PolySpace
TECHNOLOGIES

PolySpace™ for C++ Getting Started R2007a+

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. General Requirements 5
1.1. Computer Configuration 5
1.2. Structure of this document 5
2. Step 1: PolySpace Client - Setting up and launching an analysis on a single class 6
2.1. Analysis prerequisites 6
2.2. Setting up a PolySpace Client analysis 7
2.2.1. Select results directory 9
2.2.2. Select the files of the analysis 10
2.2.3. Select the class to analyse 12
2.3. PolySpace Client: running the analysis 13
2.3.1. Parsing errors during preliminary PolySpace analysis stages 15
2.3.2. Progression of the analysis 21
2.3.3. End of the analysis 22
3. Step 2: PolySpace Class Analyzer 23
3.1. Sources to be analysed 23
3.2. Architecture of the generated main 23
3.3. Log file 25
3.4. Characteristics of a class and messages of the log file .26
3.5. Behaviour of global variables and members 27
3.6. Methods and classes specificities 29

4. Step 3: PolySpace Viewer - Exploration of results 31
4.1. Modes of operation 31
4.2. Download results 32
4.3. Analysing PolySpace results in expert mode ("training.cpp") 34
4.3.1. Procedural entities view 36
4.3.2. Colours in the Source code view 41
4.3.3. More examples of diagnostic 41
4.3.4. Advanced results exploration 42
4.3.5. C++ specific checks 44
4.3.6. Miscellaneous 45
4.4. Methodological assistant 45
4.4.1. Assistant dashboard 47
4.4.2. Choose a methodological assistant 50
4.5. Report Generation 51
5. Launch PolySpace Remotely 55
5.1. Launching an analysis 55
5.2. Management of PolySpace analysis in remote: the PolySpace Spooler 57
5.3. Batch commands 59
5.4. Share analysis between account 61
6. Summary 63

Typographical conventions:

➪ The "➤" symbol indicates an action which must be performed by the user.
➪ `<PolySpaceInstallDir>` stands for the directory/folder name where the PolySpace products were installed.
➪ The "`Courier New`" font is used for mentioning data seen on the screen of the computer.

1. General Requirements

1.1. Computer Configuration

Please refer to PolySpace installation manual for the minimum hardware requirements.
The timing is the following:

- The installation of PolySpace products takes around 5 minutes (see the complete installation guide as available from the PolySpace installation CD-ROM in *\Docs\Install\PolySpace_Install_Guide.pdf*).
- The first step of this tutorial takes about 20 minutes.
- The third step of this tutorial takes about 5 minutes.
- The fourth step of this tutorial takes about 10 minutes.

1.2. 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 Spooler
Shortcut
2 KB

PolySpace Viewer
Shortcut
2 KB

This Getting Started will focus on the following exercises using PolySpace Client, PolySpace Viewer and PolySpace Remote Launcher:

- 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 more thoroughly the capabilities of the class analyzer.
- In Step 3 we will review the results obtained during Step 1 by using PolySpace Viewer
- In Step 4, we will send an analysis remotely to a server.

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 the `MathUtils` class in "`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, PolySpace Client will generate one automatically. By default, the main will build an instance of the class using 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.

2.1. Analysis prerequisites

Any analysis requires the following:

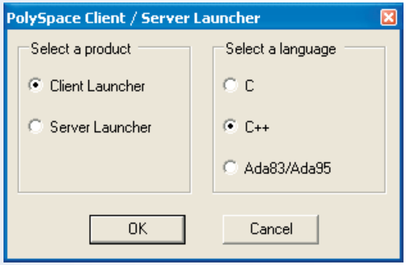
- PolySpace products and their related license files correctly installed;
- Source code files (in this case "`training.cpp`") and all header files that it may directly or indirectly include. For this tutorial we will see later that we need three header files, "`training.h`", "`zz_utils.h`" and "`math.h`" 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. Setting up a PolySpace Client analysis

➤ Double-click on the PolySpace Launcher icon:

PolySpace Launcher
Shortcut
2 KB

A window appears proposing to choose the product to be used for the analysis and the language of the file to be analyzed:


PolySpace Client / Server Launcher

Select a product
[✓ Client Launcher]
Server Launcher

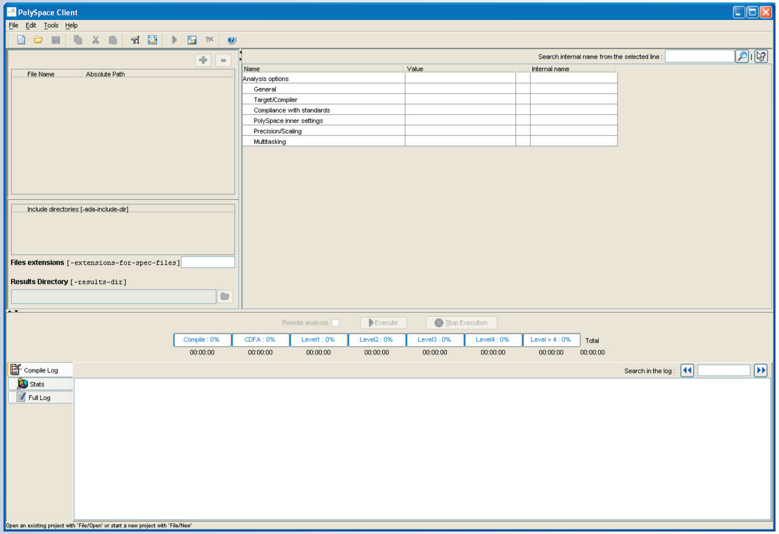
Select a language
C
[✓ C++]
Ada83/Ada95

If PolySpace is not installed for some languages, these choices of languages will be grayed out.

➤ Select "`Client Launcher`", language "`C++`" and then, click on OK.

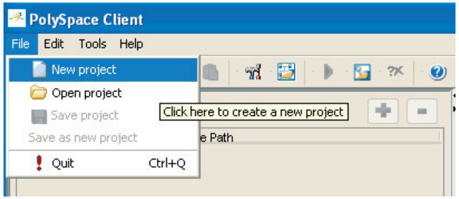
Step 1: PolySpace Client - Setting up and launching an analysis on a single class

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



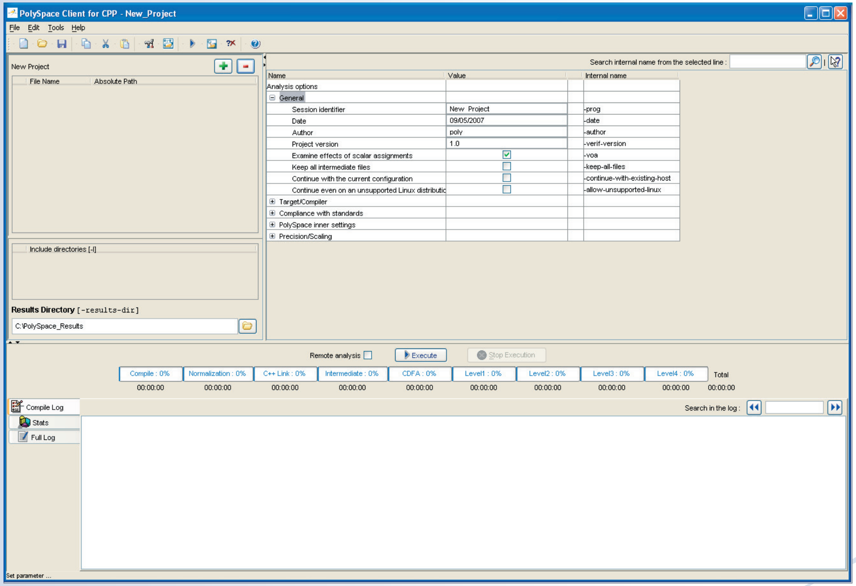
Open an existing project with 'File/Open' or start a new project with 'File/New'

➤ 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;
- The bottom area allows following the execution and progress of the analysis.


PolySpace Client for CPP - New_Project

Name	Value	Internal name
Analysis options		
⊟ General		
Session identifier	New Project	-prog
Date	09/05/2007	-date
Author	poly	-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 distributic	☐	-allow-unsupported-linux
⊞ Target/Compiler		
⊞ Compliance with standards		
⊞ PolySpace inner settings		
⊞ Precision/Scaling		

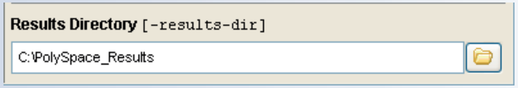
Results Directory [-results-dir]

C:\PolySpace_Results

Compile : 0%	Normalization : 0%	C++ Link : 0%	Intermediate : 0%	CDFA : 0%	Level1 : 0%	Level2 : 0%	Level3 : 0%	Level4 : 0%	Total
00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00

2.2.1. Select results directory

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

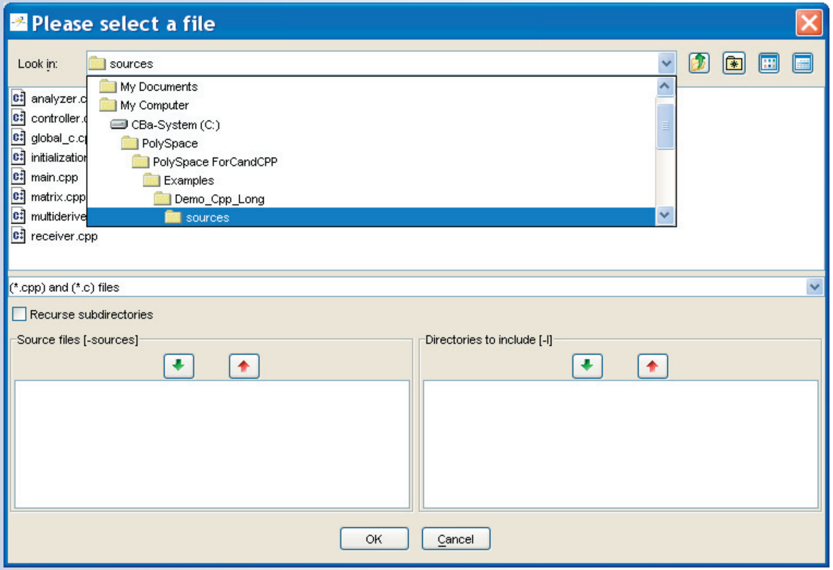

Results Directory [-results-dir]

C:\PolySpace_Results

This directory is the one where PolySpace Client will store the results of the analysis. In this Getting Started, we will choose the default directory: "`C:\PolySpace_Results`".

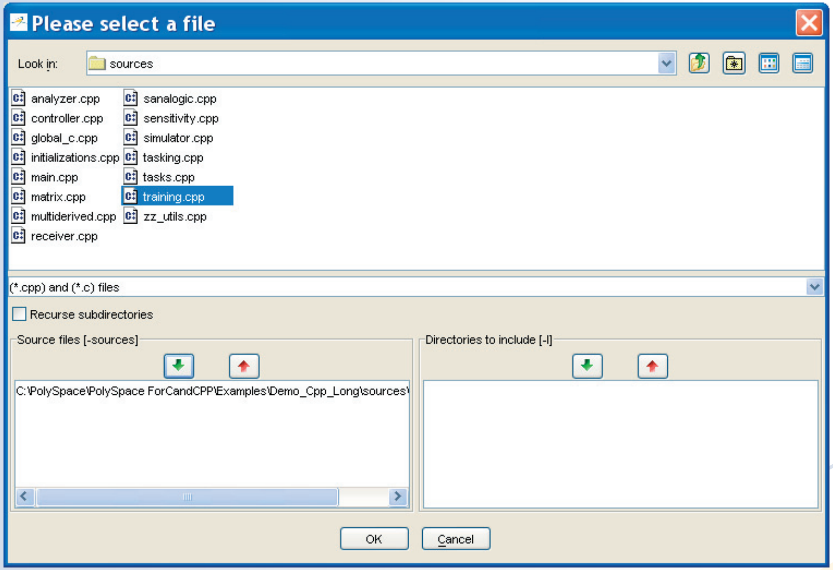
2.2.2. Select the files of the analysis

➤ 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.
The default `<PolySpaceInstallDir>` is `C:\PolySpace\PolySpaceForCandCPP`.

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



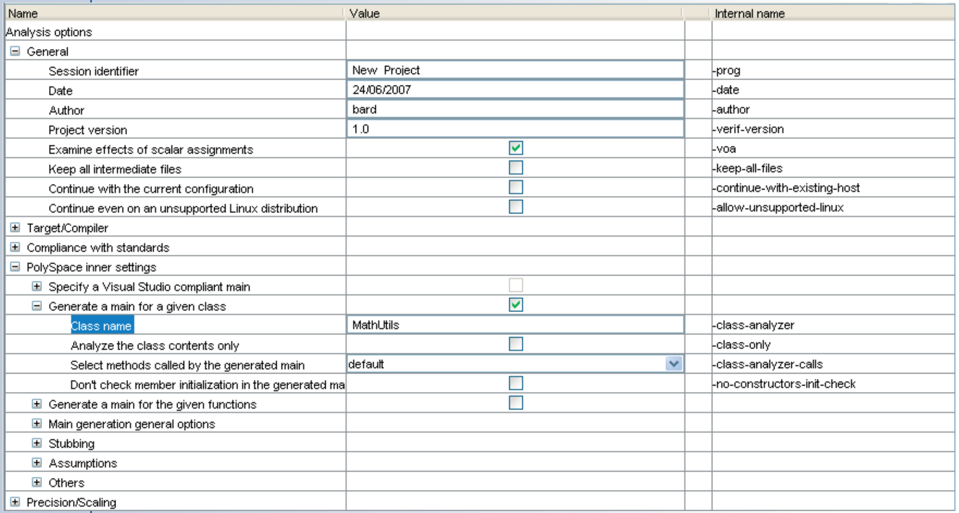
➤ Click on OK to go back to the "`PolySpace Client for CPP - New_Project`" window.

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

2.2.3. Select the class to analyse

➤ Now, click on ⊞ PolySpace inner settings 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 giving the name of the class to analyse (see also step 2). Type "`MathUtils`" in the "Value" column as shown in the figure below. If 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	24/06/2007	-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 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 generated ma	☐	-no-constructors-init-check
⊞ Generate a main for the given functions	☐	
⊞ Main generation general options		
⊞ Stubbing		
⊞ Assumptions		
⊞ Others		
⊞ Precision/Scaling		

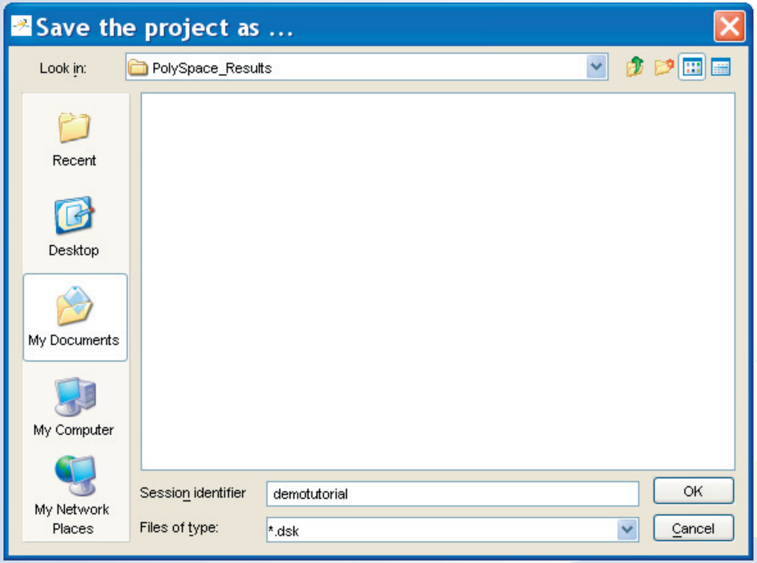
➤ It is also recommended to select the `-voa` option. If this option is selected, PolySpace will give you some information on the possible range of values for each scalar assignment, thus helping understanding the results of the analysis.

Note: When you want to analyse a class by itself, the `-class-only` option can been checked. It means that, even if you add other classes and function members definitions, PolySpace will stub them. This option accelerates analysis and allows checking robustness issues for the class. In this tutorial, it is not necessary to check this option: the "`MathUtils`" class does not depend on other classes.

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`".

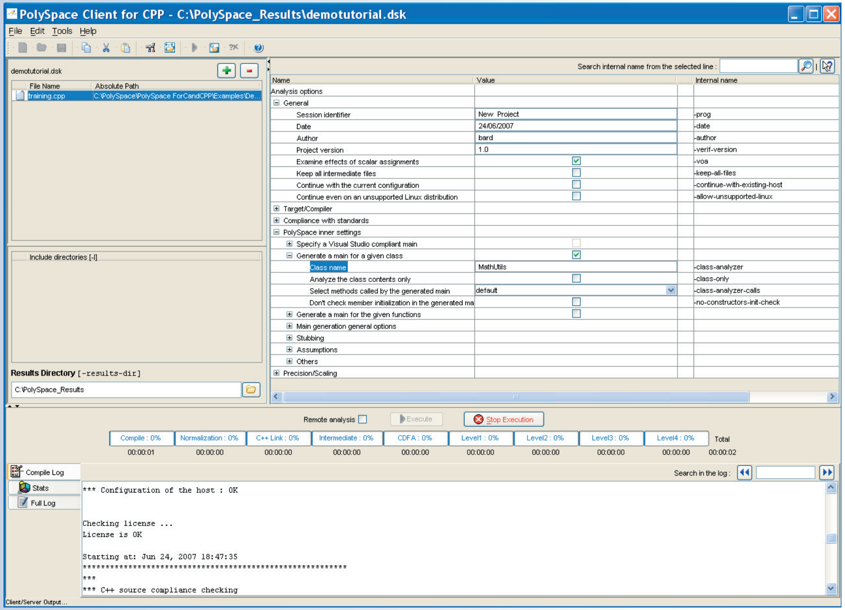


Session identifier: demotutorial

Files of type: *.dsk

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

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



PolySpace Client for CPP - C:\PolySpace_Results\demotutorial.dsk

File Edit Tools Help

demotutorial.dsk

File Name	Absolute Path
training.cpp	C:\PolySpace\PolySpace ForCandCPP\Examples\De...

Include directories [-I]

Results Directory [-results-dir]

C:\PolySpace_Results

Search internal name from the selected line :

Name	Value	Internal name
Analysis options		
⊟ General		
Session identifier	New Project	-prog
Date	24/06/2007	-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 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 generated ma	☐	-no-constructors-init-check
⊞ Generate a main for the given functions	☐	
⊞ Main generation general options		
⊞ Stubbing		
⊞ Assumptions		
⊞ Others		
⊞ Precision/Scaling		

Remote analysis ☐ Execute Stop Execution

Compile : 0%	Normalization : 0%	C++ Link : 0%	Intermediate : 0%	CDFA : 0%	Level1 : 0%	Level2 : 0%	Level3 : 0%	Level4 : 0%	Total
00:00:01	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:02

Compile Log · Stats · Full Log — Search in the log :

```
*** Configuration of the host : OK


Checking license ...
License is OK

Starting at: Jun 24, 2007 18:47:35
***********************************************************
***
*** C++ source compliance checking
```

Client/Server Output...

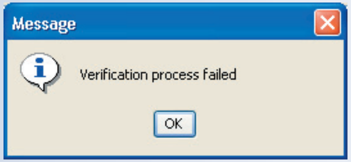
A progress report is displayed in the bottom part of the graphical interface, indicating that the analysis is being performed.

The Execute button is also grayed out.

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

2.3.1. Parsing errors during preliminary PolySpace analysis stages

After some checks, PolySpace will show an error message:

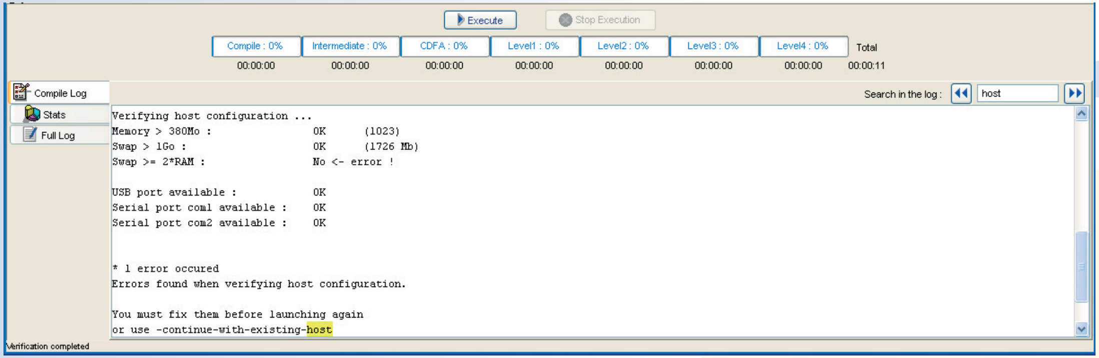


Verification process failed

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 requirements.
A message similar to the one below would be displayed in the bottom part of the graphical interface:



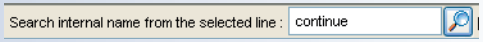
```
Verifying host configuration ...
Memory > 380Mo :               OK      (1023)
Swap > 1Go :                   OK      (1726 Mb)
Swap >= 2*RAM :                No <- error !

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
```

➤ To help you understand the issue, you can search into the log file. Type "`host`" in the "`Search in the log:`" box and click on ◀◀ to check whether the error corresponds to a hardware recommendation problem.

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

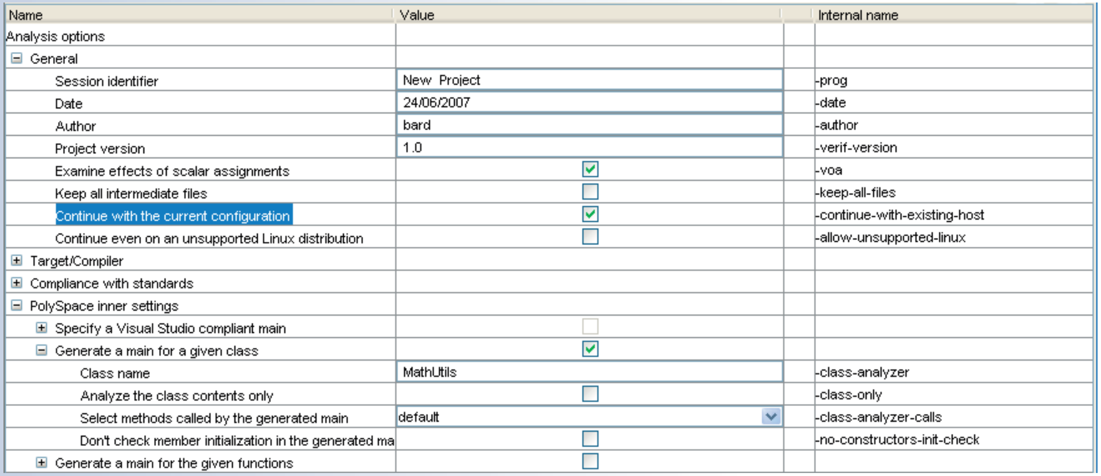
If you have a problem related to host configuration, in order to continue analysis, you can either:
• upgrade your computer to meet the minimal requirements
• or use the `-continue-with-existing-host` option which overrides the initial check for minimal hardware configuration.

➤ To set up the `-continue-with-existing-host` option, please type "`continue`" in the "`Search internal name from the selected line`" box at the top right of the window



Search internal name from the selected line : continue

➤ Then click on 🔍. It will show all options containing the word "`continue`" as shown below:

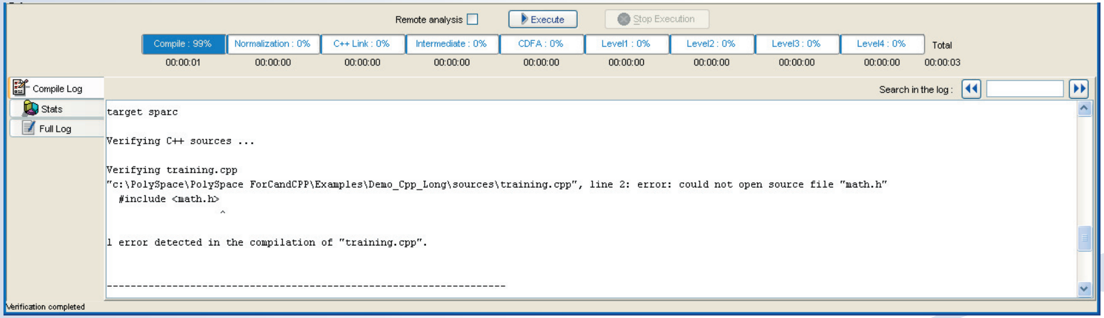


Name	Value	Internal name
Analysis options		
⊟ General		
Session identifier	New Project	-prog
Date	24/06/2007	-date
Author	bard	-author
Project version	1.0	-verif-version
Examine effects of scalar assignments	✓	-voa
Keep all intermediate files	☐	-keep-all-files
Continue with the current configuration	✓	-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 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 generated ma	☐	-no-constructors-init-check
⊞ Generate a main for the given functions	☐	

➤ Check the box ☑ in the "Value" column that is associated to the "`-continue-with-existing-host`" line.

Second possible cause for the error message: Information about Header files

Another cause of error may be that PolySpace Client misses some information about header files.

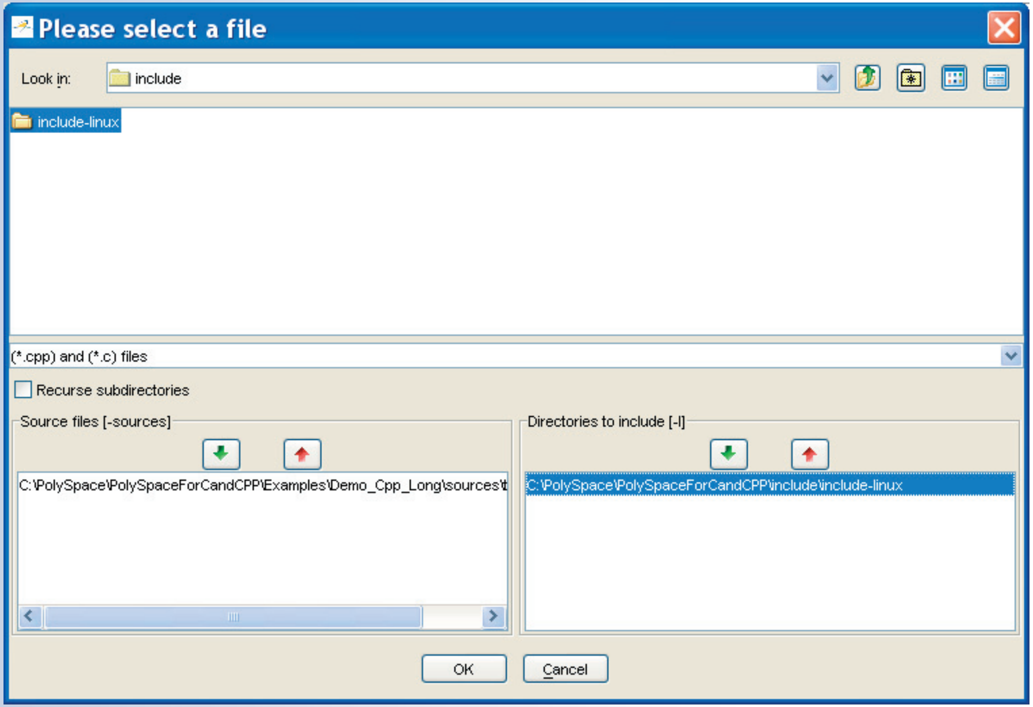


```
target sparc

Verifying C++ sources ...

Verifying training.cpp
"c:\PolySpace\PolySpace ForCandCPP\Examples\Demo_Cpp_Long\sources\training.cpp", line 2: error: could not open source file "math.h"
  #include <math.h>
                   ^

1 error detected in the compilation of "training.cpp".


---------------------------------------------------------------------
```

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 the includes distributed with PolySpace for C/C++. It is located in `<PolySpaceInstallDir>\Verifier\include\include-linux`. You shall use that include file only for the purpose of this tutorial. When analysing your own 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):



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

C:\PolySpace\PolySpaceForCandCPP\include\include-linux

➤ Select "`<PolySpaceInstallDir>\Verifier\include\include-linux`", where <"`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.

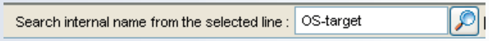
➤ Click on ⬇ in the "`Directories to include [-I]`" section, then close the window using OK.

Notes:

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

In this tutorial, as we have chosen includes of the Linux OS, 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-compilers for these platforms:

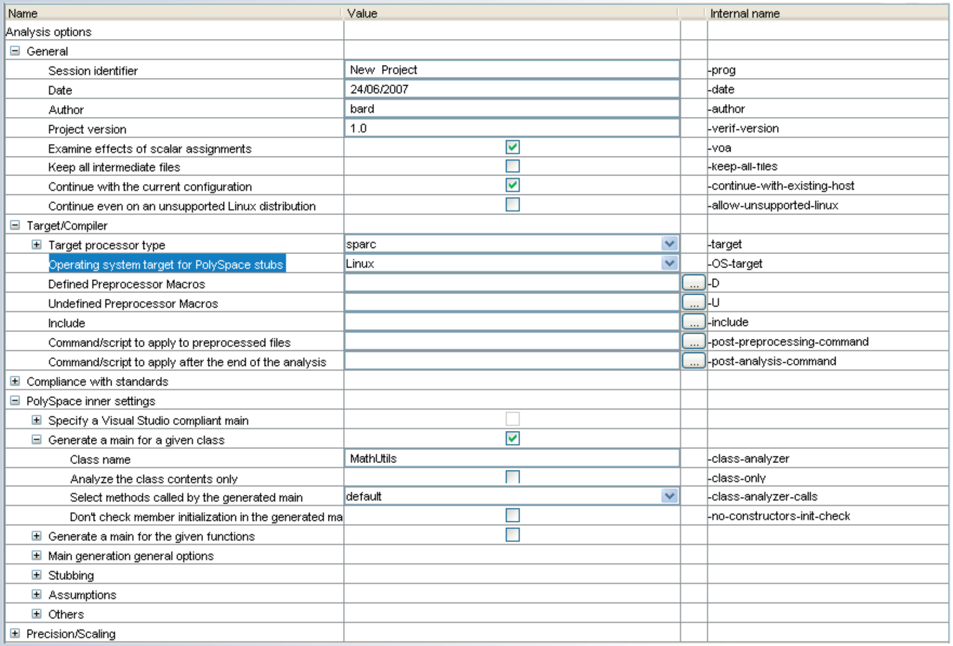
➤ To set up the `-OS-target Linux` option, please type "`OS-target`" in the "`Search internal name from the selected line`" box at the top right of the window



Search internal name from the selected line : OS-target

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

➤ Then click on 🔍. It will show all options containing "`OS-target`" as shown below:



Name	Value	Internal name
Analysis options		
⊟ General		
Session identifier	New_Project	-prog
Date	24/06/2007	-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
Command/script to apply after the end of the analysis		-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 generated ma	☐	-no-constructors-init-check
⊞ Generate a main for the given functions	☐	
⊞ Main generation general options		
⊞ Stubbing		
⊞ Assumptions		
⊞ Others		
⊞ Precision/Scaling		

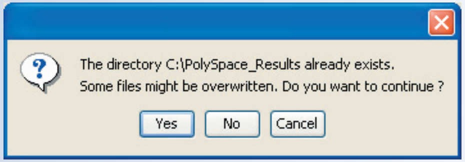
➤ Then, click on ⌄ and chose `Linux` OS target out of all predefined OS targets.

Note: Associated PolySpace defines a set of relevant stubs of standard templates and C libraries depending on the OS target chosen.

2.3.2. Progression of the analysis

➤ Click on Execute to restart the analysis.

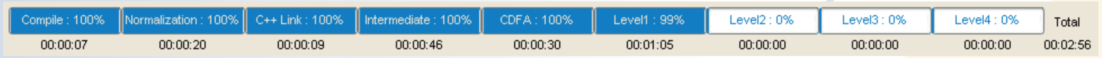
Some results may have already been written in the "`C:\PolySpace_Results`" directory, because of a previous click on Execute. Therefore a window opens to check whether you want to overwrite them


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.

Note: closing the PolySpace Client window will not stop the PolySpace analysis.
If you wish to stop it, click on Stop Execution (you will be asked for confirmation).
If the window is closed without stopping the analysis, the analysis continues in the background.
Opening again PolySpace Client with the same project automatically updates the analysis with its current status.

The progress bar allows following the progress of the analysis:

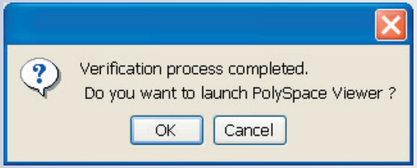


Compile : 100%	Normalization : 100%	C++ Link : 100%	Intermediate : 100%	CDFA : 100%	Level1 : 99%	Level2 : 0%	Level3 : 0%	Level4 : 0%	Total
00:00:07	00:00:20	00:00:09	00:00:46	00:00:30	00:01:05	00:00:00	00:00:00	00:00:00	00:02:56

A progress report may be obtained by clicking on Compile Log for the compilation phase, or Full Log for the full analysis in the bottom part of the window. Click on Stats to get additional information about the 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.

2.3.3. End of the analysis

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



Verification process completed.
Do you want to launch PolySpace Viewer ?

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

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

3. Step 2: PolySpace Class Analyzer

PolySpace Class Analyser analyses applications class by class, even if theses classes are only partially developed. It allows detecting errors at a very early stage, without any test case to write. The process is very simple:

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 initialised with full range (i.e. with a random value);
5. External variables are also defined to random value.

Note: In PolySpace, 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.

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.).

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 initialised 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 of PolySpace.

If we come back to the class "`MathUtils`", analysed in the first step, we can see that it contains a 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.

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. Here is the example with 'MathUtils'.
The log file can be found at root of '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 the existing 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 is not relevant for the class given to the -class-analyzer option.

3.4. Characteristics of a class and messages of the log file

The log file may contain some error messages about the class being analyzed. 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.
---------------------------------------------------------
```

3.5. Behaviour of global variables and members

• Global variables

In a class analysis, global variables are no longer considered as following the ANSI Standard if they are defined but not initialized. Indeed, the ANSI Standard considers, by default, that global variables are initialized to zero.

In a class analysis, global variables do not follow standard behaviour. The are handled as follows:

- 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 initialization 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 will be treated as defined variables.

An example on the right shows the behaviour of two global variables:

```
1
2      extern int fround(float fx);
3
4      // global variables
5      int globvar1;
6      int globvar2 = 100;
7
8      class Location
9      {
10     private:
11       void calculate_new(void);
12       int x;
13
14     public:
15       // constructor 1
16       Location(int intx = 0) { x = intx;   };
17       // constructor 2
18       Location(float fx) { x = fround(fx);  };
19
20       void setx(int intx) {  x = intx; calculate_new(); };
21       void fsetx(float fx) {
22         int tx = fround(fx);
23         if (tx / globvar1 != 0) // ZDV check is orange
24           {
25             tx = tx / globvar2; // ZDV check is green
26             setx(tx);
27           }
28       };
29     };
```

In this example, `globavar1` is defined but not initialized (see line 5): the check for a division by zero at line 23 is thus orange (unproven check). On the other hand, `globvar2` is initialized to 100 (see line 6): check for a division by zero at line 25 is green (proven check).

- **Data members of other classes**

When analysing a specific class, variable members of other classes, even members of parent classes, are considered as initialized as explained below:

1. They are considered as may be not initialized if the constructor of the class is not defined. So they are assigned to full range.
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. Please note that, if the `-class-only` option is used, the definition of the constructor is automatically missing.
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 the behaviour of a variable member of the class `OtherClass` which was given without the definition of its constructor:

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

```
class OtherClass
{
protected:
  int x;
  OtherClass (int intx);              // code is missing
public:
  int getMember(void) {return x;}; // NIV is warning
};

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

3.6. Methods and classes specificities

• Template

A template class can not be analysed alone. Only instances of a template can be analysed with the PolySpace Class Analyser.
If we have `template<class T, class Z> class A { ... }` , we want to analyse template class `A` with two class parameters `T` and `Z`. To do 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 the `-class-analyzer` option, to analyse this instance of the template `A`.

• Abstract classes

An instance of an abstract class can not be created, so it seems that abstract classes can not be analysed.
The workaround is to "remove" the pure declarations. For example, in an abstract class definition, you could change `void abstract_func () = 0;` to `void abstract_func ();`
This kind of modification is done automatically by the class analyser if it is given an abstract class to analyze.
The virtual pure functions would then be ignored in the analysis.

• Static classes

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

• 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 derivate classes, the generated main does not include theses methods when analyzing the class `Point`.

However, inherited members are considered as volatile if they are not explicitly initialised 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. As a consequence, if we analyse the above example in the current state, the method `Location::Location` (constructor) will be stubbed.

4. Step 3: PolySpace Viewer - Exploration of results

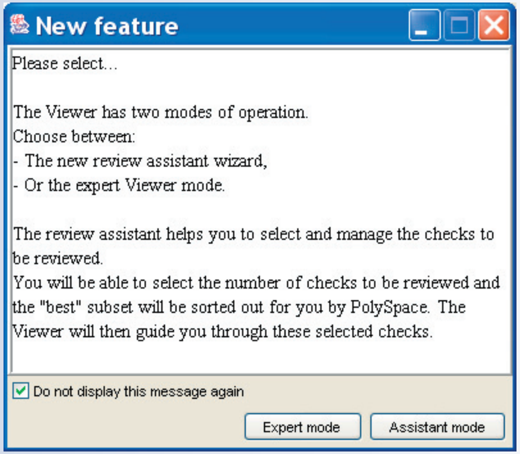
This step illustrates how to explore analysis results that were generated by either PolySpace Client or PolySpace Server. We review the results of the analysis of "`training.cpp`" performed during Step 1. You can access the results of any successful analysis by double clicking on the PolySpace Viewer icon:

PolySpace Viewer
Shortcut
2 KB

If the OK button has been clicked at the end of the previous analysis (see step 1), PolySpace Viewer automatically opens results.

4.1. Modes of operation

The first time the PolySpace Viewer is opened, a window will appear to describe the different modes of operation.



New feature

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

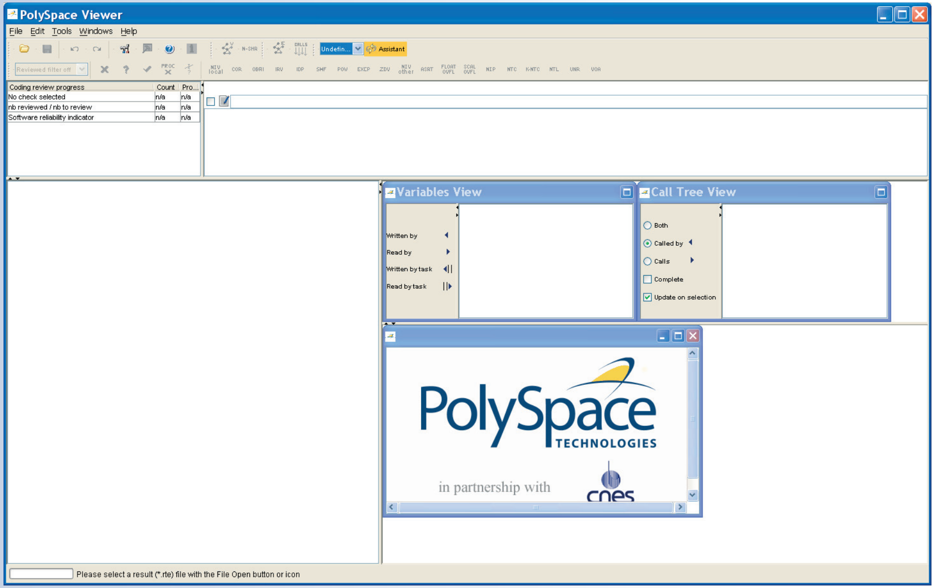
- In "`Expert mode`", all checks can be seen. The number and categories of checks to be reviewed as well as the order in which to review them can be chosen by the user (See next section).
- In "`Assistant mode`", the rules of the review follows a methodology selected by PolySpace. The "best" subset of checks will be automatically selected and sorted out. The PolySpace Viewer will then guide the user through these selected checks.

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

Note: The mode of operation may be changed later in PolySpace Viewer.

4.2. Download results

After having clicked on "`Expert mode`" the PolySpace Viewer window looks like below:



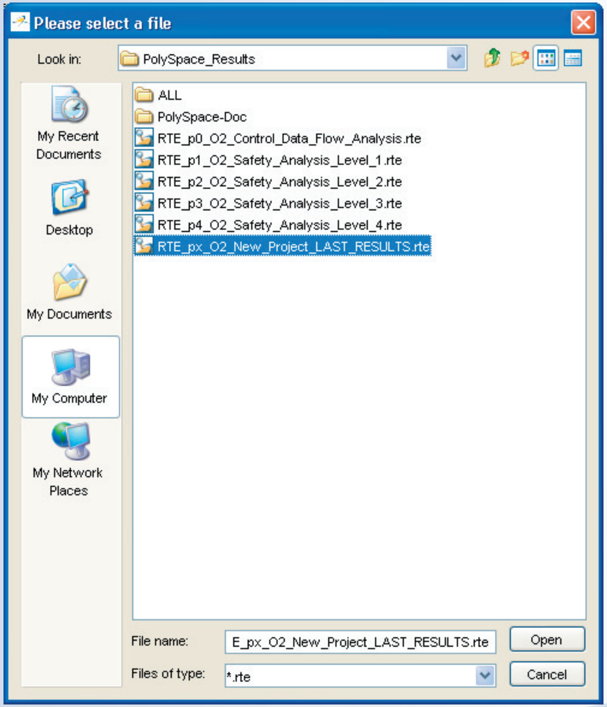
Coding review progress	Count	Pro...
No check selected	n/a	n/a
nb reviewed / nb to review	n/a	n/a
Software reliability indicator	n/a	n/a

[✓ Called by]

[✓ Update on selection]

Please select a result (*.rte) file with the File Open button or icon

➤ 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:
`<PolySpaceInstallDir>\Examples\Demo_CPP_Long\RTE_px_O2_Demo_CPP_LAST_RESULTS.rte`



Please select a file

Look in: 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

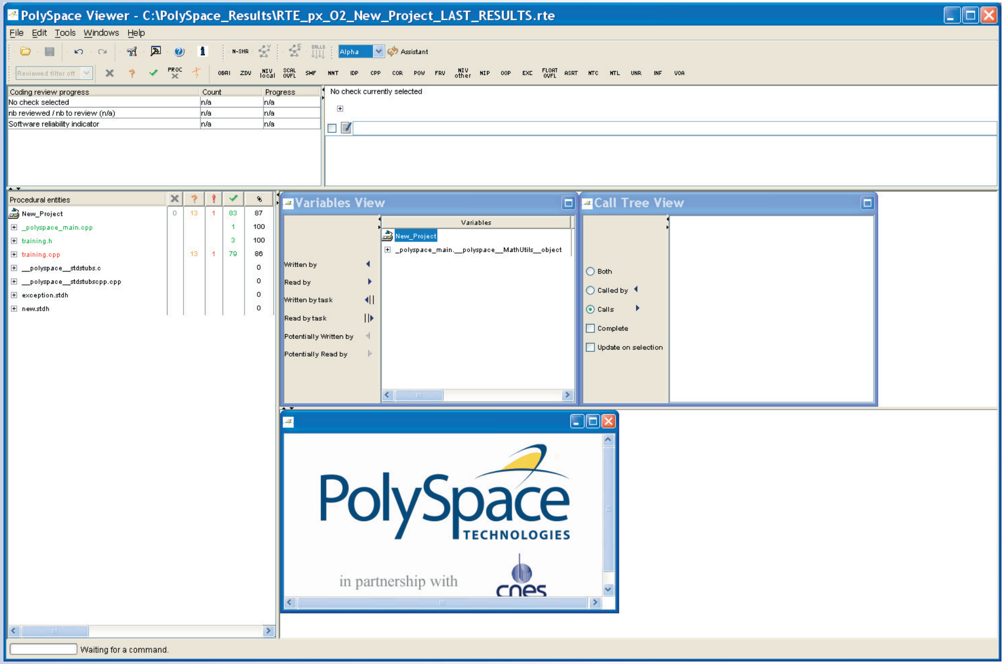
File name: E_px_O2_New_Project_LAST_RESULTS.rte
Files of type: *.rte

➤ Then click on Open to proceed with further steps

Note:
The `RTE_px_O2_New_Project_LAST_RESULTS.rte` is a sort of "link" on the best analysis in term of precision: RTE_p4_O2_Safety_Analysis_Level4.rte.
Other results have lower precision.

4.3. Analysing PolySpace results in expert mode ("`training.cpp`")

After loading the results in "`Expert mode`", PolySpace Viewer window looks like below:



Coding review progress	Count	Progress
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

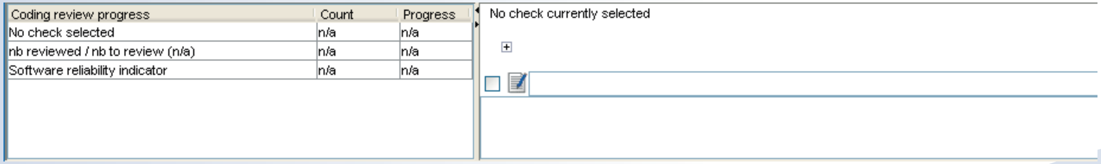
No check currently selected

Procedural entities	✗	?	!	✓	%
New_Project	0	13	1	83	87
_polyspace_main.cpp				1	100
training.h				3	100
training.cpp		13	1	79	86
__polyspace__stdstubs.c					0
__polyspace__stdstubscpp.cpp					0
exception.stdh					0
new.stdh					0

New_Project
_polyspace_main.__polyspace__MathUtils__object

Waiting for a command.

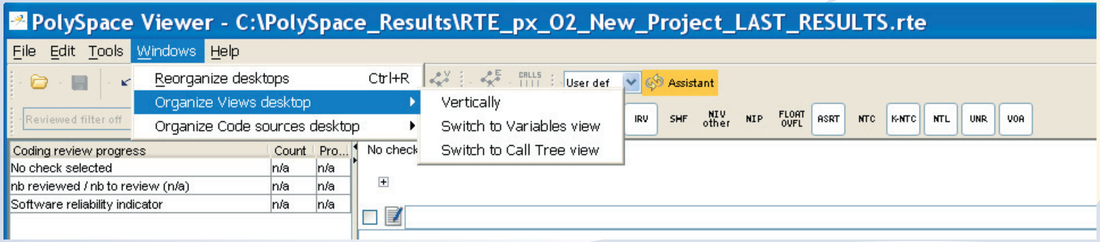
1. On the bottom left is the "`Procedural Entities`" View. It displays the list of files analyzed in the "`Procedural entities`" column.
2. In the bottom right area is the source code view. Each operation checked is displayed using meaningful colour scheme and related diagnostic:
   - Red: Errors which occur at every execution.
   - Orange: Unproven - an error may occur sometimes.
   - Grey: Shows unreachable code.
   - Green: Error condition that will never occur.
3. The two windows just below the tool bar display details about the currently reviewed check (when the check has been selected):



Coding review progress	Count	Progress
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

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:



PolySpace Viewer - C:\PolySpace_Results\RTE_px_02_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

4.3.1. Procedural entities view

Each file and underlying functions in the "Procedural entities" view is colorized according to the most critical error found:

- `exception.stdh.` contains no check. It 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.` contains no check. It contains implemented stubs of the `<new>` part of the stl library template.

- `__polyspace_main.cpp` contains the main which was automatically generated. All checks there are green: no run-time error 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.

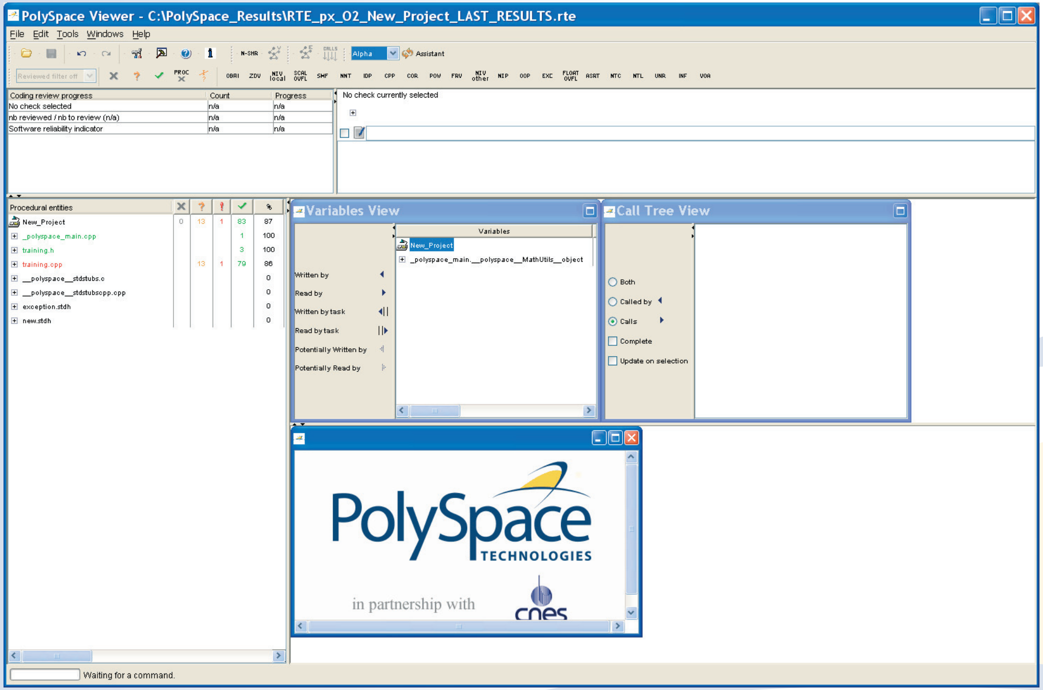
- `training.cpp.` contains the analysed Class "`MathUtils`". It is colored in red because one or more *definite* run-time errors have been found in it.

- training.h. is colored in green because no run-time error has been found.

- `__polyspace_stdstubs.c.` contains stubs of standard functions part of the `libc` library used in training.cpp. This file contains no check.

- `__polyspace__stdstubscpp.cpp.` 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 expand it and display the list of function members defined within "`MathUtils`" of "`training.cpp`". The function members in red or grey (`MathUtils::Pointer_Arithmetic()`,etc.) have code sections that need to be inspected first because they are definite diagnosis of PolySpace (either run-time errors or dead code).



Coding review progress	Count	Progress
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	✗	?	!	✓	%
New_Project	0	13	1	83	87
⊞ _polyspace_main.cpp				1	100
⊞ training.h				3	100
⊞ training.cpp		13	1	79	86
⊞ __polyspace__stdstubs.c					0
⊞ __polyspace__stdstubscpp.cpp					0
⊞ exception.stdh					0
⊞ new.stdh					0

Variables
New_Project
⊞ _polyspace_main.__polyspace__MathUtils__object

Waiting for a command.

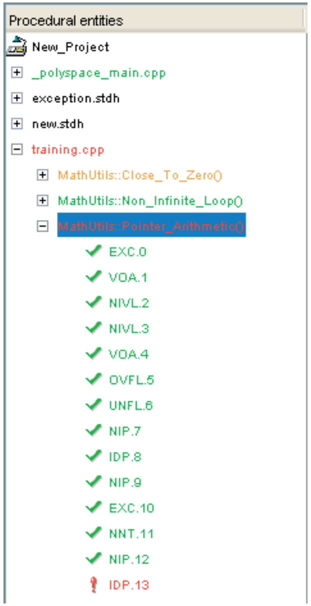
The columns (%, !, ✓, ✕, ?,...) provide information about run-time errors found in each function:

- The % column indicates the **Software reliability indicator** in percentage (level of proof). 100% means a complete reliability on the code for the category checked by PolySpace with the hypothesis taken for the analysis.
- The ! column indicates the number of definite run-time errors or reds,
- The ? column indicates the number of unproven or oranges (may be 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 an error found by PolySpace in "`training.cpp`".

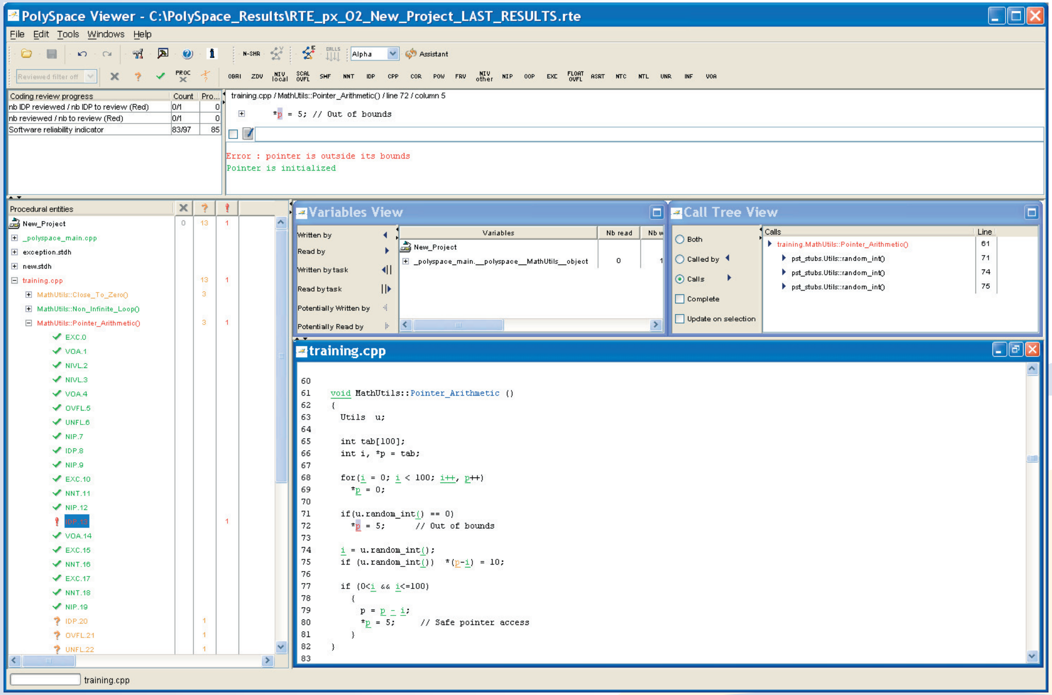
Example of runtime error found by PolySpace: Memory Corruption

➤ 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
⊞ exception.stdh
⊞ new.stdh
⊟ 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
```

➤ Click on the red "IDP.13" item - which stands for **I**llegal **De**-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

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

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

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

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

Procedural entities	✗	?	!
New_Project	0	13	1
_polyspace_main.cpp			
exception.stdh			
new.stdh			
training.cpp		13	1
MathUtils::Close_To_Zero()		3	
MathUtils::Non_Infinite_Loop()			
MathUtils::Pointer_Arithmetic()		3	1
✓ 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			1
✓ VOA.14			
✓ EXC.15			
✓ NNT.16			
✓ EXC.17			
✓ NNT.18			
✓ NIP.19			
? IDP.20		1	
? OVFL.21		1	
? UNFL.22		1	

Variables View

Variables	Nb read	Nb w
New_Project		
_polyspace_main.__polyspace__MathUtils__object	0	1

Call Tree View

Both / Called by / [✓ Calls] / Complete / Update on selection

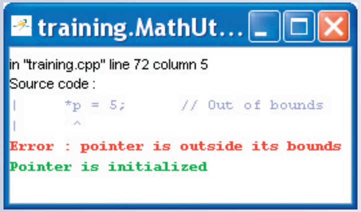
Calls	Line
training.MathUtils::Pointer_Arithmetic()	61
pst_stubs.Utils::random_int()	71
pst_stubs.Utils::random_int()	74
pst_stubs.Utils::random_int()	75

training.cpp

```
60
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  }
83
```

training.cpp

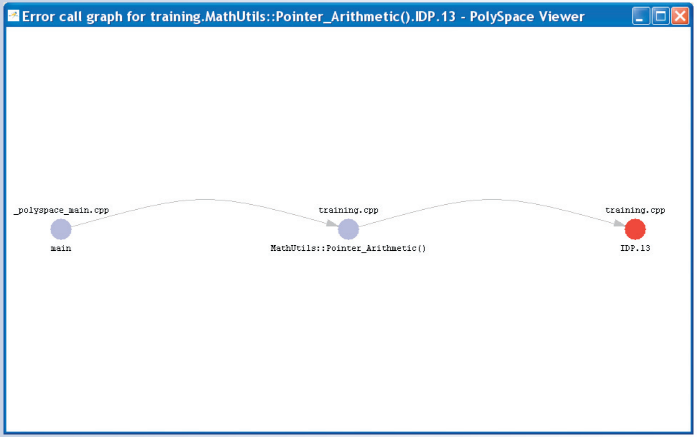
➤ Click on red symbol in the source code at line 72. An error message is opened:



```
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 the "IDP.13" item in the "`Procedural entities`" column, 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

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`".
- Underlined blue: A link to a global variable in the "Global variable View".

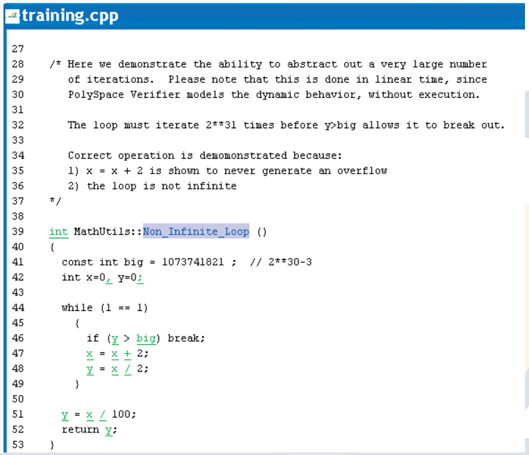
4.3.3. More examples of diagnostic

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. The function is fully green: it means that the local 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.

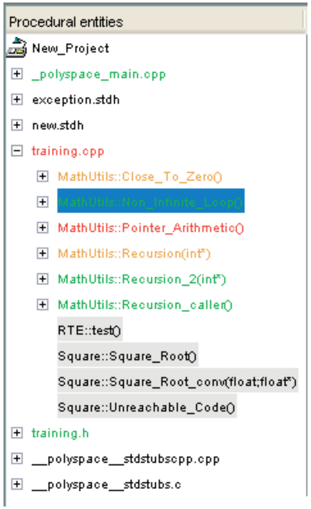
Note: using the `-voa` option when launching the analysis, PolySpace can give more information about the range on scalar assignments


training.cpp

```

/* Here we demonstrate the ability to abstract out a very large number
   of iterations.  Please note that this is done in linear time, since
   PolySpace Verifier models the dynamic behavior, without execution.

   The loop must iterate 2**31 times before y>big allows it to break out.

   Correct operation is demomonstrated because:
   1) x = x + 2 is shown to never generate an overflow
   2) the loop is not infinite
*/

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;
}
```

Other example: unreachable code

We can also see in the "`Procedural entities`" column that some function members are never called. It is materialised by a grey background:



Procedural entities

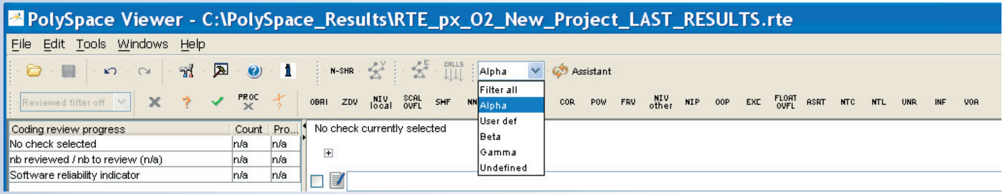
- New_Project
  - _polyspace_main.cpp
  - exception.stdh
  - new.stdh
  - training.cpp
    - MathUtils::Close_To_Zero()
    - MathUtils::Non_Infinite_Loop()
    - MathUtils::Pointer_Arithmetic()
    - MathUtils::Recursion(int*)
    - MathUtils::Recursion_2(int*)
    - MathUtils::Recursion_caller()
 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

In the figure on the right 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`".

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`", "`Gamma`", "`User def`" and "`Filter All`" that you can choose depending on your development process:


PolySpace Viewer - C:\PolySpace_Results\RTE_px_02_New_Project_LAST_RESULTS.rte
File Edit Tools Windows Help
N-SHR
CALLS
Alpha
Assistant
Reviewed filter off
PROC
OBAI ZDV NIV local SCAL OVFL SHF NN
Filter all
Alpha
User def
Beta
Gamma
Undefined
COR POW FRV NIV other NIP OOP EXC FLOAT OVFL ASRT NTC NTL UNR INF VOA

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

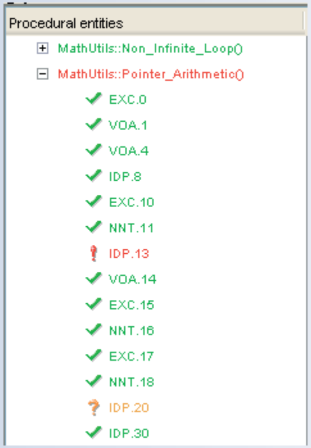
No check currently selected

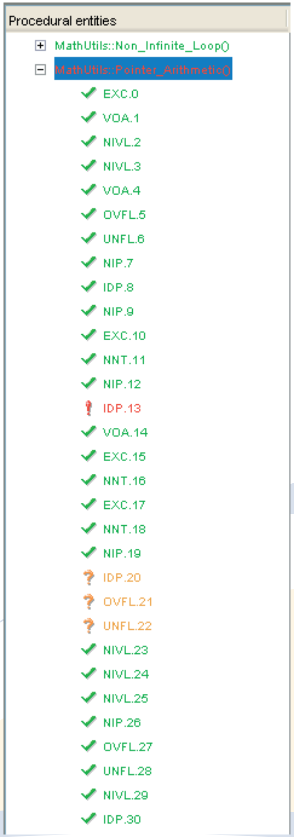
➤ Select "`Gamma`" to get all the "red" and "grey" `code sections`. It is mainly used during the earliest development stages to focus quickly on critical bugs.

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

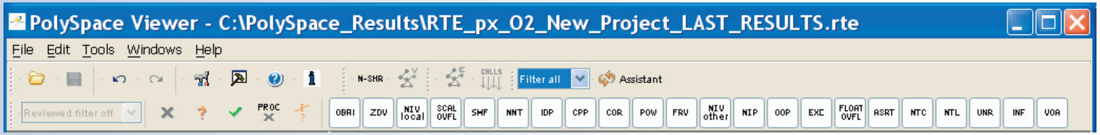
This list of acronyms - for type of operations checked - shows what kind of errors PolySpace automatically looked for.
The "`Beta`" level highlights checks that could cause a processor halt, memory corruptions or overflows.

➤ Select "`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.


Procedural entities
⊞ MathUtils::Non_Infinite_Loop()
⊟ 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

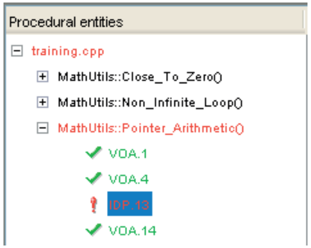

Procedural entities
⊞ MathUtils::Non_Infinite_Loop()
⊟ MathUtils::Pointer_Arithmetic()
✓ EXC.0
✓ VOA.1
✓ NIVL.2
✓ NIVL.3
✓ VOA.4
✓ OVFL.5
✓ UNFL.6
✓ NIP.7
✓ IDP.8
✓ NIP.9
✓ EXC.10
✓ NNT.11
✓ NIP.12
! IDP.13
✓ VOA.14
✓ EXC.15
✓ NNT.16
✓ EXC.17
✓ NNT.18
✓ NIP.19
? IDP.20
? OVFL.21
? UNFL.22
✓ NIVL.23
✓ NIVL.24
✓ NIVL.25
✓ NIP.26
✓ OVFL.27
✓ UNFL.28
✓ NIVL.29
✓ 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
N-SHR
CALLS
Filter all
Assistant
Reviewed filter off
PROC
OBAI ZDV NIV local SCAL OVFL SHF NNT IDP CPP COR POW FRV NIV other NIP OOP EXC FLOAT OVFL ASRT NTC NTL UNR INF VOA

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

➤ Click on ✓ (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, unproven or greens:



```
⊟ training.cpp
   ⊞ MathUtils::Close_To_Zero()
   ⊞ MathUtils::Non_Infinite_Loop()
   ⊟ MathUtils::Pointer_Arithmetic()
         ✓ VOA.1
         ✓ VOA.4
         ! IDP.13
         ✓ VOA.14
```

Note Clicking on "VOA" will hide remaining informative green code sections, leaving only the red, orange and gray checks.

4.3.5. C++ specific checks

Specific C++ checks are split into five categories:

1. NNT or "`Non Null This pointer`" (NNT)checks the `this-pointer` validity.
2. CPP checks all C++ related constructions (CPP), like positive array size verification, `dynamic_cast, typeid` parameters, etc.
3. OOP checks all C++ object oriented issues (OOP).
4. EXC checks all C++ constructions dealing with exceptions (EXC).
5. INF displays information about C++ implicit and called functions when dealing with virtual functions (INF).

When reviewing C++ code with PolySpace Viewer, it is important to do a selective review (category by category) in the order of the list of categories located at the top of the PolySpace Viewer window from left to right.
It is also important to review C "like" checks before C++ like "checks".

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`").

4.4. Methodological assistant

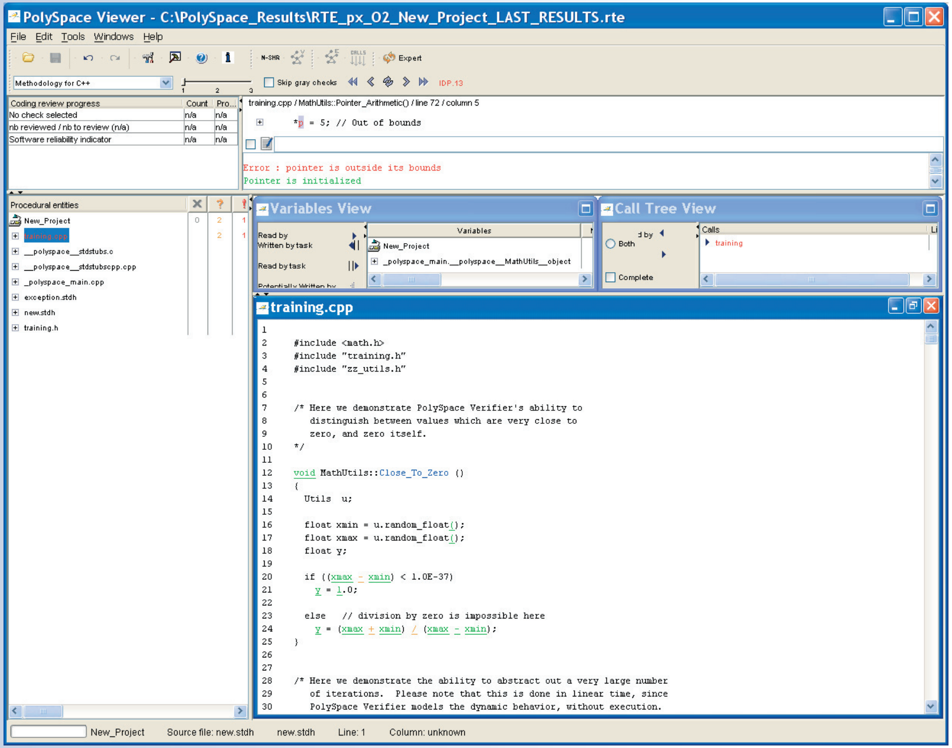
After this first usage of PolySpace Viewer, some simple questions remain:

- Do all checks need to be reviewed?
- If not, what are the checks to review?
- How many?
- What is the best order?

The Methodological assistant answers to all theses questions. It helps to select and manage the checks to be reviewed. It selects a "best" subset and sorts checks out. The Assistant mode in the PolySpace Viewer will then guide you through these selected checks.

➤ If the PolySpace Viewer is still open, close it by clicking on the upper right symbol, open it again, load the same results and chose the "`Assistant`" mode.

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



PolySpace Viewer - C:\PolySpace_Results\RTE_px_02_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

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

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

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

Procedural entities	✕	?	!
New_Project	0	2	1
training.cpp		2	1
__polyspace__stdstubs.c			
__polyspace__stdstubscpp.cpp			
_polyspace_main.cpp			
exception.stdh			
new.stdh			
training.h			

Variables View: New_Project; _polyspace_main.__polyspace__MathUtils__object

Call Tree View: Calls: training

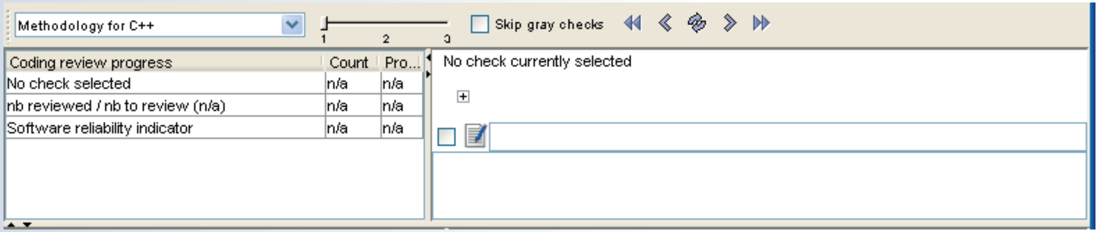
training.cpp

```

#include <math.h>
#include "training.h"
#include "zz_utils.h"


/* Here we demonstrate PolySpace Verifier's ability to
   distinguish between values which are very close to
   zero, and zero itself.
*/

void MathUtils::Close_To_Zero ()
{
  Utils  u;

  float xmin = u.random_float();
  float xmax = u.random_float();
  float y;

  if ((xmax - xmin) < 1.0E-37)
    y = 1.0;

  else   // division by zero is impossible here
    y = (xmax + xmin) / (xmax - xmin);
}


/* Here we demonstrate the ability to abstract out a very large number
   of iterations.  Please note that this is done in linear time, since
   PolySpace Verifier models the dynamic behavior, without execution.
```

New_Project   Source file: new.stdh   new.stdh   Line: 1   Column: unknown

4.4.1. Assistant dashboard

The second line of buttons on the toolbar and the two views just below are used to navigate between the checks selected by PolySpace:



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

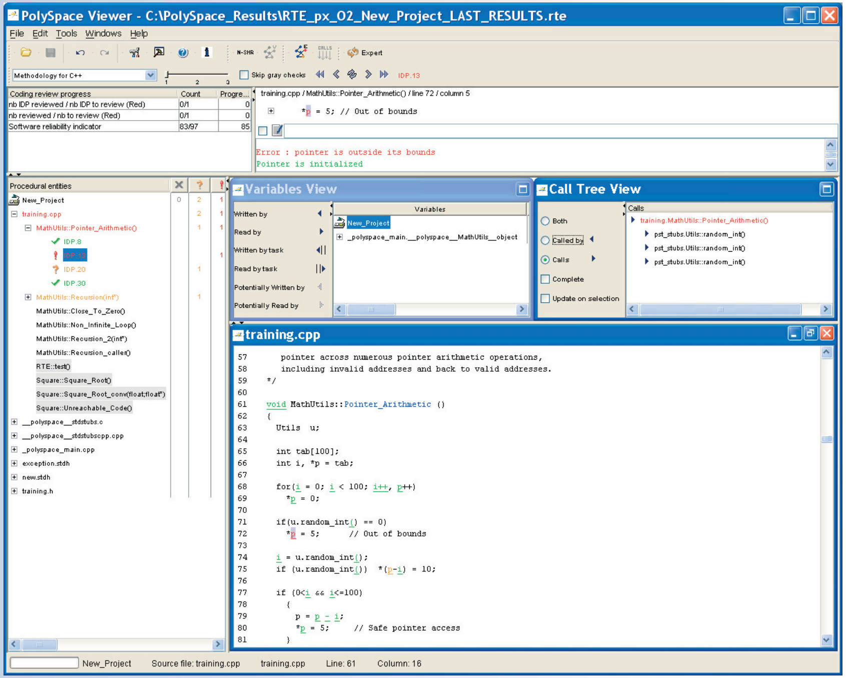
No check currently selected

PolySpace Viewer has also been updated as follows:

1. Now, in the "Procedural Entities" view the list of files analyzed is sorted out according to the methodological assistant used.
2. In the source code view, each operation will be sorted out according to the PolySpace methodology in the following order:
   - Red: The methodological assistant browses all red errors.
   - Gray: The methodological assistant browses unreachable code depending on the radio button "`Skip gray checks`".
   - Orange: The methodological assistant chooses and reviews the "best" unproven operations - those that are the most probably actual errors.

➤ Click on to go to the next check.

PolySpace Viewer has been refreshed with the first check selected by the methodological assistant:



PolySpace Viewer - C:\PolySpace_Results\RTE_px_02_New_Project_LAST_RESULTS.rte

File Edit Tools Windows Help

N-SHR Expert

Methodology for C++ | Skip gray checks | IDP.13

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

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

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

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

Procedural entities	✗	?	!
New_Project	0	2	1
training.cpp		2	1
MathUtils::Pointer_Arithmetic()		1	1
✓ IDP.8			
! IDP.13			1
? IDP.20		1	
✓ IDP.30			
MathUtils::Recursion(int*)		1	
MathUtils::Close_To_Zero()			
MathUtils::Non_Infinite_Loop()			
MathUtils::Recursion_2(int*)			
MathUtils::Recursion_caller()			
RTE::test()			
Square::Square_Root()			
Square::Square_Root_conv(float,float*)			
Square::Unreachable_Code()			
__polyspace__stdstubs.c			
__polyspace__stdstubscpp.cpp			
_polyspace_main.cpp			
exception.stdh			
new.stdh			
training.h			

Variables View

Written by / Read by / Written by task / Read by task / Potentially Written by / Potentially Read by

Variables
- New_Project
- _polyspace_main.__polyspace__MathUtils__object

Call Tree View

Both / Called by / [✓ Calls] / Complete / Update on selection

Calls
- training.MathUtils::Pointer_Arithmetic()
  - pst_stubs.Utils::random_int()
  - pst_stubs.Utils::random_int()
  - pst_stubs.Utils::random_int()

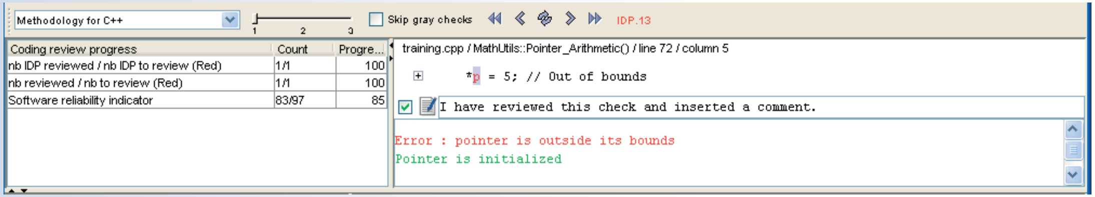
training.cpp

```
57       pointer across numerous pointer arithmetic operations,
58       including invalid addresses and back to valid addresses.
59    */
60
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        }
```

New_Project   Source file: training.cpp   training.cpp   Line: 61   Column: 16

The methodological dashboard gives details and allows reviewing the check.

➤ Tick the review checkbox and type a comment in the text box on the right as follows:



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

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

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

I have reviewed this check and inserted a comment.

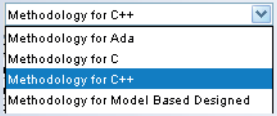
```
Error : pointer is outside its bounds
Pointer is initialized
```

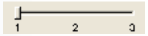
The left part of the dashboard has been updated, and displays some statistics in three lines:

- The first line gives the number and percentage of remaining checks to review in the selected category (here, red IDP checks).
- The second line gives values for the whole colour category (red, grey and unproven).
- The last line gives values for the whole software being reviewed. This is called `Software reliability indicator`. It gives the percentage of green checks compared to the total number of checks.

Other buttons in the Methodological dashboard allow navigating to the next ⏩ or previous ⏪ check that hasn't been reviewed yet. It's also possible to refresh the different views to come back to the check currently being reviewed using the button.

4.4.2. Choose a methodological assistant

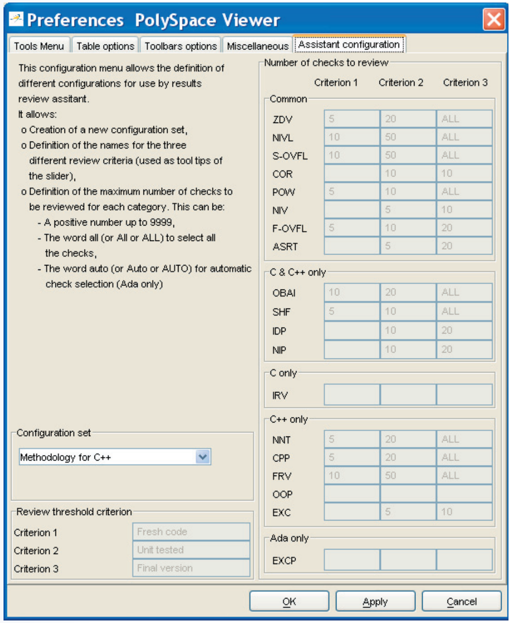
Some methodologies 

and associated levels 

have been pre-defined by PolySpace.

The level defines the number of checks to review by category. It is chosen according to the development phase during which the code has been analyzed: "`Fresh code`", "`Unit test`" and "`Code review`"
It is possible to define your own Methodology in the "`Preferences >PolySpace Viewer>Assistant methodology`" tab that is accessible from the "`Edit`" menu.

Here, you can create a new configuration set and define for each level what will be the categories of check to review and how many of each category.



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

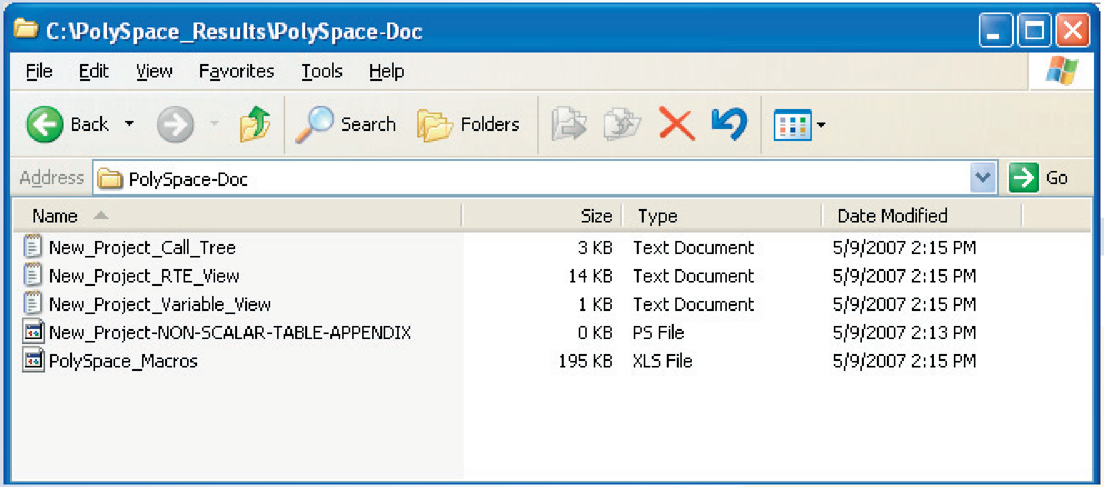
Number of checks to review

	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			

4.5. Report Generation

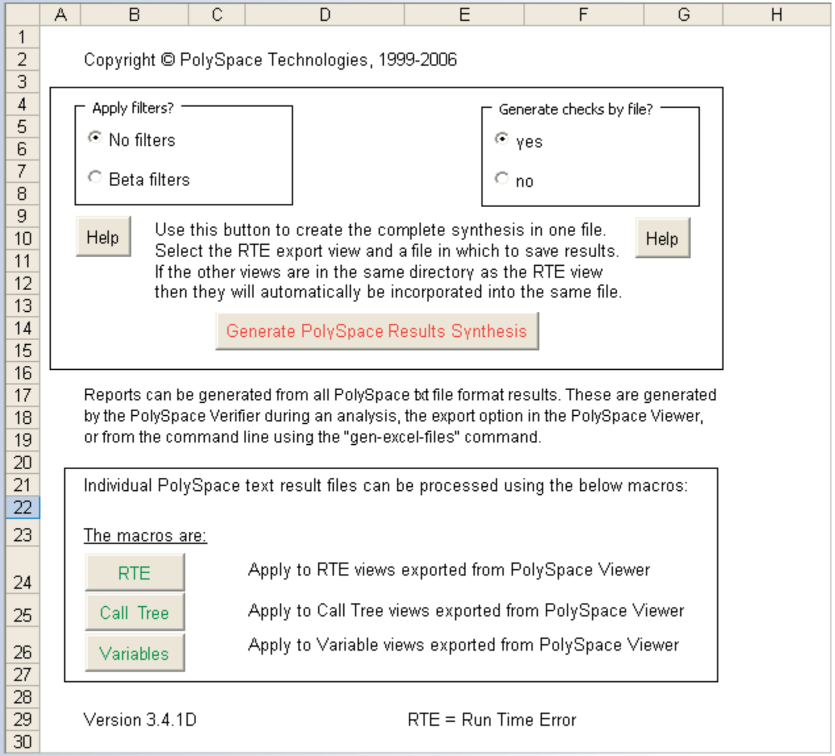
When PolySpace performs an analysis, it generates textual files that can be used to create Excel® reports. These files are located in the results directory (See "`C:\PolySpace_Results\PolySpace-Doc`" or "`<PolySpaceInstallDir>\Examples\Demo_CPP_Long\PolySpace-Doc`"). These files contain data related to all views except the source code one.

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



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`" and enable macros to display the Excel® file below:



Copyright © PolySpace Technologies, 1999-2006

Apply filters?
[✓ No filters]
Beta filters

Generate checks by file?
[✓ yes]
no

Help

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

Help

Generate PolySpace Results Synthesis

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.

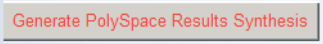
Individual PolySpace text result files can be processed using the below macros:

The macros are:

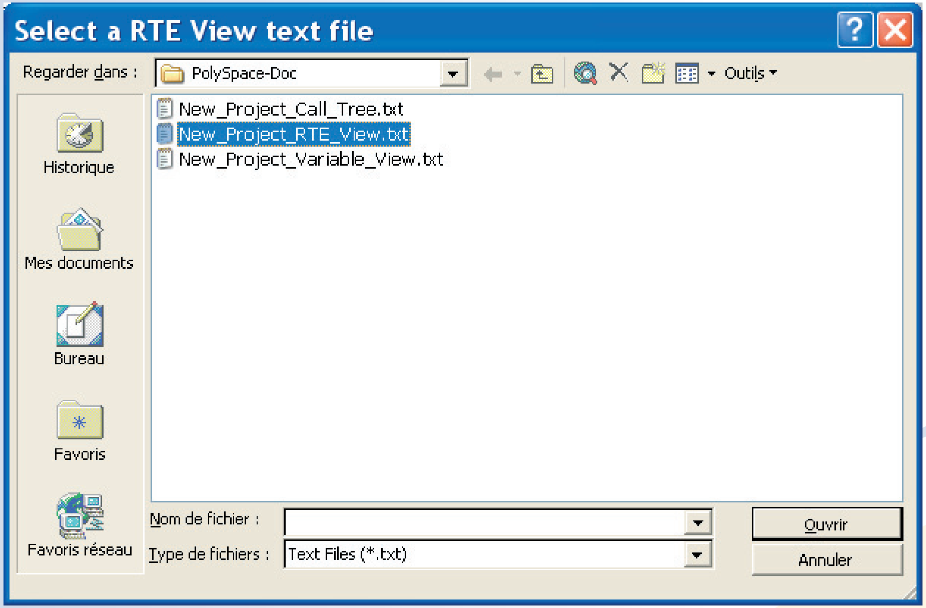
RTE	Apply to RTE views exported from PolySpace Viewer
Call Tree	Apply to Call Tree views exported from PolySpace Viewer
Variables	Apply to Variable views exported from PolySpace Viewer

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:



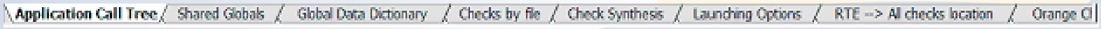
Select a RTE View text file

Regarder dans : PolySpace-Doc

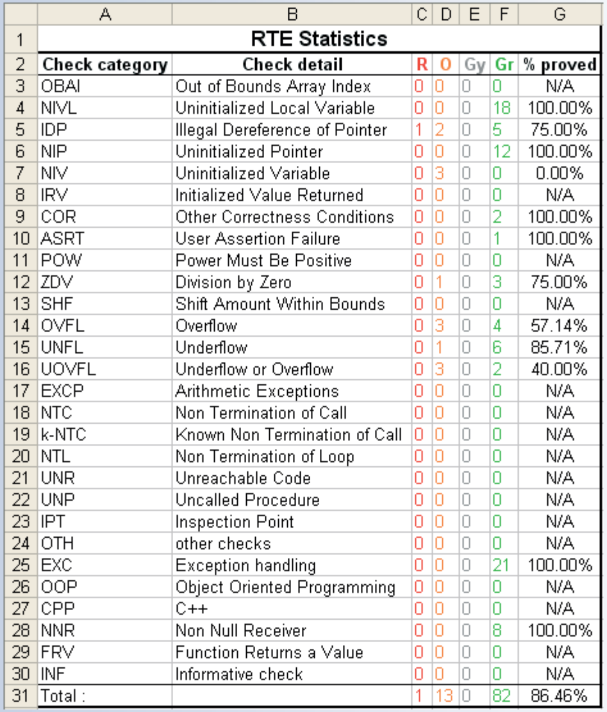
New_Project_Call_Tree.txt
New_Project_RTE_View.txt
New_Project_Variable_View.txt

Nom de fichier :
Type de fichiers : Text Files (*.txt)

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



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



	A	B	C	D	E	F	G
1		RTE Statistics					
2	Check category	Check detail	R	O	Gy	Gr	% proved
3	OBAI	Out of Bounds Array Index	0	0	0	0	N/A
4	NIVL	Uninitialized Local Variable	0	0	0	18	100.00%
5	IDP	Illegal Dereference of Pointer	1	2	0	5	75.00%
6	NIP	Uninitialized Pointer	0	0	0	12	100.00%
7	NIV	Uninitialized Variable	0	3	0	0	0.00%
8	IRV	Initialized Value Returned	0	0	0	0	N/A
9	COR	Other Correctness Conditions	0	0	0	2	100.00%
10	ASRT	User Assertion Failure	0	0	0	1	100.00%
11	POW	Power Must Be Positive	0	0	0	0	N/A
12	ZDV	Division by Zero	0	1	0	3	75.00%
13	SHF	Shift Amount Within Bounds	0	0	0	0	N/A
14	OVFL	Overflow	0	3	0	4	57.14%
15	UNFL	Underflow	0	1	0	6	85.71%
16	UOVFL	Underflow or Overflow	0	3	0	2	40.00%
17	EXCP	Arithmetic Exceptions	0	0	0	0	N/A
18	NTC	Non Termination of Call	0	0	0	0	N/A
19	k-NTC	Known Non Termination of Call	0	0	0	0	N/A
20	NTL	Non Termination of Loop	0	0	0	0	N/A
21	UNR	Unreachable Code	0	0	0	0	N/A
22	UNP	Uncalled Procedure	0	0	0	0	N/A
23	IPT	Inspection Point	0	0	0	0	N/A
24	OTH	other checks	0	0	0	0	N/A
25	EXC	Exception handling	0	0	0	21	100.00%
26	OOP	Object Oriented Programming	0	0	0	0	N/A
27	CPP	C++	0	0	0	0	N/A
28	NNR	Non Null Receiver	0	0	0	8	100.00%
29	FRV	Function Returns a Value	0	0	0	0	N/A
30	INF	Informative check	0	0	0	0	N/A
31	Total :		1	13	0	82	86.46%

5. Launch PolySpace Remotely

This paragraph describes the basic steps to launch an analysis in remote. To do so you need:

1. A Queue Manager server (QM) installed.
2. Your desktop PC configured with a PolySpace Client.
3. A networked machine configured with a PolySpace Server.

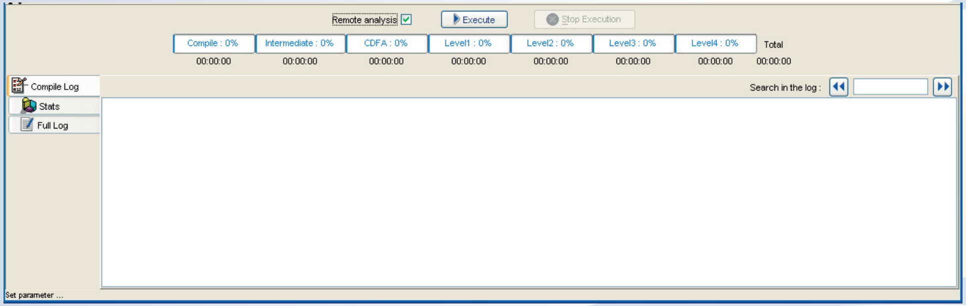
Please see the PolySpace Installation guide (available on the PolySpace CD-ROM in `\Docs\Install`) to install and configure, the Queue Manager, a Client and a Server.

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

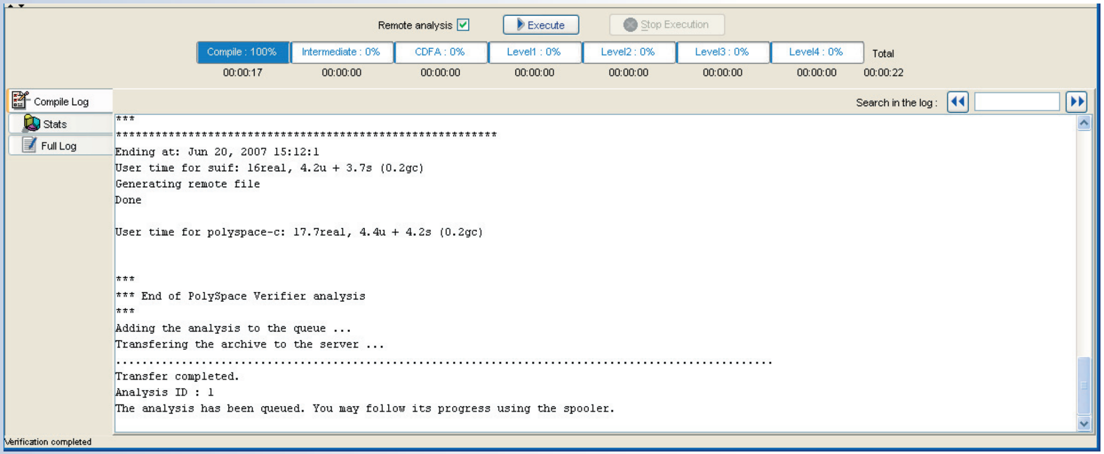
5.1. Launching an analysis

It can be done in two steps:

➤ Step 1: set up an analysis as described in Chapter 2 but do not launch it.
➤ Step 2: tick the "`Remote analysis`" checkbox (see figure below) and 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 see a message like this:



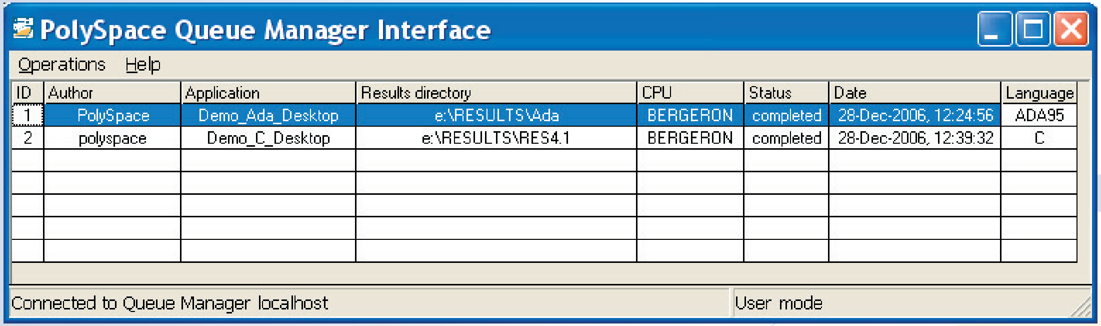
```
***
**********************************************************
Ending at: Jun 20, 2007 15:12:1
User time for suif: 16real, 4.2u + 3.7s (0.2gc)
Generating remote file
Done

User time for polyspace-c: 17.7real, 4.4u + 4.2s (0.2gc)


***
*** End of PolySpace Verifier analysis
***
Adding the analysis to the queue ...
Transfering the archive to the server ...
....................................................................................................
Transfer completed.
Analysis ID : 1
The analysis has been queued. You may follow its progress using the spooler.
```

The analysis has been queued with an ID number, and you can follow its progression using the PolySpace Spooler.
If you do not tick the "`Remote analysis`" checkbox, the analysis continues locally.

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

You can check the analysis processes in the queue by clicking on the short cut on your desktop PC

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

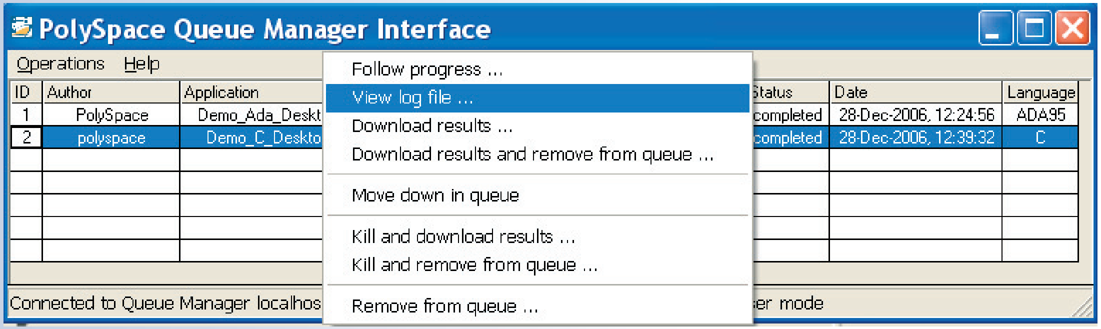


ID	Author	Application	Results directory	CPU	Status	Date	Language
1	PolySpace	Demo_Ada_Desktop	e:\RESULTS\Ada	BERGERON	completed	28-Dec-2006, 12:24:56	ADA95
2	polyspace	Demo_C_Desktop	e:\RESULTS\RES4.1	BERGERON	completed	28-Dec-2006, 12:39:32	C

Connected to Queue Manager localhost

User mode

When you select an analysis and right click, you can manage it in the queue:

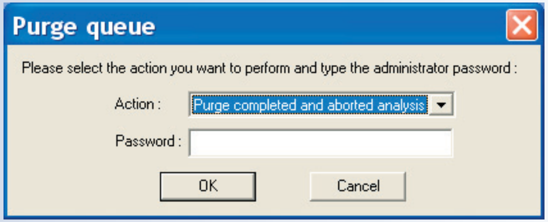

PolySpace Queue Manager Interface
Operations
Help
Follow progress ...
View log file ...
Download results ...
Download results and remove from queue ...
Move down in queue
Kill and download results ...
Kill and remove from queue ...
Remove from queue ...

ID	Author	Application	Status	Date	Language
1	PolySpace	Demo_Ada_Deskt	completed	28-Dec-2006, 12:24:56	ADA95
2	polyspace	Demo_C_Deskto	completed	28-Dec-2006, 12:39:32	C

Connected to Queue Manager localhos ... er mode

- "*Follow progress*" displays the associated log file in a Launcher window. If the analysis is running, you can follow the update of the log file and associated progress bar in real time.
- "*View log file*" displays the associated log file in a "Command prompt" window, in which you can see the last 100 updated lines of the log file in real time. This option is only available when the analysis is running.
- "*Download results*" downloads the results of an analysis to the Client. If the analysis is still running, already available partial results are downloaded on the Client, without disturbing the analysis.
  This option is not available for a "queued" analysis (that has not yet began).
- "*Move down in queue*" reduces the priority of a "`queued`" analysis.
- "*Kill and download results*" stops the analysis definitively and the latest available partial results are downloaded. The status of the analysis changes from "`running`" to "`aborted`". The analysis remains in the queue.
- "*Kill and remove from queue*" stops the analysis definitively and removes it from the queue.
  The results will be lost.
- "*Remove from queue*" removes a "`queued`", "`aborted`" or "`completed`" analysis. **The results will be lost.**

The queue can also be managed via the "`Operations>`" menu:



Purge queue

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

Action : Purge completed and aborted analysis

Password :

- "`Operations>Purge queue`" purges the entire queue or purges only completed and aborted analyses (see below). The queue manager administrator password is required.

- "`Operations>Change root password`" changes administrator password of the queue manager. By default, the password is "`administrator`".

5.3. Batch commands

• Launch analysis in batch:

A set of commands allow the launching of analyses in batch (under a Cygwin shell on a Windows machine). All commands begin with the prefix `<PolySpaceCommonDir>/RemoteLauncher/bin/polyspace-remote-`. Commands available are `polyspace-remote-cpp` and `polyspace-remote-desktop-cpp`.

They are equivalent to the commands with a prefix `<PolySpaceInstallDir>/bin/polyspace-`.
For example, `polyspace-remote-desktop-cpp -server [<hostname>:[<port>] | auto]` allows sending a C++ client 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>`: downloads an identified analysis into a results directory. `[-f]` forces download (without interactivity) and `-admin -p <password>` allows administrator to download results. Use `[-server <name>[:port]]` to select a specific Queue Manager. Use `[-v|version]` to indicate release number.
- `psqueue-kill <ID>`: kills an identified analysis.
- `psqueue-purge all|ended`: removes all or finished analyses in the queue.
- `psqueue-dump`: gives the list of all analyses in the queue associated to the default Queue Manager.
- `psqueue-move-down <ID>`: moves down an identified analysis in the queue.
- `psqueue-remove <id>`: removes an identified analysis in the queue.
- `psqueue-get-qm-server`: gives the name of the default Queue Manager.
- `psqueue-progress <ID>`: gives progression of the currently identified and running analysis. `[-open-launcher]` displays the log in PolySpace launcher graphical user interface. `[-full]` gives full log file.
- `psqueue-set-password <old password> <new password>`: changes administrator password.
- `psqueue-check-config`: checks the configuration of Queue Manager. `[-check-licenses]` checks for licenses only.
- `psqueue-upgrade`: allows upgrading a Client (see `PolySpace Install Guide` in the `<PolySpaceCommonDir>/Docs` directory). `[-list-versions]` gives the list of available releases for upgrade. `[-install-version <version number> [-install-dir <directory>]] [-silent]` allow to install an upgrade in a given directory potentially in silent mode.

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

5.4. Share analysis between account

• `analysis-key.txt` **file**

For security reasons, all analyses spooled are owned by the user who sent them.
Each analysis has a unique crypted key.

The public part of the key is stored in a file named `analysis-keys.txt` and 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 (`^t means` tabulation):

```
<ID of launching> ^t <server name of IP address> ^t <public key>
```

Example

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

When attempting to manage (download, kill and remove, etc.) a particular analysis, the Queue Manager will examine this file and check the associated public key before allowing the action.
If the key does not exist, an error message appears: "`key for analysis <ID> not found`".

So, if user A wants to manage (for example download results of) an analysis sent by user B, user A should edit his own `analysis-key.txt` file and add into it the line corresponding to that analysis in the `analysis-key.txt` file of user B.

• Sharing analyses between projects with a magic key

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

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

All analyses spooled will use this key instead of random one. This would allow any user that has this key in his `analysis-key.txt` file to manage all analyses sent with the magic key.

Note: The magic key only works for analyses launched after it has been set up. Analyses sent before, will keep their auto-generated random keys.

6. Summary

After having followed each steps of this tutorial, you are now able to launch an analysis using PolySpace Client using a class by class analysis and explore some results with PolySpace Viewer. All theses activities can be performed locally on your desktop PC or in a Client/Server architecture.

You will find more information on advanced options in "`PolySpace C++ documentation.pdf`" which is available on the PolySpace CD-ROM or by clicking on ? in PolySpace tools.

PolySpace
TECHNOLOGIES