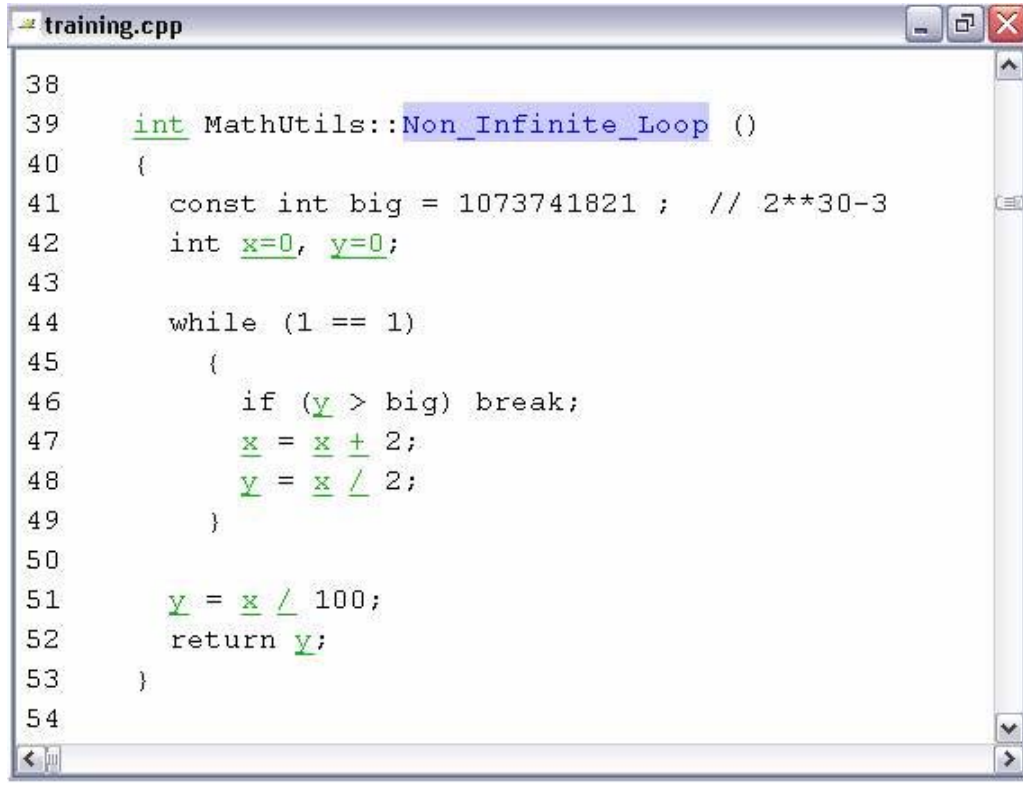
Each operation checked is also displayed using meaningful color scheme and related diagnostic in the source code view as links:

- 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 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 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.

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



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

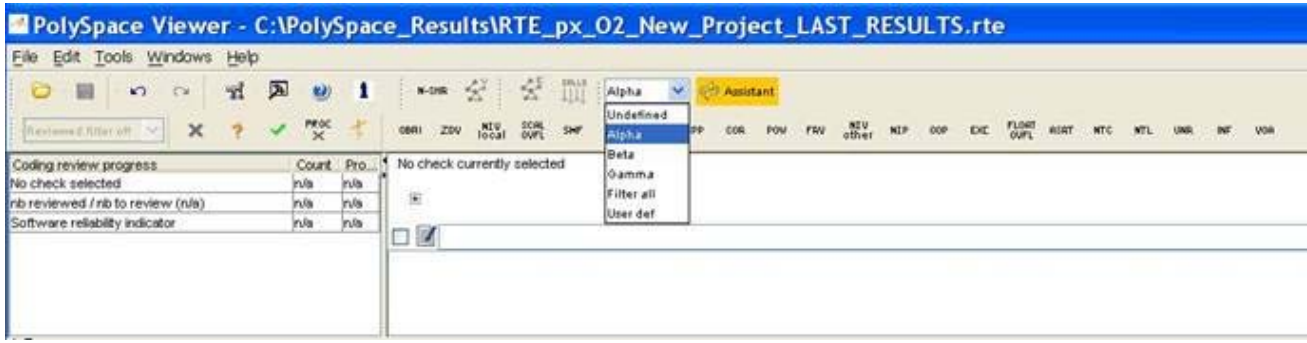
Example: Unreachable Code. We can also see in the “Procedural entities” column that some function members are never called. It is materialized by a reverse video in grey:

In the figure above it is the case for al l public and protect member functions of “Square” and “RTE” classes. Indeed, the PolySpace analysis was made for the class “MathUtils”.

Advanced Results Exploration

You can filter the information provided by PolySpace 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 pr ocess. Theses filters are accessible through a combo list:



To illustrate the use of these filters, w e will focus on the Pointer arithmetic member function that we have examined in a 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 “MathUtils::Pointer_Arithmetic()”.

This list of acronyms - for type of oper ations checked - shows what PolySpace automatically analyzed for you. In the case of member function is an illegal dereference pointer error (IDP.13).

Note VOA check (Value On Assignment) is only informative check that are never hidden.

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

To select Beta mode:

1 Click .

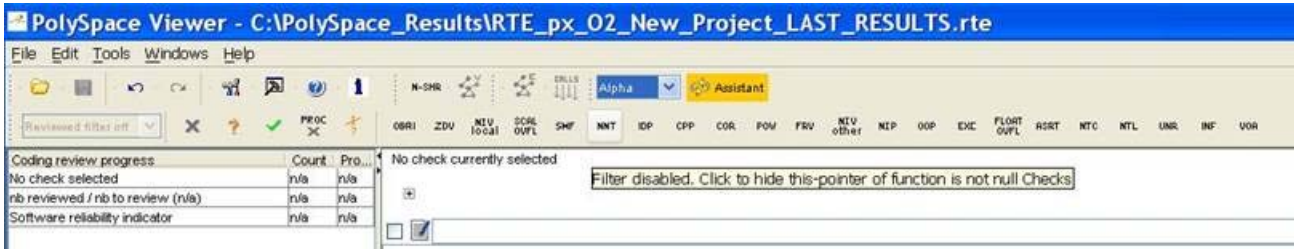
2 Select “MathUtils::Pointer_Arithmetic ()” in the “Procedural entities” view.

3 Click to get the list of the checks.

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

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 Filter all (top of the window) to suppress all checks, then click IDP.

  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:

C++ specific checks

Specific C++ checks are disp atched in five categories:

- NNT category or Non Null This pointer ( ). It checks this pointer validity.
- CPP category. It concerns C++ related constructions ( ), like positive array size verification, dynamic_cast , and typeid.
- OOP category. It concerns all C++ object oriented verification ( ): inheritance and virtual calls.
- EXC category. It concerns all C++ cons tructions dealing with exceptions ( ).
- INF category. It concerns informat ion about C++ implicit and called functions when dealing with virtual functions ( ).

When revie wing C++ code with PolySpace Viewer, it is important to have a selective review check by check which follows the list of categories located at the top of the PolySpace Viewer window. Che cks are classified from the left to the rig ht. It is important to begin a review following this order. It is also importan t to begin by C “like” checks before C++ like “checks”.

This meth odology permits to focus first by the categories which are most susceptible to hide run time errors. This met hodology has been automatically applied in the “Methodological Assistant” on page 2-54.

Miscellaneous

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

Close the PolySpace Viewer windo w by clicking on the upper right

symbol (PolySpace Viewer can also be closed using File > Close).

Methodological Assistant

- “Methodological Assistant Overview” on page 2-54
- “Opening the Methodologica l Assistant ” on page 2-54
- “Assistant Dashboard” on page 2-55
- “Choose a Methodological Assistant” on page 2-59

Methodological Assistant Overview

After a first navigation into the Poly Space 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 theses 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.

Opening the Methodological Assistant

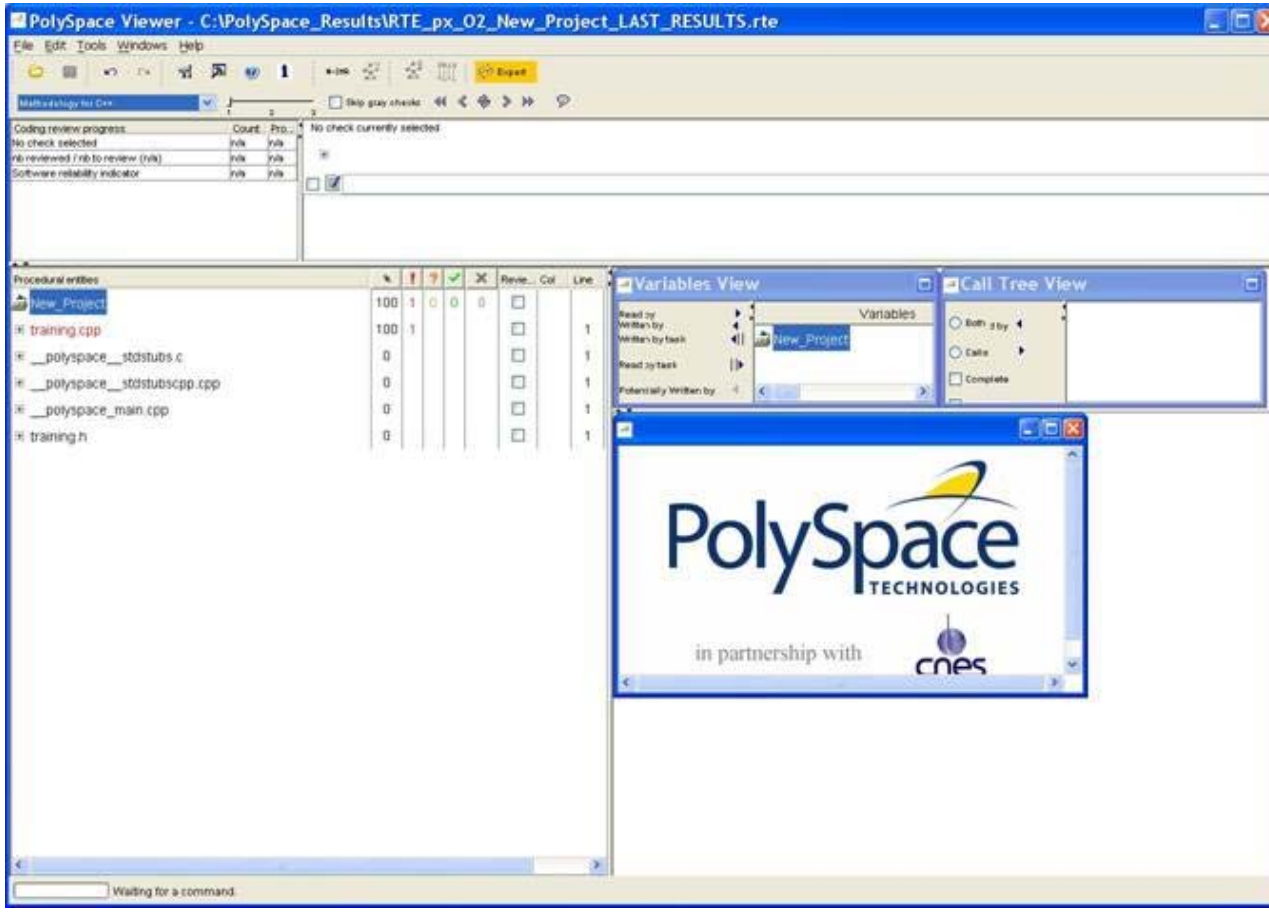
To open the assistant:

1 If the PolySpace Viewer is still open, close it.

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

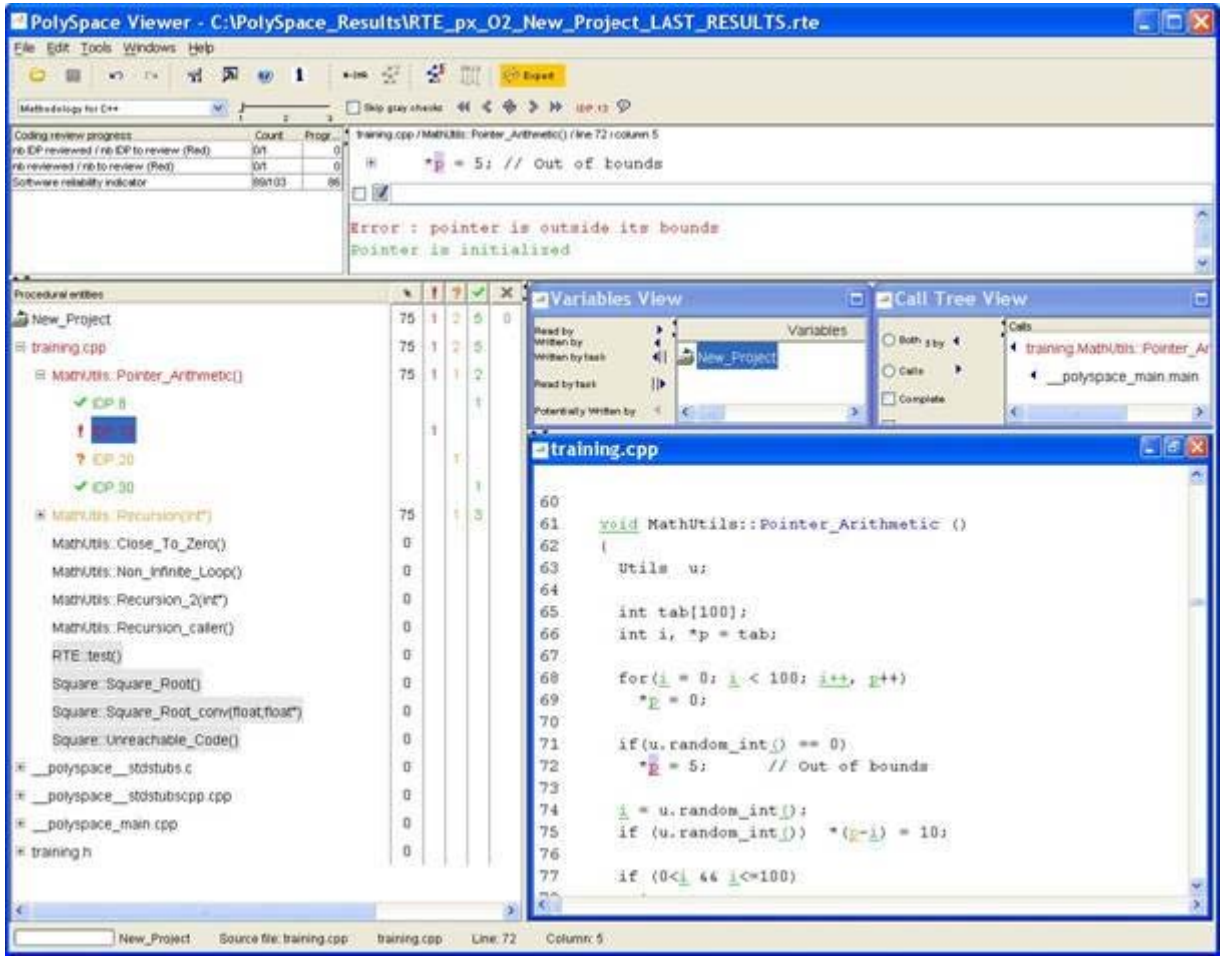
The second line of buttons on the toolbar and the two views just below are the navigation centre based on the methodological method used in the assistant mode:

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 co de view with colored instructions. Each operation will be checked and s orted 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 t he “best” unproven operations -errors that may occur sometimes.

1 Click to navigate to next check.

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

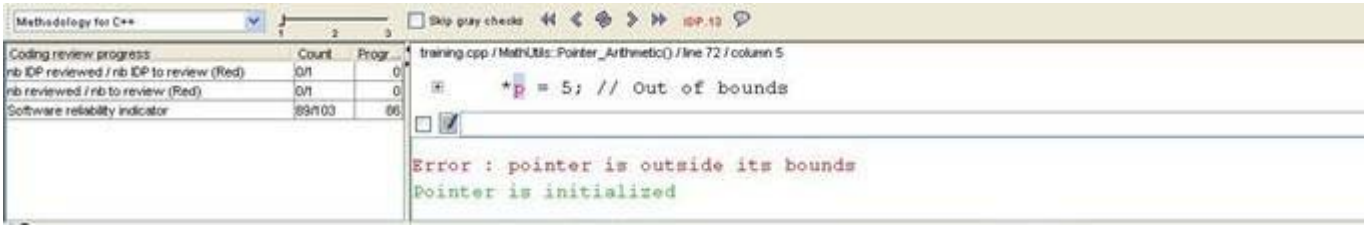


The Methodological dashboard gives details and allows reviewing the check. On the selected check, it is possible to mark the fact that it has been reviewed.

2 Select the radio button box.

3 Enter a comment in the associated edit box on the right.

After, it looks like:



The 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 col or category (red, grey and unproven).
- The Last line gives in permanence th e Software reliabi lity indicator.

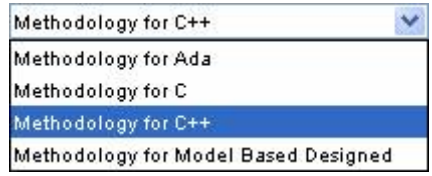
Other buttons in the Methodological das h board allow navigating to previous check, coming back to current one

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.

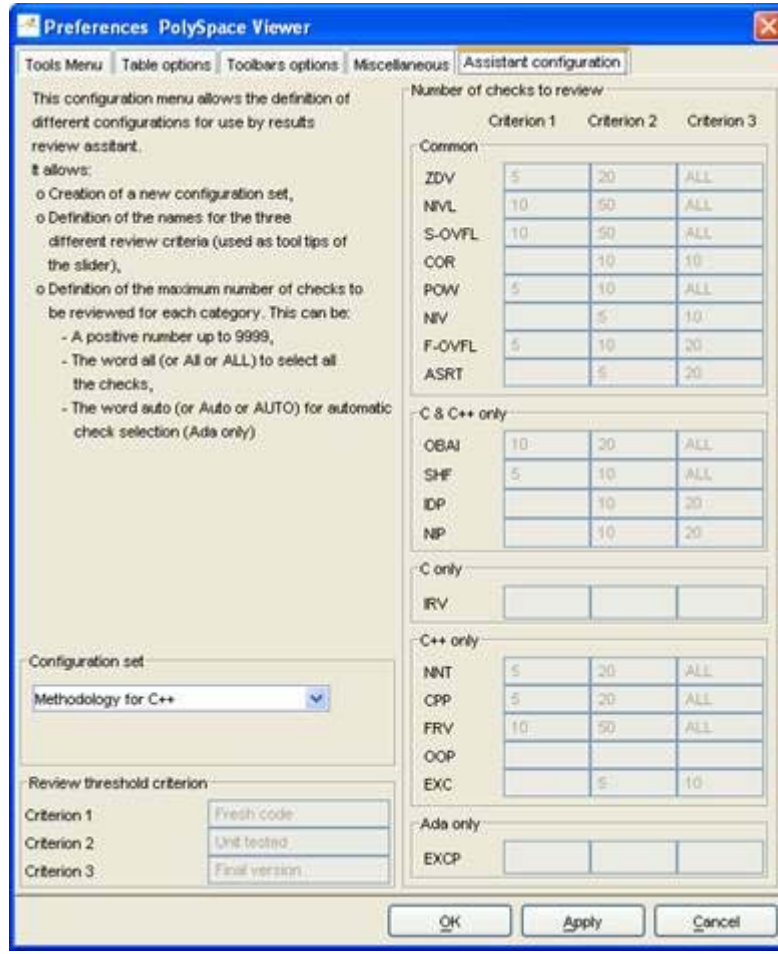
The methodology allows selecting the categories of checks to review, the number for each category and their order depending of a statistical algorithm.

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

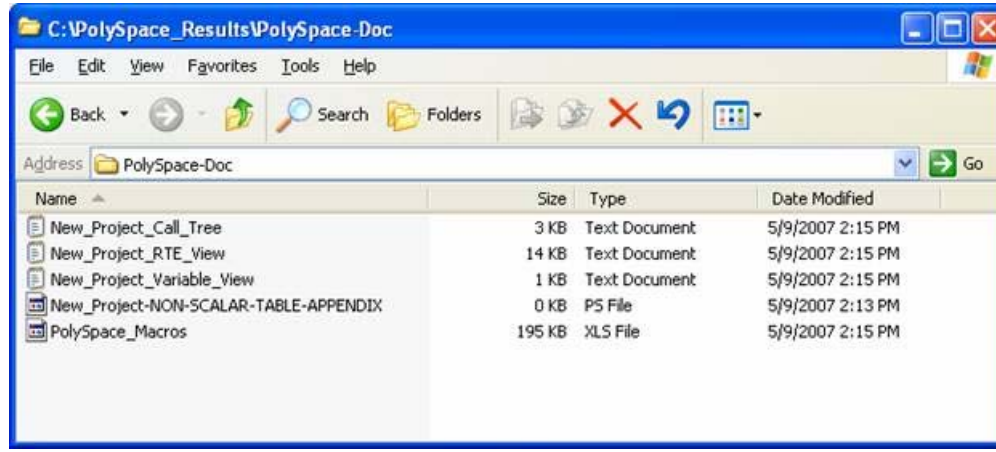
When PolySpace performs an analysis, it generates textual files that can be used to generate Microsoft ® Excel® reports.

Note Excel report is an option of PolySpace Desktop and Verifier only available under license. If you do not currently have a license and would like to learn more about it, please contact The Mathworks.

These files are located in the results directory (See ”C:\PolySpace_Results\PolySpace-Doc“ or “<PolySpaceInstallDir>\Examples\Demo_CPP\PolySpace-Doc”).

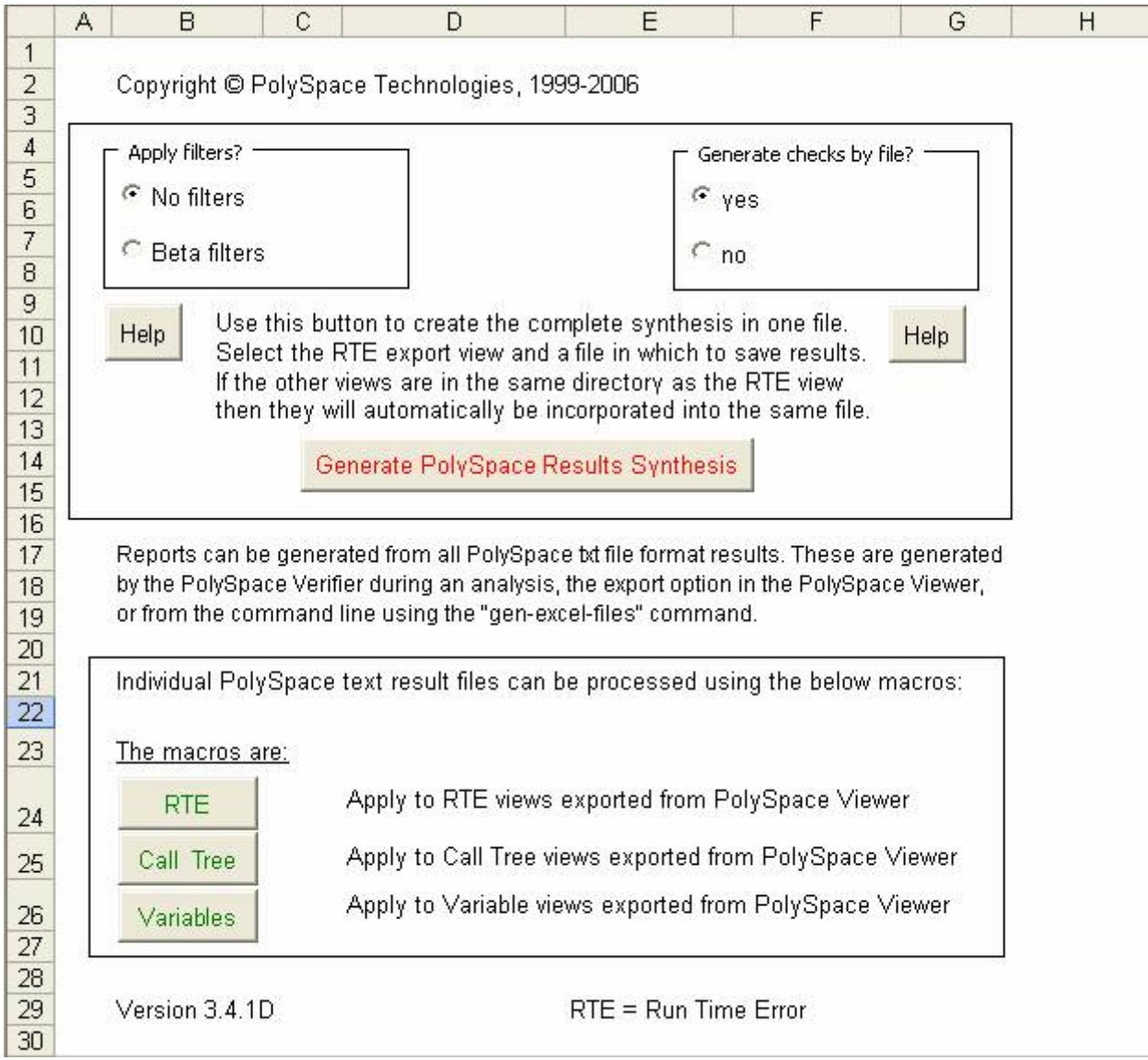
All views (except source code) are prin table and can be exported to textual or Excel format (protected by license).

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



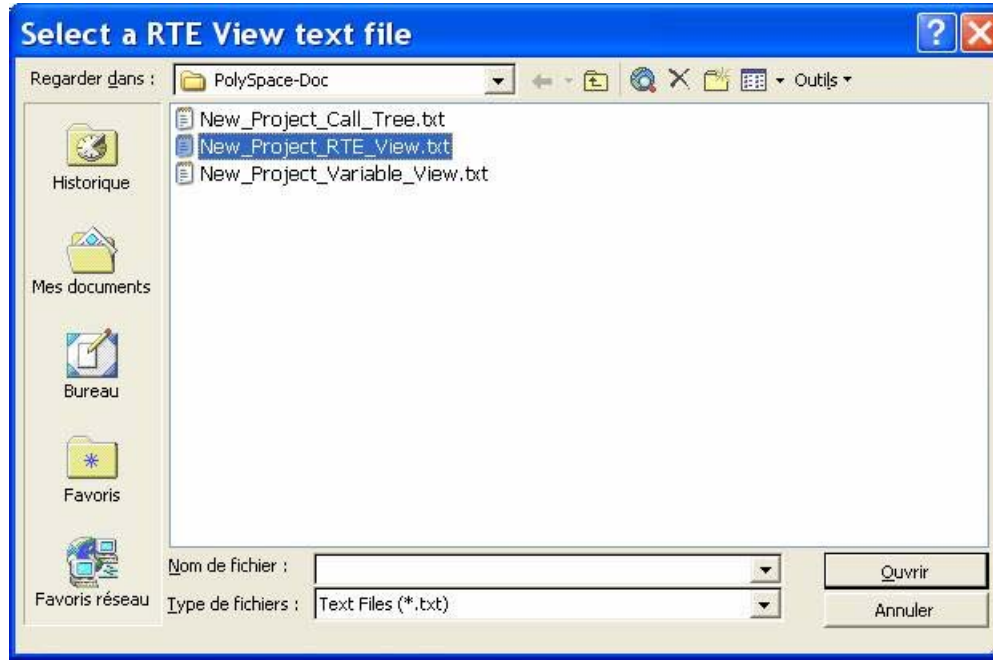
To generate a report:

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



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.

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.

Step 4: Launch PolySpace™ Remotely

In this section...

“Overview” on page 2-65

“Launching an Analysis” on page 2-65

“Managing Your Remote Analysis: the PolySpace™ Spooler” on page 2-67

“Batch Commands” on page 2-70

“Sharing Analyses Between Accounts” on page 2-72

Overview

This chapter 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 or the PolySpace Install Guide Manual) to install and configure a Client and a Server.

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

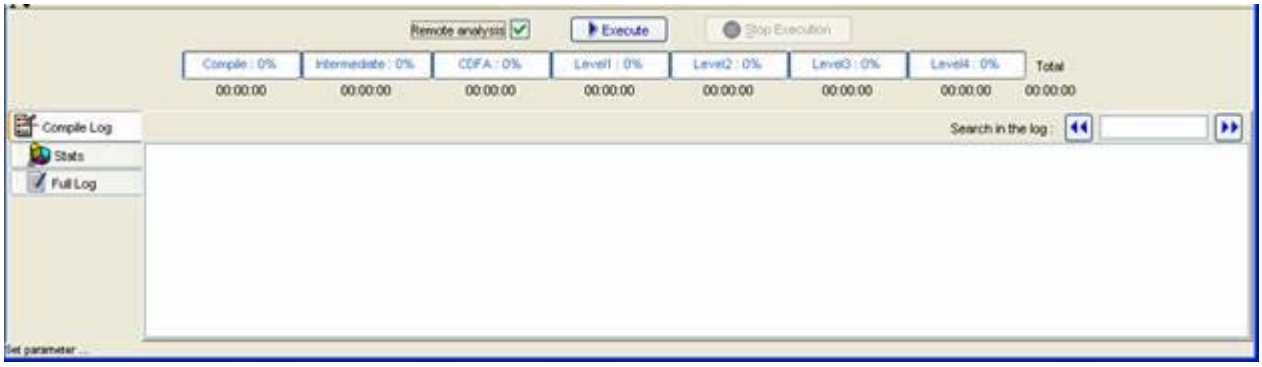
Launching an Analysis

To launch PolySpace remotely:

1 Set up an analysis as described in “Step 1: PolySpace™ Client – Setting Up and Launching an Analysis on a Single Class” on page 2-5, but do not launch it.

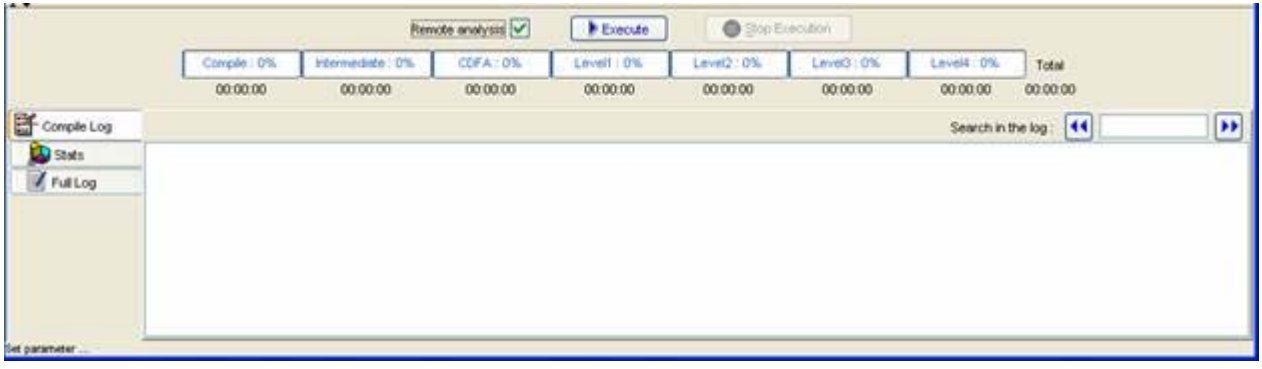
2 Select the Remote analysis checkbox (see next figure).

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 anal ysis” radio button, the analysis continues locally.

Managing Your Remote Analysis: the PolySpace™ Spooler

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

To manage an analysis in the queue:

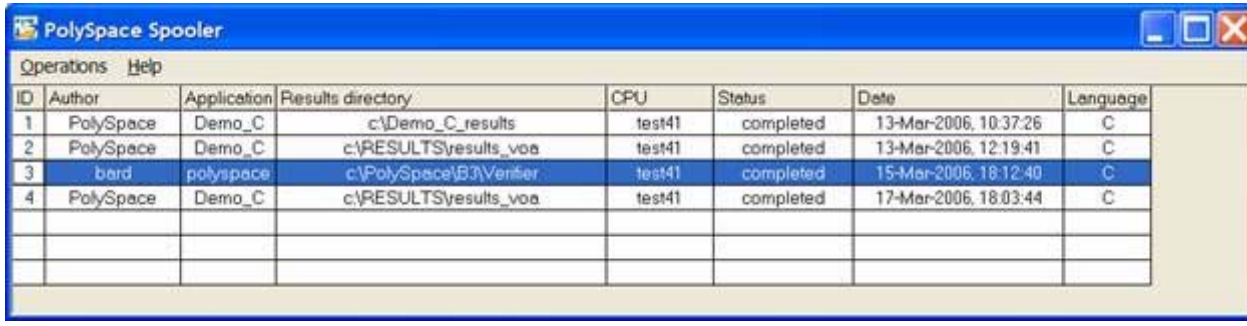
1 Open the PolySpace Spooler by either:

- Clicking on the short cut on your desktop PC

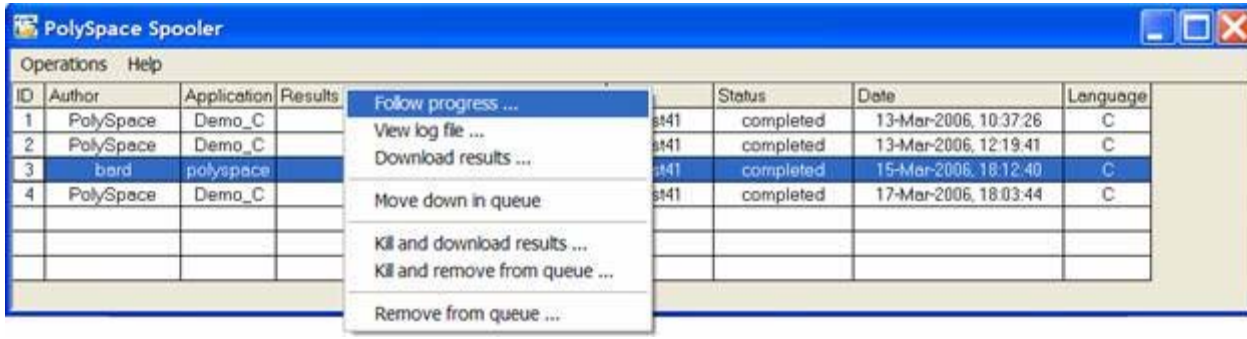
- Clicking on the icon

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 dis turbing 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.

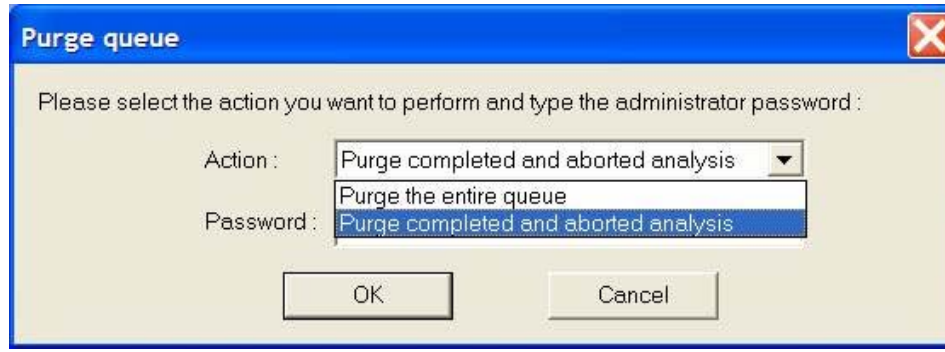
Caution The results will be lost

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

Caution	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.

Note	By default this password does not exist.

On a UNIX platform , there is no graphical user interface but a set of “Batch Commands” on page 2-70 which allow the management of analyses on the queue. All theses commands begin with the prefix <PolySpaceCommonDir>/Rem oteLauncher/psqueue-.

Batch Commands

- “Launching an An alysis in Batch” on page 2-70
- “Managing an Ana lysis in Batch” on page 2-70

Launching an Analysis in Batch

A set of commands allow you to launch an analysis in batch.

All theses comma nds begin with the following prefixes:

- Server analysis — <PolySpaceIns tallDir> /Verifier/bin/polyspace-remote-cpp
- Client analysi s — polyspace-remote-desktop-cpp

These commands are equivalent to commands with a prefix <PolySpaceIns tallDir> /bin/polyspace- .

For example, polyspace-remote-desktop-cpp -server [<hostname> : [<port>] | auto] allows you to send a C++ client analysis remo tely.

Note If your Po lySpace server is running on Windows ®, the batch commands are l ocated in the /wbin/ directory. For example, <PolySpaceIn stallDir> /Verifier/wbin/polyspace-remote-cpp.exe

---

Managing an Analysis in Batch

In batch , a set of commands allow the management of analysis in the queue.

On UNIX ®, all theses command begin with the prefix <PolySpaceC ommonDir> /RemoteLauncher/bin/psqueue- :

On Windows, t hese commands begin with the prefix <PolySpaceC ommonDir> /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 cu rrently 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 (cf. PolySpace Install 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 2-72
- “Magic Key or Shared Analysis Between Projects” on page 2-73

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 ke y 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 a class analysis using the PolySpace ™ Client™ for C/C++ product, and explore some results with PolySpace™ Viewer. All theses command can be performed locally on your desktop PC or in Client/Server architecture.

You will find more information on adva nced options available with our tools later in this guide.

Analysis Setup

Common Compile errors (p. 3-2)	Describes how to use PolySpace™ to detect compile errors
Dialect Issues (p. 3-8)	Describes issues resulting from different C++ dialects
Link Messages (p. 3-16)	Provides examples of link errors
Methodology Using the Pre-processed .ci Files (p. 3-21)	Describes how to use pre-processed files to locate compile errors
OS and Target Specifications (p. 3-26)	Describes supported operating systems and target processors
Intermediate Language Errors (p. 3-32)	Describes the log file containing error messages
Advanced Setup (p. 3-34)	Describes how to prepare your code to streamline orange checks

Common Compile errors

In this section...
“Includes” on page 3-2
“Specific Keyword or Extended Keyword” on page 3-2
“Initialization of Global Variables” on page 3-7

Includes

As for the C language, access to the sta ndard header files must be provided when the applications use the standard library.

The original code uses standard hea der files, but a message can appear:

```
Error message:
file.cpp", line 1: catastrophic error: could not open source file "iostream.h"
file.cpp:
  1 #include "iostream.h"
```

Use the -I optio n to include the correct header files, including the header files of the com piler.

Specific Keyw ord or Extended Keyword

- “Specific Key word” on page 3-2
- “Non ANSI ® Keyw ords” on page 3-3
- “Complex Post P reprocessing Command” on page 3-4
- “Perl Regular Expressions Summary” on page 3-5

Specific Keyword

Compilers of s pecific application are defined theirs owned keyword. A classic example is the compiler for micro controll er as IAR or Keil compiler.

Original code :

keyword.h	keyword.cpp

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

Error message:

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

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

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

You need to use the option -Dto not take accounts these keywords: -D far=

Non ANSI ® Keywords

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

1 Create a file called ABC.txt, and save it under c:\PolySpace

2 Open it with an ASCII editor, and copy and paste the following text:

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

```
sed "s/@interrupt//g"
```

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

Note Under Linux ®, you must:

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

---

4 Launch 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++;
}
```

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

Complex Post Preprocessing Command

If you want to ignore non-compliant keyw ords such as “far” or 0x followed by an absolute address, you can use the templa te described below to deal with them. Save it under c:\PolySpace\myTpl.pl, and select myTpl.plin the PolySpace™ Launcher using browse button associate d 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 Expressions 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 occurence 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/;
#
##############################################################################
```

Initialization of Global Variables

When a data member of a class is declared static in the class definition, then it is a static member of the class. Static data members are initialized and destroyed outside the class, as they exist even when no instance of the class has been created.

```
class Test
{
public:

  static int m_number = 0;
};
```

Error message:

```
Verifying Test.cpp
"../sources/test.h", line 33: error: data member initializer is not allowed
   static int m_number = 0;
          ^
1 error detected in the compilation of "CPP-ALL/SRC/MACROS/Test.cpp".
```

Corrected code:

in file Test.h	in file Test.cpp

class Test
{
public:
static int m_number;
};
	int Test::m_number = 0;

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

Dialect Issues

In this section...

"ISO® versus Default Dialects" on page 3-8

"CFront2 and CFront3 Dialects " on page 3-10

"Visual Dialects" on page 3-11

"GNU Dialect" on page 3-13

ISO® versus Default Dialects

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

Original code (file permissive.cpp):

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

- Using dialect iso, should be: friend class B;

```
"./sources/permissive.cpp", line 5: error: omission of "class"
is nonstandard
   friend B ;
```

- Using dialect iso,, the line 6 must be removed

```
"./sources /permissive.cpp", line 6: error: forward declaration
of enum type
is nonstandard
   enum   e ;
      ^
```

- Using dialect iso, line 7 should be: double ff = 0.0;

```
"./sources/permissive.cpp", line 7: error: invalid combination
of type
specifiers
   long float ff = 0.0 ;
   ^
```

- Using dialect iso, line 14 needs to be removed

```
"./sources/permissive.cpp", line 14: error: class member typedef
may not be
redeclared
   typedef T *   pointer ; // duplicate !
          ^
```

- Using dialect iso, line 21 needs to be changed by: typedef typename traits<T>::pointer pointer

```
"./sources/permissive.cpp", line 21: error: nontype
"traits<T>::pointer [with T=T]" is not a type name
   typedef traits<T>::pointer pointer ;
```

All these error messages will disappear if the –dialect default option is activated.

CFront2 and CFro nt3 Dialects

As mentioned at t he beginning of this section, cfront2 and cfront3 dialects were already bei ng used before the publication of the ANSI ® C++ standard in 1998. Nowadays, these two dialects are used to compile legacy C++ code.

If the cfront2 o r cfront3 options are not selected, you may get the common error messages below.

Variable Scope Issues

The ANSI C++ sta ndard specifies that the scope of the declarations occurring inside loop de finition is local to the loop. However some compilers may assume that th e scope is local to the bloc ({ }) which contains the loop.

Original code :

```
for (int i = 0; i < maxval; i++) {...}
if (i == maxval) {
  ...
}
```

Error message:

```
Verifying Test.cpp
"../sources/Test.cpp", line 26: error: identifier "i" is undefined
    if (i == maxval) {
        ^
```

Note This kind of construction has been allowed by compilers until 1999, before the Standard became more strict.

“bool” Issues

Standard type may need to be turned into boolean type

Original code:

```
enum bool
 {
```

```
    FALSE=0,
    TRUE
  };
class  CBool
{
public:
  CBool  ();
  CBool  (bool  val);
  bool  m_val;
};
```

Error message:

```
Verifying  C++  sources  ...
Verifying  CBool.cpp
"../sources/CBool.h",  line  4:  error:  expected  either  a  definition
or  a  tag  name
  enum    bool
      ^
```

Visual Dialects

The following messages will appear if the compiler is based on a visual dialect (including visual8).

Import Directory

When a Visual application uses #import directives, the Visual C++ compiler generates a header file which contains s ome definitions. These header files have a .tlh extension and tPolySpace C++ requires the directory containing those files.

Original code:

```
#include  "stdafx.h"
#include  <comdef.h>
#import  <MsXml.tlb>
MSXML::_xml_error  e  ;
MSXML::DOMDocument*  doc  ;
int  _tmain(int  argc,  _TCHAR*  argv[])
{
```

```
  return 0;
}
```

Error message:

```
"../sources/ImportDir.cpp", line 7: catastrophic error: could not
open source file
"./MsXml.tlh"
  #import <MsXml.tlb>
              ^
```

The Visual C++ compiler generates these files in its “build-in” directory (usually Debugor Release). Therefore, in order to provide those files, the application needs to be built first. Th en, the option -import-dir=<build directory> must be set with a correct path folder.

pragma Pack

Using a different value with the compile flag (#pragma pack) can lead to a linking error message.

Original code:

test1.cpp	type.h	test2.cpp

`#pragma pack(4)`
 	
`#include "type.h"`
	`struct A`
	`{`
	`  char c ;`
	`  int i ;`
	`} ;`
		`#pragma pack(2)`
		
		`#include "type.h"`

Error message:

```
Pre-linking C++ sources ...
"../sources/type.h", line 2: error: declaration of class "A" had
a different meaning during compilation of "CPP-ALL/SRC/MACROS/test1.cpp"
(class types do not match)
  struct A
       ^
    detected during compilation of secondary translation unit
```

```
"CPP-ALL/SRC/MACROS/test2.cpp"
```

The option -ignore-pragma-pack is mandatory to continue the analysis.

GNU Dialect

The GNU dialect is based on GCC 3.4. The GNU dialect supports the keyword __asm__ __volatile__ , which is used to support inline functions. For example, the <sys/io.h> header includes many inline functions. The GNU dialect supports these inline functions.

PolySpace software supports the following GNU elements:

- Variable length arrays:

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

- Anonymous structures:

  ```
  union A { struct { double x ; double y ; double z} ;
  double tab[3] ; } a ; assert(&(a.tab[0]) = &(a\ .x)) ;
  ```

- All other syntactic constructions allowed by GCC, except as noted below

Partial Support

Zero length arrays have the same support as in Visual Mode. They are allowed when used through a pointer, but not in a local variable.

Syntactic Support Only

PolySpace software provides syntactic support for the following options, but not semantic support:

- __attribute__(...) is allowed but generally not taken into account.
- No special stubs are computed for predeclared functions such as __builtin_cos , __builin_exit , and __builtin_fprintf).

Not Supported

The following options are not supported:

- The keyword __thread
- Statement expressions:

```
int i = ({ int tmp ; tmp = f() ; if (tmp > 0 ) { tmp = 0 ; } tmp ; })
```

- Taking the address of a label:

```
{ L : void *a = &&L ; goto *a ; }
```

- General C99 features supported by default in GCC, such as complex built-in types (__complex__ , __real__ , etc.).
- Extended designators initializer:

```
struct X { double a; int b[10] } x = { .b = { 1, [5] =2 },
.b[3] = 1, .a = 42.0 };
```

- Nested functions

Examples

Example 1: _asm_volatile_ keyword

In the following example, for the inb_p function to correctly manage the return of the local variable _v , the __asm__ __volatile__ keyword is used as follows:

```
extern inline unsigned char
inb_p (unsigned short port)
{
   unsigned char _v;

   __asm__ __volatile__ ("inb %w1,%0\noutb %%al,$0x80":"=a"
            (_v):"Nd" (port));
   return _v;
}
...
```

Example 2: Anonymous Structure

The following example shows an u nnamed structure supported by GNU:

```
class x
{
public:

   struct {
   unsigned int a;
   unsigned int b;
   unsigned int c;
   };
   unsigned short pcia;
   enum{
   ea = 0x1,
   eb = 0x2,
   ec = 0x3
   };

   struct {
   unsigned int z1;
   unsigned int z2;
   unsigned int z3;
   unsigned int z4;
   };
};

int main(int argc, char *argv[])
{
   class x myx;

   myx.a = 10;
   myx.z1 = 11;
   return(0);
}
```

Link Messages

In this section...
“STL Library C++ Stubbing Errors ” on page 3-16
“Lib C Stubbing Errors” on page 3-17

STL Library C++ Stubbing Errors

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

Original code:

```
#include <map>

struct A
{
  int m_val;
};

struct B
{
  int m_val;
  B& operator=(B &) ;
};

typedef std::map<A, B> MAP ;

int main()
{
  MAP m ;
  A a ;
  B b ;

  m.insert(std::make_pair(a,b)) ;
}
```

Error message:

```
Verifying template.cpp
"<Product>/Verifier/cinclude/new_stl/map", line 205: error: no operator
"=" matches these operands
  operand types are: pair<A, B> = const map<A, B, less<A>>::value_type
  { volatile int random_alias = 0 ; if (random_alias) *((pair<Key, T> * )
_pst_elements)      = x ; } ; // read of x is          done here

  detected during instantiation of
"pair<__pst__generic_iterator<bidirectional_iterator_tag, pair<const Key,
T>>, bool> map<Key, T, Compare>::insert(const map<Key, T, Compare>::
value_type &) [with Key=A, T=B, Compare=less<A>]" at line 23 of "/cygdrive/
c/_BDS/Test-Polyspace/sources/template.cpp"
```

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

The preceding message can also appear with the directory names:

```
"<Product>/cinclude/new_stl/map", line 205: error: no operator "="
matches these operands

"<Product>/cinclude/pst_stl/vector", line 64: error: more than one
operator "=" matches these operands:
```

Be careful, that other compile or link ing troubles can appear with your own template definitions.

Lib C Stubbing Errors

Extern C Functions

Some functions may be declared inside an extern “C” { } bloc in some files but not in others. In this case, the linkage is different which causes a link error, as it is forbidden by the ANSI standard.

Original code:

```
extern "C" {
```

```
  void*  memcpy(void*,  void*,  int);
}
class  Copy
{
public:
  Copy()  {};
  static  void*  make(char*,  char*,  int);
};
void*  Copy::make(char*  dest,  char*  src,  int  size)
{
  return  memcpy(dest,  src,  size);
}
```

Error message:

```
Pre-linking  C++  sources  ...
"<results_dir>/CPP-ALL/CPP-STUBS/__polyspace__stdstubs.c",  line  2996:  error:
declaration  of  function  "memcpy"  is  incompatible  with  a  declaration  in  another
translation  unit  (parameters  do  not  match)
      the  other  declaration  is  at  line  4  of  "/sources/Copy.cpp"
extern  void  *  __pst_profile__memcpy  (void  *s1,  const  void  *s2,  size_t  n)  ;
extern  "C"  void  *    memcpy  (void  *s1,  const  void  *s2,  size_t  n)
          ^

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

The function memcpy style='font-family:Arial'>is declared as an external "C" function and as a C++ function. It causes a link problem. Indeed, function management behavior differs whether it relates to a C or a C++ function.

When such error happens, the solution is to homogenize declarations, i.e. add “extern “C” { }” around pre vious listed C functions.

Another solution consists in using perm issive option -no-extern-C. It will remove all declaration extern "C"

Standard Stubs

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

Original code:

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

};
```

Error message:

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

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

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

Functional Limitations on Some of Stubbed Standard ANSI Functions

- signal.h is stubbed with functional limitations: 'signal' and 'raise' functions do not follow the associated functional model. Even if the function raise is called, the stored function pointer associated to the signal number is not called.
- No jump is performed even if the 'setjmp' and 'longjmp' functions are called.
- errno.h is partially stubbed. Some math functions, for which PolySpace uses built-in code, do no set errno b ut instead generate a red error when a range or domain error occurs (see examples with NTC checks).

You can also use the compile option POLYSPACE_STRICT_ANSI_STANDARD_STUBS (-D flag) which will only deactivate extensions to ANSI C standar d libC. Functions bzero, bcopy, bcmp, chdir, chown, close, fchown, fork, fsync, getlogin, getuid, geteuid, getgid, lchown, link, pipe, read, pread, resolvepath, setuid, setegid, seteuid, setgid, sleep, sync, symlink, ttyname, unlink, vfork, write, pwrite, open, creat, sigsetjmp, __sigsetjmp and siglongjmpare concerned.

Methodology Using the Pre-processed .ci Files

In this section...

"Overview" on page 3-21

"Example of ci File" on page 3-21

"Methodology Guide" on page 3-23

Overview

In the preceding paragraphs, commo n types of compile or linking errors messages have been detailed. They are associated to C++ dialects, or specific options used by the dialect (for instance Microsoft Visual C++ with the option -import-dir).

Nevertheless, sometimes the error me ssages are not sufficient to find the cause of problems. Indeed they do not c orrespond to common error messages listed above.

PolySpace, as others compilers, transforms a source code to a pre-processed code. These files are located in the folder: <results directory>/CPP-ALL/SRC/MACROS or <results directory>/ALL/SRC/MACROS. They have a .ciextension and they will help to understand and find precisely the error problem.

Example of ci File

A *. ci file is a copy of original file containing whole header files inside a unique file:

- compile flags activate some parts of code,
- macro commands are expanded,
- arguments which are described as "#define xxx", are replaced by their owned definition,
- etc.

Extension.cpp	Extension.h

#include "Extension.h"
 	
Extension::Extension(int val)
{
m_val = 0;
ABS(val);
 	
if (val > MAX_VALUE)
m_val = -1;
}
 	
#ifdef _DEBUG
void Extension::message(char*) {}
#else
void print(char*) {}
#endif
	#define MAX_VALUE 10
	#define ABS(x) ((x)<0?(x):-(x))
	
	class Extension
	{
	public:
	int m_val;
	Extension(int val);
	
	#ifdef _DEBUG
	void message(char*);
	#else
	void print(char*);
	#endif
	};

The associated file Extension.ci uses the compile flag _DEBUG:

```
# 1 "../sources/extension.cpp"
# 1 "<Product>/Verifier/cinclude/polyspace_std_decls.h" 1

# 1 "../sources/extension.cpp" 2
# 1 "../sources/extension.h" 1
class Extension
{
public:
 int m_val;
 Extension(int val);

 message(char*);        // _DEBUG activates the message member function

};

# 2 "../sources/extension.cpp" 2

Extension::Extension(int val)
```

```
{
  m_val = 0;
  ((val)<0?(val): -(val));    // EXPANDED MACRO ABS

  if (val > 10 )          // MAX_VALUE REPLACED BY 10
    m_val = -1;
  }

void Extension::message(char*) {}
```

Analyzing these files with the compile flag -D _DEBUG expands the code fully and may help to find the problems quickly.

Methodology Guide

This guide is designed to help understanding errors messages, as well as the differences between your compiler and PolySpace:

1 Check whether the compile error me ssages come from a dialect problem.

2 Check whether Verify that linking e rror messages are related or not to:

- A C++ Stubbing error which coul d be resolved by an option (like -no-stl-stubs)
- C-Stubbing error which could be resolved by an option or a compilation flag like POLYSPACE_NO_STANDARD_STUBS or POLYSPACE_STRICT_ANSI_STANDARD_STUBS

3 Check the pre-processed *.ci files to see the expanded files. Looking at the pre-processed code can help to find errors faster.

Example with these original codes:

Child1.c	Child2.c	Test.h

#define DEBUG
 	
#include "Test.h"
 	
class Child1 : public Test
{
public:
Child1();
Child1(int val);
 	
void search(int val);
 	
};
	#undef DEBUG
	
	#include "Test.h"
	
	class Child2 : public Test
	{
	public:
	Child2();
	Child2(int val);
	
	void qshort(int val);
	
	protected:
	int m_status;
	};
		class Test
		{
		public:
		
		Test();
		Test(int val);
		
		int getVal();
		void setVal(int val);
		
		#ifdef DEBUG
		void algorithm(int val,
		int max);
		#endif
		
		private:
		int m_val;
		};

Error message:

```
Pre-linking C++ sources ...
"../sources/test.h", line 4: error: declaration of function
"Test::Test(const Test &)" does not match function
"Test::algorithm" during compilation of "CPP-ALL/SRC/
MACROS/Child2.cpp" (one may have been removed due to #define)
    class   Test
        ^
  detected during compilation of secondary translation unit
"CPP-ALL/SRC/MACROS/Child2.cpp"
```

In this example it is clear that DEBUG is defined in child1.c but not in child2.c which creates two different definition of the class test.

The solution can also come up by comparing the two *.ci files:

Test.ci	Child2.ci

....
 	
# 1 "../sources/Test.cpp" 2 	
# 1 "../sources/test.h" 1 	
 	
class Test
{
public:
Test();
Test(int val);
 	
int getVal();
void setVal(int val);
 	
void algorithm(int val, int max);
 	
private:
int m_val;
};
 	
# 2 "../sources/Test.cpp" 2 	
 	
....
	....
	
	# 1 "../sources/Child2.cpp" 2
	# 1 "../sources/Child2.h" 1
	# 1 "../sources/test.h" 1
	
	class Test
	{
	public:
	Test();
	Test(int val);
	
	int getVal();
	void setVal(int val);
	
	private:
	int m_val;
	};
	
	# 2 "../sources/Child2.h" 2
	
	....

Looking at the pre-processed code can help to find errors faster.

OS and Target Specifications

In this section...
“List of Predefined Compilation Flags” on page 3-26
“Target Specifica tions” on page 3-29
“Generic/Custom Target” on page 3-30

List of Predefined Compilation Flags

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

-OS-target	Compilation flags	-include file	Minimum set of options
Linux	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D__STL_CLASS_PARTIAL_ SPECIALIZATION -D__GNU_SOURCE -D__STDC__ -D__ELF__ -Dunix -D__unix -D__unix__ -Dlinux -D__linux -D__linux__ -Di386 -D__i386 -D__i386__ -Di686 -D__i686 -D__i686__ -Dpentiumpro -D__pentiumpro -D__pentiumpro__	<product_dir>/ cinclude/ pst-linux.h	polyspace-[desktop-]cpp -OS-target Linux \ -I <product_dir>/include/ include-linux \ -I <product_dir>/include/ include-linux/next Where the PolySpace product has been installed in the directory <product_dir>

-OS-target	Compilation flags	-include file	Minimum set of options
vxWorks	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D__STL_CLASS_PARTIAL_ SPECIALIZATION -DANSI_PROTOTYPES -DSTATIC= -DCONST=const -D__STDC -D__GNU_SOURCE -Dunix -D__unix -D__unux__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__ -D__svr4 -D__SVR4	<product_dir>/ cinclude/ pstvxworks. h	polyspace-[desktop-]cpp \ -OS-target vxworks \ -I /your_path_to/ Vxworks_include_directories
visual /visual6	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__STRICT_ANSI__ -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D_POSIX_SOURCE -D__STL_CLASS_PARTIAL_ SPECIALIZATION	<product_dir>/ cinclude/ pstvisual. h	

-OS-target	Compilation flags	-include file	Minimum set of options
Solaris	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D__STL_CLASS_PARTIAL_SPECIALIZATION -D__GNU_SOURCE -D__STDC -D__GCC_NEW_VARARGS__ -Dunix -D__unix -D__unux__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__ -D__svr4 -D__SVR4		If PolySpace runs on a Linux machine: polyspace-[desktop-]cpp \ -OS-target Solaris \ -I /your_path_to_solaris_include If PolySpace runs on a Solaris machine: polyspace-cpp \ -OS-target Solaris \ -I /usr/include
no-predefined-OS	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__STRICT_ANSI__ -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D_POSIX_SOURCE -D__STL_CLASS_PARTIAL_SPECIALIZATION		polyspace-[desktop-]cpp \ -OS-target no-predefined-OS \ -I /your_path_to/MyTarget_include_directories

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

Target Specifications

PolySpace™ products take the type of CPU used in the target environment into account during verification. This determines various characteristics of data representation such as data sizes, addressing, etc. These are essential to correctly determine some types of errors, such as overflows.

PolySpace products support 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 same listed characteristics.

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

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

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

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

Generic/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

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	64	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

Intermediate Language Errors

The analysis log can sometimes indica te that a red error has been detected in the previous phase, and that the a nalysis has therefore stopped. If no graphical result is provided, the errors and their location s 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 14 (P_PT) took 2real,
0.5u +1s (0.1gc)
**** C++ to intermediate language translation 15 (P_IL)
* Set of cloned functions : {operator_delete(void*)}

------------------------------------------------------------------------
1 User Program Errors:
* certain failure of correctness condition [non-initialized local
variable] file stubbing.cpp line 43 column 46
Please correct the program and restart the verifier.
------------------------------------------------------------------------

**** C++ to intermediate language translation 15 (P_IL) took 32.7real,
10.2u + 19.5s (0.2gc)
**** C++ to intermediate language translation 16 (P_IPF)
**** C++ to intermediate language translation 16 (P_IPF) took 0.3real,
0.1u + 0.1s
* assigns: 63% reduction
* asserts: 33% reduction
* total : 71% reduction
************************************************************
***
*** C++ to intermediate language translation done
***
************************************************************
```

```
Ending at: Oct 6, 2005 18:18:0
User time for iabc-c2if: 65real, 14u + 27.8s
```

Advanced Setup

In this section...

"Reduce Oranges Step by Step" on page 3-34

"Approximations Made by PolySpace" on page 3-47

"Variables" on page 3-54

"Types Promotion" on page 3-58

"Built-in Functions" on page 3-61

Reduce Oranges Step by Step

Although PolySpace 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 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.

The following examples show how the selectivity can be improved by each of these four means.

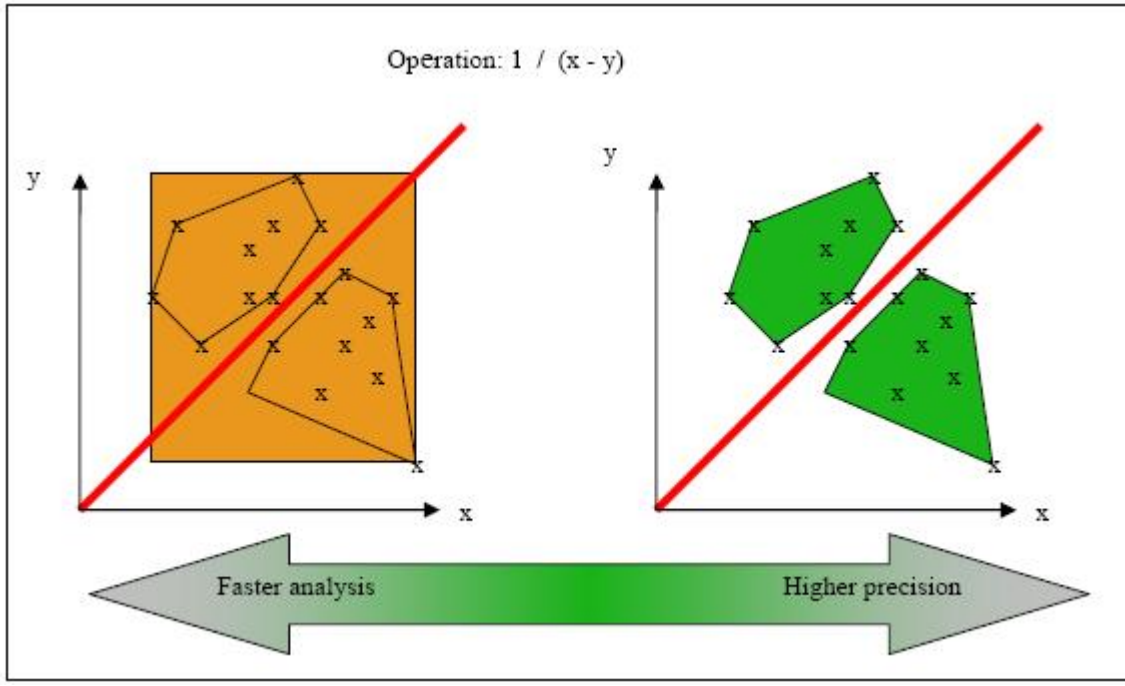
- "Vary the Precision Level" on page 3-35
- "Apply Manual Stubbing" on page 3-36

Since increasing the selectivity can br ing any of the following benefits - more red, more grey, or readable orange - the following examples will give one example of each. There is no implied link between the approach taken to improve selectivity in an example, and the particular way the improvement manifests itself.

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, -O0, -O1, -O2 or -O3 precision level.

The methods used by Verifier to represen t 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 p recision analysis will become green when analyzed at a higher precision.



Vary the Precision Rate

Apply Manual Stubbing

The default behavior of PolySpace is to automatically stub missing functions members in accordance with their proto types. If the function takes a pointer as an argument, PolySpace assumes that the stub can write into the contents of this pointer. This default behavior a voids the occurrence of red errors as the result of incomplete code sets. However, stubbin g which accurately reflects the behavior of the missing code will allow PolySpace to show more red and grey code, rather than orange checks.

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.

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 as returning all values in the full type of an integer.

It will reduce the cloud of points and therefore increase the precision if a restricted range is specified instead of the full range. Nevertheless, it is not necessary to write the exact code depending on complicated algorithm, and an interpolation between 4 parameters; only a quick stub is required, as shown in the following example:

with volatile and assert	with assert, and without volatile	without assert, without volatile, without "if"

int stub(void)
{
volatile int random;
int tmp;
tmp = random;
assert(tmp>=1 && tmp<=10);
return tmp;
}
	int other_func(void);
	int stub(void)
	{
	int tmp;
	tmp= other_func();
	assert(tmp>=1 && tmp<=10);
	return tmp;
	}
		int other_func(void);
		int stub(void)
		{
		int tmp;
		do {tmp= other_func();}
		while (tmp<1 || tmp>10);
		return tmp;
		}

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 the assembly code options).

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 va riable 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 run-time 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

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

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 knowledg e that no run-time error will be ever introduced by automatic stubbing ; to minimize preparation time.

- “Examples: Specification” on page 3-40
- “Colored Source Code Example” on page 3-41

- “Specify the Call Sequence” on page 3-42
- “Constraint for Data” on page 3-43
- “Recoding of Some Specific Functions” on page 3-45

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

Here is the first example:

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

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

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

Note Default stubbing is suitable here.

Here are the reasons why:

- The content of the in_cnx structure might be changed by this function.
- The possible return values of 0 or 1 compared to the full range of an integer wont have much impact on the Run Time Error aspect. It is unlikely that the results of this operation will be used to compute some mathematic algorithm . It is probably a Boolean status flag and if so is likely to be stored and compared to 0 or 1. The default stub would therefore have no detrimental effect.

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

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

Here are the reasons why:

- For the return value, default stubbing would be applicable as explained in the previous example.
- Since the structure is a read-only parameter it will be worth stubbing it manually to accurately reflect the behavior of the missing code. Benefits: PolySpace Desktop will find more red and grey code

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

Colored Source Code Example.

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

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

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

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

Manual stubbing example s for send_message:

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

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

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

Colored Source Code Example

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

```
static char x;
static int y;

void f(void)
{
y = 300;
}

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

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

Example of Call Sequence

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

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

Constraint for Data.

Default Behavior of Global Data

Initially, consider how PolySpace hand les the analysis of global variables.

There is a maximum range of values which may be assigned to each variable as defined by its type. By default, PolySpace assigns that full range for each global variable, ensuring that a meaningful analysis of such a variable can take place even when the functions that write to it are not included. If a range of values was not considered in these circumstances, such a variable would be assumed to have a value of zero throughout.

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

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

To some extent, the effectiveness of this technique is limited by compromises made by PolySpace to deal with issues of code complexity. For instance, it cannot be assumed that all of these ranges will be propagated throughout all function calls. Sometimes, perhaps as a result of complex function interactions or constructions where PolySpace is known to be imprecise, the

potential value of a variable will assume its full “type” range despite this technique having been applied.

Constraining the Data

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

The technique therefore requires

- A knowledge of the variables and the maximum ranges they may take in practice.
- A data dictionary in electronic forma t from which the variable names and their minimum and maximum values can be extracted.

Applying the Technique

To apply the technique:

1 Create the range setting stubs:

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

   b declare 6 global volatile variables for each type

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

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

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

Integer Example

```
volatile int tmp;
```

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

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

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

void  main(void)
{
init_all();

while(1)
  {
  if  (tmp)  function1();
  if  (tmp)  function2();
  //  ...
  }
}
```

Recoding of Some Specific Functions. Once data ranges have been specified (above), it may be benefici al to recode some functions in support of them.

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

Identify in the modules:

- API which read global variables through pointers

Replace this API:

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

#define MAX_Calibration_Constant_1 7
char Calibration_Constant_1[MAX_CALIB_1] = { 1, 50, 75, 87, 95, 97, 100} ;
T Constant_1 = { 0, 0,
        MAX_Calibration_Constant,
        &calibration_constant_1[0] } ;

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

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

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

With this one:

```
char Constant_1 ;

#define read_calibration(in,index) *in

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

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

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

```
}
```

- Points in the source code which exp and the data range perceived by PolySpace
- Functions responsible for full range data, as shown by the “Value on assignment” (voa.) feature.

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

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

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

Char read_from_data_my_global2(void)
{ }
```

- stub complicated algorithms, calibr ation read accesses and API functions reading global data - as usual. For instance, if an algorithm is iterative - stub it.
- variables
  - where the data range held by each element of an array is the same, replace that array wit h a single variable.
  - where the data range held by each ele ment of an array differs, separate it into discrete variables.

Approximations Made by PolySpace

- “Volatile Variables” on page 3-48
- “Structures with Volatile Fields” on page 3-48
- “Absolute Addresses” on page 3-48
- “Pointer Comparison” on page 3-49
- “Left Shift on Negative Variables” on page 3-49
- “Some Bitwise Operators” on page 3-49
- “Bitfields” on page 3-50
- “Float Loops” on page 3-51

- “Shared Variables” on page 3-51
- “Array of Function Pointers” on page 3-52
- “Trigonometric Functions” on page 3-52
- “Unions” on page 3-53
- “Loop Exit Conditions” on page 3-54
- “Constant Pointer” on page 3-54

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[-
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 declare d 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, a n 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 is a static tool analyzing source code. Memory management concerns dynamic considerations, and the characteristics of particular compilers and targets. PolySpace there fore 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 gives a red error on the SHF check becaus e 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==0xFFFFFF51);  //  ASRT
checked
33  }
```

Bitfields

PolySpace considers a bitfield to be a permanently full range variable.

```
4  typedef  struct  _x
5  {    unsigned  int  a:1;
7    unsigned  int  b:1;  }  bit;

12  int  main(void)
13  {  bit  z;
14    z.b  =  0;
15    z.a  =  1;
16    assert(z.a  ==  1);  //  orange  ASRT
```

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 res ults 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 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 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 can lose precision when structure unions are considered. Indeed this kind of implementation is co mpiler 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 for VOA

```
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[1]  =  1;
if  (X.A  ==  0x1111)
```

```
else // both branches are reachable
```

Loop Exit Conditions

PolySpace 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)
```

Variables

- “How are Variables Initialized” on page 3-55
- “Data and Coding Rules ” on page 3-56
- “Variables: Declaration and Definition” on page 3-56
- “How Can I Model Variable Values External to My Application?” on page 3-57

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 “Reviewing code colored by PolySpace ” 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 viewe r. In this case, the resulting default behavior is that these pointers don’t point to any valid function. For results to be meaningful here, you may well need to define these variables explicitly.

Volatile.

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

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

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

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

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

Data and Coding Rules

Data rules are design rules which dict ate 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 va riables which are produced and used, as well as the theoretical ranges th ey can take as input and/or output values.

Variables: Declaration and Definition

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

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

- for a function, the prototype: `int f(void);`
- for an external variable: `extern int x;`

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

Definition. A definition provides:

- for a function, the body of the function has been written: `int f(void) { return 0; }`
- for a variable, a part of memory ha s 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 PolySpace and variables initialization section).

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

How Can I Model Variable Values External to My Application?

There are three main considerations.

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

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

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

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

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

Note Although the volatile characteristic of a variable is also commonly used by programmers to avoid compiler opti mization, 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]
}
```

Types Promotion

- "An Example of an Unsigned Promoted to Signed" on page 3-58
- "What are the Promotions Rules in Operators?" on page 3-59

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 that 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 The type of constants and constant overflows section), and also when analyzing any operation between variables of different types without an explicit cast.

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

Example

```
2 extern char random_char(void);
3 extern int random_int(void);
4
5 void main(void)
6 {
7   char c1 = random_char();
8   char c2 = random_char();
9   int i1 = random_int();
```

```
10   int i2 = random_int();
11
12   i1 = i1 + i2;     // A typical OVFL/UNFL on a + operator
13   c1 = c1 + c2;     // An OVFL/UNFL warning on the c1 assignment
[from int32 to int8]
14 }
```

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

```
2 extern char random_char(void);
3
4 void main(void)
5 {
6   char c1 = random_char();
7   char c2 = random_char();
8
9   c1 = (char)((int)c1 + (int)c2);   // Warning 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 i ntegral target size before realizing the arithmetic operation and subsequ ently 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.

Built-in Functions

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

Stubs of stl Functions

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

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

Stubs of libc Functions

Concerning the libc, all theses 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, c type.h, errno.h, locale.h, math.h, setjmp.h (’setjmp’ and ’longjmp’ functions are partially implemented – see <polyspaceProduct>/cinclude/ __polyspace__stdstubs.c), signal.h (’signal’ and ’raise’ functions are partially implemented – see <polyspaceProduct>/cinclude/__polysp ace__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 theses 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 theses rules The following functions which deal with memory allocation can not be redefi ned: malloc(), calloc(), realloc(), valloc(), alloca(), __built_in_m alloc() and __built_in_alloca().

PolySpace™ Class Analyzer Process

Analyzing C++ Classes (p. 4-2)	Describes the purpose of the class analyzer
Types of Classes (p. 4-3)	Provides tips for analyzing different types of classes

Analyzing C++ Classes

In this section...

"Overview" on page 4-2

"Why Provide a Class Analyzer" on page 4-2

Overview

This chapter presents a strategy for analyzing C++ classes. This allows the developer to identify, and possibly remove most of the runtime errors present in a class.

For technical details on how to use Poly Space™ class analyzer please refer to "Getting started" section.

Why Provide a Class Analyzer

One aim of object languages such as C++ is reusability. A class or a class family is reusable if it is free of bugs, for all uses of the class. It can be considered free of bugs if runtime erro rs have been removed and functional tests are successful. As evidence, the first objective is to remove as much as possible runtime errors.

PolySpace class analyzer is a mean for re moving runtime errors at compilation time. PolySpace will simulate all the possible use of a class, by:

1 Creating objects using all constru ctors (default if no one exists),

2 Calling all methods (public, static, and protected) on previous objects in any orders,

3 Calling all methods of the class between zero and an infinity of times,

4 Calling every destructor on pr evious object (if they exist).

Types of Classes

In this section...

“Simple Class” on page 4-3
“Simple Inheritance” on page 4-5
“Multiple Inheritance” on page 4-6
“Abstract Class” on page 4-7
“Virtual Inheritance” on page 4-8
“Other Types” on page 4-9

Simple Class

On the following class:

Stack.h

```
#define  MAXARRAY  100

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

public:
  int  Top  (void);
  int  IsEmpty  (void);
  int  Push  (int  newval);
  void  Pop  (void);
    stack  ();
};
```

stack.cpp

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

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

The Class analyzer calls the constructor and then all methods in any orders many times.

The analyze of this class with PolySpace™ Class-Analyzer highlights 2 problems, as shown in previous Viewer results:

- The stack::push method may write after the last element of the array (then the OBAI orange check at line 28).
- The stack::top method if called before Push will access element -1 (then the OBAI and NIV checks at line 13).

Fixing these problems will turn the class runtime error bugs free.

Simple Inheritance

Consider classes as follows:

A is the base class of B and D.

B is the base class of C.

In such case PolySpace class analyzer allows to do the following analysis:

1 The A class can be analyzed just by providing its own code to PolySpace. This corresponds to the previ ous simple class paragraph.

2 The B class can be analyzed by providing B code and A class declaration. In this case A code will be stubbed automatically by PolySpace.

3 The B class can be analyzed by providing B and A codes (declaration and definition). This is a first level of integration analysis. The class analyzer will not call A methods. In this case objective is to find bugs in B class code. Bugs in class A are found during previous steps.

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

5 The C class can be analyzed by providing codes of A, B and C classes as an integration analysis. The class analyzer will call all the C methods but not inherited methods from B and A. The objective is to find bugs in C class.

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

Multiple Inheritance

Consider classes as follows:

A and B are C base classes.

In such case PolySpace class analyzer allows to do the following analysis:

1. A and B classes can be analyzed separately just by providing there own code to PolySpace. This corresponds to the previous simple class paragraph.
2. The C class can be analyzed just by providing its own code with A and B declarations. A and B methods will be stubbed automatically.
3. The C class can be analyzed by providing codes of A, B and C classes as an integration analysis. The class analyzer will call all the C methods but not inherited methods from A and B. The objective is to find bugs in C class.

Abstract Class

Consider classes as follows:

A is an abstract class

B is a simple class.

A and B are C base classes.

C is not an abstract class.

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

Virtual Inheritance

Consider classes as follows:

B and C classes virtually inherit the A class

B and C are D base classes.

A, B, C and D can be analyzed in way described in previous chapter.

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

Other Types

Template

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

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

Example

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

You have to define a “typedef” to creat e a specialization of the template:

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

and use option -class-analyzer my_template .

It will analyze one instance of the template.

Class Integration

Lets consider a C class inheriting from A and B classes and having object members of AA and BB classes.

Doing a class integration analysis consists in verifying the C class and providing the code of A, B, AA and BB cla ss. If some definitions are missing PolySpace class analyzer will stub them automatically.

5

PolySpace™ C++ add-in for Visual Studio ®

Overview (p. 5-2)	Describes the PolySpace™ C++ add-in for Visual Studio ®
Using PolySpace™ Software Within Visual Studio ® (p. 5-3)	Describes how to use PolySpace software with Visual Studio
Analyzing an Entire Project (p. 5-16)	Describes how to analyze an entire project

Overview

This chapter describes the usage of PolySpace ™ Client™ for C/C++ while integrated in the Microsoft ® Visual C++ ® .NET (see The PolySpace™ Installation Guide in <PolySpaceIns tallCommon>/Docs directory for the exact compatibility).

The PolySpace C++ add-in for Visual Studio ® provides automatic source code verification and bug detection in source code developed inside the Visual IDE. It includes the following main features:

- An automatic setting of PolySpace pro ject configuration file derived from your Visual project settings.
- A direct launching of C++ file analysis from Visual IDE.
- A report of PolySpace compilation findings back to the IDE.

Using PolySpace™ Software Within Visual Studio ®

In this section...

"Overview" on page 5-3

"PolySpace™ Parameters Within Visual Studio ®" on page 5-3

"Performing a File Verification from Visual Studio ®" on page 5-7

"Analyzing Classes" on page 5-11

"The Configuration File and Default Options" on page 5-14

Overview

The PolySpace™ for Visual .NET plug-in allows you to launch C++ analyses inside the Visual C++ ® IDE whether those analyses are local or remote.

Launching options are integrated within the Visual editor through a PolySpace menu and a toolbox.

Note Some components of the plug-in are not automatically docked at installation. They must be manually moved where user wants them. Next time the interface will open, the components will be at the same place.

PolySpace™ Parameters Within Visual Studio®

When the PolySpace Visual C++ plug-in has been installed, a Polyspace menu and a PolySpace Log tab are displayed inside the Visual Studio® IDE. Those tools are used to start local or remote analysis on current classes and files of a Visual Studio C++ project (see next figure) without getting off your own development environment.

PolySpace Menu and PolySpace Log Tab in Visual Studio®

PolySpace™ Menu

- Select PolySpace > Launcher to open the PolySpace launcher on the last configuration file updated in Visual.

Note The consistency is not checked with the current project and a warning message is always displayed. The “cfg” file could not correspond to “cfg” file of the current project.

- Select PolySpace > Spooler to start the PolySpace spooler. This tool is used to manage PolySpace jobs that are performed on remote servers.

  For more information, see “Step 4: Launch PolySpace™ Remotely” on page 2-65.
- Select PolySpace > Viewer to open 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.
- Select PolySpace > Display PolySpace Log to view the progress of an analysis (see below).
- Select PolySpace > Help to open the PolySpace Client/Server for C++ User Guide (PDF format).
- Select PolySpace > About to display the release number of the PolySpace for Visual Studio plug-in.

PolySpace Log Tab

After you launch a verification, the PolySpace Log displays a progress report. Compilation errors are highlighted as links. Click a link to display the file and line number that includes the error.

Click the X in the PolySpace Log toolbar to stop the verification. For a server analysis, this option works only during the compilation phase — before it is sent to the server. However, you can select PolySpace > Spooler and stop the analysis from the spooler dialog.

PolySpace™ Right-Click Menu

You can right-click on files in the Solution Explorer to perform PolySpace Verification (see picture below).

PolySpace™ Verification Menu Option

Select a file, then right-click and select PolySpace Verification to launch a verification on that file.

Performing a File Verification from Visual Studio®

To start your first analysis:

1 Select File > New > Project > New > Project Console Win32 to create a new project space

2 Enter the name CppExample , then save it in an appropriate location. For example: C:\PolySpace\Visual .

Some files and a Project Console Win32 are created.

3 Select the Browse the solution tab.

4 Right-click the project name, then select Add > Add existing element from the pop-up menu.

5 Add matrix.cpp and matrix.h (located in <PolySpaceProduct> /Examples/Demo_Cpp_Long/sources) to the CppExample project .

6 In the Visual Studio Solut ion Explorer, select the matrix.cpp file.

7 Right-click matrix.cpp file, then select PolySpace Verification .

The Easy Settings dialog box opens, allowing you to set parameters including precision of the current an alysis and a result folder (See next figure).

Easy Settings Window

8 Select the basic options for the current class analysis:

- Settings window allows you to select precision (-0/-quick) and level of analysis (-to);
- Parameters window allows you to select:
  - Results directory – (-results-dir).
  - Function called before main – A function, if any, called before all functions (-function-call-before-main)

  - Main generator write variables – The type of initialization for global variables (-main-generator-writes-variables).
  - Class analysis tab – By default, enables the class analysis with default options: the class to analyze (-class-analyzer) and associated options which can change behavior of the analysis (-class-only and -class-analyzer-calls).
  - File analysis tab – Allows you to choose a file analysis with associated option (-main-generator-calls).
  - Main analysis tab – Allows you to chose a partial integration analysis by choosing the name of the “main” (-main).
- Scope window allows you to modify the list of files and classes to verify.

9 Click Execute to start the verification.

You can follow the progress of the verification in the PolySpace Log tab, and later using the PolySpace Spooler if remote launching has been enabled.

Analyzing Classes

You can analyze a C++ class by modifying the scope option in the Easy Settings dialog box.

To analyze a class:

1 In the Visual Studio Solution Explorer, right-click a file, then select PolySpace Verification.

The Easy Settings dialog box opens.

2 Click in the Scope window.

The Select Files and Classes dialog box appears.

3 Select the classes you want to analyze, then click Add .

4 Click Execute in the Easy Settings dialog box to start the verification.

You can follow the progress of the verification in the PolySpace Log tab, and later using the PolySpace Spooler if remote launching has been enabled.

The Configuration File and Default Options

Some options are set by default and som e others are directly extracted from the Visual project and set in the assoc iated PolySpace configuration file.

- The list of Visual options extracted from the project file is:

Visual Option	PolySpace Option
/D <name>	-D <name>
/U <name>	-U <name>
/MT	-D_MT
/MTd	-D_MT -D_DEBUG
/MD	-D_MT -D_DLL
/MDd	-D_MT -D_DLL -D_DEBUG
/MLd	-D_DEBUG
/Zc:wchar_t	-wchar-t-is keyword
/Zc:forScope	-for-loop-index-scope in
/FX	-support-FX-option-results
/Zp[1,2,4,8,16]	-pack-alignment-value [1,2,4,8,16]

- Sources and includes directories (-I) a re also extracted automatically from Visual options.
- Default options passed to the kernel depends of the Visual Studio release: -dialect Visual7.1 (or -dialect visual8) -OS-targetVisual -target i386 -desktop

Standard PolySpace options like -voa, can be set by clicking on the “Launcher” right click menu (or from the PolySpace menu).

It starts the standard graphical interface polyspace-launcher on a particular PolySpace configuration file (with .cfg extension).

Every option selected, will be taken int o account during the analysis, except the list of options set in the “PolySpace Basic Setting [C++]” window.

Analyzing an Entire Project

The launching of PolySpace on an entire project can only be made through the PolySpace Launcher using the “Launcher” command. In this case, the option -main must be set manually.

PolySpace™ UML Link™ RH Product

Getting Started (p. 6-2)	Describes the steps needed to get started using PolySpace™ UML Link™ RH product
PolySpace™ Panel (p. 6-10)	Describes the PolySpace panel user interface
Installing the Integration into an Existing Model (p. 6-17)	Describes how to use the PolySpace integration with an existing model
Other Topics (p. 6-19)	Provides additional information about the PolySpace UML Link RH product

Getting Started

In this section...

"Overview" on page 6-2

"Step 1 – Opening the Example Airbag Model" on page 6-2

"Step 2 – Starting an Analysis" on page 6-3

"Step 3 – The Start Analysis Panel" on page 6-4

"Step 4 – Navigating from the PolySpace™ Results to the Rhapsody ® Model" on page 6-8

Overview

While using Collaborative Model-Dri ven Development, run-time errors can be caused either by design issues in th e model itself or faulty hand written code. These reliability flaws can som etimes be found using code reviews and intensive testing - but these techni ques are time-consuming and costly. PolySpace™ UML Link™ RH software saves you both time and money by performing an exhaustive verification of the code and automatically flagging flaws directly in the original Rhapsody ® model, enabling developers to fix these issues quickly and early during the design process.

This getting started guide takes you through the steps required to analyze a model.

Note The PolySpace plug-in has alread y been integrated into the example model. Before other models can be an alyzed the plug-in may need manually installing into the Rhapsody project di rectory. During the getting started the following conventions will be used: "<PolySpaceInstallCommon>" will refer to the installation location of the PolySpace common folder.

Step 1 – Opening the Example Airbag Model

To open the model and displ ay the PolySpace panel:

1 Open the airbag_CPP.rpy model in <PolySpaceInstallCommon> /PolySpaceUMLLink/example .

2 Select the Packages list in the airbag_CPP Model View.

Right-click AirBagFiles , then select PolySpace Panel from the menu.

The PolySpace Panel opens. The PolySpace Panel is the interface to the PolySpace UML Link RH within Rhapsody.

Step 2 – Starting an Analysis

To start an analysis:

1 Click on the Start button in the PolySpace Panel.

For the first analysis of the model, or if the Rhapsody configuration environment changes, the Build Environment Settings dialog are displayed.

The operating system target (-OS-target) is set automatically from the models environment. When the Linux ® environment is detected the dialect will be set to default (-dialect) and include directory will be configured to use the Linux header files supplied with PolySpace software.

2 Click OK.

Note Make sure that the generated code for the model is up to date before starting an analysis.

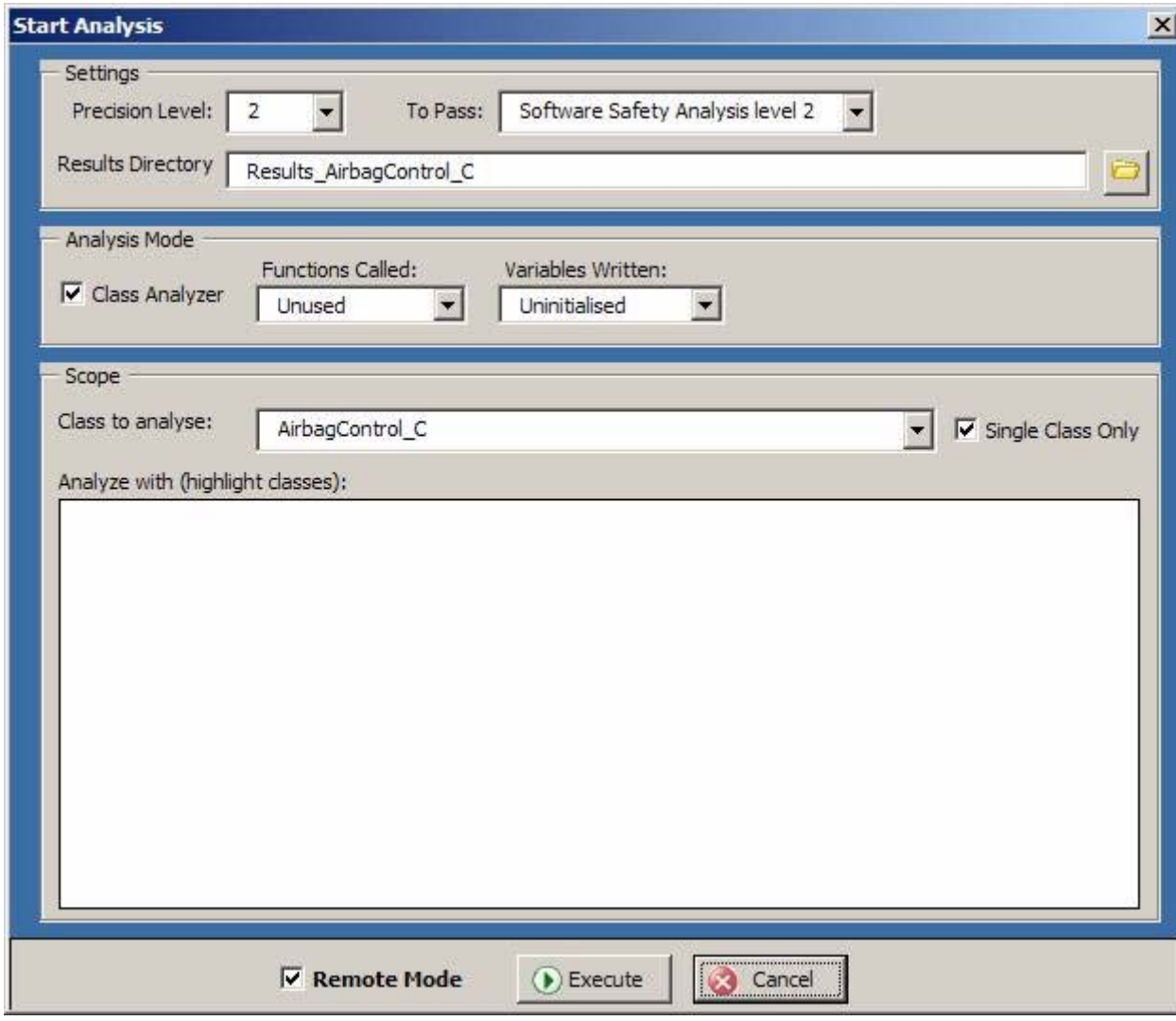
Step 3 – The Start Analysis Panel

The “Start Analysis” panel allows the selection of the type of analysis (Analysis frame) class to in the model analyze (in the Scope frame), and analysis options (Settings frame).

Note The results directory is set automa tically when the class to analyze or analysis mode is changed.

To run the analysis:

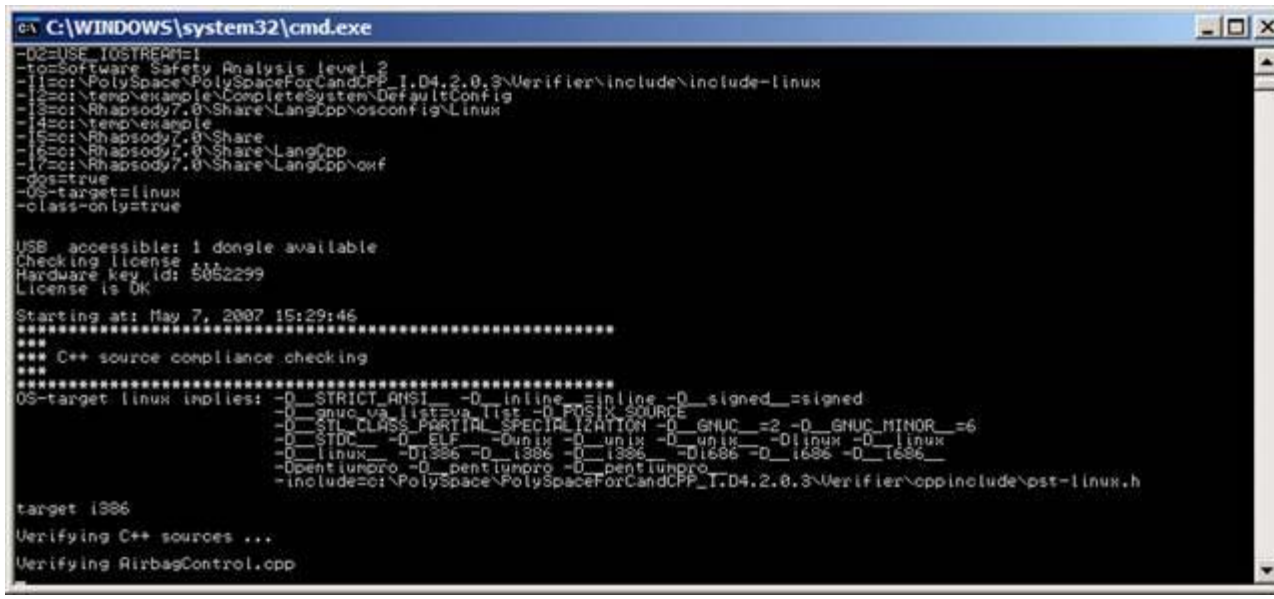
1 Set the options as they are shown below:



Note Mode	If no remote PolySpace Server is available, deselect the Remote option. The analysis will be performed locally.

2 Click the Execute button.

A command window is displayed, sh owing the phases of the analyses performed locally:



Note The settings (size of window, number of lines of history, font etc) for the command window can be changed by right clicking on the window title and selecting properties. It follows standard settings of the “Command Windows” associated with Windows ® OS.

You can follow the analysis on the server (if you selected Remote Mode), by clicking the Manage Analyses button in the PolySpace panel to display the PolySpace Queue Manager interface (or Spooler).

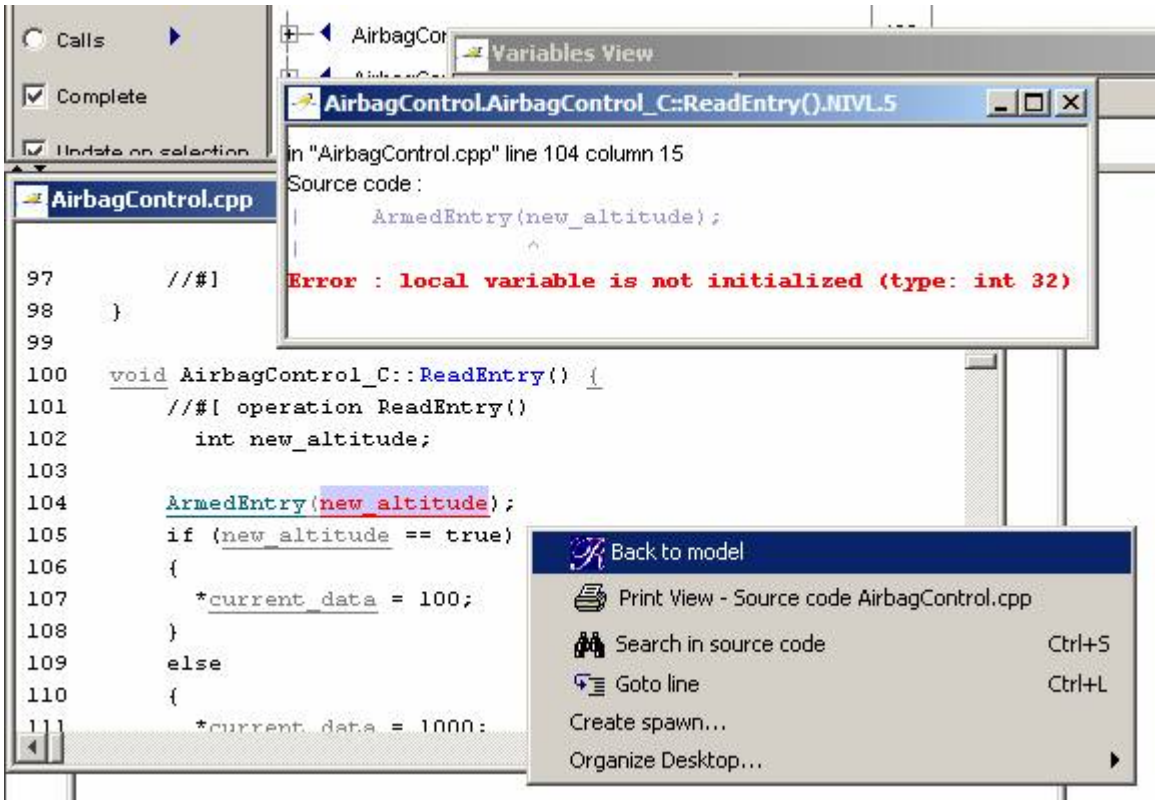
3 When the analysis has completed, down load the results and when prompted open with the PolySpace Viewer.

Step 4 – Navigating from the PolySpace™ Results to the Rhapsody ® Model

To navigate from the PolySpace results to the model:

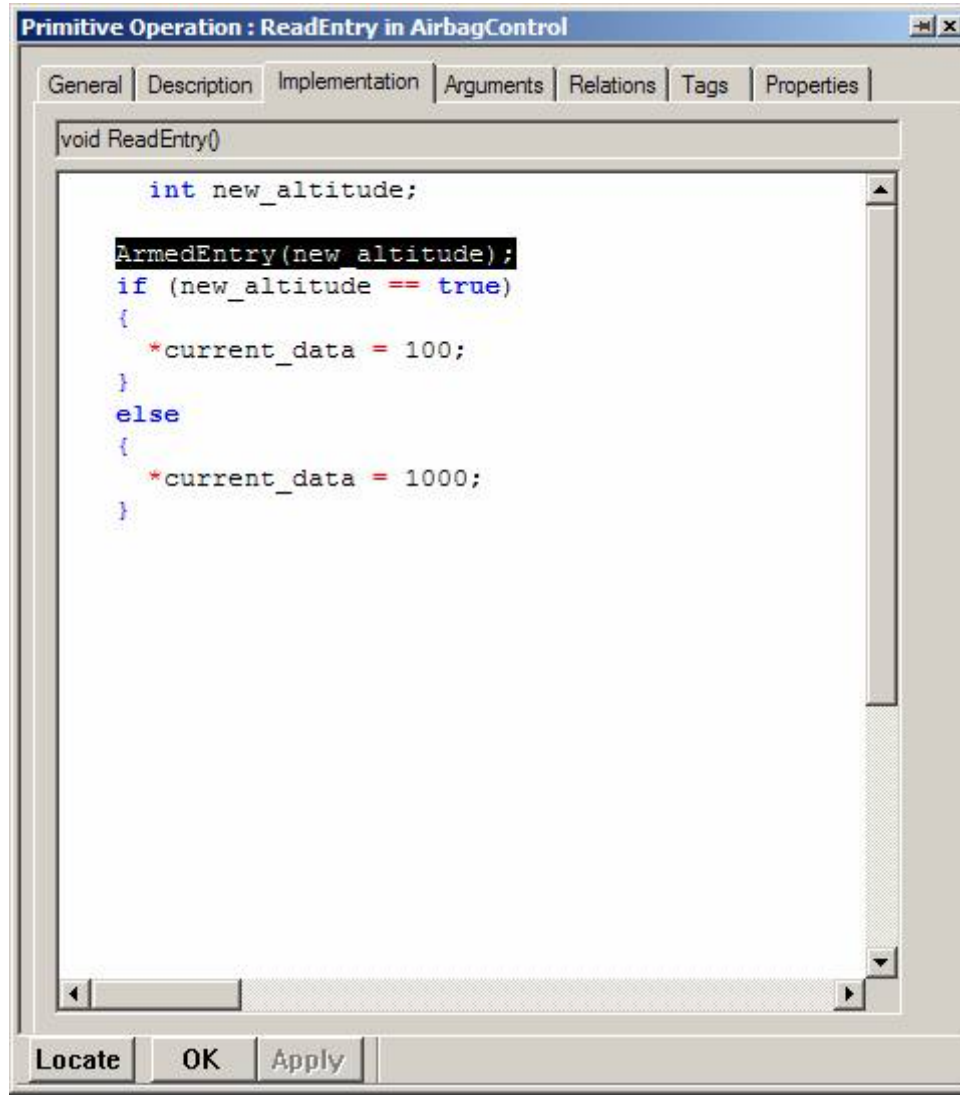
1 Navigate to the first red error, a non initialized variable detected in the model at line 104 of Airbag Control_C .

2 Right click the error, then select Back to Model from the pop-up menu.



Note For the "Back To Model" feature to work Rhapsody must be running with the model open.

This will cause the code to be located within the Rhapsody model. Depending on the Rhapsody configur ation this will either be shown in a popup dialog (such as shown below) or in the code view:



This is the end of the getting started guide.

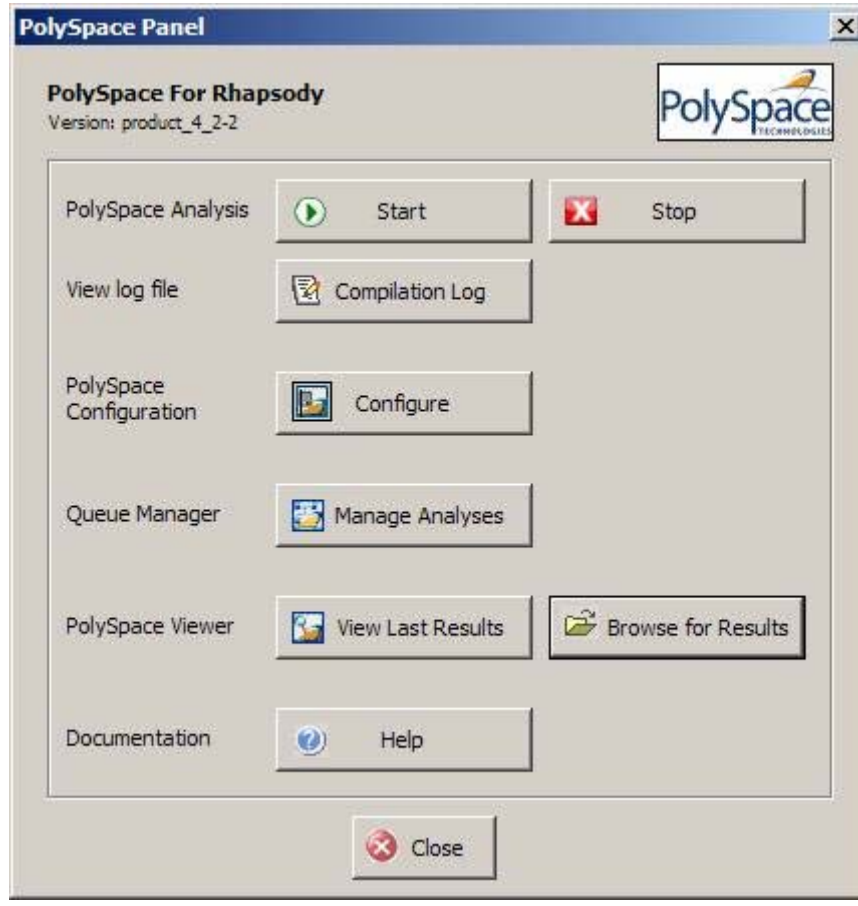
PolySpace™ Panel

In this section...

“Overview” on page 6-10
“Start Button” on page 6-11
“Stop Button” on page 6-14
“Compilation Log Button” on page 6-14
“Configure Button” on page 6-14
“Manage Analyses Button” on page 6-15
“View Results Button” on page 6-16
“Browse for Results Button” on page 6-16
“Help Button” on page 6-16

Overview

The PolySpace™ Panel is the main interface of the PolySpace UML Link™ RH integration with Rhapsody ®. The Panel can be started by right clicking on either a package or a class in the R hapsody “Entire Model View” and selecting PolySpace Analysis.



Start Button

The Start button is used to start an analysis.

Either for the first analysis of the model, or if the Rhapsody configuration environment has changed since the last analysis the Build Environment Settings dialog will be displayed:

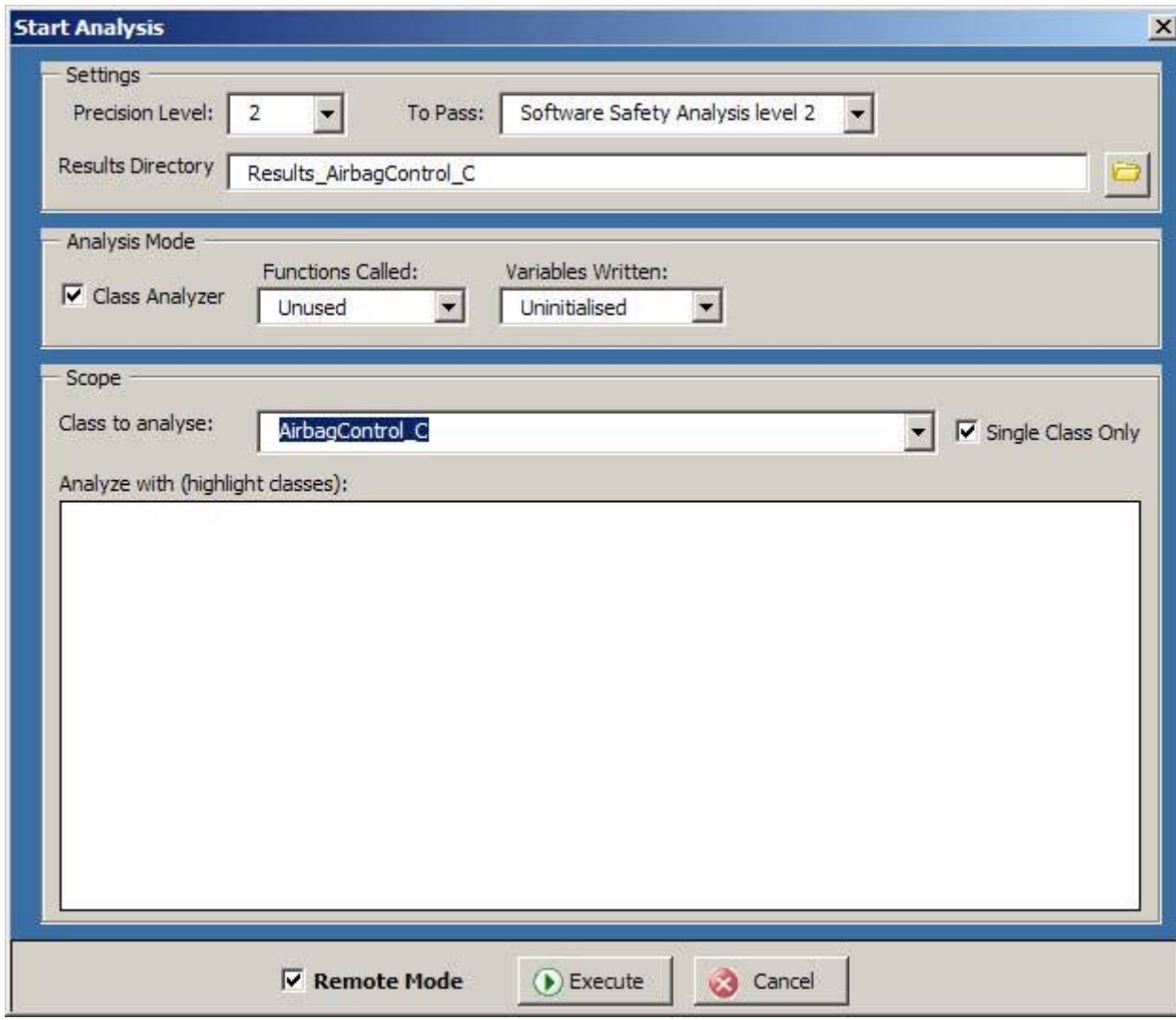
The Operating System Target (-OS-targe t) is detected automatically from the active Rhapsody build environment.

Select the appropriate C or C++ dialect an d the location of the include files for the compiler. If more than one include di rectory is required this can be added later using the “Configure” option on the PolySpace Panel.

Note If the visual OS-Target is detected and a PolySpace supported version of the Microsoft ® Visual C++ ® compiler is installed the “Dialect” (-dialect) and “Include Directory” fields will be automatically completed. This also applies if Linux ® is detected as the -OS-Target, the dialect and include directory will be configured to process the header files from the PolySpace ™ Client™ for C/C++ product directory.

Start Analysis Dialog

When you select OK in the Build Environment Settings dialog, the Start Analysis dialog is displayed:



From the Start Analysis dialog:

- Select the class to analyze from the sc ope section. The results directory is automatically set according to the na me of the selected class, but can be overwritten once the class to analyze has been selected.

- Select Execute to start the analysis. If Remote Mode is selected the analysis will be sent to the PolySpace ™ Server™ for C/C++ server at the end of the compilation phase.
- The Settings section allows setting of the analysis precision, the number of passes of the analysis to perform and the results directory.
- The Analysis Mode section allows configuration of the type of analysis to perform. The options are either to use the “Class Analyzer” to analyze individual classes, or, without in which case a valid “main” function needs to be present in the code. To analyze multiple classes at the same time deselect the “Single Class Only” option, and highlight the classes to analyze in the list. The control and shift keys can be used to control the selection of classes from the list.

Stop Button

The client based phase (compilation) of the analysis can be stopped by clicking on the Stop button.

For analysis running on the PolySpace Server for C/C++ product use the “Queue Manager” to control the jobs.

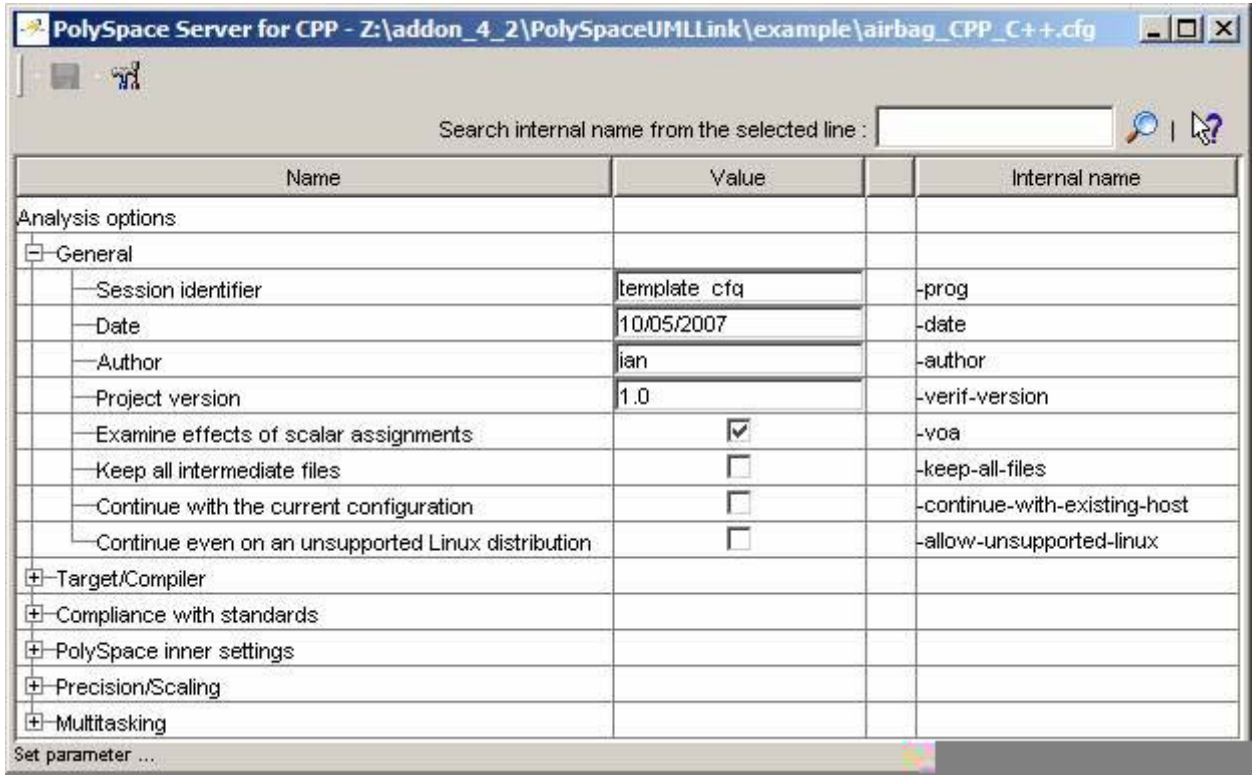
Compilation Log Button

The latest compilation log can be viewed at any time by clicking on the “Compilation Log ” button.

Configure Button

The configure button displays a cut-do wn version of the PolySpace Launcher. From this interface advanced PolySpace options can be configured. Also when required extra source co de compilation parameters can be entered.

Click the disc button in the top left corner to save the configuration.



Note The PolySpace integration extra cts the include directories and compilation flags from the current build environment. In many cases no further configuration other than that r equested by the “Build Environment” dialog should be required for a standard analysis.

Manage Analyses Button

To download the results from a PolySpace Server or follow the progress of an analysis running on a Server click o n the “Analysis Manager” button.