PolySpace™ for Ada

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

4. Working with results review

4.1. Basics: prerequisite being able to review PolySpace results

4.2. Automatic Methodology

4.3. How to find a maximum number of bugs within an hour reviewing oranges: selective orange review

4.4. Coloured source code for Ada

1. PolySpace documentation set

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

Note that this document covers both **Ada83** and **Ada95** language. In the following it only refers to **Ada** language. Also, when the document invokes `polyspace-ada` command, you have to refer to `polyspace-ada95` command with same characteristics.

Are you looking to analyse:

- **One package?**

  ○ Do you want to perform your first analysis and results review?

  ○ Is it possible for you to restrict data (functional) ranges in the package?

  ○ 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?

    ■ Do you want only to review orange using assistant mode, expert mode or given by the automatic methodology?

    ■ Reliability? Do you want to examine every result PolySpace provides?

    ■ Or do you want to find a compromise between productivity and reliability?

- **Multiple packages?**

  ○ Do you have issues related to:

    ■ Analysis launching (setup)?

      ■ Common setup issues

      ■ Advanced setup

    ■ Multitasking issues?

    ■ Shared variables?

  ○ Do you want to find bugs efficiently in the results?

○ Does your analysis takes place <u>on a server</u>, and do you want access the queued analysis?

Detailed contents

- PolySpace Installation. Please refer to `PolySpace_installation_guide.pdf` and `PolySpace_Licenses_installation_guide.pdf` located on the CD-ROM (in `<CD-ROM>\Docs\Install`) and in the `<PolySpaceCommon Dir>/Docs`.

- "<u>Setting up an analysis</u>" 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).

- "<u>Reviewing results</u>" 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 with the <u>automatic methodology</u>, and explaining how to find bugs efficiently.

- "<u>PolySpace and your development process</u>" gives guidance in the use of PolySpace as an integral part of the development process. It is presented as a narrative, and will help proficient users of the tool to get the best possible use from it. It presents different development processes, and shows how PolySpace might best be integrated in each case.

- "<u>Advanced</u>" includes multitasking information for PolySpace Verifier, hints and tips for quicker PolySpace Verifier analyses, and a complete description of those features which are used in order to launch a PolySpace analysis.

2. Getting started

Related subjects :

2.1. General Requirements
Computer Configuration

Please refer to PolySpace installation manual for the minimum hardware requirements to follow step by step this tutorial on a Windows PC.

Timing:

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

`\PolySpace_Install_Guide.pdf`).

The first step of this tutorial takes about 15 minutes.

The second step of this tutorial takes about 15 minutes.

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 analysing single files. Note that this module is available with the icon "`PolySpace Launcher`".
2. *PolySpace Server* for multi-file or composite analysis. Note that this module is available with the icon "`PolySpace Launcher`".
3. *PolySpace Viewer* is the graphical user interface to explore the results computed by PolySpace Server or PolySpace Client.
4. *PolySpace Spooler* is the graphical interface to manage analysis sent in remote.

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

Structure of this document

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

PolySpace Spooler Shortcut 2 KB   PolySpace Viewer Shortcut 2 KB   PolySpace Launcher Shortcut 2 KB

This Getting Started will focus on the following three exercises using PolySpace Client, the Viewer and the launching of an analysis remotely:

In Step 1 we will analyze a simple package "`example`" by using PolySpace Client

In Step 2.we will review the results obtained during Step 1 by using PolySpace Viewer

In the last step, instead of performing a PolySpace Client or Server analysis locally, analysis will be sent remotely to a server.

2.2. Step 1: PolySpace Client - Setting up and launching an analysis of a single Ada file

This paragraph describes a basic file analysis. It focuses on the analysis of the "`example`" package, which is included in the PolySpace installation directory and located at:
        `<PolySpaceInstallDir>\Examples\Demo_Ada\sources\example.adb`.

The PolySpace analysis process is composed of three main phases:
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: **Ada Standard compliance**, **portability** and **maintainability**.
Then, PolySpace seeks the main procedure. If none is found, PolySpace Client will generate one automatically. This function will call all the functions which are declared in the specification of the package.
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 products and its related license file and dongle correctly installed;

Source code files (in this case "`example.adb`") and all others specifications that it may directly or indirectly requires.

2.2.2. Setting up a PolySpace Client analysis

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

PolySpace Launcher
Shortcut
2 KB

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

PolySpace Client / Server Launcher

Select a product
● Client Launcher
○ Server Launcher

Select a language
○ C
○ C++
● Ada83/Ada95

OK   Cancel

The language to select depends on available installed PolySpace products.
The Graphical Interface of PolySpace analysis Launcher is displayed as below after having chosen Client Launcher and Ada95:
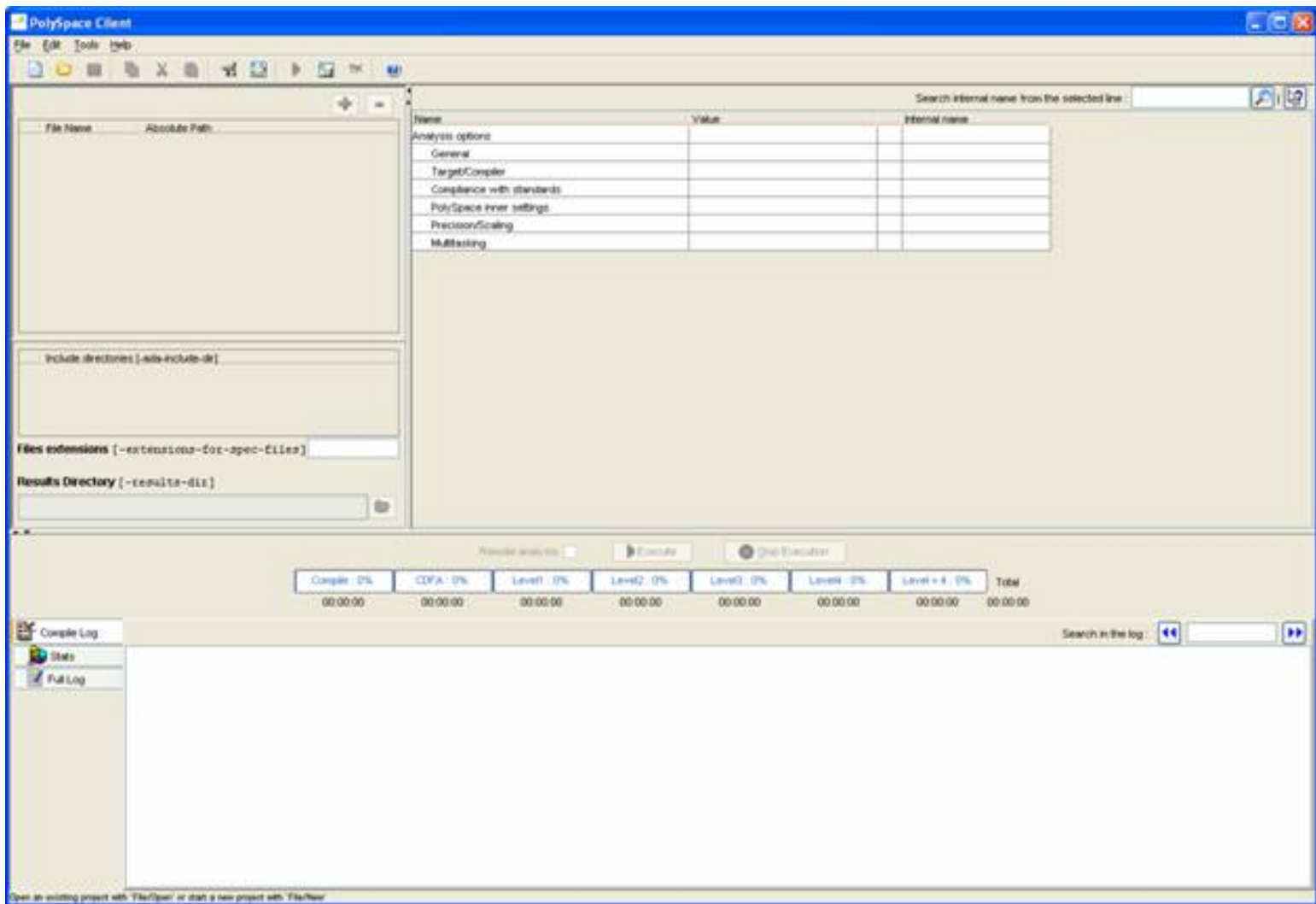
? Click on `File/New Project` to start an analysis:
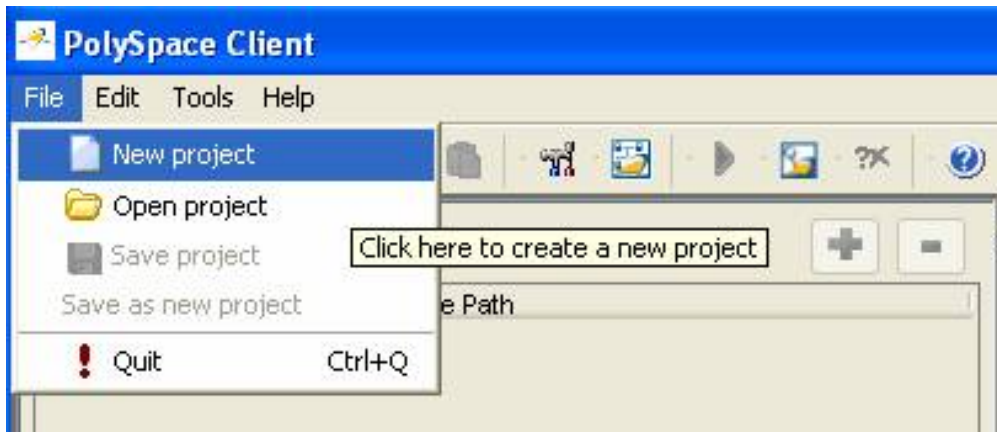
? If required, Select Ada95 as the language and click on "OK".

The PolySpace Client for Ada95 New Project window opens (see figure below). It contains four sections:

1. At the very top, the title bar, which contains usual icons and menus;
2. Top left is the list of files to analyze, along with include and results directories;
3. Top right is the set of options associated with the analysis that will be processed;
4. Finally the bottom area allows following the execution and progress of the analysis.
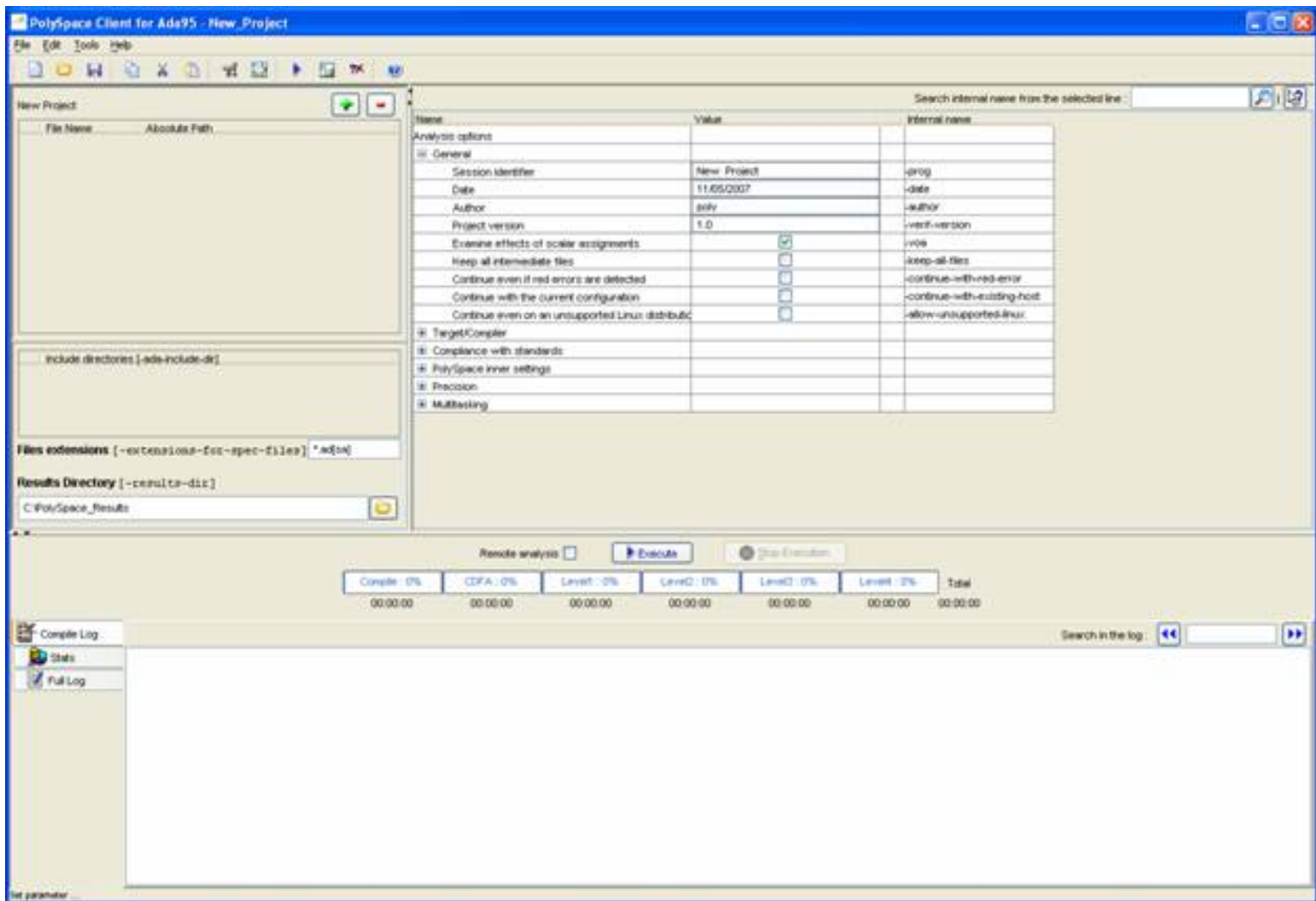
? Start by updating the result directory name by clicking on the browse button :
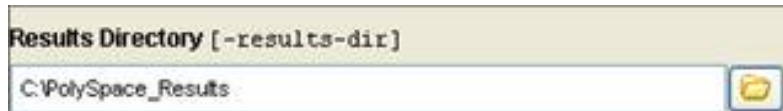
This directory is the one where PolySpace Client will store the results of the analysis. By default, PolySpace will store results in "C:\PolySpace_Results". This is the directory that we will choose for the analysis.

? 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.
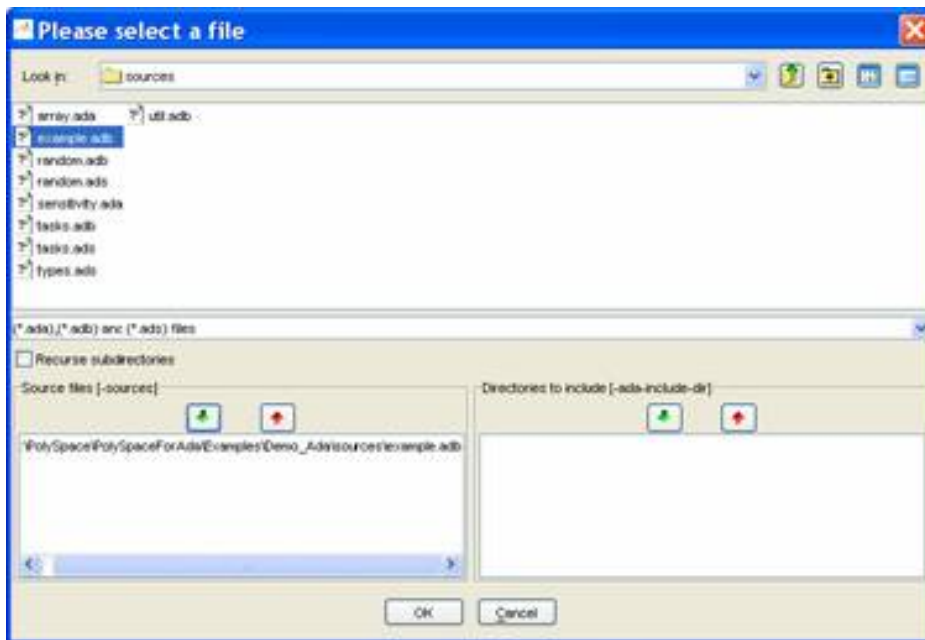
? In the "Look in" section, click on ⌄ and select "<PolySpaceInstallDir>\Examples\Demo_Ada\sources". A list of files appears in the box (<PolySpaceInstallDir> corresponds to C:\PolySpace\PolySpaceForAda in the figure above).

? Select "example.adb" and click on ⬇ in the "Source files [-sources]" section (bottom left) of the window. The file is now listed among the source files to be analyzed.
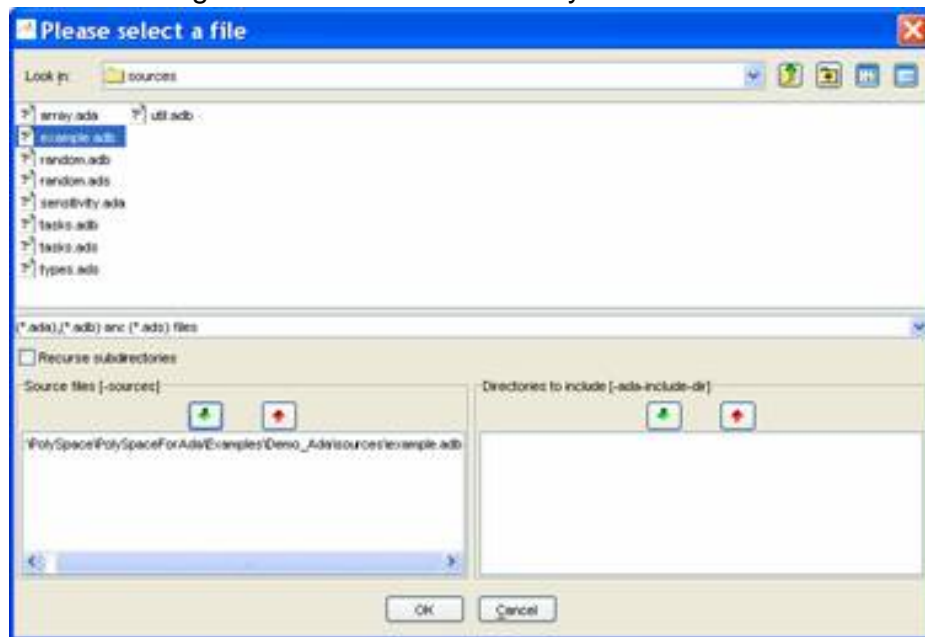
? Click on ⟨ OK ⟩ to go back to the "PolySpace Client for Ada95 – New_Project" window.

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.

2.2.3. 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 "demo" and save it under PolySpace result directory. The full name of that file will be "demo.dsk".
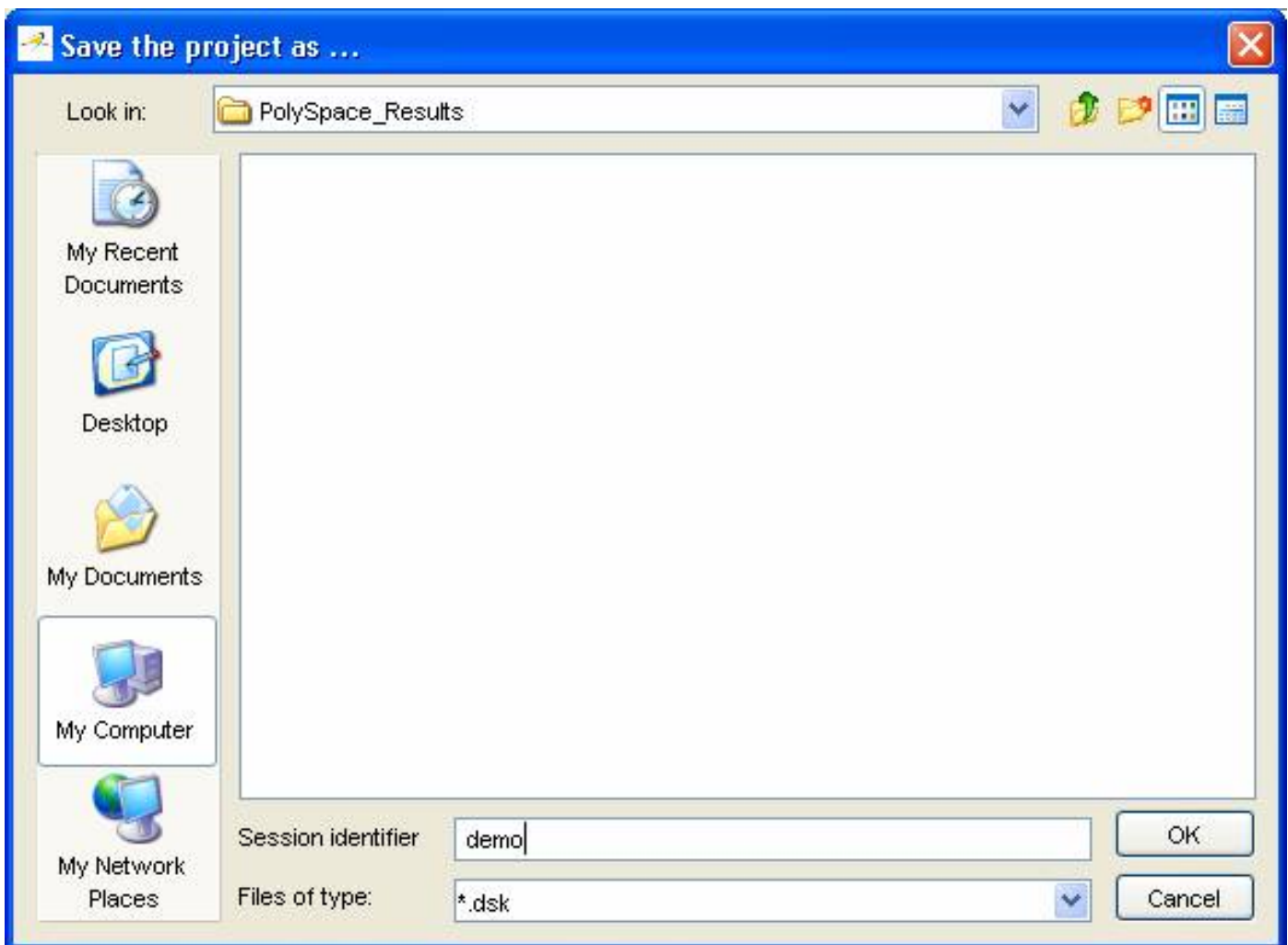
? Click on ＯＫ to go back to the "PolySpace Client for Ada95 – New_Project" window and click again on ▶ Execute to proceed forward.
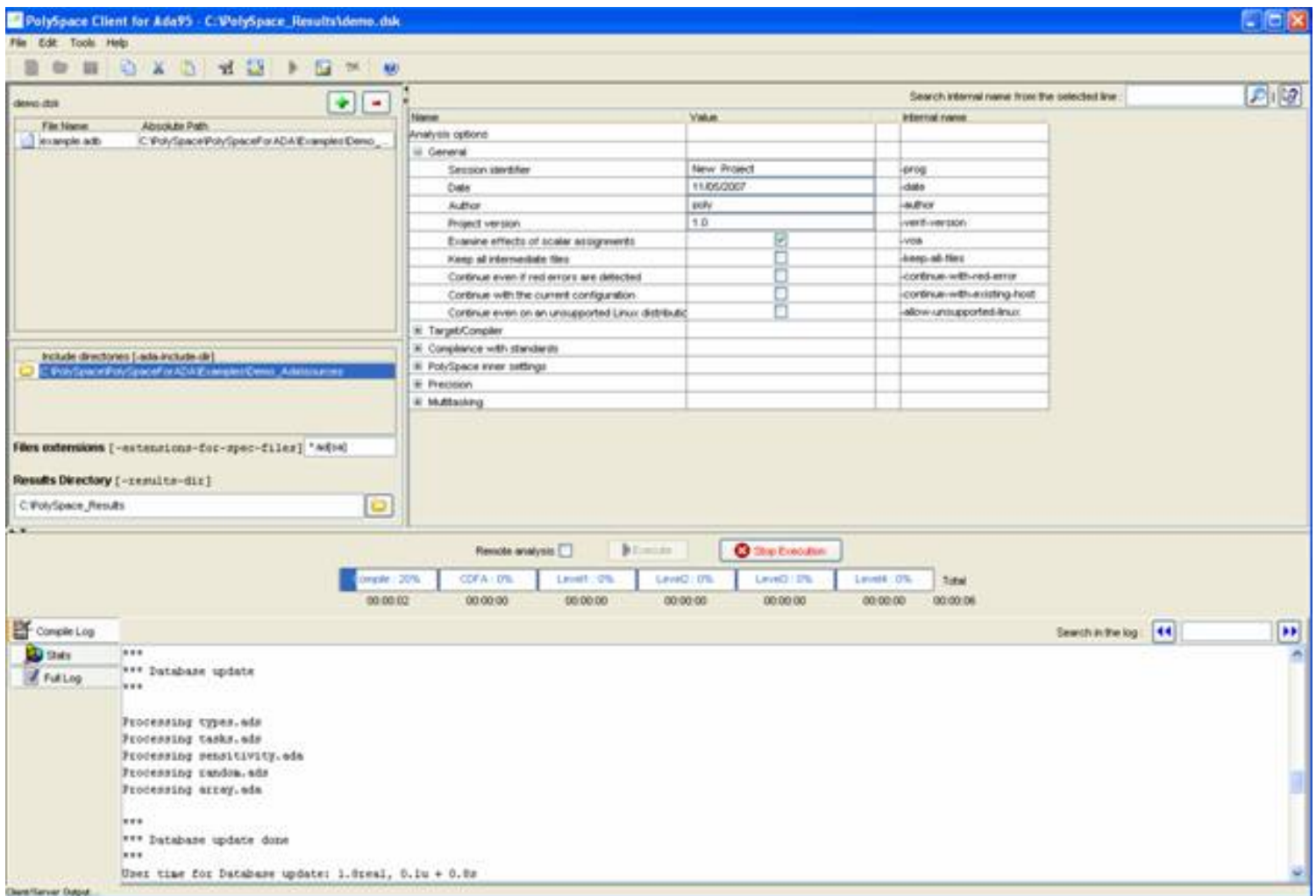
A progress report is displayed in the bottom part of the graphical interface, indicating that the analysis is being

performed. The [▶ Execute] button is also grayed out.

Note: you may press the Stop Execution button - [❌ 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

After some checks, PolySpace will show an error message:
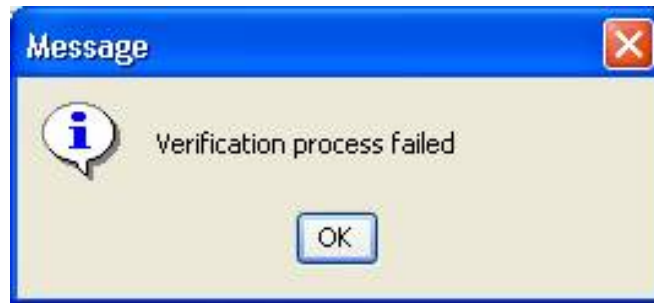
Let's try and understand why we get this error message.

First possible cause for the error message: Hardware recommendation

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

? 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:

? Check the box in the "Value" column that is associated to the "`-continue-with-existing-host`" line as shown below.

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

| Continue even if red errors are detected | ☑ | -continue-with-red-error |
| Continue with the current configuration | ☑ | -continue-with-existing-host |

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

Another cause of error may be that PolySpace Desktop misses some package specifications.
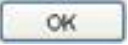
In the tutorial, as shown above, some specification are missing: "`types`", "`sensitivity`", "`pkdata`", "`runtime_error`" and "`random`". To fix theses compilation errors, you need to indicate where to find these specifications. As PolySpace is not associated with one particular compiler, it is mandatory to indicate where library files are stored.

In our "`example.adb`" file analysis, the related specifications are located in the same directory as the `adb` file: `<PolySpaceInstallDir>\Examples\Demo_Ada\sources`.

? Open the "`Please select a file`" window by using button (right of the "`demo.dsk`" label in the top right of the interface):
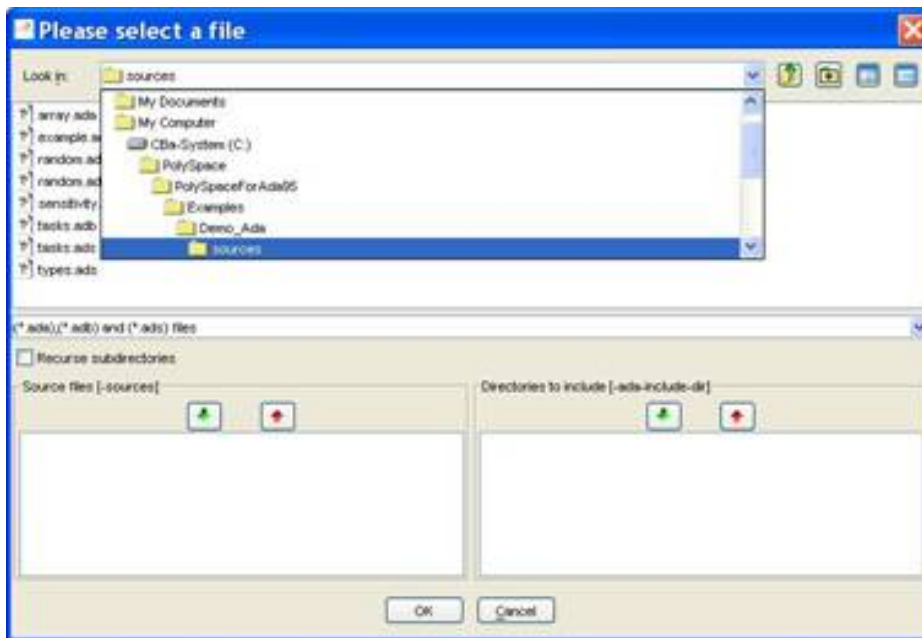
? Select "`<PolySpaceInstallDir>\Examples\Demo_Ada\sources`", where the specifications are located.

? Click on in the "`Directories to include [-ada-include-dir]`" section, and then close the window using .

Notes: All specifications are in this folder only. It is also possible to drag a directory and drop it directly in the "`include directories [-ada-include-dir]`" part (top left of PolySpace Client) without using the "`Please select a file`" window.

At the end, a last compilation error remains:

It means that "pkutil" specifications are missing even the `-ada-include-dir` directory added just before. Searching for "`pkutil`" specifications, in the "`sources`" directory, we can see that it is defined in the "`util.adb`" file. Changing the following option "`Files extension`" by "`-extensions-for-specs-file` `*.ad[sab]`" allows to indicate that specifications can be found in `*.ada`, `*.ads` and `*.adb` file extensions.

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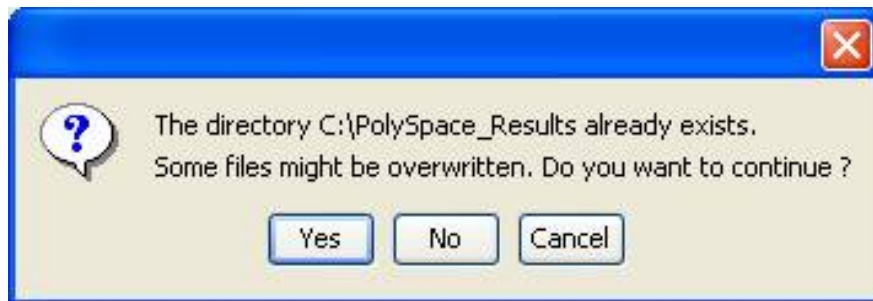
In our example, this is what we want to do. Click on [Yes], if it happens.

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 to follow the progress of the analysis:

Compile : 100%	CDFA : 59%	Level1 : 0%	Level2 : 0%	Level3 : 0%	Level4 : 0%	Total
00:00:08	00:00:12	00:00:00	00:00:00	00:00:00	00:00:00	00:00:26

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:
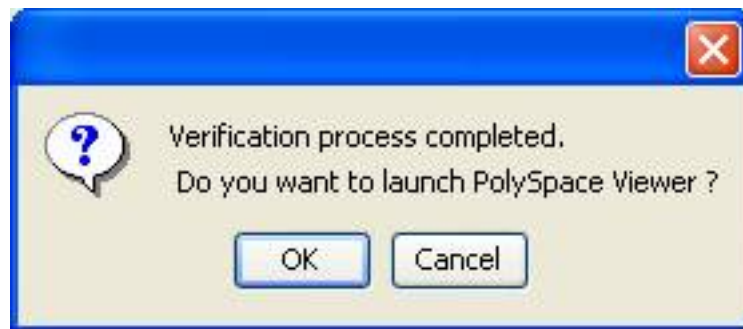
If you Click on "OK", go to the next section of the tutorial to view the results. If you click on Cancel, and if no other analyses are running, you can access the results via the icon in the title bar.

2.3. Step 2: 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 "`example.adb`" performed during Step 1 using the following icon:

If the [OK] button has been clicked at the end of the previous analysis (see previous section), PolySpace Viewer automatically opens results. Please go directly to *"analysing of PolySpace results"*.

Related subjects :

    2.3.1. Modes of operation

    2.3.2. Downlaod results into the Viewer

    2.3.3. Analyzing of PolySpace results in "Expert" mode ("example.adb")

    2.3.4. Methodological asssitant

    2.3.5. Report Generation

2.3.1. 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.
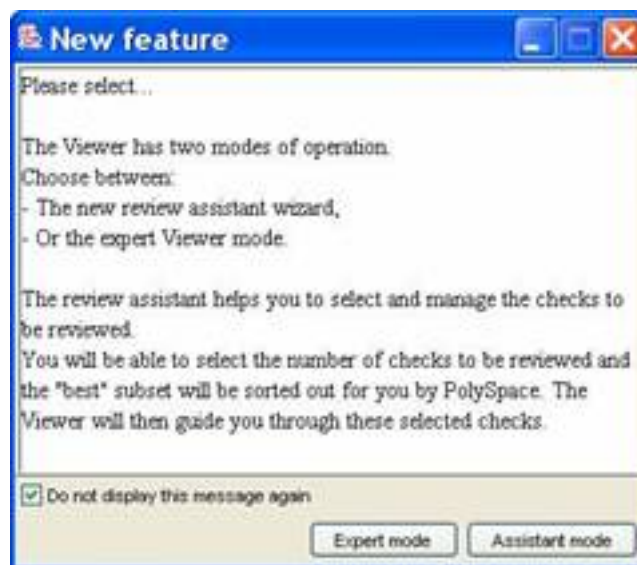
The mode will define the reviewing process of checks highlighted during an analysis:
- In "`Expert mode`": The Viewer is opened in a mode where all checks can be seen. The number, the order and the categories of checks can be reviewed can be chosen by the user himself (See next section).
- In "`Assistant mode`": the reviewing rules for an Ada 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. First selected checks concern the Automatic Methodology.

? 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.

2.3.2. Downlaod results into the Viewer

After having clicked on "`Expert mode`" the PolySpace Viewer window looks like the figure below:
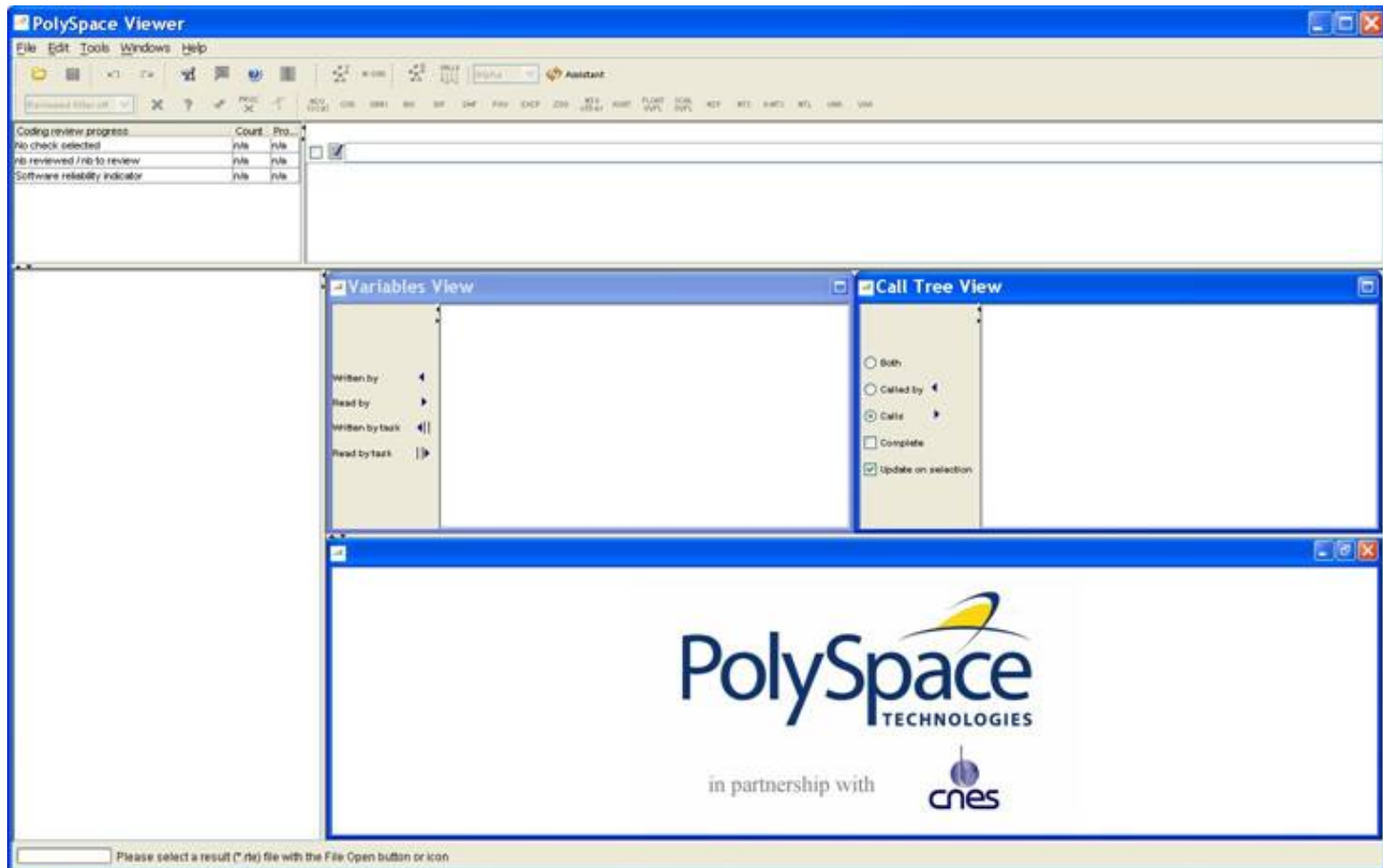
? Click `file>open` to load result files. If you did not perform the analysis, you can still review the results by opening the following file:
`<PolySpaceInstallDir>\Examples\Demo_Ada\RTE_px_O2_Demo_Ada_LAST_RESULTS.rte`

? Using the "`File>Open`" menu, select the following file located in "`C:\PolySpace_Results`".
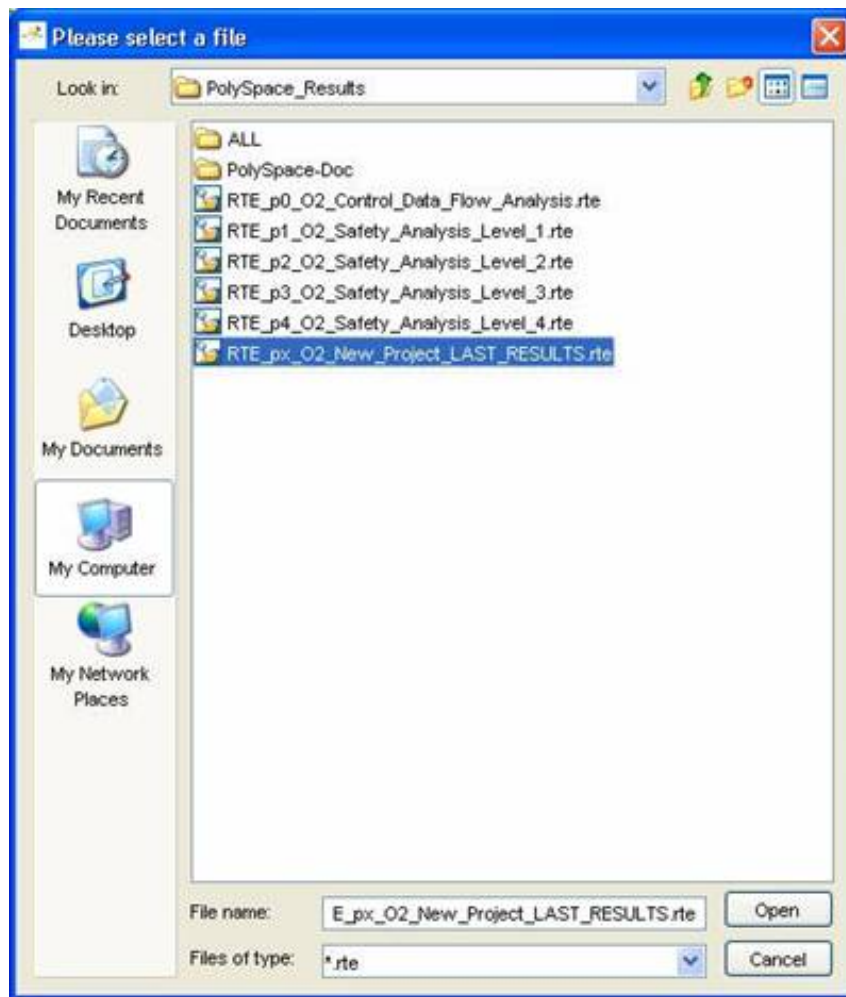
? Then click on [Open] to proceed with further steps

Note: The `RTE_px_O2_Demo_Ada_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.

2.3.3. Analyzing of PolySpace results in "Expert" mode ("example.adb")

After loading the results, and PolySpace Viewer window looks like below:
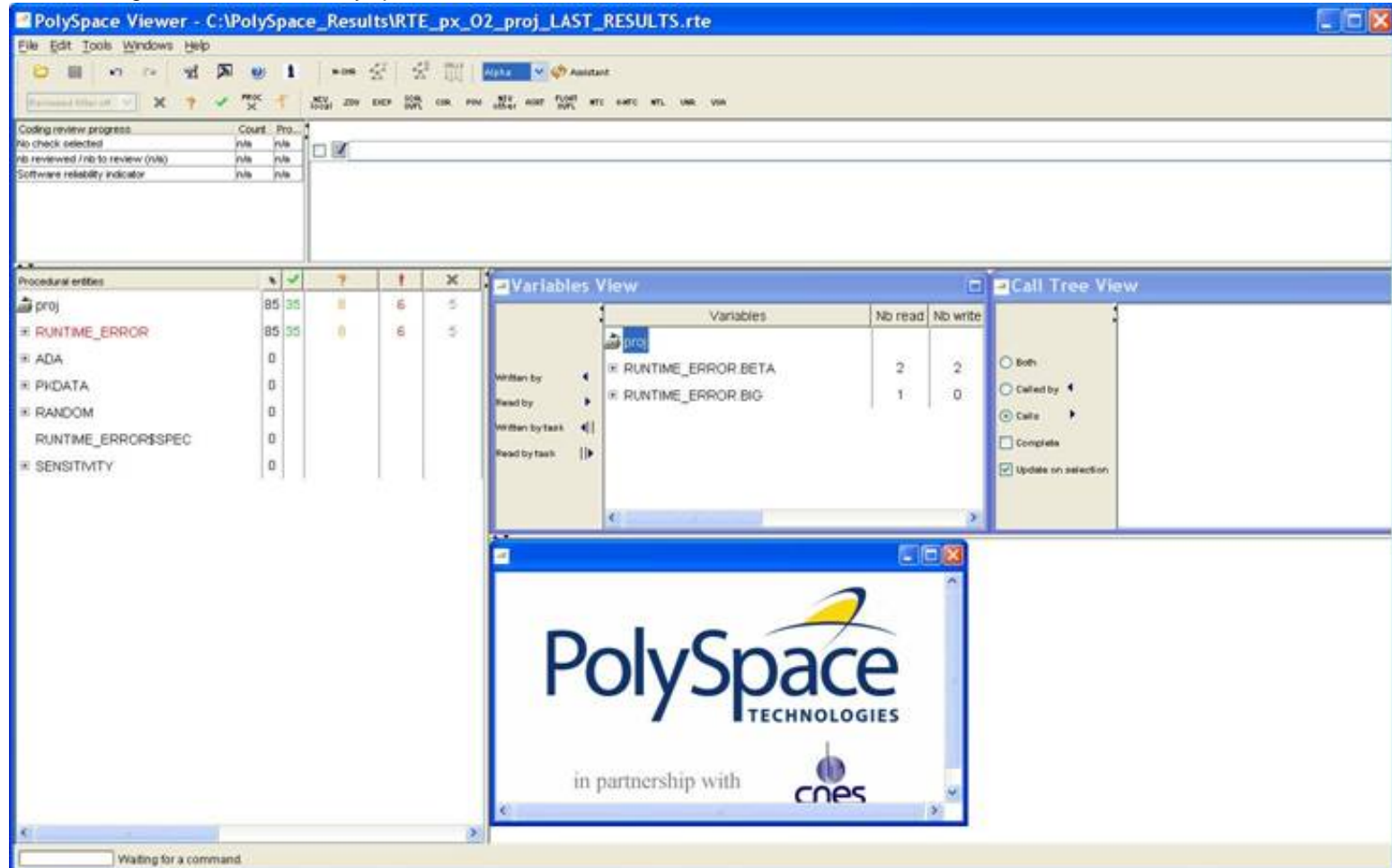
1. On the left is the Procedural entities view (or `RTE` view). It displays the list of packages which have been analysed or used during the analysis (specifications).

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):

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:
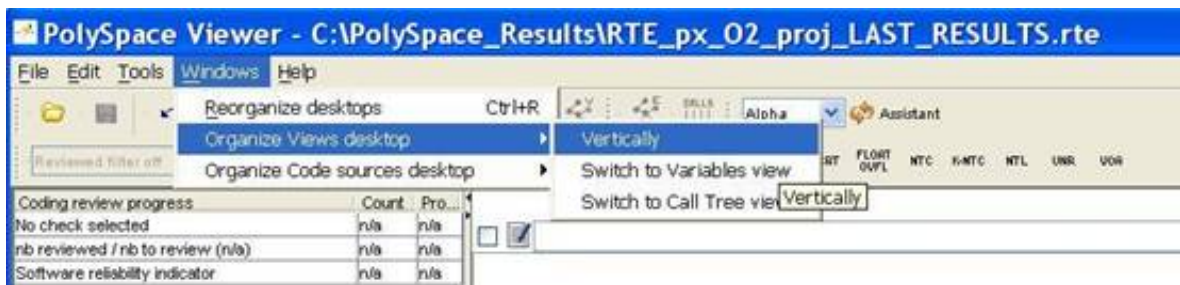
Related subjects :

2.3.3.1. Procedural entities view (RTE View)

Each package and underlying functions in the RTE view is colorized according to the most critical error found:

- In black color: The packages specification has been used to perform analysis
- In red color: The package is red; one or more *definite* run-time errors have been found in it.

? Click once on the ⊞ left of "RUNTIME_ERROR" to find out more about this package.

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

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

The ⚲ column indicates the reliability of the code (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 the analyzed package.

First example of runtime error found by PolySpace: Memory Corruption

❓ Click on ⊞ to expand "INFINITE_LOOP " 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 "INIFINITE_LOOP" function inside package RUNTIME_ERROR.
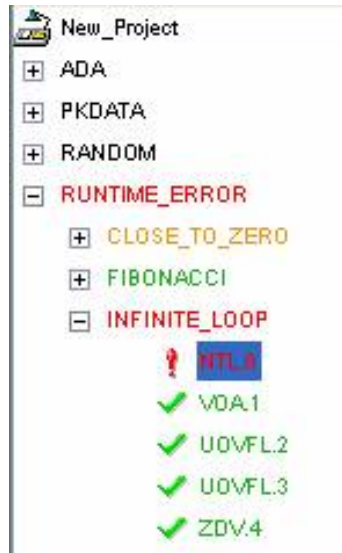
❓ Click on the red "NTL.0" item - which stands for **N**on-**T**ermination of **L**oop -, to precisely locate this error in the source code. The bottom right section is updated showing the location of the "NTL.0" item.
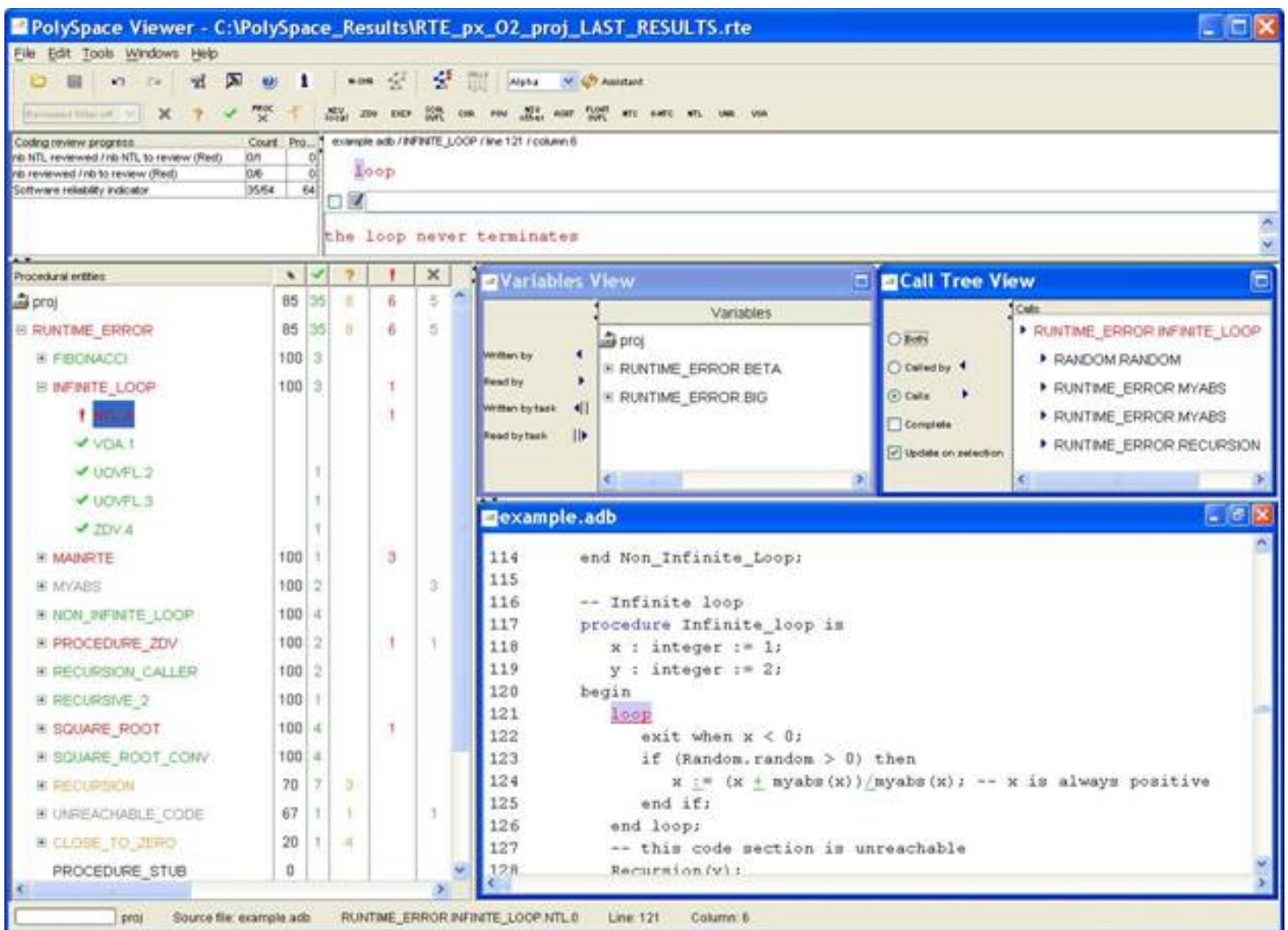
? Click on red loop in the source code at line 121. An error message is opened:
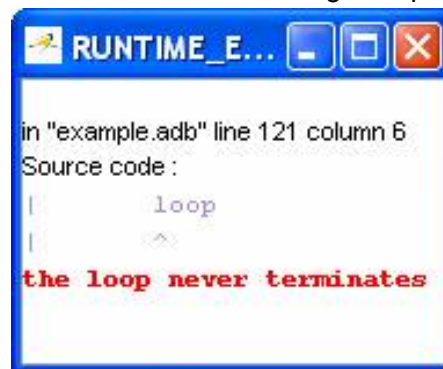
Indeed, the condition to exit the loop is that "x" becomes a negative value.


```
121          loop
122              exit when x < 0;
123              if (Random.random > 0) then
124                  x := (x + myabs(x))/myabs(x); -- x is always positive
125              end if;
126          end loop;
```

But according to the formula, "x" will always be a positive value. So, the loop cannot terminate.

Second example of runtime error found by PolySpace: Unreachable code

? Select "UNREACHABLE_CODE" in the RTE View. You can see that the division "z := x / y" is unreachable (gray colour on the check) because of the non satisfied Boolean condition: "x" is never negative when evaluating "x<0".

PolySpace has detected some dead code.

```
169        -- Here we demonstrate PolySpace Verifier's ability to
170        -- identify unreachable sections of code due to the
171        -- value constraints placed on the variables.
172     procedure Unreachable_Code is
173        x : integer := Random.random;
174        y : integer := Random.random;
175        Z : Integer;
176     begin
177        if (x > y) then
178           x := x - y;
179           if (x < 0) then
180              Z := x / Y;
181           end if;
182        end if;
183     end Unreachable_Code;
184
```

2.3.3.2. 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.
- 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". In the next figure (see next paragraph), `Beta` is an example of this kind of global variable.

2.3.3.3. 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.

In an First example of the second set: Arithmetic error

? Click on ⊞ to expand "SQUARE_ROOT" function. You can see the source code view in the bottom right.

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

"SQUARE_ROOT" is called by MAINRTE function. It is displayed in the "*Call tree view*" window (right of the top right section).

"SQUARE_ROOT" calls "RANDOM.random" (automatically stubbed function),

"SQUARE_ROOT_CONV" (from RUNTIME_ERROR package) and "SQRT" (from the standard library).

```
151       -- The table provided below the example shows the domain of
152       -- values for the expressions in the example.
153       procedure Square_Root_conv (alpha : in float; y : out long_float) is
154       begin
155           y := (1.5 + cos (long_float(alpha)))/5.0;
156       end Square_Root_conv;
157
158       Beta : Long_Float;
159       procedure Square_Root is
160           Alpha : Float := Random.random;
161           Gamma : long_float;
162       begin
163           Square_Root_conv (Alpha, Beta);
164           Beta := Beta - 0.75;
165           Gamma := sqrt(Beta);   -- always sqrt(negative number)
166       end Square_Root;
167
```

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

Note: using -voa option at launching time, PolySpace can help more suitably by giving information of

range on scalar assignment

Second example of the second set: Non-Infinite loop

? Select "NON_INFINITE_LOOP" in 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.

```
104      procedure Non_Infinite_Loop (X : out Integer) is
105          cur : Integer :=0;
106      begin
107          X := 0;
108          loop
109              exit when x > big;
110              cur := cur + 2;
111              x := cur / 2;
112          end loop;
113          X := Cur / 100;
114      end Non_Infinite_Loop;
115
```

2.3.3.4. 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:
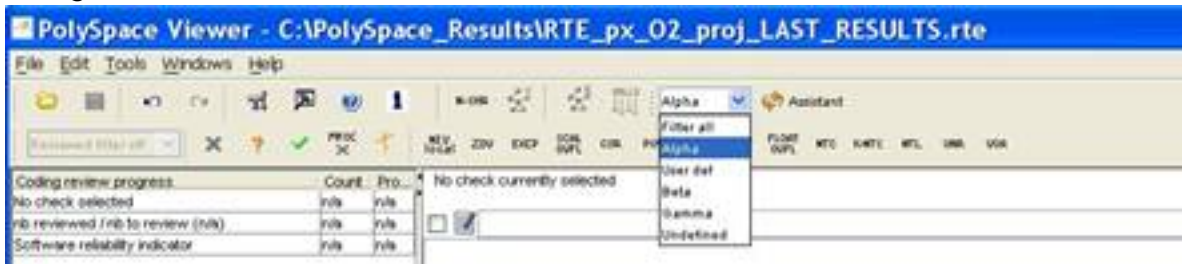
? To illustrate the use of these filters, we will focus on the Square Root function that we have examined in the previous section. Click on [Gamma] to reduce the information checks related to "SQUARE_ROOT".

This list of acronyms - for type of operations checked - shows what PolySpace automatically analyzed for you. 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 "RECURSION" in the "Procedural entities" view and then, click on ⊞ to get the list of the checks.

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:

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 [ZDV]. You will get list of checks containing only ZDV (**Z**ero **Di**Vision) reds, oranges or greens:
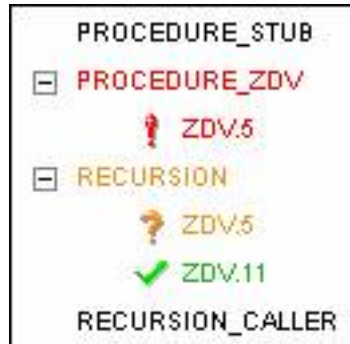
? Click on [✓] (top of the window) to suppress green code sections. You will get a reduced list of checks reds, oranges and grays:
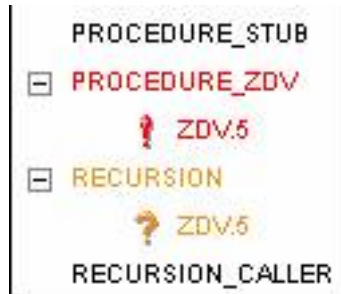
2.3.3.5. Miscellaneous

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").

2.3.4. Methodological asssitant

After a first navigation into the PolySpace Viewer, some simple questions remain:
- Do all checks need to be reviewed?
- What are the checks to review?
- How many?
- What is the best order?

The Methodological assistant is here to answer to all 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 by clicking on the upper right ✖ symbol, open it again, load same results and chose "Assistant" mode.

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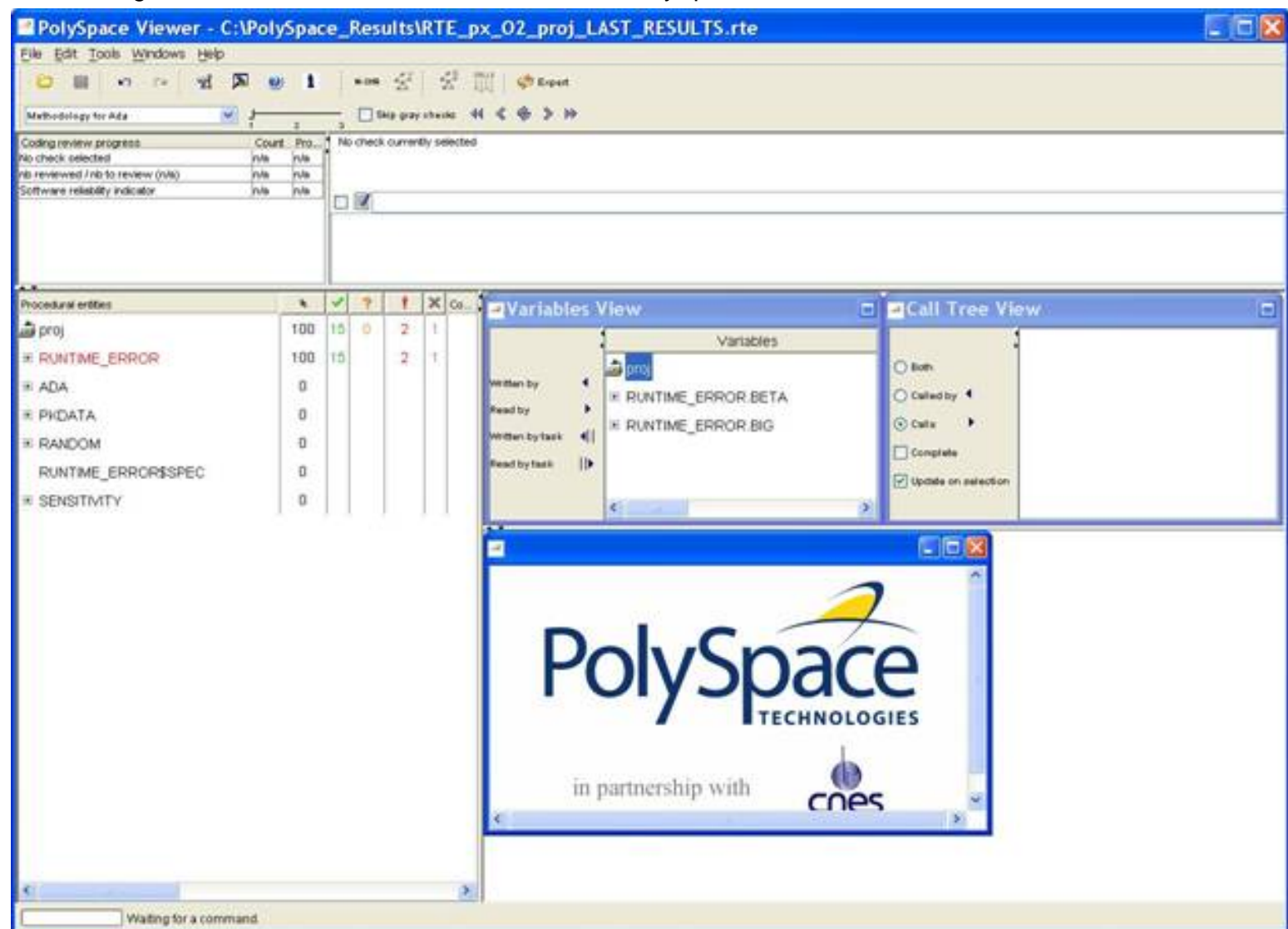
Related subjects :

2.3.4.1. 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:
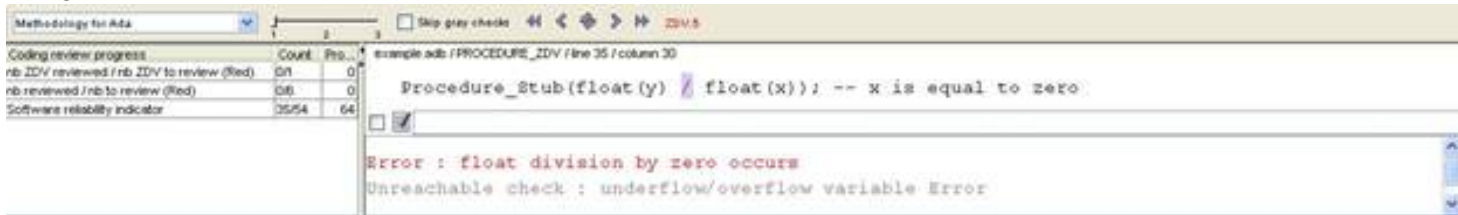
Some other changes can be seen in the viewer:

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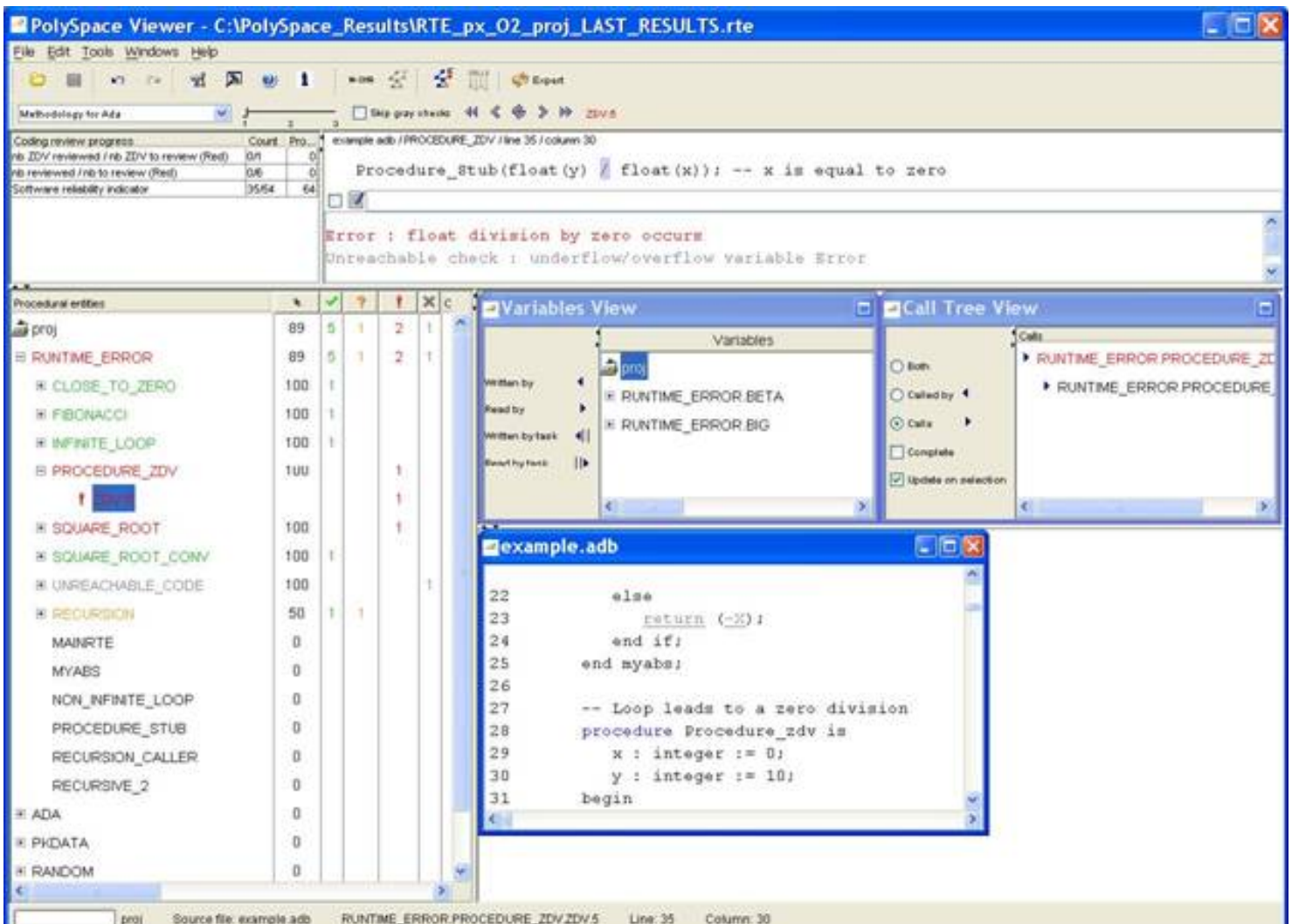
The Methodological dashboard gives details and allows reviewing the check. On the selected check, it is possible to mark the fact that it has been reviewed.

$?$ Tick the radio button box and type an associated comment in the associated edit box on the right.

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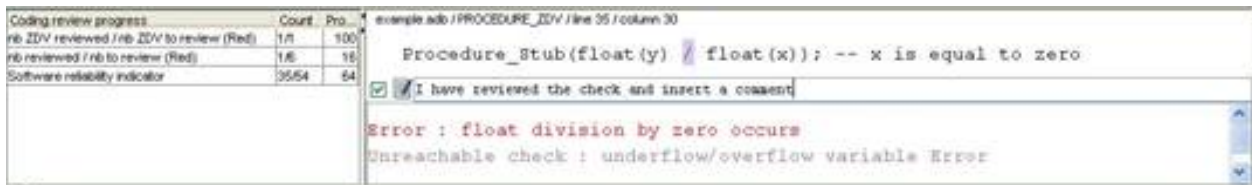
The left part of the dashboard has been updated, and displays some statistics in three lines:

- 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 unproven).
- Last line gives in permanence the Software reliability indicator.

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

 and going to next / previous category of reviewed checks selected by the Methodology.

2.3.4.2. Choose a methodological assistant
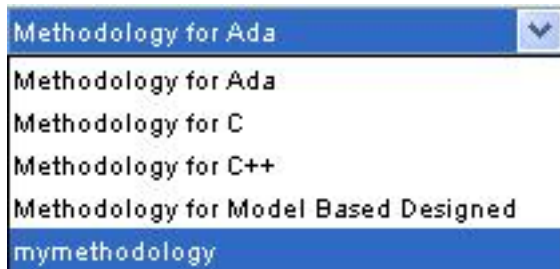
 and associated levels have been pre-selected by PolySpace.

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

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

It is possible to refine a self-created one or define its own Methodology. The "`Preferences PolySpace Viewer>Assistant methodology`" Tab is accessible from the "`Edit`" menu.
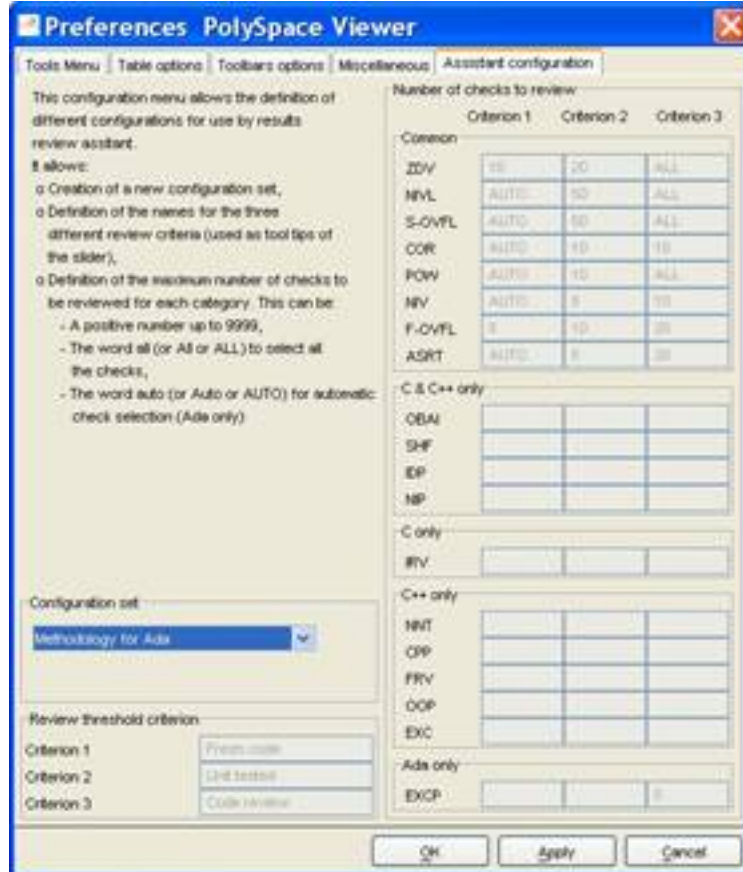
You can create a new configuration set and define for each criterion what will be the categories of check to review and how many in each one.

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

2.3.5. Report Generation

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

All views (except source code) are printable and can be exported to textual or Excel® format (protected by license).
The "`C:\PolySpace_Results\PolySpace-Doc`" directory should contain the following files:
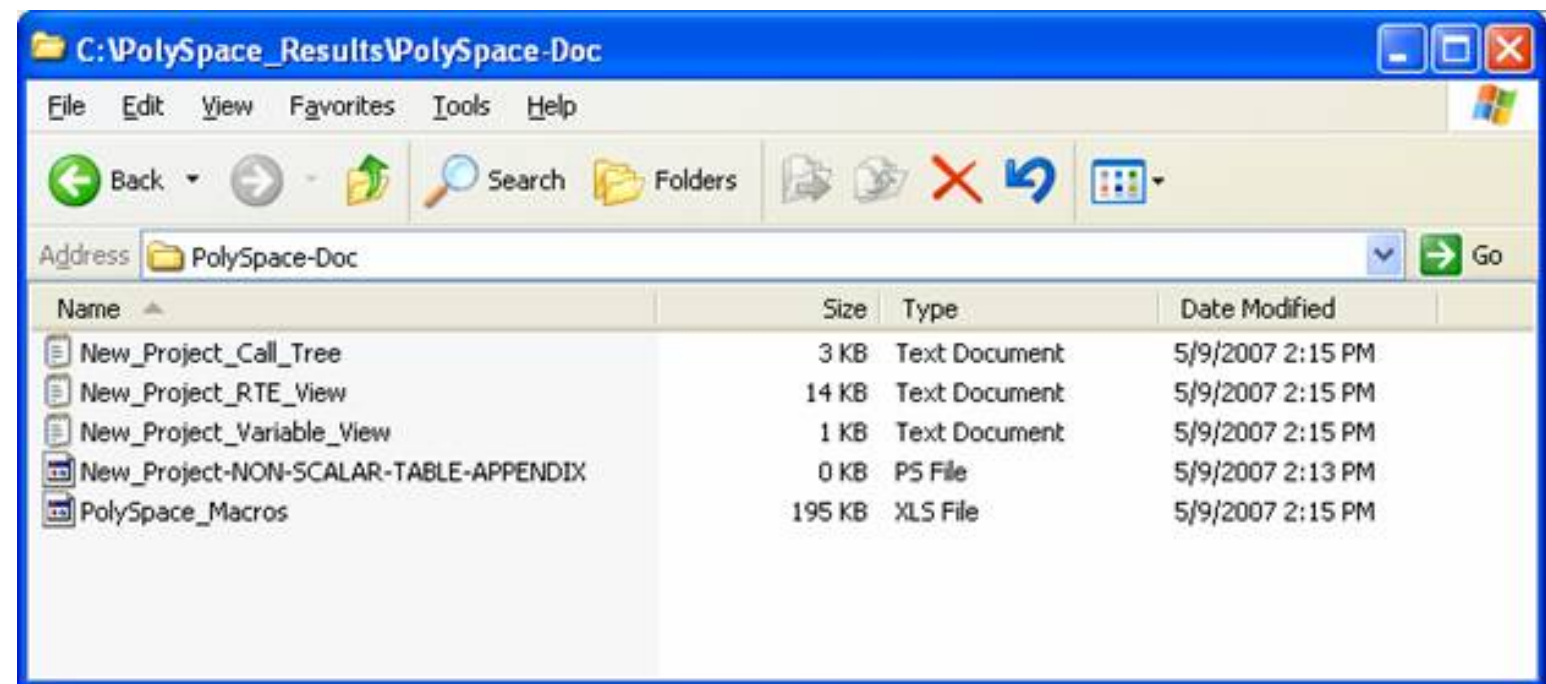
? Open the file called "`PolySpace_Macros.xls`", enable macros when asked and then the following window opens:

	A	B	C	D	E	F	G	H
1								
2		Copyright © PolySpace Technologies, 1999-2006						

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 Generate PolySpace Results Synthesis . A file browser opens. Select the file called "New_Project_RTE_View.txt" as shown below:
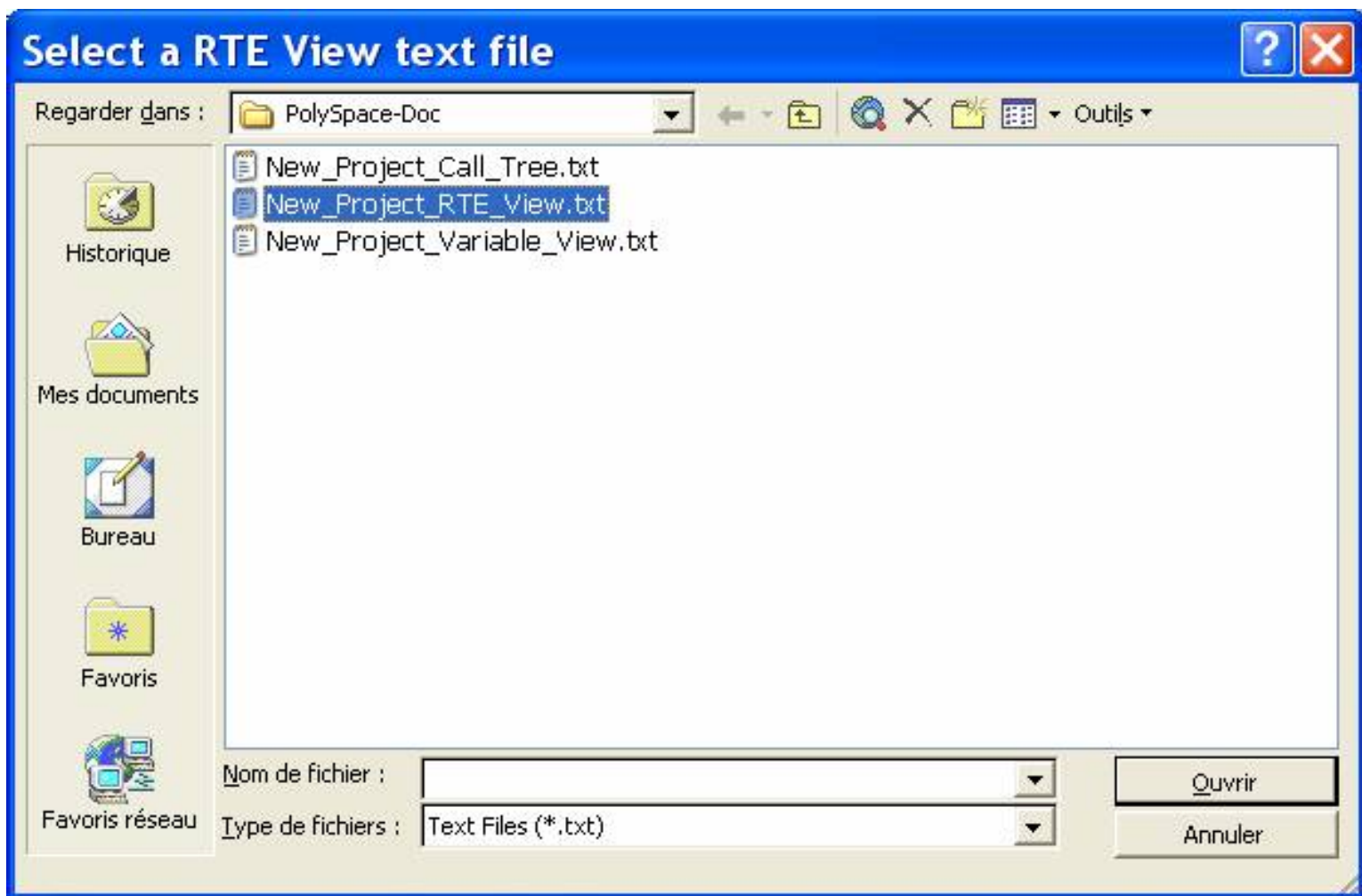
After a few seconds, an Excel® file is generated. It contains several spreadsheets related to the application analyzed.

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.

2.4. 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` or the `PolySpace Install Guide Manual`) to install and configure a Client and a Server.

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

Related subjects :

2.4.1. Steps of Launching

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

2.4.3. Batch commands

2.4.4. Share analyses between accounts

2.4.1. Steps of Launching

Only two simple steps:

? Step 1: set up an analysis as described in section "Step 1: PolySpace Desktop - Setting up and launching an analysis of a single Ada package" but do not launch it.

? Step 2: tick the "Remote analysis" radio button (see next figure) and click on [▶ Execute] to launch the analysis.
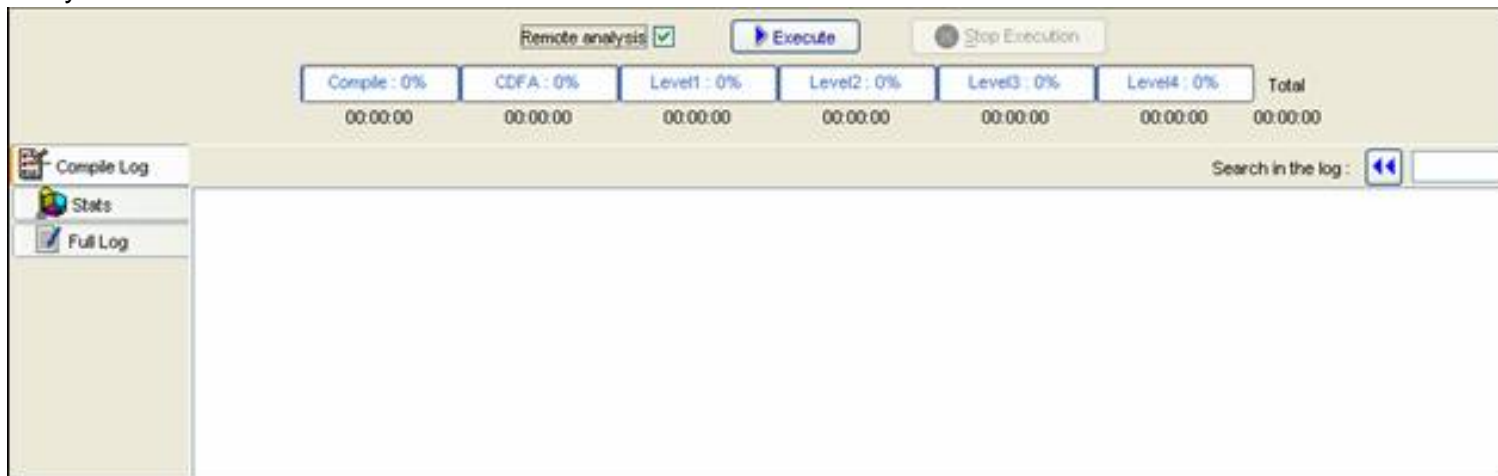
The analysis starts and the compilation phase is performed on the desktop PC. At the end of the "Compilation phase" the analysis is sent to the Queue Manager server. By clicking on the "Full Log" tab, you will have a message like this:
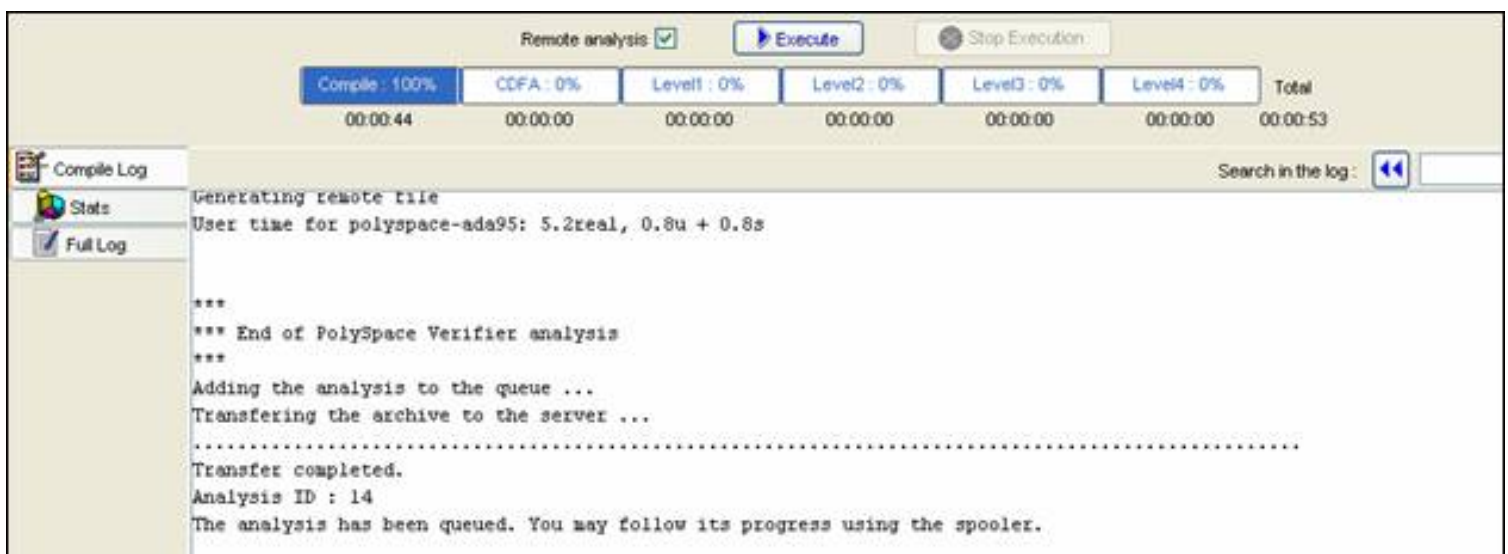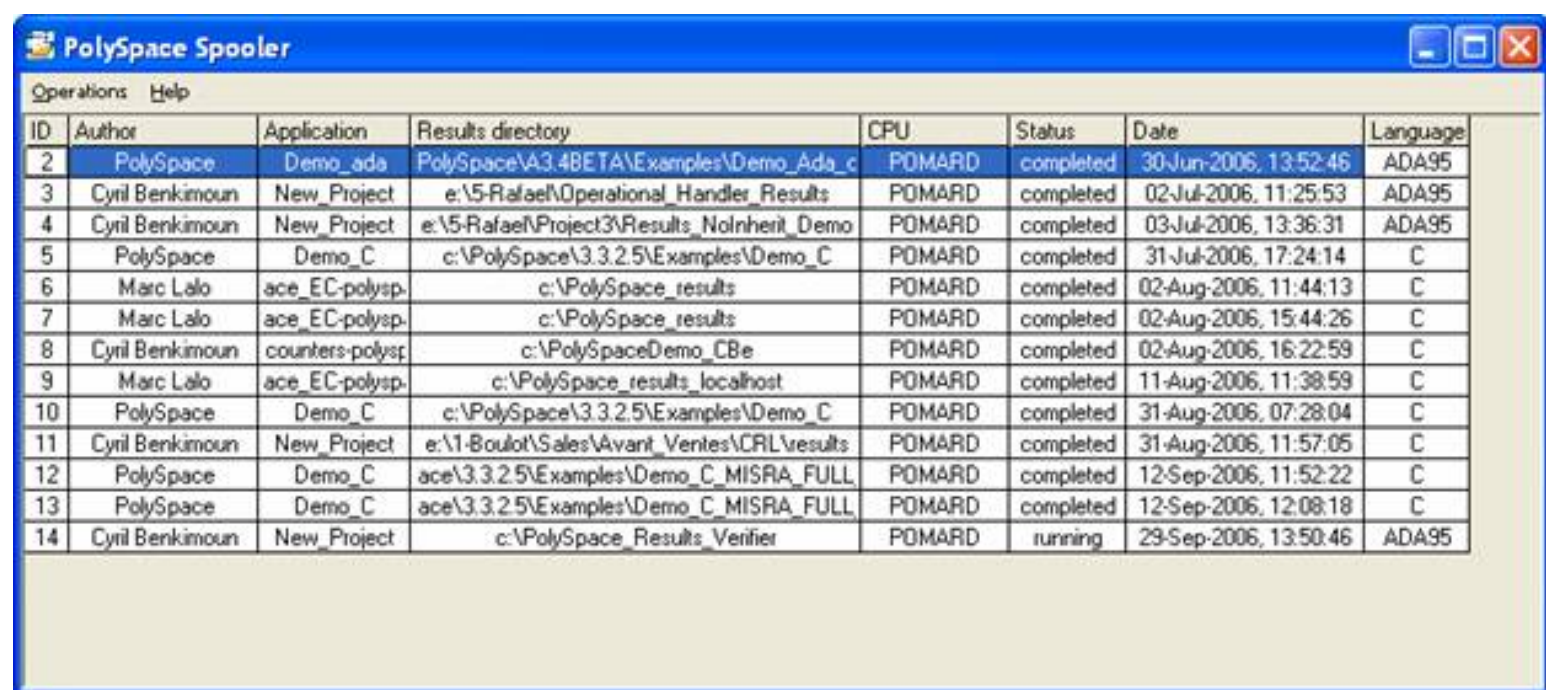
The analysis has been queued with an ID number, and you can follow its progression using the PolySpace Spooler. If you do not tick the "Remote analysis" radio button, the analysis continues locally.

2.4.2. 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

PolySpace Spooler
Shortcut
2 KB

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

ID	Author	Application	Results directory	CPU	Status	Date	Language
2	PolySpace	Demo_ada	PolySpace\A3.4BETA\Examples\Demo_Ada_c	POMARD	completed	30-Jun-2006, 13:52:46	ADA95
3	Cyril Benkimoun	New_Project	e:\5-Rafael\Operational_Handler_Results	POMARD	completed	02-Jul-2006, 11:25:53	ADA95
4	Cyril Benkimoun	New_Project	e:\5-Rafael\Project3\Results_NoInherit_Demo	POMARD	completed	03-Jul-2006, 13:36:31	ADA95
5	PolySpace	Demo_C	c:\PolySpace\3.3.2.5\Examples\Demo_C	POMARD	completed	31-Jul-2006, 17:24:14	C
6	Marc Lalo	ace_EC-polysp-	c:\PolySpace_results	POMARD	completed	02-Aug-2006, 11:44:13	C
7	Marc Lalo	ace_EC-polysp-	c:\PolySpace_results	POMARD	completed	02-Aug-2006, 15:44:26	C
8	Cyril Benkimoun	counters-polysp	c:\PolySpaceDemo_CBe	POMARD	completed	02-Aug-2006, 16:22:59	C
9	Marc Lalo	ace_EC-polysp-	c:\PolySpace_results_localhost	POMARD	completed	11-Aug-2006, 11:38:59	C
10	PolySpace	Demo_C	c:\PolySpace\3.3.2.5\Examples\Demo_C	POMARD	completed	31-Aug-2006, 07:28:04	C
11	Cyril Benkimoun	New_Project	e:\1-Boulot\Sales\Avant_Ventes\CRL\results	POMARD	completed	31-Aug-2006, 11:57:05	C
12	PolySpace	Demo_C	ace\3.3.2.5\Examples\Demo_C_MISRA_FULL	POMARD	completed	12-Sep-2006, 11:52:22	C
13	PolySpace	Demo_C	ace\3.3.2.5\Examples\Demo_C_MISRA_FULL	POMARD	completed	12-Sep-2006, 12:08:18	C
14	Cyril Benkimoun	New_Project	c:\PolySpace_Results_Verifier	POMARD	running	29-Sep-2006, 13:50:46	ADA95

When you select an analysis (right hand click), you can manage it in the queue:

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

- *View log file.* This action lists the associated log file in a "Command prompt" window, in which you can the last 100 updated lines of the log file in real time. This option is only available when the analysis is running.

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

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

- *Kill and download results.* This action stops the analysis definitively and the results are downloaded. The status of the analysis changes from "running" to "aborted". The analysis remains on the queue.

- *Kill and remove from queue.* This action stops the analysis definitively, and the analysis is removed from the queue. **Note: <u>The results will be lost</u>**.

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

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 queue manager password is required.
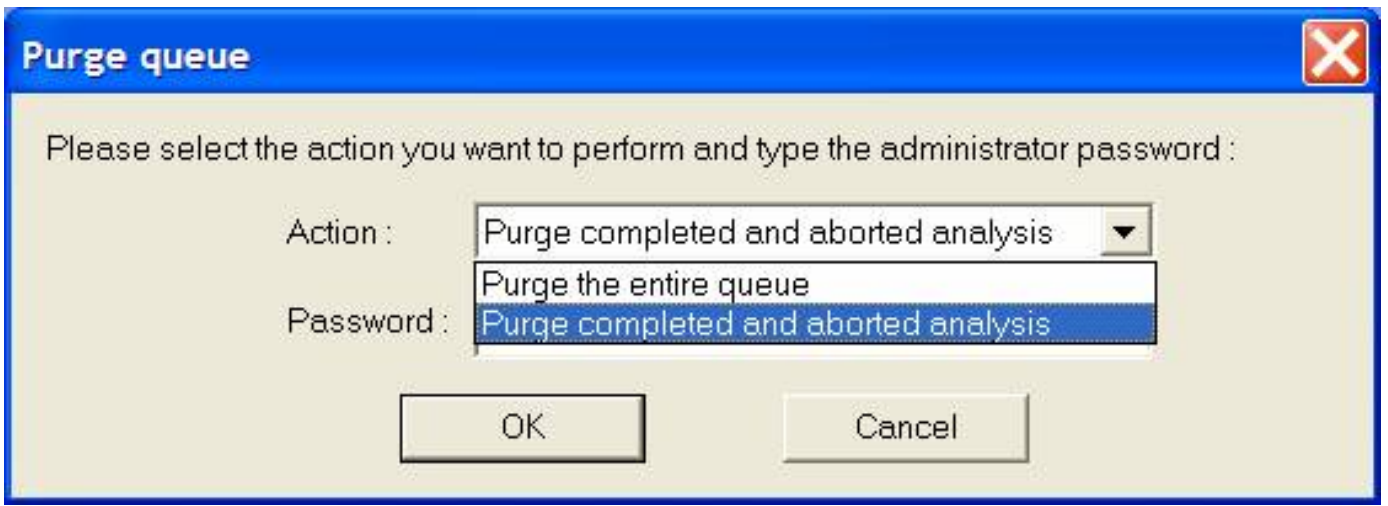
"`Operations>Change root password`". This action changes administrator password of the queue manager or the default one. **Note:** by default the password is "administrator".

2.4.3. 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 `<PolySpaceInstallDir>/bin/polyspace-remote-`: `polyspace-remote-ada95` and `polyspace-remote-desktop-ada95`.

They are equivalent to respectively the commands with a prefix `<PolySpaceInstallDir>/bin/polyspace-`. For example, `polyspace-remote-desktop-ada95 –server [<hostname>:[<port>] | auto]` allows the sending of a Ada desktop 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 command begin with the prefix `<PolySpace Common Dir>/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]` give full log file.
- `psqueue-set-password <password> <new password>`:   change administrator password.
- `psqueue-check-config`:   check the configuration of Queue Manager. `[-check-licenses]` check for licenses only.
- `psqueue-upgrade`:   Allow to upgrade a client side (cf. PolySpace Install

guide in the <PolySpace Common Dir>/Docs directory). `[-list-versions]` give the list of available release to upgrade. `[-install-version <version number> [-install-dir <directory>]] [-silent]` allow to install an upgrade in a given directory and in silent.

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

2.4.4. 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 stored 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>
```

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

Example:

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

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

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

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

Magic key or share analysis between projects

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

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

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

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

2.5. Summary

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

You will find more information on advanced options available with our tools in "`PolySpace Ada documentation.pdf`" available on the CD-ROM (in \Docs\Manual\) or by clicking on in PolySpace tools.

3. Working with analysis setup

PolySpace provides results at different stages of the analysis. They are the following:

- Compilation errors;

- Link errors;

Related subjects :

3.1. Compile errors

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

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

Related subjects :

3.1.1. OS and target issues

PolySpace takes the type of processor used in the target environment into account during verification. It determines various characteristics of data representation such as data sizes, addressing, and so on. They 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.

Target	sparc	m68k/coldFire	1750a	powerpc32bits	powerpc64bits	I386
Character	8	8	16	8	8	8
short_integer	16	16	16	16	16	16
Integer	32	32	16	32	32	32
long_integer	32	32	32	32	64	32
long_long_integer	64	64	64	64	64	64
short_float	32	32	32	32	32	32
Float	32	32	32	32	32	32
long_float	64	64	48	64	64	64
long_long_float	64	64	48	64	64	64

- Target powerpc32bits: The largest default alignment of basic types within record/array is 64.
- Target powerpc64bits: The largest default alignment of basic types within record/array is 64.
- Target i386: The largest default alignment of basic types within record/array is 32.

To identify a target processor's characteristics, compile and run the program below. If none of the characteristics described above match, please contact PolySpace Technical Support (http://www.polyspace-customer-center.com/).

```
with TEXT_IO;
procedure TEMP is
type T_Ptr is access integer;
Ptr  :T_Ptr;
begin
TEXT_IO.PUT_LINE ( Integer'Image (Character'Size) );
TEXT_IO.PUT_LINE ( Integer'Image (Short_Integer'Size) );
TEXT_IO.PUT_LINE ( Integer'Image (Integer'Size) );
TEXT_IO.PUT_LINE ( Integer'Image (Long_Integer'Size) );
--  TEXT _IO.PUT_LINE ( Integer'Image( Long_Long_Integer'Size) );
TEXT_IO.PUT_LINE ( Integer'Image (Float'Size) );
--  TEXT _IO.PUT_LINE ( Integer'Image( D_Float'Size) );
TEXT_IO.PUT_LINE ( Integer'Image (Long_Float'Size) );
TEXT_IO.PUT_LINE ( Integer'Image (Long_Long_Float'Size) );
TEXT_IO.PUT_LINE ( Integer'Image (T_Ptr'Size) );
end TEMP;
```

3.1.2. Unit analysis

PolySpace needs the complete specifications associated to a package body analysis. Some times we could face this kind of obvious error message:

```
…

Verifying _pst_main

Verifying my_package

-> Verifier found an error in ./My_Package.adb:2:14: Missing specification
for unit "My_Package"
```

PolySpace reports this kind of error when a package body is supplied as the source and the specification is supplied as one of the specifications in one of the -ada-include-dir directories.

Specifications of the package body needs to be included in the list of supplied sources.

3.2. Stubbing errors

Related subjects :

3.2.1. Manual vs. Automatic Stubbing

Only advanced users should consider manual stubbing. PolySpace can automatically stub every missing function or procedure, leading to an efficient analysis with a low loss in precision. The purpose of this chapter is to help you understand how to make analysis even faster and more precise.

What stub functions should I provide?

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.

Let's consider a *procedure_to_stub*.

If *procedure_to_stub* 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 (begin null; end;). PolySpace has no timing constraints and already takes into account all possible scheduling and interleaving and enhances all timing constraints: there is no need to stub functions that set or reset a timer.Simply declare the variable representing time as volatile.

- an I/O access: to a hardware port, a sensor, read/write of a file, read of an eeprom, write to a volatile variable, then: there is no need to stub a write access or simply stub a write access to an empty action (see above), stub read accesses as "I read all possible values (volatile)".

- a write to a global variable, you may need to consider which procedures or function write to it and why: do not stub the concerned `procedure_to_stub` if:
  - this variable is volatile;
  - this variable is a task list. Such lists are accounted for by default because all tasks declared with the -task option are automatically started.
    write a procedure_to_stub by hand
  if this 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 as well. This is easy to achieve by copying the value into a local variable.

Generally speaking, follow the Data Flow and remember that:

- PolySpace only cares about the Ada code which is provided;

- PolySpace does not need to be informed of timing constraints because all possible sequencing is taken into account,

Why should I provide stub functions?

- To avoid partial, imprecise or even wrong results;

- To replace missing code inside functions which PolySpace will ignore.

Example

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

```
function a_missing_function
           (dest: in out integer,
            src  : in integer);
procedure test is
  a: integer;
  b: integer;
begin
  a: = 1;
  b: = 0;
  a_missing_function(a,b);
  b:= 1 / a;
  -- "/" with the default stubbing
end;
```

§   By relying on PolySpace default stub, the division is shown with an **orange** warning because 'a' is assumed to be anywhere in the full permissible integer range (including 0)

§   If the function was commented out, then the division would be **green**.

§   A **red** division could only be achieved with a manual stub.

This example shows what might happen if the affects of assembly code are ignored.

```
procedure test is
begin
  a:= 1;
  b:= 0;
  -- copy "b" to "a":
  -- b:= a
  pragma asm ("move: a,b")
  b:= 1 /a;
end;
```

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.

When and why should I stub

- manually?
    - to gain precision by restricting cases covered by automatic stubs;
    - because my function writes to global variables.
- automatically?
    - Because no run time error will ever be introduced by automatic stubbing, as the worst case is always assumed;
    - It is very quick to do.

3.2.2. Automatic stubbing

Problem

What is the default behaviour for missing functions?

Explanation

Some functions may not be included in the set of Ada source files because:

- they are external,
- they are written in C, or any other language than Ada,
- they are part of the system libraries.

PolySpace relies on and trusts their specifications when stubbing them.

Solution

Add the `-automatic-stubbing` option to your launching script and PolySpace will stub missing code as follows:

- for an **in** parameter, nothing happens;
- for an **out** (or **in out**) parameter, the variable will be given the full range of its type;
- for a **return** parameter, it will be the full range of its type.

A procedure with this specification:

```
procedure a_missing_function (a: in out type_1,  b: in integer);
```

will be stubbed like so:

```
a_missing_function (var_1, var_2)
```

That is - the "var_1" variable will be overwritten with the full range of type_1.

3.2.3. Pragma assert

You can use the construct 'pragma assert' within your code to inform PolySpace of constraints imposed by the environment in which the software will run. A "pragma assert" function is:

```
pragma assert(<integer expression>);
```

If <integer expression> evaluates to zero, then the program is assumed to be terminated, therefore there is a "real" run time error. This is why PolySpace will produce checks for them. The behaviour matches the one exhibited during execution, because **all execution paths for unsatisfied conditions are truncated** (red and then grey). Thus it can be assumed that any analysis performed downstream of the assert uses value ranges which satisfy the assert conditions.

It is therefore possible to use the construct 'pragma assert' in a procedure to inform PolySpace of constraints of the environment in which the software will be embedded. User assertions can be used to describe the physical properties of the environment such as:

- the maximum and minimum speed limit (a car never goes faster than 200 miles per hour or slower than 0),
- the maximum duration of software exploitation (five years for a satellite and one hour for its launcher),
- and so on ...

Example

```
procedure main is
  counter: integer;
  -- counter is not initialized
  random: integer;
  pragma volatile (random);
begin
  counter:= random;
  -- counter~ [-2^31, 2^31-1]
  pragma assert (counter < 1000);
  pragma assert (counter > 100);
end;

end main;
```

Both assertions are orange because the conditions may or may not be fulfilled. But, from then on, counter ~ [101, 999] because any execution paths that does not meet the conditions are halted.

3.2.4. Volatile

Problem

A volatile variable can be defined as a variable which does not respect the "RAM axiom".

This axiom is:

"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."

Explanation

As the value of a volatile variable is "unknown", it can take any value (that can be) represented by the type of the variable and can change even between 2 successive memory accesses.

A volatile variable is viewed as a "permanent random" by PolySpace because the value can change within its whole range between one read access and the next.

Note that even if the volatile characteristic of a variable is also commonly used by programmers to avoid compiler optimization, it has no consequence for PolySpace.

```
function test return integer is
  random: Integer;
  pragma volatile (random);
  y: Integer;            -- random ~ [-2^31, 2^31-1] ,
                         -- although random is not initialized
begin
  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]
end;
```

3.3. Advanced setup

Related subjects :

3.3.1. 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 (for instance using constrained type or sub types), then there will be less "noise" in the form of orange checks resulting from "impossible" values.

Related subjects :

3.3.1.1. 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 (only in C or C++ language), –O0, –O1, -O2 or –O3 precision level. You can also select a particular precision for a specific body (in Ada) or a C file (in C).

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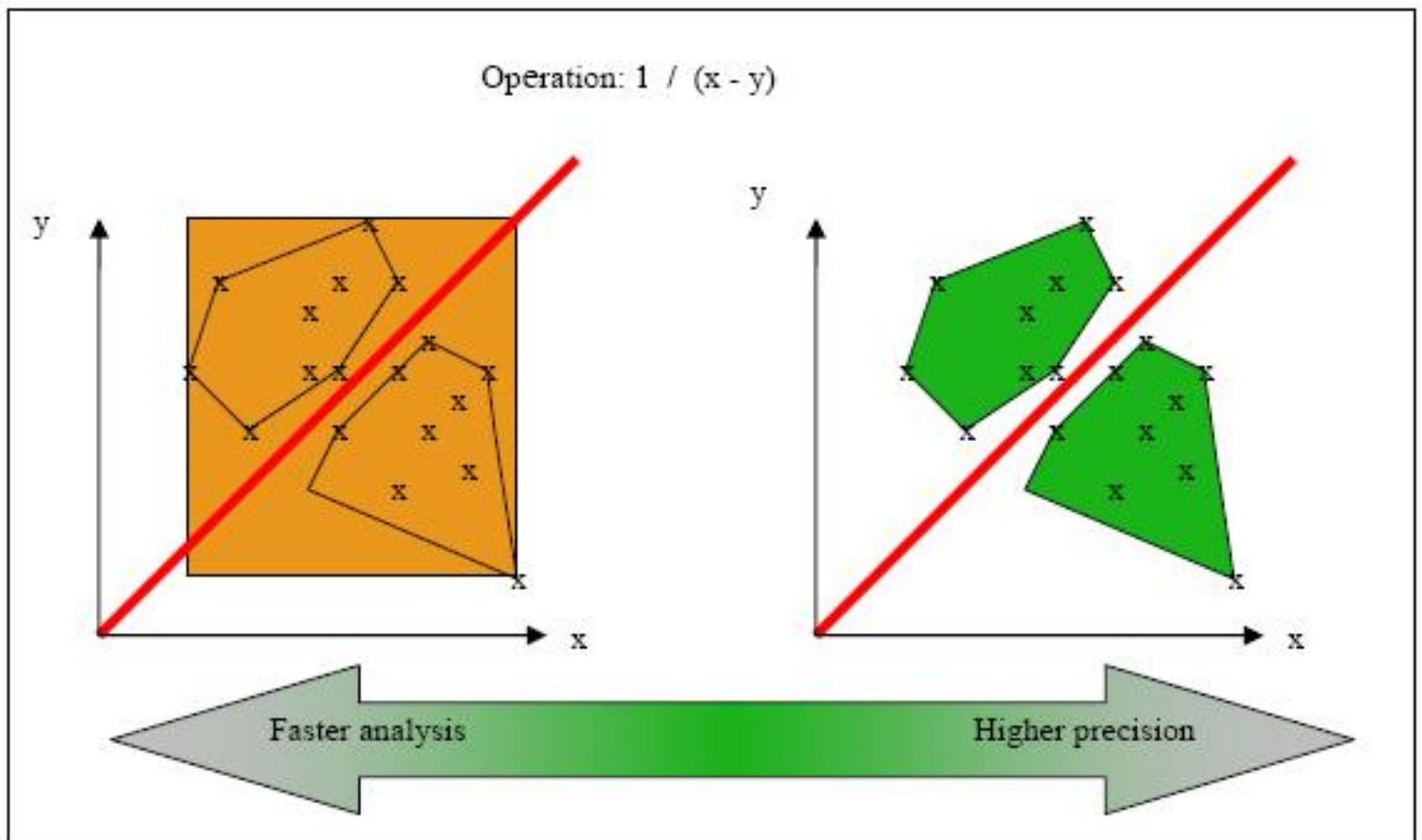
Vary the precision rate

3.3.1.2. Apply chosen coding rules

Hereunder is a list of simple rules that allow PolySpace to be more precise and will higher the selectivity of any Ada analysis:

- Use constrained types. Use subtype and not standard type
- Do not use "use at" clause
- Do not use `unchecked_conversion`
- Minimize the use of big and complex types (record of record, array of record, etc.)
- Minimize the use of volatile variables,
- Minimize the use of assembler code.
- Do not mix assembly code and Ada. Gather all assembly code in a procedure/function which can be automatically stubbed.

3.3.1.3. Increase the number of red and green checks

This example shows a header for a missing function (which might occur, for example, if the code is an incomplete subset of a project). The missing function copies the value of the 'src' parameter to 'dest':

```
function a_missing_function
           (dest: in out integer,
            src  : in integer);
```

Applying fine-level modeling of constraints in primitives and outside functions at the application periphery will propagate more precision throughout the application, which will result in a higher selectivity rate (more proven colours, i.e. more **red** + **green** + **grey**). For this function it could be only adding a simple body:

```
function a_missing_function
           (dest: in out integer,
            src : in integer)
begin
      dest := src;
end;
```

In this case, it is obvious that instead of considering full range for dest parameter, PolySpace will consider the relation between input parameter src and output parameter, propagating more precision throughout the application. See same example in "Manual vs. Automatic Stubbing" paragraph.

3.3.1.4. Apply some functional constraints to variables

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 examples.

with volatile and assert	with assert, and without volatile	without assert, without volatile, without "if"
```function stub return INTEGER is tmp: INTEGER; random: INTEGER; pragma volatile (random); begin tmp:= random; pragma assert (tmp>=1); pragma assert (tmp<=10); return tmp; end;```	```function random return INTEGER; pragma Interface (C, random); function stub return INTEGER is tmp: INTEGER; begin tmp:= random; pragma assert (tmp>=1); pragma assert (tmp<=10); return tmp; end;```	```function random return INTEGER; pragma Interface (C, random); function stub return INTEGER is tmp: INTEGER; begin tmp:= random; while (tmp<1 or tmp>10) loop tmp:=random; end loop; return tmp; end;```

Given that PolySpace models data ranges throughout the code it analyses, it will obviously produce more precise, informative results, - provided that the data it considers from the "outside world" is representative of the data that can be expected when the code is implemented. There is a certain number of mechanisms available to model such a data range within the code itself, and three possible approaches are presented here.

There is no particular advantage in using one approach or another (except, perhaps, that the assertions in the first two will usually generate orange checks) – it is largely down to personal preference.

3.3.1.5. Tuning PolySpace parameters

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 <u>–O3</u> to a lower precision, `-O2, -O1` or `-O0`

3.3.2. Variables

Related subjects :

3.3.2.1. Float rounding

PolySpace handles float rounding by following the the ANSI/IEEE 754-1985 standard. Using the -ignore-float-rounding option, PolySpace computes exact values of floats. 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. Using the option allows to first have a look on remaining unproven check OVFL.

The Following example shows the board effect of such option:

```
package float_rounding is
    procedure main;
end float_rounding;

package body float_rounding is
    procedure main is
      x : float := float'last;
      random : boolean;
      pragma import(C,random);
    begin
      if random then
          x := x + 5.0 - float'last;
          -- with    -ignore-float-rounding : overflow red on + 5.0
          -- without -ignore-float-rounding : overflow orange and x is very
close to zero
      else
          x := x - 5.0 - float'last;
          -- with    -ignore-float-rounding : x is now equal to 5.0
          -- without -ignore-float-rounding : x is very close to zero
      end if;
    end;
end float_rounding;
```

3.3.2.2. Expansion of sizes

The –array-expansion-size option forces PolySpace to analyze each cell of global variable arrays having length less or equal to number as a separate variable.

Example:
```
Package body Test is
  Glob_Array_3 : array(1..3) of Integer := (1,2,3);
  Glob_Array_8 : array(1..8) of Integer := (1,2,3,4,5,6,7,8);
  procedure Main is
    begin
       pragma Assert (Glob_Array_3(3) = 3);
       pragma Assert (Glob_Array_8(3) = 3);
    end Main;
end Test;
```

In the example above, we will have different colours on the assert, depending whether this option has been set or not:
- With -array-expansion-size 3:
  Glob_Array_3 is analysed as separate variables. Therefore, PolySpace is able to say that the assert is true, i.e. green:
  ```
  pragma Assert (Glob_Array_3(3) = 3)
  ```
  Glob_Array_8 is analysed as a single variable, and the assert is orange;
  ```
  pragma Assert (Glob_Array_8(3) = 3);
  ```
- With -array-expansion-size 8, both variables are analysed cell by cell, and both assert are green.

The -variable-to-expand option is used to specify aggregate variables (record, etc.) that will be split into independent variables for the purpose of an analysis. This option has an impact on the Global Data Dictionary results:
- Each variable specified in this option will see its fields analysed separately;
- The data dictionary will distinguish fields accessed by different tasks.
The depth of the variable to expand is controlled by the –variable-expansion-depth.

Note: Expansion options have an impact on the duration of an analysis.

4. Working with results review

Related subjects :

4.1. 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 :

4.1.1. Propagation of colors

For this step, you will 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

| ```
procedure red is
   X: integer;
begin
   X:= 1 / X;
   X:= X + 1;
end;
``` | ```
function read_an_input return integer;
procedure propagate is
   X: Integer;
   Y: array (0..99) of Integer;;
begin
   X:= Read_An_input;
   Y(X):= 0; -- [array index within bounds]
   Y(X):= 0;
end main;
``` |

Let's detail each line of code for:

- The red function:

  When PolySpace divides by X, X has not been initialized. Therefore the corresponding check (Non Initialized Variable) on X is red;

  As a result all possible execution paths are stopped, because they all produce an RTE.

- The propagate function:

  X is assigned the value of Read_An_Input. After this assignment, X ~ [-2^31, 2^31-1];

  At the first array access, an "out of bounds" error is possible since X can be equal to (say) -3 as well as 3;

  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;

  Consequently at the next instructions X ~ [0, 99];

  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 Ada: pragma assert.

4.1.2. 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 := x + 1;`

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 > 101)`

does not mean PolySpace does not know the value of X, which might be the conclusion of a hasty analysis.

So - what does it mean?

```
function Read_An_Input return integer;
procedure Main   is
    X: Integer;
begin
  if (Read_An_input) then
    X := 100;
  end if;
  if (X > 101) then  -- [orange on the NIV : non initialised variable ]
    X : = X + 1;  -- grey code
  end if;
end Main;
```

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 > 101)"` means: PolySpace does not know anything.
- TRUE: `if "( x > 101)"` means: PolySpace does not know if x has been initialized.

The first rules of reviewing results are: focus on the message given by PolySpace and do not focus on a quick interpretation.

4.1.3. What is the Ada explanation?

Try to explain results based on the code and not on:

- A physical action,
- A particular configuration, data calibration,
- Or any other reason than the code itself.

Concentrate on the source code only – remember, Verifier knows nothing of the environment in which the code will be executing.

In the example below what is the explanation of the dead code (**grey code**) following the "if" statement?

```
function Read_An_Input return integer;

procedure Main  is
X: Integer;
Y: array (0..99) of Integer;
begin
X := Read_An_input;
Y(X) := 0; -- [array index may be without its bounds] [x is
initialized]
Y(X-1):= (1 / X) + X ; [array index is within its bounds]

if (X = 0) then
Y(X) := 1; -- this line is unreachable
end if;
end Main;
```

This is a method you can use to understand any colour

- First step: the line containing the access to the Y array is unreachable (**this line is unreachable**)

  - So - the test to assess whether x is equal to 0 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.
  - X has been assigned to its full range, but the test assumes that X is never equal to 0 at this line. Why?

- Second step: "Why is the test always false?"

  - After the variable definitions, it can be seen that the first array access is orange: before this line $X \in [-2\^31, 2\^31-1]$ because of the Read_An_Input function, and afterwards, $X \in [0, 99]$ (see Examples "•*Example D*" and "•*Example E*")
  - So $X \in [0, 99]$ just after the first array access.

- The next operation to be checked by PolySpace Verifier is the addition "… + X" which is **green**

- The next operation checked after that will be the division by X which is **orange** because X ∈ [0, 99]. So after the division, X ∈ [1, 99]. The orange will truncate all execution paths that lead to a run time error, so that in our example, all instances where X is equal to 0 are stopped.

- <u>Third step</u>: the second array index is green and therefore explains why the test is always false.

When the assignment sign is reached, X ∈ [1, 99] and hence the array access is green.

- <u>Conclusion</u>: The user has found a **bug!** The **dead code** has shown that the test should be performed before the division

> Note   You must explain a colour step by step, until you find the root cause, and focus on explanation within the code only. Try to exclude the knowledge about what the code actually does in its execution environment.

<u>Note</u>: In this example, all results are located in the same procedure. The same approach is valid if a check is to be analysed involving a procedure called by others. Use the "called by" call tree to help in the analysis of the results.

4.1.4. Review run time errors: Fix red errors

All Run Time Errors 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 "IN/OUT" ADA function parameters to be initialised. Examples in C include options to deal with constant overflows, shift operation on negative values, etc.

- All **other red errors must be fixed**. They are bugs.

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

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

Related subjects :

4.1.5.1. 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 a line of code such as
      IF NOT a AND b OR c AND d
    - Now consider how misplaced parentheses might influence how that line behaves …
      IF NOT (a AND b OR c AND d)
  IF (NOT (a) AND b) OR (c AND d))
            IF NOT (a AND (b OR c) AND d)
            …
- The test of variable inside a branch where the conditions are never met;
- An unreachable "else" clause where the wrong variable is tested in the "if" statement
- A variable that is supposed to be local to the file but instead is local to the function
- Wrong variable prototyping leading to a comparison which is always false (say)
- …

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

- From one week effort of functional testing on target, trying to build a scenario going into that branch, and wondering why the functional 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.

4.1.5.2. Note on 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 aid in dead code detection. It detects dead code which might take days of effort to find by any other means.

4.1.6. How to conclude an orange review

Related subjects :

4.1.6.1. 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.)

4.1.6.2. 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 initialised 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.

4.1.6.3. 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.

4.2. Automatic Methodology

During an analysis, PolySpace is able to automatically highlight some orange checks considered as potential robustness issues in the code.
The automatic methodology separates a sub part of orange NIVL and orange OVFL from all oranges checks:

- all NIVL scalar local oranges. Theses NIV do not concern float, record (and component) and arrays.
- All OVFL/UNFL scalar oranges between subtypes: conversion of a subtype in a smaller subtype.

From a Methodology point of view, theses checks need to be addressed first. As PolySpace is very precise on them, we can always deduce that an orange of this kind is most of the time synonymous of a robustness issue.

Example:

```
1  Package body Test is
2  ATab : array(0..9) of Integer := (Others => 0);
3  function Assign_array(X : integer) return Integer is
4      Y : Integer;
5   begin
6      y := ATab( X - 12); -- Warning UOVFL on operator "-" given by
7                          -- the Automatic methodology
8      return y;
9   end Assign_Array;
10
11 function read_bus_status return boolean;  -- function stubbed
12 procedure partial_init( New_Alt : in out Integer ) is
13     Y : boolean;
14   begin
15      if read_bus_status then
16         New_Alt := 12;
17          Y := True;
18      else
19         New_Alt := 120;
20      end if;
21      if Y then               -- Warning NIVL on "Y" given by
22                              -- the automatic methodology
23         New_Alt := New_Alt * 10;
24      end if;
25    end partial_init;
```

```
26 end Test;
```

In the example above, the automatic methodology filters all orange except:

- The orange UOVFL at line 6. The associated message associated to this orange says "Scalar variable may underflow/overflow on [conversion from -2**31.. 2**31-1 to 0..9]". In this case we have a typical conversion in a smaller subtype and nothing around shows a defensive code against this robustness issue.
- The orange NIVL at line 21. The associated message associated to this orange says "Local variable may be not initialized". In this case we have a typical example which leads to a robustness issue if the right branch is not executed.

Activation and filter location:

In both mode of review (expert or assistant) the automatic methodology is always active.

Opening the Viewer on results, chose expert mode, select "Alpha" filter and then, clicking on **"I/ ?"** button associated to tool tip "**Click to hide orange not associated to additional information**", allows to show all oranges and only coming from the automatic methodology.

4.3. How to find a maximum number of bugs within an hour reviewing oranges: selective orange review

> Note   Before reading this section, it is necessary to understand how the user might conclude the status of an **orange** check. This is explained in a later section.

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

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

Related subjects :

4.3.1. How?

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

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

4.3.2. Why?

- 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.

4.3.3. In practice…

For any particular function, PolySpace may 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.

4.3.4. Step by step

1.   Select one type of RTE, such as Zero Division (ZDV) as shown in the example. Click on [Filter all] and then click on the check type of interest (ZDV in the example)

OBAI	ZDV	NIV local	SCAL OVFL	IDP	COR	IRV	SHF	NIV other	NIP	FLOAT OVFL	ASRT	NTC	K-NTC	NTL	UNR	VOA

Tool bar for checks

Here, all RTEs are filtered except ZDV

- Choose files/packages containing only 1 or 2 **orange checks** of the selected kind.

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

  - 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 RTE and repeat step 1-4.

4.3.5. Which category of checks should I choose first?

The following sequence is recommended.

- Start with the four categories found to be the most likely to yield bugs, which are described in the following sections.

    • Next, use the Beta filter which will highlight the remaining categories most likely to include any remaining critical Run Time Errors.

    • Finally, complete the remaining checks as time permits.

The impact made by the use of Ada coding rules is huge, because they reduce complexity - a key factor in limiting **orange checks** due to basic imprecision. The Ada constructions impacting each of the four are listed below.

    • Potential bug or data set issue. These are **orange checks** representing genuine problems.

    • Inconclusive check. These are **orange checks** which mostly highlight design issues, not addressed by this section.
    • Basic imprecision.

        • Unspecified Standard behaviour
        • Complexity
        • Approximations made by the tool on specific constructions

4.3.6. Exhaustive orange review at unit phase

Related subjects :

4.3.6.1. Without coding rules

An exhaustive orange review progresses at a typical rate of **50 orange checks per hour.** An hour spent on an exhaustive check review is different to an hour spent on a selective orange review in several significant ways.

- Time:

  o The first 10 minutes of the exhaustive check will be dedicated to the classification of 2/3 of the orange as false anomalies.
  o The last 40 minutes will be used to track more complex bugs.

- Cost:

  o 80% of the **orange checks** will require only a few seconds of effort before a conclusion can be reached. These are not integration bugs, so tracking the cause of an **orange check** is often much faster than the same activity in a larger piece of code.
  o The typical time spent reviewing each **orange check** would be about 1 minute.

4.3.6.2. With coding rules

The number of spurious **orange checks** per file strongly depends on coding styles within the project.

If the code follows the some rules (using subtype instead of plain type for instance, etc.), the count of checks per file will typically decrease to 3 **orange** and 3 **grey** checks, hiding at least one bug between them.

The review of the PolySpace results generated by a unit analysis would normally take no more than 15 minutes. See some of coding rules recommended by PolySpace in "Coding rules" paragraph.

4.4. Coloured source code for Ada

Related subjects :

4.4.1. Non-Initialized Variable: NIV/NIVL

Check to establish whether a variable is initialized before being read.
ADA example:

```ada
1    package NIV is
2        type Pixel is
3            record
4                X : Integer;
5                Y : Integer;
6            end record;
7        procedure MAIN;
8        function Random_Bool return Boolean;
9    end NIV;
10
11    package body NIV is
12
13        type TwentyFloat is array (Integer range 1.. 20) of Float;
14
15        procedure AddPixelValue(Vpixel : Pixel) is
16            Z : Integer;
17        begin
18            if (Vpixel.X < 3) then
19                Z := Vpixel.Y + Vpixel.X; -- NIV error: Y field not initialized
20            end if;
21        end AddPixelValue;
22
23        procedure MAIN is
24            B : Twentyfloat;
25            Vpixel : Pixel;
26        begin
27            if (Random_Bool) then
28                Vpixel.X := 1;
29                AddPixelValue(Vpixel); -- NTC Error: because of NIV error in call
30            end if;
31
32            for I in 2 .. Twentyfloat'Last loop
33                if ((I mod 2) = 0) then
34                    B(I) := 0.0;
35                end if;
36            end loop;
37            B(2) := B(4) + B(5);        -- NIV Warning because B(odd) not initialized
38        end MAIN;
39
40    end NIV;
```

Explanation:

The result of the addition is unknown at line 19 because *Vpixel.Y* is not initialized (grey code on "+" operator). In addition, line 37 shows how PolySpace prompts the user to investigate further (orange NIV warning on *B(I))* when all fields have not been initialized.

NIV check versus *IN OUT* parameter mode

Standard Ada83 says: For a scalar parameter, the above effects are achieved by copy: at the start of each call, if the mode is in or in out, the value of the actual parameter is copied into the associated formal parameter; then after normal completion of the subprogram body, if the mode is in out or out, the value of the formal parameter is copied back into the associated actual parameter.

Clearly, in out parameters necessitate initialisation before call.

ADA example:

```
1      package NIVIO is
2          procedure MAIN;
3          function Random_Boolean return Boolean;
4      end NIVIO;
5
6      package body NIVIO is
7
8          Y : Integer := 3;
9          procedure Niv_Not_Dangerous(X : in out integer) is
10         begin
11            X := 2;
12            if (Y > 2) then
13                Y := X + 3;
14            end if ;
15         end Niv_Not_Dangerous;
16
17         procedure Niv_Dangerous(X : in out integer) is
18         begin
19            if (Y /= 3) then
20                Y := X + 3;
21            end if ;
22         end Niv_Dangerous;
23
24         procedure MAIN is
25             X : Integer;
26         begin
27            if (Random_Boolean) then
28                Niv_Dangerous(X);          -- NIV ERROR: certainly dangerous
29            end if ;
30            if (Random_Boolean) then
31                Niv_Not_dangerous(X);   -- NIV ERROR: not dangerous
32            End if ;
33         end MAIN;
34
35     end NIVIO;
```

Explanation:

In the previous example, as shown at line 28, Verifier highlights a dangerous not initialized variable. Even it is not dangerous, as shown in the *Niv_Not_Dangerous* procedure, Verifier also highlights the non initialized variable at line 30. To be more permissive with standard, the *-continue-with-in-out-niv* option permits to continue the analysis of the rest of sources even red error stays in place at line 28 and 31.

Related subjects :

4.4.1.1. Pragma interface/import

The following table illustrates how variables are regarded when:

- A pragma is used to interface the code;
- An address clause is applied;
- A pointer type is declared.

	variable type		Function
	records and other types	integer types	
`pragma interface (C, variable_name)` `pragma import   (C, variable_name)`	- green NIV - Permanent random value	- No NIV check - Permanent random value	- same behaviour as -automatic-stubbing - in/out and out variables are written within their entire type range

In this case, a permanent random value means that the variable is always equivalent to the full range of its type. It is almost equivalent to a volatile variable except for the colour of the NIV.

4.4.1.2. Type access variables

The following table illustrates how variables are analysed by PolySpace when a type access is used:

	variable type	
	records and other types	integer types
```Type a_new_type is access another_type;```	• orange NIV • Permanent random value	• No NIV check • Permanent random value

In this case, a Permanent Random Variable is exactly equivalent to a volatile variable - -that is, it is assumed that the value may have change within its whole range between any read/write accesses.

4.4.1.3. Address clauses

The following table illustrates how variables are regarded by PolySpace where an address clause is used.

	variable type	
	records and other types	integer types

Address clause
`for variable_name use at 16#1234abcd#;`
`for variable_name use at other'address;`
	• orange NIV
	
	• Permanent random value
		• No NIV check
		
		• Permanent random value

In this case, a Permanent Random Variable is exactly equivalent to a volatile variable - that is, it is assumed that the value can have been changed to anywhere within its whole range between one read access and the next.

4.4.2. Division by zero: ZDV

Check to establish whether the right operand of a division (denominator) is different to 0[.0].
ADA example:

```
1    package ZDV is
2        function Random_Bool return Boolean;
3        procedure ZDVS (X : Integer);
4        procedure ZDVF (Z : Float);
5        procedure MAIN;
6    end ZDV;
7
8    package body ZDV is
9
10       procedure ZDVS(X : Integer) is
11           I : Integer;
12           J : Integer := 1;
13       begin
14          I := 1024 / (J-X); -- ZDV ERROR: Scalar Division by Zero occurs
15       end ZDVS;
16
17       procedure ZDVF(Z : Float) is
18           I : Float;
19           J : Float := 1.0;
20       begin
21          I := 1024.0 / (J-Z); -- ZDV ERROR: float Division by Zero occurs
22       end ZDVF;
23
24       procedure MAIN is
25       begin
26          if (random_bool)  then
27             ZDVS(1);  -- NTC ERROR: ZDV.ZDVS call never terminates
28          end if ;
29          if (Random_Bool)  then
30             ZDVF(1.0);  -- NTC ERROR: ZDV.ZDVF call never terminates
31          end if;
32       end MAIN;
33
34   end ZDV;
35
36
37
```

4.4.3. Arithmetic Exceptions: EXCP

Check to establish whether standard arithmetic functions are used with good arguments:

1. Argument of *sqrt* must be positive

2. Argument of *tan* must be different from pi/2 modulo pi

3. Argument of *log* must be strictly positive

4. Argument of *acos* and *asin* must be within [-1..1]

5. Argument of *exp* must be less than or equal to a specific value which depends on the processor target: 709 for 64/32 bit targets and 88 for 16 bit targets

Basically, an error occurs if an input argument is outside the domain over which the mathematical function is defined.

ADA Example:

```
1
2       With Ada.Numerics; Use Ada.Numerics;
3       With Ada.Numerics.Aux; Use Ada.Numerics.Aux;
4
5       package EXCP is
6           function Bool_Random return Boolean;
7           procedure MAIN;
8       end EXCP;
9
10      package body EXCP is
11
12      -- implementation dependant in Ada.Numerics.Aux: subtype Double is Long_Float;
13          M_PI_2 :   constant Double := Pi/2.0; -- pi/2
14
15          procedure MAIN is
16              IRes, ILeft, IRight : Integer;
17              Dbl_Random : Double;
18              pragma Volatile_ada.htm (dbl_Random);
19
20              SP : Double := Dbl_Random;
21              P  : Double := Dbl_Random;
22              SN : Double := Dbl_Random;
23              N : Double := Dbl_Random;
24              NO_TRIG_VAL : Double := Dbl_Random;
25              res : Double;
26              Fres : Long_Float;
27          begin
28              -- assert is used to redefine range values of a variable.
29              pragma assert(SP > 0.0);
30              pragma assert(P >= 0.0);
31              pragma assert(SN < 0.0);
32              pragma assert(N  <= 0.0);
33              pragma assert(NO_TRIG_VAL < -1.0 or  NO_TRIG_VAL > 1.0);
34
35              if (bool_random) then
36                  res := sqrt(sn);              -- EXCP ERROR: argument of SQRT must be positive.
37              end if ;
38              if (bool_random) then
39                  res := tan(M_PI_2);           -- EXCP Warning: Float argument of TAN may be
```

```
40                                              -- different than pi/2 modulo pi.
41          end if;
42          if (bool_random) then
43              res := asin(no_trig_val);  -- EXCP ERROR: float argument of ASIN is not in -1..1
44          end if;
45          if (bool_random) then
46              res := acos(no_trig_val);  -- EXCP ERROR: float argument of ACOS is not in -1..1
47          end if;
48          if (bool_random) then
49              res := log(n);             -- EXCP ERROR: float argument of LOG is not strictly
positive
50          end if;
51          if (bool_random) then
52              res := exp(710.0);         -- EXCP ERROR: float argument of EXP is not less
than or equal to 709 or 88
53          end if;
54
55          -- range results on trigonometric functions
56          if (Bool_Random) then
57              Res := Sin (dbl_random);   -- -1 <= Res <= 1
58              Res := Cos (dbl_random);   -- -1 <= Res <= 1
59              Res := atan(dbl_random);   -- -pi/2 <= Res <= pi/2
60          end if;
61
62          -- Arithmetic functions where there is no check currently implemented
63          if (Bool_Random) then
64              Res := cosh(dbl_random);
65              Res := tanh(dbl_random);
66          end if;
67       end MAIN;
68    end EXCP;
```

Explanation:

The arithmetic functions *sqrt*, *tan*, *sin, cos, asin*, *acos* , *atan* and *log* are derived directly from mathematical definitions of functions.

Standard *cosh* and *tanh* hyperbolic functions are currently assumed to return the full range of values mathematically possible, irrespective of the input parameters. The Ada83 standard gives more details about domain and range error for each maths function.

4.4.4. Scalar and Float Underflow/Overflow : UOVFL

Check to simultaneously establish whether an arithmetic expression on a float value overflows and/or underflows.
ADA Example:

```
1     package UOVFL is
2         function Bool_Random return Boolean;
3         procedure MAIN;
4     end UOVFL;
5
6     package body UOVFL is
7
8         procedure MAIN is
9             I : Integer;
10            DValue : Long_float;
11        begin
12            if (Bool_Random) then
13                I := 2**30;
14                I := 2 * (I - 1);            -- integer UOVFL verified on "*" and "-"
15            end if;
16            if (Bool_Random) then
17                DValue := Long_float(Float'Last);
18                DValue := 2.0 * DValue + 1.0;  -- float UOVFL verified on "*" and "+"
19            end if;
20        end MAIN;
21    end UOVFL;
```

Explanation:
PolySpace can detect that there is neither an underflow nor an overflow on * and - operators at line 16.
At line 20, there is an OVFL error. As a result, Verifier cannot evaluate the overflow and the underflow of the expression with the "+" operator, and so an UOVFL unreachable check results (see also UNR checks).

4.4.5. Scalar and Float Overflow: OVFL

Check to establish whether an arithmetic expression overflows. This is a scalar check with integer types and a float check for floating point expressions.
An overflow is also detected should an array index_ada.htm be out of bounds.
ADA Example:

```
1      package OVFL is
2          procedure MAIN;
3          function Bool_Random return Boolean;
4      end OVFL;
5
6      package body OVFL is
7
8          procedure OVFL_ARRAY is
9              A : array(1..20) of Float;
10             J : Integer;
11         begin
12             for I in A'First .. A'Last loop
13                 A(I) := 0.0 ;
14                 J := I + 1;
15             end loop;
16             A(J) := 0.0;  -- OVFL ERROR: Overflow array index_ada.htm
17         end OVFL_ARRAY;
18
19         procedure OVFL_ARITHMETIC is
20             I : Integer;
21             FValue : Float;
22         begin
23
24             if (Bool_Random) then
25                 I := 2**30;
26                 I := 2 * (I - 1) +2 ;              -- OVFL ERROR: 2**31 is an overflow value for
Integer
27             end if;
28             if (Bool_Random) then
29                 FValue := Float'Last;
30                 FValue := 2.0 * FValue + 1.0;  -- OVFL ERROR: float variable is overflow
31             end if;
32         end OVFL_ARITHMETIC;
33
34         procedure MAIN is
35         begin
36             if (Bool_Random) then OVFL_ARRAY; end if;        -- NTC propagation because of OVFL
ERROR
37             if (Bool_Random) then OVFL_ARITHMETIC; end if;
38         end MAIN;
39
40     end OVFL;
41
42
```

Explanation:
In Ada, the bounds of an array can be considered with reference to a new type or subtype of an existing one. Line 16 shows an overflow error resulting from an attempt to access element 21 in an array subtype of range *1..20*.

A different example is shown by the overflow on line 28, where adding 1 to *Integer'Last* (the maximum integer value being *2**31-1* on a 32 bit architecture platform). Similarly, if *OVFL_ARITHMETIC.FValue* represents the max floating value, *2*FValue* cannot be represented with the same type and so raises an overflow at line 32.

4.4.6. Scalar and Float Underflow: UNFL

Check to establish whether an arithmetic expression underflows. This is a scalar check with integer types and a float check for floating point expressions.
An underflow is also detected should an array index_ada.htm be out of bounds.
ADA Example:

```
1      package UNFL is
2          function Bool_Random return Boolean;
3          procedure MAIN;
4      end UNFL;
5
6      package body UNFL is
7
8          procedure UNFL_ARRAY is
9              A : array(1..20) of Float;
10             J : Integer;
11         begin
12             for I in A'Last.. A'First loop
13                 A(I) := 0.0 ;
14                 J := I - 1;
15             end loop;
16             A(J) := 0.0;                  -- UNFL ERROR: underflow array index_ada.htm
17         end UNFL_ARRAY;
18
19         procedure UNFL_ARITHMETIC is
20             I : Integer;
21             FValue : Float;
22         begin
23
24             if (Bool_Random) then
25                 I := -2**31;
26                 I := I - 1 ;             -- UNFL ERROR: -2**31-1 is integer underflow
27             end if;
28             if (Bool_Random) then
29                 FValue := Float'First;
30                 FValue := -2.0 * FValue;  -- UNFL ERROR: float variable is overflow
31             end if;
32         end UNFL_ARITHMETIC;
33
34         procedure MAIN is
35         begin
36             if (Bool_Random) then UNFL_ARRAY; end if;      -- NTC propagation because of UNFL
ERROR
37             if (Bool_Random) then UNFL_ARITHMETIC; end if;
38         end MAIN;
39
40     end UNFL;
```

Explanation:
In Ada, the bounds of an array can be considered with reference to a new type or subtype of an existing one. Line 16 shows an underflow error resulting from an attempt to access element 0 in an array subtype of range *1..20*.
A different example is shown by the underflow on line 28, where subtracting 1 from *Integer'First* (the minimum integer value being -*2**31-1* on a 32 bit architecture platform). Similarly, if *UNFL_ARITHMETIC.FValue* represents the minimum floating value, *-2*FValue*

cannot be represented with the same type and so raises an underflow at line 33.

4.4.7. Attributes check: COR

PolySpace encourages the user to investigate the attributes *SUCC*, *PRED*, *VALUE* and *SIZE* further, thanks to a COR check (failure of CORrectness condition).

ADA example:

```
1
2     package CORS is
3         function Bool_Random return Boolean;
4         procedure MAIN;
5         function INT_VALUE (S : String) return Integer;
6         type PSTCOLORS is (ORANGE, RED, GREY, GREEN);
7         type ADCFUZZY is  (LOW, MEDIUM, HIGH);
8     end CORS;
9
10    package body CORS is
11
12        type STR_ENUM is (AA,BB);
13
14        function INT_VALUE (S : String) return Integer is
15           X : Integer;
16        begin
17           X := Integer'Value (S);    -- COR Warning: Value parameter might not be in range
integer
18           return X;
19        end INT_VALUE;
20
21        procedure MAIN is
22           E      : PSTCOLORS := GREEN;
23           F      : PSTCOLORS;
24           ADCVAL : ADCFUZZY := ADCFUZZY'First;
25           StrVal : STR_ENUM;
26           X : Integer;
27        begin
28           if (Bool_Random) then
29              F := PSTCOLORS'PRED(E); -- COR Verified: Pred attribute is not used on the
first element of pstcolors
30              E := PSTCOLORS'SUCC(E); -- COR ERROR: Succ attribute is used on the last
element of pstcolors
31           end if;
32           if (Bool_Random) then
33              ADCVAL := ADCFUZZY'PRED(ADCVAL);  -- COR ERROR: Pred attribute is used on the
first element of adcfuzzy
34           end if ;
35
36           StrVal := STR_ENUM'Value ("AA");     -- COR Warning: Value parameter might not be
in range str_enum
37           StrVal := STR_ENUM'Value ("AC");     -- COR Warning: Value parameter might not be
in range str_enum
38           X := INT_VALUE ("123");              -- Information on X: -2**31<=[expr]<=2**31-1
39        end MAIN;
40    end CORS;
41
```

Explanation :

At line 36 and 37, the COR warning (orange) prompts the user to check wether the *VALUE* attribute is correct or not.

In fact, standard ADA generates a "CONSTRAINT_ERROR" exception when the string does not correspond to one of the possible values of the type.

Also note that in this case, Verifier results assume the full possible range of the returned type, irrespective of the input parameters. In this example, *strVal* has a range in *[aa,bb]* and *X* in *[Integer'First, Integer'Last]*.

The incorrect use of *PRED* and *SUCC* attributes on type is put forward by PolySpace.

SIZE attribute error: COR

```
1
2      with Ada.Text_Io; use Ada.Text_Io;
3
4      package SIZE is
5          PROCEDURE Main;
6      end SIZE;
7
8      PACKAGE BODY SIZE IS
9
10         TYPE unSTab is array (Integer range <>) of Integer;
11
12         PROCEDURE MAIN is
13            X : Integer;
14         BEGIN
15            X := unSTab'Size;             -- COR ERROR: Size attribute must not be used for
unconstrained array
16            Put_Line (Integer'Image (X));
17         END MAIN;
18
19      END SIZE;
```

Explanation:

At line 15, PolySpace shows the error on the *SIZE* attribute. In this case, it cannot be used on an unconstrained array.

4.4.8. Array length check: COR

Checks the correctness condition of an array length, including *Strings*.
ADA example:

```
1
2      with Dname;
3      package CORL is
4          function Bool_Random return Boolean;
5          type Name_Type is array (1 .. 6) of Character;
6          procedure Put (C : Character);
7          procedure Put (S : String);
8          procedure MAIN;
9      end CORL;
10
11     package body CORL is
12
13         STR_CST : constant NAME_TYPE := "String";
14
15         procedure MAIN is
16             Str1,Str2,Str3 : String(1..6);
17             Arr1 : array(1..10) of Integer;
18         begin
19
20             if (Bool_Random) then
21                 Str1 := "abcdefg";              -- COR ERROR: Too many elements in array, must
have 6
22             end if;
23             if (Bool_Random) then
24                 Arr1 := (1,2,3,4,5,6,7,8,9); -- COR ERROR: Not enough elements in array, must
have 10
25             end if ;
26             if (Bool_Random) then
27                 Str1 := "abcdef";
28                 Str2 := "ghijkl";
29                 Str3 := Str1 & Str2;           -- COR Warning: Length might not be compatible
with 1 .. 6
30                 Put(Str3);
31                 if Bool_Random then
32                     DName.DISPLAY_NAME (DNAME.NAME_TYPE(STR_CST)); -- COR ERROR: String Length
is not correct, must be 4
33                 end if;
34             end if ;
35         end MAIN;
36
37     end CORL;
38
39     package DName is
40         type Name_Type is array (1 .. 4) of Character;
41         PROCEDURE DISPLAY_NAME (Str : Name_Type);
42     end DName;
43
```

Explanation:

At lines 21 and 24, PolySpace gives the exact value needed to match the two arrays. On the other hand, PolySpace prompts the user to investigate the compatibility of concatenated arrays, by means of an orange check at line 29.
Moreover at line 32, the string length is being put forward even if it depends on another package.

4.4.9. DIGITS value check: COR

Checks the length of *DIGITS* constructions.
ADA example:

```
1      package DIGIT is
2          procedure MAIN;
3      end DIGIT;
4
5      package body DIGIT is   -- NTC ERROR: COR propagation
6
7          type T is digits 4             range 0.0 .. 100.0;
8          subtype T1 is T
9            digits 1000 range 0.0 .. 100.0;  -- COR ERROR: digits value is too large, highest
possible value is 4
10
11         procedure MAIN is
12         begin
13             null;
14         end MAIN;
15     end DIGIT;
```

Explanation:

At line 9, PolySpace shows an error on the *digits* value. It indicates in its associated message the highest available value, 4 in this case.

4.4.10. DELTA value length check: COR

Checks the length of *DELTA* constructions.

ADA example:

```
1
2      package FIXED is
3         procedure MAIN;
4         procedure FAILED(STR : STRING);
5         function Random return Boolean;
6      end FIXED;
7
8      package body FIXED is
9
10        PROCEDURE FIXED_DELTA IS
11
12           GENERIC
13              TYPE FIX IS DELTA <>;
14           PROCEDURE PROC (STR : STRING);
15
16           PROCEDURE PROC (STR : STRING) IS
17              SUBTYPE SFIX IS FIX DELTA 0.1 RANGE -1.0 .. 1.0; -- COR ERROR: delta is too
small, smallest possible value is 0.5E0
18           BEGIN
19              FAILED ( "NO EXCEPTION RAISED FOR " & STR );
20           END PROC;
21
22           BEGIN
23
24              IF RANDOM THEN
25                 DECLARE
26                    TYPE NFIX IS DELTA 0.5 RANGE -2.0 .. 2.0;
27                    PROCEDURE NPROC IS NEW PROC (NFIX);
28                 BEGIN
29                    NPROC ( "INCOMPATIBLE DELTA" );                -- NTC ERROR: propagation of
COR Error
30                 END;
31              END IF ;
32
33           END FIXED_DELTA;
34
35        procedure MAIN is
36        begin
37           FIXED_DELTA;
38        end MAIN;
39
40     end FIXED;
```

Explanation:

At line 17, Polyspace Verifier shows an error on the *DELTA* value. The message gives the smallest available value, *0.5* in this case.

4.4.11. Static range and values check: COR

Checks if constant values and variable values correspond to their range definition and construction.
ADA example:

```
1
2      package SRANGE is
3          procedure Main;
4          function IsNatural return Boolean;
5
6          SUBTYPE INT IS INTEGER RANGE 1 .. 3;
7          TYPE INF_ARRAY IS ARRAY(INT RANGE <>, INT RANGE <>) OF INTEGER;
8          SUBTYPE DINT IS INTEGER RANGE 0 .. 10;
9      end SRANGE;
10
11     package body SRANGE is
12
13         TYPE SENSOR IS NEW INTEGER RANGE 0 .. 10;
14
15         TYPE REC2(D : DINT := 1) IS RECORD      -- COR Warning: Value might not be in range
1 .. 3
16             U : INF_ARRAY(1 .. D, D .. 3) := (1 .. D =>
17                                               (D .. 3 => 1));
18         END RECORD;
19         TYPE REC3(D : DINT := 1) IS RECORD      -- COR Error: Value is not in range 1 .. 3
20             U : INF_ARRAY(1 .. D, D .. 3) := (1 .. D =>
21                                               (D .. 3 => 1));
22         END RECORD;
23
24         PROCEDURE VALUE_RANGE is
25             VAL : INTEGER;
26             pragma Volatile(VAL);
27             SLICE_A2  : REC2(VAL);   -- NIV and COR warning: Value might not be in range 0 ..
10
28             SLICE_A3  : REC3(4);     -- Unreacheable code: because of COR Error in REC3
29         BEGIN
30             NULL;
31         END VALUE_RANGE;
32
33         PROCEDURE MAIN is
34             Digval : Sensor;
35         begin
36             if IsNatural then
37                 declare
38                     TYPE Sub_sensor is new Natural range -1 .. 5; -- COR Error: Static value is
not in range of 0 .. 16#7FFF_FFFF#
39                 begin
40                     null;
41                 end;
42             end if;
43             if IsNatural then
44                 declare
45                     TYPE NEW_ARRAY IS ARRAY (NATURAL RANGE <>) OF INTEGER;
46                     subtype Sub_Sensor is New_Array (Integer RANGE -1 .. 5); -- COR Error:
```

```
Static range is not in range 0 .. 16#7FFF_FFFF#
47              begin
48                 null;
49              end;
50          end if ;
51          if IsNatural then
52             VALUE_RANGE;  -- NTC Error: propagation of the COR error in VALUE_RANGE
53          else
54             Digval := 11; -- COR Error: Value is not in range of 0 .. 10
55          end if;
56      END Main;
57    end SRANGE;
58
59
```

Explanation:

PolySpace checks the compatibility between range and value. Moreover, it tells in its associated message the expected length.

Example is shown on the record types *REC2* and *REC3*. Verifier cannot determine the exact value of the volatile variable *VAL* at line 27, because some paths lead to a safe definition, others to a red one. The results is an orange warning at line 15.

At line 19, 38, 46 and 54 PolySpace is able to prompts errors on out of range values.

4.4.12. Discriminant check: COR

Checks the usage of a discriminant in a record declaration.
ADA example:

```
1
2    package DISC is
3       PROCEDURE MAIN;
4
5       TYPE T_Record(A: Integer) is record -- COR Verified: Value is in range of 1 ..
16#7FFF_FFFF#
6          Sa: String(1..A);
7       END RECORD;
8    end DISC;
9
10   package body DISC is
11
12      PROCEDURE MAIN is
13      begin
14         declare
15            T_STRING6     : T_RECORD(6) := (6, "abcdef"); -- COR Verified: Discriminant is
compatible
16            T_StringOther : T_RECORD(6);                 -- COR Verified: Discriminant is
compatible
17            T_STRING5     : T_RECORD(5) := (5, "abcde");  -- COR Verified: Discriminant is
compatible
18         begin
19            T_StringOther := T_STRING6;                  -- COR Verified: Discriminant is
compatible
20            T_string5     := T_Record(T_STRING6);        -- COR ERROR:    Discriminant is
not compatible
21         end;
22      END Main;
23
24   END DISC;
```

Explanation:

At line 20, PolySpace shows an error while using a discriminant. *T_String6* discriminant of length 6 cannot match *T_String5* discriminant of length 5.

4.4.13. Component check: COR

Checks whether each component of a record given is being used accurately.

ADA example:

```
1    package COMP is
2
3        PROCEDURE MAIN;
4        SUBTYPE DINT IS INTEGER RANGE 0..1;
5        TYPE COMP_RECORD ( D : DINT := 0) is record
6            X : INTEGER;
7            CASE D IS
8                WHEN 0 => ZERO : BOOLEAN;
9                WHEN 1 => UN   : INTEGER;
10           END CASE;
11       END RECORD;
12
13   end COMP;
14
15   package body COMP is
16
17       PROCEDURE MAIN is
18           CZERO : COMP_RECORD(0);
19       BEGIN
20           CZERO.X    := 0;
21           CZERO.ZERO := FALSE;   -- COR Verified: zero is a component of the variable
22           CZERO.UN   := CZERO.X; -- COR ERROR:    un is not a component of the variable
23       END MAIN;
24   END COMP;
25
```

Explanation:

At line 22, Polyspace Verifier shows an error. According to the declaration of *CZERO* (line 18), *UN* is not a valid field record component of the variable.

4.4.14. Dimension versus definition check: COR

Checks the compatibility of array dimension in relation to their definition.

ADA example:

```
1       package DIMDEF is
2           PROCEDURE MAIN;
3           FUNCTION Random RETURN boolean;
4       end DIMDEF;
5
6       package body DIMDEF is
7
8           SUBTYPE ST IS INTEGER RANGE 4 .. 8;
9           TYPE BASE IS ARRAY(ST RANGE <>, ST RANGE <>) OF INTEGER;
10          SUBTYPE TBASE IS BASE(5 .. 7, 5 .. 7);
11
12          FUNCTION IDENT_INT(VAL : INTEGER) RETURN INTEGER IS
13          BEGIN
14              RETURN VAL;
15          END IDENT_INT;
16
17          PROCEDURE MAIN IS
18              NEWARRAY    : TBASE;
19          BEGIN
20              IF RANDOM THEN
21                  NEWARRAY := (7 .. IDENT_INT(9) => (5 .. 7 => 4)); -- COR Error: Dimension is
not compatible with definition
22              END IF;
23              IF Random THEN
24                  NEWARRAY := (5 .. 7 => (IDENT_INT(3) .. 5 => 5)); -- COR Error: Dimension is
not compatible with definition
25              END IF;
26          END MAIN;
27
28      END DIMDEF;
```

Explanation:

At line 21 and 24, is put forward by PolySpace Verifier the incorrect dimension of the double array *Newarray* of type *TBASE*.

4.4.15. Aggregate versus definition check: COR

Checks the correctness condition on aggregate declaration in relation to their definition.

ADA example:

```
1
2      package AGGDEF is
3         PROCEDURE MAIN;
4         PROCEDURE COMMENT (A: STRING);
5         function RANDOM return BOOLEAN;
6      end AGGDEF;
7
8      package body AGGDEF is
9
10        TYPE REC1 (DISC : INTEGER := 5) IS RECORD
11           NULL;
12        END RECORD;
13
14        TYPE REC2 (DISC : INTEGER) IS RECORD
15           NULL;
16        END RECORD;
17
18        TYPE REC3 is RECORD
19           COMP1 : REC1(6);
20           COMP2 : REC2(6);
21        END RECORD;
22
23        FUNCTION IDENT_INT(VAL : INTEGER) RETURN INTEGER IS
24        BEGIN
25           RETURN VAL;
26        END IDENT_INT;
27
28        PROCEDURE AGGDEF_INIT is -- AGGREGATE INITIALISATION
29           OBJ3 : REC3;
30        BEGIN
31           if random then
32              OBJ3 :=
33                 ((DISC => IDENT_INT(7)), (DISC => IDENT_INT(7))); -- COR ERROR: Aggregate is
not compatible with definition
34           end if;
35           IF OBJ3 = ((DISC => 7), (DISC => 7)) then          -- COR ERROR: Aggregate is
not compatible with definition
36              COMMENT ("PREVENTING DEAD VARIABLE OPTIMIZATION");
37           END IF;
38        END AGGDEF_INIT;
39
40        PROCEDURE MAIN IS
41        BEGIN
42           AGGDEF_INIT; -- NTC ERROR: propagation of COR ERROR
43        END MAIN;
44     end AGGDEF;
```

Explanation:

At line 33 and 35, PolySpace shows the incompatible aggregate declaration on *OBJ3*. The aggregate definition with a discriminant of value *6*, is not compatible with a discriminant of value *7*.

4.4.16. Aggregate array length check: COR

Checks the length for array aggregate.
ADA example:

```
1     package AGGLEN is
2         PROCEDURE MAIN;
3         PROCEDURE COMMENT(A: STRING);
4     end AGGLEN;
5
6     package body AGGLEN is
7
8         SUBTYPE SLENGTH IS INTEGER RANGE 1..5;
9         TYPE SL_ARR IS ARRAY (SLENGTH RANGE <>) OF INTEGER;
10
11        F1_CONS : INTEGER := 2;
12        FUNCTION FUNC1 RETURN INTEGER IS
13        BEGIN
14            F1_CONS := F1_CONS - 1;
15            RETURN F1_CONS;
16        END FUNC1;
17
18
19        TYPE CONSR (DISC : INTEGER := 1) IS
20            RECORD
21                FIELD1 : SL_ARR (FUNC1 .. DISC);    -- FUNC1 EVALUATED.
22            END RECORD;
23
24        PROCEDURE MAIN IS
25
26        BEGIN
27            DECLARE
28                TYPE ACC_CONSR IS ACCESS CONSR;
29                X : ACC_CONSR;
30            BEGIN
31                X := NEW CONSR;
32                BEGIN
33                    IF X.ALL /= (3, (5 => 1)) THEN -- COR ERROR: Illegal Length for array
aggregate
34                        COMMENT ("IRRELEVANT");
35                    END IF;
36                END;
37            END;
38        END MAIN;
39
40    END AGGLEN;
```

Explanation:
At line 33, PolySpace shows an error. The static aggregate length is not compatible with the definition of *X* at line 31.

4.4.17. Sub-Aggregates dimension check: COR

Checks the dimension of sub-aggregates.

ADA example:

```
1
2      package SUBDIM is
3         PROCEDURE MAIN;
4         FUNCTION EQUAL ( A : Integer; B : Integer) return Boolean;
5      end SUBDIM;
6
7      package body SUBDIM is
8
9
10        TYPE DOUBLE_TABLE IS ARRAY(INTEGER RANGE <>, INTEGER RANGE <>) OF INTEGER;
11        TYPE CHOICE_INDEX IS (H, I);
12        TYPE CHOICE_CNTR  IS ARRAY(CHOICE_INDEX) OF INTEGER;
13
14        CNTR : CHOICE_CNTR := (CHOICE_INDEX => 0);
15
16        FUNCTION CALC (A : CHOICE_INDEX; B : INTEGER)
17                      RETURN INTEGER IS
18        BEGIN
19           CNTR(A) := CNTR(A) + 1;
20           RETURN B;
21        END CALC;
22
23        PROCEDURE MAIN IS
24           A1 : DOUBLE_TABLE(1 .. 3, 2 .. 5);
25        BEGIN
26           CNTR := (CHOICE_INDEX => 1);
27           if (EQUAL(CNTR(H),CNTR(I))) then
28              A1 := (                                 -- COR ERROR: Sub-agreggates do not
have the same dimension
29                    1 => (CALC(H,2) .. CALC(I,5) => -4),
30                    2 => (CALC(H,3) .. CALC(I,6) => -5),
31                    3 => (CALC(H,2) .. CALC(I,5) => -3) );
32           END IF;
33        END MAIN;
34
35     end SUBDIM;
```

Explanation:

At line 28, PolySpace shows an error. One of the sub-aggregates declarations of *A1* is not compatible with its definition. The second sub-aggregates does not respect the dimension defined at line 24.

Sub-aggregates must be singular.

4.4.18. Characters check: COR

Checks the construction using the *character* type.
ADA example:

```
1
2      package CHAR is
3         procedure Main;
4         function Random return Boolean;
5      end CHAR;
6
7
8      package body CHAR is
9
10        type ALL_Char is array (Integer) of Character;
11        TYPE Sub_Character is new Character range 'A' .. 'E';
12        TYPE TabC is array (1 .. 5) of Sub_Character;
13
14        FUNCTION  INIT return character is
15          VAR : TabC := "abcdf"; -- COR Error: Character is not in range 'A' .. 'E'
16        begin
17          return 'A';
18        end;
19
20        procedure MAIN is
21            Var  : ALL_Char;
22        BEGIN
23          IF RANDOM THEN
24            Var(1) := Init;      -- NTC ERROR: propagation of the COR error
25          ELSE
26            Var(Integer) := ""; -- COR ERROR: the 'null' string literal is not allowed here
27          END IF;
28        END MAIN;
29     END CHAR;
```

Explanation:
At line 15, PolySpace prompts that the assigned array is not within the range of the *Sub_Character* type. Moreover, any of the character values of *VAR* does not match any value in the range *'A' ..'E'*.
At line 26, a particular detection is made by Verifier when the *null string literal* is assigned incorrectly.

4.4.19. Accessibility level on access type: COR

Checks the accessibility level on an access type. This check is defined in Ada Standard at chapter 3.10.2-29a1. It detects errors when an access pointer refers to a bad reference.
Ada Example:

```
1
2      package CORACCESS is
3          procedure main;
4          function Brand return Boolean;
5      end CORACCESS;
6
7      package body CORACCESS is
8          procedure main is
9
10             type T is new Integer;
11             type A is access all T;
12             Ref : A;
13
14             procedure Proc1(Ptr : access T) is
15             begin
16                 Ref := A(Ptr);   -- COR Verified: Accessibility level deeper than
that of access type
17             end;
18
19             procedure Proc2(Ptr : access T) is
20             begin
21                 Ref := A(Ptr);   -- COR ERROR: Accessibility level not deeper than
that of access type
22             end;
23
24             procedure Proc3(Ptr : access T) is
25             begin
26                 Ref := A(Ptr);   -- COR Warning: Accessibility level might be deeper
than that of access type
27             end;
28
29             X : aliased T := 1;
30         begin
31           declare
32               Y : aliased T := 2;
33           begin
34               Proc1(X'Access);
35               if BRand then
36                   Proc2(Y'Access); -- NTC ERROR: propagation of error at line 22
37               elsif BRand then
38                   Proc3(Y'Access); -- NTC ERROR: propagation of error at line 27
39               end if;
```

```
40          end;
41          Proc3(X'Access);
42      end main;
43    end CORACCESS;
44
```

Explanation:

In the example above at line 17: *Ref* is set to *x'access* and *Ref* is defined in same block or in a deeper one. This is authorised. On the other hand, *y* is not defined in a block deeper or inside the one in which *Ref* is defined. So, at the end of block, *y* does not exist any more and *Ref* is supposed to points to on *y*. It it prohibited and PolySpace checks at line 22 and 27.

Note: the warning at line 27 is due to the combination of a red check because of *y'access* at line 39 and a green one for *x'access* at line 42.

4.4.20. Valid variable: COR

Checks the validity of a variable. This check is defined in Ada Standard at chapter 13.9.1 and 13.9.2. It verifies the validity of variables in two following cases:

1. On results of *unchecked_conversion* on scalar type with representation clause.
2. In a case argument.

Ada Example:

```
1
2      package CORVAR is
3          procedure main;
4      end CORVAR;
5
6      with Ada.Unchecked_Conversion;
7      package body CORVAR is
8          type F is range 1..10;
9          type E is (A,B,C);
10         for E use (A => 1, B => 3, C => 4);
11
12         -- subtype F is E (A,C);
13
14         function I_E is new Ada.Unchecked_Conversion (Integer,E);
15         function random return F is separate;
16         vf : F;
17         ve : E;
18
19         procedure main is
20         begin
21            vf := random;
22            ve := I_E (3); -- COR Warning: variable might be not valid
23
24         case vf is            -- COR Warning: variable might be not valid
25            when 1 => null;
26            when 2 => null;
27            when 3 => null;
28            when 4 => null;
29            when others => null;
30         end case;
31      end main;
```

```
32    end CORVAR;
33
34
35
```

Explanation:

At lines 22 and 24, PolySpace checks the validity of variables. The check is always orange as PolySpace is not precise for theses particular constructions.

4.4.21. Explicit dereference of a null pointer: COR

When a pointer is dereferenced, PolySpace checks if it is not a null pointer.
Ada Example:

```
1       package CORNULL is
2           procedure main;
3       end CORNULL;
4
5       package body CORNULL is
6           type ptr_type is access all integer;
7           ptr : ptr_type;
8           A : aliased integer := 10;
9
10          procedure main is
11          begin
12              ptr := A'access;
13              if (ptr /= null) then
14                  ptr.all := ptr.all + 1;        -- COR Warning: Explicit dereference of
possibly null value
15                  pragma assert (ptr.all = 10); -- COR Warning: Explicit dereference of
possibly null value
16                  null;
17              end if;
18          end main;
19      end CORNULL;
20
```

Explanation:
At line 14 and line 15, PolySpace checks the null value of *ptr* pointer. As PolySpace does not have a pointer analysis, it is not able to be precise on such construction.
These checks are currently always orange.

4.4.22. Accessibility of a tagged type: COR

Checks if a tag belongs to a tagged type hierarchy. This check is defined in Ada Standard at chapter 4.6 (paragraph 42).
It detects errors when a Tag of an operand does not refer to class-wide inheritance hierarchy.
Ada Example:

```
1      package TAG is
2
3          type Tag_Type is tagged record
4              C1 : Natural;
5          end record;
6
7          type DTag_Type is new Tag_Type with record
8              C2 : Float;
9          end record;
10
11         type DDTag_Type is new DTag_Type with record
12             C3 : Boolean;
13         end record;
14
15         procedure Main;
16
17     end TAG;
18
19
20     package body TAG is
21
22         procedure Main is
23             Y : DTag_Type := DTag_Type'(C1 => 1, C2 => 1.1);
24             Z : DTag_Type := DTag_Type'(C1 => 2, C2 => 2.2);
25
26             W : Tag_Type'Class := Z; -- W can represent any object
27                                      -- in the hierarchy rooted at Tag_Type
28         begin
29             Y := DTag_Type(W); -- COR Warning: Tag might be correct
30             null;
31         end Main;
32
33     end TAG;
```

Explanation:

In the previous example *W* represents any object in the hierarchy rooted at *Tag_Type*.

At line 29, a check is made that the tag of *W* is either a tag of *DTag_Type* or *DDTag_Type*. In this example, the check should be green, *W* belongs to the hierarchy.

PolySpace is not precise on tagged types and currently always flags it as a COR warning.

4.4.23. Power Arithmetic: POW

Check to establish whether the standard power integer or float function is used with an acceptable (positive) argument.

Ada Example:

```
1       With Ada.Numerics; Use Ada.Numerics;
2       With Ada.Numerics.Aux; Use Ada.Numerics.Aux;
3
4       package POWF is
5           function Bool_Random return Boolean;
6           procedure MAIN;
7       end POWF;
8
9       package body POWF is
10
11          procedure MAIN is
12              IRes, ILeft, IRight : Integer;
13              Res, Dbl_Random : Double ;
14              pragma Volatile(Dbl_Random);
15          begin
16              -- Implementation of Power arithmetic function with **
17              if (Bool_Random) then
18                  ILeft := 0;
19                  IRight := -1;
20                  IRes:= ILeft ** IRight;          -- POW ERROR: Power must be positive
21              end if;
22              if (Bool_Random) then
23                  ILeft := -2;
24                  IRight := -1;
25                  IRes:= ILeft ** IRight;          -- POW ERROR: Power must be positive
26              end if;
27
28              ILeft := 2e8;
29              IRight := 2;
30              IRes:= ILeft ** IRight;              -- otherwise OVFL Warning
31
32              -- Implementation with double
33              Res := Pow (dbl_Random, dbl_Random); -- POW Warning : may be not positive
34          end MAIN;
35      end POWF;
```

Explanation:

An error occurs on the power function on integer values "**" with respect to the values of the left and right parameters when *left <= 0 and right < 0*. Otherwise, PolySpace prompts the user to investigate further by means of an orange check.
Note: As preconised by the Standard, PolySpace set a green check on the instruction *left**right* with *left:=right:=0*.

4.4.24. User Assertion: ASRT

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

ADA Example:

```
1
2      package ASRT is
3          function Bool_Random return Boolean;
4          procedure MAIN;
5      end ASRT;
6
7      package body ASRT is
8
9          subtype Intpos is Integer range 0..Integer'Last;
10         subtype TenInt is Integer range 1..10;
11
12         Val_Constant : constant Boolean := True;
13         procedure MAIN is
14            -- Init variables
15            Flip_Flop, Flip_Or_val : Boolean;
16            Ten_Random, Ten_Positive : TenInt;
17            pragma Volatile_ada.htm (ten_random);
18         begin
19
20            if (Bool_Random) then
21                -- Flip_Flop is randomly be True or False
22                Flip_Flop := bool_random;
23
24                -- Flip_Or_Val is always True
25                Flip_Or_Val := Flip_Flop or Val_Constant;
26                pragma assert(flip_flop=True or flip_flop=False); -- User assertion is verified
27                pragma assert(Flip_Or_Val=False); -- ASRT ERROR: User assertion fails
28            end if;
29            if (Bool_Random) then
30                ten_positive := Ten_random;
31                pragma assert(ten_positive > 5);   -- ASRT Warning: User assertion may fail
32                pragma assert(ten_positive > 5);   -- User assertion is verified
33                pragma assert(ten_Positive <= 5);  -- ASRT ERROR: Failure User Assert
34            end if;
35
36         end MAIN;
37
38     end ASRT; -- End Package
```

Explanation:

In the *ASRT.ASRT* function, *pragma assert* is used in two different manners:

1. To establish whether the values *flip_flop* and *var_flip* in the program are inside the domain which that the program is designed to handle. If the values were outside the range implied by the assert, then the program wouldn't be able to run properly. Thus they are flagged as run-time errors.

2. To redefine the range of variables as shown at line 32 where *ASRT.Ten_positive* is restrained to only a few values. Indeed, PolySpace makes the assumption that if the program is executed with no run time error at line 32, *Ten_positive*

can only have a value greater than 5 after the line.

4.4.25. Non Terminations: Calls and Loops

NTC and NTL are only informative red checks.

- They are the only red errors which can be filtered out using the filters shown below
- They don't stop the analysis
- As other reds, code placed after them are grey (unreachable): the only colour they can take is red. They are not "orange" NTL or NTC
- They can reveal a bug, or can simply just be informative

NTL	A NTL is a loop for which the break condition is never met. Among NTLs, you will find the following examples: **while**(1=1) loop function_call; end loop; // informative NTL **while**(x >=0) loop x := x+1; end loop; // with x as an unsigned int could reveal a bug, or not (an unsigned is always positive) **for** I in 0 .. 10 loop my_array(i) = 10; end loop; // with "my_array is integer in 0..9" this red NTL reveals a bug in the array access, flagged in orange
NTC	Your function called "test" calls f;. And "**f;**" is flagged as a **red NTC**. Why? There could be five distinct explanations for this NTC: 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 : it is either red or green: for this call, it makes the function crash.

Note on NTC: some information can be given when clicking on the NTC

The list of so-called "non satisfiable constraints" represents the list of variables that cause the red error inside the function. The (potentially) long list of variables is useful to understand the cause of the red NTC, as it gives the conditions causing the NTC: it can be a list of variables (global or not):

- with a given value;
- which are not initialized. Perhaps the variables are initialized outside the set of analyzed files.

Solution

Carefully check the reasons with relation to your situation.

> Note   If you can identify a function that does not terminate (loop, exit procedure) you may wish to use the -known-NTC function. You will find all the NTCs and their consequences in the known-NTC Viewer, allowing you to filter them. Benefit : you can focus on NTCs you did not expect.

Related subjects :

4.4.25.1. 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 the caller.

Ada example:

```
1     package NTC is
2         procedure MAIN;
3         -- Stubbed function
4         function Random_Boolean return Boolean;
5     end NTC;
6
7     package body NTC is
8
9         procedure FOO (X : Integer) is
10            Y : Integer;
11        begin
12            Y := 1 / X;          -- ZDV Warning: Scalar division by zero may occur
13            while (X >= 0) loop -- NTL ERROR: Loop never terminate
14                if ( Y /= X) then
15                    Y := 1 / (Y-X);
16                end if;
17            end loop;
18        end FOO;
19
20        procedure MAIN is
21        begin
22            if (Random_Boolean)  then
23                FOO(0);          -- NTC ERROR: because of zero Division in NTC.FOO (ZDV)
24            end if ;
25            if (Random_Boolean)  then
26                FOO(2);          -- NTC ERROR: Non Termination Loop in NTC.FOO (NTL)
27            end if;
28        end MAIN;
29    end NTC;
```

Explanation:

In this example, the function NTC.FOO is called twice and neither of these 2 calls ever terminates:

1. The first never returns because of a division by zero (ZDV warning) at line 12 when *X = 0*.

2. The second never terminates because of an infinite loop (red NTL) at line 13.

As an aside, note that pragma Volatile_ada.htm is used to randomly initialize a variable.

Note: a NTC check can only be red.

4.4.25.2. Non Termination of Call due to entry in tasks

Tasks or entry points are called by PolySpace at the end of the main subprogram (which is executed sequentially) at the same time (the main subprogram must terminate).

In Ada language, explicit task constructs which are automatically detected by PolySpace are also called at the end of the main subprogram. An Ada program whose main subprogram calls a task entry, for instance, violates this model. PolySpace signals violations of this hypothesis, by indicating an NTC on an entry call (or "rendez-vous") performed in the main.

In the PolySpace model, the main procedure is executed first before any other task is started.

Example:

```
1      package NTC_entry is
2
3         TASK TYPE MyTask IS
4            ENTRY START;
5            ENTRY V842;
6            END MyTask;
7            procedure Main;
8            A : Integer;
9      end NTC_entry;
10
11     package body NTC_entry is
12
13        task body MyTask is
14        begin
15           accept Start;
16           A := A + 1;                      -- Grey code
17           accept V842;
18           A := A - 1;                      -- Grey code
19           accept V842;
20           A := A + 1;                      -- Grey code
21           accept V842;
22           A := A - 1;                      -- Grey code
23        end MyTask;
24
25        procedure Main is
```

```
26           T1 : MyTask;
27        begin
28           A := 0;
29           T1.Start;                      -- NTC ERROR: entry
task in the main
30           T1.V842;
31           T1.V842;
32           T1.V842;
33           pragma Assert(A=0);            -- Grey code
34        end Main;
35     end NTC_entry;
```

Using the launching command `polyspace-ada95 –main NTC_entry.main` on the previous example leads to a red NTC in the main procedure and grey code on the main task body `MyTask`.

The only way to analyse this code with PolySpace, is to add another main procedure with a null body and considers the NTC_entry.main as a task.

```
Package mymain is Procedure null_main; End mymain;
```

The previous small piece of code added and the usage of the launching command `polyspace-ada95 –main mymain.null_main.`[–entry-points]` NTC_entry.main` allow removing the red NTC in `NTC_entry.main` and grey code in the body of `MyTask`.

Another example concerns the call of an accept "rendez-vous" in the task body from the main (using `–main main.main`):

```
--package body main is
  procedure main is
  begin
  depend.controleur.demarrer; -- red NTC because of the call to a task is
called by the main
  end main;
--end main;
with Text_Io;
package body depend is

  task body controleur is
  date : Integer := 0;
  init_date: Integer;

  begin
  loop
  select
  accept demarrer;
  if (date = 0) then
  init_date := 10;
  end if ;
  date := init_date ;
Text_Io.Put_Line ("bonjour ....");
```

```
      exit;
      end select;
      end loop;
      end;

 end depend;
```

4.4.25.3. Known Non Termination of Call: k-NTC

By using the **-known-NTC** option with a specified function at launch time, is possible to transform an NTC Check for a non termination of call to a k-NTC check. Like an NTC check, k-NTC checks are propagated to their callers.

Function which are designed to be non terminating can be filtered out during the analysis of results through the use of the appropriate filter in the viewer, in conjunction with the **-known-NTC** option at launch.

Ada example:

```ada
1     package KNTC is
2         procedure Put_io (X : Integer);
3         procedure get_data(Data : out Float; Status : out Integer);
4         procedure store_data(Data : in Float);
5         procedure SysHalt(Value : Integer);
6         procedure MAIN;
7     end KNTC;
8
9     package body KNTC is
10
11        -- known NTC function
12        procedure SysHalt(Value : Integer) is
13        begin
14           Put_io(Value);
15           loop  -- Never terminate loop
16              null;
17           end loop;
18        end SysHalt;
19
20        procedure MAIN is
21           Status : Integer := 1;
22           Data : Float;
23        begin
24
25           while(Status = 1) loop
26              -- get data
27              get_data(Data, Status);
28              if (status = 1) then
29                 store_data(data);
30              end if;
31              if (Status = 0) then
32                 SysHalt(1);  -- k-NTC check: Call never terminate
33              end if;
```

```
34            end loop;
35        end MAIN;
36    end KNTC;
```

Explanation:
In the above example, the **-known-NTC "KNTC.SysHalt"** option has been added at launch time, transforming corresponding NTC checks to k-NTC one.

4.4.25.4. Non Termination of Loop: NTL

Check to establish whether a loop (for,do-while, while) terminates.
Ada example:

```
1
2       package NTL is
3           procedure MAIN;
4           -- Prototypes stubbed as pure functions
5           procedure Send_Data (Data : in Float);
6           procedure Update_Alpha (A : in Float);
7       end NTL;
8
9       package body NTL is
10
11          procedure MAIN is
12              Acq, Vacq : Float;
13              pragma Volatile_ada.htm (Vacq);
14              -- Init variables
15              Alpha     : Float := 0.85;
16              Filtered  : Float := 0.0;
17          begin
18              loop                       -- NTL information: Loop never terminates
19                  -- Acquisition
20                  Acq := Vacq;
21                  -- Treatment
22                  Filtered := Alpha * Acq + (1.0 - Alpha) * Filtered;
23                  -- Action
24                  Send_Data(Filtered);
25                  Update_Alpha(Alpha);
26              end loop;
27          end MAIN;
28      end NTL;
29
```

Explanation:
In the above example, the "continuation condition" of the while 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 treat those cases.

Another NTL example: error propagation

Like all other red errors, Verifier does not continue furthermore the analysis in the current branch even the **-continue-with-red-error** option. Due to the inside error, the (for, do-while, while) loop never terminates.

```
1       package NTLDO is
2           procedure MAIN;
3       end NTLDO;
4
5       package body NTLDO is
6           procedure MAIN is
7               A : array(1..20) of Float;
8               J : Integer;
9           begin
10              for I in A'First .. 21 loop -- NTL ERROR:    propagation of OVFL ERROR
```

```
11            A(I) := 0.0 ;              -- OVFL Warning: 20 verification with I in [1,20] and
one ERROR with I = 21
12              J := I + 1;
13          end loop;
14        end MAIN;
15    end NTLDO;
```

Note: a NTL check can only be red.

4.4.25.5. Sqrt, sin, cos, and generic elementary functions

When the analysed code uses some mathematical functions which are not supported by PolySpace, and there are always unproven checks about overflows when two variables - which have been derived from the results of mathematical functions such as "cos" are summed. The –voa option displays the *full range* for the potential return value of these functions.

This symptom can be seen when all mathematical functions are stubbed automatically which happens when the declarations of these functions for the compiler in use are slightly different from those assumed by PoySpace. The following solution matches the user's mathematical functions to PolySpace Verifier's equivalent function. Please note it has no impact on the original source code (no modification will be made).

Original code

```
package Types is
     subtype My_Float is Float range -100.0 .. 100.0;
end Types;


3     package Main is
4        procedure Main;
5     end Main;
6
7
8     with New_Math; use New_Math;
9     with Types; use Types;
10
11    package body Main is
12       procedure Main is
13       X : My_float;
14       begin
15          X := Cos(12.3); --voa displays [-1.0 .. 1.0]
16          X := Sin(12.3); --voa displays [-1.0 .. 1.0]
17          X ::= Sqrt(-1.5); --is red: NTC Error
18       end;
19    end Main;
```

Original maths package

```
with My_Specific_Math_Lib;
with Types; use Types;

package New_Math is
  function COS (X : My_Float) return My_Float renames My_specific_math_lib.
```

```
Cos;
 function SQRT (X : My_Float) return My_Float renames My_specific_math_lib.
sqrt;
 function SIN (X : My_Float) return My_Float renames My_specific_math_lib.
sin;
end New_Math;
```

Extra package

This package may be written by the user to include more precise modelling of the mathematical functions in the analysis.

```
WITH Ada.Numerics.Generic_Elementary_Functions;
with Types; use Types;

package My_specific_math_lib is new  Ada.Numerics.
Generic_Elementary_Functions(My_Float);
```

Important

Due to a lack of precision in some areas, PolySpace is not always able to indicate a red NTC check on mathematical functions even whereas a problem exists. By default it is important to consider each call to any mathematical functions as though it had been highlighted by an unproven check, and could therefore lead to a runtime error.

4.4.26. Unreachable code: UNR

Check to establish whether different code snippets (assignments, returns, conditional branches and function calls) are reached (Unreachable code is referred to as "dead code"). Dead code is represented by means of a grey colour on every check and an UNR check entry.

ADA Example:

```
1     package UNR is
2         type T_STATE is (Init, Wait, Intermediate, EndState);
3         function STATE (State : in T_STATE) return Boolean;
4         function Intermediate_State(I : in Integer) return T_STATE;
5         function UNR_I return Integer;
6         procedure MAIN;
7     end UNR;
8
9     package body UNR is
10
11        function STATE (State : IN T_STATE)  return Boolean is
12        begin
13           if State = Init then
14              return False;
15           end if ;
16           return True;
17        end STATE;
18
19        function UNR_I return Integer is
20           Res_End, Bool_Random : Boolean;
21           I : Integer;
22           Res_State : T_STATE;
23           pragma Volatile_ada.htm (bool_random);
24        begin
25           Res_End := STATE(Init);
26           if (Res_End = False) then
27              Res_End := State(EndState);
28              Res_State := Intermediate_State(0);
29              if (Res_End = True or else Res_State = Wait) then -- UNR code
30                 Res_State := EndState;
31              end if;
32              -- Use of I which is not initialized
33              if (Bool_Random) then
34                 Res_State := Intermediate_State(I); -- NIV ERROR
35                 if (Res_State = Intermediate) then  -- UNR code because of NIV error
36                    Res_State := EndState;
37                 end if;
38              end if;
39           else
40              -- UNR code
41              I := 1;
42              Res_State := Intermediate_State(I);
```

```
43            end if;
44            return I; -- NIV ERROR: because of UNR code
45        end UNR_I;
46
47        procedure MAIN is
48            I : Integer;
49        begin
50            I := UNR_I; -- NTC ERROR because of propagation
51        end MAIN;
52
53    end UNR;
54
55
56
```

Explanation:

The example illustrates three possible reasons why code might be unreachable, and hence be coloured grey.

1. As shown at line 26, the first branch is always true (*if-then part*) and so the other branch is never executed (*else* part at lines 40 to 42).

2. At line 29 a conditionnal part of a conditionnal branch is always true and the other part never evaluated because of the standard definition of logical operator *or else*.

3. The piece of code after a red error is never evaluated by Polyspace Verifier. The call to the function and the lines following line 34 are considered to be dead code. Correcting the red error and re-launching would allow the colour to be revised.

4.4.27. Value on Assignment: VOA

Check to establish the value taken by a variable on assignment. Such checks are only available when the **-voa** option is used at launch time.

At present, voa checks are only available on scalar variables. Some examples are given below.

ADA example:

```
1
2
3      Package VOA is
4
5          subtype T_NBWAY is Integer range 1..8;
6          subtype T_DIGITAL is Integer range 0..1;
7          subtype T_ANALOGIC is Float range -10.0 .. 10.0;
8          Zero_analogic : constant T_ANALOGIC
9            := (T_ANALOGIC'Last - T_ANALOGIC'First)/ 2.0 - T_ANALOGIC'Last;   --
10
11         function Get_Analogic (Way : T_NBWAY) return T_Analogic;
12         function Get_Digit  (Way : T_NBWAY) return T_Digital;
13
14         type VerifierColor is (Red, Green, Orange, Black);
15         type RECOR is
16            record
17               A : Float;
18               B : VerifierColor;
19            end record;
20         Var_rec : RECOR;
21
22         Procedure MAIN;
23
24      end VOA;
25
26     package body VOA is
27
28         Procedure MAIN is
29            Way_io : T_NBWAY := T_NBWAY'First;
30            Val_Sensor : T_ANALOGIC;
31            Val_Digit : T_DIGITAL;
32            volatile_Color : VerifierColor;
33            pragma Volatile_ada.htm(Volatile_color);
34            Volatile_ada.htm_Float : Float;
35            pragma Volatile_ada.htm(Volatile_Float);
36         begin
37
38            for I in T_NBWAY'Range loop
39               Val_Sensor := Get_Analogic(I); -- VOA: {-1E+1<=[expr]<=1E+1}
40               Val_Digit := Get_Digit(I); -- VOA: {0<=[expr]<=1}
41               if Val_Sensor < 0.0 then
42                  Val_Sensor := Zero_Analogic; -- VOA: {[expr]=0.0}
43               end if;
44            end loop;
45
46            -- Example
```

```
47          Var_Rec.A := Volatile_ada.htm_Float; -- VOA: {[expr]=float(32) range -3.41E
+38..3.4E+38}
48          Var_Rec.B := Volatile_ada.htm_color; -- VOA: {red<=[expr]<=black}
49
50          -- Other possible but intrusive way to know a specific value
51          pragma Inspection_Point (Way_io); -- inspection point computed range: {WAY_IO=1}
52
53      end MAIN;
54
55   End VOA;
```

Explanation:

As shown in the example, inspection points (IPT) can also be used to discover the range of a variable.

4.4.28. Inspection Points: IPT

The use of *pragma Inspection_Point (var)* as a code snippet (where *(var)* is a scalar variable) represents a request to compute the specific range of a variable by means of a pragma instruction. Refer to the example below.

Ada example:

```
1
2
3      Package IPT is
4
5          subtype T_NBWAY is Integer range 1..8;
6          subtype T_DIGITAL is Integer range 0..1;
7          subtype T_ANALOGIC is Float range -10.0 .. 10.0;
8          Zero_analogic : constant T_ANALOGIC
9            := (T_ANALOGIC'Last - T_ANALOGIC'First)/ 2.0 - T_ANALOGIC'Last;   --
10
11         function Get_Analogic (Way : T_NBWAY) return T_Analogic;
12         function Get_Digit  (Way : T_NBWAY) return T_Digital;
13
14         type VerifierColor is (Red, Green, Orange, Black);
15         type RECOR is
16            record
17               A : Float;
18               B : VerifierColor;
19            end record;
20         Var_rec : RECOR;
21
22         Procedure MAIN;
23
24     end IPT;
25
26     package body IPT is
27
28         Procedure MAIN is
29            Way_io : T_NBWAY := T_NBWAY'First;
30            Val_Sensor : T_ANALOGIC;
31            Val_Digit : T_DIGITAL;
32            volatile_Color : VerifierColor;
33            pragma Volatile_ada.htm(Volatile_color);
34            Volatile_ada.htm_Float : Float;
35            pragma Volatile_ada.htm(Volatile_Float);
36         begin
37
38            for I in T_NBWAY'Range loop
39               Val_Sensor := Get_Analogic(I);
40               pragma Inspection_Point (Val_Sensor);  -- IPT: {-1E+1<=VAL_SENSOR<=1E+1}
41               Val_Digit := Get_Digit(I);
42               pragma Inspection_Point (Val_Digit);   -- IPT: {0<=VAL_DIGIT<=1}
43            end loop;
44
45            -- Example on record
46            Var_Rec.A := Volatile_ada.htm_Float;
```

```
47          Var_Rec.B := Volatile_ada.htm_color;
48          pragma Inspection_Point (Var_Rec);          -- IPT currently ignored
49          pragma Inspection_Point (Volatile_ada.htm_color); -- IPT: {VOLATILE_COLOR=red..
black}
50          pragma Inspection_Point (Way_io);          -- IPT: {WAY_IO=1}
51
52      end MAIN;
53
54   End IPT;
```

Explanation:

Note that the inspection point at line 48 is ignored. Inspection points are available for scalar variables only.

4.5. Advanced results review

Related subjects :

4.5.1. Purpose of -continue-with-red-error option

Problem

I have a red error that appears to be dead code. Is this normal?

Explanation

- **Checks in uncalled code might be displayed in red.** PolySpace performs an upper approximation of variables so it may be true that Verifier believes it (is) possible to enter in a particular condition when it could not happen during "real life" execution. In the example below, there is an attempt to compare elements in the array, and PolySpace was 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;

- **Every colour in the graph on the right can be approximated by a colour immediately above it in the graph.** This is called imprecision. It is clear when green can be approximated by orange, also clear when red is approximated by orange, but looks strange when grey is involved. Every set of points can be approximated by a superset that is bigger than what might happen in reality. An empty set of points can be approximated:

    by an empty superset;
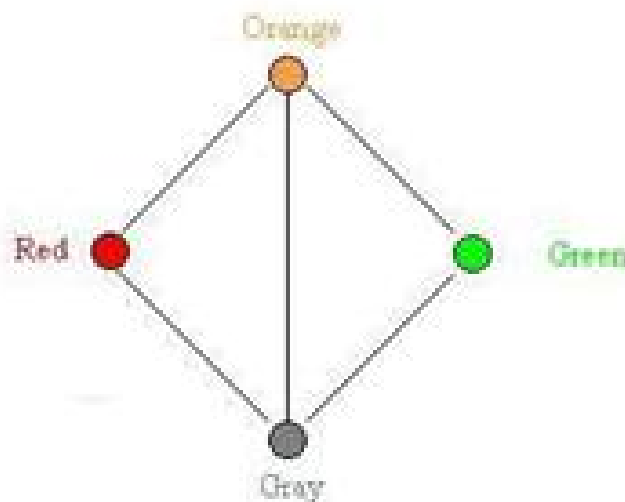
by a nonempty super set. In this case, the superset can have any position regarding the forbidden zone, therefore have any colour. That is the reason why PolySpace is not exhaustive on dead code.

- **Is it a problem to have grey code approximated by red?** In case of a red error, all executions that follow this branch of execution are stopped (that is the definition of PolySpace behaviour) because no valid execution can pass this point. This means that after the "if then" statement, no execution can continue with that condition met. This models the reality of the situation exactly.

Example

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

After the "if" statement, the only possibility to continue the execution alive is if the condition is false, otherwise, we would have a red error. (So) after this branch, it means that the condition is always false. That is the reason for the -continue-with-red-error option. You can continue analyzing your code, even with a certain error.

Remember that this propagates values throughout your application. None of the execution paths leading to the run-tim errror will continue after the error.

Solution

```
7      package body Main is
8         procedure Main is
9            X: array (1..5) of Integer;
10           Tmp: Integer;
11           Zero: Integer:= 0;
12        begin
13           X:= (1,2,3,4,5);
14           if (X(4) > X(5))
15           then
16              Tmp:= 1 / Zero;
17           end if;
18        end;
19
20     end;
```

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

4.5.2. Checks on procedure calls with default parameters

Some checks may be located on procedure calls. They correspond to default values assigned to parameters of a procedure.

Example:

```
1      package DCHECK is
2         type Pixel is
3            record
4               X : Integer;
5               Y : Integer;
6            end record;
7         procedure MAIN;
8
9         NError : Integer;
10        procedure Failure        (Val : Integer := Nerror);
11        procedure MessageFailure (str : String := "");
12     end DCHECK;
13
14     package body DCHECK is
15        type TwentyFloat is array (Integer range 1.. 20) of Float;
16
17        procedure AddPixelValue(Vpixel : Pixel) is
18        begin
19           if (Vpixel.X < 3) then
20              Failure;             -- NIV Verified: Variable is initialized
(Nerror)
21              MessageFailure;      -- COR Verified: Value is in range (string)
22           end if;
23        end AddPixelValue;
24
25        procedure MAIN is
26           B : Twentyfloat;
27           Vpixel : Pixel;
28        begin
29           NError := 12;
30           Vpixel.X := 1;
31           AddPixelValue(Vpixel);
32           NError := -1;
33           for I in 2 .. Twentyfloat'Last loop
34              if ((I mod 2) = 0) then
35                 B(I) := 0.0;
36                 if (I mod 2) /= 0 then
37                    Failure;          -- NIV Unreachable: Variable is not
initialized
38                    MessageFailure; -- COR Unreachable: Value is not in range
```

```
39                end if;
40             end if;
41          end loop;
42          MessageFailure("end of Main");
43       end MAIN;
44    end DCHECK;
```

In the previous example, at line 20 and 37, checks on the procedure calls `Failure` represent the check <u>NIV</u> made on the default parameter `Nerror` (a global parameter).

In the same way, <u>COR</u> checks at line 21 and 38 on `MessageFailure` represent verification made by PolySpace on the default assignment of a null string value on the input parameter.

Note: not all the checks have been moved to procedure calls. Checks remain on the procedure definition except for the following basic types and values:

- A numerical value (example: `1, 1.4`)
- A string (example: `"end of main"`)
- A character (example: `'A'`)
- A variable (example: `Nerror`).

4.5.3. _INIT_PROC procedures

In the PolySpace viewer, it could be possible to find nodes `_INIT_PROC$` in the "Procedural entities" view. As your compiler, PolySpace generate function `_INIT_PROC` for each record where initialization occurs. When a package define many records, each `_INIT_PROC` is differentiated by `$I` (`I in 1..n`).

Example:

```
1       package test is
2           procedure main;
3       end test;
4
5       package body test is
6
7           subtype range_0_3 is integer range 0..3;
8           Vg : Integer := 1;
9           Pragma Volatile( Vg );
10
11          function random return integer;
12          type my_rec1 is
13             record
14                a : integer := 2 + random;  -- Unproven OVFL coming from
_INIT_PROC procedure (initialization of V1)
15                b : float := 0.2;
16             end record;
17          V1 : my_rec1;
18          V2 : my_rec1 := (10, 10.10);
19
20          procedure main is
21             Function Random return Boolean;
22          begin
23             null;
24          end;
25      end test;
```

In the previous example, an unproven OVFL on the field `a` of record `my_rec1` has been detected when initialising the global variable `V1`. It initialises record of global variable `V1` at line 17. Indeed, `random` procedure could return any value in the integer type and so, leads to an overflow by adding to `2`. Check is located in the _INIT_PROC node into "Procedural entities" view.

5. Get more from PolySpace: Insert it into your development process

This section will be of interest to <u>Project managers, quality managers and developers</u> who are looking to understand PolySpace results, and are looking to optimise the timing of its use during the project development cycle. The document suggests how PolySpace might best be applied at each phase of a typical project lifecycle. The twin goals of productivity and quality are considered, and it is acknowledged that the criticality of the application will affect the balance between them.

However, the following assumes that the primary goal is to achieve maximum productivity with no quality defects. The document explains how to use PolySpace tools at each phase of the development cycle to aim for such a goal, with the financial implications of implementing each recommendation is left for assessment by the user.

How can I use PolySpace in my current process?

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

Process

How can I improve the use of PolySpace Verifier within my software development process?

PolySpace activities
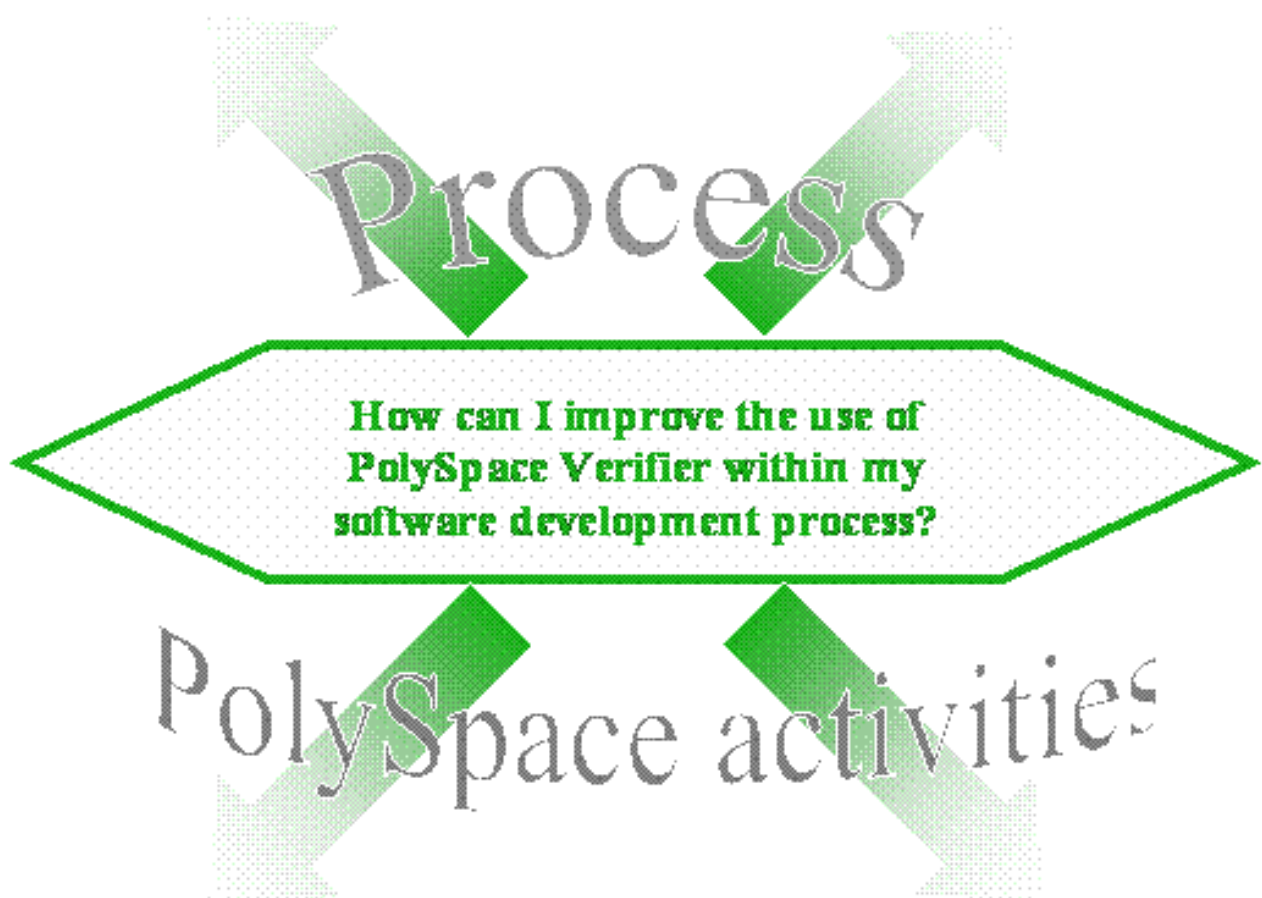
On given results, how can I find the maximum number of anomalies?

How can I get the best results?

This guide suggests answers to the following questions.
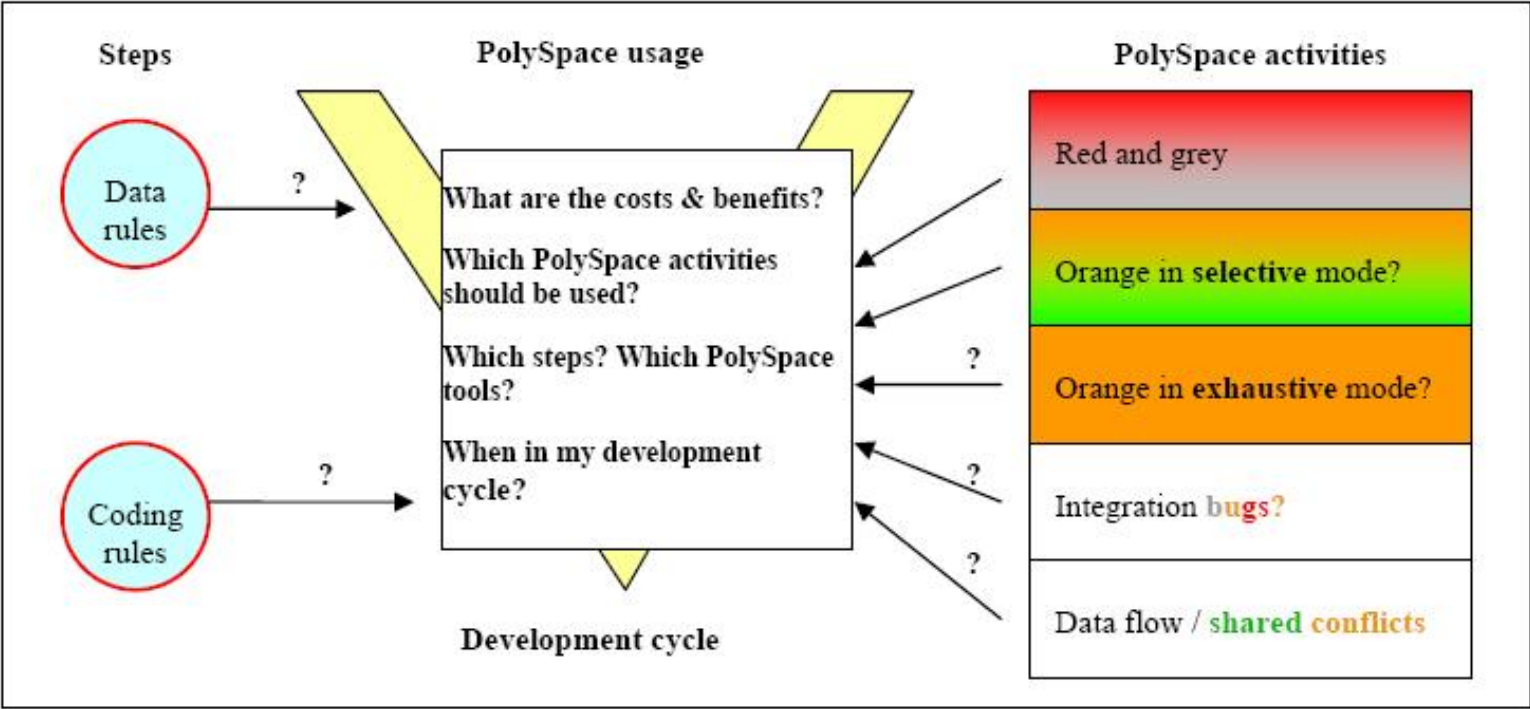
It answers those questions by means of the following topics: an explanation of each PolySpace approach. A "PolySpace approach" or "Approach" is defined in this context as the manner of use of PolySpace to achieve a particular goal, with reference to a collection of techniques and guiding principles. These include suggestions of different activities which might be completed before functional unit test or integration tests, depending on the development process:

- An explanation of the collection of techniques and guiding principles going to form each Approach.
- Fixing red and grey – review **run time errors** and **dead code** checks only
- Selective orange review – review **warnings** and find bugs quickly and efficiently. Suitable when time is short, and the aim is to maximise the number of bugs discovered.
- Exhaustive orange review – how much it costs and the value it brings at the unit phase and at the integration phase
- Shared data conflict detection – and the problems it can highlight
- Data flow analysis
- Integration bugs tracking

Related subjects :

5.1. PolySpace usages

PolySpace tools can support two main objectives concurrently.
Reduction of testing and validations costs
Improvement of the software quality
PolySpace can be used in different ways depending on the context, the primary difference being in the approach used to exploit the results generated. The following diagrams summarise the different approaches.
The aim here is not to compare the cost of certification processes, or of development processes with or without coding rules. The graphs aim to compare the costs of typical processes with and without PolySpace.

When no coding rules are adopted

During the coding activity, there are two recommended approaches:
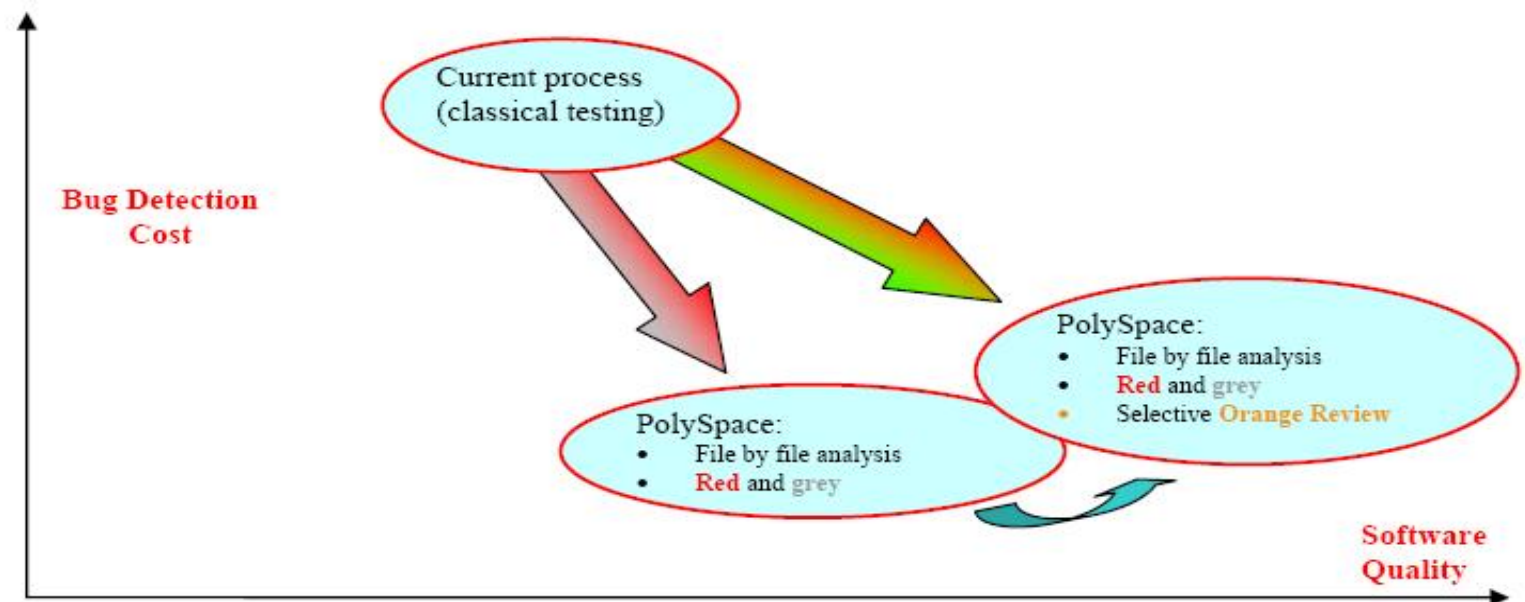
Note that the sentence in previous figure about "file by file analysis" needs to be understood as a "package by package analysis". Indeed, most of the time each package is developed in a file. The first approach is to use only the **red** and **grey** results: fix the **red bugs**, and check the **dead code** for abnormalities.

The second approach involves the same activities, and adds a partial review of the orange warnings. The aim is to find as many bugs as possible, with very limited efforts. This approach finds more bugs and therefore improves the quality. It does involve more effort, but the amount of time spent to find each bug remains very small.

Using PolySpace on one single package is efficient: even though there is no knowledge of the package context, experience shows that 50% of the bugs detected by PolySpace can be found locally. When it is has been successfully implemented, the development team can migrate to a more demanding (and more fruitful) level of usage of PolySpace. This migration is not always desirable; it of course depends on the project's context. Then, after coding, before the testing activity:
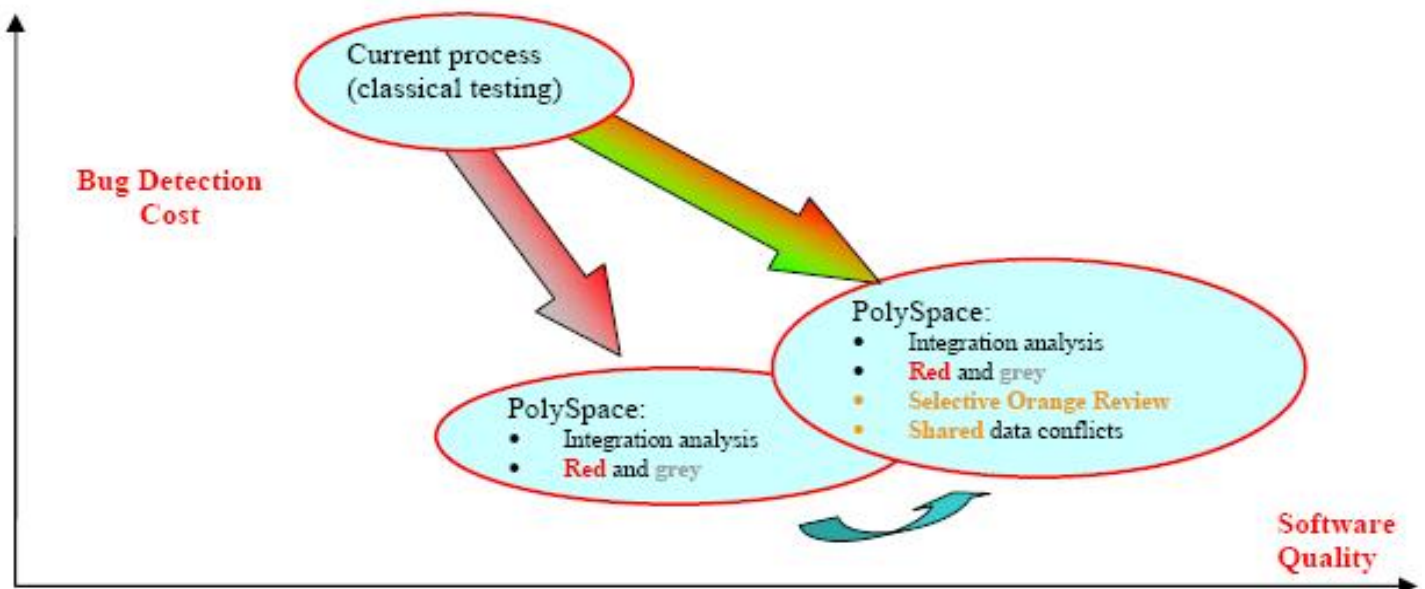
Again, the first approach is to use only the **red** and grey results: fix the **red bugs**, and check the dead code.
The second approach includes the same activities, and adds a partial review of the **orange warnings** and of the **orange shared data**.

When coding rules have been adopted

The main difference here by comparison with the previous processes is with respect to the cost of bug detection. When PolySpace is used in accordance with a set of coding rules, the bug detection cost is much lower. there are three recommended ways to use PolySpace, <u>during the coding activity</u>:
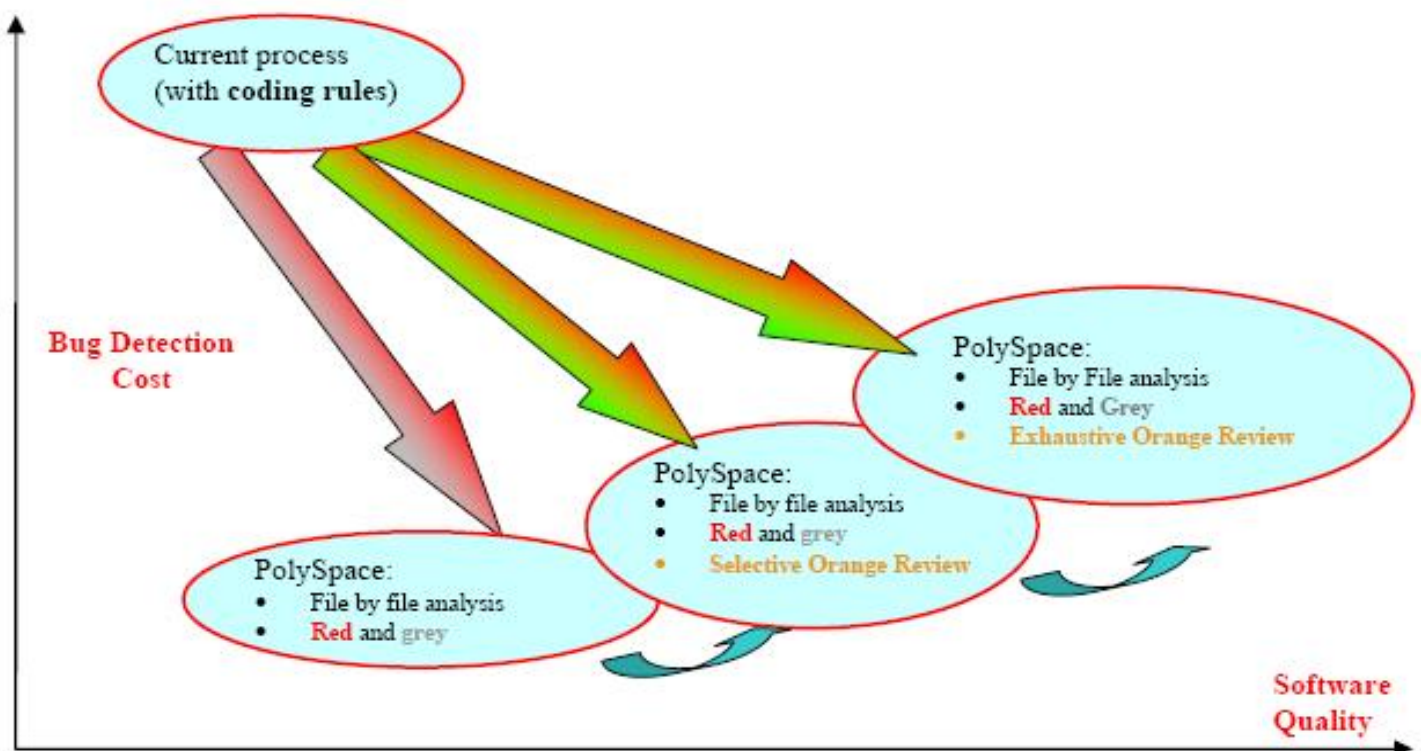
Compared to the previous situation (where no coding rules are in place), an additional possibility exists. Instead of reviewing only certain orange warnings in a file, all of them are systematically checked. This is possible as when the **right coding rules** are respected (see the end of this section for recommendations). That leads to there being only a few orange checks in a file, and therefore checking all of them is potentially very fruitful. A large proportion of those anomalies require some correction to the code, with some users reporting up to 50%.

Then, after coding, before the testing activity:
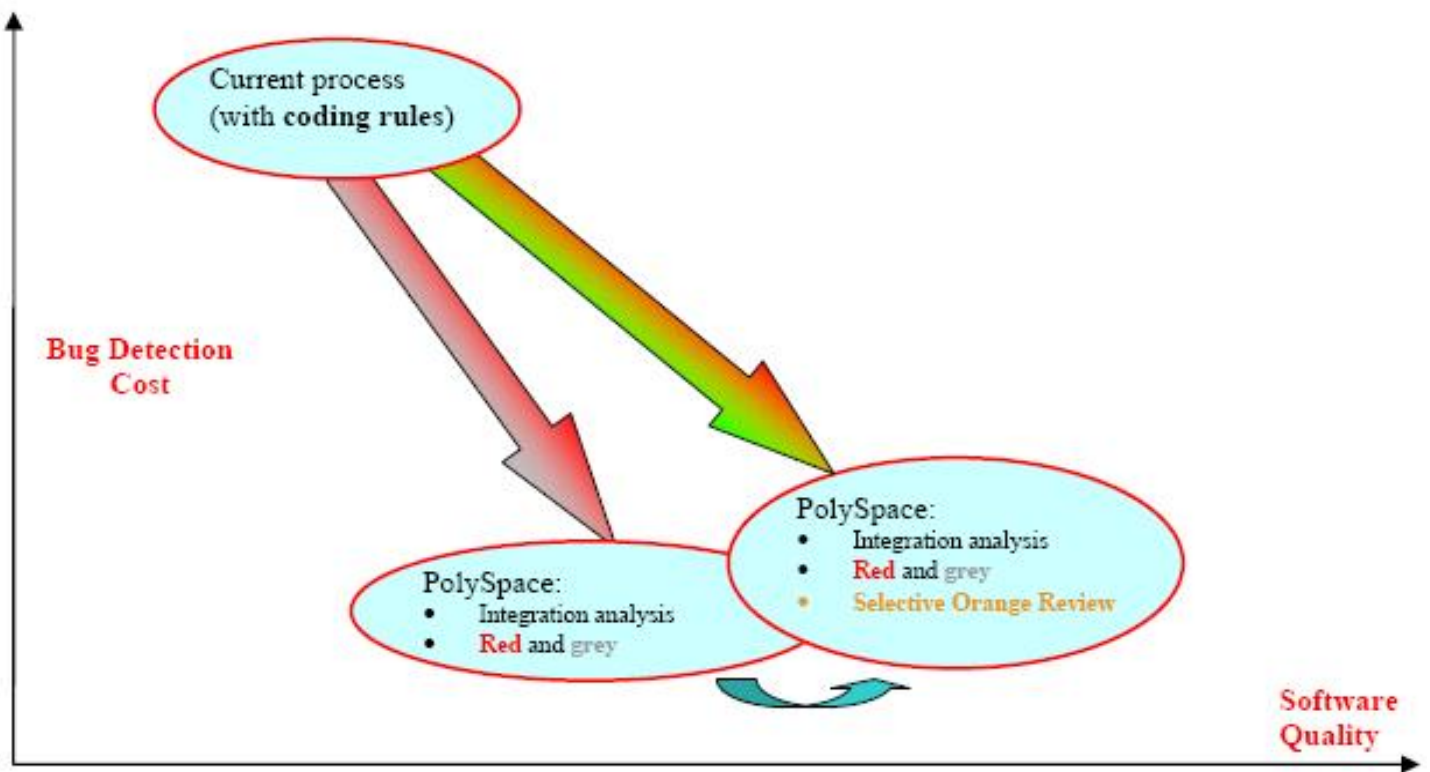
Note: It is also possible to migrate from a selective to an exhaustive **orange review** when performing an integration analysis, but this activity is very costly.

<u>In a certification context</u>

A "quality/qualification" approach where PolySpace replaces an existing activity. In this case quality is already high and maybe at a "zero defects" level, but PolySpace will reduce the cost of achieving such quality. In this context, PolySpace can replace the traditional time consuming control and data flow analysis, as well as shared data conflict detection.

<u>As an acceptance tool</u>

The fourth and last approach implies the use of PolySpace as a method of meeting an acceptance criterion.

5.2. Standard development process

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

The software development process

This section describes how to introduce PolySpace to a standard software development process. For instance,

- In Ada, no unit test tools or coverage tools are used: functional tests are performed just after coding

- In C, either no coding rules are present or they are not always followed.

The figure below illustrates the revised process, with PolySpace introduced in the tool chain. It will be used just before functional testing.
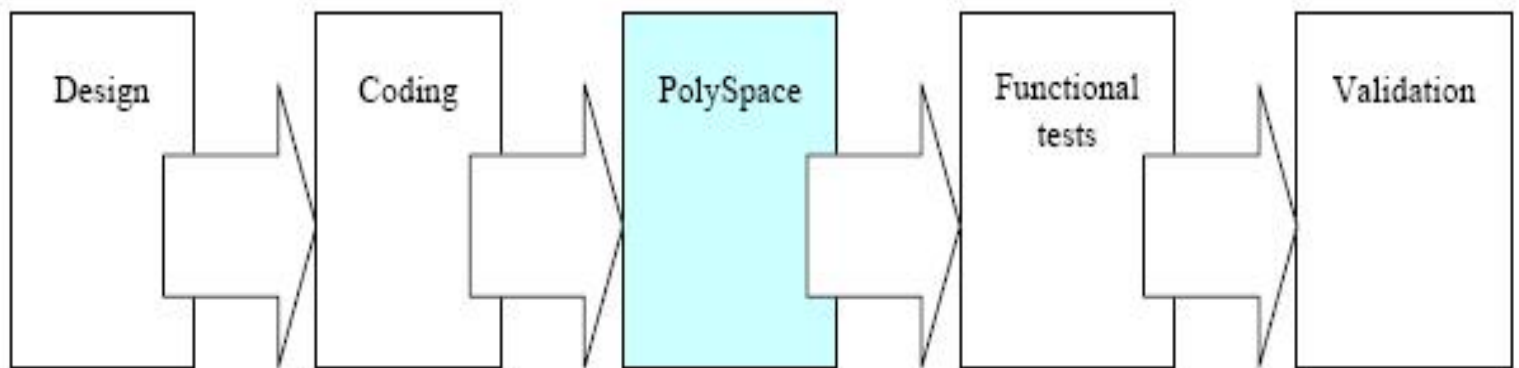
The objective of using PolySpace

PolySpace will be used to improve the software quality and productivity. It will help the developer to find and fix bugs much quicker than the existing process. It will also improve the software quality by finding bugs which would otherwise be likely to remain in the software after delivery.

It doesn't prove the robustness of the code because the prime objective is to deliver code of at least similar quality to before, but to ensure that code is produced in a predictable timeframe with controlled and minimized delay and costs. Another approach for this purpose is described in the next section.

The PolySpace approach

The way forward here is for PolySpace Desktop to be applied by developers or testers on a file-by-file/package-by-package analysis basis. The users will use the **default PolySpace Desktop options**, the most prominent feature of which is the automatically generated "main" function. This main will call all unused procedures and functions with full range parameters. The users will be required to fix **red** errors and examine **grey** code, and they will also do a **selective orange review**.
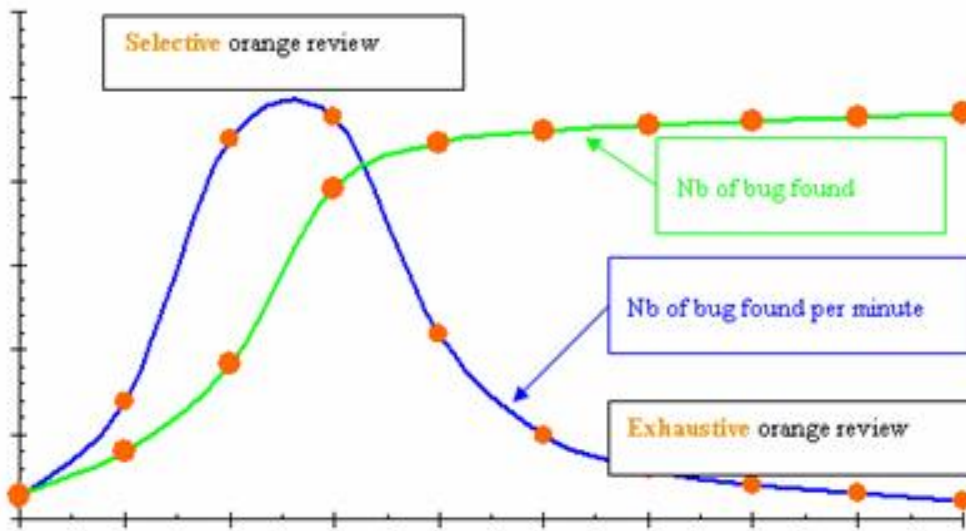
Cost/benefits of a selective orange review

This selective orange review can be applied on specific Run Time Error categories, such as "Out of Bound Array Index", or on all error categories. This depends on each individual developer's coding style.

It is true that with this approach some bugs might remain in the unchecked oranges, but it represents a significant move forward from the initial position. Coding rules would help further if more improvement is sought.

A complementary approach

A second approach is also possible which, unlike the first, focuses only on an increase in quality. If coding rules are applied, this second approach will turn into a cheap and productive one as described by the second arrow on the illustration.

Integration tests are also possible at this stage. This analysis will be performed by PolySpace on larger modules, and the orange review will be focused on **orange** Run Time errors **which were not examined** after the file-by-file/package-by-package analysis.

For instance, if the project construction is such that scalar overflows can only be reviewed at integration phase, then

- The user will ignore orange overflows with PolySpace Desktop when performing file-by-file analysis,

- He will examine them with PolySpace Verifier.

Integration with configuration management tools

PolySpace can also be used by project managers to establish and test for transition criteria to proceed to file check-in

- **Daily check-in**: PolySpace Desktop is applied to the file(s) currently under development. Compilation must complete without the permissive option.

- **Pre-unit test check-in**: PolySpace Desktop is applied to the file(s) currently under development.
- **Pre-integration test check-in**: PolySpace Verifier is applied to the whole project until compilation can complete without the permissive option. This stage will differ from the daily check-in activity because link errors will be highlighted here.
- **Pre-build for integration test check-in**: PolySpace Verifier is applied to the whole project, with all multi-tasking aspects accounted for as appropriate.

- **Pre-peer review check-in**: PolySpace Verifier is applied to the whole project, with all multi-tasking aspects accounted for as appropriate.

For each check-in activity mentioned above, the transition criterion could be: "No bug found within the allocated time defined by the process". For instance, if the process defines that 20 minutes should be dedicated to a selective review, the criterion could be: "no bug found during these 20 minutes".

Costs and benefits

Using PolySpace Desktop to find **unit/local bugs** in this way will both reduce the cost of the software and improve the quality:

- **Red** checks and bugs in grey checks. The number of bugs found thanks to these colours can vary from one user to another, but experience shows that on average, around ¼ of the analyses will reveal a **red** error (s) and/or will reveal bugs in grey code.

- **Orange checks**. Experience suggests that the time needed to find one bug per file varies from 5 minutes to 1 hour, and is typically around 30 minutes. This represents an average of two minutes per orange check review, and a total of 20 orange checks per package in Ada and 60 orange checks per file in C.

With this approach, using PolySpace to find **integration bugs** will increase the quality, but at a higher usage cost:

- **75% of bugs are local in this type of code**: the selective orange review at integration phase reveals a ¼ of integration bugs, and the rest (¾) of local bugs. Finding real integration bugs might require another process which requires coding rules to be efficient.

- **Setup time:** the time needed to setup the analysis can be higher due to a lack of coding rules. Code modifications might be needed. Most of these modifications cannot be automatic without changes in the process.

- **Anomalies and complexity:** In this configuration, any particular file will contain more oranges when analysed with PolySpace Verifier than with PolySpace Desktop (about twice as many). These oranges are likely to be anomalies, and will responsible for the orange check review becoming more time consuming.

- **A more stable software version implies a later analysis**. If PolySpace Verifier is used **instead of** PolySpace Desktop, bugs might be revealed much later because a more *complete* version of the software can only be provided at a later phase in the project.

- **An exhaustive orange review can take 25 men-days for a 50000 line project.** This would represent the effort where the aspiration is for bug free software, assuming that a 50000 line application contains about 3000 orange checks

5.3. Rigorous development process: introducing tools and coding rules

This is of interest for both project and quality managers, who are likely to be interested in this approach.

The software development process

This section describes how to use PolySpace within a process which has the following characteristics. In Ada, unit testing tools or coverage tools are used.

The picture below describes the new process, with PolySpace introduced into the tool chain. It will be used just before functional testing.
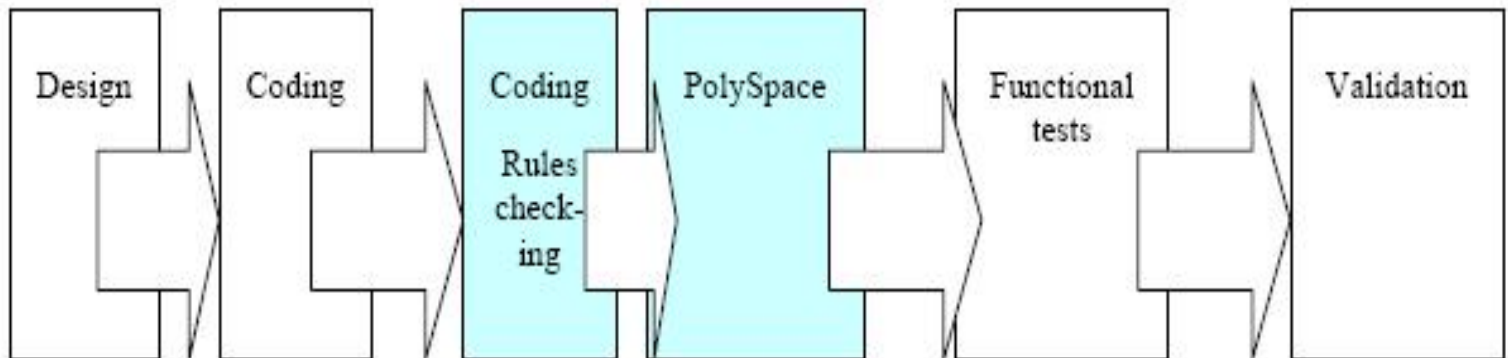
PolySpace will be used to increase both the software quality and its productivity.

The PolySpace approach

Use PolySpace Desktop on a file by file analysis basis.

- The "main" used to analyse each file is very often **automatically generated by the project**, and not by PolySpace Desktop (unlike the standard approach).

- **Initialisation ranges** should be applied to input data. For instance, if a variable "x" is read by functions in the file, and if x can be initialized to any value between 1 and 10, this information should be included as part of the analysis.

- [**Optional**] Some properties of output variables might be checked. For instance, if a variable "y" is returned by a function in the file and should always be returned with a value in the range 1 to 100, then PolySpace Desktop can flag instances where that range of values might be breached.

- Red errors will be fixed and grey code examined, and an **exhaustive orange review** will be completed.

- The usage of permissive options is not advisable at this stage.

> Note   The distinguishing feature for this approach as compared with the standard approach is that the orange check review **is exhaustive here**.

A complementary approach

A second approach is also possible. Use PolySpace Verifier at integration phase to track integration bugs, and review:-

- Red and grey integration checks;

- Orange checks on code which produced green checks when analysed by Desktop.
- The remaining orange checks with a selective review: *Integration bug tracking*.

Costs and benefits

With this approach, using PolySpace Desktop to find bugs will typically bring the following benefits

- 3-5 **orange checks** per file, 3 **grey checks** per file yielding an average of 1 bug per file. Typically, 2 of these oranges might represent the same bug, and another might represent an anomaly.

- An average of 2 analyses by PolySpace Desktop per file is typical before the file can be checked-in to the configuration management system.
- The average analysis time is about 15 minutes.

  Note   If the development process includes data rules which determine how the data flow are designed, the benefits might even be higher. The data rules would implicitly reduce the potential for PolySpace Verifier to find integration bugs.

With this approach, using PolySpace to find integration bugs might bring the following results. On a typical 50000 line project:

- A selective orange check review might reveal **one integration bug per hour of orange** code review and takes about after 6 hours, which long enough to review the main orange points throughout the whole application. This represents a step towards an exhaustive orange check review. Spending more time is unlikely to be efficient, and won't guarantee that no bugs remain.

- An exhaustive orange review takes between 4 and 6 days, given that a 50000 lines of code application might contain about 400-800 orange checks.

5.4. A quality/qualification approach

Quality managers are likely to be interested in this approach.

The software development process

This section describes how to use PolySpace within a process which includes coding and data rules. Such a process is typical of a *qualification* environment, with existing activities which must be performed. Before the introduction of PolySpace they will have been performed by hand, with classical testing methods, or using previous generation tools. PolySpace will **replace these activities**, and reduce the cost of the process.

PolySpace is not intended to improve the quality which is already at the desired level. It will complete the same tasks more efficiently, bringing improved productivity.

The objective of using PolySpace

PolySpace will be used to increase the productivity on existing activities, such as

- Data and control flow analysis

- Shared data detection
- Robustness unit tests.

The PolySpace approach

Depending on the activity replaced, both PolySpace Verifier and/or Desktop may be useful.

- For data and control flow analysis and shared data detection. PolySpace Verifier can be used on the whole application or on a sub-section of the application.

- For robustness unit tests (as opposed to functional unit tests). PolySpace Desktop might be used in the same way as the one applied to the Rigorous development process.

Costs and benefits

The replacement of these activities can lead to a significant cost reduction. For instance, the time spent on data and control flow analysis can drop from 3 months to 2 weeks.

Quality will also become much more consistent since a much greater part of the process will be automated. PolySpace tools are equally efficient on a Friday afternoon and on a Tuesday morning!

5.5. Code acceptance criterion

This is likely to be of interest for a quality manager in a company which is out-sourcing software development, and who wishes to impose acceptance criteria for the code.

The software development process

This section describes how to define transition criteria for intermediate or final deliveries.

The objective of using PolySpace

The objective is to control and evaluate the safety of an application. The means for doing so could vary from no red errors to exhaustive oranges review.

The PolySpace Approach

Either PolySpace Desktop or Verifier can be used at this stage, depending on the project size. The example list of acceptance criteria below shows increasingly stringent tests, any or all of which may be adopted.

- No compilation errors
- No compilation warning errors
- No red code sections
- No unjustified grey code section
- A selective/exhaustive orange review according to the development process
  - 20% orange code sections reviewed or a time base threshold (described in the previous sections)
  - 100% orange code sections reviewed
- ·20% concurrent access graph reviewed
- 100% concurrent access graph reviewed

6. Advanced

Related subjects :

6.1. PolySpace setup

These are the rules followed by PolySpace. It is strongly recommended that the preceding sections should be read and understood before applying the rules described below. Some rules are mandatory; others facilitate improved selectivity.

The following describes the default behaviour of PolySpace. If the code to be analysed does not conform to these assumptions, then some minor modifications to the code or to the PolySpace run time parameters will be required.

- The main procedure must terminate in order for entry-points (or tasks) to start.
- All tasks or entry-points start after the execution of the main has completed. They all start simultaneously, without any predefined assumptions regarding the sequence, priority and pre-emption.

If an entry-point is seen as dead code, it can be assumed that the main contains (a) red error(s) and therefore does not terminate. PolySpace assumes:

- no atomicity,
- no timing constraints.

Related subjects :

6.1.1. Can an application without "main" be analysed?

6.1.2. Modelling tasks, interruptions and events

6.1.3. Shared variables

6.1.4. Miscellaneous

6.1.1. Can an application without "main" be analysed?

Problem

My application doesn't have a main procedure. How can I analyse it using PolySpace?

Explanation

When your application is a function library (API) or a single module, you have to provide a main that calls those functions because of the execution model PolySpace uses. The reason why you have to do so is because it is much more powerful to take into account the calling sequence in the application to improve precision.

Solution

- Identify the API (application program interface) functions and extract their declaration;
- Create a main that will contain the declaration of a volatile variable for each type that is mentionned in the function prototypes.
- Create a loop with a volatile end condition. Inside this loop, create a switch bloc with a volatile condition and finally, for each API function, create a case branch that calls the function using, as parameters, the volatile variables created before.

Example

```
The API spec are:
function func1(x in integer) return integer;
procedure func2(x in out float, y in integer);

The main you'll have to create is the following :
procedure main is
  a,b,c,d: interger;
  e,f: float;
pragma volatile (a);
pragma volatile (e);
-- We need an integer and float variable as a function parameter

begin
  loop
f := e;
c:=a;
d:=a;
    if (a = 1) then b:= func1(c); end if;
    if (a = 1) then func2(e,d);    end if;
  end loop
end main;
```

6.1.2. Modelling tasks, interruptions and events

Related subjects :

6.1.2.1. Scheduling model

A problem can occur when some code is analysed and the results suggest that all background tasks are dead code. In the same way, the problem could the same (grey code) if several tasks (infinite loops) are defined and run concurrently in an RTOS.

In the PolySpace model, the main procedure is executed first before any other task is started. After it has finished, all task entry points are assumed to start concurrently, meaning they can interrupt each other at any time. This is an accurate upper approximation model for most concurrent RTOS.

Tasks and main loops need to simply declare as entry points. It only concerns task not defined using keyword of the Ada language.

Example:

```
procedure body back_ground_task is
begin
  loop -- infinite loop
 -- background task body
 -- operations
 -- function call
 my_original_package.my_procedure;
  end loop
end back_ground_task
```

Launching command:

```
polyspace-ada –entry-points package.other_task,package.back_ground_task
```

If the tasks are already infinite loops, simply declare them as mentioned above.

Limitation:

- A main procedure is always needed using –main option.
- **The tasks declared in -entry-points may not take parameters and may not have return values:** `procedure MyTask is …end MyTask;`

  If it is not the case, it is mandatory to encapsulate with a new procedure. In this case, the real task will be called inside.
- The main procedure cannot be called in a defined or declared task.

6.1.2.2. Modelling synchronous tasks

Problem

My application has the following behaviour:

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

My tasks never interrupt each other. My tasks are not infinite loops - they always return control to the calling context.

```
procedure tsk_10ms;
begin do_things_and_exit();
  -- it's important it returns control
end;
```

Explanation

If each task was declared to Verifier by using the option

```
polyspace-ada -tasks pack_name.tsk_10ms, pack_name.tsk_30ms, pack_name.
tsk_50ms
```

then the results **would** be valid – but there may be more warnings than necessary (that is, the results are less precise) because more scenarios than could actually happen at execution time are modelled.

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

Solution 1

Write a function that calls the cyclic tasks in the right order: this is an **exact sequencer**. This sequencer is then identified to Verifier as a single task.

This sequencer will be a single PolySpace task entry point. This solution:

- is more precise;
- but you need to know the exact sequence of events.

```
procedure body one_sequential_Ada_function is
begin
  loop
    tsk_10ms;
    tsk_10ms;
```

```
   tsk_10ms;
   tsk_30ms;
   tsk_10ms;
   tsk_10ms;
   tsk_50ms;
  end_loop
end one_sequential_function;


polyspace-ada -tasks pack_name.upper_approx_Ada_sequencer
```

Solution 2

Make an **upper approximation sequencer**, which takes into account every possible scheduling. This solution:

> is less precise;
> is quick to code, especially for complicated scheduling.

```
procedure body upper_approx_Ada_function is
  random : integer;
  pragma volatile (random);
begin
  loop
   if (random = 1) than tsk_10ms; end if;
   if (random = 1) than tsk_30ms; end if;
   if (random = 1) than tsk_50ms; end if;
  end_loop
end one_sequential_function;


polyspace-ada -tasks pack_name.upper_approx_Ada_function
```

> Note   If this is the only task, then it can be added at the end of the main.

6.1.2.3. Interruptions and asynchronous events/tasks

Problem

I have interrupt service routines which appear in grey (dead code) in the Viewer.

Explanation

The grey code indicates that this code is not executed and is not taken into account, so all interruptions and tasks are ignored by PolySpace Verifier.

The execution model is such that the main is executed initially. Only if the main terminates and returns control (i. e. if it is not an infinite loop) will the task entry points be started, with all potential starting sequences being modeled.

My interrupts it1 and it2 cannot preempt each other.

If these 3 following conditions are fulfilled:

- the it1 and it2 functions can never interrupt each other;
- each interrupt can be raised several times, at any time;
- they are returning functions, and not infinite loops.

Then you can group non preemptive interruptions in a single function and declare that function as a task entry point.

```
procedure it_1;
procedure it_2;

task body all_interruptions_and_events is
random: boolean;
pragma volatile (random);
begin
  loop
   if (random) then it_1;  end if;
   if (random) then it_2;  end if;
  end_loop
end all_interruptions_and_events;

polyspace-ada -tasks package.all_interruptions_and_events
```

My interruptions can preempt each other

If two interruption can be interrupted, then:

- encapsulate each of them in a loop;

- declare each loop as a task entry point.

```
package body original_file is
  procedure it_1 is     begin ... end;
  procedure it_2 is     begin ... end;
  procedure one_task is begin ... end;
end;


package body new_poly is
procedure polys_it_1  is    begin loop it_1; end loop; end;
procedure polys_it_2  is    begin loop it_2; end loop; end;
procedure polys_one_task is  begin loop one_task; end loop; end;

polyspace-ada -tasks new_poly. polys_it_1,new_poly. polys_it_2,new_poly.
polys_one_task
```

6.1.2.4. Are interruptions maskable or preemptive by default?

Problem

In my main task I use a critical section but I still have unprotected shared data. My application contains interrupts. Why is my variable analyzed as unprotected?

Explanation

PolySpace Verifier does not distinguish between interrupt service routines and tasks. If you specify an interrupt to be a "-task" entry point, it will have the same priority level as the other procedures declared as tasks ("-tasks" option). Therefore, as PolySpace Verifier makes an **upper approximation of all scheduling and all interleaving,** it **includes the possibility that the ISR might be interrupted by any other task**. There are more paths modelled than can happen during execution, but this has no adverse effect on of the results obtained;

Solution

Embed your interrupt in a specific procedure that uses the same critical section as the one you use in your main task. Then, each time this function is called, the task will enter a critical section which will be equivalent to a non-maskable interruption.

Original packages

```
package my_real_package is
  procedure my_main_task;
  procedure my_real_it;
  shared_X: INTEGER:= 0;
end my_real_package;

package body my_real_package is
  procedure my_main_task is
  begin
    mask_it;
    shared_x:= 12;
    unmask_it;
  end my_main_task;

  procedure my_real_it is
  begin
    shared_x:= 100;
  end my_real_it;
end my_real_package;
```

Extra packages

An extra package necessary to embed the task with body my_real_package;

```
package extra_additional_pack is
   procedure polyspace_real_it;
end extra_additional_package;

package body extra_additional_pack is
   procedure polyspace_real_it is
   begin
      mask_it;
      my_real_package.my_real_it;
      unmask_it;
   end;
end extra_additional_package;
```

Command line to launch PolySpace Verifier

```
polyspace-ada \
-tasks my_real_package.my_main_task,extra_additional_pack.polyspace_real_it
\
-main your_package.your_main
```

6.1.3. Shared variables

Abstract

All of my shared variables appear in orange in the variable dictionary.

Explanation

When you launch PolySpace Verifier without any option all tasks are examined at the same level, making no assumptions about priorities, sequence order, or timing. In this context, shared variables will always be considered as unprotected.

Solution

You can use the following mechanisms to protect your variables

- Critical section and mutual exclusion (explicit protection mechanisms);
- Access pattern (implicit protection);
- Rendezvous.

See below for more details.

Related subjects :

6.1.3.1. Critical sections

These are the most common protection mechanism in applications and they are simple to use in PolySpace Verifier:

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

package my_tasking is

```
  procedure proc1;
  procedure proc2;
  procedure my_main;
  X: INTEGER;
  Y: INTEGER;
end my_tasking;
```

package body my_tasking

```
 with pkutil; use pkutil;
package body my_tasking is
   procedure proc1 is
   begin
     begin_cs;
      X = 12; -- X is protected
      Y = 100;
     end_cs;
   end;
   procedure proc2 is
   begin
     begin_cs;
      X = 11; -- X is protected
     end_cs;
      Y = 101; -- Y is not protected
   end;
   procedure my_main is
   begin
     X := 0;
     Y := 0;
   end
end my_tasking;
```

package pkutil is

```
    procedure begin_cs;
    procedure end_cs;
end pkutil;
```

package body pkutil is

```
    procedure Begin_CS is
    begin
      null;
    end Begin_CS;
    procedure End_CS is
    begin
      null;
    end end_cs;
end pkutil;
```

Launching command

```
polyspace-ada \
    -automatic-stubbing \
    -main my_tasking.my_main \
    -tasks my_tasking.proc1,pktasking.proc2 \
    -critical-section-begin "pkutil.begin_cs:CS1" \
    -critical-section-end "pkutil.end_cs:CS1"
```

Also refer to atomicity.

6.1.3.2. Mutual exclusion

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

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

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

These entry-points cannot overlap:

- t1 and t3
- t2, t3 and t4

These entry-points can overlap:

- t1 and t2
- t1 and t4

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

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

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

```
t1 t3
t2 t3 t4
```

6.1.3.3. Access pattern

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

Consider the following example.

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

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

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

6.1.3.4. Rendez vous

All Ada rendezvous are taken into account without any input from the user. This is the only way to synchronize tasks. PolySpace Verifier does not handle atomicity so other task synchronisation mechanisms (including the use of critical sections) are not recognized by PolySpace Verifier.

package fisrt_task	other tasks
| ```
package first_task is
  task task_1 is
    entry INIT;
    entry ORDER (X: out Integer);
  end task_1;
end first_task;
package body first_task is
  task body task_1 is
  begin
    accept INIT;
    -- do things
    accept ORDER (X: out Integer)
    do
      -- do things
      -- call functions
      X:= 12;
    end; -- end accept
    -- return to main execution
  end task_1;
end first_task;
``` | ```
with first_task; use first_task;
package other_tasks is
  task task_2 is
  end task_2;
  procedure main;
end other_tasks;
package body other_tasks is
  task body task_2 is
    X: INTEGER;
  begin
    task_1.init;
    task_1.Order(X);
  end task_2;
  procedure main is
  begin;
    null;
  end;
end other_tasks;
``` |

The use of explicit tasks makes it unnecessary to use the –entry-points option in your launching script.

polyspace-ada -main other_task.main

6.1.3.5. Semaphores

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

6.1.4. Miscellaneous

Related subjects :

6.1.4.1. Mailboxes

6.1.4.2. Atomicity

6.1.4.3. Priorities

6.1.4.1. Mailboxes

Problem

My application has several tasks:

- some that post messages in a mailbox;
- others that read these messages asynchronously.

This communication mechanism is possible because the OS libraries provide send and receive procedures. I do not have the source files because these procedures are part of the OS libraries.

Explanation

By default, PolySpace Verifier will automatically stub these send/receive procedures. Such a stub will exhibit the following behaviour:

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

Solution

You can provide similar mechanisms with different levels of precision.

Let PolySpace Verifier stub automatically	- quick and easy to code - **imprecise** because there is no direct connection between a mailbox sender and receiver. It means that even if the sender is only submitting data within a small range, the full data range appropriate for the type(s) will be for the receiver data.
Provide a **real mailbox** mechanism	- can be very costly (time consuming) to implement; - can introduce errors in the stubs; - is too much effort compared with the solution below; - precise, but does not provide a much better precision than the upper approximation.
Provide an **upper approximation of the mailbox**	in which each new read to the mailbox reads **one** of the recently posted messages, but not necessarily the last one. - quick and easy to code; - gives precise results; - see detailed implementation below.

package mailboxes is

```
type BIG_ARRAY is
  array (1..100)of INTEGER;
type MESSAGE is
record
   length: INTEGER;
   content: BIG_ARRAY;
end MESSAGE;
MAILBOX : MESSAGE;
```

```
 procedure send
       (X: in MAILBOX);
 procedure receive
       (X: out MAILBOX);
end mailboxes;
```

package body mailboxes is

```
procedure send  (X: in MESSAGE) is
  random : boolean;
  pragma Volatile_ada.htm (random);
begin
  if (random) then
    MAILBOX:= X;
  end if;
   -- a potential write
   -- to the mailbox
end;
```

procedure receive

```
(X: out MESSAGE) is
begin
  X:= MAILBOX;
end;
```

task body task_1 is

```
  msg : MESSAGE;
begin
  for i in 1 .. 100 loop
    msg.content(i):= i;
  end loop;
  msg.length : = 100;
  send(msg);
end task_1;
task body task_2 is
  msg : MESSAGE;
begin
  receive(msg);
  if (msg.length = 100) ...
end;
```

Provided that each of these tasks is included in a package.

```
polyspace-ada -main a_package.a_procedure
```

6.1.4.2. Atomicity

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

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

Instructional decomposition

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

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

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

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

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

PolySpace Verifier considers write/read instructions atomic, so **task 2 can only read 0xFF55**, even if X is not protected (refer to Protecting shared variables).

Critical sections

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

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

More information is available in the critical sections section.

6.1.4.3. Priorities

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

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

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

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

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

6.2. PolySpace results analysis

Related subjects :

6.2.1. Integration bug tracking

By default, integration bug tracking can be achieved by applying the selective orange methodology to integrated code. Each error category will be more likely to reveal integration bugs, depending on the chosen coding rules for the project.

For instance, consider a function receives two unbounded integers. The presence of an overflow can only be checked at integration phase, since at unit phase the first mathematical operation will reveal an orange check.

Consider these two circumstances:

- Where integration bug tracking is performed in isolation, a selective orange review will highlight most integration bugs. In this case a PolySpace Verifier analysis has been performed integrating tasks.

- Where integration bug tracking is performed together with an exhaustive orange review at unit phase. In this case a PolySpace Desktop analysis has been performed on one or more packages.

In this second case, an exhaustive orange review will already have been performed package by package at a unit level. Therefore, at integration phase **only checks that have turned from green to another colour** are worth assessing.

For instance, if a function takes a structure as an input parameter, the standard hypothesis made at unit level is that the structure is well initialised. This will consequentially display a green NIV check at the first read access to a field. But this might not be true at integration time, where this check can turn orange if any context does not initialise these fields. These orange checks will reveal integration bugs.

6.2.2. How to find bugs in unprotected shared data

Based on the list of entry points in a multi-task application, PolySpace identifies a list of shared data and provides several pieces of information about each entry:

- The data type;

- A list of reading and writing accesses to the data through functions and entry points;
- The type of any implemented protection against concurrent access.

A shared data item is a global data item that is read from or written to by two or more tasks. It is unprotected from concurrent accesses when one task can access it whilst another task is in the process of doing so. All the possible situations are considered below.

- If there is a possible scenario which would lead to such conflict for a particular variable, then a bug exists and protection is required.

- If there are no such scenarios, then one of the following explanations may apply:

  - The compilation environment guarantees an atomic read/write access on variable of type less than 1, 2 … bytes, and therefore all conflicts concerning a particular variable type still guarantee the integrity of the variable's content. But beware when porting the code!
  - The variable is protected by a critical section or a mutual temporal exclusion. You may wish to include this information in the PolySpace Verifier launching parameters and re-analyse.

 It is also worth checking whether variables are modified which are supposed to be constant. Use the variables dictionary.

6.2.3. Dataflow analysis

Data flow analysis is often performed within certification processes - typically in the avionic, aerospace or transport markets.

This activity makes heavy use of two features of PolySpace results, which are available any time after the Control and Data Flow analysis phase.

- Call tree computation

- Dictionary containing read/write access to global variables. (This can also be used to build a database listing for each procedure, for its parameters, and for its variables.)

PolySpace can help you to build theses results by extracting information from both the call tree and the dictionary.

6.2.4. Cost and benefits of an exhaustive orange review

Related subjects :

6.2.4.1. Costs and Benefits

- **Costs**

Experience suggests that an average of 50 **orange (unproven) checks** by hour is typical.
If the checks are reviewed in the sequence suggested by the selective review approach, then the first 80% of these checks will take a disproportionately small amount of time.

- **Benefits**

The purpose of this activity is to assess the probability of missing an orange containing a bug when performing a "selective orange review". This needs to be balanced with the cost of a bug left in the code. Using the methodological assistant, unproven checks are selected and sorted by PolySpace.

6.2.4.2. Method

There are sometimes situations where files contain a particularly high number of orange checks compared with the rest of the application. This may well highlight design issues.

Consider the three possible reasons for an orange check:

1. Potential bug and Data set issues

2. Inconclusive analysis

3. Basic imprecision

The method described in the following chapter explains how to focus on finding potential bugs in the orange code. We will focus here on the first and second types. We are assuming that in the modules containing the most **orange checks**, those checks will prove inconclusive. If PolySpace is unable to draw a conclusion, the implication is often that the code itself is very complex – which in turn can identify sections of code of low robustness and quality.

Real bugs and data sets

If the data set analysed reveals real bugs, they should be corrected If it highlights potential input bugs (depending on the input data which might eventually be used) then the source code should be commented.

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.

Consider an example below. Suppose that

- a *signed* is an integer between -2^31 and 2^31-1
- an *unsigned* is an integer between 0 and 2^32-1
- The variable "Computed_Speed" is copied into a signed, and afterward into an unsigned, than signed, than added to another variable, and finally produces 20 **orange** overflows **(OVFL)**.

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

Basic imprecision

On some rare occasions, a module will contain a lot of similar occurrences of a "basic imprecision".

This is most likely to be caused by a function close to the edge of an application, or in the stub routines.

In this case, PolySpace can only assist by means of the call tree and dictionary. This code needs to be reviewed by an alternative activity – perhaps through additional unit tests or code review with the developer. These checks are usually local to functions, so their impact on the project as a whole is limited.

Examples of extra activities might be
- Checking an interpolation algorithm in a function
- Checking calibration data consisting of huge constant arrays, which are manipulated mathematically

6.2.5. PolySpace analysis duration

The duration of an analysis is impacted by:

- The size of the code
- The number of global variables
- The nesting depth of the variables (the more nested they are, the longer it takes)
- The depth of the call tree of the application
- The "intrinsic complexity" of the code, particularly with regards to arithmetic manipulation.

The fact that so many factors are involved makes it impossible to derive a precise formula to calculate analysis duration. Following sub section try to give some hints to reduce time of an analysis.

Related subjects :

6.2.5.1. An ideal application size

6.2.5.2. Why should there be an optimum size?

6.2.5.3. By selecting a subset of code

6.2.5.4. What are the benefits of these methods?

6.2.5.1. An ideal application size

There always is a compromise between the time and resources required to analyse an application, and the resulting selectivity. The larger the project size, the broader the approximations made by PolySpace. These approximations enable PolySpace to extend the range of project sizes it can manage, to perform the analysis further and to solve traditionally incomputable problems. However, they also mean that the benefits derived from analysing the whole of a large application have to be balanced against the loss of precision which results.

This is why it is recommended to begin with file by file analyses (when dealing with C language), package by package analyses (when dealing with Ada language) and class by class analyses (when dealing with C++ language). The <u>maximum</u> application size is between twenty (for C ++) and fifty thousand lines of code (for C and Ada). For such applications, approximations should not be too significant. Take care that some times analysis time should **not be reasonable**.

Experience suggests that subdividing an application prior to analysis will normally have a <u>beneficial impact on selectivity</u> – that is, more **red, green** and **grey checks**, fewer **orange unproven** and therefore more efficient bug detection.

A compromise between selectivity and size

6.2.5.2. Why should there be an optimum size?

PolySpace has been used to analyse numerous applications with greater than one hundred thousand lines of code. However, as project sizes become very large PolySpace Verifier

- Makes broader approximations, producing more oranges

- Can take much more time to analyse the application.

PolySpace is most effective when it is used **as early as possible** in the development process, i.e. **BEFORE** any other form of testing.

When a small module (file, piece of code, package, whatever) is analysed using PolySpace, the focus should be on the **red** and **grey** checks. **Orange** unproven checks at this stage are of a very useful interest, as most of them deal with robustness of the application. They will change to **red**, **grey** or **green** as the project progresses and more and more modules are integrated.

During the integration process, there might be a point where the code becomes so large (maybe 50000 lines of code or more) that the analysis of the whole project is not achievable within a reasonable amount of time. Then there are two options.

- Stop the use of PolySpace at this stage (a lot of the benefits have been achieved already), or

- Analyse subsets of the code.

6.2.5.3. By selecting a subset of code

If a project is subdivided into logical sections by considering data flow, the total analysis time will be considerably shorter than for the project considered in one pass. (See also: address clause , volatile , automatic stubbing)

In such an application, there are two distinct concepts to consider:

- function entry-points. Function entry-points refer to the PolySpace execution model since they are started concurrently, without any assumption regarding sequence or priority. They represent the beginning of your call tree;
- data entry-points. Regard lines in the code where data is acquired as "data entry points".

Consider the examples below.

Example 1

```
Procedure complete_treatment_based_on_x(input : integer) is
begin
   thousand of line of computation...
end
```

Example 2

```
procedure main is
begin
  x:= read_sensor();
  y:= complete_treatment_based_on_x(x);
end
```

Example 3

```
REGISTER_1: integer;
for REGISTER_1 use at 16#1234abcd#;
procedure main is
begin
  x:= REGISTER_1;
  y:= complete_treatment_based_on_x(x);
end
```

In each case, the "x" variable is a data entry point and "y" is the consequence of such an entry point. "y" may be formatted data, due to a very complex manipulation of x.

Since x is volatile, a probable consequence will be that y will contain all possible formatted data. An approximation could be to completely remove the procedure complete_treatment_based_on_x and let automatic

stubbing work: it will then assign a full range data to y directly.

```
-- removed body of complete_treatment_based_on_x
procedure main is
begin
  x:= ... -- what ever;
  y:= complete_treatment_based_on_x(x); -- now stubbed!
end
```

Some consequences

(-) A slight loss of precision on y. Verifier will now consider all possible values for y, including the formatted ones that were present at the first analysis.

(+) A huge investigation of the code is not necessary to isolate a meaningful subset. Any application can be split logically in this way.

(+) No functional modules are lost.

(+) The results will still be correct because there is no need to remove any thread affecting (change) shared data.

(+) The complexity of the code is considerably reduced.

(+) A high precision level (say O2) can be maintained.

Typical examples of removable components, according to the logic of the data:

- Error management modules. These modules often contain a big array of structures that are accessed through an API, but return only a Boolean value. By removing the API code and retaining the prototype, the automatically generated stub will be assumed to return a value in the range [-2^31, 2^31-1], which includes 1 and 0. The procedure will be considered to return all possible answers, just like reality;
- Buffer management for mailboxes coming from missing code. Suppose an application reads a huge buffer of 1024 char, and then uses it to populate 3 small arrays of data, using a very complicated algorithm before passing it to the main module. If the buffer is excluded from the analysis and the arrays are initialised with random values instead, then the analysis of the remaining code will just be the same.

Related subjects :

6.2.5.3.1. Subdivide according to data-flow
6.2.5.3.2. Subdivide according to real-time characteristics
6.2.5.3.3. Subdivide according to files

6.2.5.3.1. Subdivide according to data-flow

Consider the following example.

In this application, variables 1, 2 and 3 can vary between the following ranges

Var1	Between 0 and 10
Var2	Between 1 and 100
Var3	Between -10 and 10

Specification of Module A:

Module A consists of an algorithm which interpolates between var1 and var2. That algorithm uses var3 as an exponential factor, so when var1 is equal to 0, the result in var4 is also equal to 0.

As a result, var4, var5 and var6 are produced with the following specifications:

Ranges	var4	Between -60 and 110
	var5	Between 0 and 12
	var6	Between 0 and 100
Properties	And a set of properties between variables	1. If var2 is equal to 0, than var4>var5>5. 2. If var3 is greater than 4, than var4<var5<12 3. …

Subdivision in accordance with data flow allows modules A and B to be analysed separately.

- A will use variables 1, 2 and 3 initialised respectively to [0;10], [1;100] and [•10;10]

- B will use variables 4, 5 and 6 initialised respectively to [-60;110], [0;12] and [•10;10]

The consequences

(-) A slight loss of precision on the B module analysis, because now all combinations for variables 4, 5 and 6 are considered:

- It includes all of the possible combinations.

- It also includes those that would have been restricted by the A module analysis.
    For instance. If the B module included the test
        "If var2 is equal to 0, than var4>var5>5"
    then the dead code on any subsequent "*else*" clause would not be detected.

(+) An in depth investigation of the code is not necessary to isolate a meaningful subset. It means that a logical split is possible for any application, in accordance with the logic of the data

(+) The results remain valid (because there no need to remove (say) a thread that will change shared data)

(+) The complexity of the code is reduced by a significant factor

(+) The maximum precision level can be retained.

Typical examples of removable components:

- Error management modules. A function has_an_error_already_occurred might return TRUE or FALSE. Such a module may contain a big array of structures which are accessed through an API. The removal of the API code with the retention of the prototype will result in the Verifier analysis producing a stub which returns $[-2^{31}, 2^{31}-1]$. This clearly includes 1 and 0 (yes and no). The procedure has_an_error_already_occurred will therefore return all possible answers, just like the code would at execution time.

- Buffer management for mailboxes coming from missing code. Suppose a large buffer of 1024 char is read, and the data is then collated into 3 small arrays of data using a very complicated algorithm. This data is then given to a main module for treatment. For the Verifier analysis, the buffer can be removed and the 3 arrays initialised with random values.
- Display modules.

6.2.5.3.2. Subdivide according to real-time characteristics

Another way of splitting an application is to isolate files which contain only a subset of tasks, and to analyse each subset separately.

If an analysis is initiated using only a few tasks, PolySpace Verifier will lose information regarding the interaction between variables.

Suppose an application involves tasks T1 and T2, and variable x.

If T1 modifies x and T2 is scheduled to read it at a particular moment, subsequent operations in T2 will be impacted by the values of x.

As an example, consider that T1 can write either 10 or 12 into x and that T2 can both write 15 into x and read the value of x. There are two ways to achieve a sound stand-alone analysis of T2.

- x could be declared as volatile in order to take into account all possible executions. Otherwise x will take only its initial value or x variable will remain constant, and T2's analysis will be a subset of possible execution paths. You might have precise results, but it will only include one *scenario* among all possible states for the variable x.

- x could be initialised to the whole possible range [10;15], and then the T2'entry-point called. This is accurate if x is calibration data.

6.2.5.3.3. Subdivide according to files

Simply extract a subset of files and perform an analysis either

- using entry-points, or

- by creating a "*main*" that calls randomly all functions that are not called by any other within this subset of code.

This method may look too simple to be efficient but it can produce good results when the aim is to find **red** errors and bugs in **grey** code.

6.2.5.4. What are the benefits of these methods?

It may be desirable to split the code

- To reduce the analysis time for a particular precision mode

- To reduce the number of oranges (see next two sections for details)

The problems subdivision may bring are that

- Orange checks can result from a lack of information regarding the relationship between modules, tasks or variables

- Orange checks can result from using too wide a range of values for stubbed functions

Related subjects :

6.2.5.4.1. When the application is incomplete

When the code consists of a small subset of a larger project, a lot of procedures will be automatically stubbed. This is done according to the specification or prototype of the missing functions, and therefore PolySpace assumes that all possible values for the parameter type can be returned.

Consider two 32 bit integers "a" and "b", which are initialised with their full range due to missing functions. Here, a*b would cause an overflow, because "a" and "b" can be equal to 2^{31}. The number of incidences of these "data set issue" **orange check** can be reduced by precise stubbing.

Now consider a procedure f which modifies its input parameters "a" and "b", both of which are passed by reference. Suppose that "a" might be modified to any value between 0 and 10, and "b" to any value between -10 and 10. In an automatically stubbed function, the combination a=10 and b=10 is possible even though it might not be possible with the real function. This can introduce orange checks in a code snippet such as 1/(a*b - 100), where the division would be **orange**.

- So - even where precise stubbing is used, analysing a small piece of application might introduce extra orange checks. However, the net effect from reducing the complexity will be to reduce the total number of orange checks.

- When using the default stubbing, the increase in the number of orange checks as the result of this phenomenon tends to be more pronounced.

6.2.5.4.2. 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,

- -O2: less orange warnings.

- -O3: less orange warnings and bigger computation time.

7. Options description

Related subjects :

7.1. GENERAL

This section collates all options relating to the identification of the analysis, including the destination directory for the results and sources.

Related subjects :
  7.1.1. -prog program-name
  7.1.2. -date date
  7.1.3. -author author-name
  7.1.4. -verif-version verif-version
  7.1.5. -voa
  7.1.6. -keep-all-files
  7.1.7. -continue-with-red-error
  7.1.8. -continue-with-existing-host
  7.1.9. -allow-unsupported-linux
  7.1.10. -sources "files" or -sources-list-file file_name
  7.1.11. -extensions-for-spec-files and -ada-include-dir
  7.1.12. -results-dir directory
  7.1.13. -pre-analysis-command file or "command"
  7.1.14. -post-analysis-command file or "command"

7.1.1. -prog program-name

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-ada -prog myApp ...
```

7.1.2. -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-ada -date "02/01/2002"...
```

7.1.3. -author author-name

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-ada -author "John Tester"
```

7.1.4. -verif-version verif-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-ada -verif-version 1.3 ...
```

7.1.5. -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-ada -voa ...
```

7.1.6. -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.

7.1.7. -continue-with-red-error

Note: This option may yield invalid results when used improperly.

Ordinarily, red errors (other than NTC) prevent PolySpace from continuing to the next integration pass.

This option allows PolySpace to continue even if one of these red errors is encountered. In most cases, this will mean that the dynamic behavior of the code beyond the point where red errors are identified will be undefined, unless the red code is actually inaccessible.

When using this option it is not rare to when opening some results, a strange red error is encountered. it could be interesting to open results at level 1 (pass1) to verify that some other red errors have not been highlighted.

Default:

Verifier stops upon finding red errors.

Example shell script entry :

polyspace-ada -continue-with-red-error ...

7.1.8. -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-ada -continue-with-existing-host ...
```

7.1.9. -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-ada –allow-unsupported-linux ...
```

7.1.10. -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*

It gives the list of source files to be analyzed, double-quoted and separated by commas. The specified files must have valid extensions:
(A|a)d(a|b|s) for Ada
Defaults:
        sources/*.(A|a)d(a|b|s) for Ada
Examples under linux or solaris:
        polyspace-ada -sources "my_directory/mod*.ad[sb]" ...
Examples under windows:
        polyspace-ada -sources "spc/mod1.ads,bod/mod1.adb" ...

Using -sources-list-file in batch mode, the syntax of the file is the following:
- one file by line.
- file names are given with absolute or relative path. See -sources-list-file option.

7.1.11. -extensions-for-spec-files and -ada-include-dir

The -extensions-for-specs-files option specifies the file extension for files "*F*" which will be analyzed to get the type/variables names but which are not part of the [-sources](#) list.

It's like having a dictionary with only the list of words and their type (*verb, noun, adj*) without the definition. These files will allow the product to know the name and the type, but not the values (*dictionary definitions*).

The -ada-include-dir specifies the directory where the *F* files are located. However, the option can be used several times and more than one directory can be specified

Note:

Both options must be used together.

Benefits:

- faster compilation on these packages in order to focus on the [-sources](#) packages specifications and bodies
- full range for all constants defined in these packages: let's consider 1 package bodie B and 2 specifications S1 and S2

Usage examples using the graphical interface:

configuration 1:

- -sources contains B.ada and S1.ada
- -extensions-for-specs-files contains the *.ada filter
- -ada-include-dir contains the TEST folder and the TEST folder contains S2.ada

configuration 2:

- -sources contains B.ada, S1.ada, S2.ada
- If a constant S2.C is used
  - o in configuration 1: its value will be its full range
  - o in configuration 2: its value will be the real constant value

Usage examples in shell entry-script mode:

```
polyspace-desktop-ada -sources "B.ada,S1.ada" -extensions-for-specs-files
"*.ada" -ada-include-dir ./include_specs
polyspace-desktop-ada -sources sources/example.ad* -extensions-for-spec-
files "*.ad?" -ada-include-dir "sources"
```

7.1.12. -results-dir 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-ada -results-dir RESULTS ...
export RESULTS=results_`date +%d%B_%HH%M_%A`
polyspace-ada -results-dir `pwd`/$RESULTS ...
```

7.1.13. -pre-analysis-command file or "command"

When this option is used, the specified script file or command is run before the analysis phase on each source file.
The command should be designed to process the standard output from source code and produce its results in accordance with that standard output.
Default:
No command.
Example Shell Script Entry – file name:
To replace the keyword "`Volatile`" by "`Import`", you can type the following command:
```
polyspace-ada -pre-analysis-command `pwd`/replace_keywords
```
where replace_keywords is the following script :
```
#!/bin/sh
sed "s/Volatile/Import/g"
```
Example Shell Command Entry:
This example performs the same function as that illustrated above, but specifies the command line directly:
```
polyspace-ada -pre-analysis-command "sed s/Volatile/Import/g"
```

7.1.14. -post-analysis-command file 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-ada -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-ada -post-analysis-command "mailx –s \"PolySpace Analysis ended\"
\"name@domain.com\""
```

7.2. TARGET/COMPILER

This section allows details of the target processor and operating system to be specified.

Related subjects :
  7.2.1. -target target-name
  7.2.2. -OS-target OperatingSystemTarget

7.2.1. -target target-name

Specify the target processor type. This option helps PolySpace to know the size of fundamental data types and whether your machine is big or little endian.
Possible values are: `sparc, m68k, 1750a, powerpc64bit, powerpc32bit and i386.`
Default:
sparc
Example:
polyspace-ada -target m68k ...

7.2.2. -OS-target OperatingSystemTarget

It specifies the Operating system target for Standard Libraries compatibility for PolySpace stubs. This option allows PolySpace to support implementation specific declarations contained in the Ada standard libraries.
Possible values are `gnat`, `greenhills` and `no-predefined-OS`.
Default:
`no-predefined-OS`. Note that this option allows `gnat` includes.
Note:
Only the 'gnat' include files are provided with PolySpace (see the "`adainclude`" folder in the installation directory). Projects developed for use with other operating systems may be analyzed by using the corresponding include files for that OS. For instance, in order to analyse a 'greenhills' project it is necessary to use option [–ada-include-dir](#) <path_to_the_greenhills_include_folder>. Note that this set of includes is not delivered with the product.

Example shell script entry:
```
    polyspace-ada -OS-target gnat
    polyspace-ada -OS-target greenhills -ada-include-dir /complete_path_to/
greenhills_includes ...
```

7.3. COMPLIANCE WITH STANDARDS

Related subjects :

7.3.1. -storage-unit number

Allows to choose the value of the constant SYSTEM.Storage_Unit. This constant is defined in the SYSTEM package. If this option is set, *a strictly positive number*, the value found in the SYSTEM package will be ignored

Default

 The default value of the constant is 8 except for the target 1750a

 which is 16.

Example

```
-- Definition of record type
type REC is record
   A : integer;
   B : boolean;
end REC;
-- Representation clause of this record
for REC use record
   A at 0 range 0 .. 31;
   B at 1 range 0 .. 31;
end record;
```

With a target defining 8 as storage unit value, the error "A overlaps B" appears because the value of SYSTEM.Storage_Unit is 8. In the example, this value need to be 32. The use of -storage-unit 32, removes the error message and allows to compute the size of REC.

7.3.2. -base-type-directly-visible

Standard Ada is ambiguous on visibility of comparison and equality operators (=,/=,<=,=>, >, <). This option allows removing some ambiguities.

In case of compilation error concerning visibility of comparison and equality operators, such as:

- "ambiguous expression (cannot resolve "<=")
- "operator for type "X" defined at ./exemple.ada:2 is not directly visible use clause would make operation legal

Setting the option can make the code legal"

Default:

- It is the type of the operand that matters to determine whether the operator is visible
- For overloaded functions, potentially use visible means use visible for sure

Ada example:

```
Package A is
 type T1 is new Integer range 0 .. 100; -- line 1
end A;
 -- Other file:example1.adb
with A; use A;
Package B is
 subtype T2 is T1 range 2..80;
end B;

Package OTHER_IABC_ADA_4 is
 procedure Main;
end OTHER_IABC_ADA_4;

with B; use B;
Package body OTHER_IABC_ADA_4 is
 X, Y : T2;
procedure Main is
 begin
 null;
 pragma Assert (TRUE);
end Main;
 begin
 X := 12;
 Y := 10;
 if X > Y then -- line 21
 pragma Assert (True);
 null;
```

```
   end if;
 end OTHER_IABC_ADA_4;
```
Without the option, an error message appears:

- Verifier found an error in ./example1.adb:21:07: operator for type "T1" defined at ./example1.adb:1 is not directly visible
- Verifier found an error in ./example1.adb:21:07: use clause would make operation legal

With the option, there is no error message.

Shell script command:

```
polyspace-ada -base-type-directly-visible ...
```

7.3.3. Permissiveness/Strictness

Analysis mode can be chosen between two options: -permissive and -strict.
When any of this two options are selected the "customize" allows to select following options independently: -no-automatic-stubbing, -continue-with-in-out-niv and –continue-with-all-niv.

Related subjects :

7.3.3.1. -permissive

Permissive mode of PolySpace. Equivalent to -continue-with-in-out-niv and –continue-with-red-error.

7.3.3.2. -continue-with-in-out-niv

Ada Standard imposes that in/out parameters of a procedure must be initialized. With this option, such a variable is still detected as a red NIV but the following code won't be unreachable and this red error won't have any impact on the analysis. This option may be used with -continue-with-red-error.

Default:
> If a variable has not been initialized AND is passed to a procedure as an in/out parameter, PolySpace indicates a red NIV and the rest of code is grey (dead code).

Example:
```
procedure test(x : in out Integer) is
    begin
       x := 10;
    end
procedure main is
     T : integer;
    begin
     test(T);      -- red NIV on T with or without the option
     T := T + 1; -- grey code on this line by default, green with -continue-
with-in-out-niv
    end Main;
```

Note:
If some in/out NIV are detected (in level 1 for instance), the analysis will stop at the end of the Software Safety level 1, as for any other red error detection. In order for the analysis to continue (in level 2, 3, 4 in this case), the user must set the option -continue-with-red-error.

7.3.3.3. -strict

Strict mode of PolySpace.
In Ada, equivalent to -no-automatic-stubbing

7.3.3.4. -no-automatic-stubbing

Missing body of procedures or functions (functions and procedures that are declared but not defined) cause PolySpace to stop.

Defaults:

All procedures and functions are stubbed automatically according to their specification. The rules are the following:

The generated stub is the most general possible body derived from its prototype.

- Implicit and explicit tasks cannot be stubbed.
- The main procedure cannot be stubbed.
- The generated stubs can not have any side effects on global variables. If a function with global side effects must be stubbed, it must be done by hand.

Benefits:

The user may want to use this option for several reasons

- he wants to make sure the entire code is provided: this can be the case when analyzing a large piece of code. When the analysis stops, it means the code is not complete: it will avoid the user surprises to see a code with stubs instead of the original code he was expecting
- he wants to write stubs himself to increase the selectivity and speed of the analysis.

Example:

```
polyspace-ada -no-automatic-stubbing -main ...
```

7.3.3.5. -continue-with-all-niv

Detect all non initialized variables (NIV). Without this option, Verification stops after the first red NIV. Warning: Precision loss when using this option. It should only be set for the 1st run of a project. This option may be used with -continue-with-red-error.

Default:

If a variable has not been initialized, PolySpace indicates a red NIV and the rest of the procedure is grey (dead code). All remaining checks in the procedure are grey.

Example:

This example contains 3 red NIV: by default, only the first one can be detected. With the -continue-with-all-niv option, all 3 will be detected at once, at the end of Level 1 analysis.

```
procedure Main is
     I,T,No: Integer;
   begin
     if (No = 0) -- red NIV, with or without the option
     then
        I := 1/I; -- grey code by default, red NIV with the option
     end if;
     if (T = 0) -- grey code by default, red NIV with the option
     then
        I := 12312409 /120;
     end if;
   end Main;
```

Note: If some NIV are detected (in level 1 for instance), the analysis will stop at the end of the Software Safety level 1, as for any other red error detection. In order for the analysis to continue (in level 2, 3, 4 in this case), the user must set the option -continue-with-red-error.

7.4. POLYSPACE INNER SETTINGS

Related subjects :

PolySpace
TECHNOLOGIES

7.4.1. -main main_subprogram_name

The option specifies the qualified name of the main subprogram. This procedure will be analyzed after package elaboration, and before tasks in case of a multitask application or in case of the -entry-points usage.

Note:

this option is exclusive with -main-generator.

Example:

```
polyspace-ada -main mainpackage.init ...
```

7.4.2. -main-generator

The -main-generator is **exclusive** with -main option.

The -main-generator option will create automatically a procedure which calls every non called procedure within the code, avoiding for instance to create manually a main.

Notes for PolySpace Destkop and PolySpace Verifier:

For PolySpace Desktop: the -main-generator option is set by default and the -main option can replace it if activated

For PolySpace Verifier: the -main option is set by default and the -main-generator option can replace it if activated

Example shell script entry:

```
polyspace-ada -main-generator ...
polyspace-desktop-ada ... (implicit -main-generator active)
polyspace-desktop-ada -main myPack.main ... (implicit -main-generator
canceled by the usage of -main)
```

7.4.3. Stubbing

Related subjects :

7.4.3.1. -import-are-not-volatile

If a variable has a pragma import(*C/ASM/other*, my_variable), it's then considered as volatile by PolySpace. With this option, they are considered as regular variables.

Default

 Imported variable are volatile

Example

```
polyspace-ada -import-are-not-volatile -main ...
```

7.4.3.2. -export-are-not-volatile

If a variable has a pragma export(*C*/*ASM*/*other*, my_variable), it's then considered as volatile by PolySpace. With this option, they are considered as regular variables.
Default
 Exported variable are volatile
Example
```
polyspace-ada -export-are-not-volatile
```

7.4.3.3. -init-stubbing-vars-random

Force initialization of uninitialized global variables to a random value.
Default:
Unitialized global variables give warnings or errors, depending on the context.
Example:
```
polyspace-ada -init-stubbing-vars-random -main ...
```

7.4.3.4. -init-stubbing-vars-zero-or-random

Initialize uninitialized globals variables:
- with zero if the type contains zero,
- with random otherwise

Default:
Unitialized global variables give warnings or errors, depending on the context.

Example:
```
polyspace-ada -init-stubbing-vars-zero-or-random -main ...
```

7.4.4. Assumptions

Related subjects :

7.4.4.1. -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.

Default:

IEEE 754 rounding under 32 bits and 64 bits.

Example Shell Script Entry :

```
polyspace-ada -ignore-float-rounding ...
```

7.4.4.2. -known-NTC proc1[,proc2[,...]]

After a few analyses, you may discover that a few functions "never terminate". Some functions such as tasks and threads contain infinite loops by design, while functions that exit the program such as *kill_task* , *exit* or *Terminate_Thread* are often stubbed by means of an infinite loop. If these functions are used very often or if the results are for presentation to a third party, it may be desirable to filter all NTC of that kind in the Viewer.
This option is provided to allow that filtering to be applied. All NTC specified at launch will appear in the viewer in the known-NTC category, and filtering will be possible.

Default :
 All checks for deliberate Non Terminating Calls appear as red errors, listed in the same category as any problem NTC checks.

Example Shell Script Entry :
```
 polyspace-ada -known-NTC "kill_task,exit"
 polyspace-ada -known-NTC "Exit,Terminate_Thread"
```

7.4.5. Others

Related subjects :

7.4.5.1. -orange-analyzer

The option requests the computation for each orange check of necessary failure conditions expressed as constraints on variables values that hold when the orange check is red at runtime.
Message format is the following:

    Orange check description

    If <(the orange check is red)> then <(condition to have this check in a red color)>

This option is not compatible with options -continue-with-in-out-niv and -continue-with-all-niv.

Example:

On the following example you could have this kind of message associated to an orange OVFL on the + operator:

```
Var := lamba + z;
Warning: scalar overflow may overflow on [conversion from int32 range … To
int32 range 1260 … 0]
if <OVFL> is red then ( 1240 <= z <= 12)
```

Default:

Disabled by default

Script usage Example:

polyspace-ada -orange-analyzer ...

7.4.5.2. -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-ada -extra-flags -param1 -extra-flags -param2 \
  -extra-flags 10 ...
```

7.4.5.3. -ada95-extra-flags extra-flag (Ada95 only)

This option specifies an expert option to be added to the analysis. Each word of the option (even the parameters) must be preceded by *–ada95-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-ada –ada95-extra-flags -param1...

7.5. PRECISION

Related subjects :

7.5.1. -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 analyses.

Default :

 scratch

Example Shell Script Entry :

```
 polyspace-ada -from c-to-il ...
```

7.5.2. -to verification-phase

Specifies the verification phase after which the Verifier will stop.

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:

- normalize
- compile
- pass0 or CDFA or "Control and Data Flow Analysis"
- pass1 or "Software Safety Analysis level 1"
- pass2 or "Software Safety Analysis level 2"
- pass3 or "Software Safety Analysis level 3"
- pass4 or "Software Safety Analysis level 4"
- other

Note:

If you use *-to other* then PolySpace will continue until you stop it manually (via `kill-rte-kernel`) or stops until it has reached *pass20*.

Default:

pass4

Example Shell Script Entry:

```
polyspace-ada -to "Software Safety Analysis level 3"...
polyspace-ada -to pass0 ...
```

7.5.3. -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. In Ada, the option set –path-sensitivity-delta to the value Y+5, where Y is the value already set by the option.

Default:
    -O2

Example Shell Script Entry:
    polyspace-ada -O1 -to pass4 ...

7.5.4. -modules-precision mod1:O(0-3)[,mod2:O(0-3)[,...]]

This option is used to specify the list of .c files to be analyzed with a different precision from that specified generally -O(0..3) for this analysis.

In batch mode, each specified module is followed by a colon and the desired precision level for it. Any number of modules can be specified in this way, to form a comma-separated list with no spaces.

Default:

All modules are treated with the same precision.

Example Shell Script Entry:

```
polyspace-ada -O1 \
  -modules-precision myMath:O2,myText:O1, ...
```

7.5.5. -array-expansion-size number

This option forces PolySpace to analyze each cell of global variable arrays having length less than or equal to *number* as a separate variable.
Warning:
Increasing the number of global variables to be analyzed will have an impact on the analysis time. This option has an impact only on the Global Data Dictionary results.
Default:
        The default value is 3.
Example:
        polyspace-ada -O1 -array-expansion-size 8 -main ...

7.5.6. -path-sensitivity-delta number

This option is used to improve interprocedural analysis precision within a particular pass (see -to *pass1*, *pass2*, *pass3* or *pass4*). The propagation of information within procedures is done earlier than usual when this option is specified. That results in improved selectivity and a longer analysis time.

Consider two analyses, one with this option set to 1 (with), and one without this option (without)
  - a level 1 analysis in (with) (pass1) will provide results equivalent to level 1 or 2 in the (without) analysis
  - a level 1 analysis in (with) can last x times more than a cumulated level 1 +2 analysis from (without). "x" might be exponential.
  - the same applies to level 2 in (with) equivalent to level 3 or 4 in (without), with potentially exponential analysis time for (a)

Gains using the option
 (+) highest selectivity obtained in level 2. no need to wait until level 4
 (-) This parameter increases exponentially the analysis time and might be even bigger than a cumulated analysis in level 1+2+3+4
 (-) This option can only be used with less than 1000 lines of code

Default:
  0

Example Shell Script Entry:
 polyspace-ada -path-sensitivity-delta 1 ...

7.5.7. -variables-to-expand var1[,var2[,...]]

Specifies aggregate variables (record, ...) that will be split into independent variables for the purpose of analysis.
This option has an impact on the Global Data Dictionary results.
Use with -variable-expansion-depth.
Default:
 Depending on complexity issues, fields in records may not be individually analyzed.
Example:

```
 polyspace-ada -variables-to-expand pkg.rec1,pkg2.recF \
    -variable-expansion-depth 4 -main ...
```

7.5.8. -variable-expansion-depth number

Indicate the maximum depth for expansion of variables specified by the -variables-to-expand option. So, it is mandatory first to specify which variables need to be expanded first.

Warning:

Increasing the number of global variables to be analyzed will have an impact on the analysis time. This option has an impact only on the Global Data Dictionary results.

Default:

There is no default.

Example:

Consider the following code :

```
Package foo is
 Type Internal is
 Record
  FieldI : Integer;
  FieldII : Integer;
 End Record ;
 Type External is
 Record
  Data : Internal ;
  FieldE : Integer;
 End Record ;
 myVar : External ;
End foo;
```

Effects of different expansion depths if you use -variables-to-expand foo.myVar :

-variable-expansion-depth 1 : the concurrent access analysis is made on foo. myVar.FieldE and foo.myVar.Data which means that if each access on Data is protected by critical section but FieldE is not protected, then Data will be flagged as protected (green entry in the Global Data Dictionary) and FieldE as not protected (orange entry)

-variable-expansion-depth 2 : the analysis is made on foo.myVar.FieldE, foo. myVar.Data.FieldI and foo.myVar.Data.FieldII : each variable will be flagged independently.

foo.myVar is flagged as shared if any of its field are shared; it is flagged as non-protected if any of its fields are not protected.

Example (the previous one, implemented):

```
polyspace-ada -variables-to-expand pakcage_foo.myVar \
  -variable-expansion-depth 1 -main ...
```

7.6. MULTITASKING (PolySpace Server only)

Concurrency options are not compatible with -main-generator option.

Related subjects :
    7.6.1. -entry-points str1[,str2[,...]]
    7.6.2. -critical-section-[begin or end] "proc1:cs1[,proc2:cs2]"
    7.6.3. -temporal-exclusions-file file_name

7.6.1. -entry-points str1[,str2[,...]]

This option is used to specify the tasks/entry points to be analysed by PolySpace, 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.

Moreover, when tasks are declared with Ada task keyword, PolySpace takes them into account automatically.

Example Shell Script Entry:

```
polyspace-ada -entry-points proc1,proc2,proc3 ...
```

7.6.2. -critical-section-[begin or end] "proc1:cs1[,proc2:cs2]"

```
-critical-section-begin "proc1:cs1[,proc2:cs2]"
```
and
```
-critical-section-end "proc3:cs1[,proc4:cs2]"
```

These options specify the procedures beginning and ending critical sections, respectively. Each uses a list enclosed within double speech marks, with list entries separated by commas, and no spaces.
Entries in the lists take the form of the procedure name followed by the name of the critical section, with a colon separating them.
These critical sections can be used to model protection of shared resources, or to model interruption enabling and disabling.

Default:

no critical sections.

Example Shell Script Entry:
```
polyspace-ada -critical-section-begin "start_my_semaphore:cs" \
-critical-section-end "end_my_semaphore:cs"
```

7.6.3. -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 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-ada -temporal-exclusions-file sources/exclusions \
    -entry-points task1_group1,task2_group1,task1_group2,\
    task2_group2,task3_group2 ...
```

7.7. BATCH OPTIONS

Related subjects :

7.7.1. -server server_name_or_ip[:port_number]

Using `polyspace-remote[-desktop]-[ada][-server [name or IP address][:<port number>]]` allows to send analysis to a specific or referenced [PolySpace Queue manager server](#).
Note 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-ada –server 192.168.1.124:12400 …
polyspace-remote-ada …
polyspace-remote-ada –server Bergeron …
```

7.7.2. -h[elp]

Display in the shell window a simple help in a textual format giving information on all options.
Example Shell Script Entry:
```
polyspace-ada –h
```

7.7.3. -v | -version

Display the PolySpace version number.
Example Shell Script Entry:
```
polyspace-ada –v
```
It will show a result similar to:
```
PolySpace r2007a+
Copyright (c) 1999-2007 PolySpace Technologies
```

7.7.4. -sources-list-file file_name

This option is only available in batch mode. The syntax of *file_name* is the following:
-     One file per line.
-     Each file name includes its absolute or relative path.

Example Shell Script Entry for -sources-list-file:

```
polyspace-ada -sources-list-file "C:\Analysis\files.txt"
polyspace-ada -sources-list-file "files.txt"
```

7.8. COMPLETE EXAMPLES

Simple Ada example
```
polyspace-ada \
  -main a_project.root_procedure \
  -prog myProject \
  -O1 \
  -sources directory/*.ad[bs] \
  -modules-precision sri:O2,types:O0
```
HDCA_Server
An Ada example. Note that we try to minimize analysis time in going to pass2 and O0. Note also the list of files (no spaces in that file list!).
```
polyspace-ada \
  -prog HDCA_Server \
  -main hdca_main.HDCA_Server \
  -O0 \
  -from scratch -to pass2 \
  -keep-all-files \
  -no-automatic-stubbing \
  -continue-with-red-error \
  -results-dir RESULTS \
  -sources \
$working_version/hdca/clock_and_date.ada,\
$working_version/hdca/cpu_usage.ada,\
$working_version/hdca/exception_log.ada,\
$working_version/hdca/hdca_main.ada,\
$working_version/screen/monitor.ada,\
$working_version/common/utilities/letter_box.ada,\
$working_version/common/utilities/library_functions.ada,\
$working_version/common/utilities/catalog_tools.ada,\
$working_version/common/utilities/configuration.ada,\
$working_version/common/utilities/converting.ads\
$working_version/common/utilities/converting.adb
```
airplane2
An Ada example with tasks.
```
polyspace-ada \
  -target m68k \
```

```
    -entry-points Wings.wingSuperVisor,Tail.tailSuperVisor,\
Rudder.rudderSuperVisor \
    -to pass2 \
    -from scratch \
    -prog airplane2 \
    -O0 \
    -results-dir `pwd`/RESULTS_14_08 \
    -main main.pst_main
```

high_speed_train

An Ada example.

```
 polyspace-ada \
    -target sparc \
    -from scratch \
    -array-expansion-size 1 \
    -sources "sources/*.[aA]*[a-zA-Z]" \
    -prog high_speed_train \
    -O0 \
    -keep-all-files \
    -results-dir RESULTS \
    -main root_package.start
```

8. Appendix

Related subjects :
   8.1. Glossary
   8.2. What is static verification?

8.1. 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 from the command line, rather than via the launcher Graphical User Interface.
Category	One of four types of orange check: *potential bug, inconclusive check, data set issue* and *basic imprecision*
Certain error	See red error
Check	Test performed by PolySpace during analysis, 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
Orange warning	Check found to represent a possible error, which may be revealed on further investigation.
PolySpace Approach	The manner of use of PolySpace to achieve a particular goal, with reference to a collection of techniques and guiding principles.
Precision	An analysis which includes few inconclusive orange checks is said to be precise
Progress text	Output from PolySpace during analysis to indicate what proportion of the analysis has been completed. Could be considered as a "textual progress bar".
Red error	Check found to represent a definite error
Review	Inspection of the results produced by a PolySpace analysis, using the Viewer.
Scaling option	Option applied when an application submitted to PolySpace Verifier proves to be bigger or more complex than is practical.
Selectivity	The ratio of (green + grey + red) / (total amount of checks)
Unreachable code	Dead code

8.2. What is static verification?

Static Verification is a broad term, and is applicable to any tool which derives dynamic properties of a program without actually executing it. Static Verification differs significantly from other techniques, such as run-time debugging, in that the analysis it provides is not based on a given test case or set of test cases. The dynamic properties obtained in the PolySpace analysis are true for all executions of the software.

Most Static Verification tools only provide an analysis of the complexity of the software, in a search for constructs which may be potentially dangerous.

PolySpace provides deep-level analysis identifying almost all run-time errors and possible access conflicts on global shared data.

The idea is to use an approximation of the software under analysis, using safe and representative approximations of software operations and data.

An example is given below:

```
for (i=0 ; i<1000 ; ++i)
{    tab[i] = foo(i);
}
```

To check that the variable 'i' never overflows the range of 'tab' a traditional approach would be to enumerate each possible value of 'i'. One thousand checks would be needed.

Using the static verification approach, the variable 'i' is modelled by its variation domain. For instance the model of 'i' is that it belongs to the [0..999] static interval. (Depending on the complexity of the data, convex polyhedrons, integer lattices and more elaborated models are also used for this purpose).

Any approximation leads by definition to information loss. For instance, the information that 'i' is incremented by one every cycle in the loop is lost. However the important fact is that this information is not required to ensure that no range error will occur; it is only necessary to prove that the variation domain of 'i' is smaller than the range of 'tab'. Only one check is required to establish that – and hence the gain in efficiency compared to traditional approaches.

Static code verification has an exact solution but it is generally not practical, as it would in general require the enumeration of all possible test cases. As a result, approximation is required if a usable tool is to result.

Exhaustiveness

Nothing is lost in terms of exhaustiveness. The reason is that PolySpace works by performing upper approximations. In other words, the computed variation domain of any program variable is always a superset of its actual variation domain. The direct consequence is that no run time error (RTE) item to be checked can be missed by PolySpace.