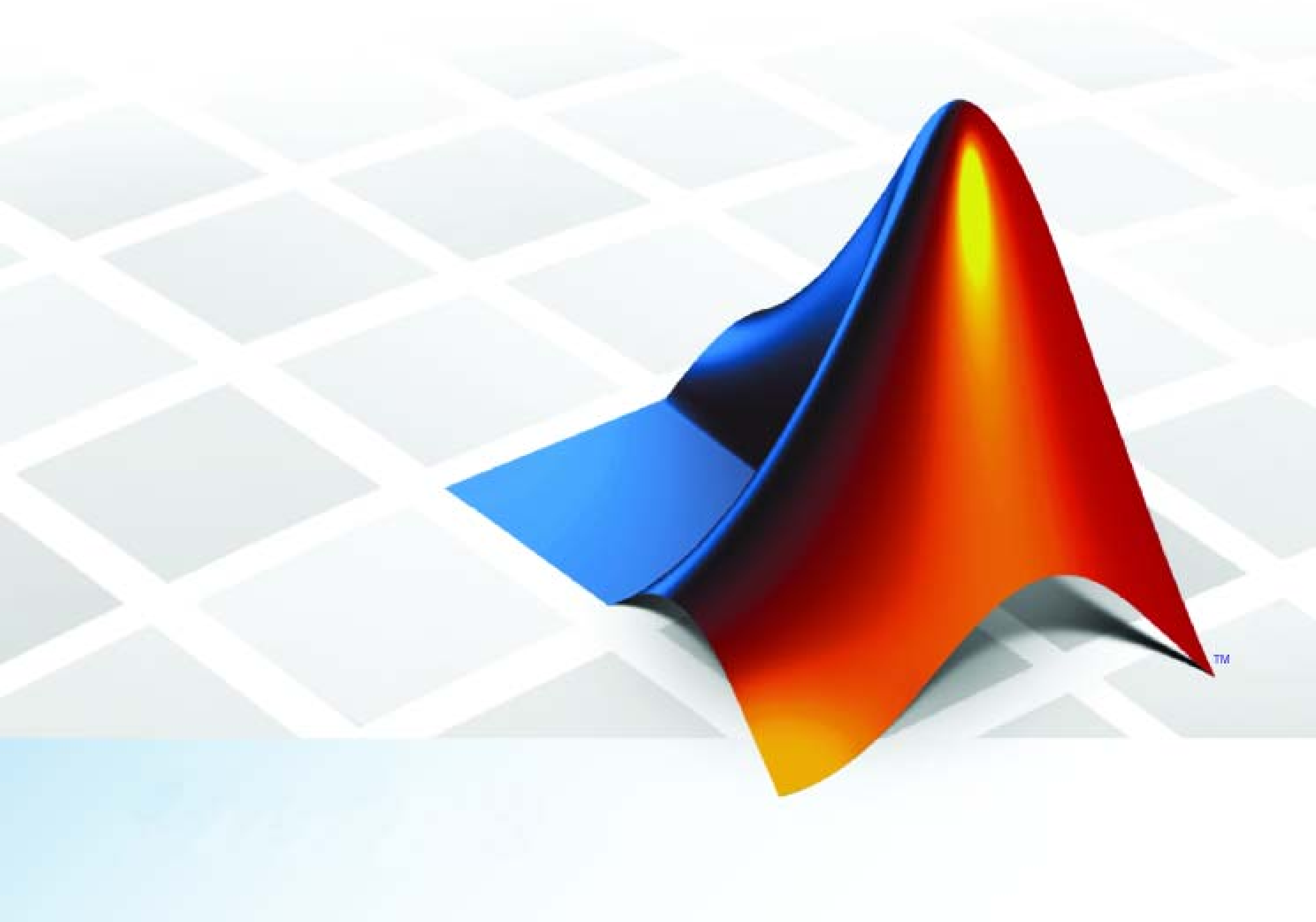
# PolySpace® Products for Ada 5

Reference



The MathWorks™

Accelerating the pace of engineering and science

**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.

*PolySpace® Products for Ada Reference*

**Revision History**

| | | |
|---|---|---|
| March 2009 | Online Only | Revised for Version 5.3 (Release 2009a) |

# Contents

## 1 Options Description

# Options Description

## General Options

**In this section...**

### Overview

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

### -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 ...
```

## -date date

This option specifies a date stamp for the verification 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 verification

**Example shell script entry:**

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

## -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"`

## -verif-version verif-version

Specifies the version identifier of the verification. This option can be used to identify different verifications. This information is identified in the GUI as the *Version*. **Default:** 1.0. **Example shell script entry:** `polyspace-ada -verif-version 1.3 ...`

## -voa (Deprecated)

**Note** This option is deprecated in R2009a and later releases. VOA checks are now enabled by default.

To disable VOA checks, you can use the option `-extra-flags -no-voa`.

This option enables the inspection of calculated domains for simple type assignments (scalar or float).

VOA checks are generated on `"="`of some scalar assignments to give the ranges. VOA checks are not available for volatile variables.

**Default**:

Enabled.

**Note** Depending on code optimization, this check may not be present at all assignment locations

## -keep-all-files

When this option is set, all intermediate results and associated working files are retained. Consequently, it is possible to restart PolySpace™ 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 PolySpace from scratch.

By default, intermediate results and associated working files are erased when they are no longer needed by PolySpace.

## -continue-with-red-error (Deprecated)

**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**:

PolySpace stops upon finding red errors.

**Example shell script entry** :

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

## -continue-with-existing-host

When this option is set, the verification 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:** PolySpace stops when the host configuration is incorrect or the system is under specified. **Example Shell Script Entry**: `polyspace-ada -continue-with-existing-host ...`

## -allow-unsupported-linux

This option specifies that PolySpace will be launched on an unsupported OS Linux® distribution.

PolySpace software supports the Linux distributions listed in "Hardware and Software Requirements" in the *PolySpace Installation Guide*.

For all other Linux distributions, you may be able to verify code using the `-allow-unsupported-linux` option, but a warning will be displayed in the log file informing you of possible incorrect behaviors:

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

```
***                   ***
***        WARNING         ***
***                   ***
***   You are running PolySpace 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 ...
```

## -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 verified, 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.

## -extensions-for-spec-files and -ada-include-dir

The -extensions-for-specs-files option specifies the file extension for files "*F*" which will be verified 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 body 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
  - in configuration 1: its value will be its full range
  - 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"
```

## -results-dir directory

This option specifies the directory in which PolySpace will write the results of the verification. 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 ...
```

## -pre-analysis-command file or "command"

When this option is used, the specified script file or command is run before the verification 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 on a Linux workstation:

```
polyspace-ada -pre-analysis-command `pwd`/replace_keywords
```

where `replace_keywords` is the following script :

```
#!/usr/bin/perl
my $TOOLS_VERSION = "V1_4_1";
binmode STDOUT;

# Process every line from STDIN until EOF
while ($line = <STDIN>)
{
  # Change Volatile to Import
  $line =~ s/Volatile/Import/;
  print $line;
}
```

---

**Note** If you are running PolySpace software Version 5.1 (r2008a) or later on a Windows system, you cannot use Cygwin shell scripts. Since Cygwin is no longer included with PolySpace software, all files must be executable by Windows. To support scripting, the PolySpace installation now includes Perl. You can access Perl in

```
%POLYSPACE_ADA%\Verifier\tools\perl\win32\bin\perl.exe
```

---

To run the Perl script provided in the previous example on a Windows® workstation, you must use the option `-pre-analysis-command` with the absolute path to the Perl script, for example:

```
%POLYSPACE_ADA%\Verifier\bin\polyspace-cpp.exe
-pre-analysis-command
%POLYSPACE_ADA%\Verifier\tools\perl\win32\bin\perl.exe
<absolute_path>\replace_keywords
```

## -post-analysis-command file or "command"

When this option is used, the specified script file or command is executed once the verification has completed.

The script or command is executed in the results directory of the verification.

Execution occurs after the last part of the verification. The last part of is determined by the –to option.

---

**Note** Depending on the architecture used, notably when using a sever verification, 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 indicate to the client that the verification is complete. The command looks like:

```
polyspace-ada -post-analysis-command `pwd`/end_email
```

where `end_email` is an appropriate Perl script:

---

**Note** If you are running PolySpace software version 5.1 (r2008a) or later on a Windows system, you cannot use Cygwin shell scripts. Since Cygwin is no longer included with PolySpace software, all files must be executable by Windows. To support scripting, the PolySpace installation now includes Perl. You can access Perl in

```
%POLYSPACE_ADA%\Verifier\tools\perl\win32\bin\perl.exe
```

---

To run the Perl script provided in the previous example on a Windows workstation, you must use this option with the absolute path to the Perl script, for example:

```
%POLYSPACE_ADA%\Verifier\bin\polyspace-cpp.exe
-post-analysis-command
%POLYSPACE_ADA%\Verifier\tools\perl\win32\bin\perl.exe
<absolute_path>\end_emails
```

## Target/Compiler Options

**In this section...**

### -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 ...
```

### -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 verified by using the corresponding include files for that OS. For instance, in order to verify a 'greenhills' project it is necessary to use the option –ada-include-dir<*`path_to_the_greenhills_include_folder`*>.

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 ...
```

## Compliance with Standards Options

**In this section...**

### -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.

## -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 ./example.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:

- PolySpace found an error in ./example1.adb:21:07: operator for type "T1" defined at ./example1.adb:1 is not directly visible.
- PolySpace 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 ...
```

## Permissiveness/Strictness

Verification mode has two options: -permissive and -strict.

When either of these two options is selected, the following options may be selected independently: -no-automatic-stubbing, -continue-with-in-out-niv and –continue-with-all-niv.

### -permissive

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

## -continue-with-in-out-niv

Ada Standard requires 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 verification. 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 gray (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; -- gray code on this line by default,
  green with -continuewith-
  in-out-niv
 end Main;
```

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

## -strict

Strict mode of PolySpace.

In Ada, equivalent to -no-automatic-stubbing

## -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 cannot have any side effects on global variables. If a function with global side effects must be stubbed, it must be done by hand.

**Benefits**:

You may want to use this option for several reasons

- You want to make sure the entire code is provided: this can be the case when verifying a large piece of code. When the verification 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
- You want to write stubs himself to increase the selectivity and speed of the verification.

**Example**:

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

## -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 gray (dead code). All remaining checks in the procedure are gray.

**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 verification.

```
procedure Main is
  I,T,No: Integer;
 begin
  if (No = 0) -- red NIV, with or without the option
  then
   I := 1/I; -- gray code by default, red NIV with the option
  end if;
  if (T = 0) -- gray 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 verification will stop at the end of the Software Safety level 1, as for any other red error detection. In order for the verification to continue (in level 2, 3, 4 in this case), the user must set the option -continue-with-red-error.

## PolySpace Inner Settings Options

**In this section...**

### -main main_subprogram_name

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

**Note** This option is exclusive with -main-generator.

**Example**:

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

### -main-generator

The -main-generator is **exclusive** with the -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 Client and PolySpace Server:**

- For PolySpace Client: the -main-generator option is set by default and the -main option can replace it if activated
- For PolySpace Server: 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)
```

## Stubbing

- "-import-are-not-volatile" on page 1-21
- "-export-are-not-volatile" on page 1-21
- "-init-stubbing-vars-random" on page 1-21
- "-init-stubbing-vars-zero-or-random" on page 1-22

### -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 ...`

### -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`

### -init-stubbing-vars-random

Force initialization of uninitialized global variables to a random value.

**Default**:

Uninitialized global variables give warnings or errors, depending on the context.

**Example**:

```
polyspace-ada -init-stubbing-vars-random -main...
```

### -init-stubbing-vars-zero-or-random

Initialize uninitialized globals variables:

- with zero if the type contains zero,
- with random otherwise

**Default**:

Uninitialized global variables give warnings or errors, depending on the context.

**Example**:

```
polyspace-ada -init-stubbing-vars-zero-or-random -main ...
```

## Assumptions

- "-ignore-float-rounding" on page 1-22
- "-known-NTC proc1[,proc2[,...]]" on page 1-23

### -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 ...
```

### -known-NTC proc1[,proc2[,...]]

After a few verifications, 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"
```

### -machine-architecture

This option specifies whether verification runs in 32 or 64-bit mode.

**Note** You should only use the option `-machine-architecture 64` for verifications that fail due to insufficient memory in 32 bit mode. Otherwise, you should always run in 32–bit mode.

Available options are:

- `-machine-architecture auto` – Verification always runs in 32-bit mode.
- `-machine-architecture 32` – Verification always runs in 32-bit mode.
- `-machine-architecture 64` – Verification always runs in 64-bit mode.

**Default**:

```
auto
```

**Example Shell Script Entry**:

```
polyspace-ada -machine-architecture auto
```

## -max-processes

This option specifies the maximum number of processes that can run simultaneously on a multi-core system. The valid range is 1 to 128.

**Note** To disable parallel processing, set: `-max-processes 1`.

**Default**:

4

**Example Shell Script Entry**:

```
polyspace-ada -max-processes 1
```

## Others

- "-extra-flags option-extra-flag" on page 1-24
- "-ada95-extra-flags extra-flag (Ada95 only)" on page 1-25

### -extra-flags option-extra-flag

This option specifies an expert option to be added to the verification. 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 verifications. **Default**: No extra flags. **Example Shell Script Entry**: `polyspace-ada -extra-flags -param1 -extra-flags -param2 \   -extra-flags 10 ...`

## -ada95-extra-flags extra-flag (Ada95 only)

This option specifies an expert option to be added to the verification. 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 verifications.

**Default**:

No extra flags.

**Example Shell Script Entry**:

```
polyspace-ada  ada95-extra-flags -param1...
```

## Precision Options

**In this section...**

### -from verification-phase

This option specifies the verification phase to start from. It can only be used on an existing verification, possibly to elaborate on the results that you have already obtained.

For example, if a verification has been completed  -to pass1, PolySpace can be restarted *-from* pass1 and hence save on verification time.

The option is usually used in a verification 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** :

- This option can only be used for client verifications. All server verifications start from *scratch*.
- Unless the *scratch* option is used, this option can be used only if the previous verification was launched using the option *-keep-all-files* .

- This option cannot be used if you modify the source code between verifications.

**Default** :

scratch

**Example Shell Script Entry** :

```
polyspace-ada -from pass0 ...
```

## -to verification-phase

Specifies the verification phase after which PolySpace 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 longer verification time

**Possible values:**

- `compile` or “`Source Compliance Checking`”
- `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 it has reached *pass20*.

**Default**:

*pass4*

**Example Shell Script Entry**:

```
polyspace-ada -to "Software Safety Analysis level 3"...

polyspace-ada -to pass0 ...
```

## -O(0-3)

This option specifies the precision level to be used during verification. It provides higher selectivity in exchange for longer verification 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 verification.

We recommend beginning verification with the -O0 option. Red errors and gray code can then be addressed before relaunching PolySpace using this option to apply a higher 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 a longer verification time
- 
  - -O0 corresponds to static interval verification.
  - -O1 corresponds to a complex polyhedron model of domain values.
  - -O2 corresponds to more complex algorithms that 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 verification time, such as an hour per 1000 lines of code.

**Default**:

-O2

**Example Shell Script Entry**:

```
polyspace-ada -O1 -to pass4 ...
```

## -modules-precision mod1:O(0-3)[,mod2:O(0-3)[,...]]

This option is used to specify the list of Ada packages to be verified with a different precision from that specified generally -O(0..3) for the verification.

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, ...`

## -array-expansion-size number

This option forces PolySpace to verify 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 verified will have an impact on the verification 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 ...
```

## -path-sensitivity-delta number

This option is used to improve interprocedural verification 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 verification time.

Consider two verification, one with this option set to 1 (with), and one without this option (without)

- a level 1 verification set to (with) (pass1) will provide results equivalent to a level 1 or 2 verification set to (without).
- a level 1 verification set to (with) can last x times longer than a cumulated level 1+2 verification set to (without). Note that "x" might be exponential.
- the same applies to a level 2 verification set to (with): it is equivalent to a level 3 or 4 set to (without), with a potentially exponential verification time for (a)

**Gains using the option**

- (+) The highest selectivity is achieved in level 2, so there is no need to wait until level 4.
- (-) This parameter may exponentially increase verification time and might result in an even longer verification time than for a cumulated verification of level 1+2+3+4.
- (-) This option can only be used for packages with less than 1000 lines of code.

**Default:**

0

**Example Shell Script Entry:**

```
polyspace-ada -path-sensitivity-delta 1 ...
```

## -variables-to-expand var1[,var2[,...]]

Specifies aggregate variables (record, ...) that will be split into independent variables for the purpose of verification. 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 verified. **Example**: `polyspace-ada -variables-to-expand pkg.rec1,pkg2.recF \    -variable-expansion-depth 4 -main ...`

## -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 verified will have an impact on the verification 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 verification 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 verification 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 ...`

## Multitasking Options (PolySpace Server Only)

**In this section...**

**Note** Concurrency options are not compatible with the "-main-generator" on page 1-20 option.

### -entry-points str1[,str2[,...]]

This option is used to specify the tasks/entry points to be verified 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 ...
```

### -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"
```

## -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 ...
```

## Batch Options

**In this section...**

### -server server_name_or_ip[:port_number]

Using `polyspace-remote[-desktop]-[ada] [ server [name or IP address][:<port number>]]` allows you to send a verification to a specific or referenced PolySpace server.

**Note** If the option –server is not specified, the default server referenced in the PolySpace-Launcher.prf configuration file will be used as the 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
```

### -h[elp]

Displays simple help in the shell window that provides information on all options.

**Example Shell Script Entry**:

```
polyspace-ada  h
```

## -v | -version

Displays the PolySpace version number.

**Example Shell Script Entry**:

```
polyspace-ada  v
```

It will show a result similar to:

```
PolySpace r2008b

Copyright (c) 1999-2008 The Mathworks, Inc.
```

## -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"
```

2

# Check Descriptions

## Colored Source Code for Ada

**In this section...**

| In this section... |
|---|
| "Value on Assignment: VOA" on page 2-51 |
| "Inspection Points: IPT" on page 2-53 |

## Non-Initialized Variable: NIV/NIVL

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

### Examples

#### Ada Example.

```
package NIV is
 type Pixel is
  record
   X : Integer;
   Y : Integer;
  end record;
 procedure MAIN;
 function Random_Bool return Boolean;
 end NIV;

 package body NIV is

  type TwentyFloat is array (Integer range 1.. 20) of Float;

  procedure AddPixelValue(Vpixel : Pixel) is
   Z : Integer;
  begin
   if (Vpixel.X < 3) then
    Z := Vpixel.Y + Vpixel.X; -- NIV error: Y field
not initialized
   end if;
  end AddPixelValue;

  procedure MAIN is
   B : Twentyfloat;
   Vpixel : Pixel;
```

```
27   begin
28    if (Random_Bool) then
29     Vpixel.X := 1;
30     AddPixelValue(Vpixel); -- NTC Error: because of NIV error
31  in call
32    end if;
33
34    for I in 2 .. Twentyfloat'Last loop
35     if ((I mod 2) = 0) then
36      B(I) := 0.0;
37     end if;
38    end loop;
39    B(2) := B(4) + B(5); -- NIV Warning because
40 B(odd) not initialized
41   end MAIN;
42
43  end NIV;
```

**Explanation.** The result of the addition is unknown at line 19 because *Vpixel.Y* is not initialized (gray 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 vs. 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 initialization 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, PolySpace highlights a dangerous non-initialized variable. Even if it is not dangerous, as shown in the *Niv_Not_Dangerous* procedure, PolySpace also highlights the non-initialized variable at line 30. To be more permissive with standard, the **-continue-with-in-out-niv** option permits to continuation of the verification for the rest of the sources even if a red error stays in place at lines 28 and 31.

## 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.

| | Records and Other Variable Types | Integer Variable Types | Function |
|---|---|---|---|
| `pragma interface (C, variable_name)` `pragma import (C, variable_name)` | • green NIV<br>• Permanent random value | • No NIV check<br>• Permanent random value | • same behavior as -automatic-stubbing<br>• 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 color of the NIV.

## Type Access Variables

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

| | Records and Other Variable Types | Integer Variable Types |
|---|---|---|
| `Type a_new_type is access another_type;` | • orange NIV<br>• Permanent random value | • No NIV check<br>• 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.

### Address Clauses

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

| Address Clause | Records and Other Variable Types | Integer Variable Types |
|---|---|---|
| `for variable_name'address use 16#1234abcd#;` `for variable_name'other'address use;` | • orange NIV<br>• Permanent random value | • No NIV check<br>• 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 anything within its whole range between one read access and the next.

## Division by Zero: ZDV

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

### Ada Example:

```
package ZDV is
 function Random_Bool return Boolean;
 procedure ZDVS (X : Integer);
 procedure ZDVF (Z : Float);
 procedure MAIN;
end ZDV;

package body ZDV is

  procedure ZDVS(X : Integer) is
   I : Integer;
   J : Integer := 1;
  begin
   I := 1024 / (J-X); -- ZDV ERROR: Scalar Division by Zero
  end ZDVS;

  procedure ZDVF(Z : Float) is
   I : Float;
   J : Float := 1.0;
  begin
   I := 1024.0 / (J-Z); -- ZDV ERROR: float Division by Zero
  end ZDVF;

  procedure MAIN is
  begin
   if (random_bool) then
    ZDVS(1); -- NTC ERROR: ZDV.ZDVS call never terminates
   end if ;
   if (Random_Bool) then
    ZDVF(1.0); -- NTC ERROR: ZDV.ZDVF call never terminates
   end if;
  end MAIN;

end ZDV;



```

## Arithmetic Exceptions: EXCP

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

- Argument of *sqrt* must be positive
- Argument of *tan* must be different from pi/2 modulo pi
- Argument of *log* must be strictly positive
- Argument of *acos* and *asin* must be within [-1..1]
- 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

```

With Ada.Numerics; Use Ada.Numerics;
With Ada.Numerics.Aux; Use Ada.Numerics.Aux;

package EXCP is
 function Bool_Random return Boolean;
 procedure MAIN;
end EXCP;

package body EXCP is

-- implementation dependant in Ada.Numerics.Aux: subtype Double is Long_Float;
 M_PI_2 : constant Double := Pi/2.0; -- pi/2

 procedure MAIN is
  IRes, ILeft, IRight : Integer;
  Dbl_Random : Double;
  pragma Volatile_ada.htm (dbl_Random);

  SP : Double := Dbl_Random;
  P : Double := Dbl_Random;
  SN : Double := Dbl_Random;
  N : Double := Dbl_Random;
  NO_TRIG_VAL : Double := Dbl_Random;
  res : Double;
  Fres : Long_Float;
 begin
  -- assert is used to redefine range values of a variable.
  pragma assert(SP > 0.0);
  pragma assert(P >= 0.0);
  pragma assert(SN < 0.0);
  pragma assert(N <= 0.0);
  pragma assert(NO_TRIG_VAL < -1.0 or NO_TRIG_VAL > 1.0);

  if (bool_random) then
   res := sqrt(sn); -- EXCP ERROR: argument of SQRT must be positive.
```

```
    end if ;
    if (bool_random) then
     res := tan(M_PI_2); -- EXCP Warning: Float argument of TAN

            -- may be different than pi/2 modulo pi.
    end if;
    if (bool_random) then
     res := asin(no_trig_val); -- EXCP ERROR: float argument of
ASIN is not in -1..1
    end if;
    if (bool_random) then
     res := acos(no_trig_val); -- EXCP ERROR: float argument of
ACOS is not in -1..1
    end if;
    if (bool_random) then
    res := log(n); -- EXCP ERROR: float argument of LOG is not
strictly positive
    end if;
   if (bool_random) then
     res := exp(710.0); -- EXCP ERROR: float argument of EXP
is not less than or equal to 709 or 88
    end if;

    -- range results on trigonometric functions
    if (Bool_Random) then
     Res := Sin (dbl_random); -- -1 <= Res <= 1
     Res := Cos (dbl_random); -- -1 <= Res <= 1
     Res := atan(dbl_random); -- -pi/2 <= Res <= pi/2
    end if;

    -- Arithmetic functions where there is no check currently
implemented
    if (Bool_Random) then
     Res := cosh(dbl_random);
     Res := tanh(dbl_random);
    end if;
   end MAIN;
  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, regardless of the input parameters. The Ada83 standard gives more details about domain and range error for each maths function.

## Scalar and Float Underflow/Overflow : UOVFL

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

### Ada Example

```
package UOVFL is
 function Bool_Random return Boolean;
 procedure MAIN;
end UOVFL;

package body UOVFL is

 procedure MAIN is
  I : Integer;
   DValue : Long_float;
  begin
   if (Bool_Random) then
    I := 2**30;
    I := 2 * (I - 1);    -- integer UOVFL verified on "*" and "-"
   end if;
   if (Bool_Random) then
    DValue := Long_float(Float'Last);
    DValue := 2.0 * DValue + 1.0; -- float UOVFL verified on "*" and "+"
   end if;
  end MAIN;
 end UOVFL;
```

### Explanation

PolySpace can detect that there is neither an underflow nor an overflow on * and - operators at lines 16 and 18.

## 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 26, 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 30.

## 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

```
package UNFL is
 function Bool_Random return Boolean;
 procedure MAIN;
end UNFL;

package body UNFL is

 procedure UNFL_ARRAY is
  A : array(1..20) of Float;
   J : Integer;
  begin
   for I in A'Last.. A'First loop
    A(I) := 0.0 ;
    J := I - 1;
   end loop;
   A(J) := 0.0; -- UNFL ERROR: underflow array index_ada.htm
  end UNFL_ARRAY;

  procedure UNFL_ARITHMETIC is
   I : Integer;
   FValue : Float;
  begin

   if (Bool_Random) then
    I := -2**31;
    I := I - 1 ; -- UNFL ERROR: -2**31-1 is integer underflow
   end if;
   if (Bool_Random) then
    FValue := Float'First;
    FValue := -2.0 * FValue; -- UNFL ERROR: float variable is overflow
   end if;
  end UNFL_ARITHMETIC;

  procedure MAIN is
  begin
   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 26, 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 30.

## 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

```

package CORS is
 function Bool_Random return Boolean;
 procedure MAIN;
 function INT_VALUE (S : String) return Integer;
 type PSTCOLORS is (ORANGE, RED, GREY, GREEN);
 type ADCFUZZY is (LOW, MEDIUM, HIGH);
end CORS;

 package body CORS is

  type STR_ENUM is (AA,BB);

  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"); -- Info 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 whether 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, PolySpace results assume the full possible range of the returned type, regardless 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 indicated 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.

## Array Length Check: COR

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

### Ada Example

```

with Dname;
package CORL is
 function Bool_Random return Boolean;
 type Name_Type is array (1 .. 6) of Character;
 procedure Put (C : Character);
 procedure Put (S : String);
 procedure MAIN;
end CORL;

 package body CORL is

  STR_CST : constant NAME_TYPE := "String";

  procedure MAIN is
   Str1,Str2,Str3 : String(1..6);
   Arr1 : array(1..10) of Integer;
  begin

   if (Bool_Random) then
   Str1 := "abcdefg"; -- COR ERROR: Too many elements in array must have 6
   end if;
   if (Bool_Random) then
    Arr1 := (1,2,3,4,5,6,7,8,9); -- COR ERROR: Not enough elements in array, must have 10
   end if ;
   if (Bool_Random) then
    Str1 := "abcdef";
    Str2 := "ghijkl";
    Str3 := Str1 & Str2; -- COR Warning: Length might not be compatible with 1 .. 6
    Put(Str3);
    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.

## DIGITS Value Check: COR

Checks the length of *DIGITS* constructions.

### Ada Example

```
package DIGIT is
 procedure MAIN;
end DIGIT;

package body DIGIT is -- NTC ERROR: COR propagation

 type T is digits 4 range 0.0 .. 100.0;
 subtype T1 is T
  digits 1000 range 0.0 .. 100.0; -- COR ERROR: digits value
is too large, highest possible value is 4

```

```
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.

## 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 Server shows an error on the *DELTA* value. The message gives the smallest available value, *0.5* in this case.

## 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;
```

```
end SRANGE;

package body SRANGE is

 TYPE SENSOR IS NEW INTEGER RANGE 0 .. 10;

 TYPE REC2(D : DINT := 1) IS RECORD -- COR Warning: Value might not be in range 1 .. 3
  U : INF_ARRAY(1 .. D, D .. 3) := (1 .. D =>
            (D .. 3 => 1));
 END RECORD;
 TYPE REC3(D : DINT := 1) IS RECORD -- COR Error: Value is not in range 1 .. 3
  U : INF_ARRAY(1 .. D, D .. 3) := (1 .. D =>
            (D .. 3 => 1));
 END RECORD;

 PROCEDURE VALUE_RANGE is
  VAL : INTEGER;
  pragma Volatile(VAL);
  SLICE_A2 : REC2(VAL); -- NIV and COR warning: Value might not be in range 0 .. 10
  SLICE_A3 : REC3(4); -- Unreacheable code: because of COR Error in REC3
 BEGIN
  NULL;
 END VALUE_RANGE;

 PROCEDURE MAIN is
  Digval : Sensor;
 begin
  if IsNatural then
   declare
    TYPE Sub_sensor is new Natural range -1 .. 5; -- COR Error: Static value is not in range of 0 .. 16#7FFF_FFFF#
   begin
    null;
   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*. PolySpace 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 lines 19, 38, 46 and 54 PolySpace displays errors on out of range values.

## 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.

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

## 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 lines 21 and 24, PolySpace Server indicates the incorrect dimension of the double array *Newarray* of type *TBASE*.

## Aggregate Versus Definition Check: COR

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

### Ada Example

```

package AGGDEF is
 PROCEDURE MAIN;
 PROCEDURE COMMENT (A: STRING);
 function RANDOM return BOOLEAN;
end AGGDEF;

package body AGGDEF is

  TYPE REC1 (DISC : INTEGER := 5) IS RECORD
   NULL;
  END RECORD;

  TYPE REC2 (DISC : INTEGER) IS RECORD
   NULL;
  END RECORD;

  TYPE REC3 is RECORD
   COMP1 : REC1(6);
   COMP2 : REC2(6);
  END RECORD;

  FUNCTION IDENT_INT(VAL : INTEGER) RETURN INTEGER IS
 BEGIN
   RETURN VAL;
  END IDENT_INT;

  PROCEDURE AGGDEF_INIT is -- AGGREGATE INITIALISATION
   OBJ3 : REC3;
  BEGIN
   if random then
    OBJ3 :=
     ((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 lines 33 and 35, PolySpace indicates the incompatible aggregate declaration on *OBJ3*. The aggregate definition with a discriminant of value *6*, is not compatible with a discriminant of value *7*.

## 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 the component FIELD1 at line 21.

## Sub-Aggregates Dimension Check: COR

Checks the dimension of sub-aggregates.

### Ada Example

```
1
```

```
package SUBDIM is
 PROCEDURE MAIN;
 FUNCTION EQUAL ( A : Integer; B : Integer) return Boolean;
end SUBDIM;

package body SUBDIM is


  TYPE DOUBLE_TABLE IS ARRAY(INTEGER RANGE <>, INTEGER RANGE <>) OF INTEGER;
  TYPE CHOICE_INDEX IS (H, I);
  TYPE CHOICE_CNTR IS ARRAY(CHOICE_INDEX) OF INTEGER;

  CNTR : CHOICE_CNTR := (CHOICE_INDEX => 0);

  FUNCTION CALC (A : CHOICE_INDEX; B : INTEGER)
    RETURN INTEGER IS
  BEGIN
   CNTR(A) := CNTR(A) + 1;
   RETURN B;
  END CALC;

  PROCEDURE MAIN IS
   A1 : DOUBLE_TABLE(1 .. 3, 2 .. 5);
  BEGIN
   CNTR := (CHOICE_INDEX => 1);
   if (EQUAL(CNTR(H),CNTR(I))) then
    A1 := ( -- COR ERROR: Sub-agreggates do not have the same dimension
     1 => (CALC(H,2) .. CALC(I,5) => -4),
     2 => (CALC(H,3) .. CALC(I,6) => -5),
     3 => (CALC(H,2) .. CALC(I,5) => -3) );
   END IF;
  END MAIN;

 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.

## Characters Check: COR

Checks the construction using the *character* type.

### Ada Example

```

package CHAR is
 procedure Main;
 function Random return Boolean;
end CHAR;


package body CHAR is

  type ALL_Char is array (Integer) of Character;
  TYPE Sub_Character is new Character range 'A' .. 'E';
  TYPE TabC is array (1 .. 5) of Sub_Character;

  FUNCTION INIT return character is
   VAR : TabC := "abcdf"; -- COR Error: Character is not in range 'A' .. 'E'
  begin
   return 'A';
  end;

  procedure MAIN is
   Var : ALL_Char;
  BEGIN
   IF RANDOM THEN
    Var(1) := Init; -- NTC ERROR: propagation of the COR error
   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 indicates 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 PolySpace when the *null string literal* is assigned incorrectly.

## 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 16: *Ref* is set to *x'access* and *Ref* is defined in same block or in a deeper one. This is authorized.

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 is prohibited and PolySpace checks at lines 21 and 26.

---

**Note** The warning at line 26 is due to the combination of a red check because of *y'access* at line 38 and a green one for *x'access* at line 41.

---

## Explicit Dereference of a Null Pointer: COR

When a pointer is dereferenced, PolySpace checks whether or not it is a null pointer.

### Ada Example

```
package CORNULL is
 procedure main;
end CORNULL;

package body CORNULL is
 type ptr_type is access all integer;
 ptr : ptr_type;
 A : aliased integer := 10;

  procedure main is
  begin
   ptr := A'access;
   if (ptr /= null) then
    ptr.all := ptr.all + 1; -- COR Warning: Explicit dereference of possibly null value
    pragma assert (ptr.all = 10); -- COR Warning: Explicit dereference of possibly null value
    null;
   end if;
  end main;
 end CORNULL;

```

### Explanation

At line 14 and line 15, PolySpace checks the null value of *ptr* pointer. As PolySpace does not perform pointer verification, it is not able to be precise on such a construction.

These checks are currently always orange.

## 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

```
package TAG is

 type Tag_Type is tagged record
  C1 : Natural;
 end record;

 type DTag_Type is new Tag_Type with record
  C2 : Float;
 end record;

  type DDTag_Type is new DTag_Type with record
   C3 : Boolean;
  end record;

  procedure Main;

 end TAG;


 package body TAG is

  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 each one with a COR warning.

## Power Arithmetic: POW

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

### Ada Example

```
With Ada.Numerics; Use Ada.Numerics;
With Ada.Numerics.Aux; Use Ada.Numerics.Aux;

package POWF is
 function Bool_Random return Boolean;
 procedure MAIN;
end POWF;

package body POWF is
```

```

 procedure MAIN is
  IRes, ILeft, IRight : Integer;
  Res, Dbl_Random : Double ;
  pragma Volatile(Dbl_Random);
 begin
  -- Implementation of Power arithmetic function with **
  if (Bool_Random) then
   ILeft := 0;
   IRight := -1;
   IRes:= ILeft ** IRight; -- POW ERROR: Power must be positive
  end if;
  if (Bool_Random) then
   ILeft := -2;
   IRight := -1;
   IRes:= ILeft ** IRight; -- POW ERROR: Power must be positive
  end if;

  ILeft := 2e8;
  IRight := 2;
  IRes:= ILeft ** IRight; -- otherwise OVFL Warning

  -- Implementation with double
  Res := Pow (dbl_Random, dbl_Random); -- POW Warning :
may be not positive
 end MAIN;
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 recognized by the Standard, PolySpace places a green check on the instruction *left**right* with *left:=right:=0*.

---

## User Assertion: ASRT

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

### Ada Example

```

package ASRT is
 function Bool_Random return Boolean;
 procedure MAIN;
end ASRT;

package body ASRT is

 subtype Intpos is Integer range 0..Integer'Last;
 subtype TenInt is Integer range 1..10;

 Val_Constant : constant Boolean := True;
 procedure MAIN is
  -- Init variables
  Flip_Flop, Flip_Or_val : Boolean;
  Ten_Random, Ten_Positive : TenInt;
  pragma Volatile_ada.htm (ten_random);
 begin

  if (Bool_Random) then
   -- Flip_Flop is randomly be True or False
   Flip_Flop := bool_random;

   -- Flip_Or_Val is always True
   Flip_Or_Val := Flip_Flop or Val_Constant;
   pragma assert(flip_flop=True or flip_flop=False); -- User assertion is verified
   pragma assert(Flip_Or_Val=False); -- ASRT ERROR: User assertion fails
  end if;
  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:

- 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 runtime errors.
- To redefine the range of variables as shown at line 32 where *ASRT.Ten_positive* is restrained to only a few values. PolySpace makes the assumption that if the program is executed with no runtime error at line 32, *Ten_positive* can only have a value greater than 5 after the line.

## 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 verification
- As other reds, code placed after them are gray (unreachable): the only color they can take is red. They are not "orange" NTL or NTC
- They can reveal a bug, or can simply just be informative

| Check | Description |
|---|---|
| NTL | A NTL is a loop for which the break condition is never met. Among NTLs, you will find the following examples:<br>• while( 1=1 )loop function_call; end loop; // informative NTL<br>• 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)<br>• 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:<br>• “f” contains a red error;<br>• “f” contains an NTL ;<br>• “f” contains an NTC;<br>• “f” contains an orange which is context dependant : it is either red or green: for this call, it makes the function crash.<br>**Note** 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 verified 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.

### 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.

```
package NTC is
 procedure MAIN;
 -- Stubbed function
 function Random_Boolean return Boolean;
end NTC;

package body NTC is

 procedure FOO (X : Integer) is
   Y : Integer;
  begin
   Y := 1 / X; -- ZDV Warning: Scalar division by zero may occur
   while (X >= 0) loop -- NTL ERROR: Loop never terminate
    if ( Y /= X) then
     Y := 1 / (Y-X);
    end if;
   end loop;
  end FOO;

  procedure MAIN is
  begin
   if (Random_Boolean) then
    FOO(0); -- NTC ERROR: Division by zero  in NTC.FOO (ZDV)
   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:

- The first never returns because of a division by zero (ZDV warning) at line 12 when $X = 0$.
- The second never terminates because of an infinite loop (red NTL) at line 13.

**Note** An NTC check can only be red.

## 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 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;
```

```
 procedure Main;
  A : Integer;
end NTC_entry;

 package body NTC_entry is

  task body MyTask is
  begin
   accept Start;
   A := A + 1; -- Gray code
   accept V842;
   A := A - 1; -- Gray code
   accept V842;
   A := A + 1; -- Gray code
   accept V842;
   A := A - 1; -- Gray code
  end MyTask;

  procedure Main is
   T1 : MyTask;
  begin
   A := 0;
   T1.Start;     -- NTC ERROR: entry task in the main
   T1.V842;
   T1.V842;
   T1.V842;
   pragma Assert(A=0); -- Gray code
  end Main;
 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 gray code on the main task body MyTask.

The only way to verify this code with PolySpace is to add another main procedure with a null body and to consider 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 gray 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`):

```
 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;
```

## Known Non Termination of Call: k-NTC

By using the **-known-NTC** option with a specified function at launch time, it 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.

Functions 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.

```
package KNTC is
 procedure Put_io (X : Integer);
 procedure get_data(Data : out Float; Status : out Integer);
 procedure store_data(Data : in Float);
 procedure SysHalt(Value : Integer);
 procedure MAIN;
end KNTC;

package body KNTC is

  -- known NTC function
  procedure SysHalt(Value : Integer) is
  begin
   Put_io(Value);
   loop -- Never terminate loop
    null;
   end loop;
  end SysHalt;

  procedure MAIN is
   Status : Integer := 1;
   Data : Float;
  begin

   while(Status = 1) loop
    -- get data
    get_data(Data, Status);
    if (status = 1) then
     store_data(data);
   end if;
   if (Status = 0) then
    SysHalt(1); -- k-NTC check: Call never terminate
   end if;
  end loop;
 end MAIN;
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.

## Non Termination of Loop: NTL

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

### Ada Example.

```

package NTL is
 procedure MAIN;
 -- Prototypes stubbed as pure functions
 procedure Send_Data (Data : in Float);
 procedure Update_Alpha (A : in Float);
 end NTL;

package body NTL is

  procedure MAIN is
   Acq, Vacq : Float;
   pragma Volatile_ada.htm (Vacq);
   -- Init variables
   Alpha : Float := 0.85;
   Filtered : Float := 0.0;
  begin
   loop    -- NTL information: Loop never terminates
    -- Acquisition
    Acq := Vacq;
    -- Treatment
    Filtered := Alpha * Acq + (1.0 - Alpha) * Filtered;
   -- Action
   Send_Data(Filtered);
   Update_Alpha(Alpha);
  end loop;
 end MAIN;
end NTL;

```

**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, PolySpace is still able to treat those cases.

**Another NTL Example: Error Propagation.** As with all other red errors, PolySpace does not continue with the verification in the current branch even with the **-continue-with-red-error** option. Due to the inside error, the (for, do-while, while) loop never terminates.

```
package NTLDO is
 procedure MAIN;
end NTLDO;

package body NTLDO is
 procedure MAIN is
  A : array(1..20) of Float;
  J : Integer;
 begin
   for I in A'First .. 21 loop -- NTL ERROR: propagation of OVFL ERROR
    A(I) := 0.0 ; -- OVFL Warning: 20 verification with I in [1,20] and one ERROR with I = 21
    J := I + 1;
   end loop;
  end MAIN;
 end NTLDO;
```

**Note** A NTL check can only be red.

## Sqrt, Sin, Cos, and Generic Elementary Functions

When the verified code uses some mathematical functions which are not supported by PolySpace, 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. VOA checks display 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 PolySpace. The following solution matches the user’s mathematical functions to PolySpace Server’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;

package Main is
 procedure Main;
end Main;


with New_Math; use New_Math;
with Types; use Types;

 package body Main is
  procedure Main is
  X : My_float;
  begin
   X := Cos(12.3); --voa displays [-1.0 .. 1.0]
   X := Sin(12.3); --voa displays [-1.0 .. 1.0]
   X ::= Sqrt(-1.5); --is red: NTC Error
  end;
 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 verification.

```
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.

## 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 gray color on every check and an UNR check entry.

## Ada Example

```
package UNR is
 type T_STATE is (Init, Wait, Intermediate, EndState);
 function STATE (State : in T_STATE) return Boolean;
 function Intermediate_State(I : in Integer) return T_STATE;
 function UNR_I return Integer;
 procedure MAIN;
end UNR;

package body UNR is

  function STATE (State : IN T_STATE) return Boolean is
  begin
   if State = Init then
    return False;
   end if ;
   return True;
  end STATE;

  function UNR_I return Integer is
   Res_End, Bool_Random : Boolean;
   I : Integer;
   Res_State : T_STATE;
   pragma Volatile_ada.htm (bool_random);
  begin
   Res_End := STATE(Init);
   if (Res_End = False) then
    Res_End := State(EndState);
    Res_State ::= Intermediate_State(0);
    if (Res_End = True or else Res_State = Wait) then -- UNR code
     Res_State := EndState;
    end if;
    -- Use of I which is not initialized
    if (Bool_Random) then
     Res_State := Intermediate_State(I); -- NIV ERROR
     if (Res_State = Intermediate) then -- UNR code because
of NIV error
      Res_State := EndState;
     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 colored gray.

- 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).
- At line 29 a conditional part of a conditional branch is always true and the other part never evaluated because of the standard definition of logical operator *or else*.
- The piece of code after a red error is never evaluated by PolySpace Server. The call to the function and the lines following line 34 are considered to be dead code. Correcting the red error and relaunching would allow the color to be revised.

## Value on Assignment: VOA

Check to establish the value taken by a variable on assignment.

VOA checks are only available on scalar variables. Some examples are given below.

---

**Note** PolySpace software does not show VOA on all assignments. To optimize performance, the software may not show VOA in some cases.

---

## Ada Example

```


Package VOA is

 subtype T_NBWAY is Integer range 1..8;
 subtype T_DIGITAL is Integer range 0..1;
 subtype T_ANALOGIC is Float range -10.0 .. 10.0;
 Zero_analogic : constant T_ANALOGIC
    := (T_ANALOGIC'Last - T_ANALOGIC'First)/ 2.0 - T_ANALOGIC'Last; -

  function Get_Analogic (Way : T_NBWAY) return T_Analogic;
  function Get_Digit (Way : T_NBWAY) return T_Digital;

  type VerifierColor is (Red, Green, Orange, Black);
  type RECOR is
   record
    A : Float;
    B : VerifierColor;
   end record;
  Var_rec : RECOR;

  Procedure MAIN;

 end VOA;

 package body VOA is

  Procedure MAIN is
   Way_io : T_NBWAY := T_NBWAY'First;
   Val_Sensor : T_ANALOGIC;
  Val_Digit : T_DIGITAL;
  volatile_Color : VerifierColor;
  pragma Volatile(Volatile_color);
  Volatile_Float : Float;
  pragma Volatile(Volatile_Float);
 begin

  for I in T_NBWAY'Range loop
   Val_Sensor := Get_Analogic(I); -- VOA: {-1E+1<=[expr]<=1E+1}
   Val_Digit := Get_Digit(I); -- VOA: {0<=[expr]<=1}
   if Val_Sensor < 0.0 then
    Val_Sensor := Zero_Analogic; -- VOA: {[expr]=0.0}
   end if;
  end loop;

  -- Example
  Var_Rec.A := Volatile_Float; -- VOA: {[expr]=float(32) range -3.41E+38..3.4E+38}
  Var_Rec.B := Volatile_color; -- VOA: {red<=[expr]<=black}

  -- Other possible but intrusive way to know a specific value
  pragma Inspection_Point (Way_io); -- inspection point computed range: {WAY_IO=1}

 end MAIN;

End VOA;
```

### Explanation

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

## 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

```


Package IPT is

 subtype T_NBWAY is Integer range 1..8;
 subtype T_DIGITAL is Integer range 0..1;
 subtype T_ANALOGIC is Float range -10.0 .. 10.0;
 Zero_analogic : constant T_ANALOGIC
  := (T_ANALOGIC'Last - T_ANALOGIC'First)/ 2.0 - T_ANALOGIC'Last; --

  function Get_Analogic (Way : T_NBWAY) return T_Analogic;
  function Get_Digit (Way : T_NBWAY) return T_Digital;

  type VerifierColor is (Red, Green, Orange, Black);
  type RECOR is
   record
    A : Float;
    B : VerifierColor;
   end record;
  Var_rec : RECOR;

  Procedure MAIN;

 end IPT;

 package body IPT is

  Procedure MAIN is
   Way_io : T_NBWAY := T_NBWAY'First;
   Val_Sensor : T_ANALOGIC;
   Val_Digit : T_DIGITAL;
   volatile_Color : VerifierColor;
   pragma Volatile(Volatile_color);
   Volatile_Float : Float;
   pragma Volatile(Volatile_Float);
  begin

   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_Float;
47    Var_Rec.B := Volatile_color;
48    pragma Inspection_Point (Var_Rec); -- IPT currently ignored
49    pragma Inspection_Point (Volatile_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.

3

# Approximations Used During Verification

## Why PolySpace Verification Uses Approximations

**In this section...**

### What is Static Verification

PolySpace software uses *static verification* to prove the absence of runtime errors. Static verification derives the dynamic properties of a program without actually executing it. This differs significantly from other techniques, such as runtime debugging, in that the verification it provides is not based on a given test case or set of test cases. The dynamic properties obtained in the PolySpace verification are true for all executions of the software.

PolySpace verification works by approximating the software under verification, using safe and representative approximations of software operations and data.

For example, consider the following code:

```
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 does have an exact solution, but that solution is generally not practical, as it would generally require the enumeration of all possible test cases. As a result, approximation is required.

## 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 runtime error (RTE) item to be checked can be missed by PolySpace.

# Examples

## Complete Examples

**In this section...**

### 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 Example

An Ada example. Note that we try to minimize verification 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 Example

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 Example

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
```