PolySpace™ for C++ Documentation

How to Contact The MathWorks

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

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

508-647-7000 (Phone)
508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098
For contact information about worldwide offices, see the MathWorks Web site.

TABLE OF CONTENTS

1. PolySpace documentation set

This document represents all the documentation required to use PolySpace Products, irrespective of whether you are a beginner or an experienced user. It covers both PolySpace Client and PolySpace Server.

Are you looking to analyse

- One class?
  - Do you want to perform your first 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 provides?
- A class coming from project using the Microsoft Visual Studio .NET IDE?
- A class analysis taking place on a server, and do you want access the queued analysis?
- Analyses code generated from UML models using PolySpace UML Link RH**?**

Detailed contents

- PolySpace Installation. Please refer to `PolySpace_installation_guide.pdf` and to `PolySpace_license_installation_guide.pdf` located on the CD-ROM (in `<CD-ROM>\Docs\Install`).
- "Getting Started" explains how to get started with PolySpace. It explains the principles of the tool, describes the installation procedure, and explains how to use the product with reference to some simple scenarios.
- "Setting up an analysis" details all features of PolySpace which are relevant when preparing to analyse 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).
- "PolySpace and 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.
- The PolySpace C++ add-in for Visual Studio provides automatic source code verification and bug detection in source code developed inside the Visual IDE.
- 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.

- "Reviewing results" details all features of PolySpace 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 colours to look at, and explaining how to find bugs efficiently.
- "Advanced setup" includes options description for PolySpace, hints and tips for quicker PolySpace Verifier analyses, and a complete description of those features which are used in order to launch a PolySpace analysis.

2. Getting started

Related subjects :

2.1. General Requirements

Related subjects :

2.1.1. 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 constraints are described as follows:

- The installation of PolySpace products takes around 5 minutes (see the complete installation guide *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.

2.1.2. Installation Guide

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

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

1. *PolySpace Client* for C/C++ analysing single class.
2. *PolySpace Server* for C/C++ analysing classes or composite analysis.
3. *PolySpace Viewer* is the graphical user interface to explore the results computed by PolySpace Client or PolySpace Server.
4. *PolySpace Spooler* is the graphical interface to manage analysis sent in remote.

Please refer to PolySpace installation manual for installing the PolySpace products.

2.1.3. Structure of this document

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

PolySpace Launcher
Shortcut
2 KB

PolySpace Viewer
Shortcut
2 KB

PolySpace Spooler
Shortcut
2 KB

This Getting Started will focus on the following two exercises and three steps using The PolySpace Client and the Viewer:

- In [Step 1](#) we will analyze a simple class in "`training.cpp`" by using the class analyzer available in PolySpace Client.
- In [Step 2](#) we will describe the capabilities of the class analyzer.
- In [Step 3](#) we will review the results obtained during Step 1 by using PolySpace Viewer
- In the [last step](#), instead of performing a PolySpace analysis locally, we will send it remotely to a server.

2.2. Step 1: PolySpace Client - Setting up and launching an analysis on a single class

This paragraph describes a basic class analysis. It focuses on the analysis of MathUtils class of "`training.cpp`", which is included in the PolySpace installation directory and located at:

`<PolySpaceInstallDir>\Examples\Demo_Cpp_Long\sources\training.cpp.`

The PolySpace analysis process is composed of three main phases:

1. First, PolySpace checks the syntax and semantic of the analyzed file(s). However, as PolySpace is not associated to a particular compiler, **benefits** of this phase are triple for the analysed source code: **ANSI C++ compliance**, **portability** and **maintainability**.
2. Then, PolySpace seeks the main procedure. If none is found, The 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.

Related subjects :

2.2.1. Analysis prerequisites
2.2.2. Setting up a PolySpace Client analysis
2.2.3. PolySpace Client: running the analysis

2.2.1. Analysis prerequisites

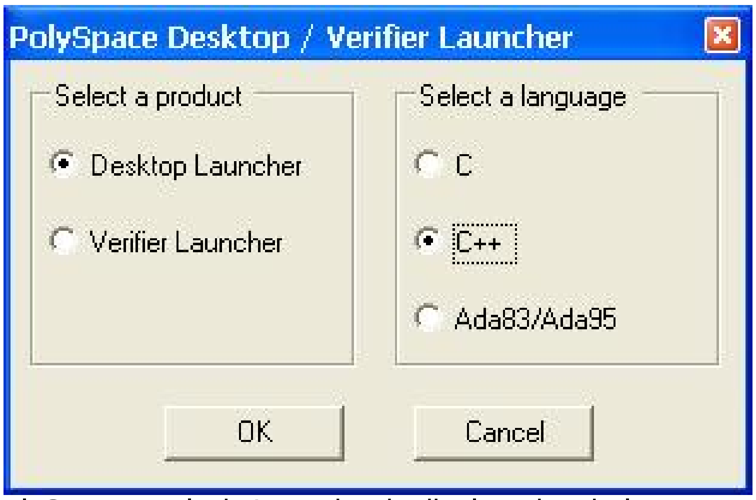
Any analysis requires the following:

- PolySpace Client For C/C++ product and 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 analyse the class `MathUtils` in "`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`".

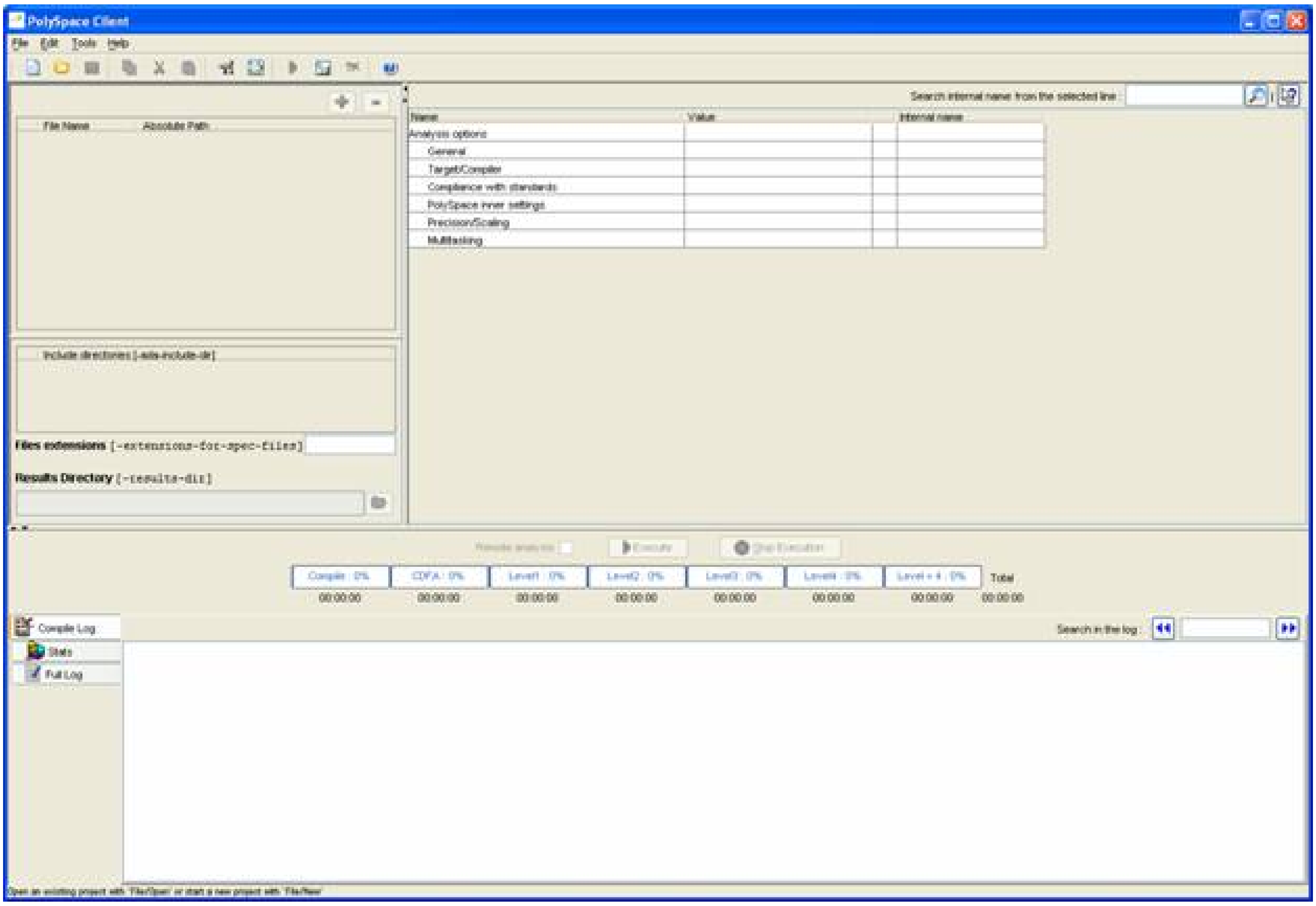
2.2.2. Setting up a PolySpace Client analysis

? Double-click on the PolySpace Launcher icon:

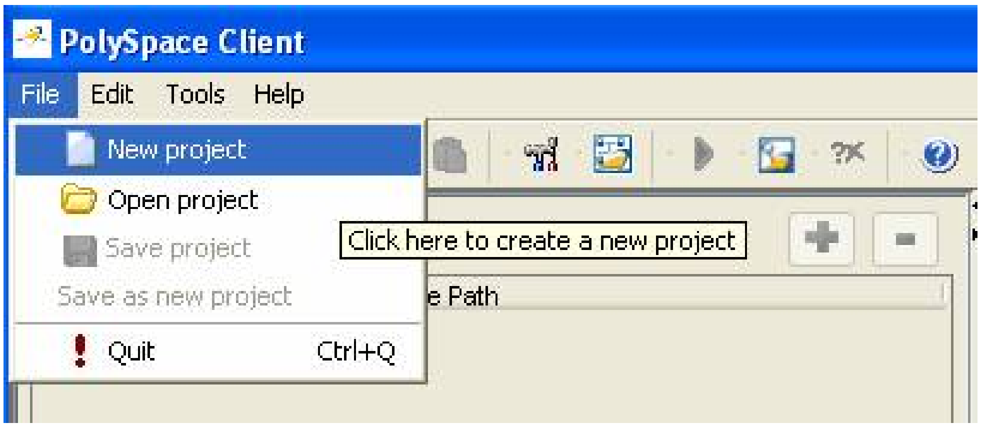
A dialog box window appears proposing to launch one of the following categories of analysis mixing the type of product and the language:



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

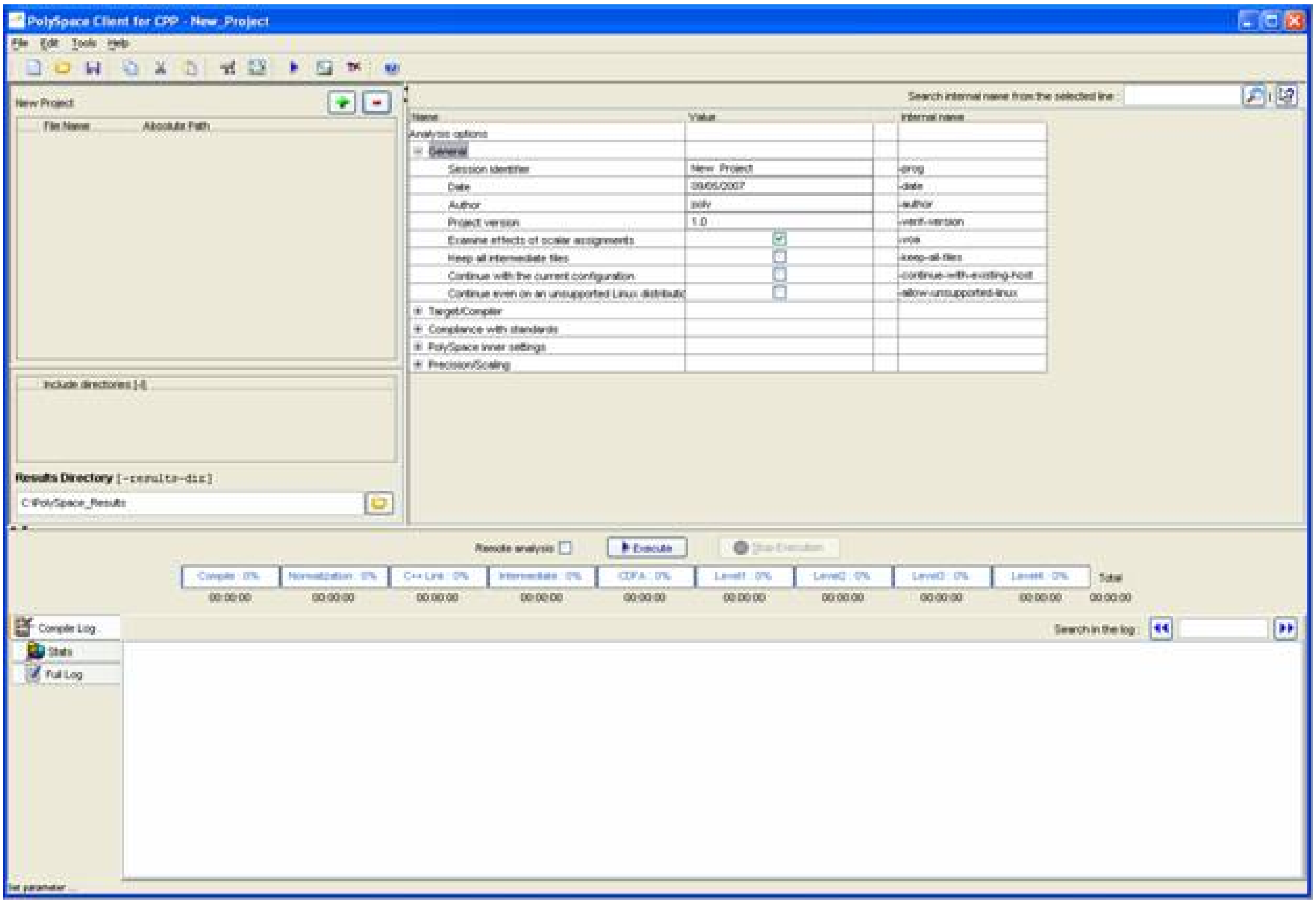


? Click on `File/New Project` to start an analysis:



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

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



Name	Value	Internal name
Analysis options		
General		
Session identifier	New_Project	-prog
Date	30/05/2007	-date
Author	polyv	-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 distributio		-allow-unsupported-linux
Target/Compiler		
Compliance with standards		
PolySpace inner settings		
Precision/Scaling		

Related subjects :

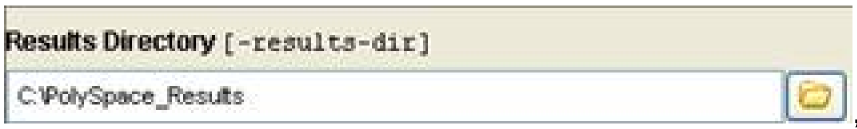
[2.2.2.1. Select results directory](#)

[2.2.2.2. Select the files of the analysis](#)

[2.2.2.3. Select the class to analyse](#)

2.2.2.1. Select results directory

? Start by updating the result directory name by clicking on the browse button:

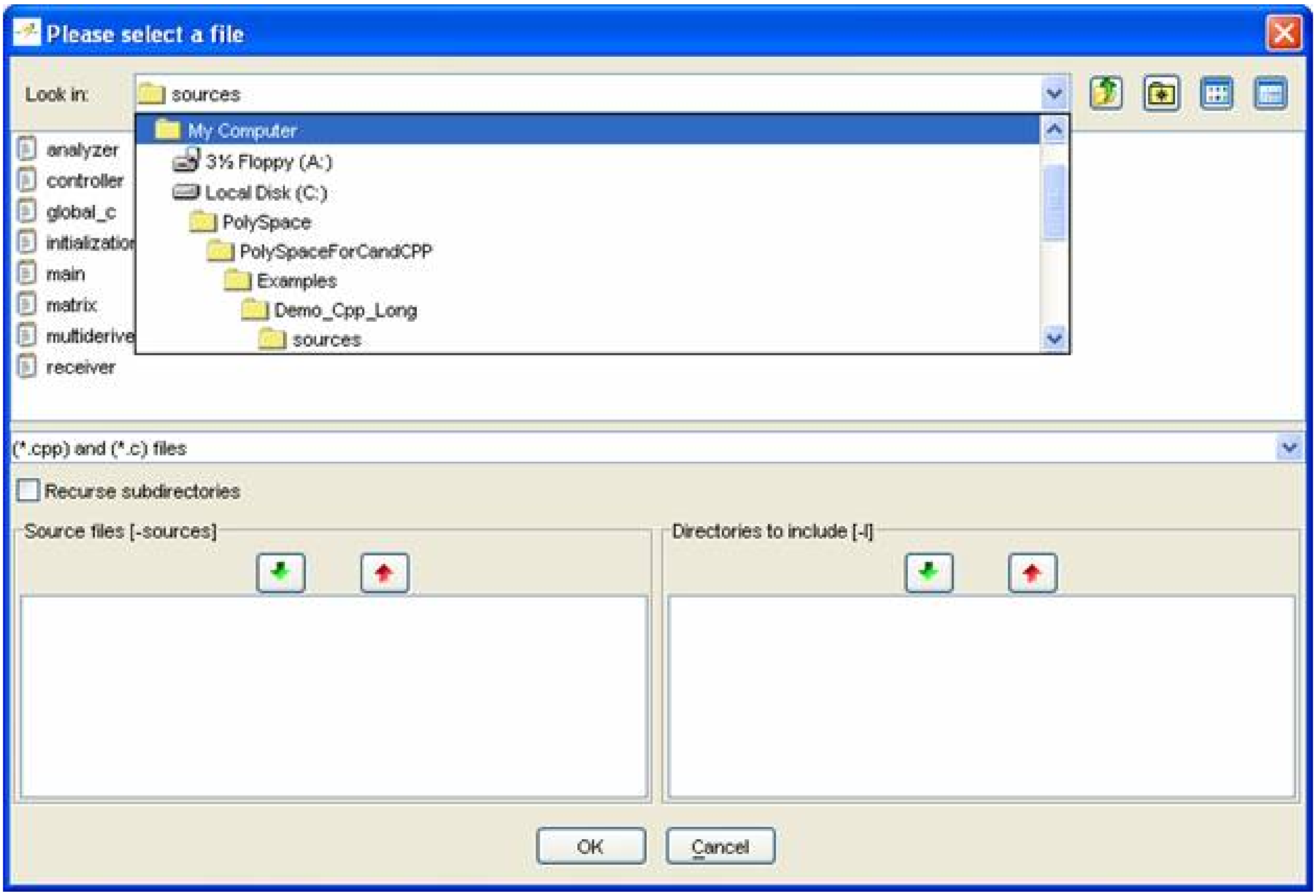


,

This directory is the one where PolySpace Desktop will store the results of the analysis. By default, PolySpace will store results in "`C:\PolySpace_Results`". This is the directory that we will choose for the analysis.

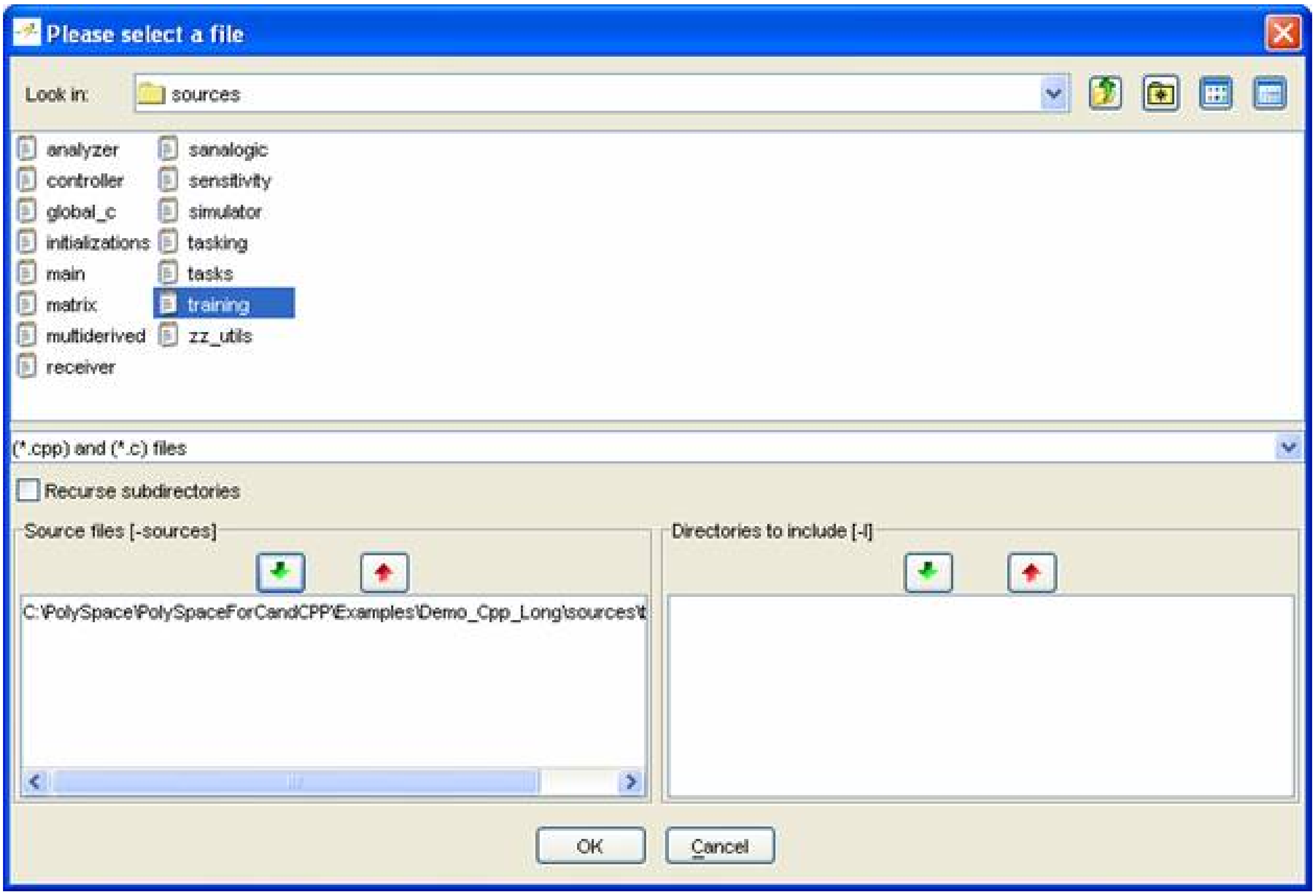
2.2.2.2. Select the files of the analysis

? 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 analyse.



? 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).

? Select "`training.cpp`" and click on in the "`Source files [-sources]`" section (bottom right) of the window. The file is now listed among the source files to be analyzed.



C:\PolySpace\PolySpaceForCandCPP\Examples\Demo_Cpp_Long\sources\t

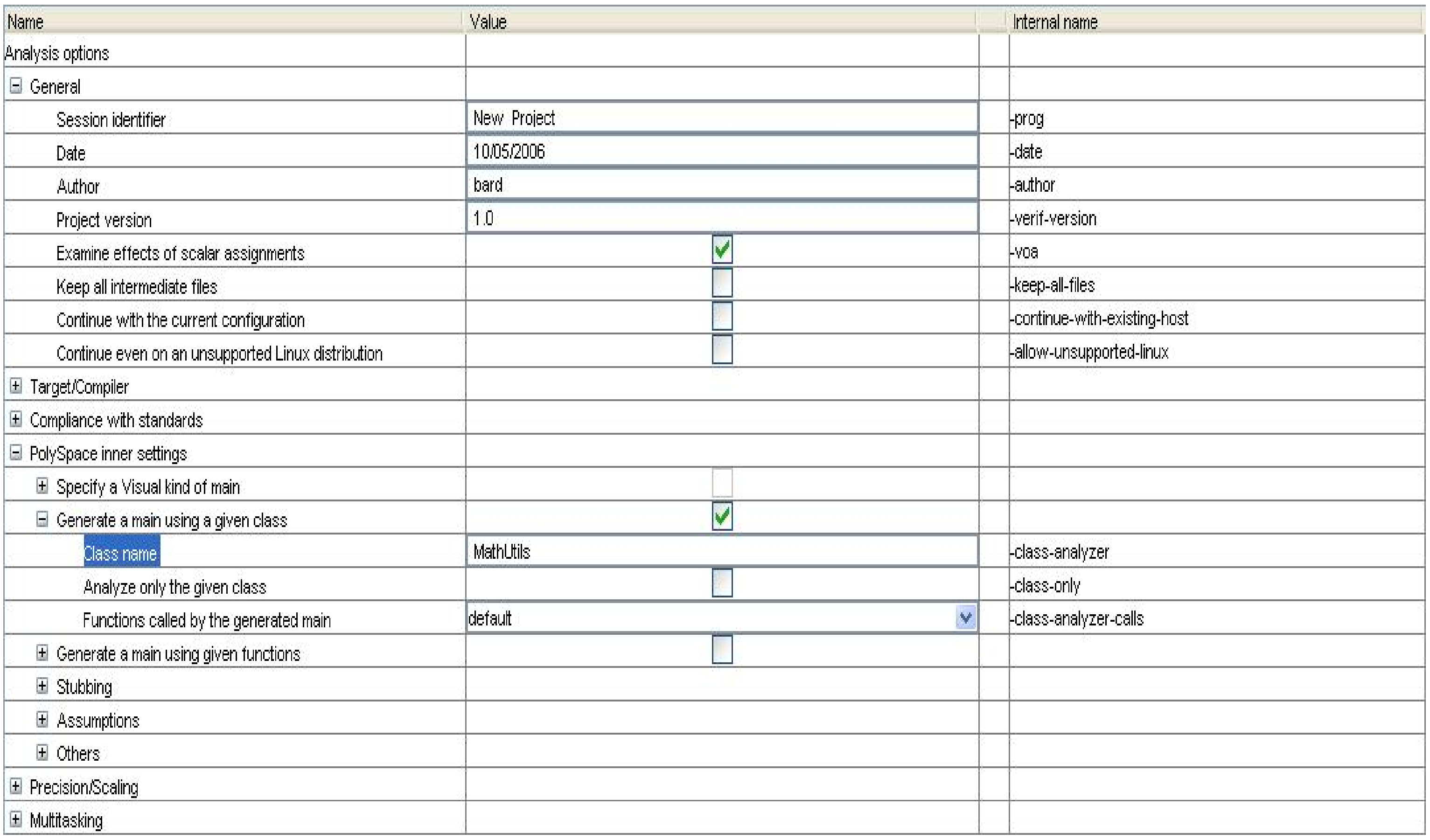
? Click on OK to go back to the "`PolySpace Client for CPP – New_Project`" window.

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

2.2.2.3. Select the class to analyse

? Now, click on and expand the "`PolySpace inner settings`" group.

? 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 analyse (see also step 2). For the needed of this tutorial, please type "`MathUtils`" in the column at the centre 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		

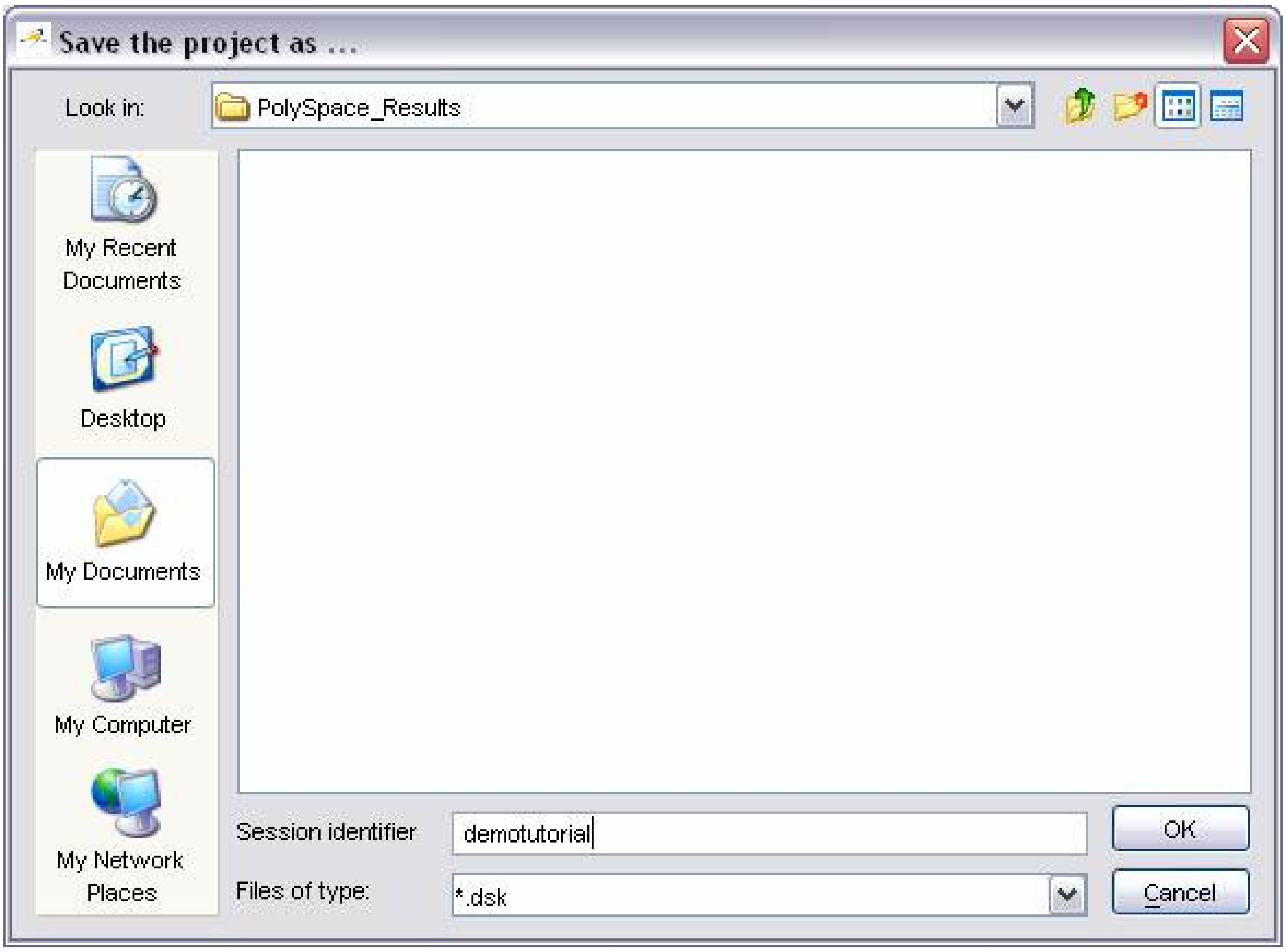
? **It is also recommended** to select the `–voa` option. This option allows you to give some information on each scalar assignment with possible range of values. It can help understanding PolySpace messages.

Notes: When you want to analyse 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.

2.2.3. PolySpace Client: running the analysis

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

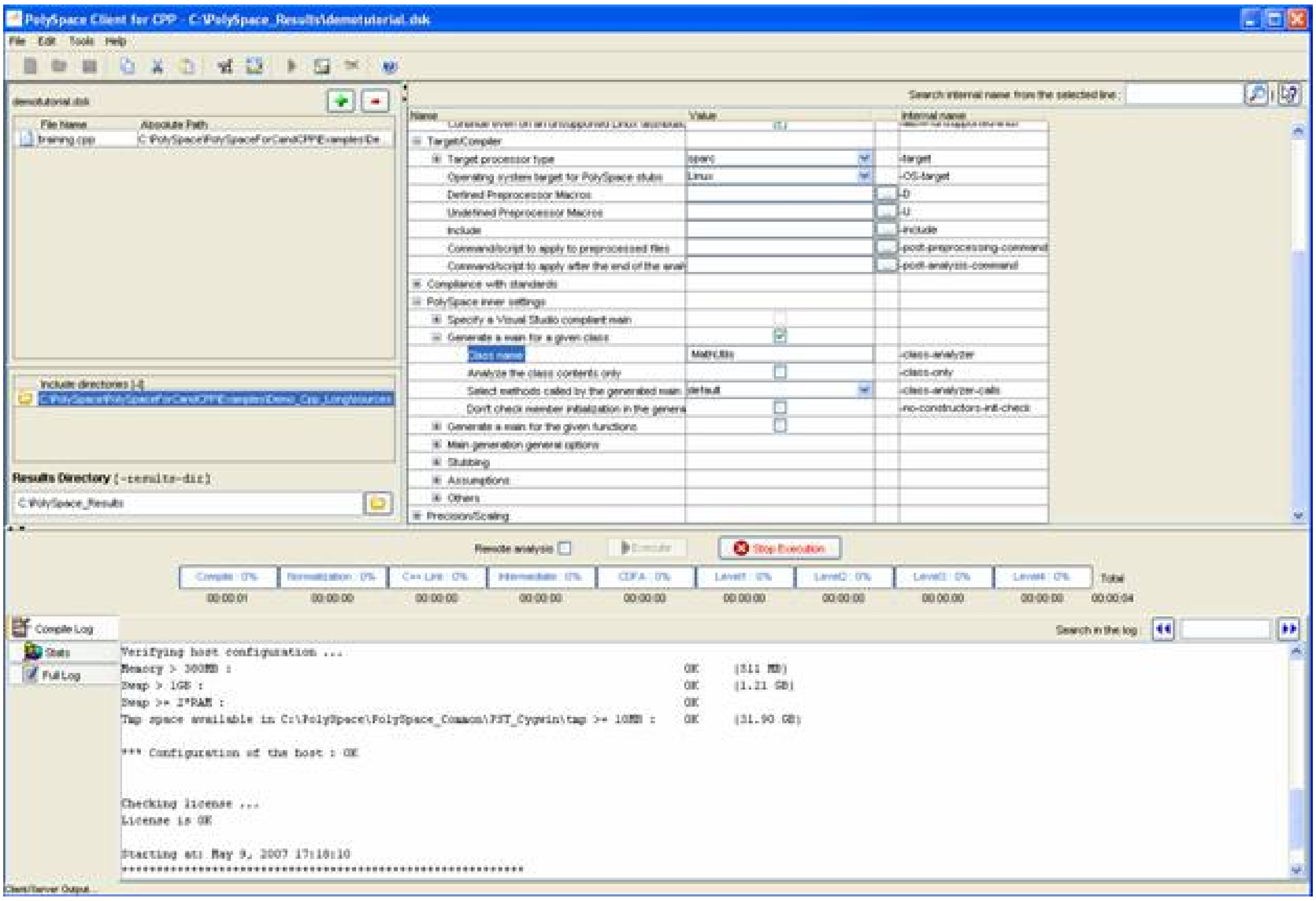
The window titled "`Save the project as` …" opens. You can decide where to store the configuration information related to the analysis. Here, create a file called "`demotutorial`" and save it under PolySpace result directory. The full name of that file will be "`demotutorial.dsk`".



Session identifier: demotutorial

Files of type: *.dsk

? Click on "`OK`" to go back to the "`PolySpace Client for CPP – New_Project`" window and click again on Execute to proceed forward.



Name	Value	Internal name
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
Command/script to apply after the end of the anal		-post-analysis-command
Compliance with standards		
PolySpace inner settings		
Specify a Visual Studio compliant main		
Generate a main for a given class	✓	
Class name	MathUtils	-class-analyzer
Analyze the class contents only	☐	-class-only
Select methods called by the generated main	default	-class-analyzer-calls
Don't check member initialization in the genera	☐	-no-constructors-init-check
Generate a main for the given functions	☐	
Main generation general options		
Stubbing		
Assumptions		
Others		
Precision/Scaling		

Results Directory (-results-dir): C:\PolySpace_Results

```
Verifying host configuration ...
Memory > 300MB :                                                   OK     (511 MB)
Swap > 1GB :                                                       OK     (1.21 GB)
Swap >= 2*RAM :                                                    OK
Tmp space available in C:\PolySpace\PolySpace_Common\PST_Cygwin\tmp >= 10MB :   OK     (31.90 GB)

*** Configuration of the host : OK


Checking license ...
License is OK

Starting at: May 9, 2007 17:18:10
****************************************************************
```

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

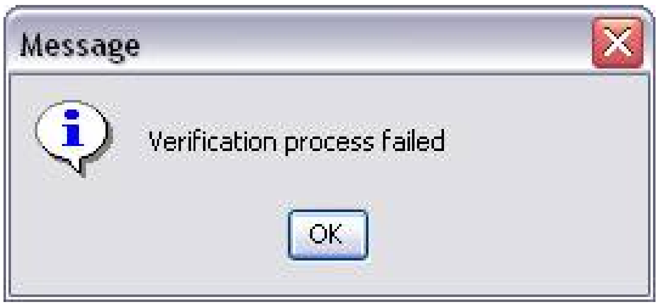
Note: you may press the Stop Execution button - Stop Execution - in order to interrupt the analysis but it is not part of the current tutorial.

Related subjects :

- 2.2.3.1. Parsing errors during preliminary PolySpace analysis stages
- 2.2.3.2. Progression of the analysis
- 2.2.3.3. End of the analysis

2.2.3.1. Parsing errors during preliminary PolySpace analysis stages

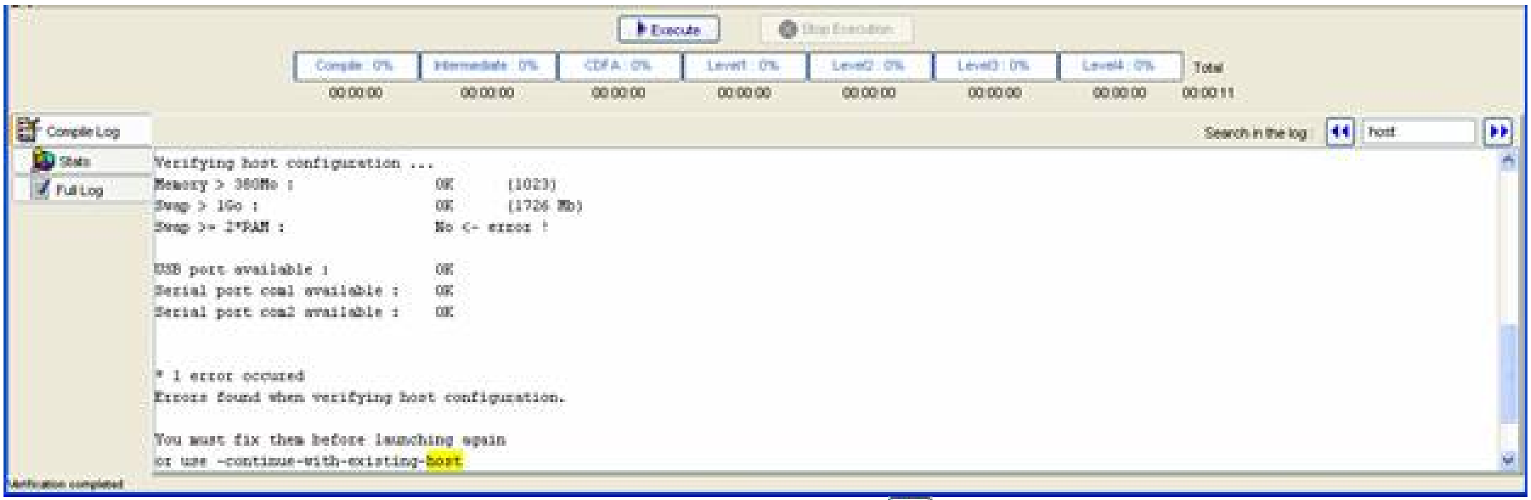
After some checks, PolySpace will show an error message:



Let's 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 meets the minimal hardware configuration requirements described in [section 1](#). Moreover, a message like the following one is displayed in the bottom part of the graphical interface:



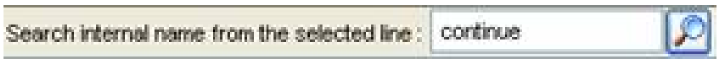
```
Verifying host configuration ...
Memory > 380Mo :                OK      (1023)
Swap > 1Go :                    OK      (1726 Mb)
Swap >= 2*RAM :                 No <- error !

USB port available :            OK
Serial port com1 available :    OK
Serial port com2 available :    OK


* 1 error occured
Errors found when verifying host configuration.

You must fix them before launching again
or use -continue-with-existing-host
```

? Type "`host`" in the "`Search in the log:`" box and click on ⏪ to search if the error corresponds to a hardware recommendation problem.

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

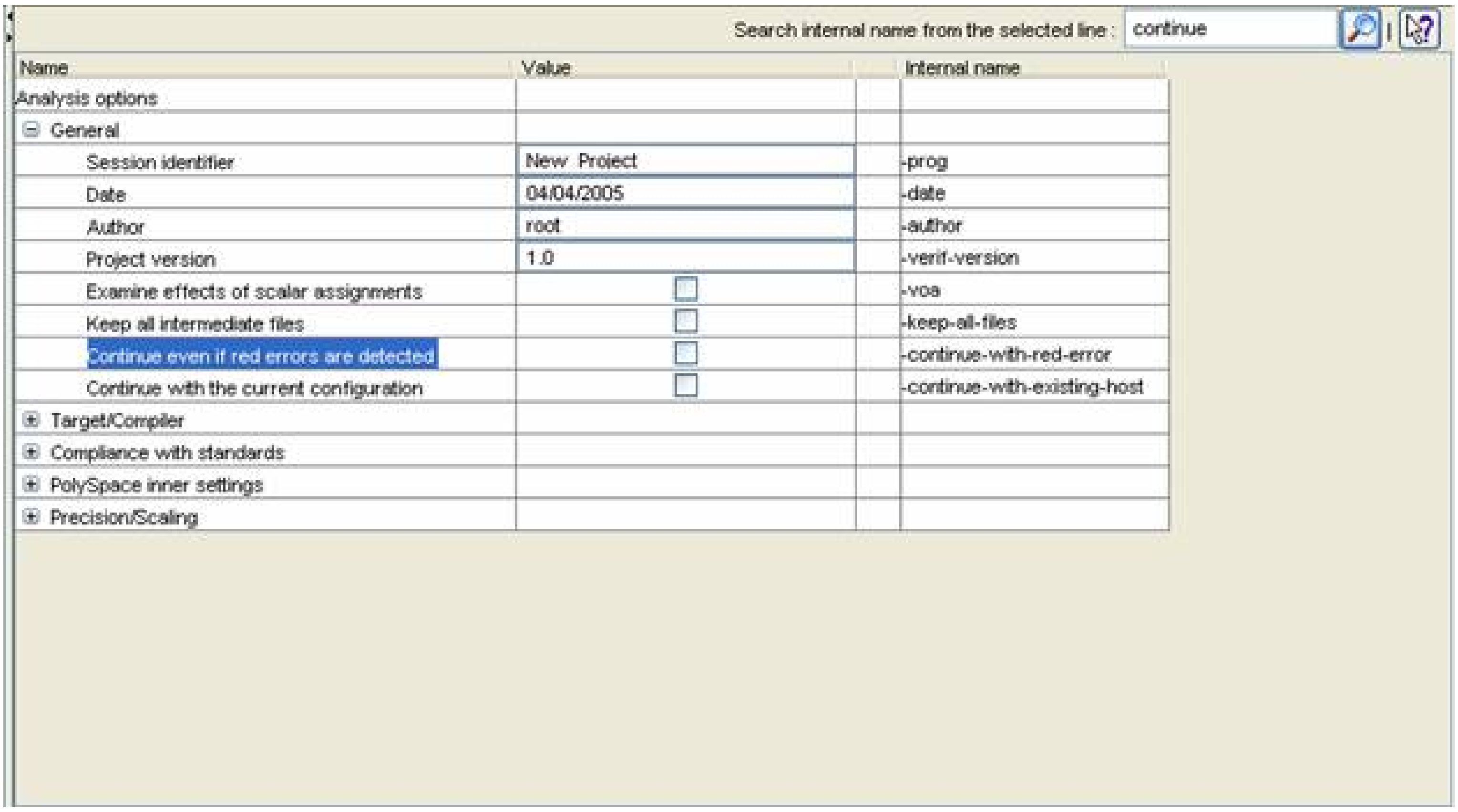
- upgrade your computer to meet the minimal requirements, or
- use the `–continue-with-existing-host` option which overrides the initial check for minimal hardware configuration. To do so, please follow the following steps:

? To set up the `–continue-with-existing-host` option, please type "`continue`" in the Search internal name from the selected line



- top right box -.

? Then click on 🔍. It will show all options containing "`continue`" in the set of options part below:



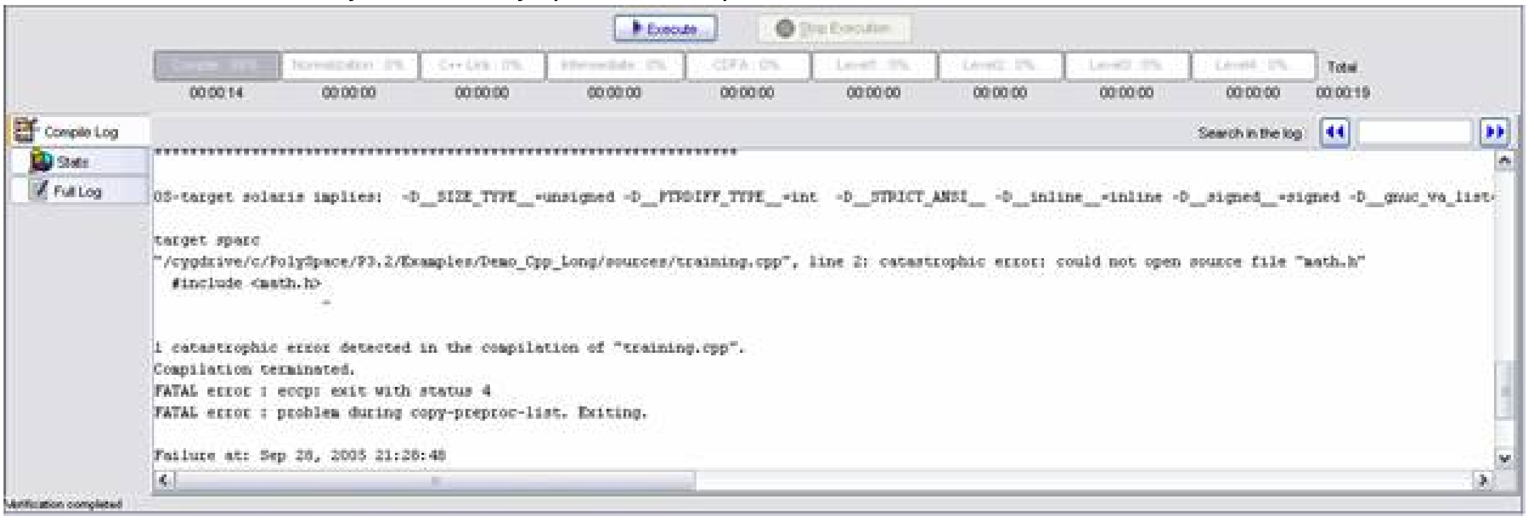
Search internal name from the selected line : continue

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

? Check the box ☑ in the “Value” column that is associated to the “`-continue-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 information about header files.



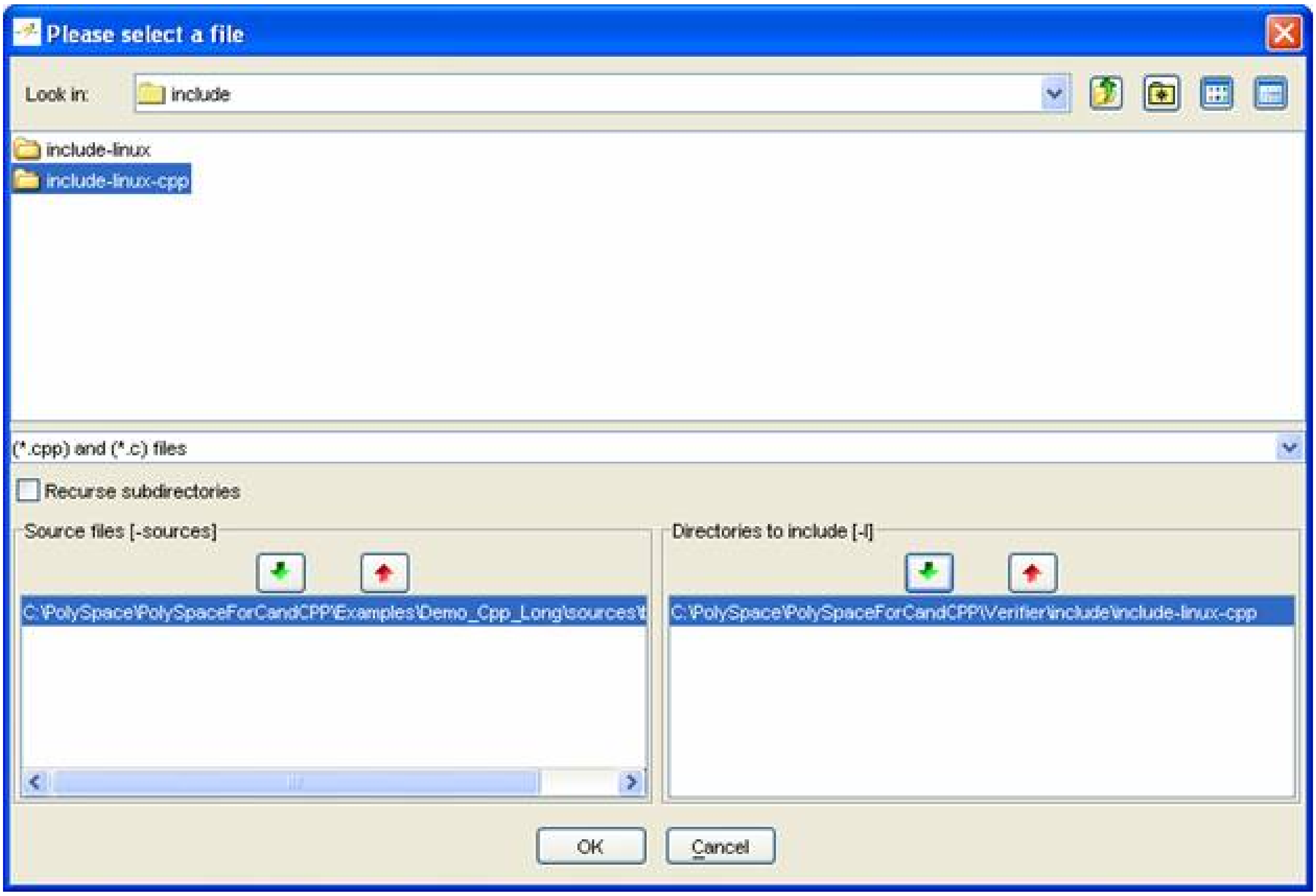
```
00:00:14  00:00:00  00:00:00  00:00:00  00:00:00  00:00:00  00:00:00  00:00:00  00:00:00  00:00:19
```

```
**********************************************************************

OS-target solaris implies:  -D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int  -D__STRICT_ANSI__  -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=

target sparc
"/cygdrive/c/PolySpace/P3.2/Examples/Demo_Cpp_Long/sources/training.cpp", line 2: catastrophic error: could not open source file "math.h"
  #include <math.h>
                   ^

1 catastrophic error detected in the compilation of "training.cpp".
Compilation terminated.
FATAL error : eccp: exit with status 4
FATAL error : problem during copy-preproc-list. Exiting.

Failure at: Sep 28, 2005 21:28:48
```

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

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

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


Please select a file

Look in: include

include-linux
include-linux-cpp

(*.cpp) and (*.c) files

☐ Recurse subdirectories

Source files [-sources]
C:\PolySpace\PolySpaceForCandCPP\Examples\Demo_Cpp_Long\sources\t

Directories to include [-I]
C:\PolySpace\PolySpaceForCandCPP\Verifier\include\include-linux-cpp

OK Cancel

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

? Click on in the “`Directories to include [-I]`” section,

? Then, select “`<PolySpaceInstallDir>\Examples\Demo_Cpp_Long\sources`”, where “`training.h`” is located.

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

Notes:

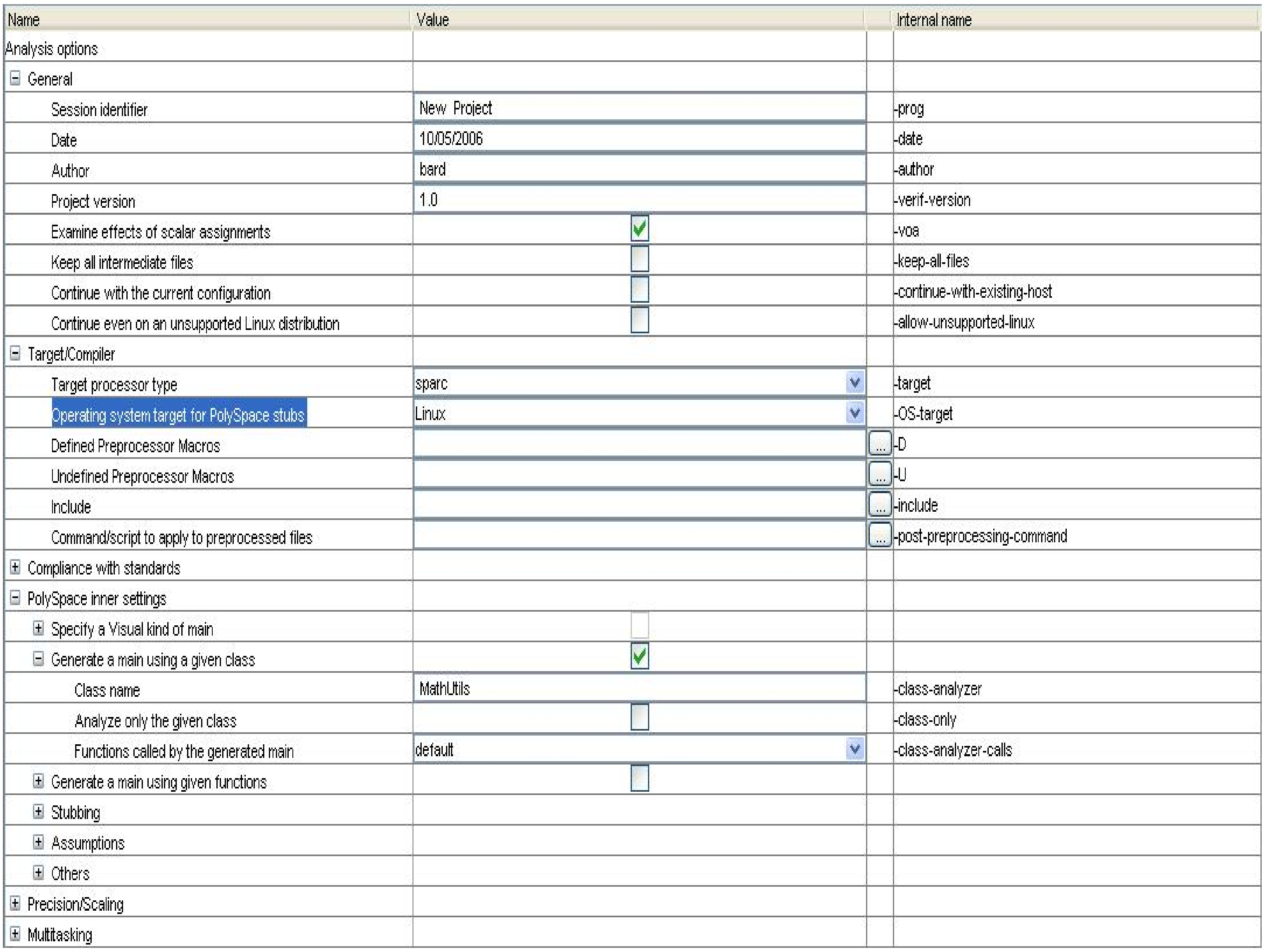
1. Other header file needed “`zz_utils.h`” is also located in same directory.
2. It is also possible to drag a directory and drop it directly in the “`include directories [-I]`” part (top left of PolySpace Desktop) without using the “`Please select a file`” window.

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:

? To set up the `–OS-target Linux` option, please type “`OS-target`” in the Search internal name from the selected line - top right box -.

Search internal name from the selected line : OS-target

? 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			

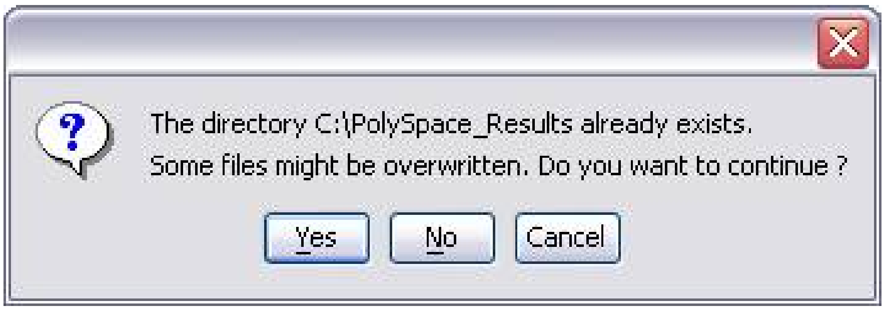
? Then, click on the , allowing to chose `linux` OS target out of some predefined Operating System targets in the list `solaris`, `linux`, `vxworks`, `no-predefined-OS` and `visual`.

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

2.2.3.2. Progression of the analysis

? Click on Execute to restart the analysis.

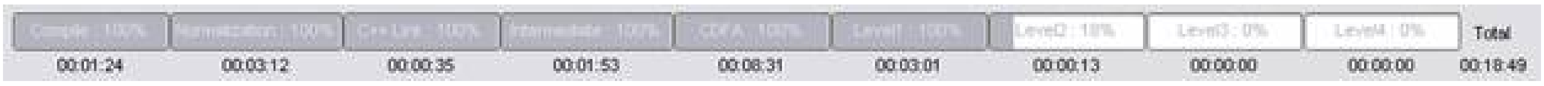
Some results may have already been written in the “`C:\PolySpace_Results`” directory, because of a previous click on Execute. Therefore a window opens to check whether you want to overwrite in this directory or not:


The directory C:\PolySpace_Results already exists.
Some files might be overwritten. Do you want to continue ?

In our example, this is what we want to do. Click on Yes, if it happens.

Note: closing the PolySpace Desktop window will not stop the PolySpace analysis. If you wish to stop it, click on 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 updates the analysis with its current status.

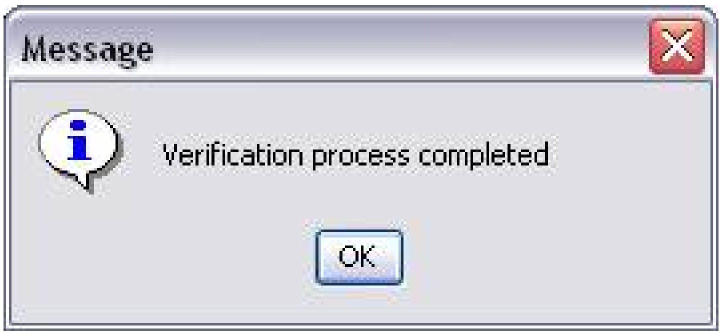
The progress bar allows following the progress of the analysis:



A progress report may be obtained by clicking on Compile Log for the compilation phase, or Full Log for the full analysis in the low level window. Click on 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.). Click on the icon to refresh the summary.

2.2.3.3. End of the analysis

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


Message

Verification process completed

? Click on OK, and go to section "Step 3" of the tutorial to view the results.

Note: You can access the results via the icon in title bar.

2.3. Step 2: Class Analyzer

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 write. The process is very simple: give the class name (see step 1), and the PolySpace Class Analyzer will analyze its robustness:

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

Note: for PolySpace only prototypes of objects (classes, methods, variables, etc.) are needed to analyse 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 some restrictions in the associated paragraph).

Related subjects :

2.3.1. Sources to be analysed

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

2.3.2. Architecture of the generated main

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

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

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

Note:

1. An ASCII file representing the “pseudo” main can be seen in `C:\PolySpace_Results\ALL\SRC\__polyspace_main.cpp`
2. If the class contains more than one constructor, they are called before the ‘`while`’ statement in an ‘`if then else`’ statement. From a PolySpace point of view, this architecture ensures that the analysis will evaluate each function method with every constructor.

2.3.3. Log file

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

You can have the details 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 main is already defined in the files you are analysing. In this case, no other main will be generated, and this one will be analysed. You will receive this 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 analysed even if it does not concern the class given to the -class-analyzer option.

2.3.4. 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.
---------------------------------------------------------
```

Previous Back to table of contents Next

2.3.5. Behaviour of Global variables and members

- **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 behaviour:

- Defined variables: they are initialized to random. Then they follow the data flow of the code to analyse.
- Initialized variables: they are used with the initialized value. Then they follow the data flow of the code to analyse.
- Extern 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 behaviour of a defined variable.

An example below shows behaviour of two global variables:

```

extern int fround(float fx);

// global variables
int globvar1;
int globvar2 = 100;

class Location
{
private:
  void calculate_new(void);
  int x;

public:
  // constructor 1
  Location(int intx = 0) { x = intx;   };
  // constructor 2
  Location(float fx) { x = fround(fx);  };

  void setx(int intx) {  x = intx; calculate_new(); };
  void fsetx(float fx) {
    int tx = fround(fx);
    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 analysing a specific class, variable members of other classes, even members of parent classes, are considered as initialized. They follow the following behaviour:

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 analyse.
2. They are considered as initialized to the 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 analyse.
3. They could be checked as run-time error, if and only if, the constructor is defined and does not initialize the considered member.

An example below shows the result of the analysis of the class `MyClass`. It shows behaviour 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.

2.3.6. Methods and classes specificities

- **Template**

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

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

In the example abowe, we want to analyse template class A with two class parameters T and Z. For that, we have to define a “`typedef`” to create a specialisation of the template, with a specific specialisation 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 `–class-analyzer` option, to analyse 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 analysed. However, it is easy to analyse 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 analyse, 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 father’s class, it is not called in the generated class. In the example below, the class Point derives from the class Location:

```
class Location
{
protected:
  int x;
  int y;
  Location (int intx, int inty);
public:
  int getx(void) {return x;};
  int gety(void) {return y;};
};

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

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 analyse the `Location` class

However, inherited members are considered as volatile if there are not explicitly initialized in the father’s constructors. In the example above, the use of the two members `Location::x` and `Location::y` will be considered as volatile. Indeed, if we analyse the above example in the current state, the method `Location:: Location` (constructor) will be stubbed.

2.4. Step 3: PolySpace Viewer - Exploration of results

This step illustrates how to explore analysis results that were generated by either the PolySpace Client or the PolySpace Server. We review the results of the analysis of "`training.cpp`" performed during Step 1.

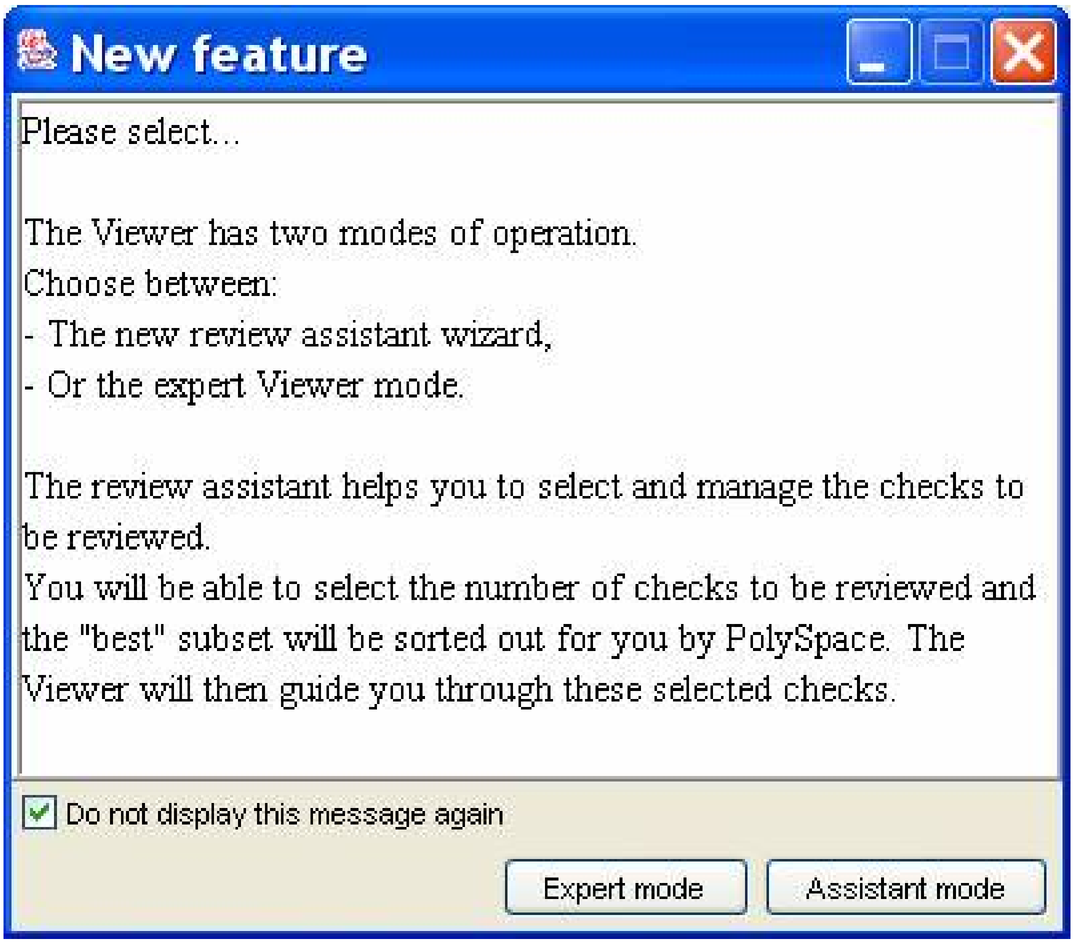
If the OK button has been clicked at the end of the analysis during Step 1, PolySpace Viewer automatically opens results.

Related subjects :

- **2.4.1. Modes of operation**
- **2.4.2. Downlaod results into the Viewer**
- **2.4.3. Analysing of PolySpace results ("training.cpp")**
- **2.4.4. Methodological asssitant**
- **2.4.5. Report Generation**

2.4.1. 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.



Please select...

The Viewer has two modes of operation.
Choose between:
- The new review assistant wizard,
- Or the expert Viewer mode.

The review assistant helps you to select and manage the checks to be reviewed.
You will be able to select the number of checks to be reviewed and the "best" subset will be sorted out for you by PolySpace. The Viewer will then guide you through these selected checks.

[✓] Do not display this message again

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

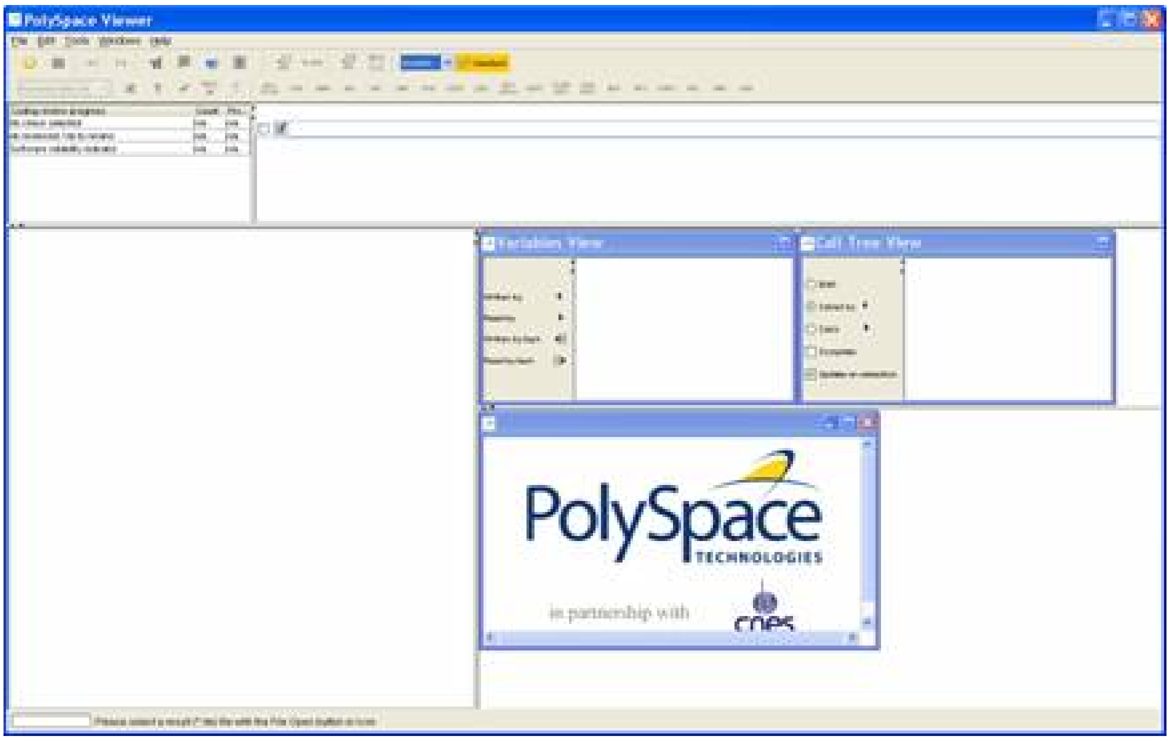
- In "`Expert mode`": The Viewer is opened in a mode where all checks can be seen. The number, the order and the categories of checks can be reviewed can be chosen by the user himself (See next section).
- In "`Assistant mode`": the reviewing rules for a C++ analysis results follows a methodology selected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

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

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

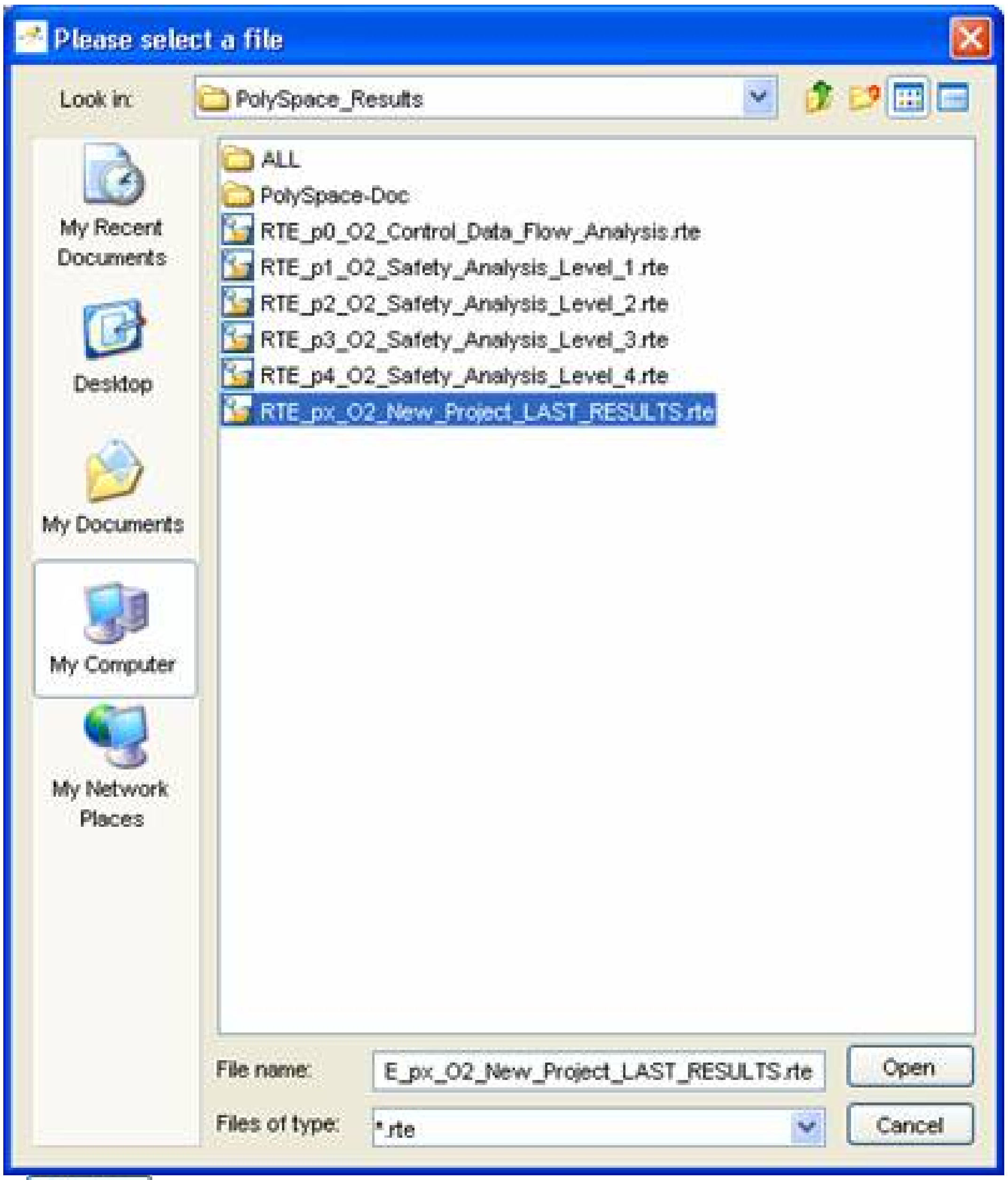
2.4.2. Downlaod results into the Viewer

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



? Click `file>open` to load result files. If you did not perform the analysis, you can still review the results by opening the following file:
`<PolySpace Install Directory>\Examples\Demo_CPP_Long`
`\RTE_px_O2_Demo_Cpp_Long_LAST_RESULTS.rte.` We will focus on “`training.cpp`” Procedural entity.

? Using the “`File>Open`” menu, select the following file located in “`C:\PolySpace_Results`”.



ALL
PolySpace-Doc
RTE_p0_O2_Control_Data_Flow_Analysis.rte
RTE_p1_O2_Safety_Analysis_Level_1.rte
RTE_p2_O2_Safety_Analysis_Level_2.rte
RTE_p3_O2_Safety_Analysis_Level_3.rte
RTE_p4_O2_Safety_Analysis_Level_4.rte
RTE_px_O2_New_Project_LAST_RESULTS.rte

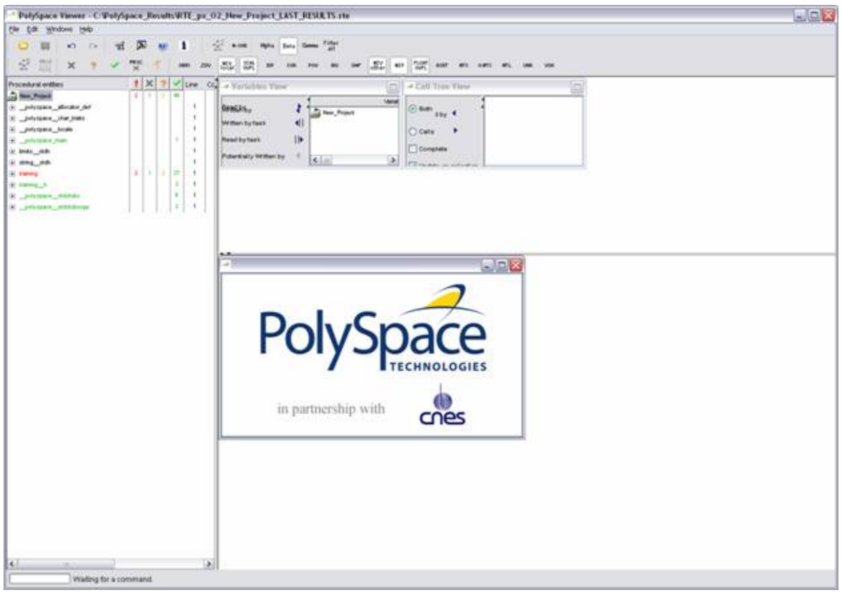
File name: E_px_O2_New_Project_LAST_RESULTS.rte
Files of type: *.rte

? Then click on Open to proceed with further steps

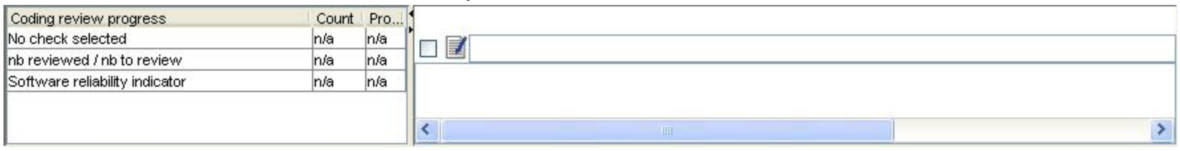
Note: The `RTE_px_O2_<Project Name>_LAST_RESULTS.rte` is a sort of "link" on the best analysis in term of precision. This analysis is represented by `RTE_p4_O2_Safety_Analysis_Level4.rte` file. Lower level files represent lower precision analysis.

2.4.3. Analysing of PolySpace results (“training.cpp”)

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

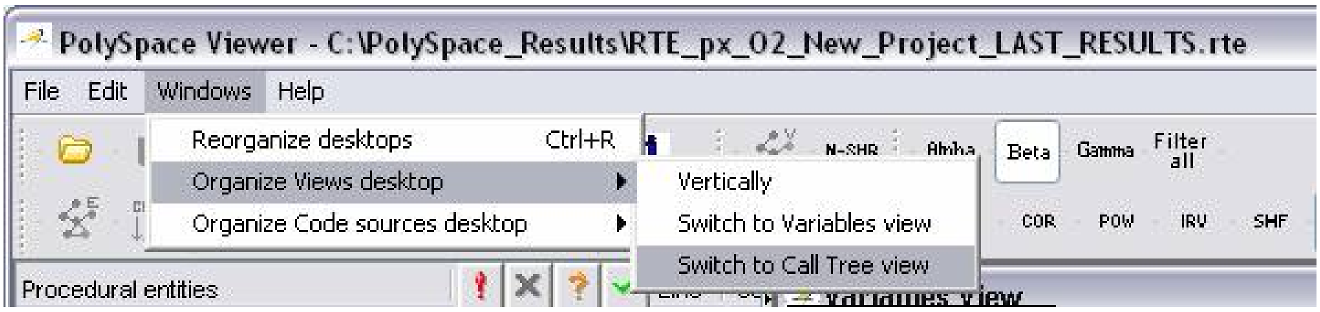


1. On the left is the run-time error view (or RTE View). It displays the list of files analyzed in the “`Procedural entities`” column.
2. In the bottom right area is the source code view with coloured instructions. Each operation checked is displayed using meaningful colour scheme and related diagnostic:
   - 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.
3. The two windows just below the tool bar concern details of a currently reviewed check (when the check has been selected):



Coding review progress	Count	Pro...
No check selected	n/a	n/a
nb reviewed / nb to review	n/a	n/a
Software reliability indicator	n/a	n/a

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



Related subjects :

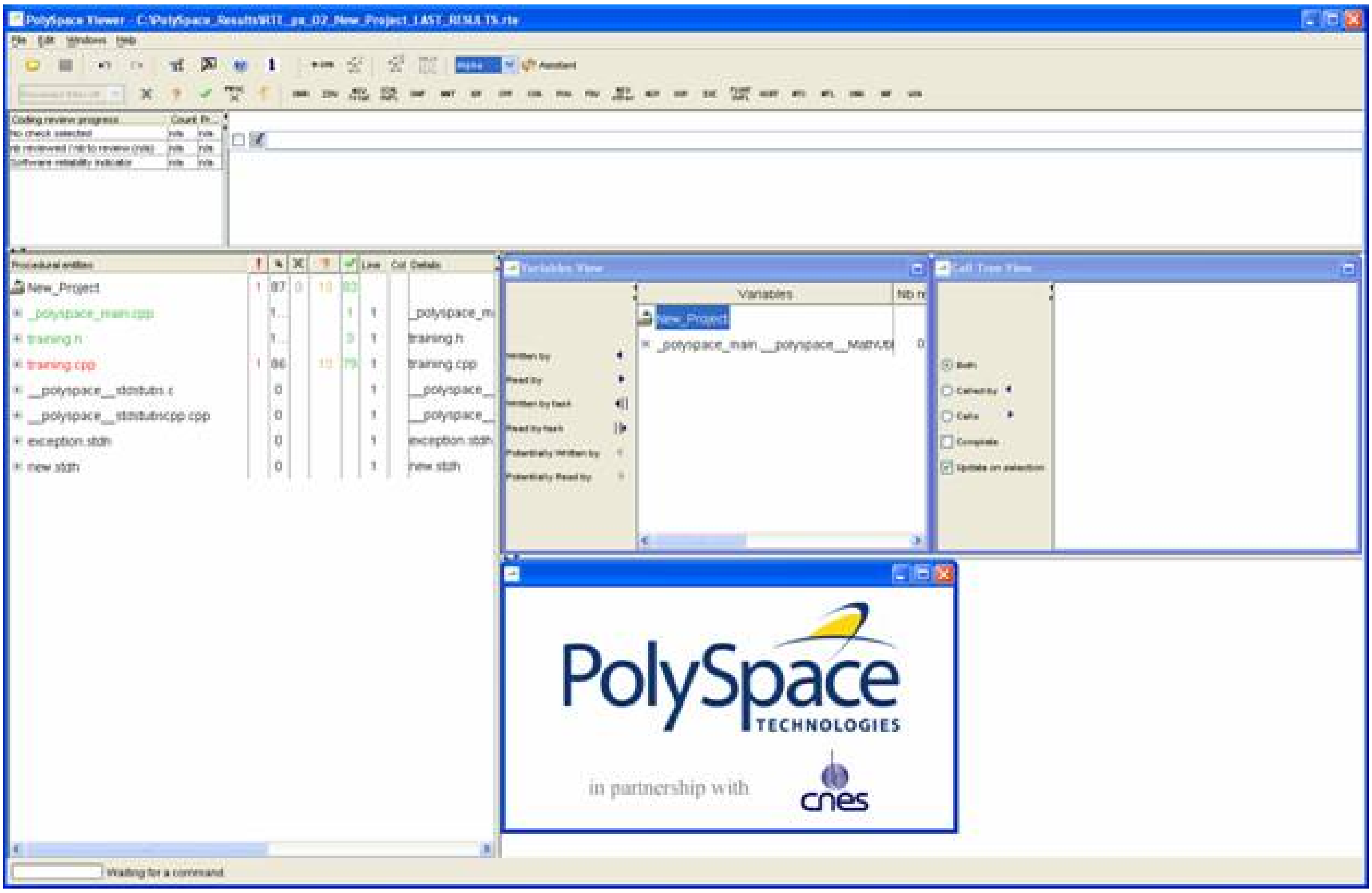
- 2.4.3.1. RTE view
- 2.4.3.2. Colours in the Source code view
- 2.4.3.3. More examples of run-time errors
- 2.4.3.4. Advanced results exploration
- 2.4.3.5. C++ specific checks
- 2.4.3.6. Miscellaneous

2.4.3.1. RTE view

Each file and underlying functions in the “Procedural entities” view or Run-time Errors view (RTE) is colorized according to the most critical error found:

- `exception.stdh`. This file contains no check. This file contains stubs of the `<exception>` template part of the standard 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` 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 analysed with PolySpace.
- `training.cpp`. This file is red. This is the famous “training.cpp” containing the analysed 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).



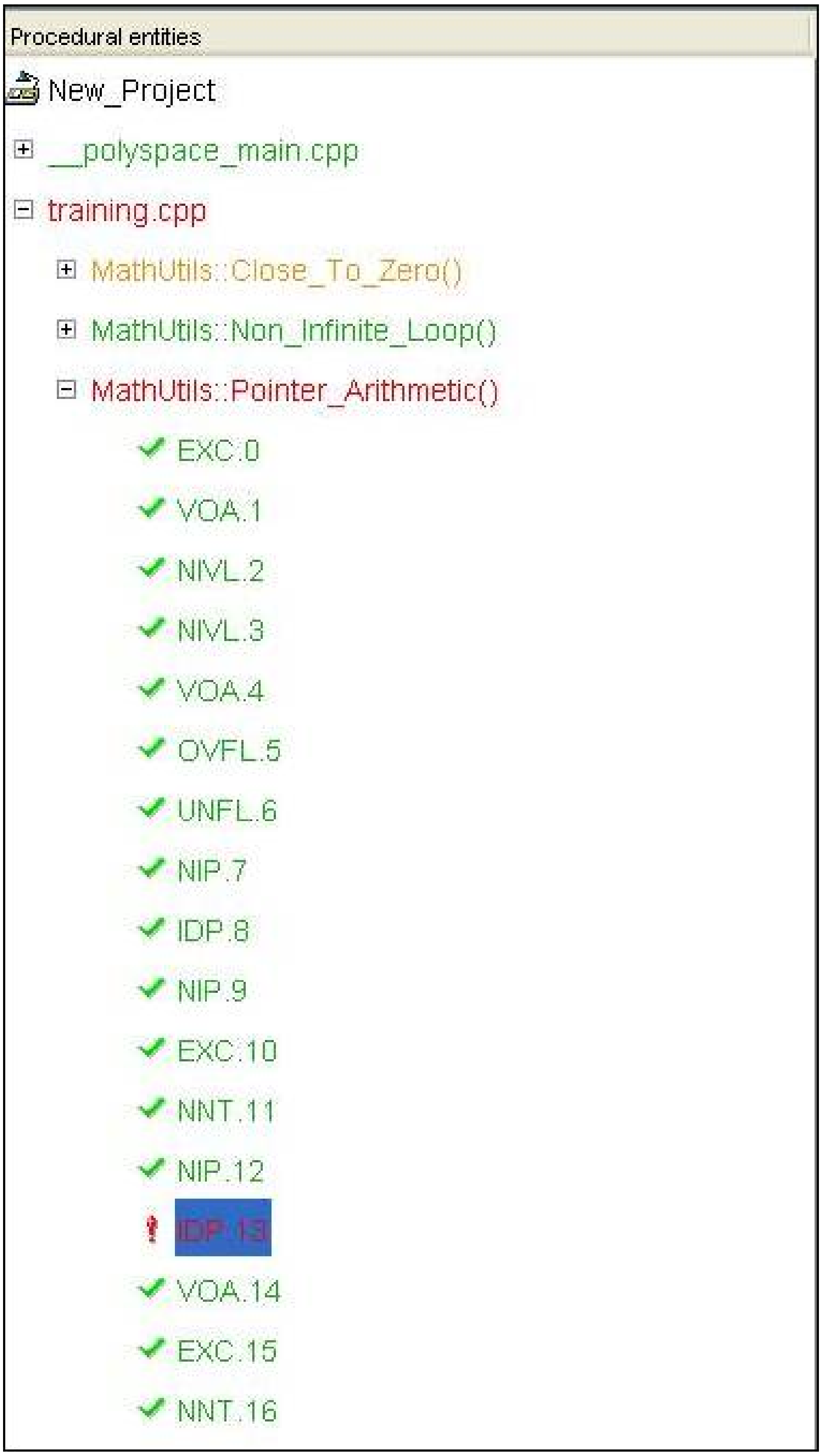
The columns (✎, !, ✓, ✗, ?,…) provide information about run-time errors found in each function:

- The ✎ column indicates the selectivity (level of proof),
- The ! column indicates the number of definite run-time errors or reds,
- The ? column indicates the number of warnings or oranges (that may hide run-time errors that do not occur systematically),
- The ✓ column indicates the number of safe operations or greens
- The ✗ column indicates the number of unreachable instructions or grey code sections.

Let's have a look at some error found by PolySpace in "`training.cpp`".

First example of runtime error found by PolySpace: Memory Corruption

? 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_Arithmetic()`" function.


Procedural entities

New_Project

__polyspace_main.cpp

training.cpp

MathUtils::Close_To_Zero()

MathUtils::Non_Infinite_Loop()

MathUtils::Pointer_Arithmetic()

✓ EXC.0

✓ VOA.1

✓ NIVL.2

✓ NIVL.3

✓ VOA.4

✓ OVFL.5

✓ UNFL.6

✓ NIP.7

✓ IDP.8

✓ NIP.9

✓ EXC.10

✓ NNT.11

✓ NIP.12

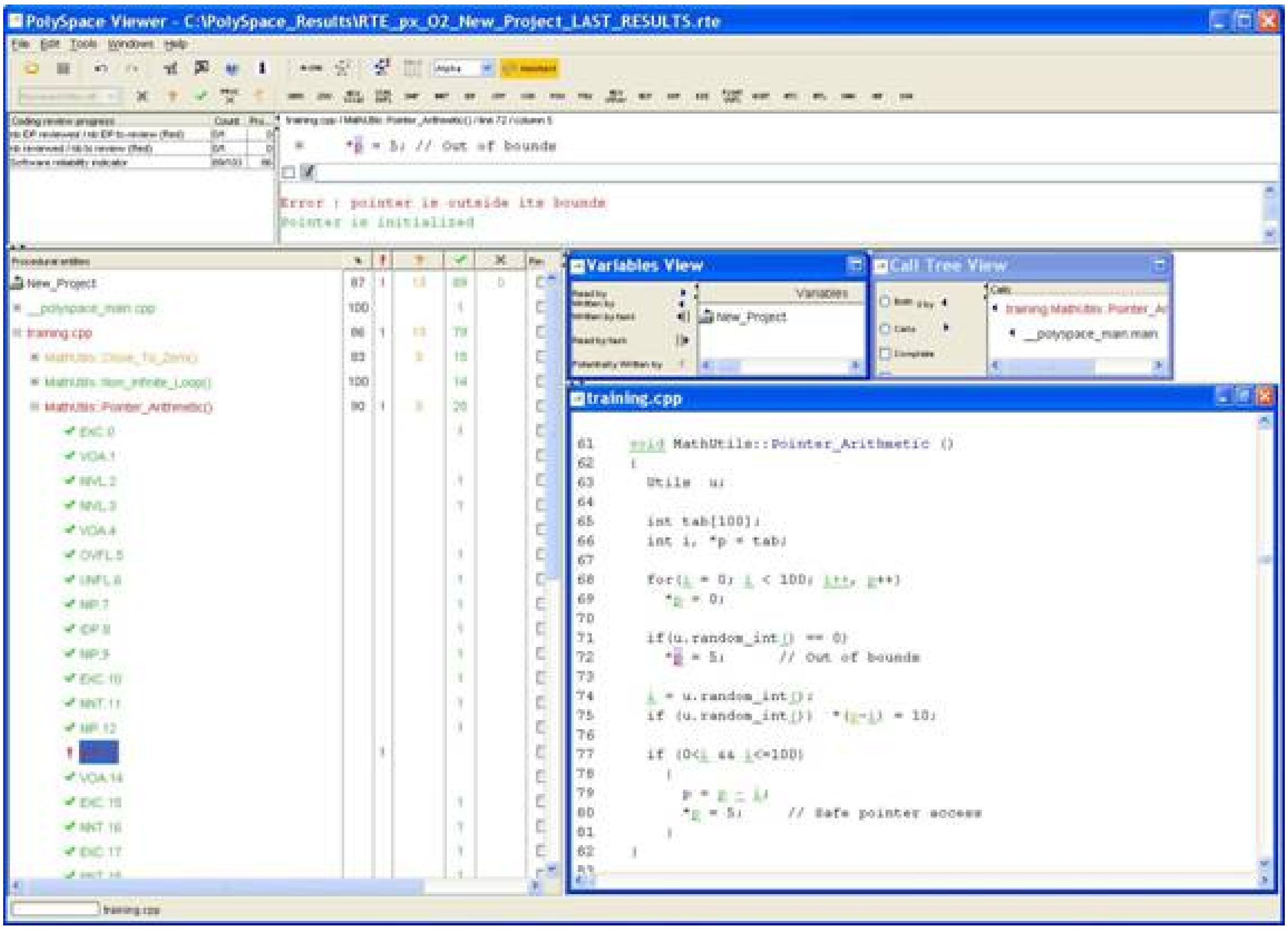
! IDP.13

✓ VOA.14

✓ EXC.15

✓ NNT.16

? Click on the red "`IDP.13`" item - which stands for **I**llegal **D**e-referenced **P**ointer -, to precisely locate this error in the source code. The bottom right section is updated showing the location of the "`IDP.13`" item.


PolySpace Viewer - C:\PolySpace_Results\RTE_px_02_New_Project_LAST_RESULTS.rte

training.cpp / MathUtils::Pointer_Arithmetic() / line 72 / column 5

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

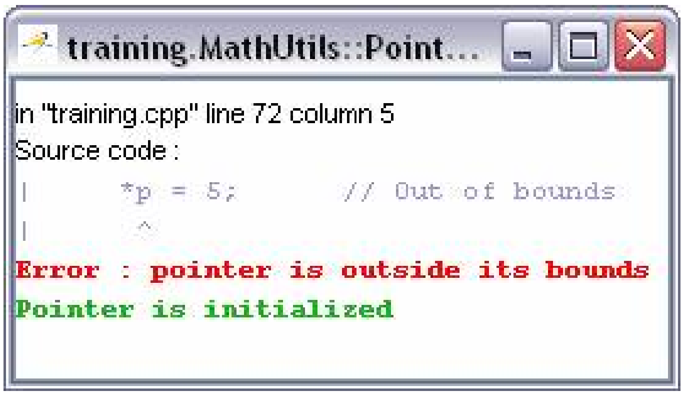
```
Error : pointer is outside its bounds
Pointer is initialized
```

training.cpp

```
61  void MathUtils::Pointer_Arithmetic ()
62  {
63    Utils  u;
64
65    int tab[100];
66    int i, *p = tab;
67
68    for(i = 0; i < 100; i++, p++)
69      *p = 0;
70
71    if(u.random_int() == 0)
72      *p = 5;       // Out of bounds
73
74    i = u.random_int();
75    if (u.random_int())  *(p-i) = 10;
76
77    if (0<i && i<=100)
78      {
79        p = p - i;
80        *p = 5;      // Safe pointer access
81      }
82  }
```

Variables View
Call Tree View
New_Project
__polyspace_main.cpp
training.cpp
MathUtils::Close_To_Zero()
MathUtils::Non_Infinite_Loop()
MathUtils::Pointer_Arithmetic()

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


training.MathUtils::Point...

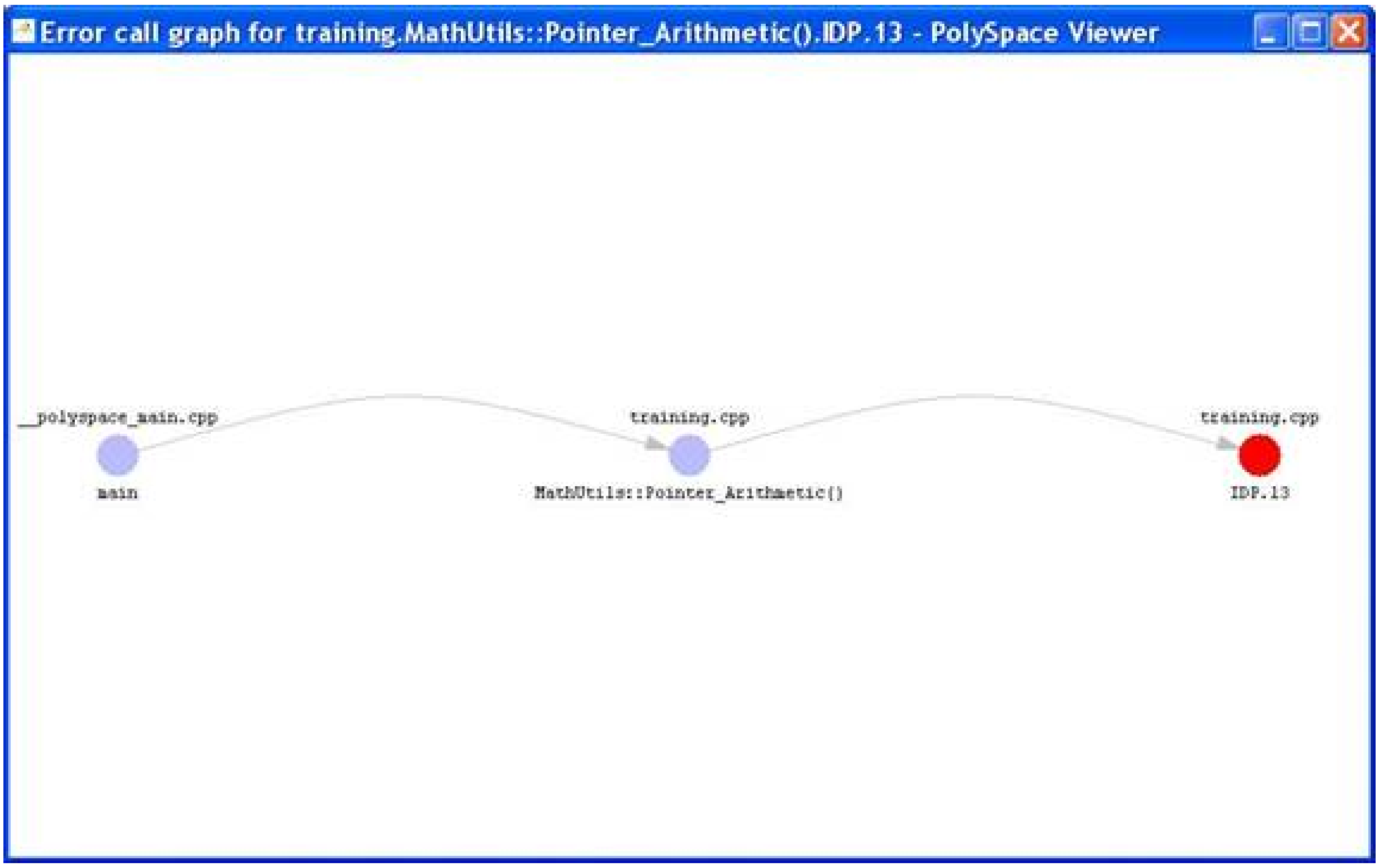
in "training.cpp" line 72 column 5
Source code :

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

Error : pointer is outside its bounds
Pointer is initialized

Pointer p is de-referenced outside of its bounds. Indeed, at the 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".

? 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 display the corresponding run-time error access graph:


Error call graph for training.MathUtils::Pointer_Arithmetic().IDP.13 - PolySpace Viewer
__polyspace_main.cpp
main
training.cpp
MathUtils::Pointer_Arithmetic()
training.cpp
IDP.13

2.4.3.2. Colours in the Source code view

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

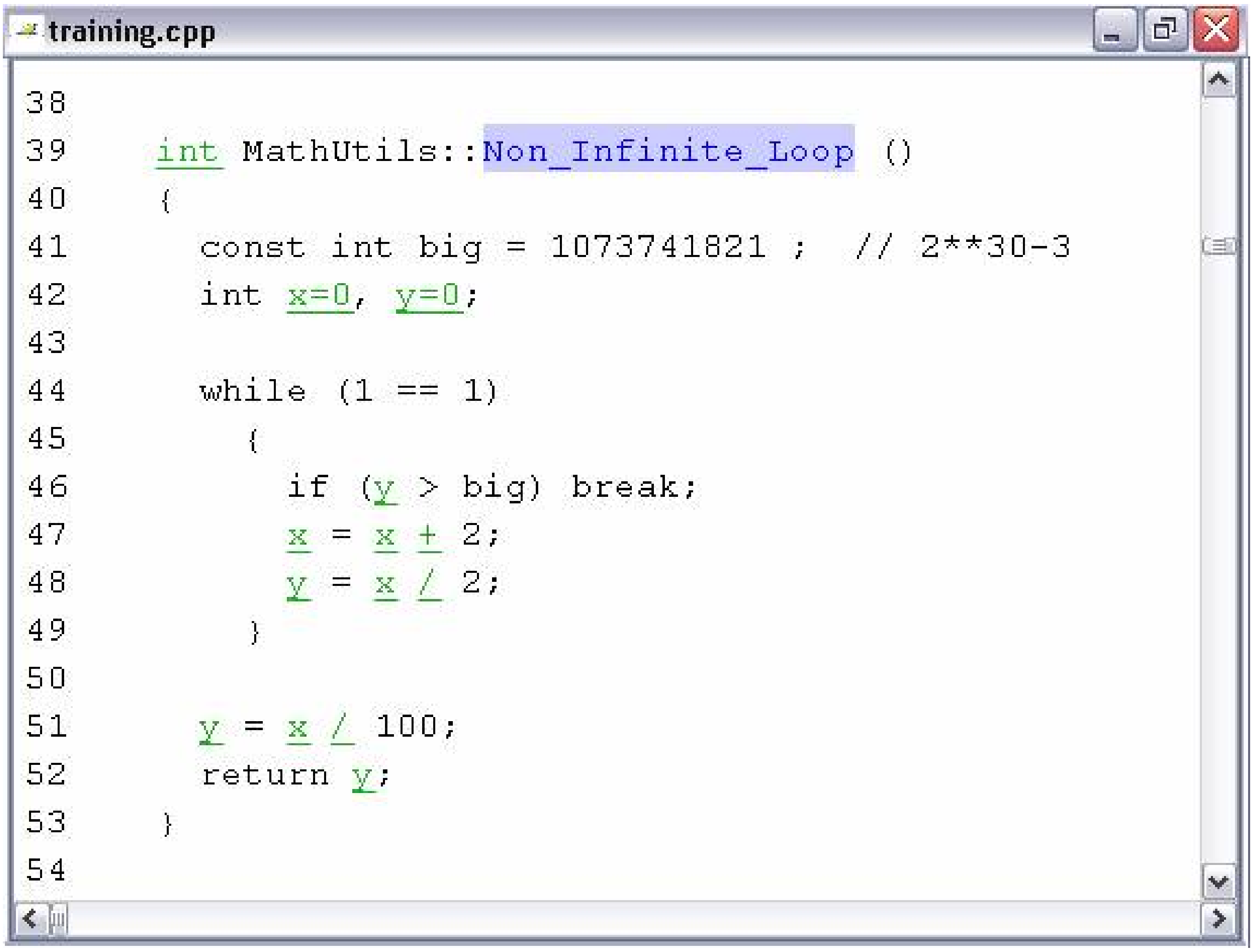
- **Red**: A link to the error message associated to the error which occurs at every execution.
- **Orange**: A link to an unproven message – an error may occur sometimes.
- **Grey**: A link to a check shown as unreachable code. The error message is in grey.
- **Green**: A link to a VOA (Value on Assignment) or an error condition that will never occur in the list of verifications made by PolySpace.
- 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`”.
- Underligned blue: A link to a global variable in the “Global variable View”.

2.4.3.3. 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 overflows, even if the `exit` condition of loop deals with y that is smaller than x. PolySpace confirms that the function always terminates.



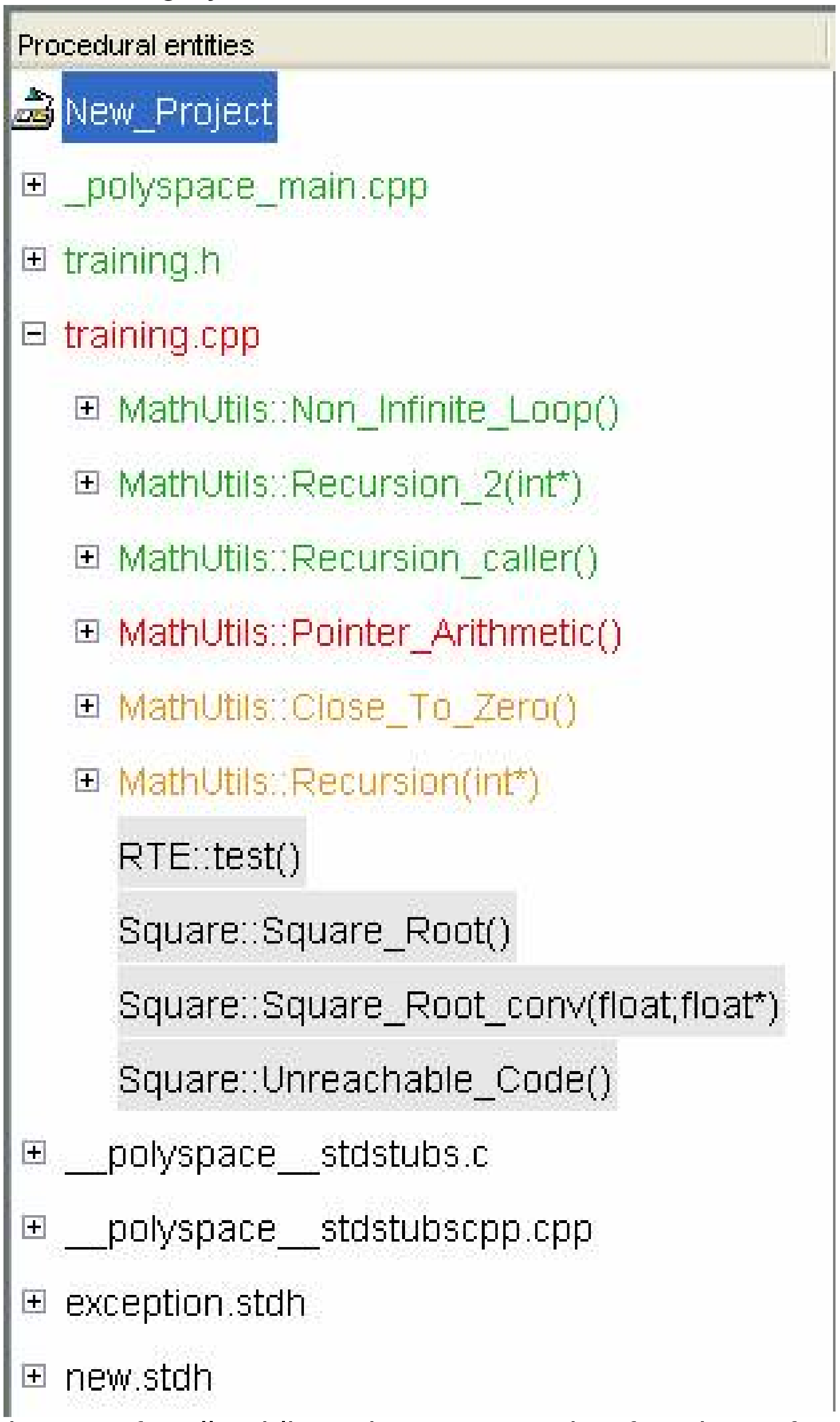
training.cpp

```
int MathUtils::Non_Infinite_Loop ()
{
  const int big = 1073741821 ;  // 2**30-3
  int x=0, y=0;

  while (1 == 1)
    {
      if (y > big) break;
      x = x + 2;
      y = x / 2;
    }

  y = x / 100;
  return y;
}
```

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

Other unreachable code

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



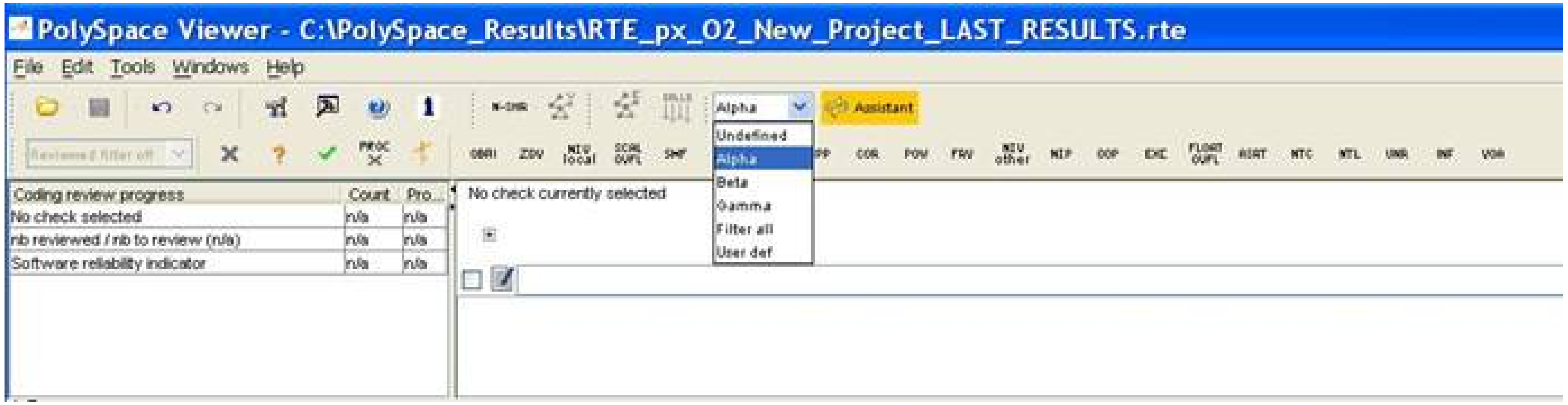
```
Procedural entities
New_Project
⊞ _polyspace_main.cpp
⊞ training.h
⊟ training.cpp
    ⊞ MathUtils::Non_Infinite_Loop()
    ⊞ MathUtils::Recursion_2(int*)
    ⊞ MathUtils::Recursion_caller()
    ⊞ MathUtils::Pointer_Arithmetic()
    ⊞ MathUtils::Close_To_Zero()
    ⊞ MathUtils::Recursion(int*)
      RTE::test()
      Square::Square_Root()
      Square::Square_Root_conv(float,float*)
      Square::Unreachable_Code()
⊞ __polyspace__stdstubs.c
⊞ __polyspace__stdstubscpp.cpp
⊞ exception.stdh
⊞ new.stdh
```

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

2.4.3.4. Advanced results exploration

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

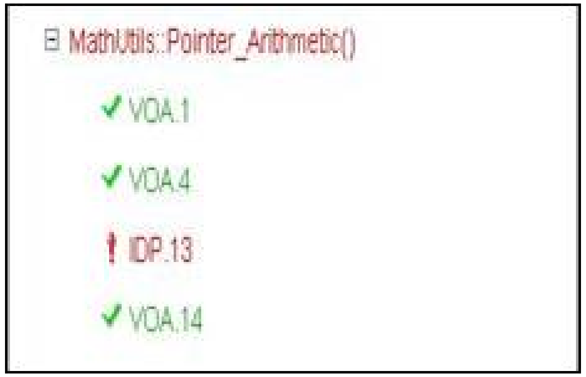
There are pre-defined composite filters Alpha, Beta and Gamma that you can choose depending on your development process. Click on the Gamma button to get all the "red" and "grey" code sections. It is mainly used during the earliest development stages to focus quickly on critical bugs. Theses filters are accessible through a combo list:


PolySpace Viewer - C:\PolySpace_Results\RTE_px_O2_New_Project_LAST_RESULTS.rte

Coding review progress	Count	Pro...
No check selected	n/a	n/a
nb reviewed / nb to review (n/a)	n/a	n/a
Software reliability indicator	n/a	n/a

No check currently selected

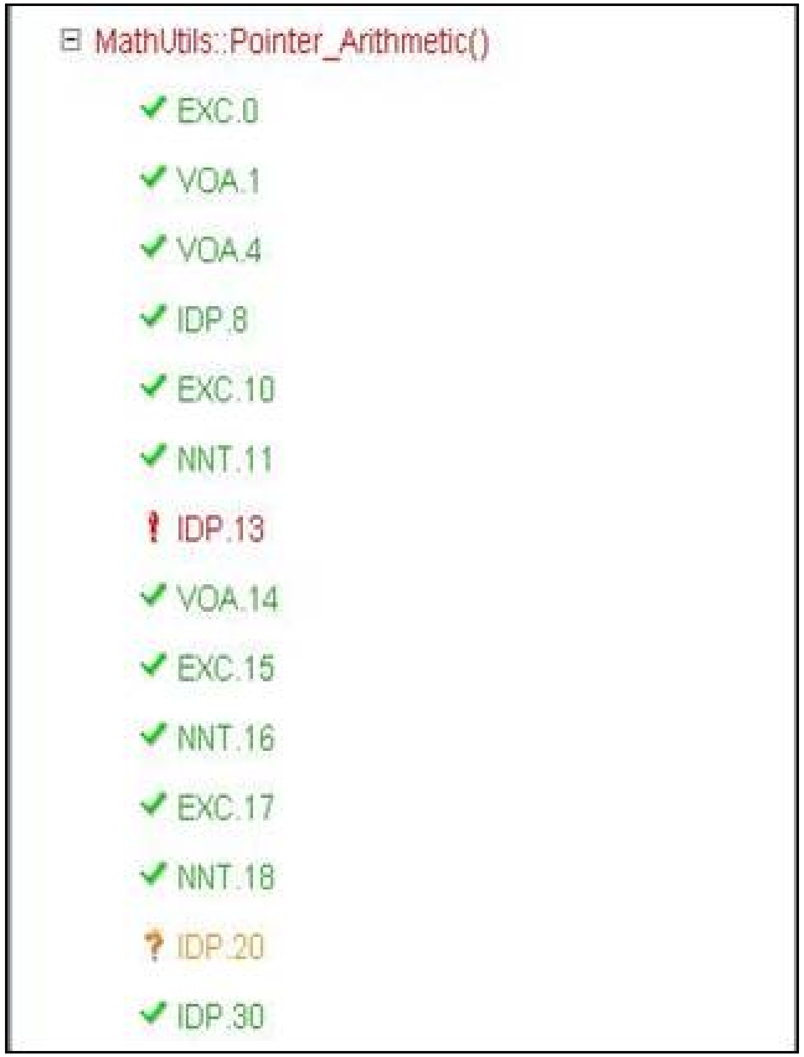
? To illustrate the use of these filters, we will focus on the Pointer arithmetic member function that we have examined in a previous section. Click on Gamma to reduce the information checks related to "`MathUtils::Pointer_Arithmetic()`".


MathUtils::Pointer_Arithmetic()
- ✓ VOA.1
- ✓ VOA.4
- ! IDP.13
- ✓ VOA.14

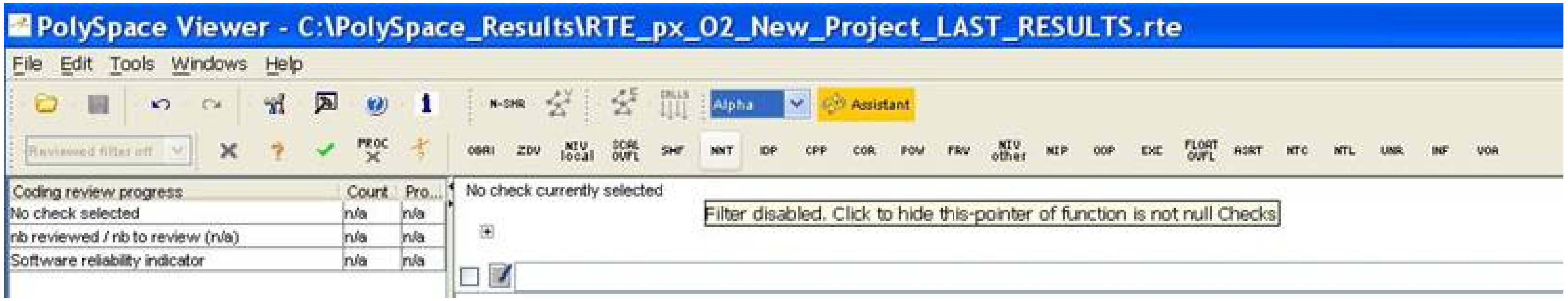
This list of acronyms - for type of operations checked - shows what PolySpace automatically analyzed for you. In the case of member function is an illegal dereference pointer error (IDP.13).
Note: VOA check (Value On Assignment) is only informative check that are never hidden.

The Beta level highlights checks that could cause a processor halt, memory corruptions or overflows.

? Click on Beta mode which is the default mode. Select again “`MathUtils::Pointer_Arithmetic()`” in the “Procedural entities” view and then, click on ⊞ to get the list of the checks.


⊟ MathUtils::Pointer_Arithmetic()
- ✓ EXC.0
- ✓ VOA.1
- ✓ VOA.4
- ✓ IDP.8
- ✓ EXC.10
- ✓ NNT.11
- ! IDP.13
- ✓ VOA.14
- ✓ EXC.15
- ✓ NNT.16
- ✓ EXC.17
- ✓ NNT.18
- ? IDP.20
- ✓ IDP.30

To get the comprehensive list of operations checked by PolySpace, you can switch to Alpha mode. 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:


PolySpace Viewer - C:\PolySpace_Results\RTE_px_02_New_Project_LAST_RESULTS.rte

File Edit Tools Windows Help

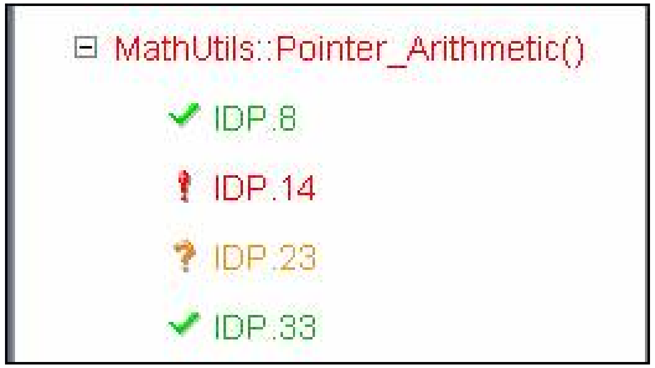
Coding review progress	Count	Pro...
No check selected	n/a	n/a
nb reviewed / nb to review (n/a)	n/a	n/a
Software reliability indicator	n/a	n/a

No check currently selected

Filter disabled. Click to hide this-pointer of function is not null Checks

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

? Click on [Filter all] (top of the window) to suppress all checks and click on [IDP]. You will get list of checks containing only IDP (**I**llegal **D**ereference **P**ointers) reds, oranges or greens:


MathUtils::Pointer_Arithmetic()

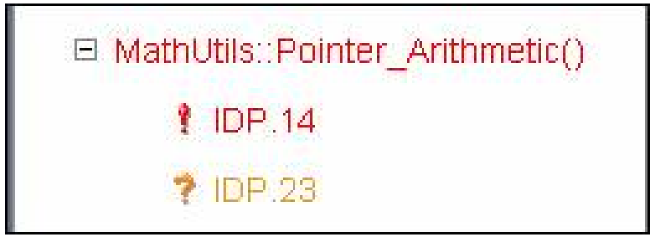
✓ IDP.8

! IDP.14

? IDP.23

✓ IDP.33

Note that clicking on ✓ (top of the window) and on the filter "VOA" to suppress green code sections, you will get a reduced list of checks reds, oranges and grays:


MathUtils::Pointer_Arithmetic()

! IDP.14

? IDP.23

2.4.3.5. C++ specific checks

Specific C++ checks are dispatched in five categories:

1. NNT category or Non Null This pointer (NNT). It checks `this` pointer validity.
2. CPP category. It concerns C++ related constructions (CPP), like positive array size verification, *`dynamic_cast`*, and `typeid`.
3. OOP category. It concerns all C++ object oriented verification (OOP): inheritance and virtual calls.
4. EXC category. It concerns all C++ constructions dealing with exceptions (EXC).
5. INF category. It concerns information about C++ implicit and called functions when dealing with virtual functions (INF).

When reviewing 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. Checks are classified from the left to the right. It is important to begin a review following this order. It is also important to begin by C "like" checks before C++ like "checks".

This methodology permits to focus first by the categories which are most susceptible to hide run time errors. This methodology has been automatically applied in the "Methodological assistant".

2.4.3.6. Miscellaneous

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

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

2.4.4. Methodological asssitant

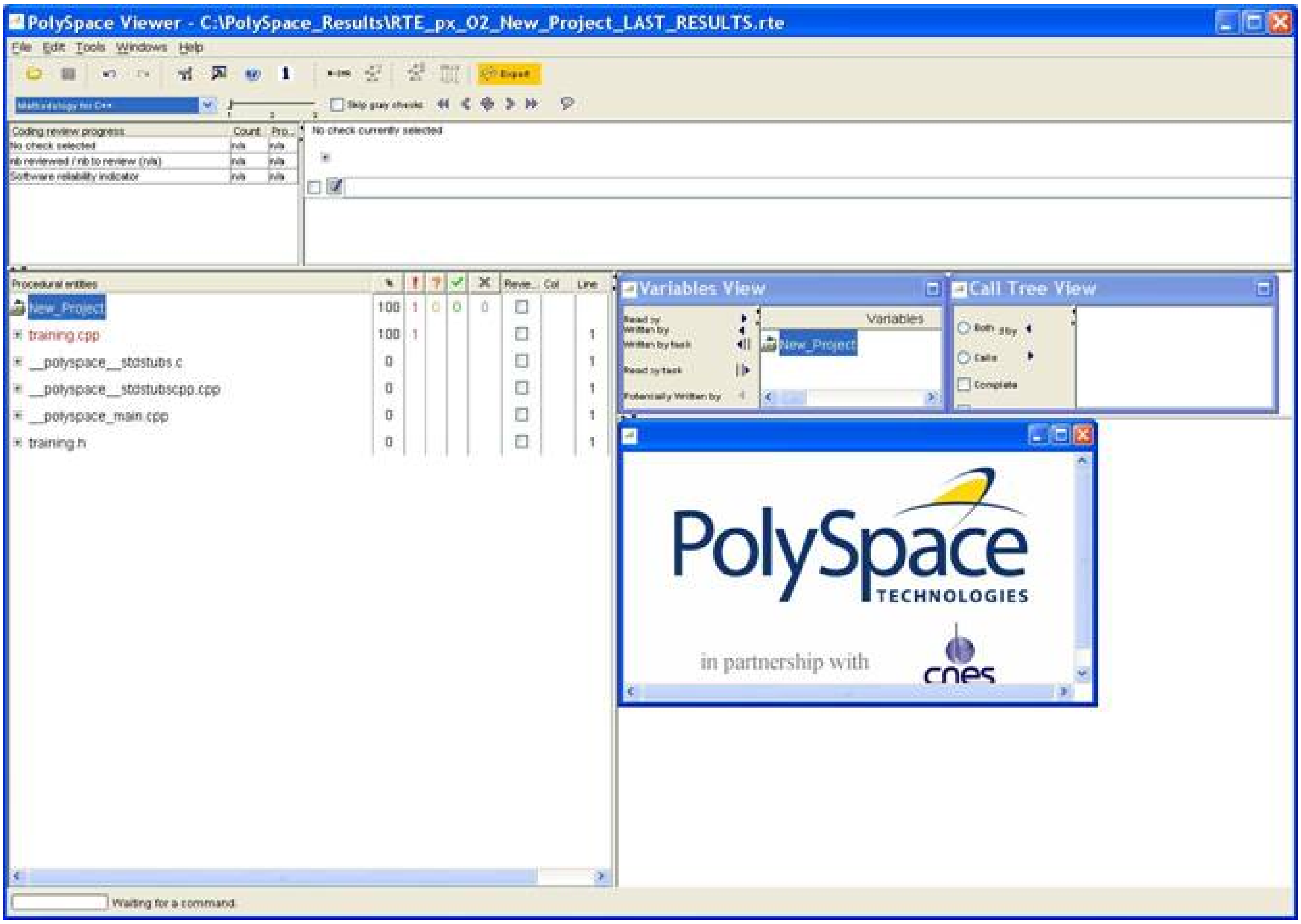
After a first navigation into the PolySpace Viewer, some simple questions remain:

- Do all checks need 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.

? If the PolySpace Viewer is still open, close it and open it again, load same results and chose “`Assistant`” mode.

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



Coding review progress	Count	Pro...
No check selected	n/a	n/a
nb reviewed / nb to review (n/a)	n/a	n/a
Software reliability indicator	n/a	n/a

No check currently selected

Procedural entities	%	!	?	✓	✗	Revie...	Col	Line
New_Project	100	1	0	0	0	☐		
training.cpp	100	1				☐		1
__polyspace__stdstubs.c	0					☐		1
__polyspace__stdstubscpp.cpp	0					☐		1
__polyspace_main.cpp	0					☐		1
training.h	0					☐		1

Waiting for a command.

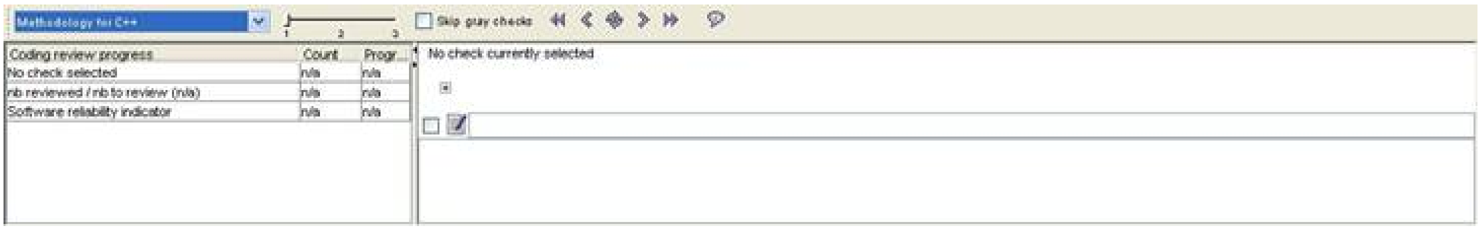
Related subjects :

[2.4.4.1. Assistant dashboard](#)

[2.4.4.2. Choose a methodological assistant](#)

2.4.4.1. 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:



Coding review progress	Count	Progr...
No check selected	n/a	n/a
nb reviewed / nb to review (n/a)	n/a	n/a
Software reliability indicator	n/a	n/a

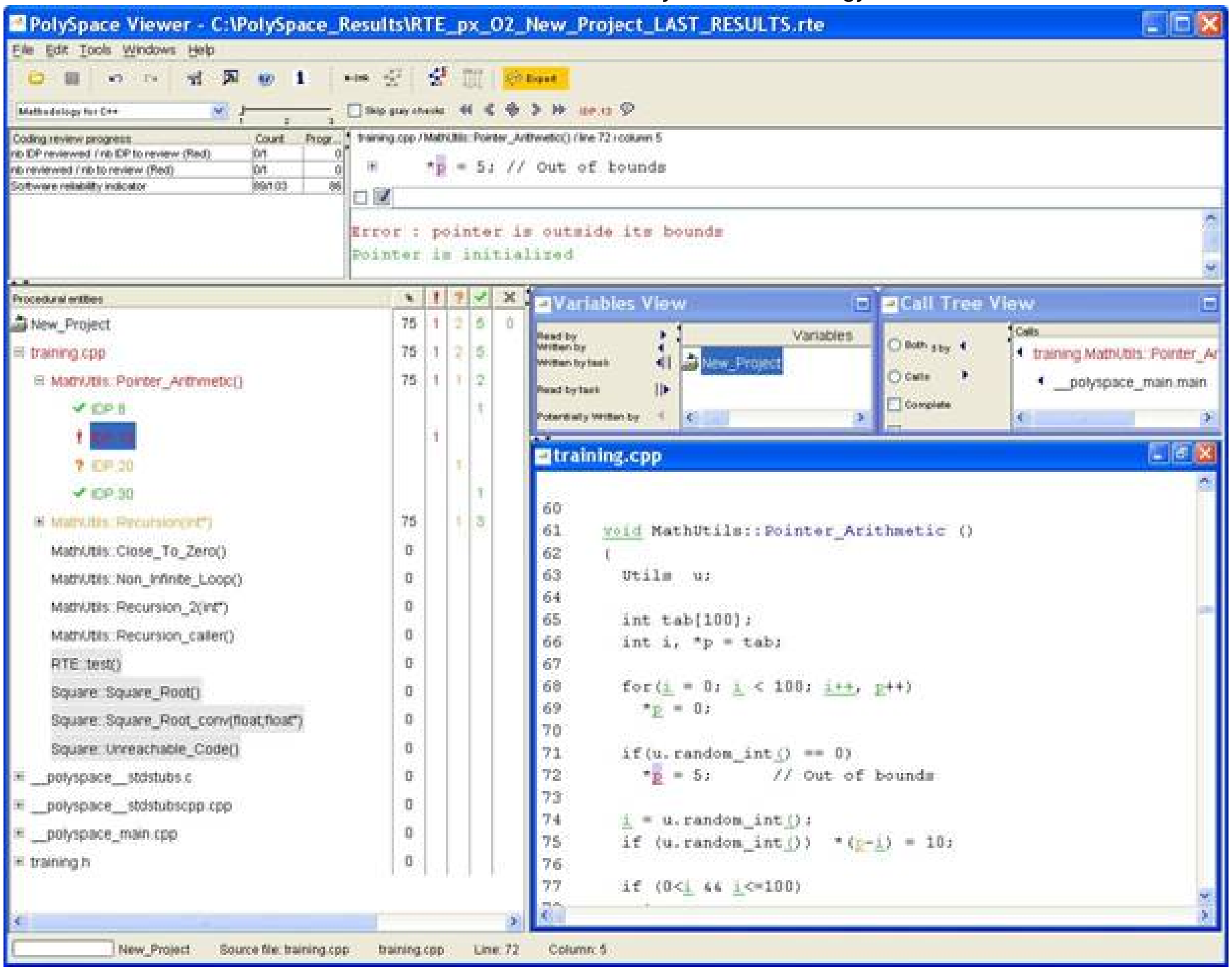
No check currently selected

Some other changes can be seen in the viewer:

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

? Click on .to navigate to next check.

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



PolySpace Viewer - C:\PolySpace_Results\RTE_px_02_New_Project_LAST_RESULTS.rte

Coding review progress	Count	Progr...
nb IDP reviewed / nb IDP to review (Red)	0/1	0
nb reviewed / nb to review (Red)	0/1	0
Software reliability indicator	89/103	86

training.cpp / MathUtils::Pointer_Arithmetic() / line 72 / column 5

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

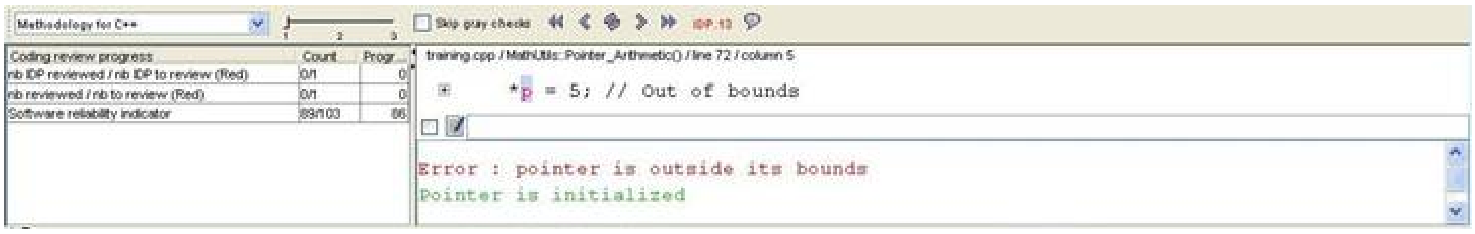
```
Error : pointer is outside its bounds
Pointer is initialized
```

```
void MathUtils::Pointer_Arithmetic ()
{
  Utils  u;

  int tab[100];
  int i, *p = tab;

  for(i = 0; i < 100; i++, p++)
    *p = 0;

  if(u.random_int() == 0)
    *p = 5;       // Out of bounds

  i = u.random_int();
  if (u.random_int())  *(p-i) = 10;

  if (0<i && i<=100)
```

Source file: training.cpp   Line: 72   Column: 5

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.

? Tick the radio button box and type an associated comment in the associated edit box on the right.
After, it looks like:



Coding review progress	Count	Progr...
nb IDP reviewed / nb IDP to review (Red)	0/1	0
nb reviewed / nb to review (Red)	0/1	0
Software reliability indicator	89/103	86

training.cpp / MathUtils::Pointer_Arithmetic() / line 72 / column 5

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

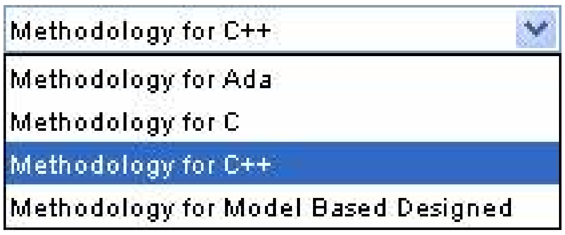
```
Error : pointer is outside its bounds
Pointer is initialized
```

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 colour category (red, grey and orange).
- Last line gives in permanence the `Software reliability indicator`.

Other buttons in the Methodological dash board allow navigating to previous check, coming back to current one and going to next / previous category selected by the Methodology.

2.4.4.2. Choose a methodological assistant

Some methodologies 

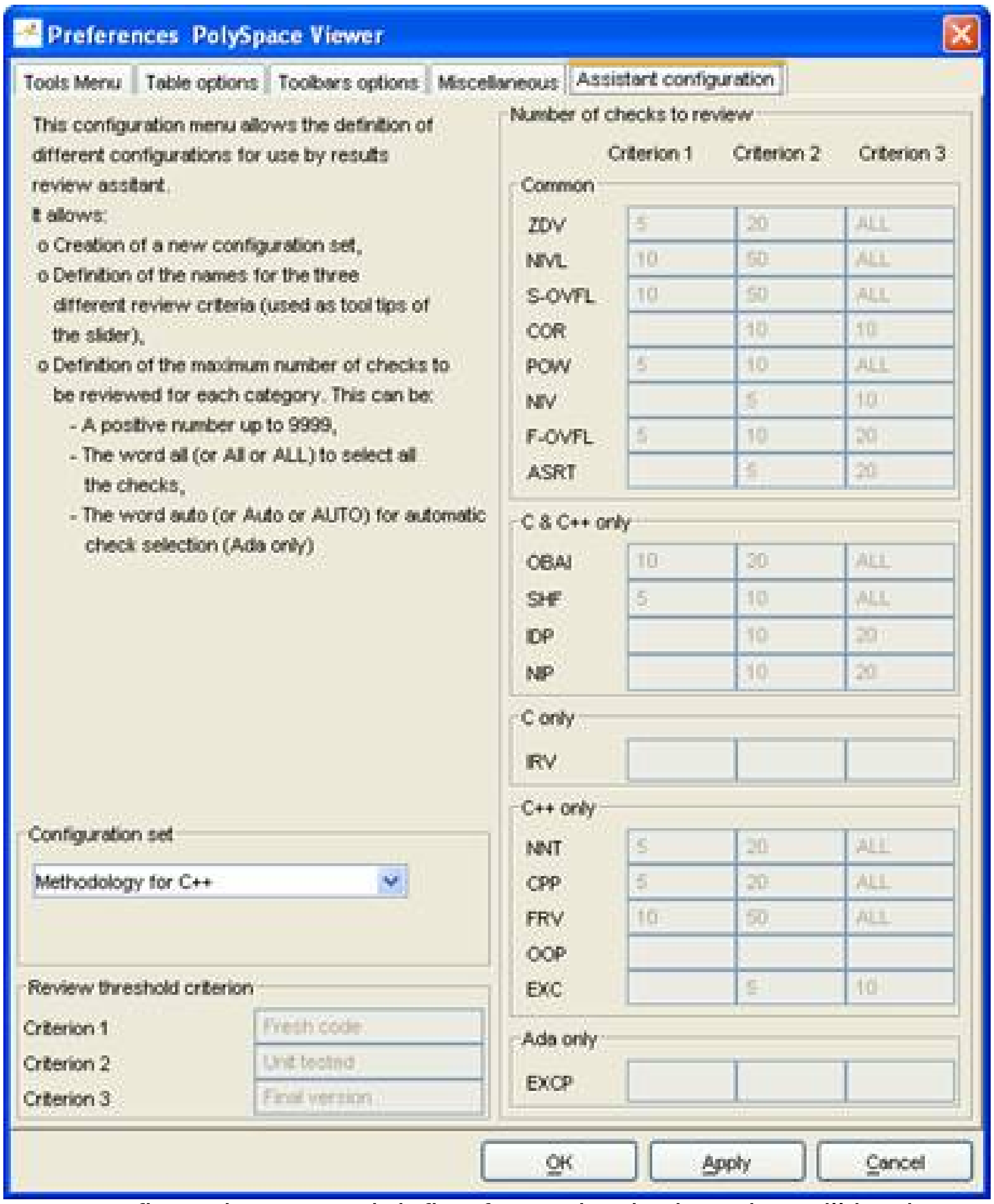
and associated levels

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. The “`Preferences PolySpace Viewer>Assistant methodology`” Tab is accessible from the “`Edit`” menu.



This configuration menu allows the definition of different configurations for use by results review assitant.
It allows:
o Creation of a new configuration set,
o Definition of the names for the three different review criteria (used as tool tips of the slider),
o Definition of the maximum number of checks to be reviewed for each category. This can be:
- A positive number up to 9999,
- The word all (or All or ALL) to select all the checks,
- The word auto (or Auto or AUTO) for automatic check selection (Ada only)

Configuration set: Methodology for C++

Review threshold criterion

Criterion 1	Fresh code
Criterion 2	Unit tested
Criterion 3	Final version

Number of checks to review

	Criterion 1	Criterion 2	Criterion 3
Common			
ZDV	5	20	ALL
NIVL	10	50	ALL
S-OVFL	10	50	ALL
COR		10	10
POW	5	10	ALL
NIV		5	10
F-OVFL	5	10	20
ASRT		5	20
C & C++ only			
OBAI	10	20	ALL
SHF	5	10	ALL
IDP		10	20
NIP		10	20
C only			
IRV			
C++ only			
NNT	5	20	ALL
CPP	5	20	ALL
FRV	10	50	ALL
OOP			
EXC		5	10
Ada only			
EXCP			

You can create a new configuration set and define for each criterion what will be the categories of check to review and how many in each one.

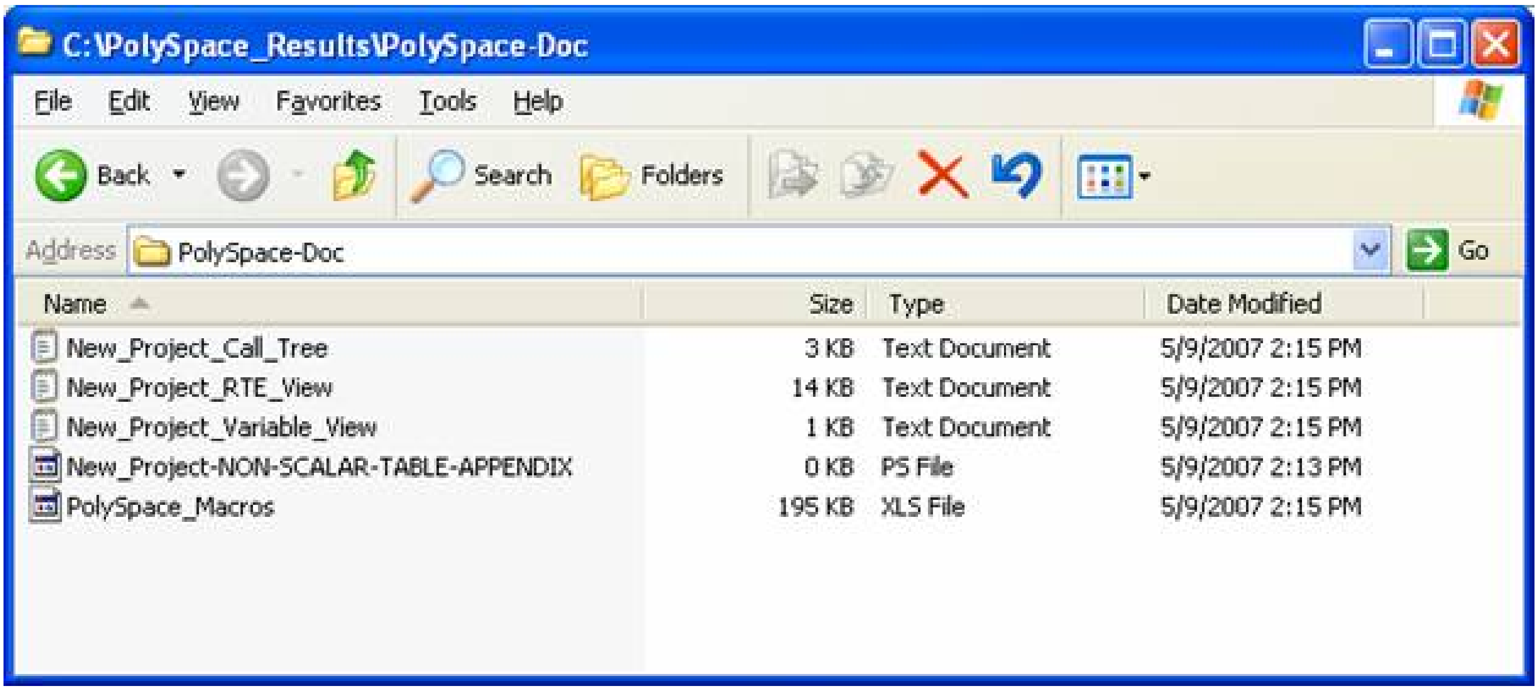
Note: This is not possible to refine an existing configuration except by duplication and refinement.

2.4.5. Report Generation

When PolySpace performs an analysis, it generates textual files that can be used to generate 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 your PolySpace representative (or http://www.polyspace.com/contact.htm).

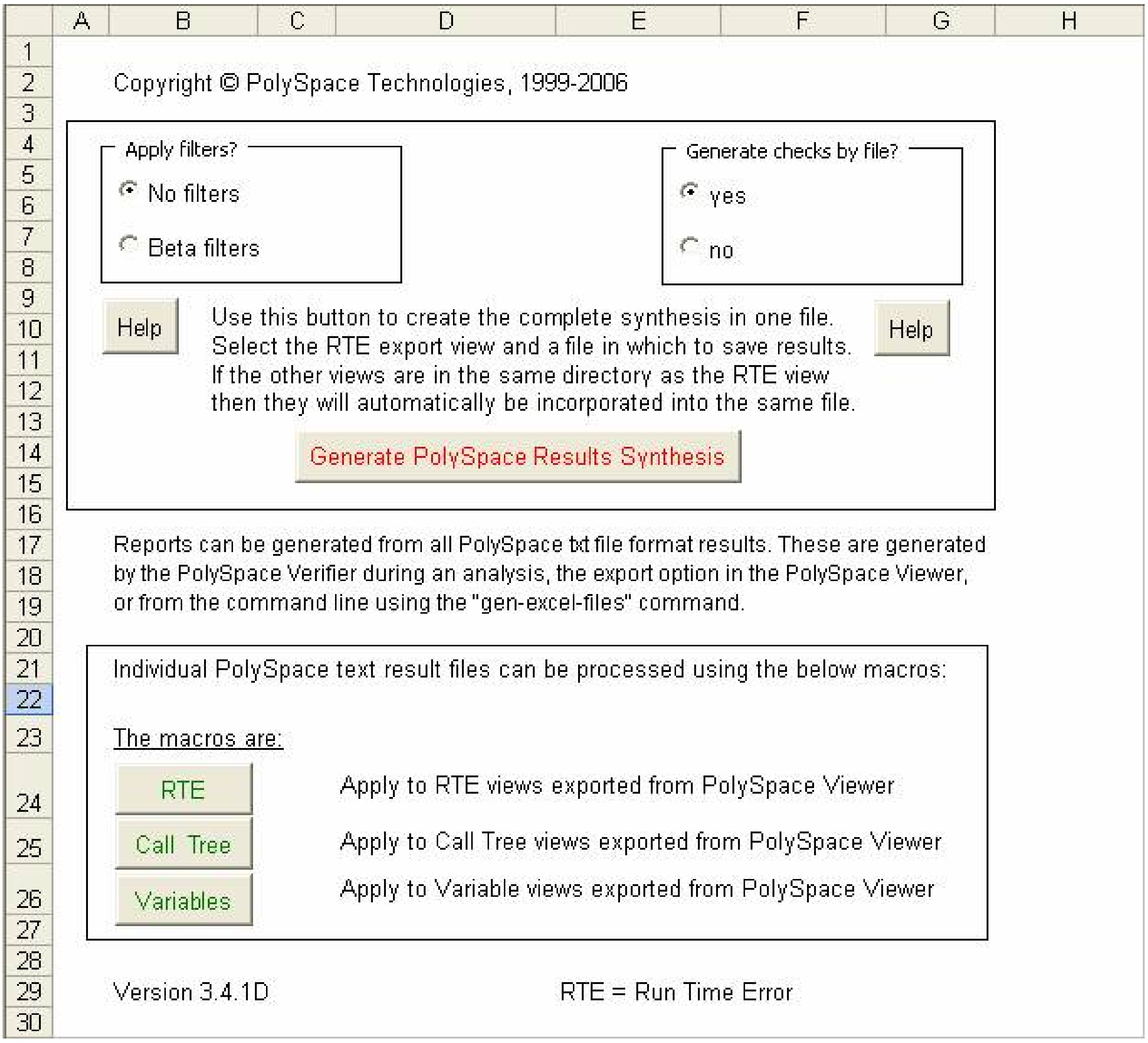
These files are located in the results directory (See ”`C:\PolySpace_Results\PolySpace-Doc`“ or “`<PolySpaceInstallDir>\Examples\Demo_CPP\PolySpace-Doc`”). All views (except source code) are printable and can be exported to textual or Excel® format.

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


C:\PolySpace_Results\PolySpace-Doc
File Edit View Favorites Tools Help
Back Search Folders
Address PolySpace-Doc Go

Name	Size	Type	Date Modified
New_Project_Call_Tree	3 KB	Text Document	5/9/2007 2:15 PM
New_Project_RTE_View	14 KB	Text Document	5/9/2007 2:15 PM
New_Project_Variable_View	1 KB	Text Document	5/9/2007 2:15 PM
New_Project-NON-SCALAR-TABLE-APPENDIX	0 KB	PS File	5/9/2007 2:13 PM
PolySpace_Macros	195 KB	XLS File	5/9/2007 2:15 PM

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



Copyright © PolySpace Technologies, 1999-2006

Apply filters?
[✓ No filters]
Beta filters

Generate checks by file?
[✓ yes]
no

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

Reports can be generated from all PolySpace txt file format results. These are generated by the PolySpace Verifier during an analysis, the export option in the PolySpace Viewer, or from the command line using the "gen-excel-files" command.

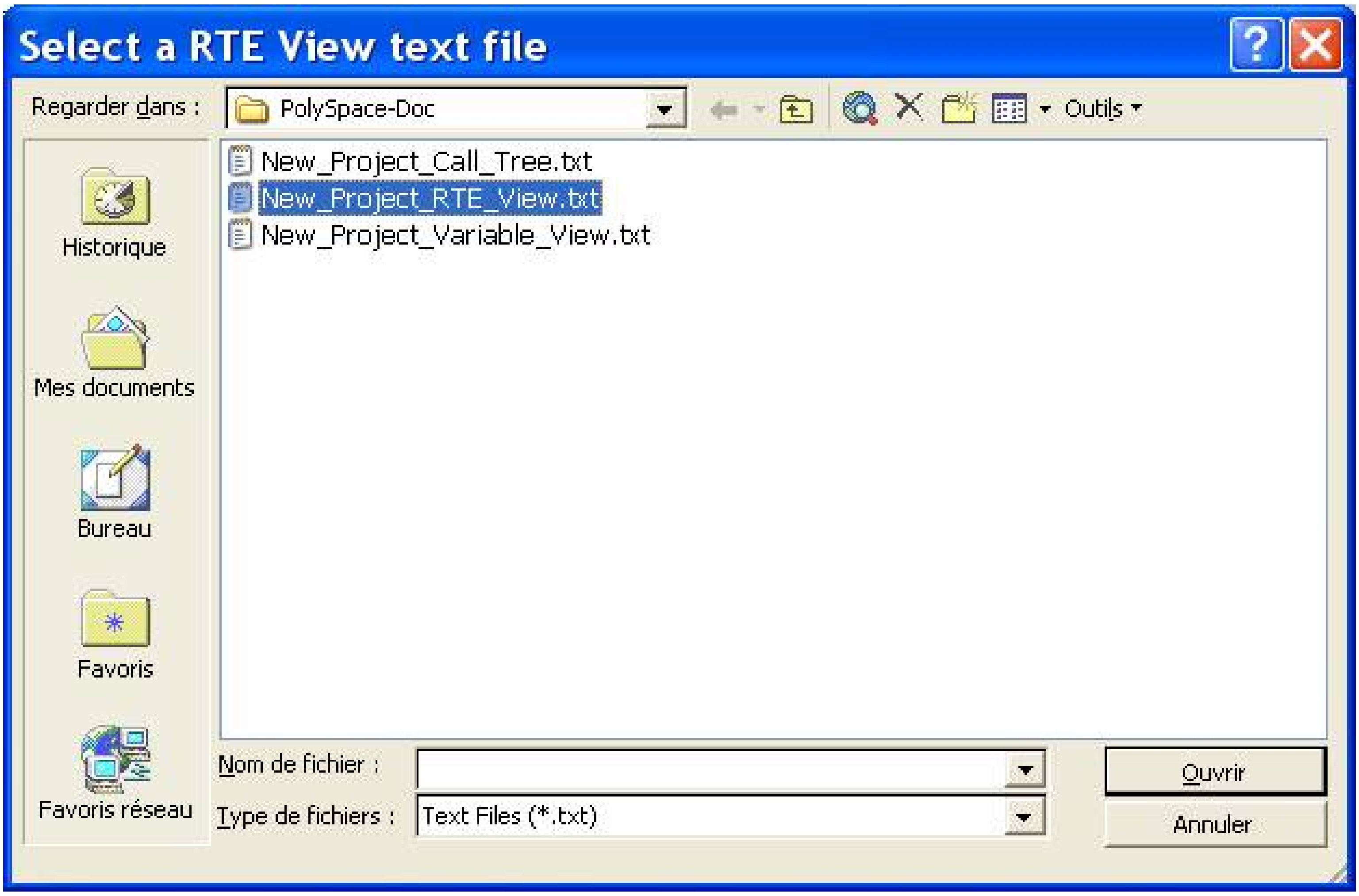
Individual PolySpace text result files can be processed using the below macros:

The macros are:

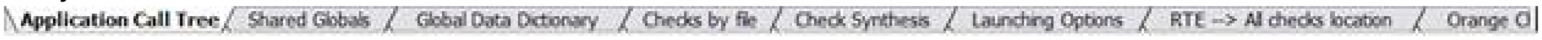
RTE	Apply to RTE views exported from PolySpace Viewer
Call Tree	Apply to Call Tree views exported from PolySpace Viewer
Variables	Apply to Variable views exported from PolySpace Viewer

Version 3.4.1D RTE = Run Time Error

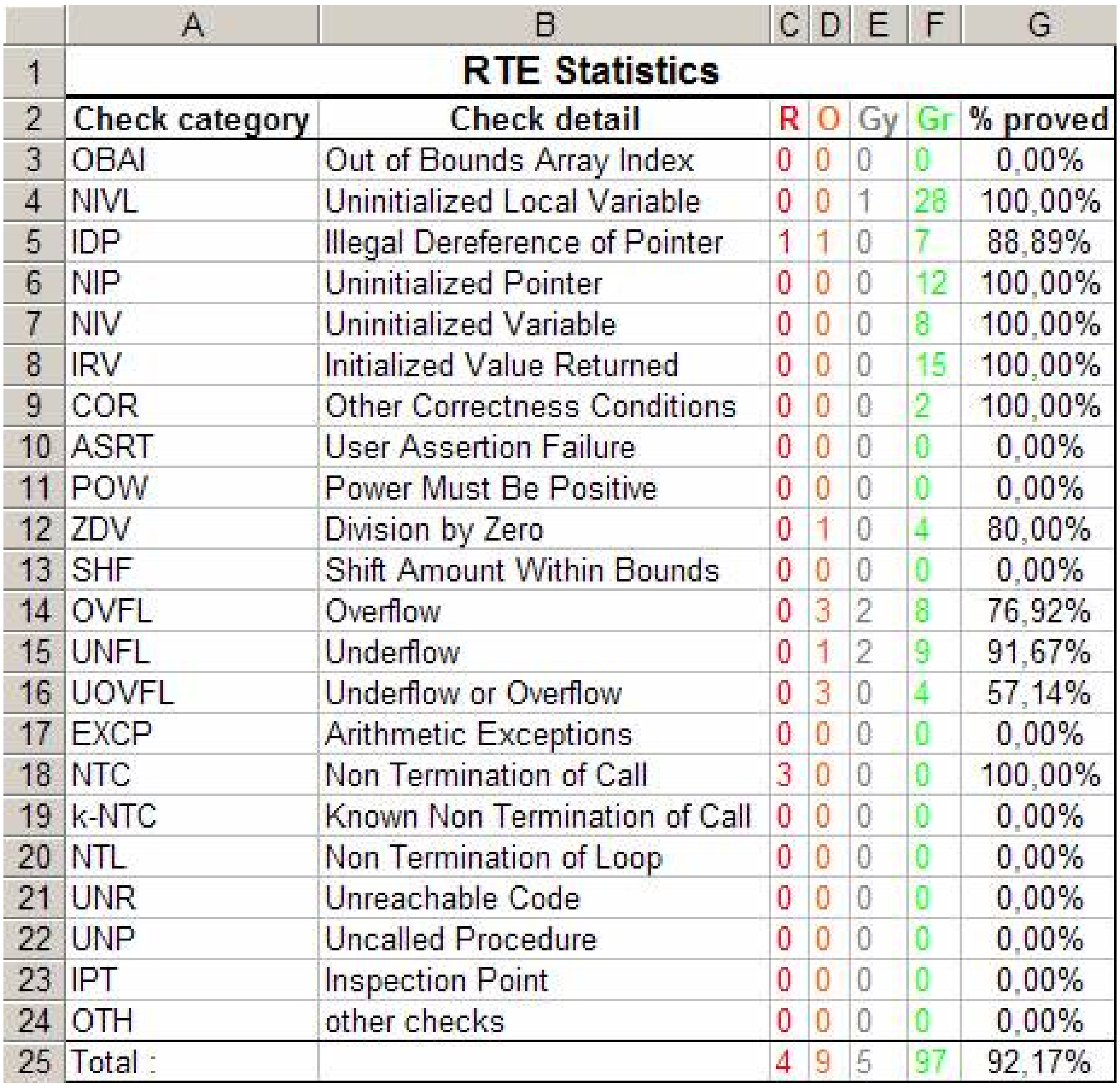
? Click on . A file browser opens. Select the file called "`New_Project_RTE_View.txt`" as shown below:



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.

2.5. Launch PolySpace Remotely

This paragraph describes the basic steps to launch an analysis in a remote way. This operation necessitates that:

1. A Queue Manager server (QM) has been previously installed.
2. Your desktop PC has been configured with a PolySpace Client.

See the PolySpace Installation guide available on the PolySpace CD-ROM in `<CD-ROM>\Docs\Install` in order to install and configure the Remote Launcher between a Client and a Server.

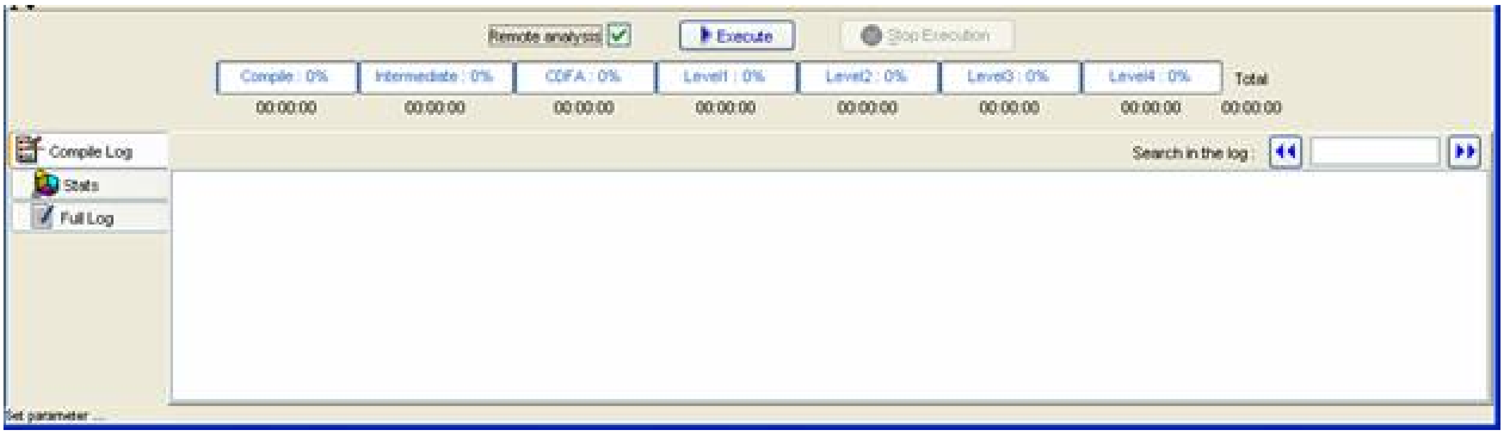
Note: Launching an analysis remotely requires a PolySpace Server product and associated license.

Related subjects :

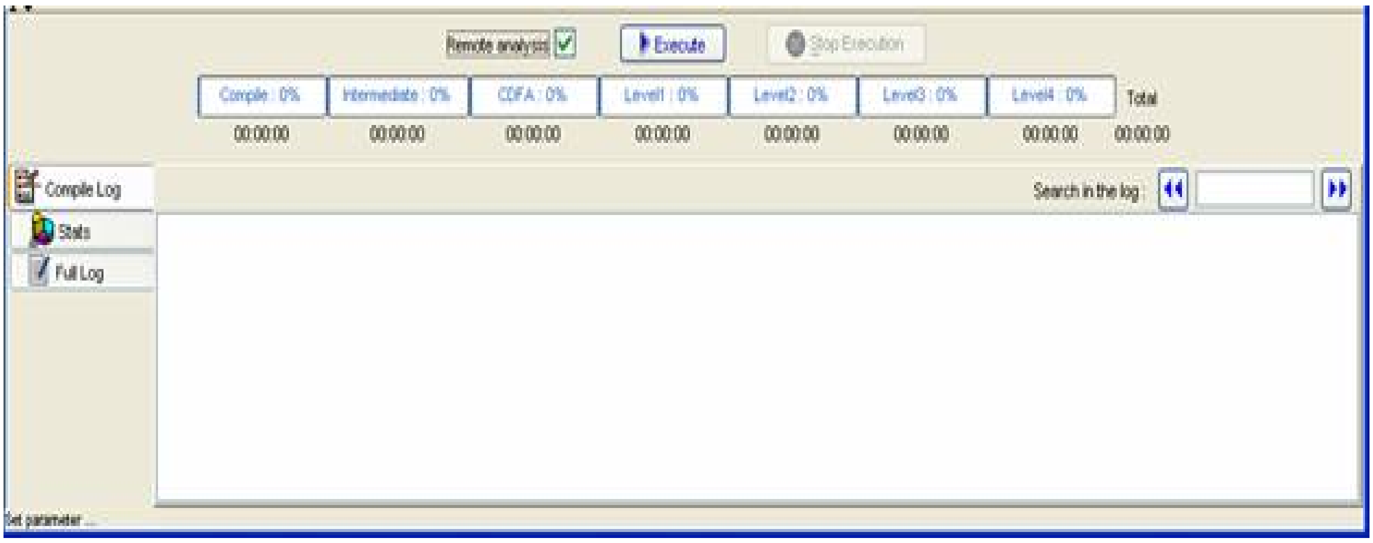
2.5.1. Steps of Launching

You need to follow theses two simple steps:

- Step 1: prepare an analysis. You need to set up an analysis like it has been described in step 1 without launching it.
- Then you just have to tick the "**Remote analysis**" radio button (see next figure) and then, click on Execute 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. By clicking on the "`Full Log`" tab, you will have a message like that:



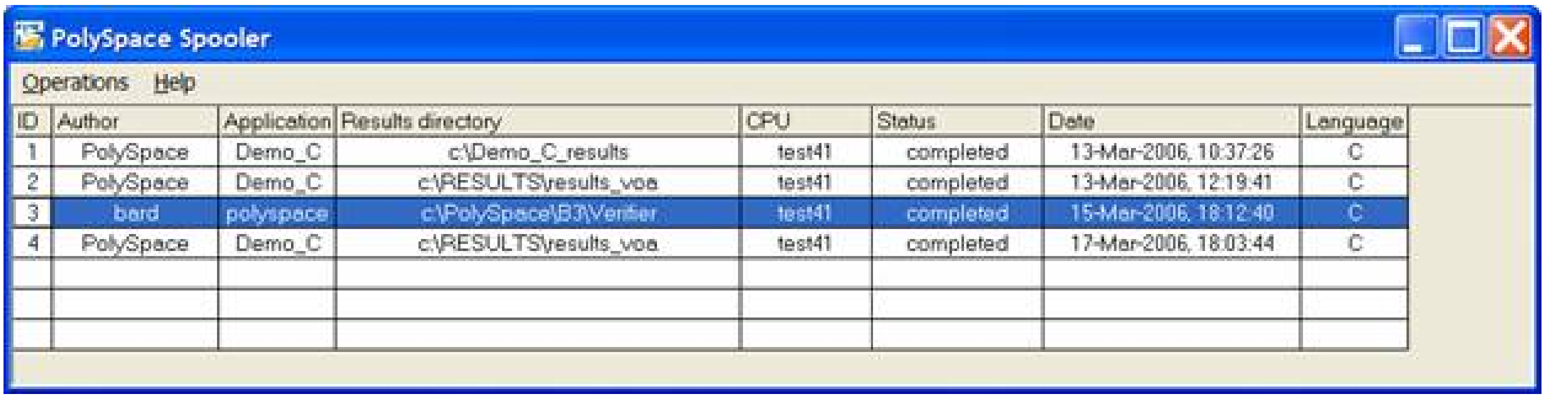
The analysis has been queued with an ID number and you can follow its progression using the PolySpace Spooler.

Without ticking the "`Remote analysis`" radio button, the analysis continues locally.

2.5.2. Management of PolySpace analysis in remote: the PolySpace Spooler

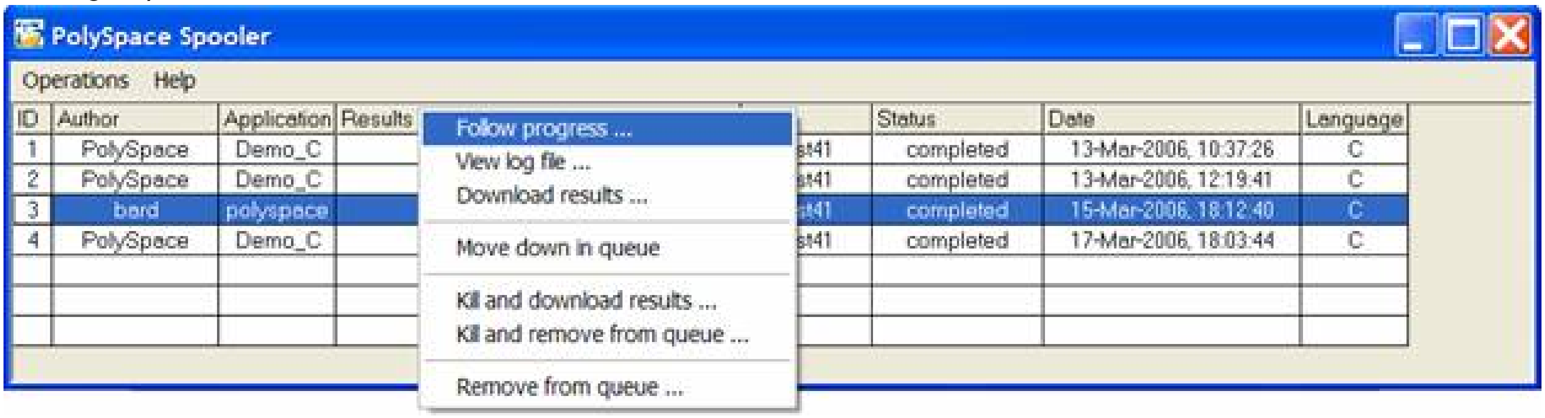
You can see the PolySpace jobs on the remote queue by clicking on the short cut on your desktop PC

or on the icon in the menu bar of the launcher.



ID	Author	Application	Results directory	CPU	Status	Date	Language
1	PolySpace	Demo_C	c:\Demo_C_results	test41	completed	13-Mar-2006, 10:37:26	C
2	PolySpace	Demo_C	c:\RESULTS\results_voa	test41	completed	13-Mar-2006, 12:19:41	C
3	bard	polyspace	c:\PolySpace\B3\Verifier	test41	completed	15-Mar-2006, 18:12:40	C
4	PolySpace	Demo_C	c:\RESULTS\results_voa	test41	completed	17-Mar-2006, 18:03:44	C

When you select an analysis (by clicking on the left button of the mouse), you can manage it in the queue (see next figure):



- *`Follow progress`*. This action lists the associated log file in a Launcher window. If the analysis is running, you can follow on the Launcher window the update of the log file and associated progress bar in real time.

- `View log file`. This action lists the associated log file in a “`Command prompt`” window. If the analysis is running you can follow on a “`Command prompt`” window the 100 last lines update of the log file in real time.

- *`Download results`*. This action downloads results of an analysis on the client. The download is not possible for a queued analysis. If the analysis is still running, available results are downloaded on the client, without disturbing the analysis.

- *`Move down in queue`*. This action reduces the priority of a queued analysis.

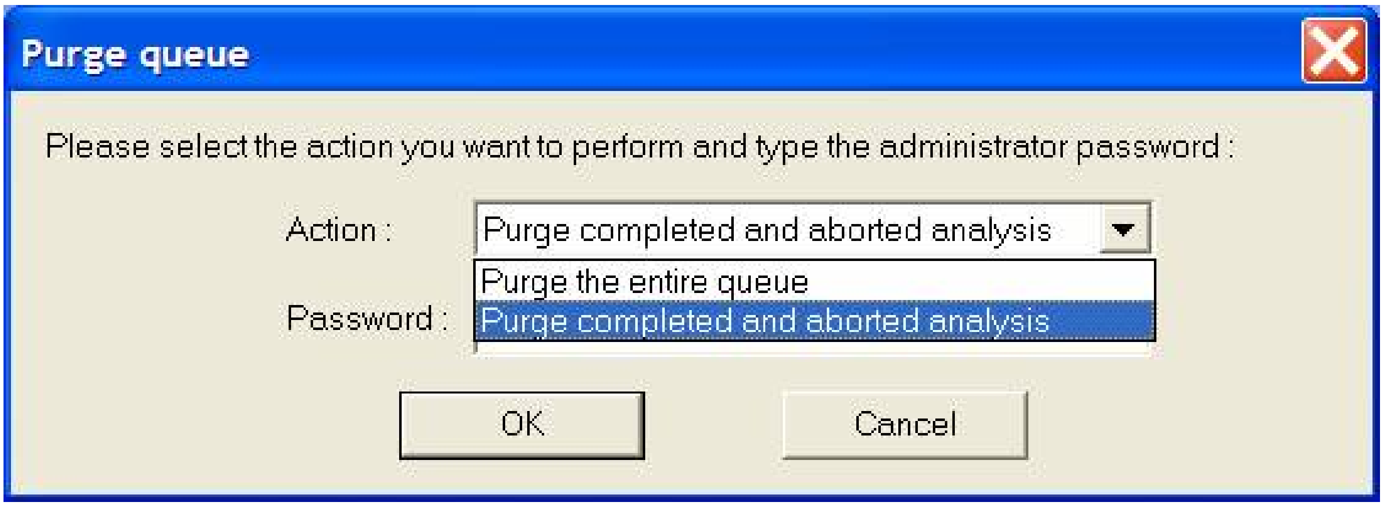
- *`Kill and download results`*. This action works if the analysis is running. The analysis will be definitively stopped and the results will be downloaded. The status of the analysis changes to aborted. The analysis remains on the queue.

- *`Kill and remove from queue`*. This action works if the analysis is running. The analysis will be definitively stopped, and the analysis will be removed from the queue. **Note: The results will be lost**.

- *Remove from queue*. This action removes a queued, aborted or a completed analysis. **Note: The results will be lost.**

The queue can be managed from an administrator point of view with the "*Operations*>" menu:

- "`Operations>Purge queue`". This action purges the entire queue or purges only completed and aborted analysis (see next figure). The password of the queue's manager is required.


Purge queue

Please select the action you want to perform and type the administrator password :

Action : Purge completed and aborted analysis

Purge the entire queue
Purge completed and aborted analysis

Password :

OK Cancel

- "`Operations>Change root password`". This action changes the password of queue manager. **Note:** by default this password does not exist.

On a UNIX platform, there is no graphical user interface but a set of commands which allow the management of analyses on the queue. All theses commands begin with the prefix `<PolySpaceCommonDir>/RemoteLauncher/psqueue-`.

2.5.3. Batch commands

- Launch analysis in batch:

A set of commands allow the launching of analysis in batch (under a cygwin shell on a Windows machine). All theses commands begin with the prefix `<PolySpaceCommonDir>/RemoteLauncher/bin/polyspace-remote-`: `polyspace-remote-cpp` (a server analysis) and `polyspace-remote-desktop-cpp` (a client analysis). By default the `<PolySpaceCommonDir>` is "`C:\PolySpace\PolySpace_Common`".

They are equivalent to respectively the commands with a prefix `<PolySpaceInstallDir>/bin/polyspace-`. For example, `polyspace-remote-desktop-cpp –server [<hostname>:[<port>] | auto]` allows the sending of a PolySpace client for C++ analysis remotely.

- Manage analysis in batch:

In batch and on a UNIX platform, a set of commands allow the management of analysis in the queue. All theses commands begin with the prefix `<PolySpaceCommonDir>/RemoteLauncher/bin/psqueue-`:

  - `psqueue-download <ID> <results dir>`: download an identified analysis into a results directory. `[-f]` force download (without interactivity) and `–admin –p <password>` allows administrator to download results. `[–server <name>[:port]]` selects a specific Queue Manager. [-v|version] gives release number.
  - `psqueue-kill <ID>`: kill an identified analysis.
  - `psqueue-purge all|ended`: remove all or finished analyses in the queue.
  - `psqueue-dump`: gives the list of all analyses in the queue associated to default Queue Manager.
  - `psqueue-move-down <ID>`: move down an identified analysis in the Queue.
  - `psqueue-remove <id>`: remove an identified analysis in the queue.
  - `psqueue-get-qm-server`: give the name of the default Queue Manager.
  - `psqueue-progress <ID>`: give progression of the currently identified and running analysis. `[-open-launcher]` display the log in the graphical user interface of launcher. `[-full]` gives full log file.
  - `psqueue-set-password <old 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 `<PolySpaceCommonDir>/Docs` directory). `[-list-versions]` gives 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>/RemoteLauncher/bin/psqueue-<command> -h` gives information about all available options for each command.

2.5.4. Share analyses between accounts

Analysis-key.txt 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 stocked in a file `analysis-keys.txt` associated to a user account. On a Unix account, this file is located in:

- `“/home/<username>/.PolySpace”`

On a Windows account, it is located in: `“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>
...
```

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 share analysis between projects
A magic key allows to share analyses without taking into account the <id>. It allows same key for all analysis launched by a user account. The format is the following:

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

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

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

2.6. Summary

After having followed each steps of this tutorial, you are now able to launch a class analysis using PolySpace Client, 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 advanced options available with our tools in “`PolySpace C++ documentation.pdf`” available on the CD-ROM or in `<PolySpaceCommonDir>\Docs\Manuals`.

3. Analysis setup

Related subjects :

3.1. Common Compile errors

Related subjects :

3.1.1. Includes

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

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

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

Use the `–I` option to include the correct header files, including the header files of the compiler.

3.1.2. Specific keyword or extended keyword

- **Specific keyword**

Compilers of specific application are defined theirs owned keyword. A classic example is the compiler for micro controller 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 `-D` to 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: that 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`.

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

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

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

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

- **Complex post preprocessing command**

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

```
#!/usr/bin/perl

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

$version = 0.1;

$INFILE  = STDIN;
$OUTFILE = STDOUT;

while (<$INFILE>)
{

  # Remove far keyword
  s/far//;

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

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

  # Remove "@ ((unsigned)&LATD*8)+2" type constructs
  s/\@\s\(\(unsigned\)\&[A-Z0-9]+\*8\)\+\d//g;
  # Convert current line to lower case
#  $_ =~ tr/A-Z/a-z/;

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

- **Perl regular 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/;
#
```

```
################################################################################
```

3.1.3. 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.

Original code:

```
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 offered by PolySpace C++, accept the default initialization of static data member during the declaration.

3.2. Dialect issues

Related subjects :

3.2.1. iso versus default dialects

The 5 common permissiveness options used by PolySpace 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 ;
}
```

1.

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

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

2.

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

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

3.

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

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

4.

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

Using dialect `iso`, line 14 needs to be removed

5.

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

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

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

3.2.2. CFront2 and CFront3 dialects

As mentioned at the beginning of this section, `cfront2` and `cfront3` dialects were already being used before the publication of the ANSI C++ standard in 1998. Nowadays, these two dialects are used to compile legacy C++ code.
If the cfront2 or cfront3 options are not selected, you may get the common error messages below.

- **Variable scope issues**

The ANSI C++ standard specifies that the scope of the declarations occuring inside loop definition is local to the loop. However some compilers may assume that the scope is local to the bloc ({ }) which contains the loop.

Original code:

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

Error message:

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

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

- **'bool' issues**

Standard type may need to be turned into 'boolean' type

Original code:

```
enum bool
  {
    FALSE=0,
    TRUE
  };

class CBool
{
public:
   CBool ();
   CBool (bool val);
   bool m_val;
};
```

Error message:

```
Verifying  C++ sources ...

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

3.2.3. Visual dialects

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

- **Import directory**

When a Visual application uses `#import` directives, the Visual C++ compiler generates a header file which contains some definitions. These header files have a *.tlh* extension and 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 `Debug` or `Release`). Therefore, in order to provide those files, the application needs to be built first. Then, the option `–import-dir=<build directory>` must be set with a correct path folder.

- **pragma pack**

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

Original code:

test1.cpp	type.h	test2.cpp

`#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.

3.3. Link messages

Related subjects :

3.3.1. STL library C++ Stubbing errors

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

Original code:

```
#include <map>

struct A
{
  int m_val;
};

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

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


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

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

Error message:

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

  detected during instantiation of
"pair<__pst__generic_iterator<bidirectional_iterator_tag, pair<const Key,
T>>, bool> map<Key, T, Compare>::insert(const map<Key, T, Compare>::
```

```
value_type &) [with Key=A, T=B, Compare=less<A>]" at line 23 of "/cygdrive/
c/_BDS/Test-Polyspace/sources/template.cpp"
```

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

The preceding message can also appear with the directory names:

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

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

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

3.3.2. Lib C stubbing errors

- **Extern C functions**

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

Original code:

```
extern "C" {
  void* memcpy(void*, void*, int);
}

class Copy
{
public:
  Copy() {};
  static void* make(char*, char*, int);
};

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

Error message:

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

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

The function *memcpy* is declared as an external "C" function and as a C++ function. It causes a link problem. Indeed, function management behavior differs whether it relates to a C or a C++ function.
When such error happens, the solution is to homogenize declarations, i.e. add “`extern “C” { }”` around previous listed C functions.
Another solution consists in using permissive option `-no-extern-C`. It will remove all declaration extern "C"

- **Standard stubs**

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

```
#include <signal.h>

extern "C" {
  extern void (*signal (int, void (*)(int)))(int);
}

class Copy
```

```
{
public:
  Copy() {};

};
```

Error message:

```
Pre-linking C++ sources ...

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

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

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

It is possible to use compiler prototypes by deactivating standard prototype provided by ANSI. To do so, you have to add the flag `POLYSPACE_NO_STANDARD_STUBS` to the 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`' but instead generate a red error when a range or domain error occurs (see examples with **NTC** checks).

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

3.4. Methodology using the pre-processed .ci files

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

Nevertheless, sometimes the error messages are not sufficient to find the cause of problems. Indeed they do not correspond to common error messages listed above.
PolySpace, as others compilers, transforms a source code to a 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 *.ci* extension and they will help to understand and find precisely the error problem.

1.4.1. 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.

1.4.2. 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 messages come from a dialect problem.
2. Check whether Verify that linking error messages are related or not to:
   a. A C++ Stubbing error which could be resolved by an option (like –no-stl-stubs)
   b. 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`	`# 1 "../sources/Child2.cpp" 2`
	`# 1 "../sources/Child2.h" 1`
`class Test`	`# 1 "../sources/test.h" 1`
`{`	
`public:`	`class Test`
`   Test();`	`{`
`   Test(int val);`	`public:`
	`   Test();`
`   int getVal();`	`   Test(int val);`
`   void setVal(int val);`	
	`   int getVal();`
`   void algorithm(int val, int max);`	`   void setVal(int val);`
`private:`	
`   int m_val;`	
`};`	`private:`
	`   int m_val;`
`# 2 "../sources/Test.cpp" 2`	`};`
`....`	`# 2 "../sources/Child2.h" 2`
	`....`

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

3.5. OS and target specifications

Related subjects :

3.5.1. List of already 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__STRICT_ANSI__ -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D_POSIX_SOURCE -D__STL_CLASS_PARTIAL_SPECIALIZATION -D__GNUC__=2 -D__GNUC_MINOR__=6 -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>

vxWorks	-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 -D__GNUC__=2 -DANSI_PROTOTYPES -DSTATIC= -DCONST=const -D__STDC__ -D__GNUC__=2 -Dunix -D__unix -D__unix__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__ -D__svr4__ -D__SVR4	<product_dir>/cinclude/pst-vxworks.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/pst-visual.h	
Solaris	-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 -D__GNUC__=2 -D__GNUC_MINOR__=8 -D__STDC__ -D__GCC_NEW_VARARGS__ -Dunix -D__unix -D__unix__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__		**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

	-D__svr4__ -D__SVR4		
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.

3.5.2. Target specifications

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

Title	Sizeof (size in bits)									char is	LITTLE/ BIG ENDIAN	ptr diff type
	char	short	int	long	long long	float	double	long double	ptr			
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

#1 The M68k family (68000, 68020, etc.) also has the so-called ColdFire processors as members.

Note: The following table describes target processors that are not fully supported by PolySpace. 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.

Title	Sizeof (size in bits)									char is signed	ptr diff type	Nearest target processor
	char	short	int	long	long long	float	double	long double	ptr			
tms470r1x	8	16	32	32	N/A	32	64	64	32	TRUE	int, long	i386
mpc555	8	16	32	32	64	32	64	64	32	TRUE	int, long	i386
hc-12	8	16	16	32	32	32	32	32	16	TRUE	int	c-167

3.6. Intermediate language errors

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

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
```

Note: This example only explains *where* to find the error list. Moreover, theses errors will be displayed in the Viewer and shown graphically.

3.7. Advanced setup

Related subjects :

3.7.1. 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 analyse 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.

Examples

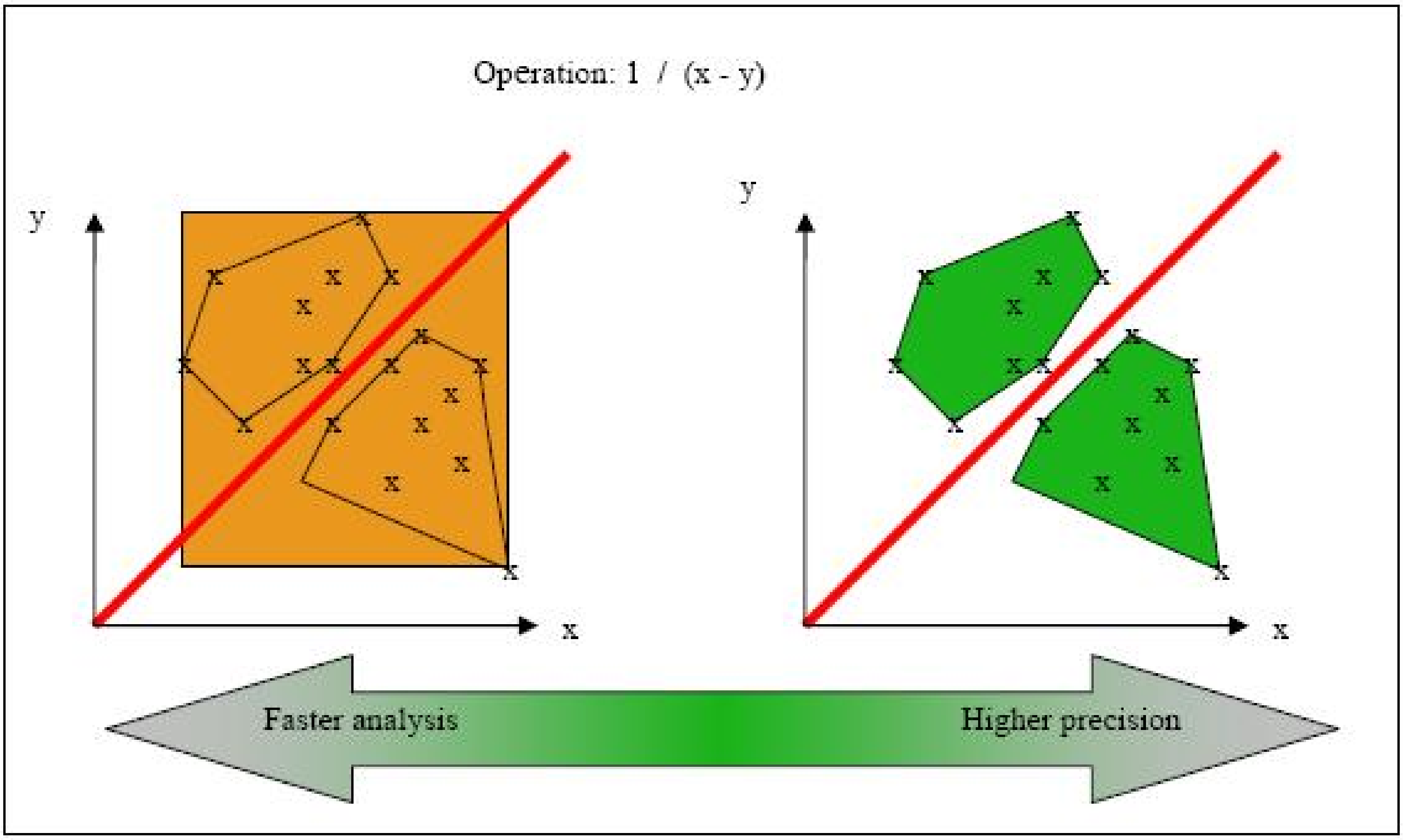
Since increasing the selectivity can bring any of the following benefits – more **red**, more **grey**, or readable **orange** - the following examples will give one example of each. There is no implied link between the approach taken to improve selectivity in an example, and the particular way the improvement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

Related subjects :

3.7.1.1. 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 represent the data internally are reflected in the level of precision to be seen in the results. As illustrated below, the same orange check which results from a low precision analysis will become green when analysed at a higher precision.



Vary the precision rate

3.7.1.2. Apply some manual stubbing

The default behaviour of PolySpace is to automatically stub missing functions members in accordance with their prototypes. If the function takes a pointer as an argument, PolySpace assumes that the stub can write into the contents of this pointer. This default behaviour avoids the occurrence of red errors as the result of incomplete code sets. However, stubbing which accurately reflects the behaviour 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 variable representing time as volatile.

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

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

- The variable is volatile;
- The variable is a task list. Such lists are accounted for by default because all tasks declared with the -task option are automatically modelled as though they have been started. Write a procedure_to_stub by hand if
- The variable is a regular variable read by other procedures or functions.
- A read from a global variable: If you want PolySpace to detect that it is a shared variable, you need to stub a read access. This is easily achieved by copying the value into a local variable.

In general, follow the Data Flow and remember that:

- PolySpace only cares about the C code which is provided;
- PolySpace need not be informed of timing constraints because all possible sequencing is taken into account;
- You can refer to execution hypotheses made by PolySpace for a complete list of constraints.

Example

The following example shows a header for a missing function (which might occur, for example, if the code is a subset of a project.) The missing function copies the value of the 'src' parameter to 'dest' so there would be a division by zero – a run-time error - at run time.

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

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

Default Stubbing	Manual Stubbing	Function ignored

`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 Verifier's default stub, the assembly code is ignored and the division " **/**" is green. The red division "**/**" could only be achieved with a manual stub.

Summary

- Stub manually: to gain precision by restricting return values generated by automatic stubs; to deal with a function which writes to global variables.
- Stub automatically in the knowledge that no run-time error will be ever introduced by automatic stubbing; to minimize preparation time.

Related subjects :

- 3.7.1.2.1. Examples: specification
- 3.7.1.2.2. Coloured source code example
- 3.7.1.2.3. Specify the call sequence
- 3.7.1.2.4. Constraint for data
- 3.7.1.2.5. Recoding of some specific functions

[Previous](#) [Back to table of contents](#) [Next](#)

3.7.1.2.1. 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 won't 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 parameter. cnx_id, port and data remain unchanged
Param in/out	None	
returns	Int	0 : failure of connection establishment 1 : success

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

Here are the reasons why:

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

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

3.7.1.2.2. Coloured source code example

```
typedef struct _m { int a,b; } T;
void send_message(T *);

void main(void)
{
int i;
T x = {10, 20};

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

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

Manual stubbing examples for send_message:

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

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

3.7.1.2.3. *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.

Coloured 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 sequence between g and f: if g is called first, the assignment is green, otherwise it’s red. Thanks to the exact call order, an attempt to place 300 in a char fails, displaying a red.

Example of call sequence

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

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

3.7.1.2.4. Constraint for data

- **Default behaviour of global data**

Initially, consider how PolySpace handles 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 doesn’t just apply to global variables; it is equally appropriate where such a variable is passed as a parameter to a function, or where return values from stubbed functions are under consideration.

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

- **Constraining the data**

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

The technique therefore requires

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

- **Applying the technique**

Create the range setting stubs:

- create 6 functions for each type (8,16 or 32 bits, signed and unsigned)
- declare 6 global volatile variables for each type
- write the functions which returns sub-ranges (an example follows)

Gather the initialisation of all relevant variables into a single procedure

Call this procedure at the beginning of the main. This should replace any existing initialisation 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();
      // ...
      }
}
```

3.7.1.2.5. Recoding of some specific functions

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

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

Identify in the modules:

- API which read global variables through pointers

Replace this API

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

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

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

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

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

with this one

```
char Constant_1 ;

#define read_calibration(in,index) *in

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

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

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

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

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

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

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

```
Char read_from_data_my_global2(void)
{ }
```

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

3.7.2. Approximations made by PolySpace

Related subjects :

Previous Back to table of contents Next

3.7.2.1. Volatile variables

Volatile variables are potentially uninitialised and their content is always full range.

```
int volatile_test (void)
{
  volatile int tmp;
  return(tmp);  // NIV orange: the variable content is full range[-2^31;2^31-1]
}
```

In the case of a global variable the content would also be full range, but the NIV check would be **green**.

3.7.2.2. Structures with volatile fields

In this example, although only the b field is declared as volatile, in practice any read access to the “a” field will be full range and **orange**.

```
typedef struct {
  int a;
  volatile int b;
} Vol_Struct;
```

3.7.2.3. Absolute addresses

Both reading from, and writing to, an absolute address leads to warning checks on the pointer dereference. An absolute address is considered as a volatile variable.

```
Val = *((char *) 0x0F00); // NIV and IDP orange: access to an
absolute address
```

3.7.2.4. Pointer comparison

PolySpace is a static tool analysing source code. Memory management concerns dynamic considerations, and the characteristics of particular compilers and targets. PolySpace therefore doesn’t consider where objects are actually implanted in memory

```
int *i, *j, k;
i = (int *) 0x0F00;
j = (int *) 0x0FF0;

if ( i < j) // the condition can be true or false
  k = 12; // this line is reachable
else
  k = 23; // this line is reachable too.
```

It’s the same situation if “i” and “j” points to real variable

```
i = & one_variable;
j = & another_one;
if ( i < j) // the condition can still be true or false
```

3.7.2.5. 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 because behaviour 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.

```
char x, y;
x = 0x8F;
y = x << 3 ; // OVFL and UNFL Warnings
```

Previous Back to table of contents Next

3.7.2.6. Some bitwise operators

PolySpace results are not equally precise with all bitwise operators - AND, OR, XOR, and NOT (resp. &, |, ^,))

```
int random_uint(void);

void test (void)
{       unsigned int var1, var2, var3;
   var1=0; var2=0;

  // precision with zero on values with AND bitwise operator
  var3= 0x01 & var2;
  if (random_uint()) assert(var3==0);    // ASRT Checked
  var3= 0x02 & 0xF3;
  if (random_uint()) assert(var3==0x02); // ASRT checked
  // Full range with other values
  var3 = random_uint();
  var3 = var3 & 0x02;
  if (random_uint()) assert(var3==0x02 || var3==0); // ASRT Warning

  // Full range on values with OR bitwise operator
  var3=var1|var2;
  if (random_uint()) assert(var3==0);       // ASRT Warning
  if (random_uint()) assert(var3!=0);       // ASRT Warning

  // Full range on values with XOR bitwise operator
  var3=var1^var2;
  if (random_uint()) assert(var3==0);       // ASRT Warning
  if (random_uint()) assert(var3!=0);       // ASRT Warning

  // precision with zero values on NEGATIVE bitwise operator
  var3 = ~var1;
  if (random_uint()) assert(var3==0xFFFFFFFF); // ASRT checked
      // precision on values with NEGATIVE bitwise operator
      var3 = ~0xAE;
      if (random_uint()) assert(var3==0xFFFFFF51); // ASRT checked
}
```

3.7.2.7. Bitfieds

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

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

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

3.7.2.8. Float loops

Values on constructions are less precise when floats are used in loops.

```
int i;
double X = 0.0;

// less precision on float evaluation in loops
for (i = 0 ; i < 6; i++)
  X = X + 10.56;         // OVFL warning
// VOA says 10.561 >= EXPR >= 10.559  OR EXP >= 21.119
```

3.7.2.9. 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.

```
void p_task1(void)
{
  begin_cs();
  X = 0;
  if (X) {
    Y = X;               // Verified NIV, even it should be grey
    assert (Y == 12); // Warning assert, even it should be grey
  }
  end_cs();
}

void p_task2(void)
{
  begin_cs();
  X = 12;
  Y = X + 1;      // Verifier considers [X==1] or [X==13]
  if (Y == 13)
    Y = 14;
  else
    Y = X - 1 ;   // Verified checks even it should be grey
  end_cs();
}
```

3.7.2.10. Array of function pointers

In the following example, PolySpace results show an orange check despite the test for a NULL function pointer test. However, it does accurately track the functions being called.

```
ptr_func array_func[] = {
 f1,
 f2,
 NULL,
};

void main(void)
{
 int   i;

 i = 0;
 while (i < 3) {
  if (array_func[i] != NULL)
     array_func[i](); // COR Warning [function must point to a valid function]
      i++;
 }
```

3.7.2.11. 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.

```
double res;

res = sin(3.141592654);
assert(res == 0.0);   // VOA says  [-1..1]

res = asin(0.0);
assert(res == 0.0);   // VOA Always in [-pi/2..pi/2]

res = acos(0.0);
assert(res == 0.0);   // VOA always in [0..pi]
```

3.7.2.12. Unions

It is recognised nonetheless that there are situations in which the careful use of unions is desirable in constructing an efficient implementation. Nevertheless, the kinds of implementation behaviour 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 compiler dependant. Conversions from one type a union to another will cause a loss of precision on two checks:

- Is the other field initialized? **Orange** NIV
- What is the content of the other field? **Full range 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
```

Previous Back to table of contents Next

3.7.2.13. Loop exit conditions

PolySpace is more precise in loops where a test other than "does not equal" is used. Consider the loop index exit `value` in the following examples.

The orange check in this example …:

```
x = 0;
While (x != value)
{
   …;
   x++;
}
```

is not evident here…:

```
While (x <= value)
…
x++;
```

3.7.2.14. Constant pointer

To increase PolySpace precision where pointers are analysed, replace

```
const int *p = &y;
```

with

```
#define p (&y)
```

3.7.3. Variables

Related subjects :

- **3.7.3.1. How are variables initialized?**
- **3.7.3.2. Data and coding rules**
- **3.7.3.3. Variables: Declaration and definition**
- **3.7.3.4. How can I model variable values external to my application?**

3.7.3.1. How are variables initialized?

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

- **Extern**

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

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

Refer to "Reviewing code coloured by PolySpace " for more information on colour propagation.

For extern structures containing field(s) of type "pointer to function", this principle leads to red errors in the viewer. In this case, the resulting default behaviour 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`

3.7.3.2. Data and coding rules

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

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

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

3.7.3.3. Variables: Declaration and definition

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

Declaration

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

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

Definition

- for a function : the body of the function has been written : `int f(void) { return 0; }`
- for a variable : a part of memory has been reserved for the variable : `int x; or extern int x=0;`

When a variable is not defined, the `-allow-undef-variable` is required to start the analysis. Where that option is used, PolySpace will consider the variable to be initialized, and to potentially take any value in its full range (see PolySpace and variables initialisation section).

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

3.7.3.4. How can I model variable values external to my application?

There are three main considerations.

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

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

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

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

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

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

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

3.7.4. Types promotion

Related subjects :

3.7.4.1. 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 behaviour.

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.

3.7.4.2. What are the promotions rules in operators?

Knowledge of the rules applying to the standard operators of the C language will help you to analyse 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 analysing 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     {
      char c1 = random_char();
      char c2 = random_char();

      c1 = (char)((int)c1 + (int)c2);  // Warning UOVFL: due to integral promotion
    }
```

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

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

Integral promotion respects the size hierarchy of basic types:

- *`char`* *(signed or not)* and *`signed short`* are promoted to *`int`*.
- *`unsigned short`* is promoted to *`int`* only if *`int`* can represent all the possible values of an *`unsigned short`*. If that is not the case (perhaps because of a 16-bit target, for example) then *`unsigned short`* is promoted to *`unsigned int`*.
- Other types like *`(un)signed int`*, *`(un)signed long int`* and *`(un)signed long long int`* promote themselves.

3.7.5. Built-in functions

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

Stubs of stl functions

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

Note that 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, ctype.h, errno.h, locale.h, math.h, setjmp.h` ('`setjmp`' and '`longjmp`' functions are partially implemented – see `<polyspaceProduct>/cinclude/__polyspace__stdstubs.c`), `signal.h` ('`signal`' and '`raise`' functions are partially implemented – see `<polyspaceProduct>/cinclude/__polyspace__stdstubs.c`), `stdio.h, stdarg.h, stdlib.h, string.h`, and `time.h.`

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

Most of the time theses functions can be redefined and analysed 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 redefined: `malloc(), calloc(), realloc(), valloc(), alloca(), __built_in_malloc() and __built_in_alloca()`.

4. PolySpace class analyzer process

This paragraph 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 PolySpace class analyzer please refer to "Getting started" section.

Related subjects :

4.1. Why providing 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 errors 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 removing runtime errors at compilation time. PolySpace will simulate all the possible use of a class, by:

1. Creating objects using all constructors (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 previous object (if they exist).

4.2. 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

```
#include "stack.h"

stack::stack ()
{
  toparray = -1;
  for (int i = 0 ; i < MAXARRAY; i++)
    array[i] = 0;
}

int stack::top (void)
{
  int i = toparray;
  return (array[i]);
}

bool stack::isempty (void)
{
  if (toparray >= 0)
    return false;
  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    }
```

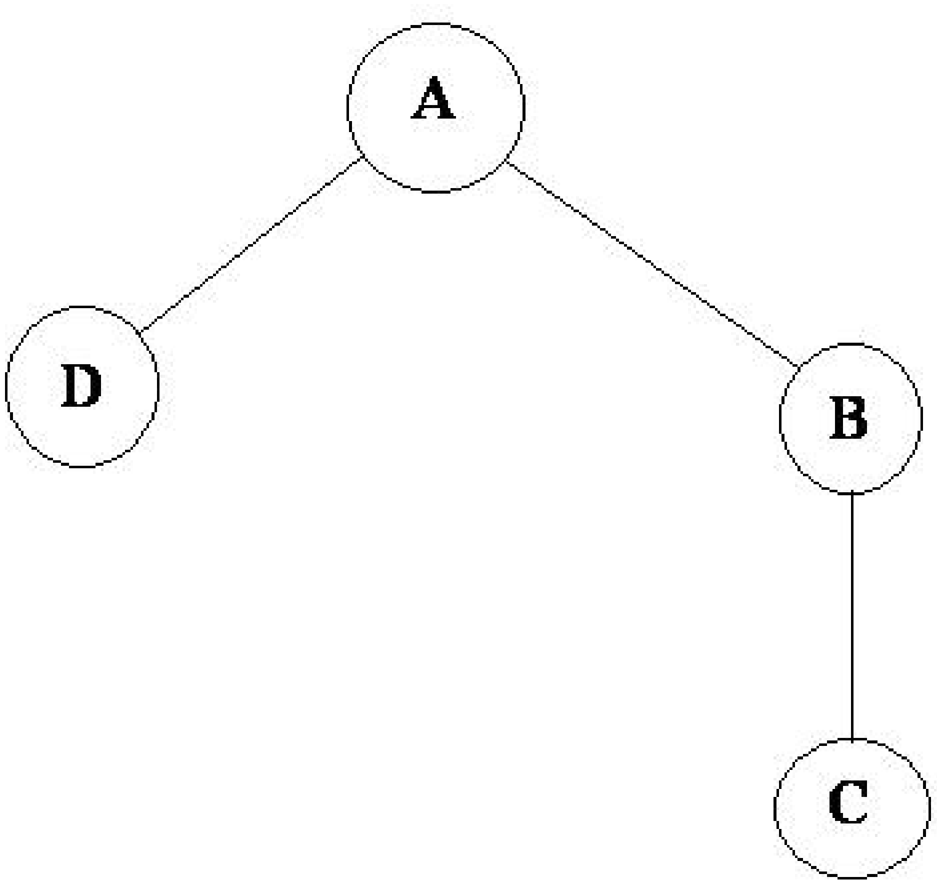
The Class analyzer calls the constructor and then all methods in any orders many times.
The analyze of this class with PolySpace Class-Analyzer highlights 2 problems, as shown in previous Viewer results:

- 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.

4.3. Simple inheritance

Consider classes as follow:



A is the base class of B and D.
B is the base class of C.

In such case PolySpace allows to do the following analysis:

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

2. The B class can be analysed 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 analysed 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.

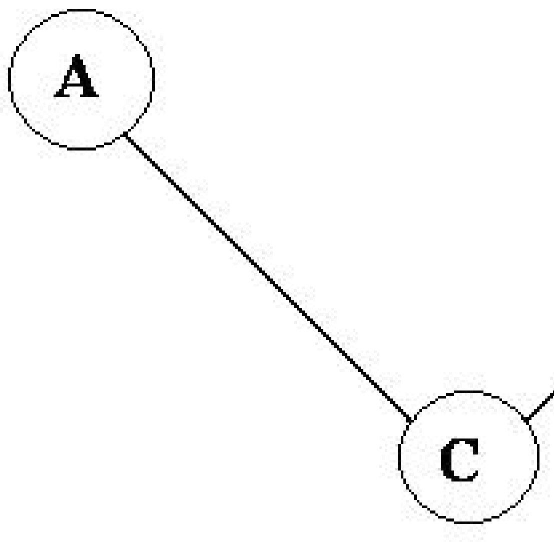
4. The C class can be analysed by providing C code, B class declaration and A class declaration. In this case A and B codes will be stubbed automatically.

5. The C class can be analysed 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 analysing A, B and C class as long as they do not use the class (e.g. member type) or need it (e.g. inherit).

4.4. Multiple inheritance

Consider classes as follow:



A and B are C base classes.

In such case PolySpace allows to do the following analysis:

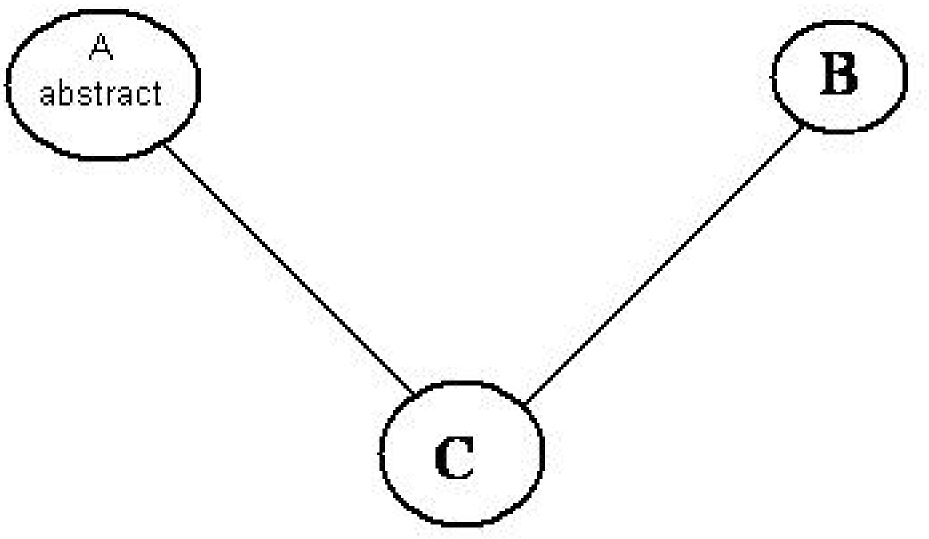
1. A and B classes can be analysed separately just by providing there own code to PolySpace. This corresponds to the previous simple class paragraph.

2. The C class can be analysed 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 analysed 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.

4.5. Abstract class

Consider classes as follow:

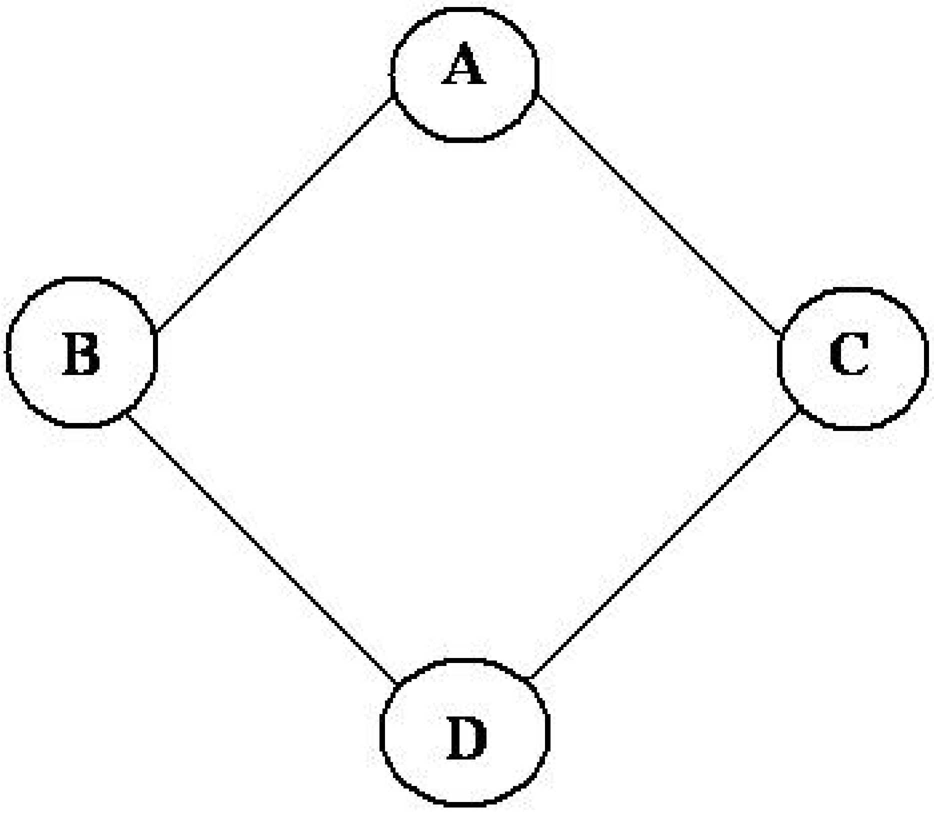


A is an abstract class
B is a simple class.
A and B are C base classes.
C is not an abstract class.

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 analysed in the same way as in previous paragraph.

4.6. Virtual inheritance

Consider classes as follow:



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.

4.7. Other types

- Template

A template class can not be analyzed directly. But a class instantiating a template can be analyzed by PolySpace. **Note**: if only the template declaration is provided, missing functions definitions will 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 create a specialization of the template:

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

and use option `–class-analyzer` `my_template.` It will analyze one instance of the template.

- Class integration

Let’s 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 class. If some definitions are missing PolySpace will stub them automatically.

5. PolySpace C++ add-in for Visual Studio

This paragraph describes the usage of PolySpace for C++ while integrated in the Microsoft Visual C++ . NET (see The PolySpace Install guide in `<PolySpaceInstallCommon>/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 project configuration file derived from your Visual project settings.
- A direct launching of C++ classes and files analyses from Visual IDE.
- A report of PolySpace compilation findings back to the IDE.

Related subjects :

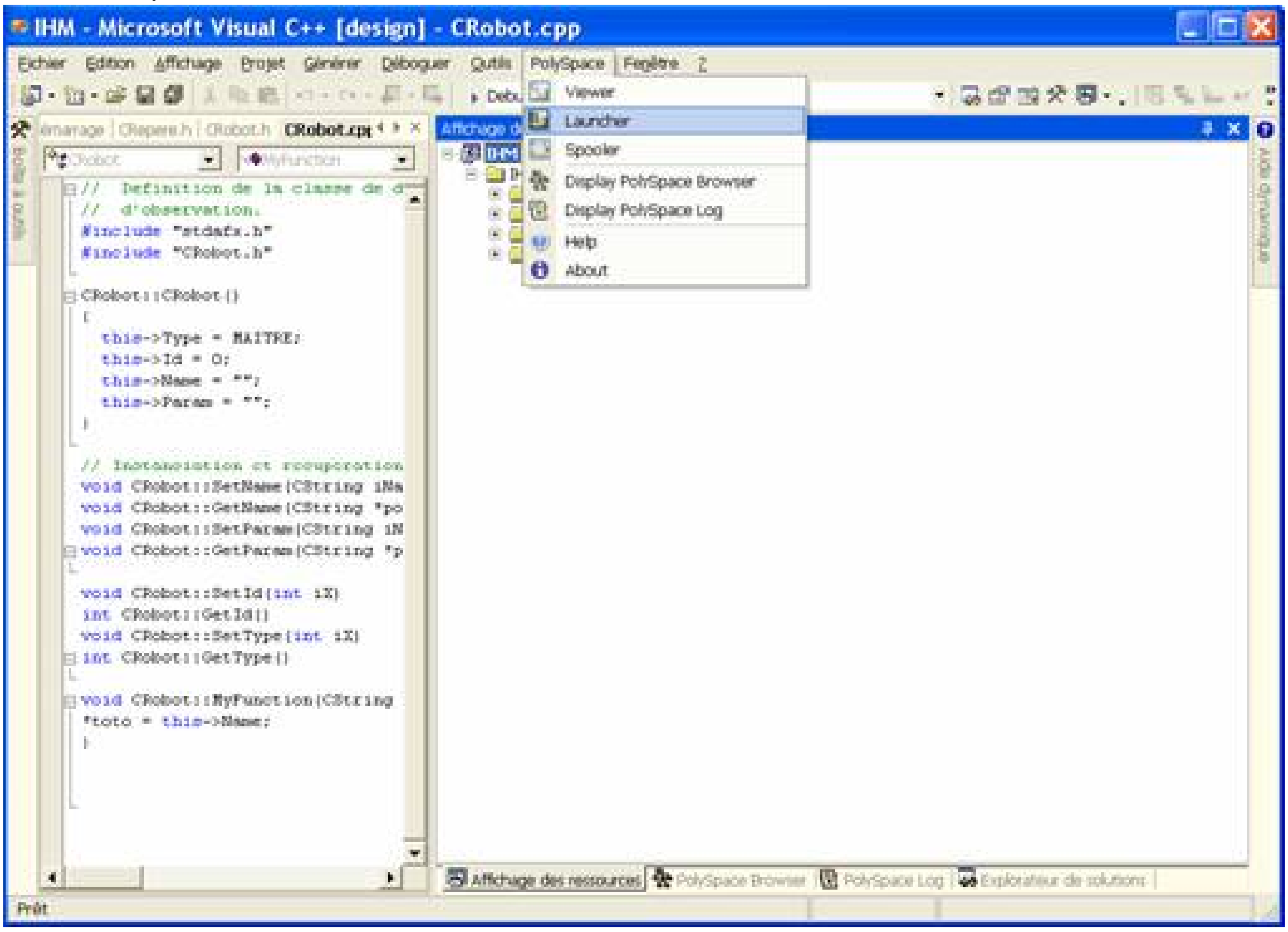
5.1. PolySpace usage inside Visual Studio

The PolySpace for Visual .NET plug-in allows launching C++ analyses inside the Visual C++ IDE whatever those analyses are local or remote.
Launching options are integrated within the Visual editor through a PolySpace menu and a toolbox.
Note that 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.

Related subjects :

5.1.1. PolySpace Parameters Inside Visual Studio

When the PolySpace/Visual C++ plug-in has been installed, a PolySpace toolbar, a PolySpace menu and two tabs 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.



```
//   Definition de la classe de d
//   d'observation.
#include "stdafx.h"
#include "CRobot.h"

CRobot::CRobot()
{
   this->Type = MAITRE;
   this->Id = 0;
   this->Name = "";
   this->Param = "";
}

// Instanciation et recuperation
void CRobot::SetName(CString iNa
void CRobot::GetName(CString *po
void CRobot::SetParam(CString iN
void CRobot::GetParam(CString *p

void CRobot::SetId(int iX)
int CRobot::GetId()
void CRobot::SetType(int iX)
int CRobot::GetType()

void CRobot::MyFunction(CString
*toto = this->Name;
}
```

PolySpace Toolbar, tabs and PolySpace menu in Visual Studio

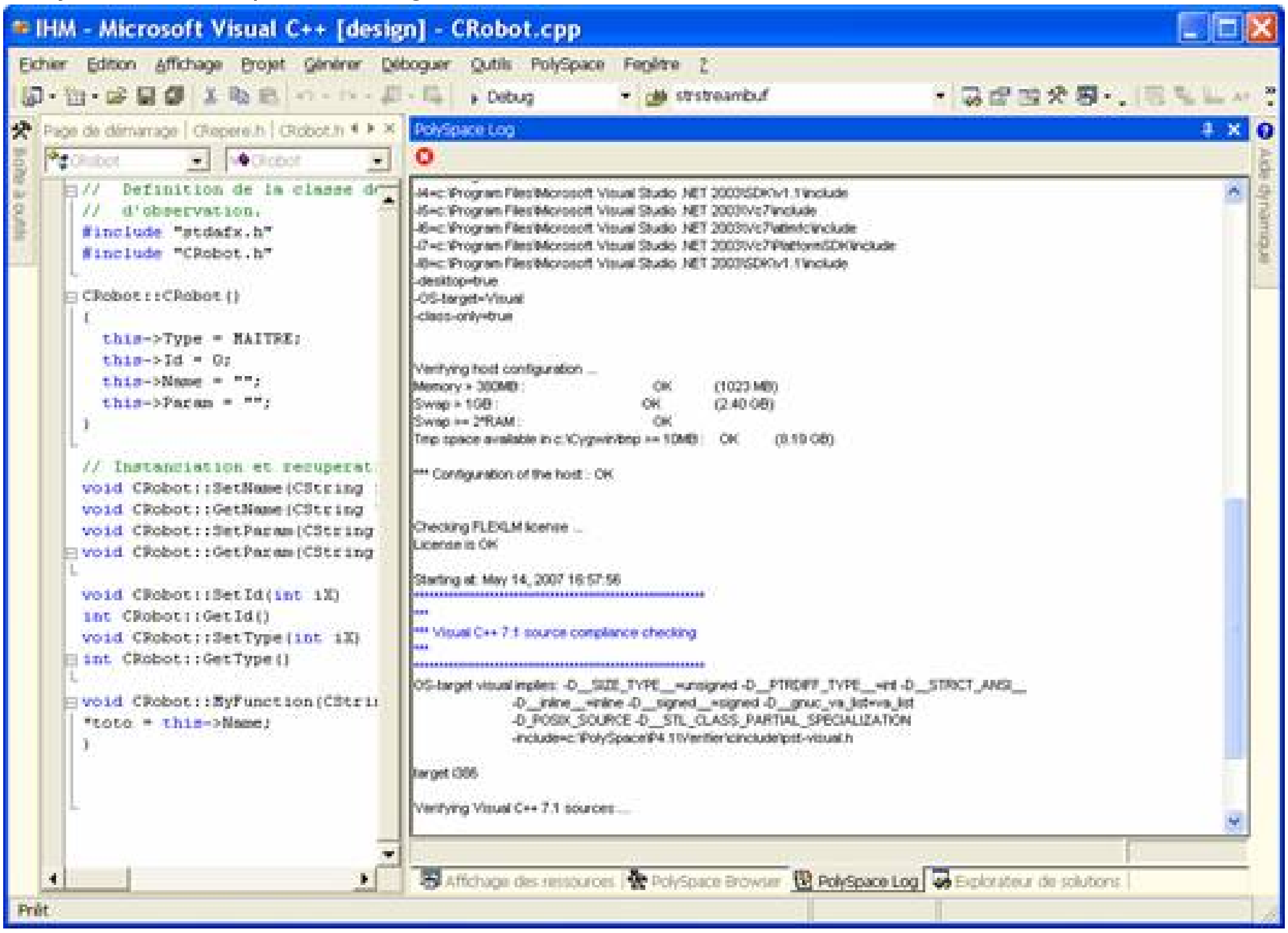
The PolySpace Menu and Toolbar

- Click on "`Launcher`" (or use "PolySpace> Launcher" menu) to open the PolySpace launcher on the **<u>last</u>** 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.
- Click on "`Spooler`" (or use "PolySpace> Spooler") to start the PolySpace spooler. This tool is used to manage PolySpace jobs that are performed on remote servers.
  See "Getting Started section in the PolySpace for C++ documentation".

- Click on “Viewer” (or use “PolySpace> Viewer” menu) 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.
- Display “Display `PolySpace Browser`” and “`Display PolySpace Log`” allow to see other Tab (see below).
- Click on “Help” to start the PolySpace for C++ documentation (PDF format).
- Click on “About” to display the release number of the PolySpace for Visual Studio plug-in.

The “`PolySpace Log`” tab

When an analysis is launched, the “PolySpace Log Window” tab displays the PolySpace progress report. Compilation error if any, are highlighted as a hyper-link. Click on those links to display the corresponding file and line number that includes the compilation error.
Click on red X just below the “`PolySpace Log`” header to stop the PolySpace analysis running locally. If the analysis has been remotely spooled this option will only work during the compilation phase before the analysis is sent to the server. However, you can use the “`PolySpace>Spooler`” button and stop the analysis from the spooler dialog.



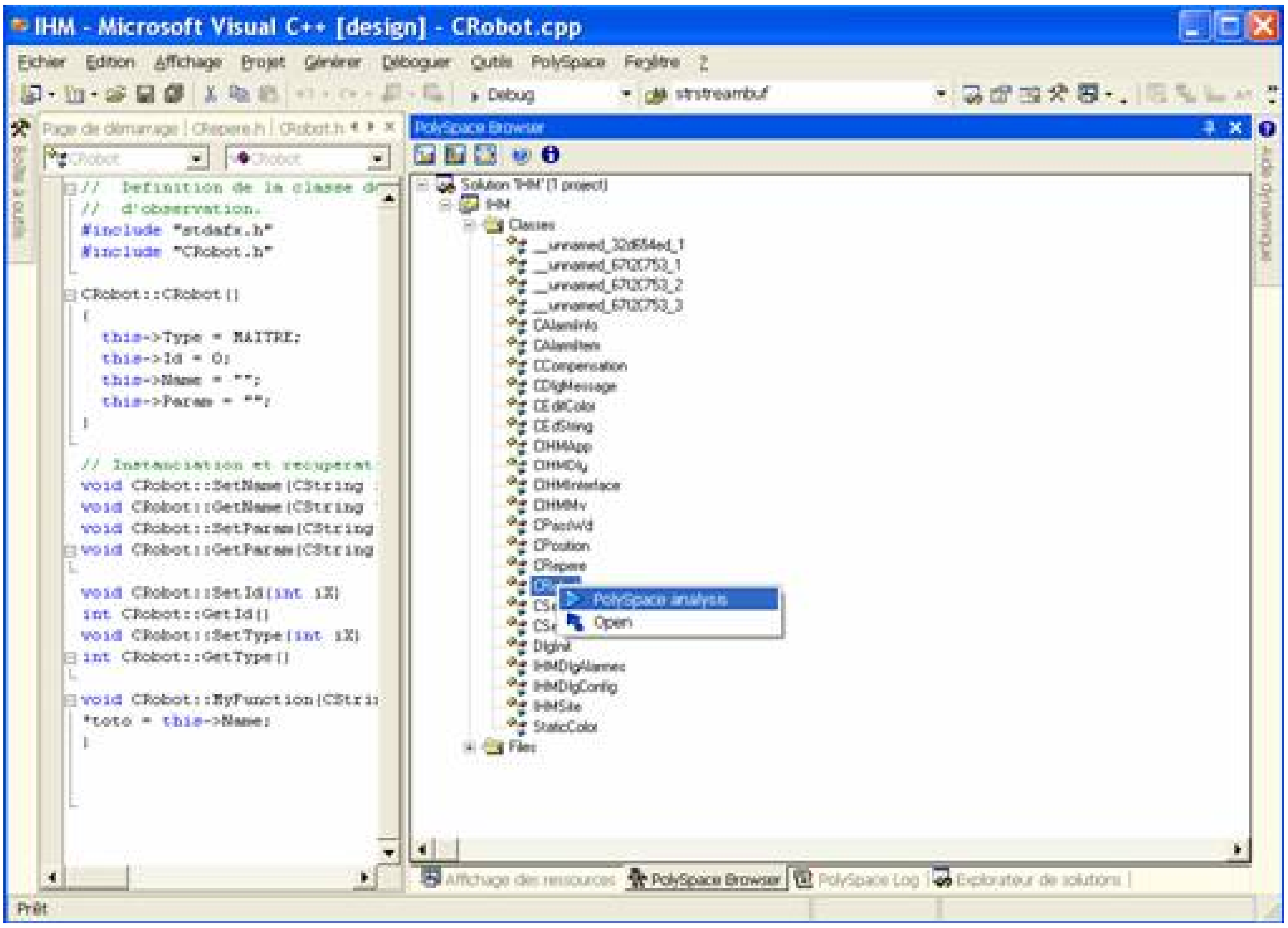
```
//   Definition de la classe de
//   d'observation.
#include "stdafx.h"
#include "CRobot.h"

CRobot::CRobot()
{
   this->Type = MAITRE;
   this->Id = 0;
   this->Name = "";
   this->Param = "";
}

// Instanciation et recuperat
void CRobot::SetName(CString
void CRobot::GetName(CString
void CRobot::SetParam(CString
void CRobot::GetParam(CString

void CRobot::SetId(int iX)
int CRobot::GetId()
void CRobot::SetType(int iX)
int CRobot::GetType()

void CRobot::MyFunction(CStri
*toto = this->Name;
}
```

```
-I4=c:\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\include
-I5=c:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\include
-I6=c:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\atlmfc\include
-I7=c:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\PlatformSDK\include
-I8=c:\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\include
-desktop=true
-OS-target=Visual
-class-only=true


Verifying host configuration ...
Memory > 300MB :                        OK        (1023 MB)
Swap > 1GB :                           OK        (2.40 GB)
Swap >= 2*RAM :                         OK
Tmp space available in c:\Cygwin\tmp >= 10MB :    OK        (8.19 GB)

*** Configuration of the host : OK


Checking FLEXLM license ...
License is OK

Starting at: May 14, 2007 16:57:56
*******************************************************
***
*** Visual C++ 7.1 source compliance checking
***
*******************************************************
OS-target visual implies: -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
                -include=c:\PolySpace\P4.1\Verifier\cinclude\pst-visual.h

target i386

Verifying Visual C++ 7.1 sources ...
```

The PolySpace Log Window

The “`PolySpace Browser`” tab

The “PolySpace Browser” tab displays classes and files of the project on which PolySpace can perform analyses. A dynamic link is established between classes and files of the project and the “PolySpace browser” window. All classes available in the project are accessible in the browser (see picture below)



```
//   Definition de la classe de
//   d'observation.
#include "stdafx.h"
#include "CRobot.h"

CRobot::CRobot()
{
   this->Type = MAITRE;
   this->Id = 0;
   this->Name = "";
   this->Param = "";
}

// Instanciation et recuperat
void CRobot::SetName(CString
void CRobot::GetName(CString
void CRobot::SetParam(CString
void CRobot::GetParam(CString

void CRobot::SetId(int iX)
int CRobot::GetId()
void CRobot::SetType(int iX)
int CRobot::GetType()

void CRobot::MyFunction(CStri
*toto = this->Name;
}
```

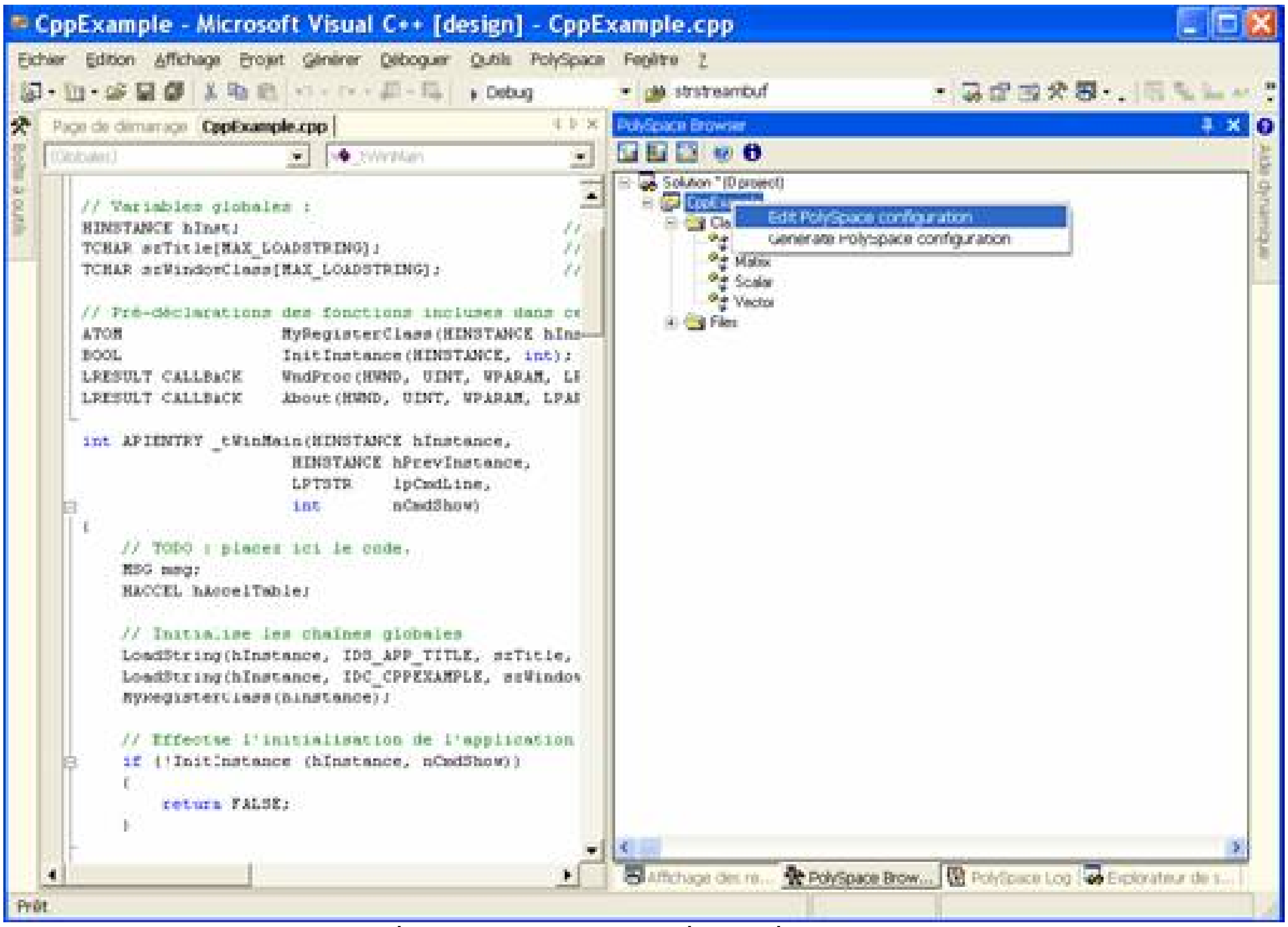
The PolySpace Browser window

Select a class or a file and apply right click to get class specific pop-up menu:

- Select the "`PolySpace Analysis`" option to launch an analysis on all selected classes and files in the "PolySpace Browser" tab.
- When a class is selected, the "`Open`" option opens the file which contains the declaration of the class into the Visual Studio IDE. When a file is selected, this option opens it in Visual Studio.

If you right click on the project name, another pop-up menu appears (see below):

- Select the "`Edit PolySpace Configuration`" option to launch the PolySpace Launcher on the associated cfg file of the Visual project. It updates the PolySpace Configuration file located in the project directory. It is possible to add some more options than theses implicitely extracted from the Visual project (see "default options").
- Select the "`Generate PolySpace configuration`" field menu to update the PolySpace Configuration file. Be aware that it will remove the previous cfg file if it exists.



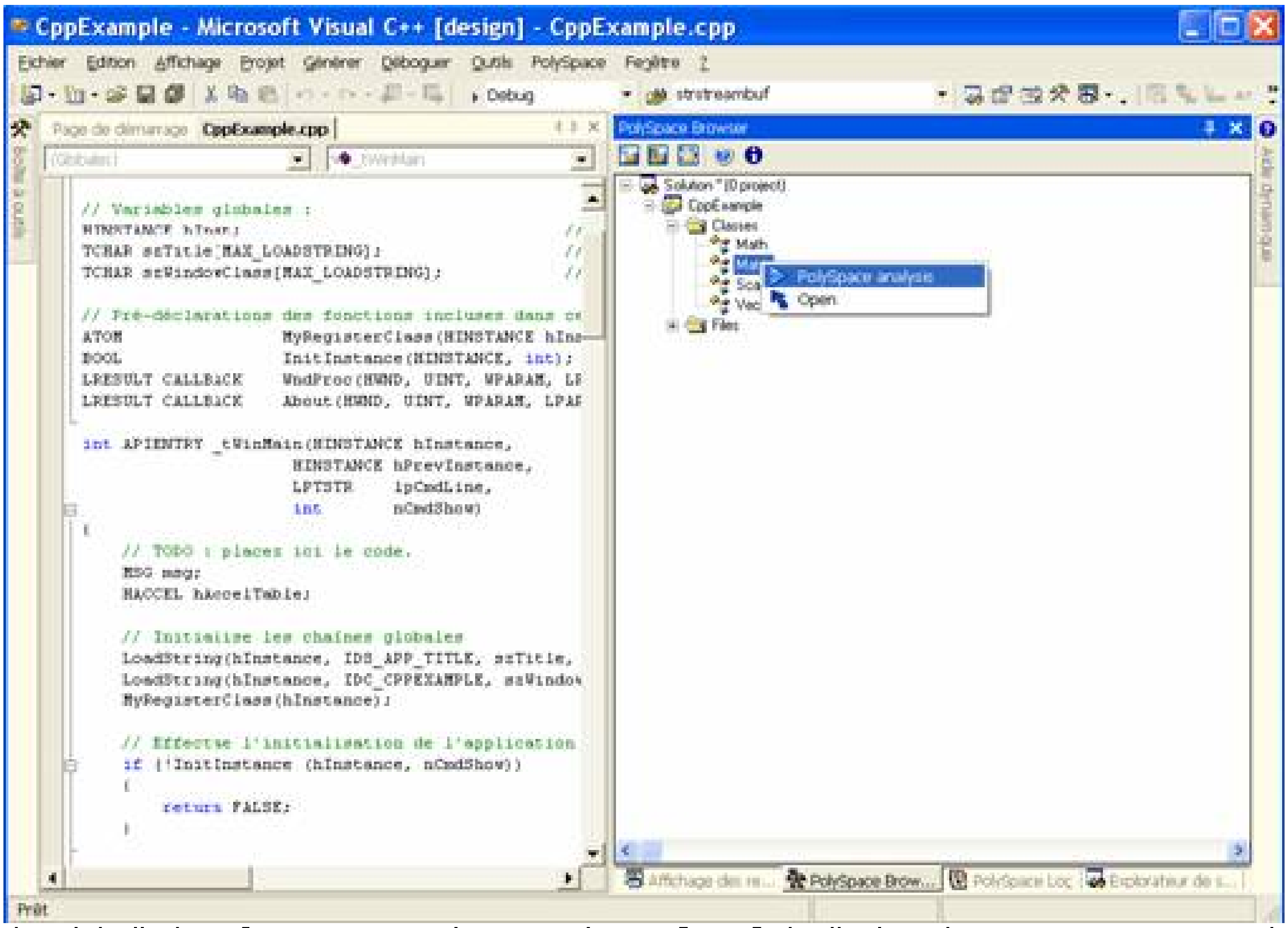
```
// Variables globales :
HINSTANCE hInst;                                //
TCHAR szTitle[MAX_LOADSTRING];                  //
TCHAR szWindowClass[MAX_LOADSTRING];            //

// Pré-déclarations des fonctions incluses dans ce
ATOM                MyRegisterClass(HINSTANCE hIns
BOOL                InitInstance(HINSTANCE, int);
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LP
LRESULT CALLBACK    About(HWND, UINT, WPARAM, LPAR

int APIENTRY _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
{
    // TODO : placez ici le code.
    MSG msg;
    HACCEL hAccelTable;

    // Initialise les chaînes globales
    LoadString(hInstance, IDS_APP_TITLE, szTitle,
    LoadString(hInstance, IDC_CPPEXAMPLE, szWindow
    MyRegisterClass(hInstance);

    // Effectue l'initialisation de l'application
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }
```

The pop-up menu on the project name

5.1.2. Your first PolySpace Class analysis inside Visual Studio

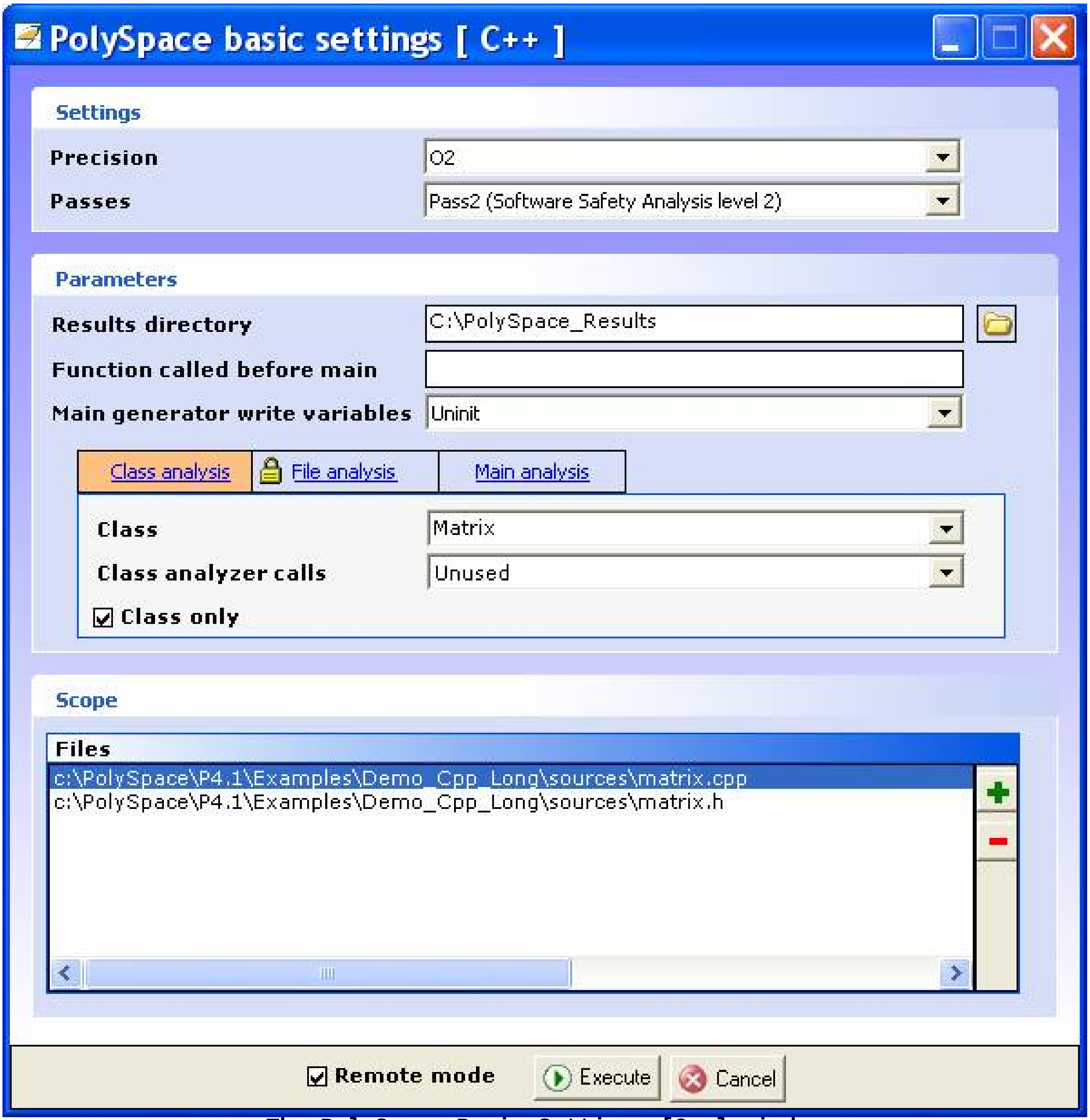
Start your first analysis step by step:

- Create a new project space (use "`File>New>Project >New`" menu), select "`Project Console Win32`", type the name "`CppExample`" and save it under an adequate location. For example: "`C:\PolySpace\Visual`". Some files and a Project Console Win32 is created.
- Add "`matrix.cpp`" and "`matrix.h`" located in `<PolySpaceProduct>/Examples/Demo_Cpp_Long/sources` to the "`CppExample`" project (goto tab "`Browse the solution`", right click into the project name and use "`Add>Add existing element…`" pop-up item).
- Go into "`PolySpace Browser`", expands "`CppExample>CppExample>Classes`", select the "`Matrix`" class and right click on "`PolySpace Analysis`" (See next figure).



```
// Variables globales :
HINSTANCE hInst;                                //
TCHAR szTitle[MAX_LOADSTRING];                  //
TCHAR szWindowClass[MAX_LOADSTRING];            //

// Pré-déclarations des fonctions incluses dans ce
ATOM                MyRegisterClass(HINSTANCE hIns
BOOL                InitInstance(HINSTANCE, int);
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LP
LRESULT CALLBACK    About(HWND, UINT, WPARAM, LPAR

int APIENTRY _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
{
    // TODO : placez ici le code.
    MSG msg;
    HACCEL hAccelTable;

    // Initialise les chaînes globales
    LoadString(hInstance, IDS_APP_TITLE, szTitle,
    LoadString(hInstance, IDC_CPPEXAMPLE, szWindow
    MyRegisterClass(hInstance);

    // Effectue l'initialisation de l'application
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }
```

A dialog box labelled "`PolySpace Basic Settings [C++]`" is displayed so you can set some default pieces of information including precision of the current analysis and a result folder (See next figure).


PolySpace basic settings [C++]

Settings

Precision: O2

Passes: Pass2 (Software Safety Analysis level 2)

Parameters

Results directory: C:\PolySpace_Results

Function called before main:

Main generator write variables: Uninit

Class analysis | File analysis | Main analysis

Class: Matrix

Class analyzer calls: Unused

☑ Class only

Scope

Files

```
c:\PolySpace\P4.1\Examples\Demo_Cpp_Long\sources\matrix.cpp
c:\PolySpace\P4.1\Examples\Demo_Cpp_Long\sources\matrix.h
```

☑ Remote mode

The PolySpace Basic Settings [C++] window

Then select some basic options for the current class analysis:

- Sub window "`Settings`" allow to select precision (-0/-quick) and level of analysis (–to);
- Sub window "`Parameters`" allow to select:
  - the results directory (-results-dir); the name of a function, if any, called before all functions (-function-call-before-main) and the type of initialisation for global variables (-main-generator-writes-variables).
  - By default the "`Class analysis`" Tab enables the class analysis with default options: the class to analyse (-class-analyzer) and associated options which can change behaviour of the analysis (-class-only and –class-analyzer-calls).
  - You can chose a partial integration analysis by using the "`Main analysis`" tab allowing to choose the name of the "main" (-main).

- You can also choose a file analysis (by ticking the “File analysis” Tab) with associated option (-main-generator-calls).

- Sub window “`Scope`” allows to give the list of files and classes which are used with the analysis. When more than one class is selected, PolySpace selects in an automatic way the list of cpp files from the project to add to the analysis.

Then, Click on “`Execute`” to proceed. The progress of the analysis can be followed in the “`PolySpace Log`” window and later using the PolySpace Spooler if remote launching has been enabled.

5.1.3. The configuration file and default options

Some options are set by default and some others are directly extracted from the Visual project and set in the associated PolySpace configuration file.

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

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

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

Standard PolySpace options 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 into account during the analysis, except the list of options set in the “`PolySpace Basic Setting [C++]`” window.

5.2. Launching an analysis on the 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.

6. PolySpace UML Link RH

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 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.

Related subjects :

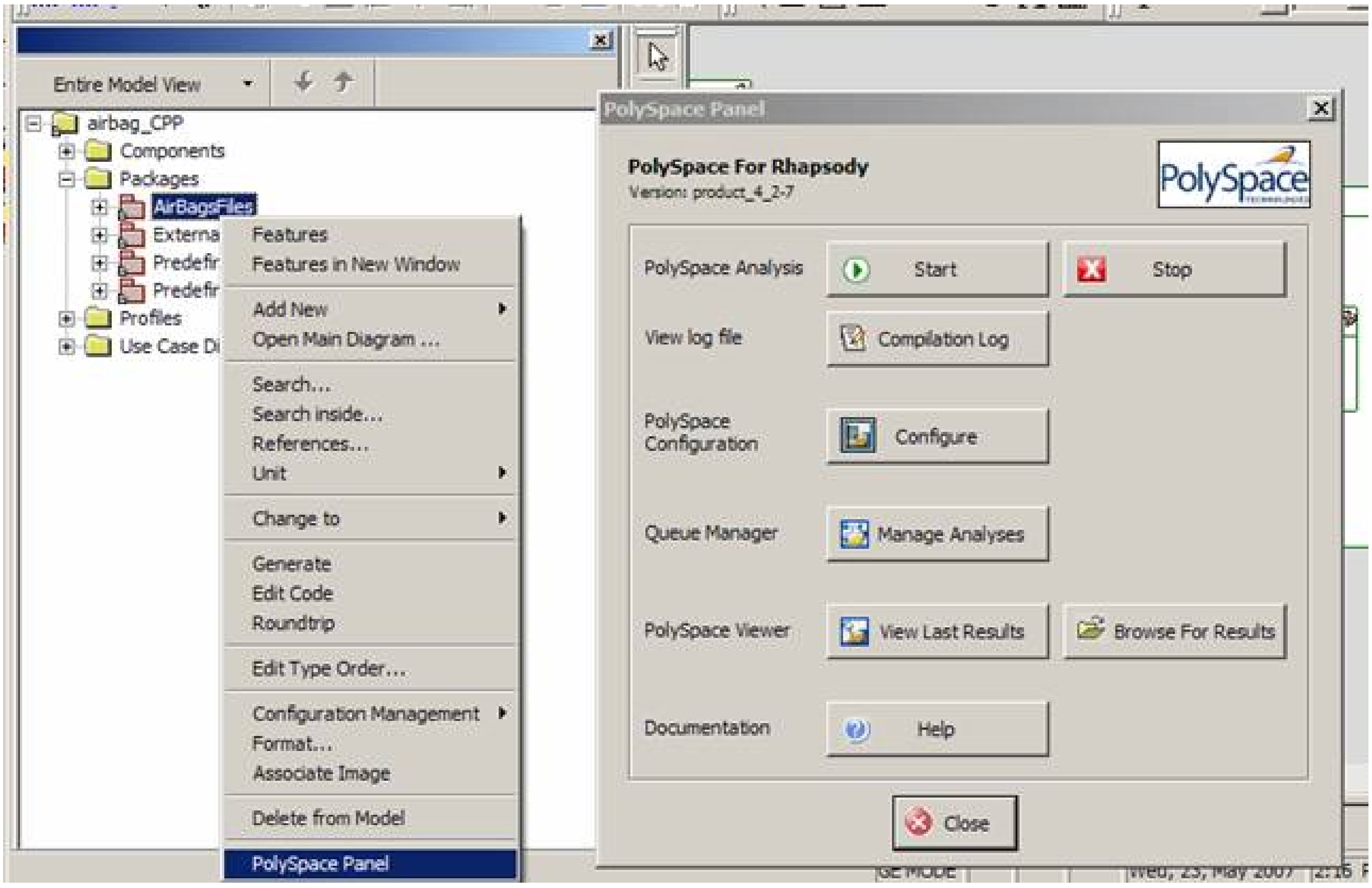
6.1. Getting Started

The getting started guide takes you through the steps required to analyze a model.
Note that the PolySpace plug-in has already been integrated into the example model. Before other models can be analyzed the plug-in may need manually installing into the Rhapsody project directory.
During the getting started the following conventions will be used: “`<PolySpaceInstallCommon>`” will refer to the installation location of the PolySpace common folder.

Related subjects :

6.1.1. Step 1 - Open and display the example airbag model

1. Open the `airbag_CPP.rpy` model in “`<PolySpaceInstallCommon>/PolySpaceUMLLink/example`”.
1. Open the PolySpace Panel by expanding the package list and right click on “`AirBagFiles`”, select “`PolySpace Panel`” from menu.

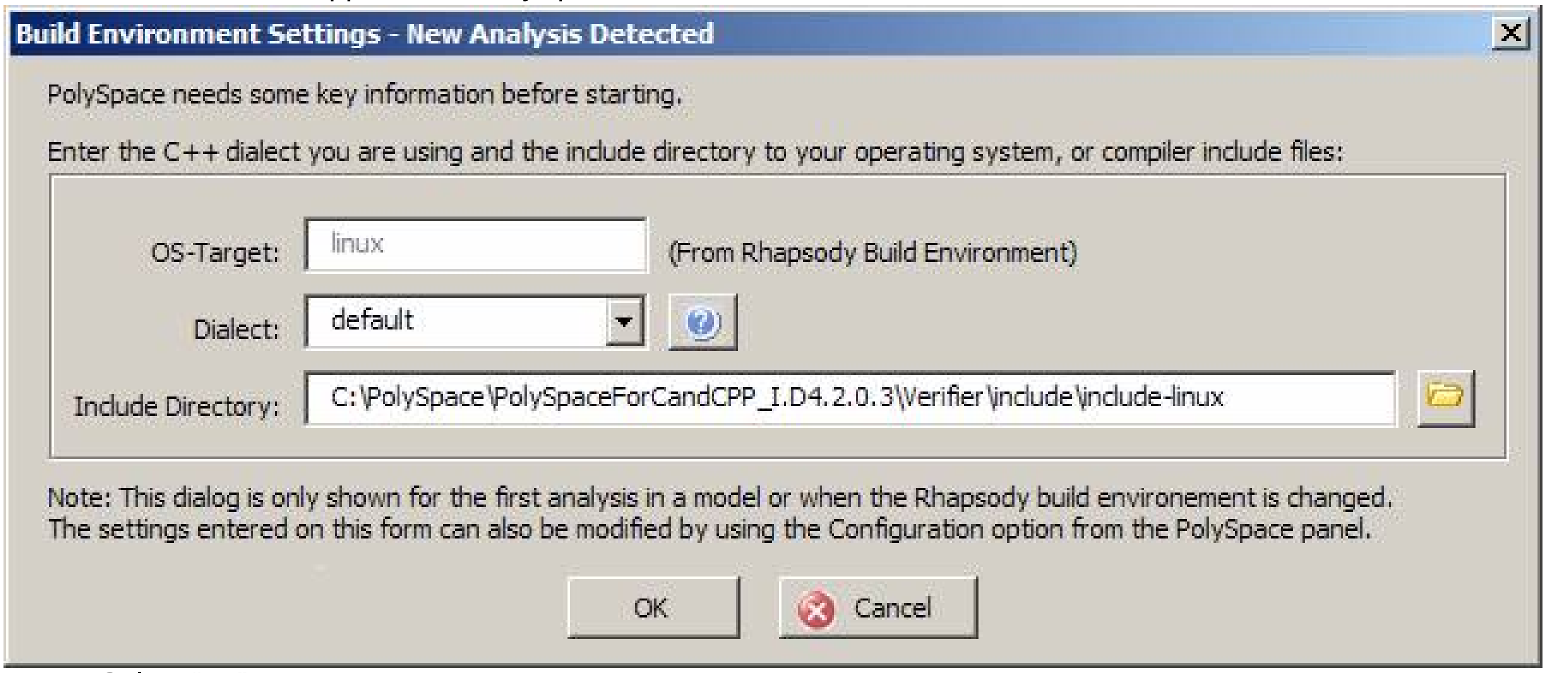


The PolySpace Panel is the interface to the PolySpace UML Link RH within Rhapsody.

6.1.2. Step 2 - Starting an analysis

- 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 will be displayed.
The operating system target (-OS-target) is set automatically from the model's 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.


Build Environment Settings - New Analysis Detected

PolySpace needs some key information before starting.

Enter the C++ dialect you are using and the include directory to your operating system, or compiler include files:

OS-Target: linux (From Rhapsody Build Environment)

Dialect: default

Include Directory: C:\PolySpace\PolySpaceForCandCPP_I.D4.2.0.3\Verifier\include\include-linux

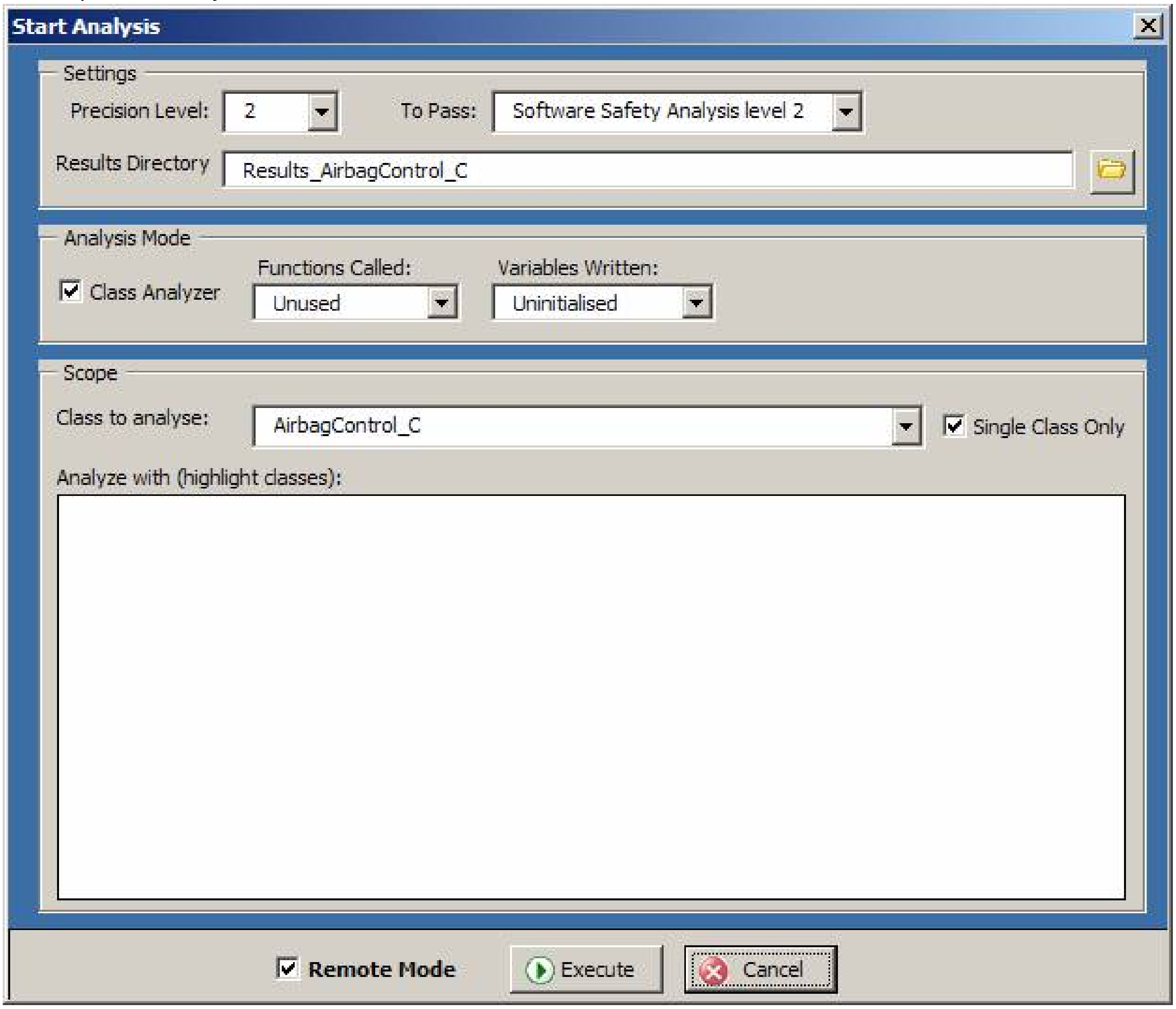
Note: This dialog is only shown for the first analysis in a model or when the Rhapsody build environement is changed.
The settings entered on this form can also be modified by using the Configuration option from the PolySpace panel.

- Select "OK".

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

6.1.3. 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 that the results directory is set automatically when the class to analyze or analysis mode is changed.
Set the options as they are shown below:


Start Analysis

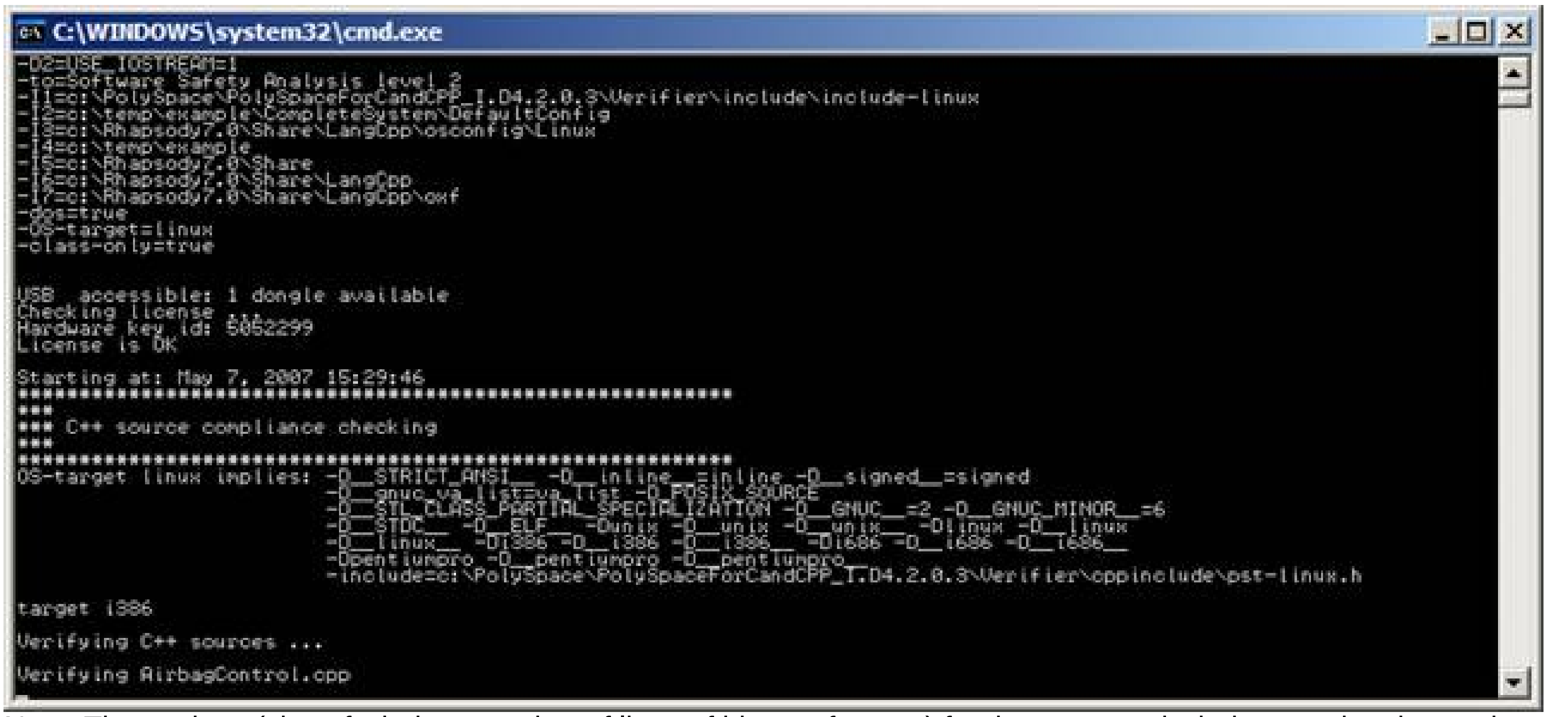
Settings
Precision Level: 2
To Pass: Software Safety Analysis level 2
Results Directory: Results_AirbagControl_C

Analysis Mode
☑ Class Analyzer
Functions Called: Unused
Variables Written: Uninitialised

Scope
Class to analyse: AirbagControl_C
☑ Single Class Only
Analyze with (highlight classes):

☑ Remote Mode

Select the Execute button - if no remote PolySpace Server is available deselect the “`Remote Mode`” option and the analysis can be performed locally.
A command window will be displayed during which the phases of the analyses performed locally can be viewed:



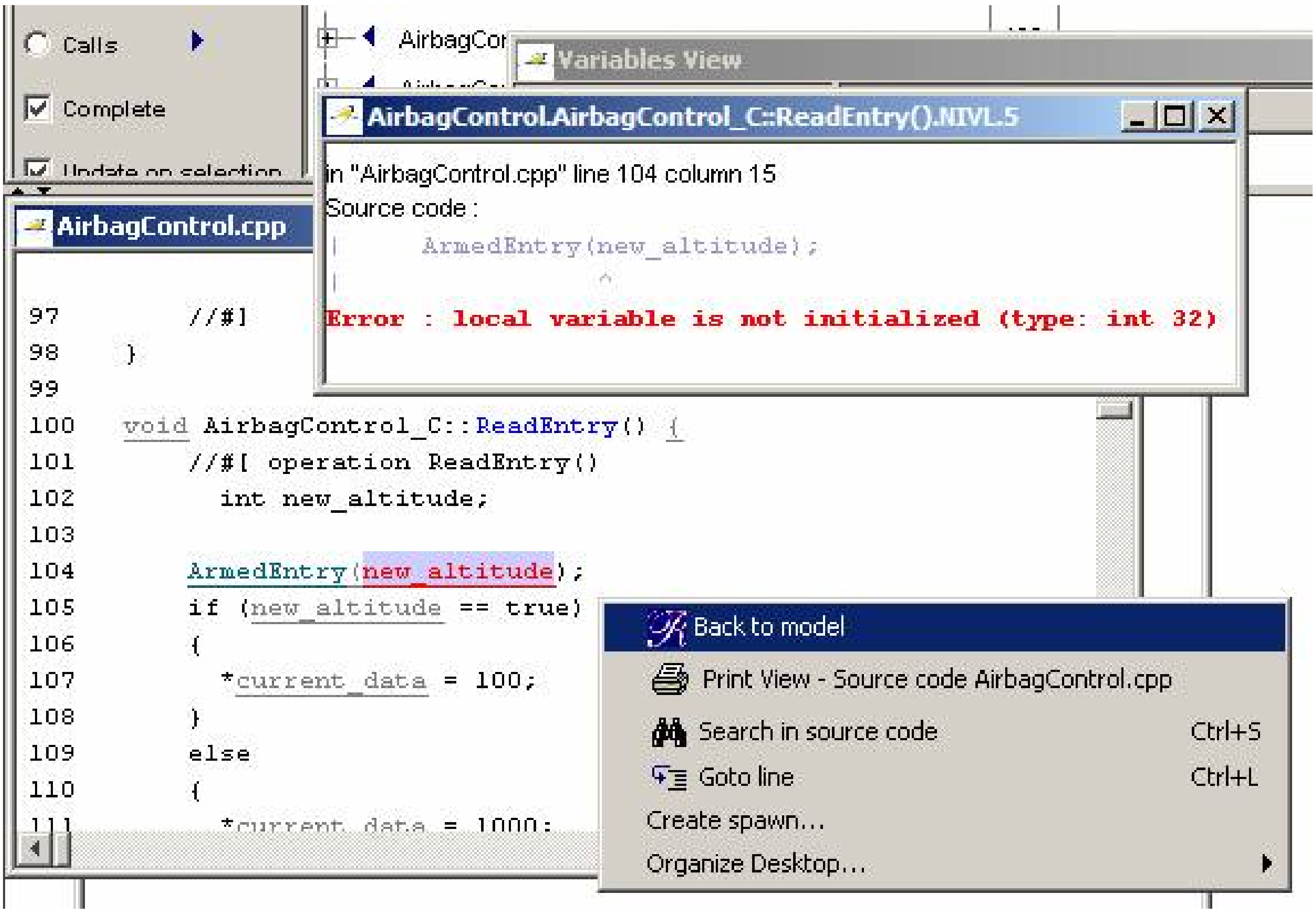
```
-D2=USE_IOSTREAM=1
-to=Software Safety Analysis level 2
-I1=c:\PolySpace\PolySpaceForCandCPP_I.D4.2.0.3\Verifier\include\include-linux
-I2=c:\temp\example\CompleteSystem\DefaultConfig
-I3=c:\Rhapsody7.0\Share\LangCpp\osconfig\Linux
-I4=c:\temp\example
-I5=c:\Rhapsody7.0\Share
-I6=c:\Rhapsody7.0\Share\LangCpp
-I7=c:\Rhapsody7.0\Share\LangCpp\oxf
-dos=true
-OS-target=linux
-class-only=true

USB  accessible: 1 dongle available
Checking license ...
Hardware key id: 5052299
License is OK

Starting at: May 7, 2007 15:29:46
****************************************************************
***
*** C++ source compliance checking
***
****************************************************************
OS-target linux implies: -D__STRICT_ANSI__ -D__inline__=inline -D__signed__=signed
                         -D__gnuc_va_list=va_list -D_POSIX_SOURCE
                         -D__STL_CLASS_PARTIAL_SPECIALIZATION -D__GNUC__=2 -D__GNUC_MINOR__=6
                         -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__
                         -include=c:\PolySpace\PolySpaceForCandCPP_I.D4.2.0.3\Verifier\cppinclude\pst-linux.h

target i386

Verifying C++ sources ...

Verifying AirbagControl.cpp
```

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.

The following of the analysis on the server, if the “`Remote Mode`” has been ticked, can by clicking on the “`Manage Analyses`” button in the PolySpace panel which will display the PolySpace Queue Manager interface (or Spooler). When the analysis has completed download the results and when prompted open with the PolySpace Viewer.

6.1.4. Step 4 - Navigating from the PolySpace results to the Rhapsody model

Navigate to the first red error, a non initialized variable detected in the model at line 104 of Airbag Control_C and right click. From the pop-up menu select “`Back to Model`”.

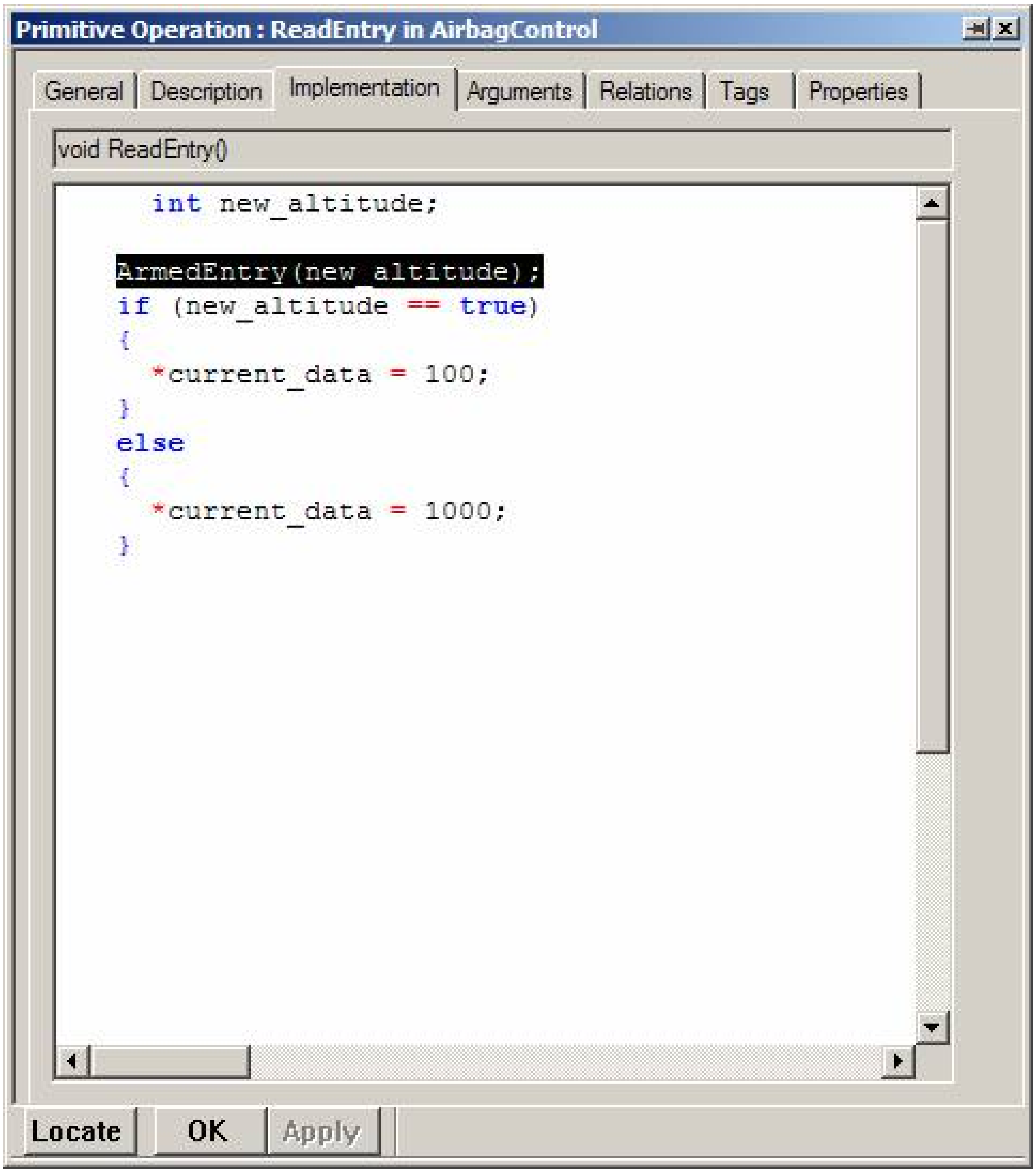


```
AirbagControl.AirbagControl_C::ReadEntry().NIVL.5
in "AirbagControl.cpp" line 104 column 15
Source code :
|      ArmedEntry(new_altitude);
|                 ^
Error : local variable is not initialized (type: int 32)
```

```
//#]
    }

    void AirbagControl_C::ReadEntry() {
        //#[ operation ReadEntry()
          int new_altitude;

        ArmedEntry(new_altitude);
        if (new_altitude == true)
        {
          *current_data = 100;
        }
        else
        {
          *current_data = 1000;
```

Important 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 configuration this will either be shown in a popup dialog (such as shown below) or in the code view:



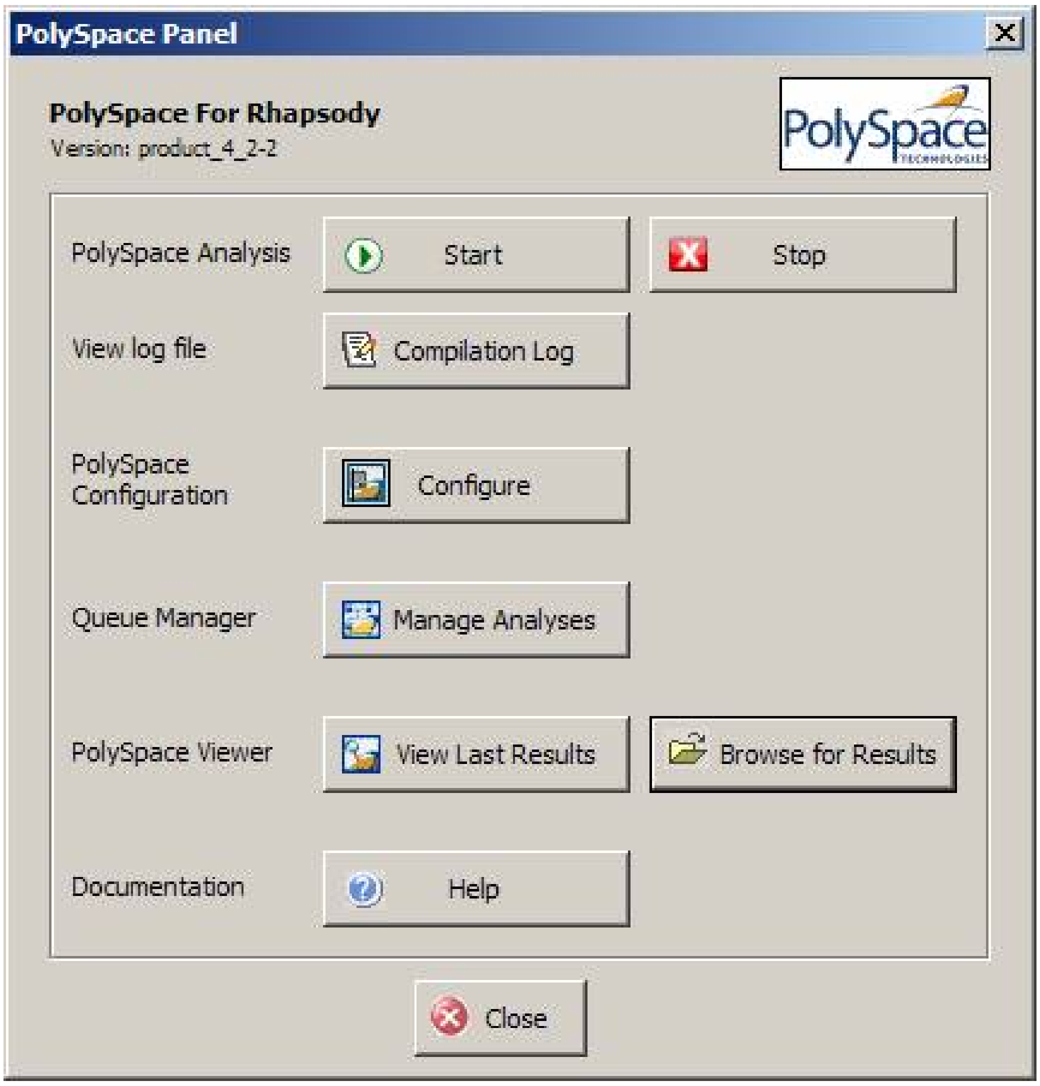
void ReadEntry()

```
   int new_altitude;

 ArmedEntry(new_altitude);
 if (new_altitude == true)
 {
   *current_data = 100;
 }
 else
 {
   *current_data = 1000;
 }
```

This is the end of the getting started guide.

6.2. PolySpace Panel

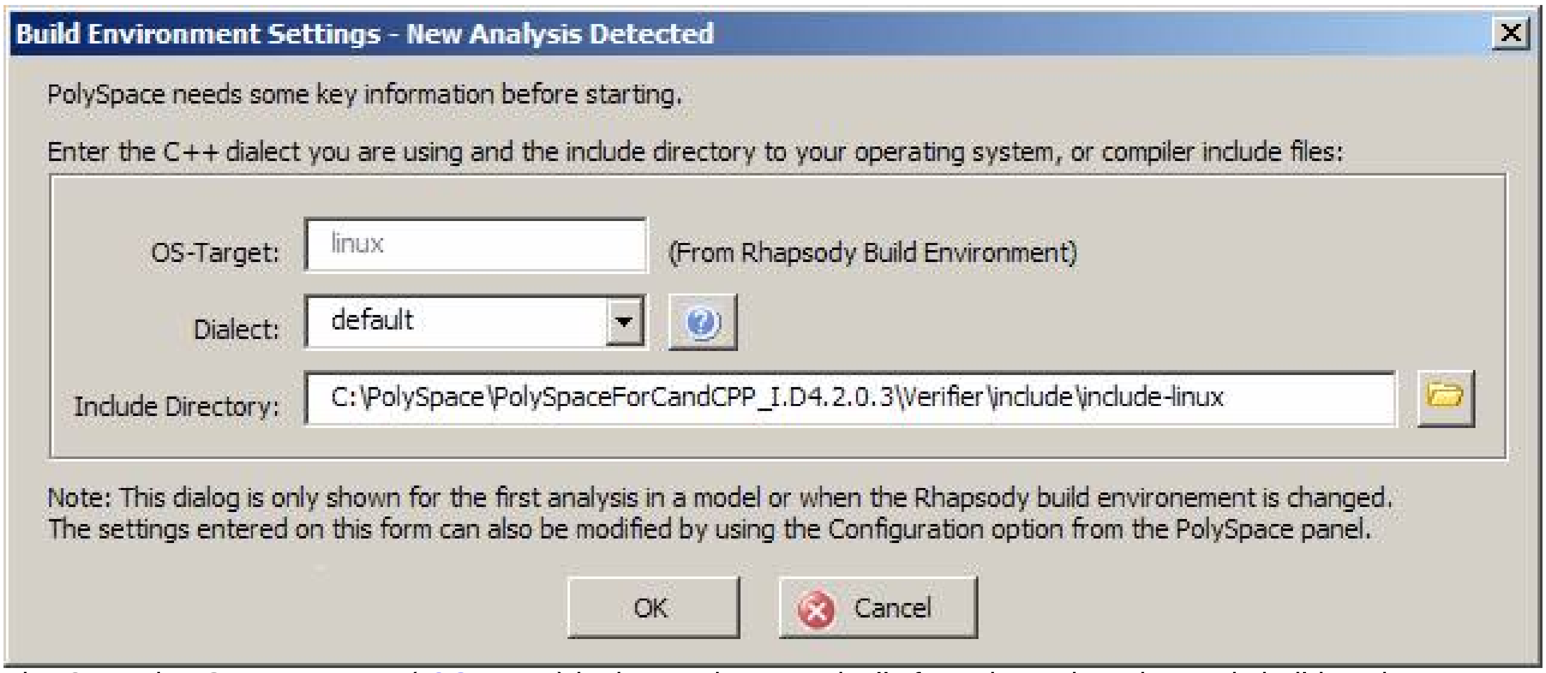
The PolySpace Panel is the main interface of the PolySpace integration with Rhapsody. The Panel can be started by right clicking on either a package or a class in the Rhapsody "Entire Model View" and selecting PolySpace Analysis.



"`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:


Build Environment Settings - New Analysis Detected

PolySpace needs some key information before starting.

Enter the C++ dialect you are using and the include directory to your operating system, or compiler include files:

OS-Target: linux (From Rhapsody Build Environment)

Dialect: default

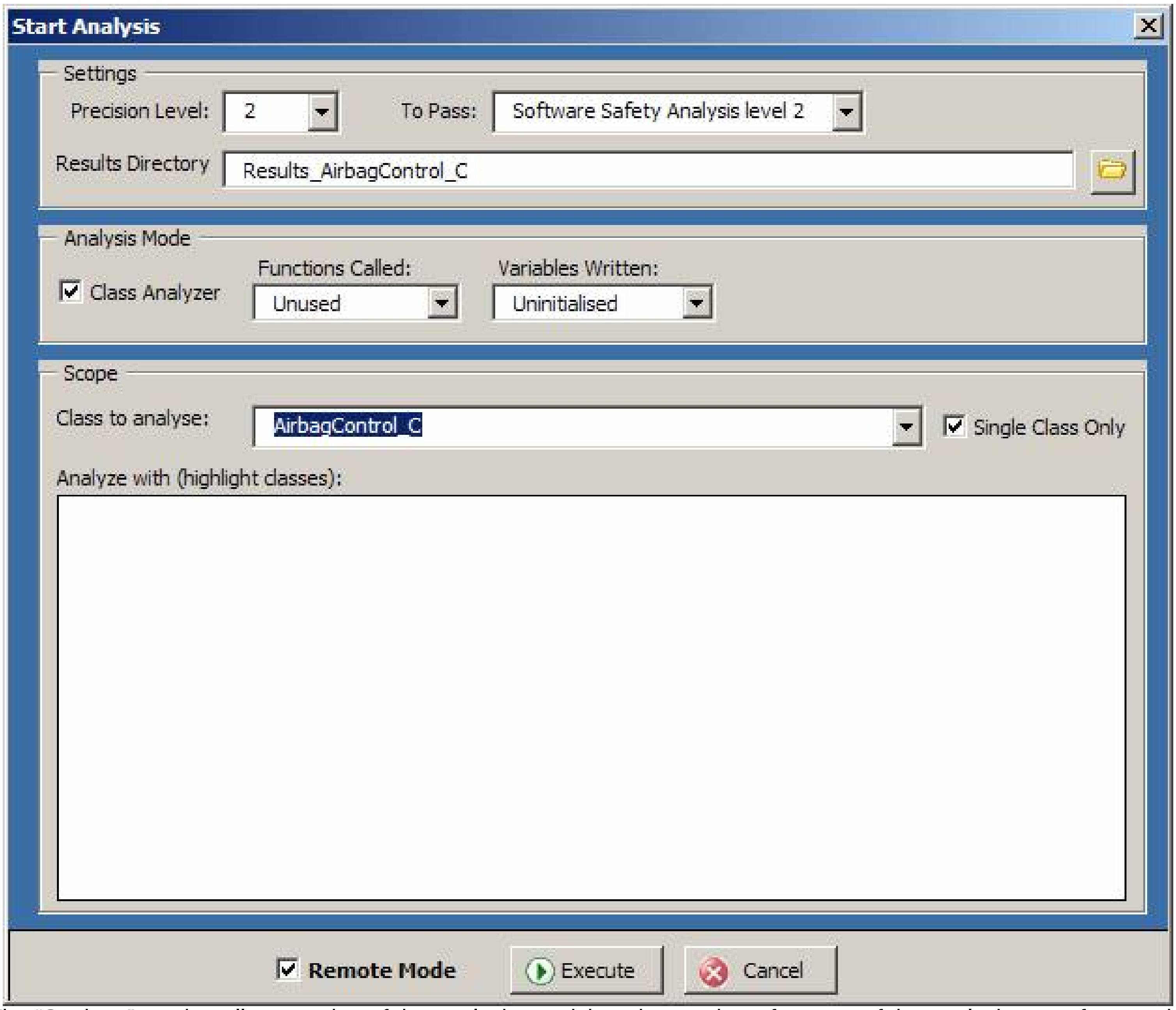
Include Directory: C:\PolySpace\PolySpaceForCandCPP_I.D4.2.0.3\Verifier\include\include-linux

Note: This dialog is only shown for the first analysis in a model or when the Rhapsody build environement is changed.
The settings entered on this form can also be modified by using the Configuration option from the PolySpace panel.

The Operating System Target (-OS-target) is detected automatically from the active Rhapsody build environment. Select the appropriate C++ dialect and the location of the include files for the compiler. If more than one include directory 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/Server for C/C++ product directory.

Start Analysis Dialog:

- Selecting OK will result in the Start Analysis dialog being displayed.
- Select the class to analyze from the scope section. The results directory is automatically set according to the name of the selected class, but can be overwritten once the class to analyze has been selected.
- Select execute to start the analysis. When “`Remote Mode`” is selected the analysis will be sent to the PolySpace Server at the end of the compilation phase.


Start Analysis

Settings

Precision Level: 2

To Pass: Software Safety Analysis level 2

Results Directory: Results_AirbagControl_C

Analysis Mode

Class Analyzer: ✓

Functions Called: Unused

Variables Written: Uninitialised

Scope

Class to analyse: AirbagControl_C

Single Class Only: ✓

Analyze with (highlight classes):

Remote Mode: ✓

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

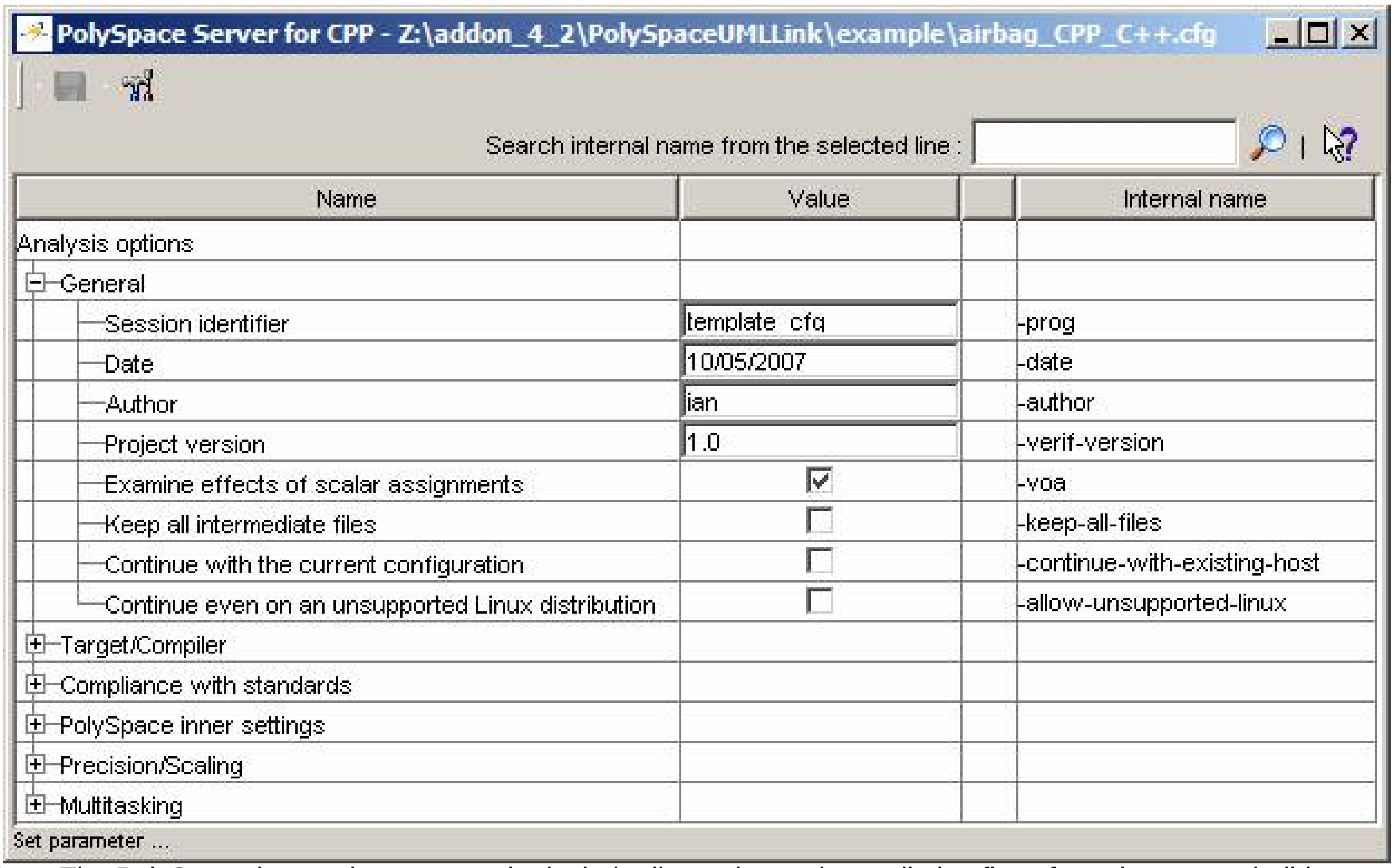
The client based phase (compilation) of the analysis can be stopped by clicking on the “**`Stop`**” button. For analysis running on the PolySpace Server use the “Queue Manager” to control the jobs.

“`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-down version of the PolySpace Launcher. From this interface advanced PolySpace options can be configured. Also when required extra source code compilation parameters can be entered.
Click the disc button in the top left corner to save the configuration.


PolySpace Server for CPP - Z:\addon_4_2\PolySpaceUMLLink\example\airbag_CPP_C++.cfg

Search internal name from the selected line :

Name	Value		Internal name
Analysis options			
⊟ General			
Session identifier	template_cfg		-prog
Date	10/05/2007		-date
Author	ian		-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			
⊞ Precision/Scaling			
⊞ Multitasking			

Set parameter ...

Note: The PolySpace integration extracts the include directories and compilation flags from the current build environment. In many cases no further configuration other than that requested 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 on the "Analysis Manager" button.

"`View Results`" Button

To view results the results from the last performed analysis click on the "View Results" button. If no results are available (they are still on the server) the user will be prompted to start the PolySpace Queue Manager. From the "Queue Manager" the results can be downloaded and opened.

"`Browse for Results`" Button

This option allows browsing and opening of all the PolySpace results for a model.

"`Help`" Button

The help button opens the PDF document containing this help.

6.3. Installing the Integration into an existing model

This section details the configuration required to use PolySpace for UML link in a Rhapsody project. The integration is written using the Visual Basic extension provided by Rhapsody. To install the integration into new Rhapsody projects it uses the `copy`VBA feature contained in the Rhapsody `ini` file (Windows directory ”`c:\rhapsody.ini`”).

However, for existing projects, the PolySpace Visual Basic module needs copying from `<PolySpaceCommonInstall>\PolySpaceUMLLink\bin\polyspace.vba` to the project directory and then renaming to `<project_name>.vba`, replacing the existing `vba` file.

For sites already using the Rhapsody visual basic feature, a merge of the PolySpace code and the existing code can be performed. To perform the merge:

a) Export each form and module for the existing code using the Rhapsody Visual Basic editor.

b) Close you model and copy the PolySpace.vba file into the model directory and rename it to “`<project_name>.vba`”.

c) Re-open the model, start the Visual Basic Editor and import all of the module/form code that was exported in the item ‘`a)`’.

d) Optional: The master `polyspace.vba` file can be updated if required with the contents of “`<project_name>.vba`” for use in new projects.

6.4. Other Topics

Analysis Information

The first time an analysis is performed, a template PolySpace configuration file is copied from `<PolySpaceInstallCommon>/PolySpaceForUML/etc/template_<Language>.cfg` to the project directory and is renamed `<model>_<language>.cfg` where `<model>` is the name of your model and `<language>` is the name of the language the model is targeted at e.g. `C++` for this release. The template `cfg` file can be updated either by editing it with the PolySpace Launcher by double clicking the file in a Windows Explorer window or by manually copying a `cfg` file from a Rhapsody model directory over the template file. In this way configuration can be shared amongst project members and used with other projects.

Default template CFG Options

The following options are set by default in the `template_C++.cfg` file:

```
-lang=CPP
-prog=template_cfg
-results-dir=r->results
-allow-undef-variables=true
-respect-types-in-globals=true
-respect-types-in-fields=true
-dos=true
-target=i386
-D=[OM_NO_FRAMEWORK_MEMORY_MANAGER]
-to=7
-OS-target=no-predefined-OS
-voa=true
```

Back to Model Viewer link

The “`BackToModel`” command in the Viewer (right click on a check) is currently limited to source code lines containing a PolySpace check and also not containing a macro.

- A warning “`Unable to go back to the UML model from this location`” will be given when trying to use the command on a line containing a macro.
- A warning “`No element found in model`” will be given for locations that Rhapsody does not support the back to model feature, or for lines of code not in the model.

7. Working with results review

Related subjects :

7.1. Basics: prerequisite being able to review PolySpace results

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

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

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

- The next instruction is reached providing no run-time error was met at the previous one.
- Each run-time error implies a “core dump” for PolySpace. The corresponding execution is considered to have stopped, even if the run-time execution of the code might not. SO – red checks will be followed by grey checks, and orange checks only propagate the green parts through to subsequent checks.
- You should focus on the message given by PolySpace, and try not to jump to false conclusions. You must explain the colour of a check step by step, until you find the root cause.
- You should focus on an explanation by examining the code, and try not to be influenced by knowledge of what the code actually does.

Related subjects :

7.1.1. Grey follows red

For this step, you'll find why **green** is propagated out of **orange**. In the example below, consider the explanation of:

- the grey after the red in the **red** function;
- and the **green** colour of the array.

Explanation

```
void red(void)
{
int x;
x = 1 / x ;

x = x + 1;
}
```

```
extern int read_an_input(void);
void propagate(void)
{
  int X;
  int y[100];
  X = Read_An_input();
  y[X] = 0; // [array index within bounds]
  y[X] = 0;
}
```

Let's detail each line of code for:

- The red function:
  - o When PolySpace divides by X, X has not been initialized. Therefore the corresponding check (Non Initialized Variable) on X is red;
  - o As a result all possible execution paths are stopped, because they all produce an RTE.
- The propagate function:
  - o X is assigned the value of Read_An_Input. After this assignment, X ~ [-2^31, 2^31-1].
  - o At the first array access, an “out of bounds” error is possible since X can be equal to (say) -3 as well as 3;
  - o All conditions leading to an RTE are assumed to have been truncated – they are no longer considered in the analysis. So on the following line, the executions for which X ~ [-2^31, -1] and [100, 2^31-1] are stopped;
  - o Consequently at the next instructions X ~ [0, 99];
  - o Hence at the second array access, the check is green because X ~ [0, 99].

Summary

Green is propagated out of **orange**. When doing manual stubbing and by using assert, you can use value propagation to restrict input values for data. See Manual stubbing section

7.1.2. What is the message and what does it mean?

PolySpace numbers the results in the same order than an execution would have performed the associated operations.

Consider the instruction:

```
X++;
```

In each case, PolySpace first checks for a potential NIV (Non Initialized Variable) for x, then checks the potential OVFL (overflow). An awareness of such sequences will help to understand the message which PolySpace is presenting before going on to assess what that means for the code.

In the example below, the orange NIV on X in the test:

```
if (x > 100)
```

doesn't mean PolySpace doesn't know the value of X, which might be the conclusion of a hasty analysis.

So - what does it mean?

```
extern int read_an_input(void);

void main(void)
{
int x;
if (read_an_input()) x = 100;
if (X > 101) // [orange on the NIV : non initialized variable ]
  { X++; } // grey code
}
```

Explanation

When you click on the check under the Viewer, you see the category of the check. Here, the category is NIV (Non Initialized Variable). However, PolySpace may well analyze subsequent lines of code, and continue with an understanding of the possible values as if initialization has taken place.

The correct analysis of this result might be that if X has been initialized, the only possible value for X is { 100 }, which is not greater than 101, so the rest of the code is grey. Hence we can conclude that PolySpace did know the values - which is different from our first, hasty analysis.

Summary

- FALSE: `if "( x > 100)"` means: PolySpace doesn't know anything;
- TRUE: `if "( x > 100)"` means: PolySpace doesn't know if X has been initialized.

The first rules of reviewing results are:

- focus on the message given by PolySpace Verifier,
- do not focus on a quick interpretation.

7.1.3. What is the C++ explanation?

Results can only be explained based on the code and not on:

- a physical action;
- a particular configuration;
- or any reason other than the code itself.

Remember, all the tool is aware of is the C++ code itself!

Let's consider the example below, with regards to the explanation for the dead code (grey code) following the "if" statement.

```
extern int read_an_input(void);

void main(void)
{
  int x;
  int y[100];
  x = read_an_input();
  y[x ] = 0; // [array index within bounds]
  y[x-1] = (1 / X) + X ;
  if (x == 0)
     y[x] = 1; // grey code on this line
}
```

Once step 1 has been performed, the analysis is:

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

So, why is the test always false?

- The orange will truncate all executions paths that lead to a run-time error, so that in our example, all instances where X is equal to 0 are stopped. From the division on, X ~ [1; 99];
- That is why X is never equal to 0 **at this line** – and hence, the array access is green (Y(x- 1).

Note: for the array access at the previous line (y[x]), we have X ~ [-2^31, 2^31-1] - hence the orange on (1 /X) .

Summary

In this example, all results are located in the same procedure. But it is the same work to perform using the call tree if a result comes from a procedure called by many others. Follow the "called by" call tree. **Focus on the explanation within the code only!**

7.1.4. Review run-time errors: Fix red errors

All run-time Errors (RTE) highlighted by PolySpace are determined by reference to the language standard, and are sometimes implementation dependant – that is, they may be acceptable for a particular compiler but unacceptable according to the language standard.

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

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

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

PolySpace identifies two kinds of red checks

- Red errors which are **compiler-dependant** in a specific way. On some occasions a PolySpace option may be used to allow particular compiler specific behaviour, and on others the code must be corrected in order to comply. An example of a PolySpace option to permit compiler specific behaviour would be the option to force dealing with constant overflows, shift operation on negative values, etc.
- All **other red errors must be fixed**. They are bugs.

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

7.1.5. Review dead code checks: why is grey code interesting?

• Functional bugs can be found in grey code

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

- Defensive code which is never reached
- Dead code due to a particular configuration
- Libraries which are not used to their full extent in a particular context
- Dead code resulting from bugs in the source code.

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

- A lack of parenthesis and operand priorities in the testing clause can change the meaning significantly.
  Consider this line of "pseudo" code:
  ```
     IF NOT a AND b OR c AND d
  ```
  Now consider how misplaced parentheses might influence how that line behaves:
  ```
     IF NOT (a AND b OR c AND d)
  IF (NOT (a) AND b) OR (c AND d))
        IF NOT (a AND (b OR c) AND d), …
  ```
- The test of variable inside a branch where the conditions are never met;
- An unreachable "else" clause where the wrong variable is tested in the "if" statement
- A variable that is supposed to be local to the file but instead is local to the function
- Wrong variable prototyping leading to a comparison which is always false (say)

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

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

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

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

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

• Structural coverage

PolySpace always performs upper approximations of all possible executions. Therefore even if a line of code is shown in green, there remains a possibility that it is a dead portion of code. Because

PolySpace made an upper approximation, it could not conclude that the code was dead, but it could conclude that no run-time error could be found. PolySpace will find around 80% of dead code that the developer would find by doing structural coverage.

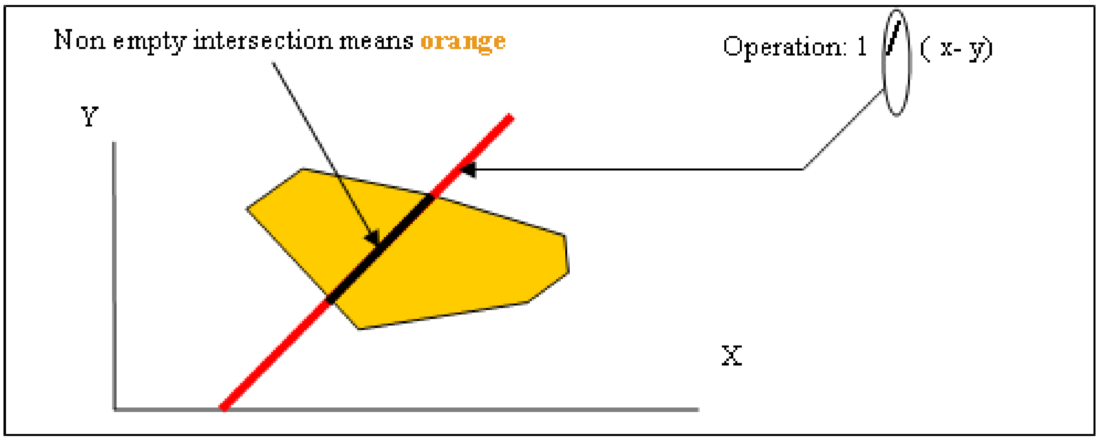
PolySpace is intended to be used as a productivity help in dead code detection. It detects dead code which might take days of effort to find by any other means.

7.1.6. How to conclude an orange review

Related subjects :

7.1.6.1. What is an orange?

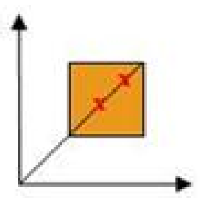
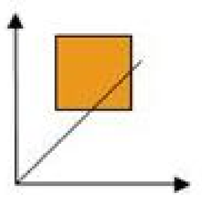
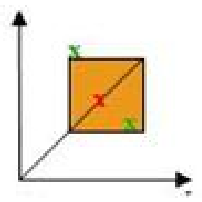
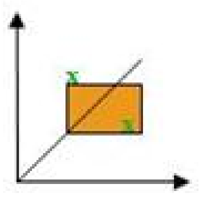
If a check is orange, it means that the approximate data set assumed by the analysis to represent a variable intersects with the error zone.



Graphical representation of an orange check

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

Note that any an orange check can approximate a check of any other colour.

	Red approximated by orange		Grey approximated by orange
	Any other situation: real orange		Green approximated by orange

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

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

7.1.6.2. What are the different sources of oranges?

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

1. Potential bug: an **orange check** can represent a real bug.

   - Example – loop with division by zero

2. Inconclusive check: an **orange check** can represent a situation where PolySpace is unable to conclude whether a problem exists. It is sometimes in the nature of software code that it cannot be concluded whether there is a potential error. In the example below, the task T1 can be started before or after T2, so PolySpace can't conclude without the calling sequence being defined.

   - Consider a variable X initialized to 0, and two concurrent tasks T1 and T2.
   - Suppose that T1 assigns a value of 12 to variable X
   - Now suppose that T2 divides a local variable by X. The division is shown as an **orange check** because T1 can be started before or after T2 (so a division by zero is possible).

3. Data set issue: an **orange check** resulting from a theoretical set of data. PolySpace considers all combinations of input data rather than *one* particular combination (that is, it uses an upper approximation of the data set). Therefore a check may be coloured **orange** as the result of a combination of input values which is analysed by PolySpace, but which will not be possible at execution time.

   - Consider three variables X, Y and Z which can vary between 1 and 1000
   - Now suppose that the code computes a value of X*Y*Z on a type 16 bits. The result can potentially overflow. It may be known when the code is developed that the variables can't all take the value 1000 at the same time, but this information is not available to PolySpace. The code will be coloured **orange**, accordingly.

4. Basic imprecision: an **orange check** can be due to an imprecise approximation.

   - Consider X, a signed integer between -2^31 and 2^31-1.
   - Suppose a function is called which performs the assignment x=1/x
   - The parameters passed to the function imply that x must be equal to -5, -3, 8 or [10..20]. It is clear from inspection that there is no problem here, but in this case PolySpace has made an imprecise approximation.

7.1.6.3. How to determine the cause of one orange?

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

1. Potential bug: An **orange check** can reveal code which will fail under some circumstances. The following section describes how to find them.

2. Inconclusive analysis: Most inconclusive **orange checks** will take some time to investigate. An inconclusive **orange check** may well result from a very complex situation such that it may take an hour or more to understand the cause. You may decide to recode in order to be certain that there is no risk, bearing in mind the criticality of the function and the required speed of execution.

3. Data set issue:. It is normally possible to conclude that an **orange check** is the result of data set problem in a couple of minutes. You may wish to comment the code to flag this warning, or alternatively modify the code in order to take constraints into account.

4. Basic imprecision: PolySpace cannot help to debug this code. You may or may not have a problem here, but you will need a supplementary activity to be sure. Most of the time, a quick code review is a suitable path to take, perhaps using the Viewer’s navigation facilities.

7.2. Methodology: selective orange review

The purpose of this activity is to assess the probability of missing an orange containing a bug when performing a “selective orange review”. This needs to be balanced with the cost of a bug left in the code. Nevertheless**,** before reading this section, it is necessary to understand how the user might conclude the status of an **orange** check. This is explained in a later section.

Suppose, for example, that the user wishes to spend the first hour of the day reviewing an analysis which was performed overnight. This is an approach which can be adopted to enhance the quality of code under development, perhaps supported by more extensive analysis as the project nears completion.

Experience suggests that such approach can highlight 5 bugs in orange checks in a timescale: “finding 5 bugs in an hour”.

Related subjects :

7.2.1. The basic principles

Focus on modules which have the highest selectivity in the application, where the **selectivity** is the ratio of (**green** + grey + **red**) / (total number of checks).

The selectivity ratio can be found in the "`procedural entities`" view on PolySpace Viewer in the percentage column. For example, review will be applied only on modules which have a selectivity ratio greater than 85%, indeed by beginning the module which has the highest ratio. Then:

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

7.2.2. The rationale behind the approach

If PolySpace finds only one or two **orange checks** in a module or function, there is a very good possibility that they are not caused by "basic imprecision". Consequently, the concentration of bugs in **orange checks** here will be higher than in those found elsewhere in the code.

If you come across an **orange check** which takes more than a few minutes to understand, it might well be the result of inconclusive PolySpace analysis. To optimise the number of bugs found in a limited time, you should move on to another check. A good rule of thumb is to spend no more than 5 minutes on each check, remembering that the goal is to review at least 50 checks per hour to maximise the number of bugs found.

Considering such an orange, you have to classify it in one of the four possible reasons explained in a section above:

- Potential bug

If the data set analysed reveals real bugs, they should be corrected.

- Inconclusive check

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

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

- Basic imprecision.

On some rare occasions, a module will contain a lot of similar occurrences of a "basic imprecision". This is most likely to be caused by a function close to the edge of an application, or in some stub routines. In this case, PolySpace can only assist by means of the call tree and dictionary. This code needs to be reviewed by an alternative activity – perhaps through additional unit tests or code review with the developer. These checks are usually local to functions, so their impact on the project as a whole is limited.

7.2.3. In practice

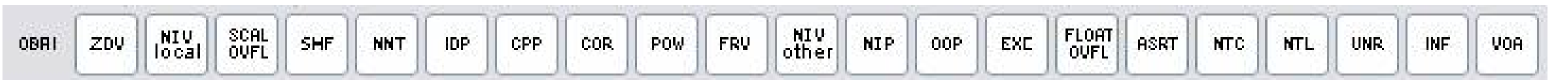
For any particular function, PolySpace may be better at detecting some kinds of Run Time Errors than others. For instance, the analysis of one function may yield imprecise results from the analysis of Non Initialised Variables (NIV) but very precise results from the analysis of overflows (OVFL). In the analysis of another function, the precise opposite may be true.

So, the **“high selectivity focus**” should be applied to each Run Time Error category **separately**.

Previous Back to table of contents Next

7.2.4. Step by step

1. Select one type of Run Time Error category, beginning from the left side (see figure below: “`Tool bar for checks`”) such as Out Of Bound Array Index (OBAI) as shown in the example. Click on Filter all and then click on the check type of interest (**OBAI** in the example)



Tool bar for checks

In figure above, all categories of checks are filtered except OBAI category.

2. Choose function member containing a high selectivity ration, i.e. few **orange checks** of the selected kind surrounded by a lot of **green** checks.

3. Proceed with a quick code review on each **orange check**, spending no more than 5 minutes on each. The goal is to identify the **orange check** as a *potential bug, inconclusive check* or *data set issue*, navigating the code using the call tree and the dictionary. If the check proves too complicated to explain, it may well be the result of a *basic imprecision.*

4. Once this job done, the user can select the “`Verified`” checkbox in the PolySpace Viewer, and put an explanation of the check in the comment field (for instance, “`inconclusive`”, or “`data set issue`”, “`when calibration of <x> is set greater than 100`”, …)

Select another type of category and repeat step 1-4. **Note**: It is important to review checks from left to right and finish with “C++” checks (CPP, EXC and OOP).

7.2.5. Considering the effects of application code size

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

For instance, in a relatively small application, PolySpace Verifier might retain very detailed information about the data at a particular point in the code, so that for example the variable VAR can take the values { -2; 1; 2; 10; 15; 16; 17; 25 }. If VAR is used to divide, the division is green (because 0 is not a possible value).

If the program being analyzed is large, PolySpace Verifier would simplify the internal data representation by using a less precise approximation, such as [-2; 2] U {10} U [15 ; 17] U {25} . Here, the same division appears as an orange check.

If the complexity of the internal data becomes even greater later in the analysis, PolySpace Verifier might further simplify the VAR range to (say) [-2; 20].

This phenomenon leads to the increase or the number of orange warnings when the size of the program becomes large.

Note that the amount of simplification applied to the data representations also depends on the required precision level (O0, O2), PolySpace Verifier will adjust the level of simplification, viz.:

- -O0 and –quick: shorter computation time. Focus only red and grey.
- -O2: less orange warnings.
- -O3: less orange warnings and bigger computation time.

7.3. Category of checks

This section presents all categories of checks that PolySpace analyses. Theses checks are classified into acronyms. Each acronym represents one or more verification made by PolySpace. The list of acronyms, checks and associated coloured messages are listed in the following tables.

- Acronyms associated to C++ specific constructions:

Category	Acronym	Green	Grey
function returns a value	FRV	function returns a value	Unreachable check: function returns a value
non null this-pointer	NNT	this-pointer [of f] is not null	Unreachable check: this-pointer [of f] is not null
C++ related instructions	CPP	array size is strictly positive	Unreachable check: array size is strictly positive
	CPP	typeid argument is correct	Unreachable check: typeid argument is correct
	CPP	dynamic_cast on pointer is correct	Unreachable check: dynamic_cast on pointer is correct
	CPP	dynamic_cast on reference is correct	Unreachable check: dynamic_cast on reference is correct
	INF	Informative check: f is implicitly called	Informative check: implicit call of f is unreachable
Display of errors that relate to Object Oriented Programming and inheritance	OOP	call of virtual function [f] is not pure	Unreachable check: call of pure virtual function [f]
	OOP	this-pointer type [of f] is correct	Unreachable check: this-pointer type [of f] is correct
	INF	Informative check: f is called if this-pointer is of type T	Informative check: call of f depending on this type is unreachable
	OOP	pointer to member function points to a valid member function	Unreachable check: pointer to member function points to a valid member function
	OOP		Unreachable check: call to no function Information
	INF	Informative check: f is potentially called through pointer to member function	Informative check: potential call to f through pointer to member function is unreachable
	INF	Informative check: f is called during construction of T	Informative check: call of f during construction of T is unreachable

	INF	Informative check: f is called during destruction of T	Informative check: call of f during destruction of T is unreachable
Display of errors that relate to exception handling	EXC	exception raised as specified in the throw list	Unreachable check: exception raised as specified in the throw list
	EXC	catch parameter construction does not throw	Unreachable check: catch parameter construction does not throw
	EXC	dynamic initialization does not throw	Unreachable check: dynamic initialization does not throw
	EXC	destructor or delete does not throw	Unreachable check: destructor or delete does not throw
	EXC	main, task or C library function does not throw	Unreachable check: main, task or C library function does not throw
	EXC	logic_error is not thrown	Unreachable check: logic_error is not thrown
	EXC	runtime_error is not thrown	Unreachable check: runtime_error is not thrown
	EXC	call [to f] does not throw	Unreachable check: call [to f] does not throw
	EXC	function does not throw	Unreachable check: function does not throw
	EXC	expression value is not EXCEPTION_CONTINUE_EXECUTION	Unreachable check: expression value is not EXCEPTION_CONTINUE_EXECUTION
	EXC		Unreachable check: throw is not allowed with option -no-exception

Category	Acronym	Red	Orange
function returns a value	FRV	Error: function does not return a value	Warning: function may not return a value
non null this-pointer	NNT	Error: this-pointer [of f] is null	Warning: this-pointer [of f] may be null
C++ related instructions	CPP	Error: array size is not strictly positive	Warning: array size may not be strictly positive
	CPP	Error: incorrect typeid argument	Warning: typeid argument may be incorrect
	CPP	Error: incorrect dynamic_cast on pointer (analysis continues using a null pointer)	Warning: dynamic_cast on pointer may be incorrect
	CPP	Error: incorrect dynamic cast on reference	Warning: dynamic_cast on reference may be incorrect
	INF		
Display of errors that relate to Object Oriented Programming and inheritance	OOP	Error: call of pure virtual function [f]	Warning: call of virtual function [f] may be pure
	OOP	Error: incorrect this-pointer type [of f]	Warning: this-pointer type of [f] may be incorrect

	INF		
	OOP	Error: pointer to member function is null or points to an invalid member function	Warning: pointer to member function may be null or point to an invalid member function
	OOP	Internal PolySpace error: please contact support	
	INF		
	INF		
	INF		
Display of errors that relate to exception handling	EXC	Error: exception raised is not specified in the throw list	Warning: exception raised may not be specified in the throw list
	EXC	Error: throw during catch parameter construction	Warning: possible throw during catch parameter construction
	EXC	Error: throw during dynamic initialization	Warning: possible throw during dynamic initialization
	EXC	Error: throw during destructor or delete	Warning: possible throw during destructor or delete
	EXC	Error: main, task or C library function throws	Warning: main, task or C library function may throw
	EXC	Error: logic_error is thrown (analysis jumps to enclosing handler)	Warning: logic_error may be thrown
	EXC	Error: runtime_error is thrown (analysis jumps to enclosing handler)	Warning: runtime_error may be thrown
	EXC	Error: call [to f] throws (analysis jumps to enclosing handler)	Warning: call [to f] may throw
	EXC	Error: function throws (analysis jumps to enclosing handler)	Warning: function may throw
	EXC	Error: expression value is EXCEPTION_CONTINUE_EXECUTION (limitation)	Warning: expression value may be EXCEPTION_CONTINUE_EXECUTION (limitation)
	EXC	Error: throw is not allowed with option -no-exception	

- Acronym not related to C++ constructions (also used for C code):

Category	Acronym	Green	Grey
Out of bound array index	OBAI	Array index is within its bounds	Unreachable check: out of bounds array index error
Zero division	ZDV		Unreachable check:

Non-initialized variable	NIV local/other	[local] variable is initialized	Unreachable check:
scalar or float overflows	OVFL/UNFL		Unreachable check: variable overflow error
Illegal dereference pointer	IDP		Unreachable check: illegal dereference pointer error
Correctness condition	COR	Function pointer must point to a valid function	Unreachable check: Function pointer must point to a valid function
	COR		
	COR		
	COR		
Power must be positive	POW	Power is positive	Unreachable check: power positive error
Shift amount out of bounds	SHF	Scalar shift amount is within its bounds	Unreachable check: shift error
	SHF		
Non initialized pointer	NIP		Unreachable check:
user assertion failures	ASRT	User assertion is verified	Unreachable check: user assertion error
non termination of call	NTC		
non termination of loop	NTL		
Unreachable check	UNR		Unreachable code
Value on assignment	VOA	{Range inf. <= [expr] <= range sup.}	

Category	**Acronym**	**Red**	**Orange**
Out of bound array index	OBAI	Out of bound array	Array index may be outside its bounds
Zero division	ZDV	[scalar | float] division by zero occurs	[scalar | float] division by zero may occur
Non-initialized variable	NIV local/other	[local] variable is not initialized	[local] variable may not initialized
scalar or float overflows	OVFL/UNFL		
Illegal dereference pointer	IDP	Pointer is outside its bounds	Pointer may be outside its bounds
Correctness condition	COR	Function pointer must point to a valid function	Function pointer may point to a valid function
	COR		wrong type for argument of call to function
	COR		wrong number of arguments for call to function
	COR	Array conversion must not extend range	
Power must be positive	POW	Power is not positive	Power may be not positive
Shift amount out of bounds	SHF	Scalar shift amount is outside its bounds	
	SHF	Left operand of left shit is negative	
Non initialized pointer	NIP		
user assertion failures	ASRT	User assertion fails	User assertion may fail
non termination of call	NTC	[f] call never terminates	
non termination of loop	NTL	non termination of loop	

Unreachable check	UNR		
Value on assignment	VOA		

Related subjects :

- 7.3.1. Function returns a value: FRV
- 7.3.2. Non null this-pointer: NNT
- 7.3.3. Positive array size: CPP
- 7.3.4. incorrect typeid argument: CPP
- 7.3.5. incorrect dynamic_cast on pointer: CPP
- 7.3.6. incorrect dynamic_cast on reference: CPP
- 7.3.7. invalid pointer to member: OOP
- 7.3.8. Call of pure virtual function: OOP
- 7.3.9. incorrect type for this-pointer: OOP
- 7.3.10. potential call to: INF
- 7.3.11. Non-Initialized Variable: NIV/NIVL
- 7.3.12. Non-Initialized Pointer: NIP
- 7.3.13. User Assertion failure: ASRT
- 7.3.14. Overflows and underflows
- 7.3.15. Scalar or Float Division by zero: ZDV
- 7.3.16. Shift amount is outside its bounds: SHF
- 7.3.17. Left operand of left shift is negative: SHF
- 7.3.18. Power must be positive: POW
- 7.3.19. Array index is outside its bounds: OBAI
- 7.3.20. Function pointer must point to a valid function: COR
- 7.3.21. Wrong number of arguments: COR
- 7.3.22. Wrong type of argument: COR
- 7.3.23. Pointer is outside its bounds: IDP
- 7.3.24. logic_error is thrown: EXC
- 7.3.25. runtime_error is thrown: EXC
- 7.3.26. Function throws: EXC
- 7.3.27. Call to throws: EXC
- 7.3.28. destructor or delete throws: EXC

7.3.1. Function returns a value: FRV

Check to establish whether on every value-returning function there is no flowing off the end the function.
C++ Example:

```
static volatile int rand;

class function {
public:
  function() { rep = 0; }
  int reply(int msg) {     // FRV Verified: [function returns a value]
    if (msg > 0) return rep;
  };

  int reply2(int msg) {    // FRV ERROR: [function does not return a value]
    if (msg > 0) return rep;
  };

  int reply3(int msg) {    // FRV Warning: [function may not return a value]
    if (msg > 0) return rep;
  };

protected:
  int rep ;
};

void main(void){

  int ans;
  function f;

  if (rand)
    ans = f.reply(1);

  else if (rand)
    ans = f.reply2(0);  // NTC ERROR: propagation of FRV ERROR
  else
    f.reply3(rand);
}
```

Explanation:

Variables are often initialized using the return value of functions. However it may occur that, as in the above example, the return value is not initialized for all input parameter values (which should always be the case). In this case, the target variable will not be properly initialized with a valid return.

Previous Back to table of contents Next

7.3.2. Non null this-pointer: NNT

This check verifies that the *this* pointer is null during call of a member function.

C++ Example:

```
1      #include <new>
2      static volatile int random_int = 0;
3
4      class Company
5      {
6      public:
7        Company(int numbClients):numberClients(numbClients){};
8        void newClients (int numb) {
9          numberClients = numberClients + numb;
10       }
11     protected:
12       int numberClients;
13     };
14
15     void main (void)
16     {
17       Company *Tech = 0;
18
19       if (random_int)
20         Tech->newClients(2); // NNT ERROR: [this-pointer of newClients is null]
21
22       Company *newTech = new Company(2);
23       newTech->newClients(1); // NNT Verified: [this-pointer of newClients is not
null]
24
25     }
26
```

Explanation:

Polyspace verifies that all functions, virtual or not virtual, by a direct calling, and through pointer calling are never called with a null this-pointer.

In the above example, a pointer to a *Company* object is declared and initialized to null. When the *newClients* member function of the *Company* class is called (line 20), PolySpace detects that the class object is a null pointer.

On the new allocation at line 22, as standard *new* operator returns an initialized pointer or raises an exception, the *this-pointer* is considered as correctly allocated at line 23.

7.3.3. Positive array size: CPP

This check verifies that the array size is always a non-negative value. In the following example, the array is defined with a negative value by a function call.

C++ Example:

```
1     static volatile int random_int = 1;
2     static volatile unsigned short int random_user;
3
4     class Licence {
5     public:
6       Licence(int nUser);
7       void initArray();
8     protected:
9       int numberUser;
10      int (*array)[2];
11    };
12
13    Licence::Licence(int nUser) : numberUser(nUser) {
14      array = new int [numberUser][2]; // PAS ERROR: [array size is not strictly
positive]
15      initArray();
16    }
17
18    void Licence::initArray() {
19      for (int i = 0; i < numberUser; i++) {
20        array[i][2]=0;
21      }
22    };
23
24    void main (void)
25    {
26      if (random_int && random_user != 0)
27        Licence FirmUnknown (-random_user); // NTC ERROR: propagation of PAS ERROR
28    }
```

Explanation:

The property, the non-negative value of an array size, is checked at line 14, where the *array* is defined with the *[numberUser][1]* dimension. Unfortunately the *numberUser* variable is always negative as an opposite of an *unsigned short int* type. PolySpace detects a red error and displays a message.

Previous Back to table of contents Next

7.3.4. incorrect typeid argument: CPP

Check to establish whether a *typeid* argument is not a null pointer dereference. This check only occurs using typeid function declared in stl library .

C++ Example:

```
#include <typeinfo>

static volatile int random_int=1;

class Form
{
public:
  Form (){};
  virtual void trace(){};
};

class Circle : public Form
{
public:
  Circle() : Form () {};
  void trace(){};
};


int main ()
{

  Form* pForm = 0 ;
  Circle *pCircle = new Circle();

  if (random_int)
    return (typeid(Form) == typeid(*pForm));    // CPP ERROR: [incorrect typeid argument]
  if (random_int)
    return (typeid(Form) == typeid(*pCircle));  // CPP Verified: [typeid argument is correct]
}




```

Explanation:

In this example, the *pForm* variable is a pointer to a *Form* object and initialized to a null pointer. Using the *typeid* standard function, an exception is raised. In fact here, the typeid parameter of an expression obtained by applying the unary "*" operator is a null pointer leading to this red error.
At line 29, **pCircle* is not null and typeid can be applied.

7.3.5. incorrect dynamic_cast on pointer: CPP

Check to establish when only valid pointer casts are performed through *dynamic_cast* operator.

C++ Example:

```
1     #include <new>
2     static volatile int random = 1;
3
4     class Object {
5     protected:
6       static Object* obj;
7     public:
8       virtual ~Object() {}
9     };
10
11    class Item : Object {
12    private:
13      static Item* item;
14    public:
15      Item();
16    };
17
18    Object* Object::obj = new Object;
19
20    Item::Item() {
21      if (obj != 0) {
22        item = dynamic_cast<Item*>(obj); // CPP ERROR: [incorrect dynamic_cast on
pointer (analysis continue using a null pointer)]
23        if (item == 0) { // here analysed and reachable code
24          item = this;
25        }
26      }
27    }
28
29    void main()
30    {
31      Item *first= new Item();
32    }
```

Explanation:

Only the dynamic casting between a subclass and its upclass is authorized. So, the casting of *Object* object to a *Item* object is an error on *dynamic_cast* at line 21, because *Object* is not a subclass of *Item*.
Behaviour follows ANSI C++ standard, in sense that even if dynamic_cast is forbidden, analysis continue using null pointer. So at line 22, *item* is considered as null and assigned to *this* at line 23.
Note that this is only check where we can have another colour after a red. It is not the case for a dynamic_cast on a

reference.

7.3.6. incorrect dynamic_cast on reference: CPP

Check to establish when only valid reference casts are performed through *dynamic_cast* operator.

C++ Example:

```
1     #include <new>
2     static volatile int random = 1;
3     class Object {
4     protected:
5       static Object* obj;
6     public:
7       virtual ~Object() {}
8     };
9
10    class Item : public Object {
11    private:
12      static Item* item;
13    public:
14      Item& get_item();
15      Item& other_item();
16    };
17
18    Object* Object::obj = new Object;
19
20    Item& Item::get_item() {
21      Item& ref = dynamic_cast<Item&>(*Object::obj);  // CPP ERROR: [incorrect
dynamic_cast on reference]
22      *item = ref;                                    // unreachable code
23    }
24
25    void main ()
26    {
27      Item * first= new Item();
28      if (random)
29        first->get_item();                          // NTC ERROR: propagation of
dynamic_cast reference error
30      Object &refo = dynamic_cast<Object&>(first->other_item()); // CPP Verified:
[dynamic_cast on reference is correct]
31    }
```

Explanation:

Only the dynamic casting between a subclass and its upclass is authorized. So, the casting of reference *Object* object to a reference *Item* object is an error on *dynamic_cast* at line 20, because *Object* is not a subclass of *Item*.
The analysis stops at line 20 and the error is propagated to a NTC error at line 28. The behaviour is different with a dynamic_cast on a pointer.

7.3.7. invalid pointer to member: OOP

PolySpace checks that the pointer to a function member is invalid or null.

C++ Example:

```

static volatile int random = 1;

class Object {
public:
  Object(int numero) : numeroObject(numero) {};
  int returnNumero(){return numeroObject;};
  virtual void displayNumero(Object* o){ o->returnNumero();};
  virtual void displayObject(){};
private:
  int numeroObject;
};

class DerivedBase : public Object {
public:
  DerivedBase(): Object(0){};
  virtual void initDerivedVirtual();
};

typedef void (DerivedBase::*pDerived)();
typedef int  (Object::*pDisplayNumero)(Object*);

typedef void  (Object::*pDisplayObj)();

void main (void)
{

  Object* newObject = new Object(5);
  pDisplayNumero ptDisp = 0 ;

  if (random)
    (newObject ->*ptDisp)(newObject); // OOP ERROR: [pointer to member function is null or points to an invalid member function]

  pDerived pDeriv = &DerivedBase::initDerivedVirtual;
  pDisplayObj pbv = static_cast <pDisplayObj> (pDeriv);

  if (random)
    (newObject->*pbv)(); // OOP ERROR: [pointer to member function is null or points to an invalid member function]
}
```

Explanation:

When a function pointer operates on a null pointer to a member value, the behavior is undefined. In the above example, the *ptDisp* pointer is declared and initialized to a null member function. When the function is called (at line 32) a red error is raised.

In the second set of inctructions, *DerivedBase* inherit from Object class. A function that operates on a *DerivedBase* can possibly access fields that a *Object* would not have, therefore it is not type-safe to call a pointer to member that is of a *DerivedBase* if it has been obtained by up-cast. A down-cast should be performed first. PolySpace displays a red message when the function obtained by up-cast is called (line 38).

Previous Back to table of contents Next

7.3.8. Call of pure virtual function: OOP

This check detects a pure virtual function call.

C++ Example:

```
1
2     class Form
3     {
4     public:
5       Form(Form* f){};
6       Form(Form* f, char* title){
7         f->draw();  // OOP Warning: [call of pure virtual function draw()]
8       };
9       virtual void draw() = 0;
10    };
11
12    class Rectangle : public Form
13    {
14    public:
15      Rectangle(): Form (this, "Rectangle"){} ;
16      void draw();
17    };
18
19    void Rectangle::draw () {
20      Form::draw(); // Draw the rectangle
21    };
22
23    void main (void)
24    {
25      Rectangle Rect1;
26      Rect1.draw();
27    }
```

Explanation:

The effect of making a virtual call to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined (see Standard ANSI ISO/IEC 1998 pp. 199).

One *Rectangle* object is declared: *Rect1* calls the constructor (line 15), and so the *Form* constructor (line 6) whose the *draw()* function member is called. Unfortunately, this function is a pure virtual function. PolySpace points out a warning at line 7.

Previous Back to table of contents Next

7.3.9. incorrect type for this-pointer: OOP

Check to verify that a member function is associated to the right instance of a class.
Three principal causes lead to an incorrect this-pointer type:

- An out of bounds pointer access
- A non initialized variable member
- An inadequat cast.

Following example shows the three possible cases.

C++ Example:

```
#include <new>

extern int rand;

struct A {
  virtual int f();
};

struct C {
  virtual int h() { return 7; }
};

struct T {
  int m_j;
  C m_field;
  T() : m_j(m_field.h()) {} // OOP ERROR (initialisation): [incorrect this-pointer type of T]
} badInit;

class Bad
{
public:
  int i;
  void f();
  Bad() : i(0) {}
};


class Good
{
public:
  virtual void g() {}
  void h() {}
  static void k() {}
};
```

```
35
36    int main()
37    {
38
39      A* a = new A;
40      Good *ptr = (Good *)(void *)(new Bad);
41
42      a->f();             // OOP Verified: [this-pointer type of A is correct]
43
44      if (rand) {
45        C* c = new C;
46        ++c;
47        c->h();           // OOP ERROR (out of bounds): [incorrect this-pointer type of C]
48      }
49
50      if (rand) ptr->g(); // OOP ERROR (cast): [incorrect this-pointer type of Bad]
51      if (rand) ptr->h(); // OOP ERROR (cast): [incorrect this-pointer type of Bad]
52
53      ptr->k();  // correct call to a static function
54    }
55
```

Explanation:

At line 16 of previous example, PolySpace points out here a this-pointer type problem (OOP category), because of an initialisation missing for member field *m_field*. THis problem raises at definition of badInit variable at line 55.

At line 47 of previous example, PolySpace points out that even if the function member *h* is part of the c Class, we are outside the structure. It could be compared to IDP for simple class.

At last, lines 50 and 51 show anothers this-pointer problems: function members g and h are not part of the *Bad* Class. *Good* does not inherite from *Bad*. Note that there is no problem with static function member *k* because it is only syntaxic.

7.3.10. potential call to: INF

[potential call to] are informative checks that help to understand reasonning of PolySpace during function calls, constructions and destructions of objects through

C++ Example:

```
#include <iostream>
static volatile int random_int = 1 ;

typedef enum { AOP, UTC, GET } valueKind;

class SubVal {
  valueKind val;
  void init();
public:
  SubVal(valueKind v);
  virtual ~SubVal() {}
  virtual void log(const char* msg);
  valueKind getVal() {return val;};
  void undef();
};

SubVal::SubVal(valueKind v) : val(v) {
  init();
}

void SubVal::init() {
  log("SubVal creation");                // INF informative: [call of log during construction]
}

void SubVal::log(const char* msg) {
  cout << msg;
}

void SubVal::undef() {
  log("ArithVal destruction");           // INF informative: [call of log during destruction]
}

class ArithVal : SubVal {
public:
  ArithVal(double d) : SubVal(GET) {}
  ~ArithVal();
  void ArithAdd(double d) {};
  virtual void log(const char* msg) {
    cout << getVal();
  };
  };

  ArithVal::~ArithVal() {
    undef();
  }

  void main(void){
    ArithVal *xVal = new ArithVal(10.0);
    xVal->ArithAdd(1.0);    // INF informative: [call to X::function()]

    SubVal *eVal = new SubVal(AOP);
    eVal->log("new");         // INF informative: [potential call to X::function
()]

    delete xVal;
    delete eVal;
  }
```

Explanation:

In this example, a base and derived classes are described. From main program, we create objects, call member functions and delete them. Associated to each function call, including constructors and destructors, some informative checks are put giving (potential) call of functions, during construction and destruction of objects.

Theses checks can only be green or grey.

7.3.11. Non-Initialized Variable: NIV/NIVL

Check to establish whether a variable local or not is initialized before being read. We make a distinction between local variables (including parameters of functions) and others. So PolySpace checks for same problems into two categories.

C++ example:

```
extern int random_int(void);
typedef double tab[20];


class operation
{
public:
  int addI(int x, int y) { return y+=x; };

  void initTab(){
    for (int i = 1; i < 20; i++) {
      twentyFloat[i] = 0.0;
    }
  };

  void addD(int x, int y){
    twentyFloat[x] = twentyFloat[y] + 5.0; // Unproven NIV: index 0 is not initialized.
  };

protected:
  tab twentyFloat;
};


void main(void)
{
  operation calculate;
  int x, y = 0;

  if (random_int()) {
    calculate.addI(x,y);                          // NIV ERROR: Non Initialized Variable
  }

  calculate.initTab();
  calculate.addD(2,4);

}
```

Explanation:

The result of the addition is unknown at line 28 because *x* is not initialized, (UNR unreachable code on "+" operator).
In addition, line 16 shows how Polyspace Verifier prompts the user to investigate further (by means of an orange check) when all cells have not been initialized.
A local variable is notified with a NIVL acronym.

Note that the message associated with the check NIV or NIVL can give the type of the variable if it concerns a basic type: *"variable may be non initialized (type unsigned int32)"*. The modifier *volatile* can also be notified: *(type : volatile unsigned int 8*).

Previous Back to table of contents Next

7.3.12. Non-Initialized Pointer: NIP

Check to establish whether a pointer is initialized before being dereferenced.

C++ example:

```
1      class declare
2      {
3      public:
4        declare(int* p):pointer(p){};
5        int changeValue(int val){*pointer = 0;};
6      protected:
7        int* pointer;
8      };
9
10     void main(void)
11     {
12       int* p;
13       declare newPointer(p);         // NIP ERROR: pointer not initialized
14       newPointer.changeValue(0);
15     }
```

Explanation :

As *p* is not initialized, the line 5 (**pointer = 0*) would overwrite an unknown memory cell (corresponding to the unreachable grey code on "*").

7.3.13. User Assertion failure: ASRT

Check to establish whether a user assertion is valid. If the assumption implied by an assertion is invalid, then the standard behavior of the assert macro is to abort the program. Verifier therefore considers a failed assertion to be a runtime error.

C++ Example:

```
#include <assert.h>

typedef enum
{
  monday=1, tuesday,
  wensday,  thursday,
  friday,   saturday,
  sunday
} dayofweek  ;

// stubbed function
dayofweek random_day(void);
int random_value(void);

void main(void)
{
  unsigned int var_flip;
  unsigned int flip_flop;
  dayofweek curDay;
  unsigned int constant = 1;

  if (random_value()) flip_flop=1; else flip_flop=0; // flip_flop randomly be 1 or 0
  var_flip = (constant | random_value());            // var_flip is always > 0

  if(random_value()) {
    assert(flip_flop==0 || flip_flop==1); // ASRT Verified: user assertion is verified
    assert(var_flip>0);                   // ASRT Verified
    assert(var_flip==0);                  // ASRT ERROR: user assertion fails
  }

  if (random_value()) {
    curDay = random_day();                // Random day of the week
    assert( curDay > thursday);           // ASRT Warning: User assertion may fail
    assert( curDay > thursday);           // ASRT Verified
    assert( curDay <= thursday);          // ASRT ERROR: user assertion fails
  }
}
```

Explanation:

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

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

7.3.14. Overflows and underflows

Related subjects :

Previous Back to table of contents Next

7.3.14.1. Scalar and Float Overflows: OVFL

Check to establish whether an arithmetic expression overflows. This is a scalar check with integer type and float check for floating point expression.

C++ Example:

```
#include <float.h>

extern int random_int(void);

class Calcul
{
public:
  int makeOverflow(int i){
    return  2 * (i - 1) + 2;   // OVFL ERROR: [scalar variable overflows on [+] ...]
    // 2^31 is an overflow value for int32
  }
  float overflow (float value){
    return  2 * value + 1.0;   // OVFL ERROR: [float variable overflows on [conversion from ...]]
  }
};


void main(void)
{
  Calcul c;
  int i = 1;
  float fvalue = FLT_MAX;

  i = i << 30;                          // i = 2**30

  if (random_int())
    i = c.makeOverflow(i);              // NTC ERROR: propagation of OVFL ERROR

  if (random_int())
    fvalue = c.overflow(fvalue);        // NTC ERROR: propagation of OVFL ERROR
}
```

Explanation:

On a 32-bits architecture platform, the maximum integer value is *2^31-1*, thus *2^31* will raise an overflow. In the same manner, if *fvalue* represents the biggest float its double cannot be represented with same type and raises an overflow.

7.3.14.2. Scalar and Float Underflows: UNFL

Check to establish whether an arithmetic expression underflows. This is a scalar check with integer type and a float check for floating point expressions.

C++ Example:

```
#include <float.h>

extern int random_int(void);

class Calcul
{
public:
  int makeUnderflow(int i){
    return i - 1;              // UNFL ERROR: scalar variable is underflow
  }
  float underflow (float value){
    return - 2 * value;        // UNFL ERROR: float variable is underflow
  }
};


void main(void)
{
  Calcul c;
  int i = 1;
  float fval = FLT_MAX;

  i = -2 * (i << 30);            // i = -2**31

  if (random_int())
    i = c.makeUnderflow(i);    // NTC ERROR: propagation of UNFL ERROR

  if (random_int())
    fval = c.underflow(fval); // NTC ERROR: propagation of UNFL ERROR
}
```

Explanation:

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

7.3.14.3. Float underflow and overflow: UOVFL

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

Example:

```
#include <math.h>
extern int prand (void);
#define FLT_MAX 3.40282347e+38F

int toto(void)
{
  float x;
   if (prand ())
    {
      x = -FLT_MAX;
    }
   else if (prand ())
     {
       x = FLT_MAX;
     }
   else
     {
       x = 0;
     }
   x = 2.0F * x;              // UOVFL unproven: possible overflow and underflow
   return 1;
}
```

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

7.3.14.4. Overflow on the biggest float

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

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

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

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

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

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

7.3.14.5. Constant overflow

Consider the following example, which would cause an overflow.

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

The type given to a constant is the first type which can accommodate its value, from the appropriate sequence shown below. (Please refer to "Target specification" section for information about the size of a type depending on the target.)

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

For examples (assuming 16-bits target):

`5.8`	`double`
`6`	`int`
`65536`	`long`
`0x6`	`int`
`0xFFFF`	`unsigned int`
`5.8F`	`float`
`65536U`	`unsigned int`

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

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

7.3.14.6. Float underflow versus values near zero

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

PolySpace analyses apply the former definition.

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

Consider the following example.

```
#define FLT_MAX      3.40282347e+38F   // maximum representable float found in <float.h>
#define FLT_MIN      1.17549435e-38F   // minimum normalised float    found in <float.h>

void main(void)
{
  float zer_float =   FLT_MIN;
  float min_float = -(FLT_MAX);

  zer_float = zer_float * zer_float; // No check underflow near zero. VOA says {[expr] = 0.0}
  min_float = min_float * min_float; // UNFL ERROR: underflow checked by verifier

}
```

7.3.15. Scalar or Float Division by zero: ZDV

Check to establish whether the right operand of a division (denominator) is different from 0[.0].

C++ example:

```
extern int random_value(void);

class Operation {
public:
  int zdvs(int p){
    int j = 1;
    return (1024 / (j-p));   // ZDV ERROR: Scalar Division by Zero
  }
  float zdvf(float p){
    float j = 1.0;
    return (1024.0 / (j-p)); // ZDV ERROR: float Division by Zero
  }
};

int main(void)
{
  Operation op;

  if (random_value())
    op.zdvs(1);                   // NTC ERROR: propagation of ZDV ERROR.

  if (random_value())
    op.zdvf(1.0);                 // NTC ERROR: propagation of ZDV ERROR.
}
```

7.3.16. Shift amount is outside its bounds: SHF

Check to establish that a shift (left or right) is not bigger than the size of integral type (int and long int). The range of allowed shift depends on the target processor: 16 bits on *c-167*, 32 bits on *i386* for int, etc.

C++ Example:

```
extern int random_value(void);

class Shift {
public:
  Shift(int val) : k(val){};
  void opShift(int x, int l){
    k = x << l;                  // SHF ERROR: [scalar shift amount is outside its bounds 0..31]
  }
  void opShiftSup(int x, int l){
    k = x >> l;                  // SHF ERROR: [scalar shift amount is outside its bounds 0..31]
  }
  void opShiftUnsigned(unsigned int x, int l){
    unsigned int v = 1024;
    v = x >> l;                  // SHF ERROR: [scalar shift amount is outside its bounds 0..31]
  }
protected:
  int k;
};


void main(void)
{
  int m, l = 1024;           // 32 bits on i386
  unsigned u = 1024;

  Shift s(1024);

  if (random_value()) s.opShift(l ,32 );         // NTC ERROR: propagation of SHF ERROR
  if (random_value()) s.opShiftUnsigned(u ,32 ); // NTC ERROR: propagation of SHF ERROR
  if (random_value()) s.opShiftSup(l ,32 );      // NTC ERROR: propagation of SHF ERROR

}
```

Explanation:

In this example, we just show that shift amount is greater than the integer size.

7.3.17. Left operand of left shift is negative: SHF

Check to establish whether the operand of a left shift is a signed number.

C++ example:

```
1      extern int random_value(void);
2
3      class Shift {
4      public:
5        Shift(){};
6        int operationShift(int x, int y){
7          return x << 1;  // SHF ERROR: left operand of left shift is negative
8        }
9      };
10
11
12     void main(void)
13     {
14       Shift* s = new Shift();
15
16       if (random_value())
17         s->operationShift(-200,1); // NTC ERROR: propagation of SHF ERROR
18     }
```

Explanation:

As signed number representation is stored in the higher order bit, you can not left-shift a signed number without loosing sign information.

As an aside, note that the ***-allow-negative-operand-in-shift*** option used at launching time instructs PolySpace to allow explicitly signed numbers on shift operations. Using the option in the current example, the red check at line 8 is transformed in a green one.

7.3.18. Power must be positive: POW

Check to establish whether the left operand of the *pow* mathematical function declared in is positive (directly or in generated constructors or destructors)

C++ example:

```cpp
#include <math.h>

static volatile int random_int = 1;
static unsigned int rPositive;

class Numeric_power
{
public:
  Numeric_power(unsigned int *baseV, int *exponentV);
  ~Numeric_power(){};
private:
  double powerValue;
};

Numeric_power::Numeric_power(unsigned int *baseV, int *exponentV){
  powerValue = pow(*baseV,*exponentV);   // POW Warning: [power may be not positive]
}

double calculate_power (int baseValue,int exponentValue) {
  return pow(baseValue,exponentValue);  // POW Warning: [power may be not positive]
}

void main (void)
{
  int x[3] = {3, 5, -1};
  int negative = -(13%2);
  volatile unsigned int pr;

  if (random_int) {
    rPositive = pr;
    Numeric_power p(&rPositive,x);
  }

  if (random_int) calculate_power(negative,4);
  if (random_int) pow(-7,4); // POW Warning: [power may be not positive]

}
```

Explanation:

The *numeric_power* constructor initialises its class member *power_value* with two arguments. At line 26, the first argument, the left operand, is a volatile unsigned int, and even it is safe, a warning message is display by PolySpace. The *calculate_power* function has a negative argument on the left but Polyspace points out a warning instead of a real problem. Remeber that behind orange there is also run-time errors.

7.3.19. Array index is outside its bounds: OBAI

Check to establish whether an index is compatible with the length of the array being accessed.

C++ example:

```
#define TAILLE_TAB  1024
typedef int tab[TAILLE_TAB];

class Array
{
public:
  Array(){};
  void initArray();
private:
  tab table;
};


void Array::initArray()
{
  int index;

  for (index = 0; index < TAILLE_TAB ; index++){
    table[index] = 10;
  }
  table[index] = 1;  // OBAI ERROR: [out of bounds array index]
};


void main(void)
{
  Array* test = new Array();
  test->initArray();    // NTC ERROR: propagation of OBAI ERROR
}
```

Explanation:

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

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

7.3.20. Function pointer must point to a valid function: COR

Check to establish whether a function pointer points to a valid function, or to function with a valid prototype.

C++ example:

```
typedef void (*CallBack)(void *data);

struct {
  int ID;
  char name[20];
  CallBack func;
} funcS;

float fval;

void main(void)
{
  CallBack cb =(CallBack)((char*)&funcS + 24 * sizeof(char));

  cb(&fval); // COR ERROR: function pointer must point to a valid function
}
```

Explanation:

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

7.3.21. Wrong number of arguments: COR

Check to establish whether the number of arguments passed to a function matches the number of argument in its prototype.

C++ example:

```
extern int random_value(void);

typedef int (*t_func_2)(int);
typedef int (*t_func_2b)(int,int);

int foo_nb(int x)
{
  if (x%2 == 0)
    return 0;
  else
    return 1;
}

void main(void)
{
  t_func_2b ptr_func;
  int i = 0;

  ptr_func = (t_func_2b)foo_nb;
  if (random_value())
    i = ptr_func(1,2); // COR ERROR: [function pointer must point on a valid function]
  // COR Warning: [wrong number of arguments for call to function foo_nb(int): got 2 instead of 1]
}
```

Explanation:

In this example, *ptr_func* is a pointer to a function that takes two arguments but it has been initialized to point to a function that only takes one.
In this case this is the associated COR warning which explains the COR ERROR: *[wrong number of arguments for call to function : got instead of]*, where is the number of argument used and the number of argument waited.

7.3.22. Wrong type of argument: COR

Check to establish whether each argument passed to a function matches the prototype of that function.

C++ example:

```
static volatile int random = 1;

int f(float f) { return 0; }
int g(int i) { return i; }

typedef int (*func_int)(int);

func_int ftab = (func_int)f;

void badTab(int i) {
  ftab(++i) ;   // COR ERROR: [function pointer must point on a valid function]
  // COR Warning: [wrong type for argument #1 of call to function f(float)]
}

int main()
{
  int idx = 0;

  for (int i = 9; i < 10; ++ i) {
    if (random)
      badTab(++idx); // NTC ERROR: propagation of COR ERROR
  }
}
```

Explanation:

In this example, *tab* is an function pointer to functions which expects a float as input argument. However, the parameter used is an *int*. So PolySpace Viewer prompts the user to check the validity oh the code.
In this case, this is the associated COR warning which explains the COR ERROR: *[wrong type for argument # of call to function]*, where gives the location of the wrong argument in the function.

7.3.23. Pointer is outside its bounds: IDP

Check to establish whether the dereferenced pointer is still inbound of the pointed object.

C++ example:

```
#define TAILLE_TAB  1024

typedef int tab[TAILLE_TAB];

class Array {
public:
  Array(tab a){
    p = a;
    initArray();
  }
  void initArray(){
    int index;
    for (index = 0; index < TAILLE_TAB ; index++, p++) {
      *p = 0;
    }
  }
  void changeNextElementWithValue(int i){
    *p = i;                      // IDP ERROR: pointer is outside its bound
  }

private:
  int *p;
};


void main(void)
{
  tab t;

  Array a(t);
  a.changeNextElementWithValue(1); // NTC ERROR: propagation of IDP ERROR
}
```

Related subjects :

7.3.23.1. Understanding addressing
7.3.23.2. Understanding pointers

Explanation:

The pointer *p*is initialized to point to the first element of *tab*at line 4. When the loop exits, *p*points one past the last element of the array. Thus line 16 overwrites this cell.

7.3.23.1. Understanding addressing

Related subjects :

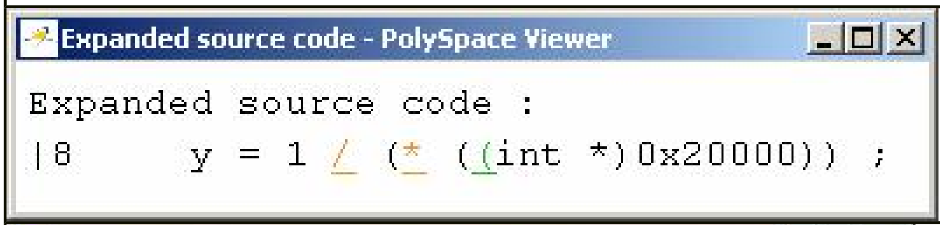
7.3.23.1.1. hardware registers

Many code analyses exhibit **orange** out of bound checks with respect to accesses to absolute addresses and/or hardware registers.

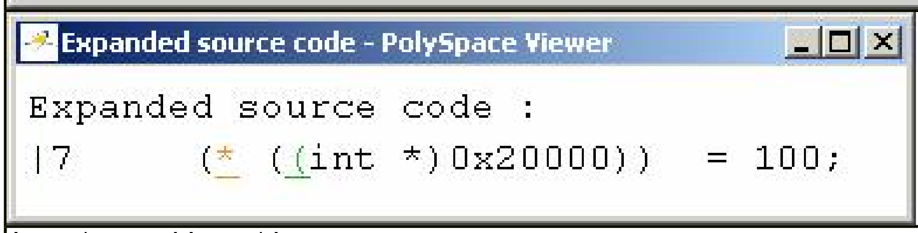
(Also refer to the discussion on Absolute Addressing)

Here is an example of what such code might look like:

```
#define X (* ((int *)0x20000))
X = 100;
y = 1 / X;  // ZDV check is orange because X ~ [-2^31, 2^31-1] permanently.
            // The pointer out of bounds check is orange because 0x20000
            // may address anything of any length
            // NIV check is orange on X as a consequence
```



```
Expanded source code :
|8      y = 1 / (* ((int *)0x20000)) ;
```



```
Expanded source code :
|7      (* ((int *)0x20000))  = 100;
```

```
3  void  main (void)
4  {
5  int y;
6
7  X = 100;
8  y = 1 / X;
9
10    }
```

```
int *p = (int *)0x20000;
*p = 100;
y = 1 / *p; // ZDV check is orange because *p ~ [-2^31, 2^31-1] permanently
            // The pointer out of bounds is orange because 0x20000
            // may address anything of any length
            // NIV check on *p is orange as a consequence
```

This can be addressed by defining registers as regular variables:

Replace	By
`#define X ....`	`int X;`

`int *p;`	`int _p;`
	`#define p (&_p)`
	Note Check that the chosen variable name (p in this example) doesn't already exist

`int *p;`	`volatile int _p;`
	`int *p = &_p;`

The [volatile section](#) discusses an approach which will help avoid the orange check on the pointer dereference, but retains the representation of a "full range" variable.

7.3.23.1.2. NULL pointer

Consider the NULL address, viz.

```
#define NULL 0
```

· It is illegal to dereference this 0 value
· 0 **is not** treated as an absolute address.

```
*NULL = 100; // produces a red - Illegal Dereference Pointer (IDP)
```

- Assuming these declarations:-

```
int *p = 0x5;
volatile int y;
```

- and these definitions:-

```
#define NULL 0
#define RAM_MAX ((int *)0xffffffff)
```

- consider the code snippets below.

```
While (p != (void *)0x1)
  p--; // terminates
```

0x1 is an absolute address, it can be reached and the loop terminates

```
for (p = NULL; p <= RAM_MAX; p++)
{
  *p = 0; // illegal dereference of pointer
}
```

At the first iteration of the loop p is a NULL pointer. Dereferencing a NULL pointer is forbidden.

```
While (p != NULL)
{
  p--;
  *p = 0; // Orange dereference of a pointer
}
```

When p reaches the address 0x0, there is an attempt to considered it as an absolute address
In effect, it is an attempt to dereference a NULL pointer – which is forbidden.
Note that in this case, the **check** is **orange** because the execution of the code here is ok (green) until 0x0 is reached (red)

The best way to address this issue depends on the purpose of the function.

- Thanks to the default behaviour of PolySpace, it is easy to automatically stub a function whose purpose is to copy data from/to RAM or to compute a checksum on RAM.
- If a function is supposed to copy calibration data, it should also be stubbed automatically.
- If the purpose of a function is to map EEPROM data to global variables, then a manually written stub is essential to ensure the assignment of the correct initialisation values to them.

7.3.23.1.3. Comparing address

PolySpace only deals with the information referred to by a pointer, and not the physical location of a variable. Consequently it does not compare addresses of variables, and makes no assumption regarding where they are located in memory.

Consider the following two examples of PolySpace behaviour:

```
int a,b;
if (&a > &b) // condition can be true and/or false
{ } // both branches are reachable
else
{ } // both branches are reachable
```

and

```
int x,z;
void main(void)
{ int i;
  x = 12;
  for (i=1; i<= 0xffffffff; i++)
  {
    *((int *)i) = 0;
  }
  z = 1 / x; // ZDV green check because PolySpace doesn't consider any
             // relationship between x and its address
}
```

“x” is aliased by no other variable. No pointer points to “x” in this example, so as far as the PolySpace analysis is concerned, “x” remains constantly equal to 12.

7.3.23.2. Understanding pointers

PolySpace doesn't analyse anything which would require the physical address of a variable to be taken into account.

- Consider two variables x and y. PolySpace analysis will not make a meaningful comparison of “&x” (address of x) and “&y”
- So, the Boolean (&x < &y) can be true or false as far as PolySpace analysis is concerned.

However, PolySpace analysis does keep track of the pointers that point to a particular variable.

- So, if ptr points to X, *ptr and X will be synonyms.

Related subjects :

[**7.3.23.2.1. How does malloc work for PolySpace?**](#)
[**7.3.23.2.2. Structure Handling**](#)

7.3.23.2.1. How does malloc work for PolySpace?

PolySpace analysis accurately models malloc, such that both the possible return values of a null pointer and the requested amount of memory are taken into account.

Consider the following example.

```
void main(void)
{
  char *p;
  char *q;
  p = malloc(120);
  q = p;
  *q = 'a';  // results in an orange dereference check
}
```

This code will avoid the orange dereference:

```
void main(void)
{
  char *p;
  char *q;
  p = malloc(120);
  q = p;
  if (p!= NULL)
    *q = 'a';  // results in a green dereference check
}
```

Previous Back to table of contents Next

7.3.23.2.2. Structure Handling

Related subjects :

7.3.23.2.2.1. Array conversions: COR

Check to establish whether a small array is mapped onto a bigger one through pointer cast.

C++ Example:

```
1      typedef int Big[100];
2      typedef int Small[10];
3      typedef short EquivBig[200];
4
5      Small smalltab;
6      Big bigtab;
7
8      extern int random_val();
9
10     void main(void)
11     {
12
13       Big   * ptr_big   = &bigtab;
14       Small * ptr_small = &smalltab;
15
16       if (random_val()){
17         Big *new_ptr_big = (Big*)ptr_small;     // COR ERROR: array conversion must not extend range
18       }
19
20       if (random_val()){
21         EquivBig *ptr_equivbig = (EquivBig*)ptr_big;
22         Small *ptr_new_small = (Small*)ptr_big; // COR Verified
23       }
24     }
```

Explanation:

In the example above, a pointer is initialized to the *Big* array with the address of a the *Small* array. This is not legal since it would be possible to dereference this pointer outside of the *Small* array. Line 22 shows that the mapping of arrays with same length and different prototypes is authorised.

7.3.23.2.2.2. Mapping of a small structure into a bigger one

For example, if *p* is a pointer to an object of type *`t_struct`* and it is initialized to point to an object of type *`t_struct_bis`* whose size is less than the size of *`t_struct`*, it is illegal to dereference *p* because it would be possible to access memory outside of *`t_struct_bis`.* PolySpace prompts user to investigate further by means of an orange check. See the following example.

```
#include <malloc.h>

typedef struct {
  int a;
  union {
    char c;
    float f;
  } b;
} t_struct;

void main(void)
{
  t_struct *p;

  // optimize memory usage
  p = (t_struct *)malloc(sizeof(int)+sizeof(char));

  p->a = 1;  // IDP Warning: pointer may be outside its bounds

}
```

7.3.24. logic_error is thrown: EXC

This check determines whether a logic_error is raised.

C++ Example:

```cpp
#include <stdexcept>
#include <vector>
#include <stdio.h>

using namespace std;

int maxSizeTable = 10;

class ComputerFirm : private out_of_range
{
public:
  ComputerFirm(int number): out_of_range ("error") { // EXC Verified: [logic_error is not thrown]
    numberC = number;
  };
  int whichQuantity(int i) throw (logic_error);
  void InitComputerFirm()  throw (out_of_range);

protected:
  unsigned int numberC ;
  vector<int> table;
};

void ComputerFirm::InitComputerFirm() throw (out_of_range) // EXC Warning: [logic_error may be thrown]
{
  table.resize(numberC);
  for (int i = 0; i < table.size(); i++) {
    try {
      if (i >= maxSizeTable)
        throw out_of_range("out_of_range");
      table[i] = 1 ;
    }
    catch (...){
      throw;
    };
  };
};

int ComputerFirm::whichQuantity(int i) throw (logic_error) // EXC ERROR:
```

```
[logic_error is thrown (analysis jumps to enclosing handler)]
39    {
40      if (i > maxSizeTable)
41        throw logic_error ("logic_error");
42      return table[i];
43    }
44
45    void main (void)
46    {
47      try {
48        ComputerFirm* pRichardFirm = new ComputerFirm(10);
49        pRichardFirm->InitComputerFirm();
50        int q = pRichardFirm->whichQuantity(12);  // EXC ERROR: [call to
whichQuantity thrown (analysis jumps to enclosing handler)]
51      }
52      catch (const exception& e) {
53        int ret = printf("error"); // display error message
54      }
55    }
```

Explanation:

Here, a pointer of class *ComputerFirm* named *pRichardFirm* is created, with three functions members and a vector of integers with a maximum size of 10 elements. The class *logic_error* defines the type of objects thrown as exceptions to report errors presumably detectable before the program executes, such as violation of logical preconditions.

At line 49, *InitComputerFirm* is called. This function initializes all elements to 1. In this function (line 28), the size of the created vector is checked against the maximum authorized size. In this case, the vector's size and the maximum size are both equal to 10, so no logic_error is raised but du to imprecision PolySpace does not exactly know if it is the case or not leading to a warning message.

At line 50, a second function member is called, *whichQuantity* which returns the value of the element in parameter. For this call the value is 12, greater than the maximum size, therefore a logic error is detected. A red message is displayed on the function definition on word *throw*(see line 36).

Note: this check is positionned only when class logic error is found in the source code. A code which does not include stdexcept header will not get this useless information.

7.3.25. runtime_error is thrown: EXC

Check to establish that a runtime_error exception is raised.

C++ Example:

```
#include <stdexcept>
#include <math.h>
#include <float.h>
#include <limits.h>
using namespace std;

static volatile int random_int = 1;
double maxFloat = FLT_MAX;
short int sInt = SHRT_MAX;


class RUE : runtime_error
{
public:
  RUE() : runtime_error("rien"){}
  RUE(double numberToAdd) : runtime_error("overflow"){   // EXC Verified: [runtime_error is not thrown]
    result = pow(maxFloat,maxFloat) ;
  };
protected:
  float result;
};


long addDouble(float firstOperand, int secondOperand) throw (int,runtime_error) { // EXC ERROR: [runfunction throws (analysis jumps to enclosing handler)]
  if ((firstOperand+secondOperand) > LONG_MAX)
    throw runtime_error("overflow");
  else
    return firstOperand + secondOperand;
}

int f(){ // EXC unreachable: [function does not throw]
  RUE r;
  throw r;
};

void main(void)
{
  try {
    RUE testAddition(maxFloat);
    addDouble(maxFloat,sInt); // EXC ERROR: [call to addDouble() throws (analysis jumps to enclosing handler)]
41        f();
42      }
43      catch (runtime_error) {
44        cout << "Error : overflow";
45      }
46    }
```

Explanation:

The class *runtime_error* defines the type of objects thrown as exceptions to report errors presumably detectable only when the program executes.
In this example, we create an object RUE with the maximum size of a float as parameter (line 39). Its constructor can catch runtime_error. A green message acknowledges that no runtime_error has been thrown.
We define the function *addDouble* (line 24) which can catch *int* exception and *runtime_error* exception. Its definition shows that if the addition of the two parameters is greater than the maximum size of a long integer, a runtime_error is thrown.
Indeed, in the following execution *addDouble* receives as parameter the maximum size of a short integer and float. So, a runtime_error exception is thrown (see line 26). PolySpace propagates to the definition of the function with a runtime_error red error.

As the logic_error, PolySpace checks functions using stdexcept library.

7.3.26. Function throws: EXC

Check to verify that a function never raises an exception for every returned values.

C++ Example:

```
#include <vector>

static volatile int random_int = 1;
class error{};

class InitVector
{
public:
  InitVector (int size) {
    sizeVector = size;
    table.resize(sizeVector);
    Initialisation();
  };
  void Initialisation ();
  void reSize(int size);
  int getValue(int number) throw (error);
  int returnSize();
private:
  int sizeVector;
  vector<int> table;
};

void InitVector::Initialisation() { // EXC Warning: [functions may throw]
  int i;
  for (i = 0; i < table.size(); i++){
    table[i] = 0;
  }
  if (random_int) throw i;
}

void InitVector::reSize(int sizeT) {
  table.resize(sizeT);
  sizeVector = table.size();
}

int InitVector::getValue(int number) throw (error) { // EXC ERROR: [function throws (analysis jumps to enclosing handler)]
  if (number >= 0 && number < sizeVector)
    return table[number];
  else throw error();
}

```

```
42    int InitVector::returnSize() { // EXC Verified: [function does not throw]
43      return table.size();
44    }
45
46    void main (void)
47    {
48      InitVector *vectorTest = new InitVector(5);
49
50      if (random_int)
51        vectorTest->returnSize();
52
53      if (random_int)
54        vectorTest->getValue(5); // EXC ERROR: [call to getValue throws (analysis
jumps to enclosing handler)]
55    }
```

Explanation:

The class *InitVector* allows to create a new vector with a defined size. The *resize* member function allows to change the size, without any size limit. *returnSize* returns the vector's size, and no exception can be thrown. A green check is displayed for this function: *[function does not throw]*.
The *getValue* function returns the array's value for a given index. If the parameter is outside vector bounds, an exception is raised. For a vector'size of 5 elements, valid index are [0..4]. At line 53, the programmers tries to access the fifth element *table[5]*. An exception is raised and Polyspace displays a red message.

Polyspace Verfier tests functions that raises exception or no, with void or no-void type:

- always: function throws (analysis jumps to enclosing handler)
- never: function does not throw
- sometimes: function may throw

When this check happens, a propagation to caller is made with another exception check [call to throws] (see line 53).

7.3.27. Call to throws: EXC

Check to verify that a function call raises or not an exception.

C++ Example:

```
static volatile int random_int =1 ;

class error{};

class A
{
public:
  A() {value=9;};
  int badReturn() throw (int);
  int goodReturn() throw (error);
protected:
  int value;
};

int A::badReturn() throw (int) { // EXC ERROR: [function throws (analysis jumps to enclosing handler)]
  if(!value)
    return value;
  else
    throw 2;
};

int A::goodReturn() throw (error) { // EXC Verified: [function does not throws]
  int p = 7;
  if (p>0)
    return value;
  else
    throw error();
};

void main (void)
{
  A* a = new A();
  if(random_int)
    a->badReturn(); // EXC ERROR: [call to badRetrun throws (analysis jumps to enclosing handler)]
  if(random_int)
    a->goodReturn(); // EXC Verified: [call to goodRetrun does not throw]
}
```

Explanation:

In the first call, Polyspace propages to caller that the function always raises an exception because member variable value is always different from 0.
In the second call, PolySpace checks that no throw has been made in the function because the conditional test at line 24 is always true.
Most of the time, the *[call to throws]* is associated to *[function throws]* check.

7.3.28. destructor or delete throws: EXC

Check to establish whenever an exception is throw and not catch in a destructor or during a delete.

C++ Example:

```
#include <math.h>
using namespace std;
volatile unsigned int random_int = 1 ;

class error{};

class Rectangle
{
public:
 Rectangle(){};
 Rectangle (unsigned int longueur, unsigned int large):longueurRect(longueur), largeRect(large){};

 virtual ~Rectangle(){   // EXC Warning: [possible throw during destructor or delete]
  if (!random_int)
   throw error();
 };

 virtual double calculArea() {
  return longueurRect * largeRect;
 };

protected:
 unsigned int longueurRect;
 unsigned int largeRect;
};

class Cube : public Rectangle
{
public:
 Cube():cote(3){};
 ~Cube(){          // EXC ERROR: [throw during destructor or delete]
  if(random_int>=0)
   throw error();
 };
 double calculArea(){
  return pow(cote,cote);
 };
protected:
 int cote ;
};

```

```
42    void main (void)
43    {
44      try {
45        Rectangle* form1 = new Rectangle(10,2);
46        double k = form1->calculArea();
47
48        Cube* form2 = new Cube;
49        double l = form2->calculArea();
50
51        delete form1;
52        delete form2;         // NTC ERROR: propagation of throw during destructor
53      }
54      catch (error){
55        //raised when an error occurs in a destructor
56      }
57      catch (...){}
58    }
```

Explanation:

In the class Cube's destructor at line 31, an error is raised when *random_int* is greater than 0. As random_int was declared as a volatile unsigned int, this condition is always true.
At line 13, in the desctructor of class Rectangle, the test on the *random_int* value may be true when it is different from 0. Thus, it is possible that the exception is raised or not in the destructor, and an orange warning is displayed instead.

Destructors are called during stack unwinding when an exception is thrown. In this case any exception thrown by a destructor would cause the program to terminate. Therefore it is better programming to catch exceptions in destructors.

7.3.29. Main, tasks or C library function throws: EXC

Check that functions used at C level, in a task or in main do not raise exceptions.

C++ Example:

```
#include <cstdlib>
#include <iostream>
static volatile int random_int = 1;

extern "C" {
  int compare (const void * a, const void * b) { // EXC Verifeid: [main, task or C library function does not throw]
    return ( *(int*)a - *(int*)b );
  }
  int c_compare_bad (const void *k, const void *e) { // EXC ERROR: [main, task or C library function throws]
    throw 1;
  }
};

typedef int arrayT[5];

class arrayToRange
{
public:
  arrayToRange(arrayT* a) :tab(a) {};
  arrayT* returnTabInOrder() {
    qsort(*tab, 5, sizeof(int), compare);
    return tab;
  };
  arrayT* returnTabInOrderBad() {
    qsort(*tab, 5, sizeof(int), c_compare_bad);
    return tab;
  };
protected:
  arrayT* tab;
};

void main(void) // EXC Verified: [main, task or C library function does not throw]
{
 try
 {
  arrayT tabInit = {1,3,4,2,5};
  arrayT* table = &tabInit;
  arrayToRange ArrayTest(table);
  ArrayTest.returnTabInOrderBad(); // No jump to enclosing handler
  ArrayTest.returnTabInOrder();
41     }
42     catch (...) {                       // grey code
43       cout << "error raised:" <<  "bye"; // grey code
44     }
45   }
```

Explanation:

In this example, we called a C stubbed function, *qsort* defined in the include file cstlib, which returns a sorted array of integers. Two functions, defined in a class called *arrayToRange*, call this *qsort* function:

- The first one, *returnTabInOrder*, calls *qsort*, with a C function pointer as third parameter, which can not raise an exception. So PolySpace displays a green message (line 6).
- The second one, *returnTabInOrderBad*, uses a C function pointer which always raises an exception. PolySpace displays a red message on the C function (line 9).

  Limitation: even if *c_compare_bad* function always raise an exception, PolySpace does not propagate to enclosing handler. Indeed at line 39, all is green and the analysis continue even if call is surrounded by a *try/catch* leading to grey code in catch block.

Previous Back to table of contents Next

7.3.30. exception raised is not specified in the throw list: EXC

Check to determine whether a function has thrown a non authorized exception.

C++ Example:

```
#include <string>

using namespace std;

int negative_balance = -300;

class NotPossible
{
public:
  >_&).COR.0.error.html" name="L10-C2">NotPossible(const string & s) : Error_Message(>_&).NIP.1.error.html" name="L10-C48">s)>_&).COR.2.error.html" name="L10-C50">{};
  ~NotPossible(){};
  string Error_Message;
};

class Account
{
public:
  Account(long accountInit):account(accountInit) {}
  void debit (long amount) throw (int, char);
  long getAccount () { return account; };
protected:
  long account;
};

void Account::debit(long amount) throw (int, char) { // EXC ERROR: [exception raised is not specified in the throw list]
  if ((account - amount) < negative_balance)
    throw NotPossible ("error");
  account = account - amount;
}

void main (void)
{
  try {
    Account *James = new Account(12000);
    James -> debit(13000);                // NTC ERROR: propagation of not specified exception
    long total = James -> getAccount();
  }
  catch (NotPossible&){}
```

```
39      catch (...){};
40    }
41
```

Explanation:

In the above example, the *Account* class is defined with the *debit* function which allows to throw the specified exception. This function can only catch the *int* and *char* exceptions. The bank authorized an overdraft of 300 euros. The James's account is created with an initial balance of 12000 Euros. So, at line 35, his account is debited with 13000. In the *debit* function, the *if* condition (line 27) is true, thus a *NotPossible* exception is raised. Unfortunately, this exception type is not allowed within the throw list at line 25 even if the catch operand allows it. So PolySpace detects an error.

7.3.31. throw during catch parameter construction: EXC

Check to prevent throw during dynamic initialisation in constructors and during initialization of arguments in in *catch*.

C++ Example:

```
#include <string>

static volatile int random_int = 1;
static volatile int random_red = 0;

class error{};

class NotPossible
{
public:
  NotPossible(const NotPossible&) // EXC ERROR: [function throws (analysis jump to enclosing handler)]
  {
    throw error();
  };
  NotPossible()                   // NRE ERROR: [function throws (analysis jump to enclosing handler)]
  {
    throw NotPossible(7);
  };
  NotPossible(int){};
  ~NotPossible(){};
private:
  string Error_Message;
};

class Test
{
public:
  Test(int val) : value(val){};
  int returnVal(){
    if (random_int)
      throw error();
    else
      return value;
  };
private:
  int value;
};

```

```
39    int main() {
40
41      try {
42        Test* T = new Test(1);
43        if (random_red)
44          throw NotPossible();  // EXC ERROR: [call to NotPossible throws
(analysis jumps tp enclosing handler)]
45        else
46          T->returnVal();
47        if (random_red) {
48          NotPossible * Npos = new NotPossible(); // EXC ERROR: [throw during
dynamic initialisation]
49        }
50      }
51      catch(NotPossible a) {}   // EXC ERROR: [throw during catch parameter
conctruction]
52      catch(...) {}
53    }
```

Explanation:

At line 48 of the previous example, during dynamic initialisation of Npos, a call to default constructor NotPossible is made. This constructor raises an exception leading to the EXC error. Indeed, raising an exception during a dynamic initialisation is not authorized.

In same example at line 51, an exception is catched by the throw coming from line 44. A variable of type *NotPossible* is created at line 48 using also same default constructor. However, this constructor throws an an *integer* exception leading to red error at line 48.
Each catch clause (exception handler) is like a function that takes a single argument of one particular type. The identifier may be used inside the handler, just like a function argument. Moreover, the throw of an exception in a catch block is not authorized.

7.3.32. Continue execution in __except: EXC

Check to establish whether in a __except catch block the use of MACRO EXCEPTION_CONTINUE_EXECUTION. This check can only occur using a visual dialect.

C++ Example:

```
1
2      #include <windows.h>
3      #include <excpt.h>
4
5      void* data;
6      struct No_Data {};
7
8      void* check_glob() {          // EXC ERROR: [function throws (analysis jumps
to enclosing handler)]
9        if (!data) throw No_Data(); // EXC ERROR: []
10       return data;
11     }
12
13     int main() {
14       __try {
15         data = 0;
16         check_glob();   // EXC ERROR: [call to check_glob() throws (analysis jumps
to enclosing handler)]
17       }
18       __except(data == 0
19                ? EXCEPTION_CONTINUE_EXECUTION  // EXC ERROR: [expression value is
EXCEPTION_CONTINUE_EXECUTION]
20                : EXCEPTION_EXECUTE_HANDLER) {
21         data = new (void*);                            // Grey code
22       }
23     }
```

Explanation:

In this example, the call to function check_glob() throws an exception. This exception jumps to enclosing handler, in this case the *__except* block. Using EXCEPTION_CONTINUE_EXECUTION, it could be possible normally to continue analysis and comes back at line 9 as if exception never happend. In the example, data is assigned to new value at line 21 in __except block and no more throw will occur.
PolySpace cannot handle this kind of behaviour and put a red error on the EXCEPTION_CONTINUE_EXECUTION keyword since it has found a path to this instruction. It results grey code at line 21 and at line 10. All other red errors concern management of the exception: function throws and call throws].

Note that it is possible to match functional behaviour using volatile keyword by replacing code at line 5: *volatile void *data;*

7.3.33. Unreachable code: UNR

Check to establish whether different code snippets (assignments, returns, conditionnal branches and function calls) are reached (Unreached code is referred to as "dead code"). Dead code is represented by means of a grey color coding on every check and an UNR check entry.

C++ example:

```

typedef enum {
  Intermediate,   End,   Wait,   Init
}  enumState;

// automatic stubs
int intermediate_state(int);
int random_int(void);

bool State (enumState stateval)
{
  int i;
  if (stateval == Init) return false;
  return true;
}

int main (void)
{
  int i;
  bool res_end;
  enumState inter;

  res_end = State(Init);
  if (res_end == false) {
    res_end = State(End);
    inter = (enumState)intermediate_state(0);
    if (res_end || inter == Wait) {             // Unreachable code for inter == Wait
      inter = End;
    }
    // use of i not initialized
    if (random_int()) {
      inter = (enumState)intermediate_state(i); // NIV ERROR: [non initialized variable]
      if (inter == Intermediate) {              // Unreachable code after runtime error
        inter = End;
      }
    }
  } else {
    i = 1;                                      // Unreachable code
39        inter = (enumState)intermediate_state(i);   // UNR check
40      }
41      return res_end;
42    }
43
```

Explanation:

The example illustrates three possible reasons why code might be unreachable, and hence be coloured grey:

1. At line 30 a conditionnal part of a conditionnal branch is always true and the other part never evaluated because of the standard definition of logical operator "||".
2. The piece of code after a red error is never evaluated by Polyspace Verifier. The call to the function and the following line after line 35 are considered to be lines of dead code. Correcting the red error and re-launching would allow the colour to be revised.
3. At line 27, the first branch is always evaluated to true (*if-{* part) and the other branch is never executed (*else* part at lines 41 to 42).

7.3.34. Values on assignment: VOA

Check to establish whether the range taken by variables on assignment. Theses checks are only available when the ***-voa*** option is used at launching time. Moreover, they are only available on scalar variables.
C++ Example:

```
static volatile int var_int = 1;
static volatile float volatile_float = 1;

#define MAX_ANA (9.999)
#define MIN_ANA (-10.0)
#define ZERO_ANA ((MAX_ANA - MIN_ANA)/2.0 - MAX_ANA)

float get_analogic (int);
bool   get_digit (int);

typedef enum {Red, Green, Orange, Black} VerifierColor;

typedef struct {
  float a;
  VerifierColor b;
  int c;
} Record;

int main(void)
{
  bool var_digit;
  Record var_rec;
  int i;
  float var_sensor;
  VerifierColor var_color = Green;                 // Currently no VOA on enum

  var_digit = 0;                                   // VOA: {[expr]=0}
  var_sensor = (float)(ZERO_ANA);                  // VOA: {[expr] <= FLT_MAX} and {FLT_MIN <= [expr]}
  for (i = 0 /* VOA:{[expr]=0} */ ; i < 8 ; i++) { // VOA: {1<=[expr]<=8}
    var_sensor = get_analogic(i);                  // VOA: currently not concise
    var_digit =  get_digit(i);                     // VOA: {0<=[expr]<=1}
  }

  // Float examples
  var_sensor = volatile_float;                     // VOA: currently not concise
  var_sensor = MAX_ANA;                            // VOA: {[expr]=9.9989}

  var_rec.a = var_sensor;                          // Curently no VOA on structures
  var_rec.b = var_color;
```

```
        var_rec.c = 5;
    }
```

Explanation:

Value on assignment are an informative checks than can only be green. They can be very helpfull to understand what PolySpace knows about index of arrays and variables. Thus, it is easier to statuate on orange checks.

7.3.35. Non Terminations: Calls and Loops

NTC and NTL are informative red checks.

- They are the only red checks which can be filtered out as shown below
- They don't stop the analysis
- As for other red checks, code found after them are grey (unreachable)
- These checks may only be red. There are no "orange" NTL or NTC checks.
- They can reveal a bug, or can simply just be informative

NTL	In a Non Terminating Loop, the break condition is never met. Here are some examples.
	
	while(1) { function_call(); } // informative NTL
	
	while(x>=0) {x++; } // where x is an unsigned int. This may reveal a bug?
	
	for(i=0; i<=10; i++) my_array[i] = 10; // where "int my_array[10];" applies. This red NTL reveals a bug in the array access, flagged in orange
	
	ptr = NULL; **for**(i=0; i<=100; i++) *ptr=0; // the first iteration of the loop is red, and therefore it is flagged as an NTL. The "i++" will be grey, because the first iteration crashed.

NTC	Suppose that a function calls f(), and that function call is flagged with a **red NTC check**. There could be five distinct explanations: 1. "f" contains a red error; 2. "f" contains an NTL ; 3. "f" contains an NTC; 4. "f" contains an orange which is context dependant; that is, it is either red or green. For this particular call, it makes the function "f" crash. 5. "f" is a mathematic function, such as sqrt, acos which has always an invalid input parameter Remember, additional information can be found when clicking on the NTC

Note that a `sqrt` check is only coloured if the input parameter is never valid. For instance, if the variable x may take any value between -5 and 5, then sqrt(x) has no colour.

The list of constraints which cannot be satisfied (found by clicking on the NTC check) represents the variables that cause the red error inside the function. The (potentially) long list of variables can help to understand the cause of the red NTC, as it shows each condition causing the NTC

- where the variable has a given value; and
- where the variable is not initialized. (Perhaps the variable is initialized outside the set of files under analysis?).

If a function is identified which is not expected to terminate (such as a loop or an exit procedure) then the -known-NTC function is an option. You will find all the NTCs and their consequences in the known-NTC facility in the Viewer, allowing you to filter them.

Related subjects :

7.3.35.1. Non Termination of Call: NTC
7.3.35.2. Non Termination of Loop: NTL

7.3.35.1. Non Termination of Call: NTC

Check to establish whether a procedure call returns. It is not the case when the procedure contains an endless loop or a certain error,
or if the procedure calls another procedure which does not terminate. In the latter instance, the status of this check is propagated to caller.
C++ example:

```
1
2      static  volatile int _x = 1;
3
4      void foo(int x)
5      {
6        int y = 1 / x;           // ZDV Warning: depends of the context
7        while(1) {               // NTL ERROR:  loop never terminates
8          if ( y != x) {
9            y = 1 / (y-x);      // ZDV Verified
10         }
11       }
12     }
13
14     void main(void) {
15
16       if (_x)
17         foo(0); // NTC ERROR: Zero DiVision (ZDV) in foo
18       if (_x)
19         foo(2); // NTC ERROR: Non Termination Loop (NTL) in foo
20     }
21
```

Explanation:
In this example, the function *foo* is called twice in *main* and neither of these 2 calls ever terminates:

1. The first never returns because a division by zero occurs at line 6 (bad argument value), and propagation of this error is propagated to caller at line 17.
2. The second never terminates because of an infinite loop (red NTL) at line 7. This error is propagated to caller at line 19.

As an inside, note that by using either the ***-context-sensitivity "foo"*** option or the ***-contex-sensitivity-auto*** option at launch time, it is possible for PolySpace to show explicitely that a ZDV error comes from the first call of *foo* in *main*.

Previous Back to table of contents Next

7.3.35.2. Non Termination of Loop: NTL

Check to establish whether a loop (for, do-while, while) terminates.
C++ example:

```

// prototypes fo functions
void send_data(double data);
void update_alpha(double *a);
static volatile double _acq =0.0;
static volatile int start_ = 0;
typedef void (*pfvoid)(void);

extern void launch (pfvoid);

void task(void)
{
  double acq, filtered_acq, alpha;

  // Init
  filtered_acq = 0.0;
  alpha = 0.85;

  while (1) {            // NTL ERROR: [non termination of loop]
    //  Acquisition
    acq = _acq;
    // Treatment
    filtered_acq = acq + (1.0 - alpha) * filtered_acq;
    // Action
    send_data(filtered_acq);
    update_alpha(&alpha);
  }
}

void rte_loop(void)
{
  int i;
  double twentyFloat[20];

  for (i = 0; i <= 20; i++) {  // NTL ERROR: propagation of OBAI ERROR
    twentyFloat[i] = 0.0;      // OBAI Warning: 20 verification with i in [0,19]
                               // and one ERROR with i = 20
  }
}
```

```
40
41    void main()
42    {
43      if (start_)
44        launch(task);
45
46      rte_loop();        // NTC ERROR: propagation of NTL error
47    }
```

Explanation:

In the example at line 19, the "continuation condition" is always true and the loop will never exit. Thus PolySpace will raise an error. In some case, the condition is not trivial and may depend on some program variables. Nevertheless Verifier is still able to analyse those cases.

On the other error at line 35, the red OBAI related to the 21th execution of the loop has been tranformed in an orange warning because of the 20 first verified executions.

7.4. Advanced results review

Related subjects :

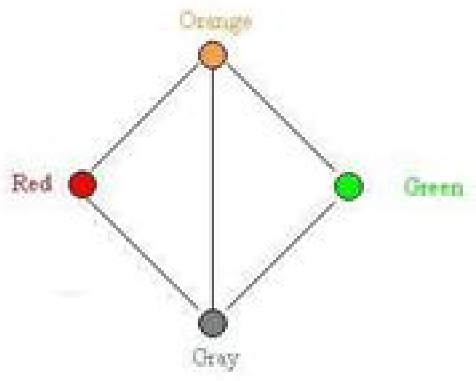
7.4.1. Red checks where grey checks were expected

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

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

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

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



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

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

- by an empty superset;
- by a nonempty super set, members of which may generate checks of any colour.

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

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

Consider the following “example”: `if (condition) then action_producing_a_red;`

After the "`if`" statement, the only way execution can continue is if the condition is false; otherwise a **red check** would be produced. Therefore, after this branch the condition is always false. For that reason, the code analysis continues, even with a specific error. Remember that this propagates values throughout your application. None of the execution paths leading to a runtime error will continue after the error and if the **red check** is a real problem rather than an approximation of a grey check, then the analysis will not be representative of how the code will behave when the red error has been addressed.

It is applicable on the current example:

```
int a[] = { 1,2,3,4,5,7,8,9,10 };
 void main(void)
 {
   int x=0;
   int tmp;
  if (a[5] > a[6])
    tmp = 1 /x; // RED ERROR [scalar division by zero] in grey code
}
```

7.4.2. Potential side effect of a red error

This section explains why when a red error has been found the analysis continues but some cautions need to be taken. Consider this piece of code:

```
int *global_ptr;

int variable_it_points_to;


void big_red(void)

{

int r;

int my_zero = 0;

if (condition==1)

      r = 1 / my_zero; // red ZDV

...

... // hundreds of lines

global_ptr = &variable_it_points_to;

other_function();

}
```

```
void other_function(void)

{

if (condition==1)

     *global_ptr = 12;

}
```

PolySpace works by propagating data sets representing ranges of possible values throughout the call tree, and throughout the functions in that call tree. Sometimes, PolySpace internally subdivides the functions for analysis, and the propagation of the data ranges need several iterations (or integration levels) to complete. That effect can be observed by examining the colour of the checks on completion of each of those levels. It can sometimes happen that:

- PolySpace will detect grey code which exists due to a terminal RTE which will not be flagged in red until a subsequent integration level.
- PolySpace flags a **NTC** in red with the content in grey. This red NTC is the result of an imprecision, and should be grey.

Suppose that an NTC is hard to understand at given integration level (level 4):

- If other **red checks** exist at level 4, fix them and restart the analysis
- Otherwise, look back through the results from each previous level to see whether other red errors can be located. If so, fix them and restart the analysis

8. Options description

This section describes all options available using PolySpace Desktop and PolySpace Verifier. All options, excepted multitasking options, are accessible through the two graphical user interfaces "`PolySpace launcher`" and "`PolySpace Desktop Launcher`".
They are also accessible using the associated batch command: `polyspace-cpp` and `polyspace-desktop-cpp`. In the following, it only refers to "`polyspace-cpp`" batch command.

Related subjects :

8.1. Sources/Includes

Related subjects :

8.1.1. -results-dir Results_Directory

This option specifies the directory in which Verifier will write the results of the analysis. Note that although relative directories may be specified, particular care should be taken with their use especially where the tool is to be launched remotely over a network, and/or where a project configuration file is to be copied using the "Save as" option.

Default:

Shell Script: The directory in which tool is launched.

From Graphical User Interface: C:\PolySpace_Results

Example Shell Script Entry:

```
polyspace-cpp -results-dir RESULTS ...
export RESULTS=results_`date +%d%B_%HH%M_%A`
polyspace-cpp -results-dir `pwd`/$RESULTS ...
```

8.1.2. -sources files or -sources-list-file file_name

```
-sources "file1[ file2[ ...]]" (Linux and Solaris)
or
-sources "file1[,file2[, ...]]" (windows, Linux and Solaris)
or
-sources-list-file file_name (not a graphical option)
```

List of source files to be analyzed, double-quoted and separated by commas. Note that UNIX standard wild cards are available to specify a number of files.
Note:
The specified files must have valid extensions: `*.(c|C|cc|cpp|CPP|cxx|CXX)`

Defaults:
`sources/*.(c|C|cc|cpp|CPP|cxx|CXX)`
Example Shell Script Entry under linux or solaris (*files are separated with a white space)*:
```
polyspace-cpp -sources "my_directory/*.cpp" ...
polyspace-cpp -sources "my_directory/file1.cc other_dir/file2.cpp" ...
```
Example Shell Script Entry under windows (*files are separated with a comma):*
```
polyspace-cpp -sources "my_directory/file1.cpp,other_dir/file2.cc" ...
```

Using `-sources-list-file`, each file *name* need to be given with an absolute path. Moreover, the syntax of the file is the following:

- One file by line.
- Each file name is given with its absolute path.

Note:
This option is only available in batch mode.

Example Shell Script Entry for -sources-list-file:
```
polyspace-cpp -sources-list-file "C:\Analysis\files.txt"
polyspace-cpp -sources-list-file "/home/poly/files.txt"
```

8.1.3. -I directory

This option is used to specify the name of a directory to be included when compiling C++ sources. Only one directory may be specified for each –I, but the option can be used multiple times.

Default:

- When no directory is specified using this option, the ./sources directory (if it exists) is automatically included
- If several include-dir are mentioned, the ./sources directory (if it exists), is implicitly added at the end of the "-I" list

Example Shell Script Entry-1:

```
polyspace-cpp -I /com1/inc -I /com1/sys/inc
```

is equivalent to

```
polyspace-cpp -I /com1/inc -I /com1/sys/inc -I ./sources
```

Example Shell Script Entry-2:

```
polyspace-cpp
```

is equivalent to

```
polyspace-cpp -I ./sources
```

8.2. General

This section collates all options relating to the identification of the analysis.

Related subjects :

8.2.1. -prog Session identifier

This option specifies the application name, using only the characters which are valid for Unix file names. This information is labelled in the GUI as the *Session Identifier*.

Default:

Shell Script:polyspace

GUI: New_Project

Example shell script entry:

```
polyspace-cpp -prog myApp ...
```

8.2.2. -date Date

This option specifies a date stamp for the analysis in dd/mm/yyyy format. This information
is labelled in the GUI as the *Date*. The GUI also allows alternative default date formats, via the Edit/Preferences window.

Default:

Day of launching the analysis

Example shell script entry:

```
polyspace-cpp -date "02/01/2002"...
```

8.2.3. -author Author

This option is used to specify the name of the author of the verification.

Default:

the name of the author is the result of the *whoami* command

Example shell script entry:

```
polyspace-cpp -author "John Tester"
```

8.2.4. -verif-version Version

Specifies the version identifier of the verification. This option can be used to identify different analyses. This information is identified in the GUI as the *Version*.

Default:

1.0.

Example shell script entry:

```
polyspace-cpp -verif-version 1.3 ...
```

8.2.5. -voa

When applied at launch time, this option enables the inspection of calculated domains for simple type assignments (scalar or float).
A new category of checks - named VOA - is generated on `"="` of some scalar assignments to give the ranges. VOA checks are not available for volatile variables.
Default:
 Disabled by default
Note:
 Depending on code optimisation, this check may not be present at all assignment locations
Example Shell Script Entry:

```
 polyspace-cpp -voa ...
```

8.2.6. -keep-all-files

When this option is set, all intermediate results and associated working files are retained. Consequently, it is possible to restart Verifier from the end of any complete pass (provided the source code remains entirely unchanged). If this option is not used, it is only possible to restart Verifier from scratch.

By default, intermediate results and associated working files are erased when they are no longer needed by the Verifier.

8.2.7. -continue-with-existing-host

When this option is set, the analysis will continue even if the system is under specified or its configuration is not as preferred by PolySpace. Verified system parameters include the amount of RAM, the amount of swap space, and the ratio of RAM to swap.

Default:

Verifier stops when the host configuration is incorrect or the system is under specified.

Example Shell Script Entry:

```
polyspace-cpp -continue-with-existing-host ...
```

8.2.8. -allow-unsupported-linux

This option specifies that PolySpace will be launched on an unsupported OS Linux distribution.
In such case a warning is displayed in he log file against possible incorrect behaviours:

```
**********************************************************
***                                                    ***
***                       WARNING                      ***
***                                                    ***
***   You are running PolySpace Verifier on an         ***
***   unsupported Linux distribution. It may lead      ***
***   to incorrect behaviour of the product. Please    ***
***   note that no support will be available for       ***
***   this operating system.                           ***
***                                                    ***
**********************************************************
```

Default:
Disable
Example Shell Script Entry:

```
polyspace-cpp –allow-unsupported-linux ...
```

8.3. Targets/Compilers

Related subjects :

8.3.1. -target TargetProcessorType

Specifies the target processor type.
This option informs Verifier of the size of fundamental data types and of the endianess of the target machine.
Possible values are:

sparc, m68k, powerpc, i386, c-167.

However, code which is to be run on other processor types can also be analysed if the data properties which are relevant to Verifier are common to one of the processor types listed.

Default:

sparc

Example shell script entry:

```
polyspace-cpp -target m68k ...
```

8.3.2. -OS-target OperatingSystemTarget

Specifies the operating system target for PolySpace stubs.
Possible values are '`solaris`', '`linux`', '`vxWorks`', '`visual`' and '`no-predefined-OS`'. This option allows PolySpace to determine which system definitions should be given to the preprocessor in order to analyze the included files properly. *`-OS-target`* `no-predefined-OS` may be used in conjunction with -include or/and -D to give all of the proper system preprocessor flags. Details of these may be found by executing the compiler for the project in verbose mode. They are also listed in "OS and target specifications".

Default:
`Solaris`

Note:
Only the '`linux`' include files are provided with Verifier (see the include folder in the installation directory). Projects developed for use with other operating systems may be analysed by using the corresponding include files for that OS. For instance, in order to analyse a `vxWorks` project it is necessary to use the option `-I <<path_to_the_vxWorks_include_folder>>`.

Example shell script entry:
```
   polyspace-cpp -OS-target no-predefined-OS \
            -D GCC_MAJOR=2 –include /complete_path/inc/gn.h
```

8.3.3. -D compiler-flag

This option is used to define macro compiler flags to be used during compilation phase.
Only one flag can be used with each –D as for compilers, but the option can be used several times as shown in the example below.
Default:
Some defines are applied by default, depending on your -OS-target option.
Example Shell Script Entry:

```
polyspace-cpp -D HAVE_MYLIB -D USE_COM1 ...
```

8.3.4. -U compiler-flag

This option is used to undefine a macro compiler flags
As for compilers, only one flag can be used with each –U, but the option can be used several times as shown in the example below.
Default:
Some undefines may be set by default, depending on your -OS-target option.
Example Shell Script Entry:

```
 polyspace-cpp -U HAVE_MYLIB -U USE_COM1 ...
```

8.3.5. -include file1[,file2[,...]]

This option is used to specify files to be included by each C++ file involved in the analysis.

Default:

No file is universally included by default, but directives such as "#include <*include_file.h*>" are acted upon.

Example Shell Script Entry:

```
 polyspace-cpp -include `pwd`/sources/a_file.h -include /inc/
inc_file.h ...
 polyspace-cpp -include /the_complete_path/my_defines.h ...
```

8.3.6. -post-preprocessing-command command

When this option is used, the specified script file or command is run after the pre-processing phase on each source file. The command should be designed to process the standard output from pre-processing and produce its results in accordance with that standard output.

Default:

No command.

Example Shell Script Entry – file name:

to remove the key word `interrupt` or `@near`, you can type the following command

```
  polyspace-c -post-preprocessing-command `pwd`/
remove_bad_keywords
```

where `remove_bad_keywords` is the following script :

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

Example Shell Command Entry:

This example performs the same function as that illustrated above, but specifies the command line directly:

```
  polyspace-c -post-preprocessing-command "sed s/@near//g"
```

8.3.7. -post-analysis-command <file_name> or "command"

When this option is used, the specified script file or command is executed once the analysis has completed.
The script or command is executed in the results directory of the analysis.
Execution occurs after the last part of the analysis. The last part of is determined by the –to option.
Note that depending of the architecture used, notably when using remote launcher, the script can be executed on the client side or the server side.

Default:
No command.

Example Shell Script Entry – file name:
This example shows how to send an email to tip the client side off that his analysis has been ended.
This example supposes that the `mailx` command is available on the machine. So the command looks like:

```
     polyspace-cpp -post-analysis-command `pwd`/end_email.sh
```

where `end_emails.sh` is the following script:

```
     #!/bin/sh
     echo ”analysis finished” | mailx –s ”PolySpace Analysis ended”
     “name@domain.com”
```

Example Shell Command Entry:
This example performs the same function as that illustrated above, but specifies the command line directly:

```
polyspace-cpp -post-analysis-command "mailx –s \”PolySpace Analysis ended\”
\“name@domain.com\”"
```

8.4. Compliance with standards

Related subjects :

8.4.1. -dos

This option must be used when the contents of the **include** or **source** directory comes from a DOS or Windows file system. It deals with upper/ lower case sensitivity and control characters issues. Concerned files

- header files: all include dir specified (-I option)
- source files: all sources files selected for the analysis (-sources option)

```
#include "..\mY_TEst.h"^M
#include "..\mY_other_FILE.H"^M
```

into

```
#include "../my_test.h"
#include "../my_other_file.h"
```

Default:
disabled by default
Example Shell Script Entry:
polyspace-cpp -I /usr/include -dos -I ./my_copied_include_dir -D test=1

8.4.2. Embedded Assembler

PolySpace stops the execution when detecting assembler code and displays an error message. It can continue the execution if it is requested by the user with the option –discard-asm.
PolySpace ignores the assembler code by assuming that the assembler code does not have any side effect on global variables. Delimiters for assembler code to ignore are given by the user with the options –asm-begin and –asm-end or can be recognized by PolySpace following C++ standard specified asm declarations: `__asm` and `__asm__`.

Related subjects :

8.4.2.1. -discard-asm
8.4.2.2. Pragmas asm

8.4.2.1. -discard-asm

This option instructs the PolySpace analysis to discard assembler code. If this option is used, the assembler code should be modelled in cpp.
This option is not compatible with -asm-begin and -asm-end options.

Default:
Embedded assembler is treated as an error.
Example Shell Script Entry:
polyspace-cpp -discard-asm ...

8.4.2.2. Pragmas asm

This option is used to allow compiler specific `asm` functions to be excluded from the analysis, with the offending code block delimited by two `#pragma` directives. Consider the following example:

```
#pragma asm_begin_1
int foo_1(void) { /* asm code to be ignored by PolySpace */ }
#pragma asm_end_1
#pragma asm_begin_2
void foo_2(void) { /* asm code to be ignored by PolySpace */ }
#pragma asm_end_2
```

Where "`asm_begin_1`" and "`asm_begin_2`" mark the beginning of `asm` sections which will be discarded and “`asm_end_1`” and "`asm_end_2`" mark the end of those sections.
Also refer to the -discard-asm option with regards to the following code:

```
asm int foo_1(void) { /* asm code to be ignored by PolySpace */ }
asm void foo_2(void) { /* asm code to be ignored by PolySpace */ }
```

Example Shell Script Entry:

```
 polyspace-cpp -discard-asm -asm-begin "asm_begin_1,asm_begin_2" -asm-end
"asm_end_1,asm_end_2"
```

8.4.3. -wchar-t-is-unsigned-long

This option modify the target model.
It forces the wchar_t type to be unsigned long.
Example Shell Script Entry:

```
 polyspace-cpp -wchar-t-is-unsigned-long ...
```

8.4.4. -size-t-is-unsigned-long

This option modify the target model.
It forces the size_t type to be unsigned long.
Example Shell Script Entry:

```
 polyspace-cpp -size-t-is-unsigned-long ...
```

8.4.5. -no-extern-C

Some functions may be declared inside an extern “C” { } bloc in some files and not in others. Then, their linkage is not the same and it causes a link error according to the ANSI standard. Using this option will make PolySpace to ignore this error.
This permissive option may not solve all the extern C linkage errors.
Example Shell Script Entry:
```
 polyspace-cpp -no-extern-C ...
```

8.4.6. -no-stl-stubs

PolySpace provide an efficient implementation of part of the Standard library (STL). This implementation may not be compatible with includes files of the applications. In that case some linking errors could arise.
With this option Verifier does not use his implementation of the STL.

Example Shell Script Entry:

```
polyspace-cpp -no-stl-stubs ...
```

8.4.7. -dialect DialectName

Specifies the dialect in which the code is written. Possible values are:
`default, iso, cfront2, cfront3, visual`, `visual6, visual7.0, visual7.1 and visual8.`
`visual6` activate dialect associated with code used for Microsoft Visual 6.0 compiler and `visual` activates dialect associated with Microsoft Visual 7.1 and subsequent.
If the dialect is `visual* (visual, visual6`, `visual7.0`, `visual7.1 and visual8`) -OS-target must be set to Visual.
If the dialect is `visual*` option -dos, -OS-target Visual and -discard-asm are set by default.
`visual8` dialect activates support for Visual 2005 .NET specific compiler. All Visual 2005 .NET given include files can compile with -no-stl-stubs option and without (recommended).

Default:

```
    default
```

Example Shell Script Entry:

```
    polyspace-cpp -dialect visual8 ...
```

8.4.8. -wchar-t-is

This option forces `wchar_t` to be treated as a keyword as per the C++ standard or as a `typedef` as with Microsoft Visual C++ 6.0/7.x dialects.
Possible values are '`keyword`' or '`typedef`':

- `typedef` is the default behaviour when using -dialect option associated to `visual6`, `visual7.0` and `visual7.1`.
- `keyword` is the default behaviour for all others dialects including `visual8`.

This option allows the default behaviour implied by the PolySpace dialect option to be overridden.
This option is equivalent to the Visual C++ `/Zc:wchar` and `/Zc:wchar-` options.

Default:

`default` (depends on -dialect value).

Example in sheel script:

```
polyspace-cpp –wchar-t-is typedef …
```

8.4.9. -for-loop-index-scope

This option changes the scope of the index variable declared within a for loop.
Example:

```
for (int index=0; ...){};
index++; // index variable is usable (out) or not (in) at this point
```

Possible values are '`in`' and '`out`':

- `out` is the default for the -dialect option associated with values `cfront2, crfront3, visual6, visual7` and `visual 7.1`.
- `in` is the default for all other dialects, including `visual8`.

The C++ ANSI standard specifies the index be treated as '`in`'.
This option allows the default behaviour implied by the PolySpace dialect option to be overridden.
This option is equivalent to the Visual C++ options `/Zc:forScope` and `Zc:forScope-`.
Default:

`default` (depends on –dialect value)

Example in sheel script:

```
polyspace-cpp –for-loop-index-scope in …
```

8.4.10. Visual specific options

Related subjects :

8.4.10.1. -import-dir directory

One directory to be included by #*import* directive. This option must be used with -OS-target visual or -dialect visual* (6, 7.0, 7.1 and 8). It gives the location of `*.tlh` files generated by a Visual Studio compiler when encounter #import directive on `*.tlb` files.
Example Shell Script Entry:
polyspace-cpp -dialect visual8 -import-dir /com1/inc ...

8.4.10.2. -ignore-pragma-pack

Visual C++ `#pragma` directives specify packing alignment for structure, union, and class members. These directives may be ignored to prevent link errors using option –ignore-pragma-pack. PolySpace will stop the execution and display an error message if this option is used in non visual mode or without dialect gnu (without `-OS-target visual` or `–dialect visual*`). See also “Link messages” section.

Example Shell Script Entry:

```
polyspace-cpp –dialect visual –ignore-pragma-pack ...
```

8.4.10.3. -pack-alignment-value value

Visual C++ /Zp option specifies the default packing alignment for a project. Option -pack-alignment-value transfers the default alignment value to PolySpace analysis.
The argument value must be: 1, 2, 4, 8, or 16. Analysis will stop the execution and display an error message with a bad value or if this option is used in non visual mode (-OS-target visual or -dialect visual* (6, 7.0 or 7.1)).
Default:

    8

Example Shell Script Entry:

```
polyspace-cpp –dialect visual –pack-alignment-value 4 ...
```

8.4.10.4. -support-FX-option-results

Visual C++ /FX option allows the partial translation of sources making use of managed extensions to Visual C++ sources without managed extensions. Theses extensions are currently not taken into account by PolySpace and can be considered as a limitation to analyse this kind of code.
Using /FX, the translated files are generated in place of the original ones in the project, but the names are changed from `foo.ext` to `foo.mrg.ext`.

Option – support-FX-option-results allows the analysis of a project containing translated sources obtained by compilation of a Visual project using the /FX Visual option. Managed files need to be located in same directory than original ones and PolySpace will analyses managed files instead of the original ones without intrusion, and will permit to remove part of limitations due to specific extensions.
PolySpace will stop the execution and display an error message if this option is used in non visual mode (-OS-target visual or -dialect visual* (6, 7.0 or 7.1)).

Example Shell Script Entry:

```
polyspace-cpp –dialect visual - support-FX-option-results
```

8.4.11. -ignore-constant-overflows

This option specifies that the analysis should be permissive with regards to overflowing computations on constants. Note that it deviates from the ANSI C standard.
For example,

```
char x = 0xff;
```

causes an overflow according to the standard, but if it is analysed using this option it becomes effectively the same as

```
char x = -1;
```

With this second example, a red overflow will result irrespective of the use of the option.

char x = (rnd?0xFF:0xFE);

Default:
`char x = 0xff;` causes an overflow
Example Shell Script Entry:

```
polyspace-cpp -ignore-constant-overflows ...
```

8.4.12. -allow-undef-variables

When this option is used, PolySpace will continue in case of linkage errors due to undefined global variables. For instance when this option is used, PolySpace will tolerate a variable always being declared as extern

Default:

Undefined variables causes PolySpace to stop.

Example Shell Script Entry:

```
 polyspace-cpp -allow-undef-variables ...
```

8.4.13. -allow-negative-operand-in-shift

This option allows a shift operation on a negative number.
According to the ANSI standard, such a shift operation on a negative number is illegal – for example,
-2 << 2
With this option in use, PolySpace considers the operation to be valid. In the previous example, the result would be *-2 << 2 = -8*

Default:
 A shift operation on a negative number causes a red error.
Example Shell Script Entry:

```
 polyspace-cpp -allow-negative-operand-in-shift ...
```

8.4.14. -Wall

Force the C++ compliance phase to print all warnings.
Default:
By default, only warnings about compliance across different files are printed.
Example Shell Script Entry:
polyspace-cpp -Wall ..

8.5. Inner settings

Related subjects :

8.5.1. -main sub_program_name

The option specifies the qualified name of the main subprogram when a visual –OS-target is selected. This procedure will be analyzed after class elaboration, and before tasks in case of a multitask application or in case of the -entry-points usage.

Possible values are:

`main, _tmain, wmain, _tWinMain, wWinMain, WinMain` and `DllMain`.

However, if the main subprogram does not exist and the option -main-generator is not set, PolySpace will stop the analysis with an error message.

Default:

```
main
```

Example Shell script entry:

```
polyspace-cpp -main WinMain –OS-target visual
```

8.5.2. Generate a main using a given class

Related subjects :

8.5.2.1. -class-analyzer

PolySpace C++ is a class analyzer. The user needs to know which part of his design he wants to analyze. The user has two alternatives:

1. If a main program exists in the set of files given to the PolySpace analysis, then the analysis continue with this main
2. Otherwise the user **MUST** specify one class name

PolySpace Verifier and Desktop have the same facility. You can choose or not to provide a main in your application, and select one class instead.
If *MyclassName* does not exist in the application, analysis stops also. All public and protected function members declared within the class, called within the code or not, will be analyzed separately and called by a generated main.
This generated main, is not code compliant but visible in the graphical user interface within `__polyspace_main.cpp` file. It also initializes all global variables to random (see Getting started section).
Example shell script entry:

```
polyspace-cpp –class-analyzer MyClass
polyspace-desktop-cpp –class-analyzer MyNamespace::MyClass
```

8.5.2.2. -class-only

This option can only be used with option `–class-analyzer` `MyClass`. If option `–class-analyzer` is not used, Analysis stops and displays an error message. With the option –class-only, only functions associated to `MyClass` are analyzed. All functions out of class scope are automatically stubbed even though they are defined in the source code.

Default:

disable

Example Shell Script Entry:

```
polyspace-cpp –class-analyzer MyClass –class-only...
```

8.5.2.3. -class-analyzer-calls

This option can only be used with option `–class-analyzer` `MyClass`. If option `–class-analyzer` is not used, Analysis stops and displays an error message.

- By `default`, all public and protected function members declared within the class, called within the code or not, will be analyzed separately and called by a generated main. We call in this case of eligible method or functions.
- If `unused` is specified, only functions not called by another eligible function are called.

Default:

`default` is used

Example Shell Script Entry:

```
polyspace-cpp –class-analyzer MyClass –class-analyzer-calls unused ...
```

8.5.2.4. -no-constructors-init-check

By default, PolySpace checks for member initialization just after object construction and initialization with -function-called-before-main when using –class-analyzer.

This option can only be used with option –class-analyzer. If option `–class-analyzer` is not used, analysis stops and displays an error message.

Without this option, in the generated main in `__polyspace_main.cpp` file, you will find some added code checks like on the simple example below using `–class-analyzer A` options:

```
class A {
      public: int i ; int *j ;
      A() : i(0), j(0) { ; }
A(int a) : i(a) { ; }
      };
```

In `__polyspace_main.cpp` after a call to the constructor(s) and function called before main:

```
{ /* check NIV/NIP section */
   check_NIV( __polyspace_this->i ); // Proven NIV check
   check_NIP( __polyspace_this->j ); // Unproven NIP check: j is not
initialized in one constructor
}
```

Using the option, **no more check** of members is made.

Default:

Check is made for member scalars, floats and pointer member variables.

Example Shell Script Entry:

```
polyspace-cpp –class-analyzer MyClass –no-constructors-init-check ...
```

8.5.3. -main-generator-calls

This option is used with the `-main-generator` option, to specify the functions to be called.
Note that this option is protected by a license.

Eligible functions:

Every function declared outside a class and defined in the source code to analyse, is considered as eligible when using the option.
The list of functions contains a list of short name (name without signature) separated by comas. If the name of a function from the list is associated to a function not defined in the source code, PolySpace stops and displays an error message. If the name of a function from the list is ambiguous, all the functions with the same short name are called. If a function from the list does not belong or is not eligible, PolySpace stops and displays an error message. This error message is put in the log file.

Default values:

- `none`: No function is called. This can be used with a multitasking application without main for instance.
- `unused (default)`: Call all functions not already called within the code. Inline functions will not be called by the generated main.
- `all`: all functions except inline will be called by the generated main.
- `custom`: Only functions present in the list are called from the main. Inline functions can be specified in the list and will be called by the generated main.

An `inline` (static or extern) function is not called by the generated main program with values `all` or `unused`. An `inline` function can only be called with custom value: `-main-generator-calls custom=my_inlined_func`.

Example:

```
     polyspace-cpp -main-generator -main-generator-calls custom=function_1,
function_2
```

8.5.4. General options for the generation of mains

Related subjects :

8.5.4.1. -function-called-before-main

This option is used with the `main generator` options –class-analyzer and –main-generator-calls options to specify a function which will be called before all selected functions in the main.

Eligible functions:

Every function or method defined in the source code to analyse is considered as eligible when using the option.

If the given name is ambiguous or is associated to a function not defined in the source code, PolySpace stops and displays an error message. This error message is put in the log file.

Example:

```
     polyspace-cpp -main-generator-calls unused –function-called-before-
main MyFunction …
```

8.5.4.2. -main-generator-writes-variables

This option is used with the main generator options –class-analyzer and –main-generator-calls to dictate how the generated main will initialize global variables.

Settings available:

- *`uninit`*: main generator writes random on not initialized global variables.
- *`none`*: no global variable will be written by the main.
- *`public`*: every variable except static and const variables are assigned a "random" value, representing the full range of possible values
- *`all:`* every variable is assigned a "random" value, representing the full range of possible values
- *`custom`*: only variables present in the list are assigned a "random" value, representing the full range of possible values

Example

```
     polyspace-cpp –class-analyzer MyClass -main-generator-writes-variables
uninit
     polyspace-cpp -main-generator -main-generator-writes-variables
custom=variable_a,variable_b
```

8.5.5. -no-automatic-stubbing

By default, PolySpace automatically stubs all functions. When this option is used, the list of functions to be stubbed is displayed and the analysis is stopped.

Benefits:

This option may be used when:

- The entire code is to be provided, which may be the case when analyzing a large piece of code. When the analysis stops, it means the code is not complete.
- Manual stubbing is preferred to improve the selectivity and speed of an analysis.

Default:

All functions are stubbed automatically

Previous Back to table of contents Next

8.5.6. -ignore-float-rounding

Without this option, PolySpace rounds floats according to the IEEE 754 standard: simple precision on 32-bits targets and double precision on target which define double as 64-bits. With the option, exact computation is performed.
Example:

```

void ifr(float f)
{
 double a = 1.27;
  if ((double)1.27F == a) {
   assert (1);
    f = 1.0F * f;
     // reached when -ignore-float-rounding is used or not
  }
  else {
    assert (1);
    f = 1.0F * f;
    // reached when compiled under Visual and when -ignore-float-rounding is not used
  }
 }
```

Using this option can lead to different results compared to the "real life" (compiler and target dependent): Some paths will be reachable or not for PolySpace while they are not (or are) depending of the compiler and target. So it can potentially give approximate results (green should be unproven). This option has an impact on OVFL/UVFL checks on floats.

However, this option allows reducing the number of unproven checks because of the “delta” approximation.
For example:

- FLT_MAX (with option set) = 3.40282347e+38F
- FLT_MAX (following IEEE 754 standard) = 3.40282347e+38F ± Δ

```

void ifr(float f)
{
 double a = 1.27;
  if ((double)1.27F == a) {
    assert (1);
    f = 1.0F * f;  // Overflow never occurs because f <= FLT_MAX.
                   // reached when -ignore-float-rounding is used
  }
 else {
  assert (1);
  f = 1.0F * f; // OVFL could occur when f = (FLT_MAX + Δ)
                // reached when -ignore-float-rounding is not used
 }
}
```

Default:

IEEE 754 rounding under 32 bits and 64 bits.

Example Shell Script Entry:

```
polyspace-cpp -ignore-float-rounding ...
```

8.5.7. -detect-unsigned-overflows

When this option is selected, PolySpace becomes more pedantic than the ANSI standard requires, with regards overflowing computations on unsigned. Consider the examples below, which apply when the option is in use.

Example 1:

```
unsigned char x;
x = 255;
x = x+1; //causes an overflow according to this option.
```

Without this option in place, example above would generate no error.

```
unsigned char x;
x = 255;
x = x+1; // turns x into 0 (wrap around).
```

Example 2:

```
unsigned char x, y=1;
x = ~y; // causes an overflow because of type promotion
```

Default:
disable
Example Shell Script Entry:
`polyspace-cpp -detect-unsigned-overflows ...`

8.5.8. -extra-flags option-extra-flag

This option specifies an expert option to be added to the analyzer. Each word of the option (even the parameters) must be preceded by *-extra-flags*.
These flags will be given to you by PolySpace Support as necessary for your analyses.
Default:
No extra flags.
Example Shell Script Entry:
```
polyspace-cpp -extra-flags -param1 -extra-flags -param2
```

8.5.9. -cpp-extra-flags flag

It specifies an expert option to be added to a PolySpace C++ analysis. Each word of the option (even the parameters) must be preceded by *-cpp-extra-flags*.
These flags will be given to you by PolySpace support as necessary.
Default:
no extra flags.
Example Shell Script Entry:
```
      polyspace-cpp -cpp-extra-flags -Wall
```

8.6. Precision/Scaling

Related subjects :

8.6.1. -quick

Very fast mode for PolySpace C++. This option is exclusive with -O(0-3), -from and -to *verification-phase* options.

Benefits:

This option allows the user to have results very quickly. He will then focus on red and grey errors only, as oranges are unreadable using this option. Up to 25 times faster than classical analysis using a mix of O(precision level) and integration level.

Limitations:

- No NTL or NTC are displayed (non termination of loop/call)
- The global variable dictionary is not available
- No check is performed on floats
- The call tree is partially available but navigation is not possible
- Focus on red and grey only and do not look at oranges.

Example shell entry:

```
polyspace-cpp -quick
```

8.6.2. -O(0-3)

This option specifies the precision level to be used. It provides higher selectivity in exchange for more analysis time, therefore making results review more efficient and hence making bugs in the code easier to isolate. It does so by specifying the algorithms used to model the program state space during analysis.

It is recommended that analyses should begin with the -quick option. Red errors and grey code can then be addressed before re-launching Verifier using this option, applying a precision level as described below.

Benefits

- A higher precision level contributes to a higher selectivity rate, making results review more efficient and hence making bugs in the code easier to isolate.
- A higher precision level also means higher analysis time:
  - -O0 corresponds to static interval analysis.
  - -O1 corresponds to complex polyhedron model of domain values.
  - -O2 corresponds to more complex algorithms to closely model domain values (a mixed approach with integer lattices and complex polyhedrons).
  - -O3 is only suitable for code smaller than 1000 lines of code. For such codes, the resulting selectivity might reach high values such as 98%, resulting in a very long analysis time, such as an hour per 1000 lines of code.

Default:
-O2
Example Shell Script Entry:
```
polyspace-cpp -O1 -to pass4 ...
```

8.6.3. -from verification-phase

This option specifies the verification phase to start from. It can only be used on an existing analysis, possibly to elaborate on the results that you have already obtained.
For example, if an analysis has been completed `-to` `pass1`, PolySpace can be restarted *`-from`* `pass1` and hence save on analysis time.
The option is usually used in an analysis after one run with the -to option, although it can also be used to recover after power failure.
Possible values are as described in the -to *`verification-phase`* section, with the addition of the *`scratch`* option.
Notes:

- Unless the *`scratch`* option is used, this option can be used only if the previous analysis was launched using the option *`-keep-all-files`* .
- This option cannot be used if you modify the source code between two analyses.

Default:
From `scratch`
Example Shell Script Entry:
```
polyspace-cpp -from cpp-to-il ...
```

8.6.4. -to verification-phase

Specifies the verification phase after which PolySpace will stop analysis.

Benefits:

This option allows you to have a higher selectivity, and therefore to find more bugs within the code.

- A higher integration level contributes to a higher selectivity rate, leading to "finding more bugs" with a given code.
- A higher integration level also means higher analysis time.

Possible values:

- `cpp-compliance` (Reaches the compilation phase)
- `cpp-normalize` (Reaches the normalization phase)
- `cpp-link` (Reaches the link phase)
- `cpp-to-il` (Reaches the transformation to intermediate language)
- `pass0` *or* CDFA *or* "Control and Data Flow Analysis"
- `pass1` *or* "Software Safety Analysis level 1"
- `pass2` *or* "Software Safety Analysis level 2"
- `pass3` *or* "Software Safety Analysis level 3"
- `pass4` *or* "Software Safety Analysis level 4"
- `other` (stop analysis after level 20)

Note:

If you use *-to other* then PolySpace will continue until you stop it manually (via *"PolySpace Install Directory"/bin/kill-rte-kernel "Results directory"/"log file name"*) or stops until it has reached pass20.

Default:

pass4

Example Shell Script Entry:

```
polyspace-cpp -to "Software Safety Analysis level 3"...
polyspace-cpp -to pass0 ...
```

8.6.5. -path-sensitivity-delta number

This option is used to improve interprocedural analysis precision within a particular pass (see -to *pass1*, *pass2*, *pass3* or *pass4*). The propagation of information within procedures is done earlier than usual when this option is specified. That results in improved selectivity and a longer analysis time.

Consider two analyses, one with this option set to 1 (with), and one without this option (without)

- a level 1 analysis in (with) (pass1) will provide results equivalent to level 1 or 2 in the (without) analysis
- a level 1 analysis in (with) can finish x times more than a cumulated level 1+2 analysis from (without). "x" might be exponential.
- the same applies to level 2 in (with) equivalent to level 3 or 4 in (without), with potentially exponential analysis time for (with).

Gains using the option

(+) highest selectivity obtained in level 2. No need to wait until level 4
(-) This parameter increases exponentially the analysis time and might be even bigger than a cumulated analysis in level 1+2+3+4
(-) This option can only be used with less than 1000 lines of code.

Default:

0

Example Shell Script Entry:

```
polyspace-cpp -path-sensitivity-delta 1
```

8.6.6. -context-sensitivity "proc1[,proc2[,...]]"

This option allows the precise analysis of a procedure with regards to the discrete calls to it in the analysed code.
Each check inside the procedure is split into several sub-checks depending on the context of call. It has same effects than inlining a function but without duplicating as clones (see –inline).

Therefore if a check is red for one call to the procedure and green for another, both colours will be revealed (e.g. by highlighting a special button in PolySpace Viewer).
Note: This option is especially useful when a run time error has been detected in a function and it is called from a multitude of places.

8.6.7. -context-sensitivity-auto

This option is similar to the -context-sensitivity option, except that PolySpace automatically chooses the procedures to be considered.
Usually, the ten functions which are the most called are automatically selected.

8.6.8. -respect-types-in-globals

This is a scaling option, designed to help process complex code. When it is applied, PolySpace assumes that global variables not declared as containing pointers are never used for holding pointer values. This option should only be used with Type-safe C/C++ code, when it does not cause a loss of precision. See also -respect-types-in-fields.

In the following example, we will lose precision using option `–respect-types-in-globals` option:

```
int x;
void t1(void) {
  int y;
  int *tmp = &x;
  *tmp = (int)&y;
  y=0;
  *(int*)x = 1;     // x contains address of y
  assert (y == 0);  // green with the option
}
```

PolySpace will not take care that x contains the address of y resulting a green assert.

Default:

PolySpace assumes that global variables may contain pointer values.

Example Shell Script Entry:

```
polyspace-cpp -respect-types-in-globals ...
```

8.6.9. -k-limiting number

This is a scaling option to limits the depth of analysis into nested structures during pointer analysis (see tuning parameters).

Default:

There is no fixed limit.

Example Shell Script Entry:

```
polyspace-cpp -k-limiting 1 ...
```

In the usage above, analysis will be precise to only one level of nesting.

8.6.10. -respect-types-in-fields

This is a scaling option, designed to help process complex code. When it is applied, PolySpace assumes that structure fields not declared as containing pointers are never used for holding pointer values. This option should only be used with Type-safe C/C++ code, when it does not cause a loss of precision. See also -respect-types-in-globals .

In the following example, we will lose precision using option `–respect-types-in-fields` option:

```
struct {
  unsigned x;
  int f1;
  int *z[2];
} S1;

void funct2(void) {
  int *tmp;
  int y;
  ((int**)&S1)[0] = &y; /* S1.x points on y */
  tmp = (int*)S1.x;
  y=0;
  *tmp = 1;             /* write 1 into y */
  assert(y==0);
}
```

PolySpace will not take care that `S1.x` contains the address of y resulting a green assert.

Default:

PolySpace assumes that structure fields may contain pointer values.

Example Shell Script Entry:

```
polyspace-cpp -respect-types-in-fields ...
```

8.6.11. -inline "proc1[,proc2[,...]]"

A scaling option that creates a clone of each specified procedure for each call to it. Cloned procedures follow a naming convention viz:
`procedure1_pst_cloned_nb`, where nb is a unique number giving the total number of cloned procedures.

Such an inlining allows the number of aliases in a given procedure to be reduced, and may also improve precision.
It is some times recommended to inline standard functions permitting to make copy or set amount of memory, such as “`memset`”, “`strcpy`”, “`strncpy`”, “`memcpy`”, etc.
It can permit to find in an easy way run time errors (NTC for instance) which relate in this case the copy or set of a big structure in a smaller one.
Limitations:

- Extensive use of this option may duplicate too much code and may lead to other scaling problems. Carefully choose procedures to inline.
- This option should be used in response to the inlining hints provided by the alias analysis (log file some times can give such kind of information).
- This option should not be used on `main`, task entry points and critical section entry points.
- When using this option with a method of a class, all overload of the method will apply to the inline.

Example Shell Script Entry:

```
polyspace-cpp –inline “myclass::myfunc” …
```

8.6.12. Tuning precision and scaling parameters

- **Precision versus time of analysis**

There is a compromise to be made to balance the time required to obtain results, and the precision of those results. Consequently, launching PolySpace with the following options will allow the time taken for analysis to be reduced but will compromise the precision of the results. It is suggested that the parameters should be used in the sequence shown – that is, if the first suggestion does not increase the speed of analysis sufficiently then introduce the second, and so on.

- switch from -O2 to a lower precision;
- set the -respect-types-in-globals and -respect-types-in-fields options;
- set the -k-limiting option to 2, then 1, or 0;
- stub manually missing functions which write into their arguments.

- **Precision versus code size**

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

For instance, in a relatively small application, PolySpace Verifier might retain very detailed information about the data at a particular point in the code, so that for example the variable VAR can take the values { -2; 1; 2; 10; 15; 16; 17; 25 }. If VAR is used to divide, the division is green (because 0 is not a possible value). If the program being analyzed is large, PolySpace Verifier would simplify the internal data representation by using a less precise approximation, such as [-2; 2] U {10} U [15 ; 17] U {25} . Here, the same division appears as an orange check.

If the complexity of the internal data becomes even greater later in the analysis, PolySpace might further simplify the VAR range to (say) [-2; 20].

This phenomenon leads to the increase or the number of orange warnings when the size of the program becomes large.

Note that the amount of simplification applied to the data representations also depends on the required precision level (O0, O2), PolySpace Verifier will adjust the level of simplification, viz.:

-O0 and –quick: shorter computation time. You only need to focus on red and grey checks.

-O2: less orange warnings.

-O3: less orange warnings and bigger computation time.

8.7. Multitasking (PolySpace Server only)

Concurrency options are not compatible with -main-generator options.

Related subjects :

8.7.1. -entry-points task1[,task2[,...]]

This option is used to specify the tasks/entry points to be analysed by the PolySpace Server, using a comma-separated list with no spaces.
These entry points must not take parameters. If the task entry points are functions with parameters they should be encapsulated in functions with no parameters, with parameters passed through global variables instead.
Format:

- All tasks must have the prototype "`void` *`any_name`*`()`".
- It is possible to declare a member function as an entry point of an analysis, only and only if the function is declared “`static void task_name()`”.

Example shell script Entry:
```
polyspace-cpp -entry-points class::task_name,taskname,proc1,proc2
```

8.7.2. Critical sections

```
-critical-section-begin "proc1:cs1[,proc2:cs2]"
```
and
```
-critical-section-end "proc3:cs1[,proc4:cs2]"
```

These options specify the procedures beginning and ending critical sections, respectively. Each uses a list enclosed within double speech marks, with list entries separated by commas, and no spaces. Entries in the lists take the form of the procedure name followed by the name of the critical section, with a colon separating them.
These critical sections can be used to model protection of shared resources, or to model interruption enabling and disabling.
Limitation:

- Name of procedure accept only `void any_name()` as prototype.
- The beginning and the end of the critical section need to be defined in same block of code.

Default:
No critical sections.
Example Shell Script Entry:

```
polyspace-cpp \
-critical-section-begin "start_my_semaphore:cs" \
-critical-section-end "end_my_semaphore:cs"
```

8.7.3. -temporal-exclusions-file file_name

This option specifies the name of a file. That file lists the sets of tasks which never execute at the same time (temporal exclusion).
The format of this file is:

- one line for each group of temporally excluded tasks,
- on each line, tasks are separated by spaces.

Default:
no temporal exclusions.
Example of a task specification file:
File named 'exclusions' (say) in the 'sources' directory and containing:

```
      task1_group1 task2_group1
      task1_group2 task2_group2 task3_group2
```

Example Shell Script Entry:

```
polyspace-cpp -temporal-exclusions-file sources/exclusions \
                    -entry-points task1_group1,task2_group1,task1_group2,\
                    task2_group2,task3_group2 ...
```

8.8. Specific batch options

Related subjects :

8.8.1. -server server_name_or_ip[:port_number]

Using `polyspace-remote[-desktop]-[cpp] [–server [name or IP address][:<port number>]]` allows to send analysis to a specific or referenced PolySpace Queue manager server.
Note that If the option `–server` is not specified, the default server referenced in the `PolySpace-Launcher.prf` configuration file will be used as server.
When a `–server` option is associated to the batch launching command, the name or IP address and a port number need to be specified. If the port number does not exist, the 12427 value will be used by default.
Note also that polyspace-remote- accepts all other options.
Option Example Shell Script Entry:

```
polyspace-remote-desktop-cpp –server 192.168.1.124:12400 …
polyspace-remote-cpp …
polyspace-remote-cpp –server Bergeron …
```

8.8.2. -h[elp]

It displays on screen a textual help including a short description of all options.

8.8.3. -v |-version

Display the PolySpace version number.
Example Shell Script Entry:

```
      polyspace-cpp -v
```

Which will show a result similar to:

```
 PolySpace r2007a+
  Copyright (c) 1999-2007 PolySpace Technologies
```

8.8.4. -sources-list-file file_name

This option is only available in batch mode. The *`file_name`* file needs to be given with an absolute path and its syntax is the following:

- One file per line.
- Each file name includes absolute path location.

Example shell script Entry:

```
polyspace-cpp -sources-list-file "C:\Analysis\files.txt"
polyspace-cpp -sources-list-file "/home/poly/files.txt"
```

9. Appendix

Related subjects :

9.1. Glossary

Analysis	In order to use a PolySpace tool, the code is prepared and an analysis is launched which is turn produces results for review.
Atomic	In computer programming, atomic describes a unitary action or object that is essentially indivisible, unchangeable, whole, and irreducible.
Atomicity	In a transaction involving two or more discrete pieces of information, either all of the pieces are committed or none are.
Batch mode	Execution of PolySpace Verifier from the command line, rather than via the launcher GUI.
Category	One of four types of orange check: *potential bug, inconclusive check, data set issue* and *basic imprecision*
Certain error	See red error
Check	Test performed by PolySpace during analysis, coloured red, orange, green or grey in the viewer
Dead code	Code which is inaccessible at execution time under all circumstances, due to the logic of the software executed before it.
Development Process	Development process used within a company to progress through the software development lifecycle.
Green check	Check found to be confirmed as error free
Grey code	Dead code
Imprecision	Approximations made during PolySpace analysis, so that data values possible at execution time are represented by supersets including those values
mcpu	Micro Controller/Processor Unit
Orange warning	Check found to represent a possible error, which may be revealed on further investigation.
PolySpace Approach	The manner of use of PolySpace to achieve a particular goal, with reference to a collection of techniques and guiding principles.
Precision	An analysis which includes few inconclusive orange checks is said to be precise
Progress text	Output from PolySpace during analysis to indicate what proportion of the analysis has been completed. Could be considered as a “textual progress bar”.
Red error	Check found to represent a definite error
Review	Inspection of the results produced by a PolySpace analysis, using the Viewer.

Scaling option	Option applied when an application submitted to PolySpace Verifier proves to be bigger or more complex than is practical.
Selectivity	The ratio of (**green** + **grey** + **red**) / (total amount of checks)
Unreached code	Dead code

9.2. Abstract semantic

Static Verification is a broad term, and is applicable to any tool which derives dynamic properties of a program without actually executing it. Static Verification differs significantly from other techniques, such as run-time debugging, in that the analysis it provides is not based on a given test case or set of test cases. The dynamic properties obtained in the PolySpace analysis are true for all executions of the software.

Most Static Verification tools only provide an analysis of the complexity of the software, in a search for constructs which may be potentially dangerous. PolySpace provides deep-level analysis identifying almost all run-time errors and possible access conflicts on global shared data.

The idea is to use an approximation of the software under analysis, using safe and representative approximations of software operations and data.

An example is given below:

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

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

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

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

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

Exhaustiveness

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