Polyspace® Products for Ada

Reference

R2013a

MathWorks®

How to Contact 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)
September 2009	Online Only	Revised for Version 5.4 (Release 2009b)
March 2010	Online Only	Revised for Version 5.5 (Release 2010a)
September 2010	Online Only	Revised for Version 6.0 (Release 2010b)
April 2011	Online Only	Revised for Version 6.1 (Release 2011a)
September 2011	Online Only	Revised for Version 6.2 (Release 2011b)
March 2012	Online Only	Revised for Version 6.3 (Release 2012a)
September 2012	Online Only	Revised for Version 6.4 (Release 2012b)
March 2013	Online Only	Revised for Version 6.5 (Release 2013a)

Contents

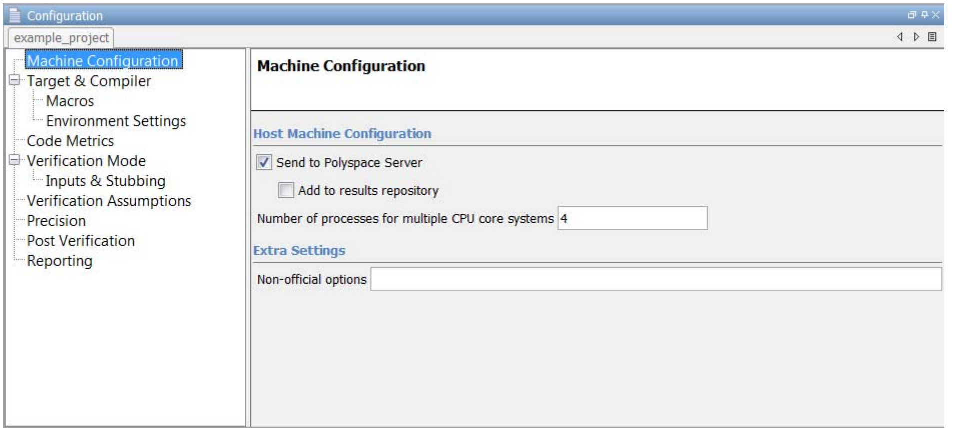
1 Option Descriptions

2 Check Descriptions

3 Approximations Used During Verification

Option Descriptions

Machine Configuration



In this section...

Machine Configuration Overview

Use **Machine Configuration** to specify where the verification is run, data storage, and host machine features. You can also specify options that MathWorks® might provide for fine-tuning your verifications.

Send to Polyspace Server

Specify whether verification runs on the server or client system

Settings

Default: On

☑ On
: Run verification on the Polyspace® server. You specify the server in the Polyspace Preferences dialog box.

☐ Off
: Run verification on the client system

Tips

- Specifying this option in the GUI sends the verification to the default server.
- You specify the default server in the **Server Configuration** tab of the Polyspace preferences dialog box (**Options > Preferences**).
- When specifying the `-server` option at the command line, you can specify the name or IP address of a specific server, along with the port number.
- If you do not specify a server, the default server referenced in the preferences file is used.
- If you do not specify a port number, port 12427 is used by default.

Command-Line Information

Parameter: `server`
Value: *`name`* or *`IP address:port number`*
Shell script example: `polyspace-ada -server 192.168.1.124:12400`

See Also

"Creating a Project"

Add to results repository

Specify whether verification results are stored in the Polyspace Metrics results repository, allowing Web-based reporting of results and code metrics.

Settings

Default: Off

☑ On
Verification results are stored in the Polyspace Metrics results database. This allows you to use the Polyspace Metrics Web interface to view verification results and code metrics.

☐ Off
Verification results are not added to the database.

Dependency

This option is available only for server verifications.

Command-Line Information

Parameter: `add-to-results-repository`
Shell script example: `polyspace-ada -server -add-to-results-repository`

See Also

"Creating a Project"

Number of processes for multiple CPU core systems

Specify maximum number of processors that can run simultaneously on multi-core system

Settings

Default: `4`

- Valid range is 1 to 128
- Reduces Polyspace verification time on multi-core computers.
- To disable parallel processing, set to 1.

Command-Line Information

Parameter: `max-processes`
Value: any integer between 1 and 128

Default: `4`
Shell script example: `polyspace-ada -max-processes 1`

Non-official options

Specify extra Polyspace options

Settings

Default: `None`

- Add expert option flags to verification. Place the option `-extra-flags` before each flag (parameter or value), for example:

  ```
  -extra-flags -param1 -extra-flags -param2 -extra-flags 10
  ```

  and

  ```
  -ada95-extra-flags -param1 -ada95-extra-flags -param2
  ```

- Polyspace supplies these flags, which depend on your verification requirements.
- Use `ada95-extra-flags` for Ada95 only.

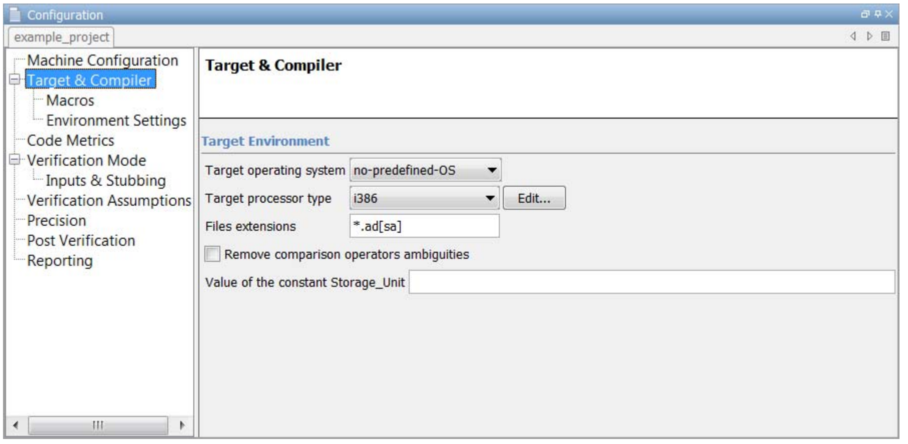
Command-Line Information

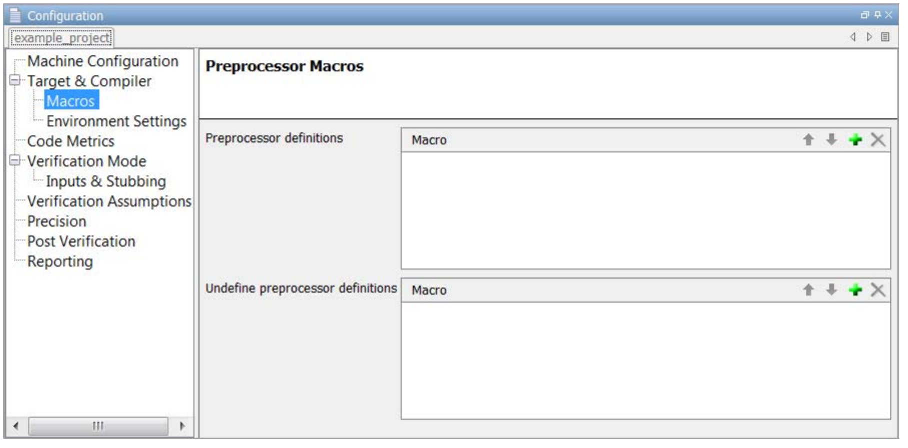
Parameter: `extra-flags | ada95-extra-flags`
Value: Supplied by Polyspace but depend on your requirements
Default: None

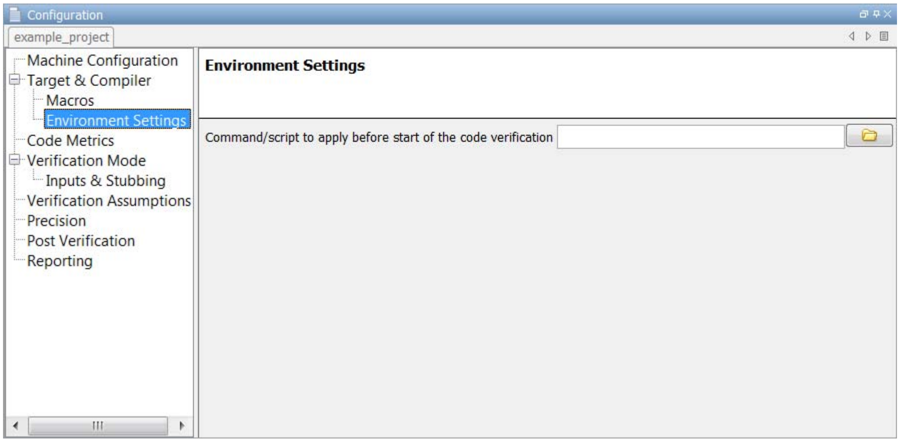
See Also

- "Function Stubbing"
- "Source Code Preparation"
- "Source Code Annotation"

Target & Compiler







In this section...

Target operating system

Specify operating system target for which there are implementation-specific declarations in the Ada Standard Libraries

Settings

Polyspace supplies only `gnat` include files, which you can find in the `ada` include folder within the installation folder. You can verify projects for other operating systems by using the corresponding include files (not supplied). For instance, to verify a `greenhills` project, specify files from the *`greenhills_include_folder`* in the `Include` folder for your project. See "Specifying Include Folders".

Default: `no-predefined-OS`

`no-predefined-OS`
: No operating system (with implementation-specific declarations in Ada Standard Libraries) specified

`gnat`
: GCC Ada95

`greenhills`
: Greenhills® Software real-time operating system (RTOS)

`rational`
: IBM® Rational® Apex compiler

`aonix`
: Aonix® compiler.

Command-Line Information

Parameter: `OS-target`
Type: string
Value: `no-predefined-OS | gnat | greenhills | rational | aonix`
Default: `no-predefined-OS`

Shell script examples:

```
polyspace-ada -OS-target gnat
```

```
polyspace-ada -OS-target greenhills
```

See Also

“Setting Up a Target”

Target processor type

Specify the target processor type.

Settings

Default: `i386`

`i386`
: Intel® 80386 (i386) processor

`sparc`
: Sun® Microsystems SPARC® processor

`m68k`
: Freescale™ ColdFire® m68k processor

`1750a`
: MIL-STD-1750A 16-bit instruction set architecture

`powerpc64bit`
: PowerPC® 64-bit instruction set architecture

`powerpc32bit`
: PowerPC 32–bit instruction set architecture

Command-Line Information

Parameter: `target`
Type: string
Value: `sparc | m68k | 1750a | powerpc64bit | powerpc32bit | i386`
Default: `i386`
Shell script example: `polyspace-ada -target m68k`

See Also

“Setting Up a Target”

Files extensions

Specify extensions used by package specification files in the `Include` folder of your project. Package specification files contain definitions and declarations referenced by your Ada body files. The software assumes that body files and the corresponding package specification files have the same names except for the extensions.

Settings

Default: `*.ad[sa]`

Command-Line Information

Parameter: `-extensions-for-spec-files`
Type: string
Value: any valid value
Default: `*.ad[sa]`

Remove comparison operators ambiguities

Specify whether to remove ambiguities regarding the visibility of relational operators (=, /=, <=, =>, >, and <).

In the following code:

```
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;
```

If you select the check box, the software does not generate errors. If you do not select the check box, the software generates errors:

- 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`

Settings

Default: Off

☑ On
  Remove ambiguities.

☐ Off
  Do not remove ambiguities. The type of operand determines whether the operator is visible.

Command-Line Information

Parameter: `base-type-directly-visible`

See Also

"Compilation Errors"

Value of the constant Storage_Unit

Specify a positive value for SYSTEM.Storage_Unit.

Settings

Default: `8`, except for target processor type `1750a` whose default is 16

- If you do not specify a value, the default in the SYSTEM package is used.
- The value required depends on the code that you write. For example, if the value for Storage_Unit is 8, the following code generates an error message `A overlaps B`:

  ```
  -- 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
  ```

  In this case, set the value of Storage_Unit to 32.

Command-Line Information

Parameter: `storage-unit`
Type: string
Value: any valid value
Default: `8`, except for target processor type `1750a` whose default is 16

See Also

"Compilation Errors"

Preprocessor definitions

Define compiler flags for compilation of preprocessor macros.

The software supports the following forms of preprocessor macros in your code:

```
# if expression
   ... code statements ...
# end if;
```

```
# if expression
   ... statements ...
# else
   ... statements ...
# end if;
```

```
# if expression
   ... statements ...
# elsif expression
   ... statements ...
# end if;
```

expression can be one of the following:

- *compiler_flag*
- *compiler_flag*="*value*"
- not (*expression*)
- *expression* and *expression*
- *expression* or *expression*
- *expression* and then *expression*
- *expression* or else *expression*

This option allows you to specify compiler flags that are present in *expression*.

Settings

No Default

- To define a compiler flag, in the Defined Preprocessor Macros dialog box, enter:

  compiler_flag="*value*"

  Then, click the **Adds this item to the list** button .
- Omitting the flag value is equivalent to specifying *compiler_flag*="True".
- Flag values are case-insensitive strings.
- To remove a compiler flag from the list, in the Defined Preprocessor Macros dialog box, select the compiler flag. Then, click the button .
- Consider the following example.

```
with Apex_Processes;
with Apex_Types;

package Lift_Load_Control_Process_P is

    procedure Start_S;

    use type Apex_Processes.Process_Name_Type;
    Process_Attr : constant Apex_Processes.Process_Attribute_Type :=
       (Name            => "Lift_Load_Control_Process_P   ",
        Entry_Point     => Apex_Types.System_Address_Type(Start_S'Address),
        Stack_Size      => 40000,
        Base_Priority   => 101,
#if VEROCODE
        Period          => Apex_Types.System_Time_Type(160000000),
#else
                Period          => Apex_Types.System_Time_Type(16000000),
#end if;
        Time_Capacity   => Apex_Types.System_Time_Type(10000000000),
        Deadline        => Apex_Processes.SOFT);

    Process_Id : aliased Apex_Processes.Process_Id_Type;
end Lift_Load_Control_Process_P;
```

If you specify `VEROCODE="True"`, then Polyspace does not verify code associated with the `#else` and `#end if` parts of the `if` statement. You will still see this code when you view results in the Results Manager perspective. However, as this code is not verified, there are no colored checks.

- As in the command line with compilers, you must specify only one flag for each `-D` option. However, you can use this option several times.

Command-Line Information

Parameter: `D`
Type: string
Shell script example:

```
polyspace-ada95 -D HAVE_MYLIB -D No_debug="Yes" -D USE_COM1="true" ...
```

See Also

- "Undefine Preprocessor definitions" on page 1-15
- "Setting Up a Target"

Undefine Preprocessor definitions

Nullify (undefine) macro compiler flags during compilation phase

Settings

No Default

- In the Undefined Preprocessor Macros dialog box, enter *`compiler_flag`*. Then click the **Adds this item to the list** button .
- Nullifying a macro compiler flag is equivalent to specifying in **Defined Preprocessor Macros** *`compiler_flag`*`="False"`.
- To remove a compiler flag from the list, in the Undefined Preprocessor Macros dialog box, select the compiler flag. Then, click the button .

- As in the command line with compilers, you must specify only one flag for each `-U` option. However, you can use this option several times.

Command-Line Information

Parameter: `U`
Type: string
Shell script example:

```
polyspace-ada95 -U HAVE_MYLIB -U USE_COM1 ...
```

See Also

- "Preprocessor definitions" on page 1-13
- "Setting Up a Target"

Command/script to apply before start of the code verification

Specify script file or command to run before the verification of each source file.

Settings

No Default

- Design the script or command to process the standard output from source code. For example, consider the following script `replace_keywords`:

  ```
  #!/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;
  }
  ```

  To replace the keyword `Volatile` by `Import`, run the following command on a Linux® machine:

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

- If you are running Polyspace software Version 5.1 (r2008a) or later on a Windows® system, you cannot use Cygwin™ shell scripts. Cygwin is no longer included with Polyspace software, so all files must be executable by Windows. To support scripting, the Polyspace installation includes Perl:

  ```
  Polyspace_Install\sys\perl\win32\bin\perl.exe
  ```

  To run the Perl script `replace_keywords` on a Windows machine, use the option `-pre-analysis-command` with the absolute path to the Perl script:

  ```
  Polyspace_Install\polyspace\bin\polyspace-ada.exe
  -pre-analysis-command
  Polyspace_Install\sys\perl\win32\bin\perl.exe
  <absolute_path>\replace_keywords
  ```

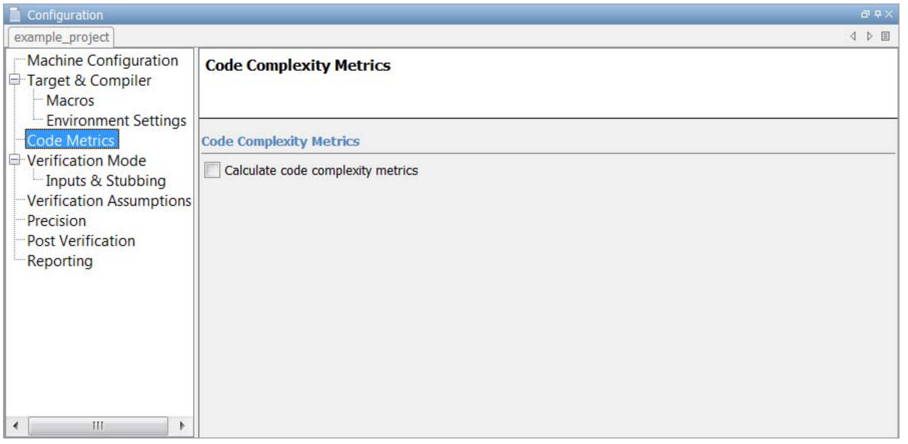
Command-Line Information

Parameter: `pre-analysis-command`
Type: string
Value: any valid script file name or command

See Also

"Setting Up a Target"

Code Metrics



Calculate code complexity metrics

Specify whether to calculate code metrics during verification

Settings

Default: Off

☑ On
  Calculate code complexity metrics.

☐ Off
  Do not calculate code complexity metrics.

Tips

- You can view code metrics data using the Polyspace Metrics Web interface, or by running a Software Quality Objectives report in the Polyspace verification environment.
- Project metrics include number of files, lines of code, packages, packages that appear in `with` statements, subprograms that appear in `with` statements, protected shared variables, and unprotected shared variables.

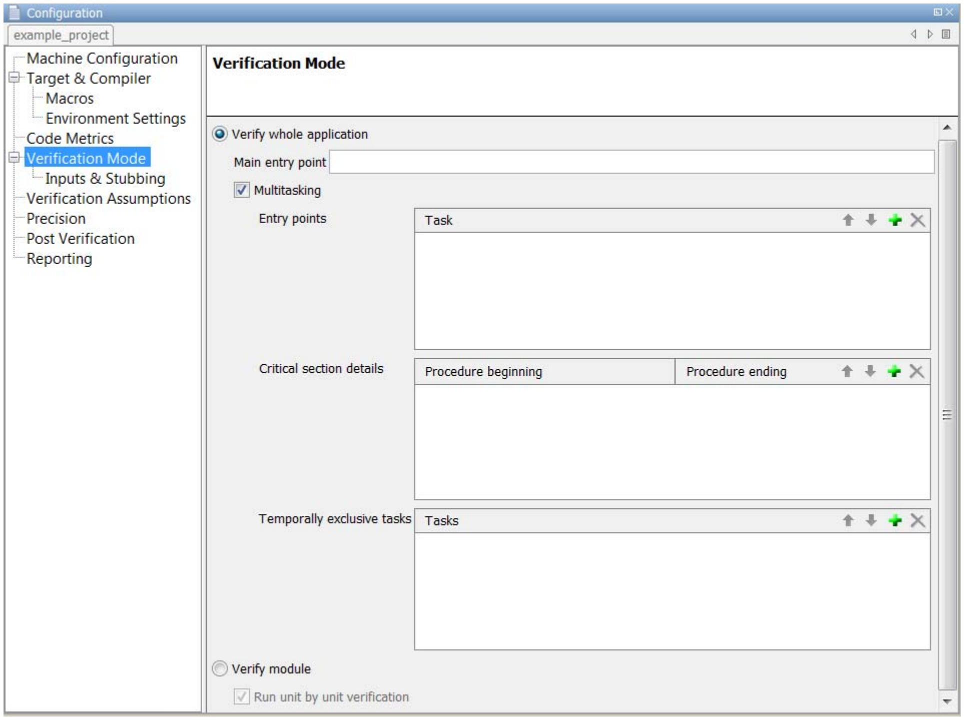
Command-Line Information

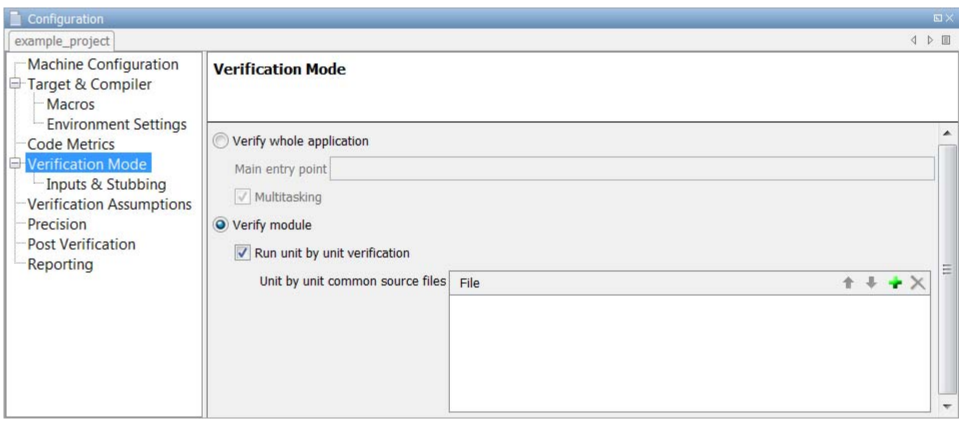
Parameter: `code-metrics`
Shell script example: `polyspace-ada -code-metrics`

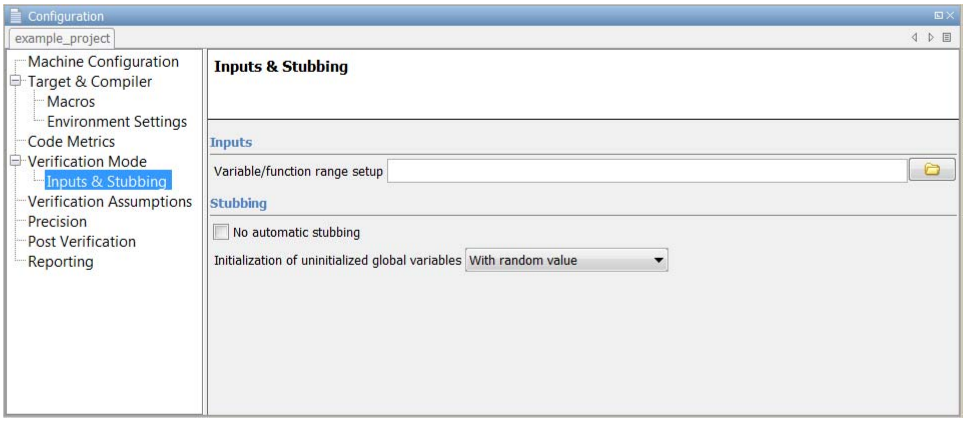
See Also

"Creating a Project"

Verification Mode







In this section...

Note Concurrency options are not compatible with the `-main-generator` option.

Main entry point

Specifies the name of the `main` subprogram

Settings

No Default

- Verify procedure after package elaboration, and before tasks (for a multi-task application or if `-entry-points` option is specified).
- Cannot be used with the `main-generator` option.

Command-Line Information

Parameter: `main`
Shell script example: `polyspace-ada -main mainpackage.init`

See Also

"Verifying an Application Without a Main"

Entry points

Specify tasks/entry points to verify

Settings

No Default

- The entry points must not take parameters. If the task entry points are functions with parameters, encapsulate the entry points in functions with no parameters and pass the parameters through global variables.
- If you declare tasks with the Ada keyword `task`, the software takes the tasks into account automatically.

Command-Line Information

Parameter: `entry-points`
Type: string
Value: any valid name, for example, `str1`, `str2`, `str3`
Shell script example: `polyspace-ada -entry-points pack1.proc1, pack2.proc2, pack3.proc3`

See Also

"Preparing Multitasking Code"

Critical section details

Specify critical sections in procedures

Settings

You can use critical sections to model, for example, protection of shared resources or interruption enabling and disabling.

Default No critical sections

- The options `critical-section-begin` and `critical-section-end` use lists to specify the procedures that begin and end critical sections.
- A list holds pairs of values. Each pair contains a procedure name along with the name of the critical section, for example, `"package1.proc1:cs1,package2.proc2:cs2"`. Do not put spaces in a list.

Command-Line Information

Parameter: `critical-section-begin`
Type: string
Value: any valid list, for example, `"package1.proc1:cs1,package2.proc2:cs2"`

Parameter: `critical-section-end`
Type: string
Value: any valid list, for example, `"proc3:cs1,proc4:cs2"`

Shell script example:

```
polyspace-ada -critical-section-begin "start_my_semaphore:cs" \
-critical-section-end "end_my_semaphore:cs"
```

See Also

"Preparing Multitasking Code"

Temporally exclusive tasks

Specify file that lists sets of temporally exclusive tasks (tasks that never execute at the same time)

Settings

Default No temporally exclusive tasks

Format of file:

- Use one line for each group of temporally exclusive tasks
- On each line, use spaces to separate tasks

For example, the file `exclusions` in the `sources` folder contains the following lines:

- `task1_group1 task2_group1`
- `task1_group2 task2_group2 task3_group2`

Command-Line Information

Parameter: `temporal-exclusions-file`
Type: string
Value: any valid file name, including path, for example,

```
sources/exclusions -entry-points task1_group1,task2_group1,task1_group2
```

Shell script example:

```
polyspace-ada  -temporal-exclusions-file sources/exclusions \
-entry-points task1_group1,task2_group1,task1_group2,\
task2_group2,task3_group2
```

See Also

"Preparing Multitasking Code"

Verify module

Specifies whether to automatically generate a `main` program.

The generated `main` program:

- Calls only procedures and functions that are specified in a package. Polyspace software assigns random, initialized values to all `in` and `in out` parameters for these procedures and functions.
- Assigns values to global variables that are specified in a package. If a global variable is initialized in a specification, then Polyspace software assigns this variable a random, initialized value. If a global variable is not initialized, Polyspace software assigns the global variable a random, possibly uninitialized, value (specifying the `init-stubbing-vars-random` or `init-stubbing-vars-zero-or-random` parameters has no effect).

  This behavior models the fact that the variable can be changed outside the package.
- Assigns values to global variables that are declared in a package body. If a global variable is initialized in a package body, then Polyspace software assigns this variable a random, initialized value. If a global variable is not initialized, Polyspace software leaves it uninitialized except when you specify the `init-stubbing-vars-random` or `init-stubbing-vars-zero-or-random` parameters. In this case, the variable is assigned a random, initialized value.

  This behavior models the fact that the variable can only be changed inside the package, but the functions of the package can be called several times.

If the source code has explicit tasks, then task bodies are verified like subprograms and `accept` statements are not executed. After verification, code associated with these `accept` statements is gray.

In this mode, the software does not compute information about sharing and protection of global variables that are accessed by explicit tasks.

```
package body Package1 is
  G : Integer := random;
```

```
  task body T is
  begin
    G := 5;
    x := 1/G; -- value of G? => 5
  end T;

  procedure Foo is
  begin
    G := 2;
    x := 1/G; -- value of G? => 2
  end Foo;
End
```

You cannot use this option with the `main` option.

Settings

Default: On for Polyspace Client™ for Ada; Off for Polyspace Server™ for Ada

☑ On

Create a procedure that calls every uninvoked procedure in the code.

Deactivates `main` option for Polyspace Server for Ada.

☐ Off

Selected by Polyspace Client for Ada if `main` option is activated

Command-Line Information

Parameter: `main-generator`

Shell script examples:

```
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)
```

See Also

"Initialization of uninitialized global variables" on page 1-32

Run unit by unit verification

Specify separate verification job for each source file in the project

Settings

Default: Off

☑ On

Each file is compiled, sent to the Polyspace Server, and verified individually. You can view results for the entire project or for individual units.

Unit by unit verification is available only for verification on a server.

☐ Off

No unit by unit verification.

Command-Line Information

Parameter: `unit-by-unit`
Shell script example: `polyspace-ada -unit-by-unit`

Unit by unit common source files

Specify list of files to include with each unit verification

Settings

No Default

- Compile files in the list once, and then link to each unit before verification.
- Stub functions not included in the list.

Command-Line Information

Parameter: `unit-by-unit-common-source`
Type: string
Value: any valid file name

Shell script example: `polyspace-ada -unit-by-unit-common-source c:/polyspace/function.adb`

Variable/function range setup

Specify a file that constrains the range of values for global variables, values returned by stubbed functions, `out` or `in/out` parameters of stubbed procedures, or input parameters of user subprograms called by the main generator during verification.

Settings

No Default

- Format for each line in file:

  `var_func_param min_val max_val` `<reinit|init|permanent>`

  - *`var_func_param`* — A variable name, the name of a function that returns a value, or a subprogram parameter name
  - *`min_val`*, *`max_val`* — Constants that specify minimum and maximum range values. Data type of these values can be character, enumerator, integer, or float. The integer or float values may be binary, octal, decimal, or hexadecimal.
  - `reinit` — Sets global variables to the specified range at the entry point for each subprogram called by the main generator, or the entry point for the user defined main subprogram.
  - `init` — Initializes subprogram input parameters to a specified range when the subprogram is called by the main generator.
  - `permanent` — Sets the return, `out`, or `in/out` parameters to the specified range of a stubbed subprogram each time the subprogram is called.
- You can:
  - Replace *`min_val`* and *`max_val`* by the words "`min`" or "`max`". In this case, the software uses the corresponding minimum and maximum value for the declared data subtype.
  - Use tab, comma, space, or semi-colon as column separators.
  - Apply data range specification to variables and subprograms declared within a package specification or body, or subprograms outside a package. For the latter case, use the subprogram name as package name.
- You cannot apply data range specification to:

- Local subprograms or task entries
- Constant qualified variables, record discriminants, variables of access type, or variables defined in a protected type or task type

Command-Line Information

Parameter: `data-range-specifications`
Type: string
Value: any valid file name
Shell script example: `polyspace-ada -data-range-specifications c:\Polyspace\drs.txt`

See Also

"Specifying Data Ranges for Variables, Functions, and Procedures (Contextual Verification)"

No automatic stubbing

Specify whether to stub a procedure or function without a body, to allow verification.

Settings

Default: Off

☑ On

A procedure or function that has no body (a function that you declare but do not define) stops verification. Use this option when you want to :

- Make sure that all code is supplied, for example, when verifying a large piece of code.
- Write stubs yourself, to increase the selectivity and speed of verification

☐ Off

Stub all procedures and functions automatically according to these rules:

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

- Implicit and explicit tasks are not stubbed.
- The `main` procedure is not stubbed.
- The generated stubs do not have side-effects on global variables. If a function affects global variables, then stub this function manually.

Dependency

This parameter cannot be used with **Initialization of uninitialized global variables**.

Command-Line Information

Parameter: `-no-automatic-stubbing`
Type: string
Value: `on | off`
Default: `off`
Shell script example: `polyspace-ada -no-automatic-stubbing`

See Also

"Stubbing"

Initialization of uninitialized global variables

Specify how uninitialized global variables are initialized.

Settings

Default: `No initialization`

The following setting descriptions apply only when the**Verify module** (`main-generator`) option is *not* selected. For information on how the settings apply when you select the `-main-generator` option, see "Verify module" on page 1-26.

`No initialization`
: Uninitialized global variables produce warnings or errors. No values assigned.

`With random value`
: Assign random values to uninitialized global variables.

`With zero or random value`
: Assign zero to uninitialized global variables if the type contains zero. Otherwise, assign random values.

Dependency

You cannot use this parameter with `-no-automatic-stubbing`.

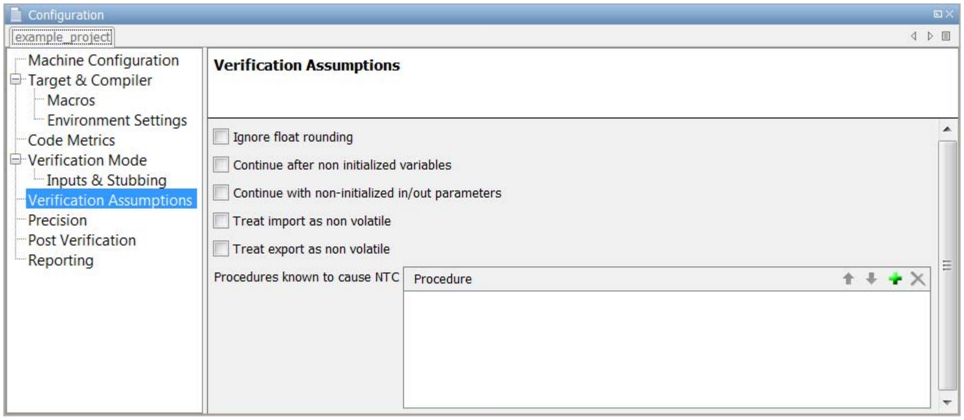
Command-Line Information

Parameter: `init-stubbing-vars-random | init-stubbing-vars-zero-or-random`
Shell script example: `polyspace-ada -init-stubbing-vars-random`

See Also

"Stubbing"

Verification Assumptions



In this section...

Ignore float rounding

Specify whether to round data type `float` according to the IEEE® 754 standard

Settings

Default: Off

On
    No rounding. Computation is exact.

☐ Off

Round data type `float` according to IEEE 754 standard: simple precision on 32-bit targets and double precision on targets that define `double` as 64-bit.

Command-Line Information

Parameter: `ignore-float-rounding`
Shell script example: `polyspace-ada -ignore-float-rounding`

See Also

"Float Rounding"

Continue after non initialized variables

Specify whether verification detects all non-initialized variables

Settings

Default: Off

☑ On

Detect all non-initialized variables (NIV). In the following code, the software detects all three NIVs in the first verification run.

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

You lose precision when using this option. Use this option only for the first verification run of a project.

☐ Off

Verification stops after the first red non-initialized variable (NIV).

Command-Line Information

Parameter: `continue-with-all-niv`

Continue with non-initialized in/out parameters

Specify whether to continue with verification if `in` and `out` parameters of a procedure are not initialized.

Settings

Default: Off

☑ On

In Ada, the `in` and `out` parameters of a procedure must be initialized. With this option, if the software detects one of the parameters as a red NIV, then subsequent code is *not* declared unreachable. The red error does not affect the verification.

```
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;  -- green with -continue-with-in-out-niv, gray otherwise
  end Main;
```

☐ Off

Verification stops if `in` and `out` parameters of a procedure are not initialized.

Command-Line Information

Parameter: `continue-with-in-out-niv`

Treat import as non volatile

Specify whether pragma import variable is volatile

Settings

Default: Off

☑ On
  Consider imported variable to be regular.

☐ Off
  Consider imported variable to be volatile.

Command-Line Information

Parameter:`import-are-not-volatile`
Shell script example: `polyspace-ada -import-are-not-volatile`

See Also

"Stubbing" and "Volatile Variables"

Treat export as non volatile

Specify whether pragma export variable is volatile

Settings

Default: Off

☑ On
  Consider exported variable to be regular.

☐ Off
  Consider exported variable to be volatile.

Command-Line Information

Parameter: `export-are-not-volatile`
Shell script example: `polyspace-ada -export-are-not-volatile`

See Also

"Stubbing" and "Volatile Variables"

Procedures known to cause NTC

Filter known non-terminating calls (NTC) to functions

Settings

Default: Empty. All NTCs appear as red errors.

- There can be functions that "never terminate". Some functions, such as tasks and threads, contain infinite loops by design, while functions that halt the program, such as `kill_task`, `exit`, or `Terminate_Thread` are often stubbed by an infinite loop. If your code has many of these functions or you want to present results to a third party, you may want to filter out certain types of NTCs when viewing results:
  1. Before a verification, specify the known NTCs.
  2. When you view results from the Results Manager perspective of the Polyspace verification environment, filter out checks that belong to the known-NTC category.

Command-Line Information

Parameter: `known-NTC`
Type: string
Value: any valid value, for example, "`kill_task,exit`",
Default: Empty

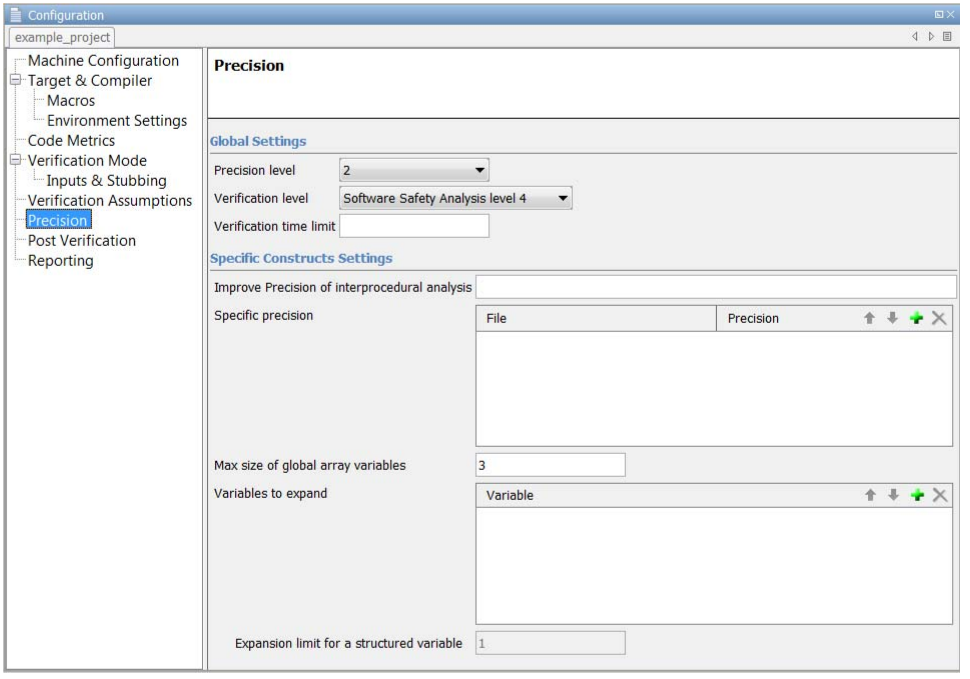
Shell script examples:

```
polyspace-ada  -known-NTC "kill_task,exit"
polyspace-ada  -known-NTC "Exit,Terminate_Thread"
```

See Also

"Preparing Multitasking Code"

Precision



In this section...
"Precision level" on page 1-41
"Verification level" on page 1-42
"Verification time limit" on page 1-43
"Improve precision of interprocedural analysis" on page 1-43
"Specific Precision" on page 1-45
"Max size of global array variables" on page 1-45
"Variables to expand" on page 1-46
"Expansion limit for a structured variable" on page 1-47

Precision level

Specify precision level used in verification

Settings

The precision level determines the algorithm used to model the program state space during verification. A higher precision level contributes to a higher selectivity rate, which makes reviewing results more efficient and isolating bugs easier. However, a higher precision level also results in a longer verification time.

Default: 2

0
: Static interval verification. MathWorks recommends that you begin verification at this level. Once you have addressed red errors and gray code from a verification at this level, you can relaunch the verification with a higher precision level.

1
: Complex polyhedron model of domain values

2
: Complex algorithms that closely model domain values (a mixed approach with integer lattices and complex polyhedrons)

3
: Suitable for code that is less than 1000 lines long. For such code, the selectivity can be very high, for example, 98%. This high selectivity results in a very long verification time, for example, an hour for 1000 lines of code.

Command-Line Information

Parameter: `O`
Value: `0 | 1 | 2 | 3`
Default: `2`
Shell script example: `polyspace-ada -O1`

See Also

"Reducing Orange Checks in Your Results"

Verification level

Specify the end point of verification

Settings

- Use with the `from` option.
- Specifies the point at which the verification ends.
- Allows you to have a higher selectivity to find more bugs within the code. A higher integration level contributes to a higher selectivity rate. However, a higher integration level also leads to a longer verification time

Default: `Software Safety Analysis level 4`

- `compile` or "`Source Compliance Checking`"
- `CDFA` or "`Control and Data Flow Analysis`" — Not available from the Polyspace verification environment
- `pass0` or "`Software Safety Analysis level 0`"
- `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 specify `other`, the verification continues until you halt it manually (through `kill-rte-kernel`) or the verification reaches `pass20`

Command-Line Information

Parameter: `to`
Type: string
Value: `compile | pass0 |pass1 | pass2 | pass3 | pass4 | other`
Default: `pass4`
`pass4`
Shell script examples:

```
polyspace-ada -to "Software Safety Analysis level 3"
polyspace-ada -to pass0
```

See Also

"Reducing Orange Checks in Your Results"

Verification time limit

Specifies a time limit for the verification (in hours).

Tips

- If the verification does not complete within the specified time, the verification fails.
- You can specify fractions of an hour in decimal form. For example:
  - `-timeout 5.75` – Five hours, 45 minutes.
  - `-timeout 3,5` – Three hours, 30 minutes.

Command-Line Information

Parameter: `timeout`
Shell script example: `polyspace-ada -timeout 5.75`

Improve precision of interprocedural analysis

Improve inter-procedural verification precision within a pass

Settings

The propagation of information within procedures happens earlier than usual with this option, which results in improved selectivity but a longer verification time.

Consider two cases, one where you set this option (`-path-sensitivity-delta`) to 1, and another where you do not set this option, that is, the option value is `0`.

- A level 1 verification with this option set provides results equivalent to a level 1 or 2 verification without the option.
- A level 1 verification with this option set can be many times longer than a cumulative level 1 and 2 verification without the option.

The same effect and results apply to a level 2 verification with this option set. A level 2 verification is equivalent to a level 3 or 4 verification without the option. Verification time also increases correspondingly.

Using this option results in the following:

- The highest selectivity is achieved in level 2, so you do not need to wait until level 4.
- Verification time can increase exponentially and result in an even longer verification time than that for the cumulative level 1, 2, 3, and 4 verification.

Use this option only for packages with fewer than 1000 lines of code.

Default: `0`

Command-Line Information

Parameter: `path-sensitivity-delta`
Value: any valid value
Default: `0`
Shell script example: `polyspace-ada -path-sensitivity-delta 1`

See Also

"Expansion of Sizes"

Specific Precision

Specify Ada packages to verify with a different precision from the general precision level (defined by the `O` option)

Settings

Default Verify all packages or modules with the same precision

To apply different precision levels to Ada packages in the Specific Precision dialog box:

- Under **Ada Package name**, specify the name of package or module.
- Under **Precision**, specify the required level: `0, 1, 2, or 3`

Command-Line Information

Parameter: `modules-precision`
Type: string
Value: *`package_name1`*`:O(`*`precision1`*`),` *`package_name2`*`:O(`*`precision2`*`),` *`package_name3`*`:O(`*`precision3`*`) ...` *`package_nameN`*`:O(`*`precisionN`*`)`
Shell script example: `polyspace-ada -O1 -modules-precision myMath:O2,myText:O1`

See Also

"Reducing Orange Checks in Your Results"

Max size of global array variables

Force Polyspace software to verify each cell of global variable arrays as a separate variable, if length is less than or equal to threshold value

Settings

Default: `3`

- Threshold value.
- Increasing the number of global variables to verify affects verification time.

- Affects only Global Data Dictionary results.

Command-Line Information

Parameter: `array-expansion-size`
Value: any valid value
Default: `3`
Shell script example: `polyspace-ada -O1 -array-expansion-size 8`

See Also

"Expansion of Sizes"

Variables to expand

Specify aggregate variables to split into independent variables during verification

Settings

Default Do not split aggregate variables into independent variables

- Use together with `-variable-expansion-depth` option.
- Affects the Global Data Dictionary results.

Command-Line Information

Parameter: `variables-to-expand`
Type: string
Value: list of aggregate variables, for example, `var1, var2,`

Shell script example:

```
polyspace-ada -variables-to-expand pkg.rec1,pkg2.recF    \
 -variable-expansion-depth 4
```

See Also

"Expansion of Sizes"

Expansion limit for a structured variable

Specify maximum depth for expansion of variables

Settings

In 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;
```

the effects of using different expansion depths are:

- `-variable-expansion-depth 1` — The concurrent access verification is done on `foo.myVar.FieldE` and `foo.myVar.Data`. If each access on `Data` is protected by critical section but `FieldE` is not protected, then `Data` is flagged as protected (a green entry in the Global Data Dictionary) and `FieldE` is flagged as not protected (an orange entry).
- `-variable-expansion-depth 2` — The verification is done on `foo.myVar.FieldE`, `foo.myVar.Data.FieldI` and `foo.myVar.Data.FieldII`. Each variable is flagged independently.

  `foo.myVar` is flagged as shared if any of its fields are shared. It is flagged as non-protected if any of its fields are not protected.

No Default:

- Use with `-variables-to-expand` option
- Increasing the number of global variables to verify extends verification time.

- This option affects only the Global Data Dictionary results.

Command-Line Information

Parameter: `variable-expansion-depth`
Value: any valid value, for example, `1`, `2`

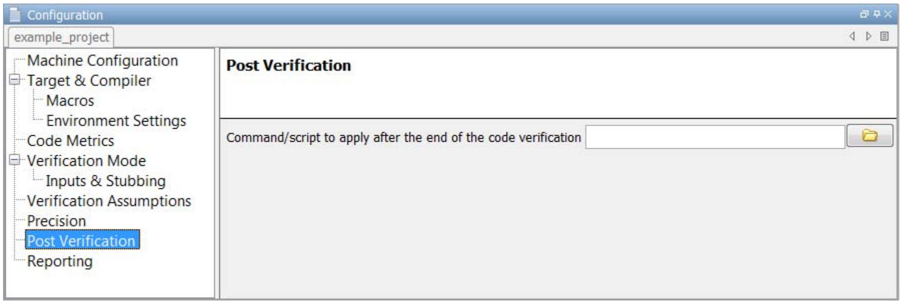
Shell script example:

```
polyspace-ada -variables-to-expand packcage_foo.myVar -variable-expansion-depth 1
```

See Also

"Expansion of Sizes"

Post Verification



Command/script to apply after the end of the code verification

Specify script file or command to run after the completion of verification.

Settings

No Default

- You execute the script or command in the verification results folder from either the client or server side, depending your configuration.

  To send an email to the client indicating that the verification is complete, run the following command:

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

  where `end_mail` is your Perl script.

- If you are running Polyspace software Version 5.1 (r2008a) or later on a Windows system, you cannot use Cygwin shell scripts. Cygwin is no longer included with Polyspace software, so all files must be executable by Windows. To support scripting, the Polyspace installation includes Perl:

  ```
  Polyspace_Install\sys\perl\win32\bin\perl.exe
  ```

  To run the Perl script `end_mail` on a Windows machine, use the option `-post-analysis-command` with the absolute path to the Perl script:

  ```
  Polyspace_Install\polyspace\bin\polyspace-ada.exe
  -post-analysis-command
  ```

```
Polyspace_Install\sys\perl\win32\bin\perl.exe
<absolute_path>\end_email
```

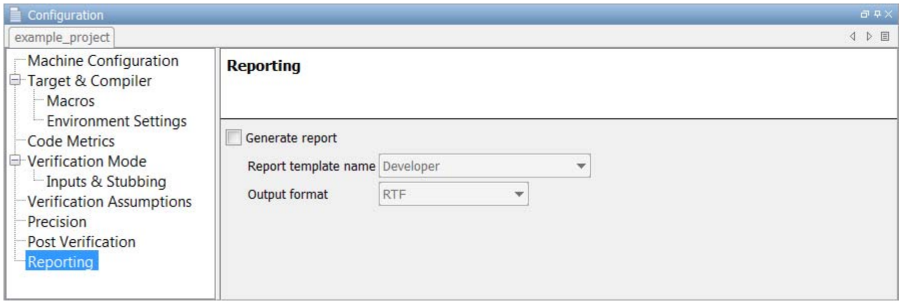
Command-Line Information

Parameter: `post-analysis-command`
Type: string
Value: any valid script file name or command

See Also

"Setting Up a Target"

Reporting



In this section...

Generate report

Specify whether to create verification report using report generation options

Settings

Default: Off

On
: Create report.

Off
: No report created.

Report template name

Specify template for generating verification report.

Settings

No Default

`Polyspace_Install\toolbox\psrptgen\templates\Developer.rpt`

Report templates provided with the software include:

- `Developer.rpt`
- `Developer_WithGreenChecks.rpt`
- `DeveloperReview.rpt`
- `Quality.rpt`
- `SoftwareQualityObjective.rpt`

Tip

Single report generated at the end of the verification process, before execution of any `-post-analysis-command`.

Command-Line Information

Parameter: `report-template`
Type: string
Value: any valid script file name
Shell script example: `polyspace-ada -report-template` *`filepath`*`\my_template`

Output format

Specify output format of report

Settings

Default: `RTF`

`RTF`
Generate an .rtf format report.

`HTML`
Generate an .html format report.

`PDF`
Generate a .pdf format report.

`Word`
: Generate a .doc format report.

  `Word` is not available on UNIX® platforms. RTF is used instead.

`XML`
: Generate and .xml format report.

Command-Line Information

Parameter: `report-output-format`
Type: string
Value: `RTF | HTML | PDF | Word | XML`
Default: `RTF`

Shell script example:

```
polyspace-ada -report-template my_template -report-output-format pdf
```

Batch Options

In this section...

-author *name*

Specify author of verification. See also "Creating a Project" in the *Polyspace Products for Ada User's Guide*.

Default: user ID

Example Shell Script Entry:

```
polyspace-ada -author "A. Tester"
```

-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 you do not specify the option server, the default server referenced in the Polyspace-Launcher.prf configuration file is used as the server.

When you use the server option in the batch launching command, you must specify the name or IP address and a port number. If the port number does not exist, the 12427 value is used as the default.

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
```

produces an output like the following:

```
Polyspace r2011b
```

```
Copyright (c) 1999-2011 The Mathworks, Inc.
```

-sources-list-file *file_name*

This option is available only in batch mode.

`file_name` specifies:

- The name of one file
- The absolute or relative path of the file

Example Shell Script Entry for -sources-list-file:

```
polyspace-ada -sources-list-file "C:\Analysis\files.txt"

polyspace-ada -sources-list-file "files.txt"
```

-from

Specify starting point of verification

Settings

- Use with the `to` option.
- Use only on a verification that you have run partially, to specify the restart point of the verification. For example, if you have previously run a verification to `Software Safety Analysis level 1` (`pass1`), you can restart the verification at this point. You do not have to run the verification from `scratch`.
- Use only for client-based verification (server-based verification always starts from `scratch`).
- Use only for restarting a verification launched with the option `keep-all-files` (unless you restart from `scratch`).
- You cannot use this option if you modify the source code between verifications.

Command-Line Information

Parameter: `from`
Type: string
Value: `scratch | compile | pass0 |pass1 | pass2 | pass3 | pass4 | other`
Default: `scratch`
Shell script example: `polyspace-ada -from pass0`

See Also

"Reducing Orange Checks in Your Results"

-keep-all-files

Specify whether to retain all intermediate results and associated working files.

Settings

Default: Off

On
: Retain all intermediate results and associated working files. You can restart a verification from the end of any complete pass if the source code remains unchanged.

Off
: Erase all intermediate results and associated working files. If you want to restart a verification, do so from the beginning.

Tips

- When you select this option, you can restart Polyspace verification from the end of any complete pass (provided the source code is unchanged). If you do not use this option, you must restart the verification from the beginning.
- This option is applicable only to client verifications. Intermediate results are always removed before results are downloaded from the Polyspace server.

- To cleanup intermediate files at a later time, you can select **Tools > Clean Results** in the Launcher. This option deletes the preliminary result files from the results folder.

Command-Line Information

Parameter: `keep-all-files`
Shell script example: `polyspace-ada -keep-all-files`

See Also

"Creating a Project"

-report-output-name

Specify name of verification report file

Settings

Default: *`Prog_TemplateName.Format`* where:

- *`Prog`* is the argument of the `prog` option
- *`TemplateName`* is the name of the report template specified by the `report-template` option
- *`Format`* is the file extension for the format specified by the `report-output-format` option.

Command-Line Information

Parameter: `report-output-name`
Type: string
Value: any valid value
Default: *`Prog_TemplateName.Format`*

Shell script example:

```
polyspace-ada -report-template my_template -report-output-name Airbag_V3.rtf
```

`-less-range-information`

Limit amount of range information displayed in verification results

Settings

Default: Off

On

Provide range information on assignments, but not read operations.

Consider the following example:

```
x := y + z;
```

If you enable this option, you see range information only when you place your cursor over `x`.

As computing range information for read operations may take a long time, selecting this option can reduce verification time significantly.

Off

Range information available on assignments and read operations.

For the same example, you see range information when you place your cursor over `x`, `y`, or `z`.

Command-Line Information

Parameter: `less-range-information`
Shell script examples:

```
polyspace-ada -less-range-information ...
```

See Also

"Using Range Information in Results Manager Perspective"

`-strict` or `-permissive`

Select strict or permissive verification mode

Settings

Default: `-permissive`

`-strict`
: Selects `no-automatic-stubbing` option

`-permissive`
: Selects `continue-with-in-out-niv` option

Command-Line Information

Parameter: `-strict or -permissive`
Default: `-permissive`

-import-comments

Use option to automatically import coding rule and run-time check comments and justifications from specified folder at the end of verification.

Command-Line Information

Default:

Disabled

Shell script examples: :

```
polyspace-ada -version 1.3 -import-comments
C:\PolyspaceResults\1.2
```

-tmp-dir-in-results-dir

If you specify the new option `-tmp-dir-in-results-dir`, Polyspace does not use the standard `/tmp` or `C:\Temp` folder to store temporary files. Instead, Polyspace uses a subfolder of the results folder. If the results folder is mounted on a network drive, this action may affect processing speed . Use this option only when the temporary folder partition is not large enough and troubleshooting is required.

Default:

Disabled

Example Shell Script Entry:

```
polyspace-c -tmp-dir-in-results-dir -results-dir
C:\Polyspace\Results
```

2

Check Descriptions

Colored Source Code for Ada

In this section...

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;
  begin
   if (Random_Bool) then
    Vpixel.X := 1;
    AddPixelValue(Vpixel); -- NTC Error: because of NIV error
 in call
   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.

```
package NIVIO is
 procedure MAIN;
 function Random_Boolean return Boolean;
end NIVIO;

package body NIVIO is

 Y : Integer := 3;
 procedure Niv_Not_Dangerous(X : in out integer) is
  begin
   X := 2;
   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: very significant
29    end if ;
30    if (Random_Boolean) then
31    Niv_Not_dangerous(X); -- NIV ERROR: not significant
32    End if ;
33   end MAIN;
34
35  end NIVIO;
```

Explanation. In the previous example, as shown at line 28, Polyspace highlights a non-initialized variable that could be a significant error. In the *Niv_Not_Dangerous* procedure, Polyspace highlights the non-initialized variable at line 30, even though the error is not as significant. To be more permissive with reference to the standard, the **-continue-with-in-out-niv** option permits continuation of the verification for the rest of the sources even if a red error remains e 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 • Permanent random value	• No NIV check • Permanent random value	• same behavior as -automatic-stubbing • in/out and out variables are written within their entire type range

In this case, a permanent random value means that the variable is always equivalent to the full range of its type. It is almost equivalent to a volatile variable except for the 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 • Permanent random value	• No NIV check • Permanent random value

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

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 • Permanent random value	• No NIV check • Permanent random value

In this case, a Permanent Random Variable is exactly equivalent to a pvolatile 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 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

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

package body OVFL is

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

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

   if (Bool_Random) then
    I := 2**30;
    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.

Attributes Check: COR

Polyspace encourages the user to investigate the attributes *SUCC*, *PRED*, *VALUE* and *SIZE* further through 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, gray, 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
   X : Integer;
  begin
   X := Integer'Value (S); -- COR Warning: Value parameter might not be in range integer
   return X;
  end INT_VALUE;

  procedure MAIN is
   E : PSTCOLORS := GREEN;
   F : PSTCOLORS;
   ADCVAL : ADCFUZZY := ADCFUZZY'First;
   StrVal : STR_ENUM;
   X : Integer;
  begin
   if (Bool_Random) then
    F := PSTCOLORS'PRED(E); -- COR Verified: Pred attribute is not used on the first element of pstcolors
    E := PSTCOLORS'SUCC(E); -- COR ERROR: Succ attribute is used on the last element of pstcolors
   end if;
   if (Bool_Random) then
    ADCVAL := ADCFUZZY'PRED(ADCVAL); -- COR ERROR: Pred attribute is used on the first element of adcfuzzy
   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"); --X info: -2**31<=[expr]<=2**31-1
39   end MAIN;
40  end CORS;
41
```

Explanation

At line 36 and 37, the COR warning (orange) prompts you to check the *VALUE* attribute.

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

```

with Ada.Text_Io; use Ada.Text_Io;

package SIZE is
 PROCEDURE Main;
end SIZE;

PACKAGE BODY SIZE IS

  TYPE unSTab is array (Integer range <>) of Integer;

  PROCEDURE MAIN is
   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
22    end if;
23    if (Bool_Random) then
24     Arr1 := (1,2,3,4,5,6,7,8,9); -- COR ERROR: Not enough
elements in array, must have 10
25    end if ;
26    if (Bool_Random) then
27     Str1 := "abcdef";
28     Str2 := "ghijkl";
29     Str3 := Str1 & Str2; -- COR Warning: Length might not be
compatible with 1 .. 6
30     Put(Str3);
31     if Bool_Random then
32      DName.DISPLAY_NAME (DNAME.NAME_TYPE(STR_CST));
-- COR ERROR: String Length is not correct, must be 4
33     end if;
34    end if ;
35   end MAIN;
36
37  end CORL;
38
39  package DName is
40   type Name_Type is array (1 .. 4) of Character;
41   PROCEDURE DISPLAY_NAME (Str : Name_Type);
42  end DName;
43
```

Explanation

At lines 21 and 24, Polyspace gives the exact value required for the two arrays to match. At line 29, Polyspace prompts you, through an orange check, to investigate the compatibility of concatenated arrays.

In addition, at line 32, the required string length is given even if the string length depends on another package.

DIGITS Value Check: COR

Checks the length of *DIGITS* constructions.

Ada Example

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

Explanation

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

DELTA Value Length Check: COR

Checks the length of *DELTA* constructions.

Ada Example

```
package FIXED is
 procedure MAIN;
 procedure FAILED(STR : STRING);
 function Random return Boolean;
end FIXED;

package body FIXED is

  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

```

package SRANGE is
 procedure Main;
 function IsNatural return Boolean;

 SUBTYPE INT IS INTEGER RANGE 1 .. 3;
 TYPE INF_ARRAY IS ARRAY(INT RANGE <>, INT RANGE <>) OF INTEGER;
 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;
  end if;
  if IsNatural then
   declare
    TYPE NEW_ARRAY IS ARRAY (NATURAL RANGE <>) OF INTEGER;
    subtype Sub_Sensor is New_Array (Integer RANGE -1 .. 5);
-- COR Error: Static range is not in range 0 .. 16#7FFF_FFFF#
   begin
    null;
   end;
  end if ;
  if IsNatural then
   VALUE_RANGE; -- NTC Error: propagation of the COR error
in VALUE_RANGE
  else
   Digval := 11; -- COR Error: Value is not in range of 0..10
  end if;
 END Main;
end SRANGE;
```

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 green definition, others to a red definition. The result is an orange warning at line 15.

At lines 19, 38, 46 and 54 Polyspace displays errors for out of range values.

Discriminant Check: COR

Checks the usage of a discriminant in a record declaration.

Ada Example

```

package DISC is
 PROCEDURE MAIN;

 TYPE T_Record(A: Integer) is record -- COR Verified: Value is in range of 1 .. 16#7FFF_FFFF#
  Sa: String(1..A);
 END RECORD;
end DISC;

package body DISC is

 PROCEDURE MAIN is
 begin
  declare
   T_STRING6 : T_RECORD(6) := (6, "abcdef"); --COR Verified: Discriminant is compatible
   T_StringOther : T_RECORD(6); -- COR Verified: Discriminant is compatible
   T_STRING5 : T_RECORD(5) := (5, "abcde"); -- COR Verified: Discriminant is compatible
  begin
   T_StringOther := T_STRING6; -- COR Verified: Discriminant is compatible
   T_string5 := T_Record(T_STRING6); -- COR ERROR: Discriminant is not compatible
  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

```
package COMP is

 PROCEDURE MAIN;
 SUBTYPE DINT IS INTEGER RANGE 0..1;
 TYPE COMP_RECORD ( D : DINT := 0) is record
  X : INTEGER;
  CASE D IS
   WHEN 0 => ZERO : BOOLEAN;
   WHEN 1 => UN : INTEGER;
   END CASE;
  END RECORD;

 end COMP;

 package body COMP is

  PROCEDURE MAIN is
   CZERO : COMP_RECORD(0);
  BEGIN
   CZERO.X := 0;
   CZERO.ZERO := FALSE; -- COR Verified: zero is a component of the variable
   CZERO.UN := CZERO.X; -- COR ERROR: un is not a component of the variable
  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

```
package DIMDEF is
 PROCEDURE MAIN;
 FUNCTION Random RETURN boolean;
end DIMDEF;

package body DIMDEF is

 SUBTYPE ST IS INTEGER RANGE 4 .. 8;
 TYPE BASE IS ARRAY(ST RANGE <>, ST RANGE <>) OF INTEGER;
  SUBTYPE TBASE IS BASE(5 .. 7, 5 .. 7);

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

  PROCEDURE MAIN IS
   NEWARRAY : TBASE;
  BEGIN
   IF RANDOM THEN
    NEWARRAY := (7 .. IDENT_INT(9) => (5 .. 7 => 4)); -- COR Error: Dimension is not compatible with definition
   END IF;
   IF Random THEN
    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
```

```
24  BEGIN
25   RETURN VAL;
26   END IDENT_INT;
27
28   PROCEDURE AGGDEF_INIT is -- AGGREGATE INITIALISATION
29    OBJ3 : REC3;
30   BEGIN
31    if random then
32     OBJ3 :=
33      ((DISC => IDENT_INT(7)), (DISC => IDENT_INT(7))); -- COR ERROR: Aggregate is not compatible with definition
34    end if;
35    IF OBJ3 = ((DISC => 7), (DISC => 7)) then -- COR ERROR: Aggregate is not compatible with definition
36     COMMENT ("PREVENTING DEAD VARIABLE OPTIMIZATION");
37    END IF;
38   END AGGDEF_INIT;
39
40   PROCEDURE MAIN IS
41   BEGIN
42    AGGDEF_INIT; -- NTC ERROR: propagation of COR ERROR
43   END MAIN;
44  end AGGDEF;
```

Explanation

At 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

```
package AGGLEN is
 PROCEDURE MAIN;
 PROCEDURE COMMENT(A: STRING);
end AGGLEN;
```

```
package body AGGLEN is

 SUBTYPE SLENGTH IS INTEGER RANGE 1..5;
 TYPE SL_ARR IS ARRAY (SLENGTH RANGE <>) OF INTEGER;

  F1_CONS : INTEGER := 2;
  FUNCTION FUNC1 RETURN INTEGER IS
  BEGIN
   F1_CONS := F1_CONS - 1;
   RETURN F1_CONS;
  END FUNC1;


  TYPE CONSR (DISC : INTEGER := 1) IS
   RECORD
    FIELD1 : SL_ARR (FUNC1 .. DISC); -- FUNC1 EVALUATED.
   END RECORD;

  PROCEDURE MAIN IS

  BEGIN
   DECLARE
    TYPE ACC_CONSR IS ACCESS CONSR;
    X : ACC_CONSR;
   BEGIN
    X := NEW CONSR;
    BEGIN
     IF X.ALL /= (3, (5 => 1)) THEN -- COR ERROR: Illegal Length for array aggregate
      COMMENT ("IRRELEVANT");
     END IF;
    END;
   END;
  END MAIN;

 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

```

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
29      1 => (CALC(H,2) .. CALC(I,5) => -4),
30      2 => (CALC(H,3) .. CALC(I,6) => -5),
31      3 => (CALC(H,2) .. CALC(I,5) => -3) );
32    END IF;
33   END MAIN;
34
35  end SUBDIM;
```

Explanation

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

Sub-aggregates must be singular.

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
```

```
17    return 'A';
18   end;
19
20   procedure MAIN is
21    Var : ALL_Char;
22   BEGIN
23    IF RANDOM THEN
24     Var(1) := Init; --NTC ERROR: propagation of the COR err
25    ELSE
26     Var(Integer) := ""; -- COR ERROR: the 'null' string
literal is not allowed here
27    END IF;
28   END MAIN;
29  END CHAR;
```

Explanation

At line 15, Polyspace 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

```

package CORACCESS is
 procedure main;
 function Brand return Boolean;
end CORACCESS;

package body CORACCESS is
 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
16     null;
17    end if;
18   end main;
19  end CORNULL;
20
```

Explanation

At line 14 and line 15, Polyspace checks the null value of *ptr* pointer. As Polyspace does not 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;
```

```
16
17  end TAG;
18
19
20  package body TAG is
21
22   procedure Main is
23    Y : DTag_Type := DTag_Type'(C1 => 1, C2 => 1.1);
24    Z : DTag_Type := DTag_Type'(C1 => 2, C2 => 2.2);
25
26    W : Tag_Type'Class := Z;  -- W can represent any object
27             -- in the hierarchy rooted at Tag_Type
28   begin
29    Y := DTag_Type(W); -- COR Warning: Tag might be correct
30    null;
31   end Main;
32
33  end TAG;
```

Explanation

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

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

Polyspace is not precise on tagged types and currently always flags 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
```

```
21     -- Flip_Flop is randomly be True or False
22     Flip_Flop := bool_random;
23
24     -- Flip_Or_Val is always True
25     Flip_Or_Val := Flip_Flop or Val_Constant;
26     pragma assert(flip_flop=True or flip_flop=False); --
User assertion is verified
27     pragma assert(Flip_Or_Val=False); -- ASRT ERROR: User
assertion fails
28    end if;
29    if (Bool_Random) then
30     ten_positive := Ten_random;
31     pragma assert(ten_positive > 5); -- ASRT Warning: User
assertion may fail
32     pragma assert(ten_positive > 5); -- User assertion
is verified
33     pragma assert(ten_Positive <= 5); -- ASRT ERROR:
Failure User Assert
34    end if;
35
36   end MAIN;
37
38  end ASRT; -- End Package
```

Explanation

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

- 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 Termination of Call or Loop: NTC and NTL

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:
	• while(1=1)loop function_call; end loop; // informative NTL
	• while(x >=0) loop x := x+1; end loop; // with x as an unsigned int could reveal a bug, or not (an unsigned is always positive)
	• for I in 0 .. 10 loop my_array(i) = 10; end loop; // with "my_array is integer in 0..9" this red NTL reveals a bug in the array access, flagged in orange

NTC	Your function called "test" calls f;. And “f;” is flagged as a red NTC. Why? There could be five distinct explanations for this NTC: • “f” contains a red error; • “f” contains an NTL ; • “f” contains an NTC; • “f” contains an orange which is context dependant : it is either red or green: for this call, it makes the function crash. **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.

If you set the Review Level slider to 0, the software does not display NTC checks on the **Results Explorer** or **Results Summary** tab.

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
```

```
13    while (X >= 0) loop -- NTL ERROR: Loop never terminate
14     if ( Y /= X) then
15      Y := 1 / (Y-X);
16     end if;
17    end loop;
18   end FOO;
19
20   procedure MAIN is
21   begin
22    if (Random_Boolean) then
23     FOO(0); --NTC ERROR: Division by zero  in NTC.FOO (ZDV)
24    end if ;
25    if (Random_Boolean) then
26     FOO(2); --NTC ERROR: Non Termination Loop in NTC.FOO (NTL)
27    end if;
28   end MAIN;
29  end NTC;
```

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

- 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 the 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.

```
package NTC_entry is

 TASK TYPE MyTask IS
  ENTRY START;
  ENTRY V842;
  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. When you analyze results in the Results Manager perspective, you can filter out functions that are designed to be non-terminating.

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.

If you set the Review Level slider to 0, the software does not display NTL checks on the **Results Explorer** or **Results Summary** tab.

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 opposed to other red errors, Polyspace does not continue with the verification in the current branch. 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 your code has mathematical functions that Polyspace does not support and variables derived from these mathematical functions are summed, the verification produces unproven checks arising from overflows.

You encounter this issue when Polyspace stubs all mathematical functions automatically, which happens if the function declarations for your compiler are slightly different from the declarations assumed by Polyspace. In following example, you resolve the issue by providing an extra package that matches your mathematical functions to Polyspace functions. The extra package has no impact on the original source code, that is, the software does not modify your code.

The original source 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); --range [-1.0 .. 1.0]
   X := Sin(12.3); --range [-1.0 .. 1.0]
   X ::= Sqrt(-1.5); --is red: NTC Error
  end;
end Main;
```

The 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;
```

Create the following package for more precise modeling of your 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);
```

Note Due to a lack of precision in some areas, Polyspace does not always generate a red NTC check for mathematical functions even when a problem exists. It is important to consider each call to a mathematical function as an unproven check that could lead to a run-time 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;
```

```
32    -- Use of I which is not initialized
33    if (Bool_Random) then
34     Res_State := Intermediate_State(I); -- NIV ERROR
35     if (Res_State = Intermediate) then -- UNR code because
of NIV error
36      Res_State := EndState;
37     end if;
38    end if;
39   else
40    -- UNR code
41    I := 1;
42    Res_State := Intermediate_State(I);
43   end if;
44   return I; -- NIV ERROR: because of UNR code
45  end UNR_I;
46
47  procedure MAIN is
48   I : Integer;
49  begin
50   I := UNR_I; -- NTC ERROR because of propagation
51  end MAIN;
52
53 end UNR;
54
55
56
```

Explanation

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

Approximations Used During Verification

Why Polyspace Verification Uses Approximations

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 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 required.

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.

Limitations of Polyspace Verification

Code verification has certain limitations. The *Polyspace Limitations* document describes known limitations of the code verification process.

You can access the *Polyspace Limitations* document in the installed PDF folder:

`Polyspace_Common\R2012b\help\pdf\polyspace_limitations.pdf`

Note By default, the `Polyspace_Common` folder is installed in the following location:

- **Windows systems** – `C:\Polyspace\Polyspace_Common`
- **UNIX systems** – `/usr/local/Polyspace/Polyspace_Common`
