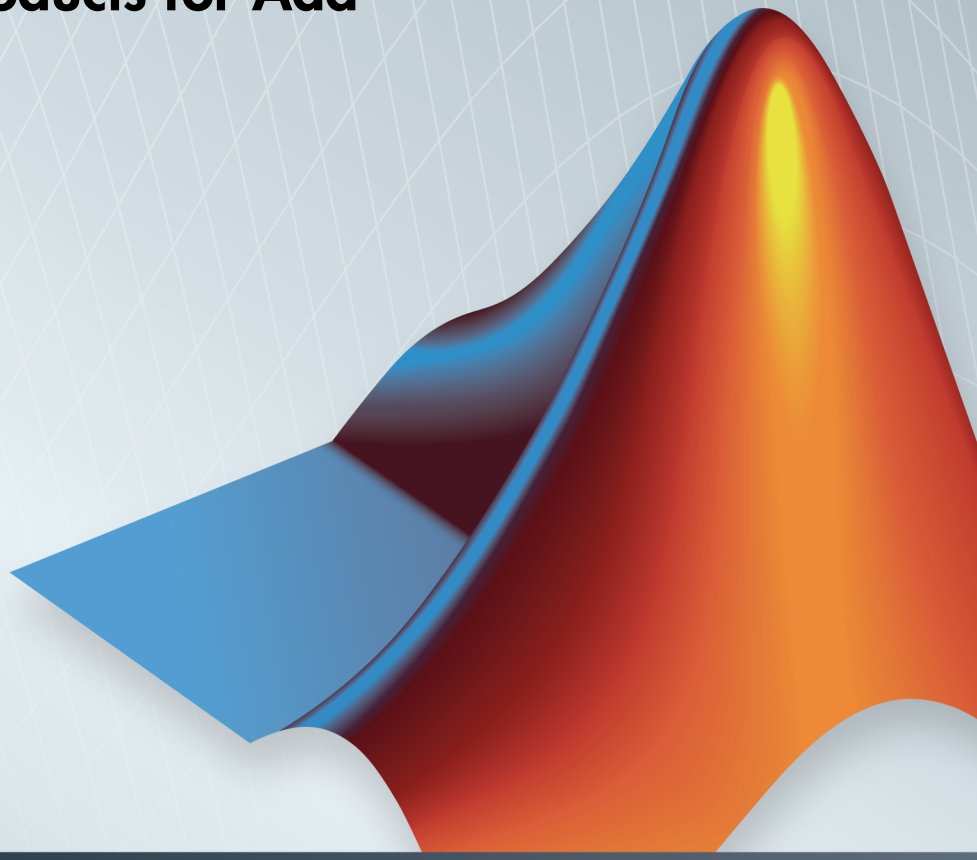
Polyspace® Products for Ada

Reference

R2014**b**



MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

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)
September 2013	Online Only	Revised for Version 6.6 (Release 2013b)
March 2014	Online Only	Revised for Version 6.7 (Release 2014a)
October 2014	Online Only	Revised for Version 6.8 (Release 2014b)

Contents

1 Option Descriptions

2 Check Descriptions

3 Approximations Used During Verification

Option Descriptions

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” on page 1-4

Add to results repository

Specify upload of analysis results to the Polyspace Metrics results repository, allowing Web-based reporting of results and code metrics.

Settings

Default: Off

☑ On

Analysis results are stored in the Polyspace Metrics results repository. This allows you to use a Web browser to view results and code metrics.

☐ Off

Analysis results are stored locally.

Dependency

This option is available only for remote verifications.

Command-Line Information

Parameter: `-add-to-results-repository`
Default: off
Example: `polyspace-ada -server IPaddress -add-to-results-repository`

See Also

“Creating a Project” | “Send to Polyspace Server”

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 "Add Source Files and 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: Valid file extensions
Default: `*.ad[sa]`

Remove ambiguities in comparison operators

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`
Value: Integer
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

Default: None

- 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, its operations are not assigned a color.
- 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

- “Disable preprocessor definitions” on page 1-15

- “Setting Up a Target”

Disable preprocessor definitions

Nullify (undefine) macro compiler flags during compilation phase

Settings

Default: None

- 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 1-12
- “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

Default: None

- 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 not 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: Script file name or command

See Also

"Setting Up a Target"

Include folders

View the include folders used for verification.

- In the Project Manager perspective, to add include folders, on the **Project Browser**, right-click your project. Select **Add Source**.
- In the Results Manager perspective, to view the include folders you used, select **Window > Show/Hide View > Settings**. Under the node **Environment Settings**, you see the folders listed under **Include folders**.

Settings

This is a read-only option available only from the Results Manager perspective. Unlike other options, in the Project Manager perspective, you do not specify include folders on the **Configuration** pane. Instead, you add your include folders on the **Project Browser** pane.

Command-Line Information

Parameter: `-I`
Value: Folder name
Example: `polyspace-ada -I /com1/inc -I /com1/sys/inc`

Verify whole application

Specify that Polyspace verification must use a procedure you designate as the `main` subprogram.

Settings

Default: Off

On

Polyspace uses the procedure you designate as the `main` subprogram. Enter the name of the procedure in the **Main entry point** field.

Off

Polyspace generates a `main` procedure to wrap uncalled procedures in the module you are verifying.

Command-Line Information

The command-line option `-main` combines the two user interface options **Verify whole application** and **Main entry point**.
Parameter: `-main`
Value: Procedure name
Example: `polyspace-ada -sources filename -main mainpackage.init`

See Also

“Main entry point” | “Entry points”

Related Examples

- “Specify Analysis Options”
- “Automatically Generating a Main”

More About

- “Main Generator Overview”

Main entry point

Specify the procedure that Polyspace verification must use as the `main` subprogram. This procedure is verified after package elaboration and before other tasks in case of multitasking code.

Settings

Enter procedure name.

Dependencies

This option is enabled only if you select the option **Verify whole application**.

Command-Line Information

The command-line option `-main` combines the two user interface options **Verify whole application** and **Main entry point**.

Parameter: `-main`
Value: Procedure name
Example: `polyspace-ada -sources` *`filename`* `-main mainpackage.init`

See Also

"Verify whole application" | "Entry points"

Related Examples

- "Specify Analysis Options"
- "Automatically Generating a Main"

More About

- "Main Generator Overview"

Multitasking

Specify whether the code is intended for a multitasking application.

Settings

Default: Off

 On

The code is intended for a multitasking application.

 Off

The code is not intended for a multitasking application. Polyspace verifies only those functions that are called by the “Main entry point”.

Command-Line Information

There is no command-line option to solely turn on multitasking verification. However, using the option `-entry-points` turns on multitasking verification.

See Also

“Entry points” | “Critical section details” | “Temporally exclusive tasks”

Related Examples

- “Specify Analysis Options”
- “Modelling Synchronous Tasks”
- “Interruptions and Asynchronous Events/Tasks”

More About

- “Priorities”
- “Polyspace Software Assumptions”

Entry points

For multitasking code, specify the procedures that Polyspace must consider as entry points.

Settings

Default: None

Click  to add a field. Enter the procedure name.

Dependencies

This option is enabled only if you select the **Multitasking** box.

Tips

- The procedures designated as entry points must not have input parameters. If they have input parameters, encapsulate them in procedures without parameters and pass the parameters through global variables.
- You can also specify entry points in your code with the Ada keyword `task`. Specifying entry points using this keyword overrides entry point specification through the **Configuration** pane.

Command-Line Information

Parameter: `-entry-points`
Value: Name of task
Shell script example: `polyspace-ada -sources` *`filename`* `-entry-points pack1.proc1, pack2.proc2, pack3.proc3`

See Also

“Main entry point” | “Critical section details” | “Temporally exclusive tasks”

Related Examples

- “Specify Analysis Options”

- “Modelling Synchronous Tasks”
- “Interruptions and Asynchronous Events/Tasks”

More About

- “Priorities”
- “Polyspace Software Assumptions”

Critical section details

Specify the procedures that begin and end critical sections. You can use this option to model protection of shared resources, or to model interruption enabling and disabling.

Settings

Default: None

Click  to add a field.

- In the column **Procedure beginning**, enter the name of the procedure that begins the critical section.
- In the column **Procedure ending**, enter the name of the procedure that ends the critical section.

Dependencies

This option is enabled only if you select the **Multitasking** box.

Command-Line Information

Parameter: `-critical-section-begin | -critical-section-end`
Value: Entries in the form "*`procedure_1_name:critical_section_name`*"
Example: `polyspace-ada -sources filename -entry-points pktasking.one_interrupt1, pktasking.one_interrupt2 -critical-section-begin "pkutil.begin_cs" -critical-section-end "pkutil.end_cs"`

See Also

“Entry points” | “Temporally exclusive tasks”

Related Examples

- “Specify Analysis Options”
- “Modelling Synchronous Tasks”

More About

- “Shared Variables”

Temporally exclusive tasks

Specify the tasks that do not execute simultaneously. You can use this option to implement temporal exclusion of tasks.

Settings

Default: None

Click  to add a field. In each field, enter the name of a group of temporally excluded tasks. For the command-line option, create a temporal exclusions file in the following format:

- On each line, enter one group of temporally excluded tasks.
- Within a line, use spaces to separate tasks.

Dependencies

This option is enabled only if you select the **Multitasking** box.

Command-Line Information

Parameter: `-temporal-exclusions-file`
Value: Name of temporal exclusions file
Example: `polyspace-ada -sources` *`filename`* `-entry-points pktasking.one_interrupt1, pktasking.one_interrupt2 -temporal-exclusions-file "C:\exclusions_file.txt"`

See Also

“Entry points” | “Critical section details”

Related Examples

- “Specify Analysis Options”
- “Modelling Synchronous Tasks”

More About

- “Shared Variables”

Verify module

Specify that Polyspace should generate a `main` subprogram during verification.

Settings

Default: On

On

Polyspace generates a `main` subprogram.

Entities	Action of generated main
Procedures and functions	The generated `main` calls procedures and functions that are: • Specified in package declarations in the code. If a package declaration is inside a procedure or another package, the generated `main` does not call procedures in the declaration. • Specified outside a package. For these procedures and functions, Polyspace initializes the `in` and `out` parameters with random values. The generated `main` does not call procedures and functions that are already called in the code.
Global variables	The generated `main` assigns a random value to global variables that are specified in the : • Package declaration. If you do not initialize a global variable before reading it in the package body, Polyspace generates an orange `Non-initialized`

Entities	Action of generated main
	`variable` check. The check is orange because there can be some execution paths where the global variable is written outside the package body before it is read. • Package body. If you do not initialize a global variable before reading it in the package body, Polyspace generates a red `Non-initialized variable` check if you read the variable in the package body. The check is red because there cannot be an execution path where the global variable is written outside the package body before it is read.

Off

Polyspace does not generate a `main` subprogram. Instead it uses the procedure you specified using the option **Main entry point** as the `main` subprogram.

Tips

- If you use the option **Verify module**, the software treats tasks specified in the code using the `task` keyword as ordinary procedures. In particular, it ignores:
  - Entry calls using the `accept` keyword.
  - Protection mechanism for shared variables.

Command-Line Information

Parameter: `-main-generator`

See Also

"Verify whole application" | "Main entry point" | "Initialization of uninitialized global variables"

Related Examples

- “Specify Analysis Options”
- “Modelling Synchronous Tasks”

Verify files independently

Specify that a separate verification job will be created for each source file. Each file is compiled, sent to the remote verification server, and verified individually. Verification results can be viewed for the entire project, or for individual units.

Settings

Default: Off

☑ On

Polyspace creates a separate verification job for each source file.

☐ Off

Polyspace creates a single verification job for all source files in a module.

Dependencies

This option is enabled only if you select **Verify module** on the **Configuration** pane.

Command-Line Information

Parameter: `-unit-by-unit`
Example: `polyspace-ada -sources filename -unit-by-unit`

See Also

"Common source files"

Related Examples

- "Specify Analysis Options"
- "Running Verification Unit-by-Unit"

Common source files

Specify files that you want to include with each source file verification. These files are compiled once, and then linked to each verification. For instance, if multiple source files call the same procedure, you can use this option to specify the file that contains the procedure definition. Otherwise, Polyspace stubs procedures that are called but not defined in the source files.

Settings

Default: None

Click  to add a field. Enter full path to file. Alternatively, you can use  to navigate to file location.

Command-Line Information

Parameter: `-unit-by-unit-common-source`
Value: Full path to file
Example: `polyspace-ada -sources` *`filename`* `-unit-by-unit -unit-by-unit-common-source "C:/polyspace/function.adb"`

See Also

"Verify files independently"

Related Examples

- "Specify Analysis Options"
- "Running Verification Unit-by-Unit"

Variable/function range setup

Specify range for global variables or `in` and `in out` parameters of procedures and functions using a **Data Range Specifications** template file.

Settings

Default: None

Enter full path to template file. Otherwise use  to navigate to file location.

The template file can be a text file where you provide the ranges in a specific format. For more information, see "DRS Text File Format".

Command-Line Information

Parameter: `-data-range-specifications`
Value: Full path to Data Range Specifications template file
Example: `polyspace-ada -sources` *`filename`* `-data-range-specifications "C:\Polyspace\drs.txt"`

See Also

"Initialization of uninitialized global variables" | "No automatic stubbing"

Related Examples

- "Specify Analysis Options"
- "Specifying Data Ranges Using Text Files"
- "Performing Efficient Module Testing with DRS"
- "Reducing Orange Checks with DRS"

More About

- "Overview of Data Range Specifications (DRS)"

No automatic stubbing

Specify that verification must stop if a procedure is not defined in the source files. Unless you select this option, Polyspace stubs procedures that are not defined and continues verification. If you select this option, the software displays procedures that are not defined and stops verification

Settings

Default: Off

☑ On

Polyspace displays a list of undefined procedures and stops verification.

☐ Off

Polyspace stubs undefined procedures only.

Tips

Use this option when:

- The code you are verifying must be complete.
- You prefer to stub undefined procedures manually.

In either case, this option allows you to find procedures that are not defined in your source.

Dependencies

You cannot use this option with **Initialization of uninitialized global variables**.

Command-Line Information

Parameter:`-no-automatic-stubbing`

Related Examples

- "Specify Analysis Options"
- "Manual vs. Automatic Stubbing"

- “Automatic Stubbing”

More About

- “Stubbing Overview”

Initialization of uninitialized global variables

Specify how Polyspace treats global variables that are not initialized.

Settings

Default: `No initialization`

`No initialization`

Polyspace considers the global variables as uninitialized. If the variable is read before being written, Polyspace produces a red or orange `Non initialized variable` check.

`With random value`

Polyspace initializes the global variables with random values.

`With zero or random value`

Polyspace initializes the global variables with zero if the variable type allows the value zero. Otherwise, it initializes them with random values.

Dependencies

You cannot use this option if you select:

- **Inputs & Stubbing > No automatic stubbing**
- **Verification Mode > Verify module**

Command-Line Information

Parameter: `-init-stubbing-vars-random | -init-stubbing-vars-zero-or-random`
Example: `polyspace-ada -sources` *`filename`* `-init-stubbing-vars-random`

See Also

"Verify module"

Related Examples

- "Specify Analysis Options"

- “Choosing Contextual Verification Options”

Ignore float rounding

Specify that operations involving the type `float` does not involve rounding.

Settings

Default: Off

☑ On

The verification considers that operations involving the type `float` do not involve rounding.

☐ Off

The verification assumes that results of operations involving float are rounded according to the IEEE® 754 standard:

- Simple precision on 32-bit targets
- Double precision on 64–bit targets

Command-Line Information

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

Related Examples

- "Specify Analysis Options"
- "Float Rounding"

Continue after noninitialized variables

Specify that verification must continue past a red non-initialized variable.

Settings

Default: Off

☑ On

Polyspace continues verification even after it detects a red non-initialized variable.

```
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;
```

☐ Off

Polyspace does not continue verification after it detects the first red non-initialized variable. Polyspace declares the subsequent code as unreachable.

Tips

Use this option for first runs of the verification. This option causes loss of precision.

Command-Line Information

Parameter: `-continue-with-all-niv`

See Also

"Continue with noninitialized in/out parameters" | "Initialization of uninitialized global variables" | "Non-Initialized Local Variable: NIVL" | "Verify module"

Related Examples

- “Specify Analysis Options”

Continue with noninitialized in/out parameters

Specify that verification must continue even if `in` and `in out` parameters of a procedure are not initialized.

Settings

Default: Off

☑ On

Polyspace continues verification even after it detects a red non-initialized parameter.

```
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

Polyspace does not continue verification after it detects the first red non-initialized parameter. Polyspace declares the subsequent code as unreachable.

Command-Line Information

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

See Also

"Continue after noninitialized variables" | "Non-Initialized Local Variable: NIVL" | "Variable/function range setup"

Related Examples

- "Specify Analysis Options"

Treat import as nonvolatile

Specify that Polyspace must not consider variables imported through a `pragma Import` as volatile variables. `pragma Import` is used to import variables from code written in a language other than Ada.

Settings

Default: Off

☑ On

Polyspace considers the imported variables as volatile.

☐ Off

Polyspace does not consider the imported variables as volatile.

Command-Line Information

Parameter: `-import-are-not-volatile`

See Also

"Treat export as nonvolatile"

Related Examples

- "Specify Analysis Options"
- "Volatile Variables"
- "Stubbing"

Treat export as nonvolatile

Specify that Polyspace® must not consider variables exported through a `pragma Export` as volatile variables. `pragma Export` is used to export variables to code written in a language other than Ada.

Settings

Default: Off

☑ On

Polyspace considers the exported variables as volatile.

☐ Off

Polyspace does not consider the exported variables as volatile.

Command-Line Information

Parameter: `export-are-not-volatile`

See Also

"Treat import as nonvolatile"

Related Examples

- "Specify Analysis Options"
- "Volatile Variables"
- "Stubbing"

Procedures known to cause NTC

Specify procedures and functions that Polyspace must exclude from checks for non terminating calls.

Settings

Default: None

Click  to add a field. Enter the procedure or function name.

Tips

Use this option when:

- You have intentionally specified infinite loops in some procedure or function. For instance, you can use infinite loops in multitasking code.
- You want to present your results to a third party and filter out certain types of non terminating calls from your results.

Command-Line Information

Parameter: `-known-NTC`
Value: Procedure or function name
Example: `polyspace-ada -sources` *`filename`* `-known-NTC "procedure_1,procedure_2"`

See Also

"Non Terminating Call: NTC"

Related Examples

- "Specify Analysis Options"
- "Preparing Multitasking Code"

Precision level

Specify the precision level that the verification must use. Higher precision leads to greater number of proven results but also requires more verification time. Each precision level corresponds to a different algorithm used for verification.

Settings

Default: 2

0

This option corresponds to a static interval verification.

1

This option corresponds to a complex polyhedron model of domain values.

2

This option corresponds to more complex algorithms closely modelling domain values. The algorithms combine both complex polyhedrons and integer lattices.

Command-Line Information

Parameter: `-O`
Value: `0 | 1 | 2`
Default: `-O2`
Example: `polyspace-ada -sources file_name -O1`

See Also

"Verification level"

Related Examples

- "Specify Analysis Options"
- "Improve Verification Precision"

Verification level

Specify the number of times the Polyspace verification process runs on your source code. Each run can lead to greater number of proven results but also requires more verification time

Settings

Default: `Software Safety Analysis level 2`

`Source Compliance Checking`

The verification process checks for compliance of source code.

`Software Safety Analysis level 0`

The verification process runs once on your code.

`Software Safety Analysis level 1`

The verification process runs twice on your code.

`Software Safety Analysis level 2`

The verification process runs thrice on your code.

`Software Safety Analysis level 3`

The verification process runs four times on your code.

`Software Safety Analysis level 4`

The verification process runs five times on your code.

other

If you use this option, Polyspace verification will make 20 passes unless you stop it manually.

Command-Line Information

Parameter: `-to`
Value: `compile | pass0 |pass1 | pass2 | pass3 | pass4 | other`
Example: `polyspace-ada -sources` *`filename`* `-to pass2`

See Also

"Precision level"

Related Examples

- “Improve Verification Precision”

Verification time limit

Specify a time limit for the verification in hours. If the verification does not complete within that limit, it stops.

Settings

Enter the time in hours. For fractions of an hour, specify decimal form.

Command-Line Information

Parameter: `-timeout`
Value: Time in hours
Example: `polyspace-ada -sources` *`file_name`* `-timeout 5.75`

Related Examples

- "Specify Analysis Options"
- "Improve Verification Precision"

Sensitivity context

Specify that the software must store call context information during verification. If a line of code in a procedure causes a red and green check for two different calls of the procedure, both checks will be stored.

Settings

Default: `auto`

`none`

The software does not store call context information for procedures.

`auto`

The software stores call context information for checks in the following procedures:

- Procedures that form the leaves of the call tree. These procedures are called by other procedures, but do not call procedures themselves.
- Small procedures. The software uses an internal threshold to determine whether a procedure is small.
- Procedures that are called more than once.

`custom`

The software stores call context information for procedures that you specify. Click  to enter the name of a procedure.

Command-Line Information

Parameter: `-context-sensitivity`
Value: `auto | none | -custom` *`procedure_name`*
Example: `polyspace-ada -sources` *`file_name`* `-context-sensitivity auto`

Related Examples

- "Specify Analysis Options"
- "Improve Verification Precision"

Improve precision of interprocedural analysis

Use this option to propagate greater information about function arguments into the called function.

Settings

Default: Off

Enter 0 to turn off this option and 1 to turn it on. Turning on this option leads to greater number of proven results, but also increases verification time.

Tips

- Using this option, you can prove maximum possible number of results when the **Verification level** is set to `Software Safety Analysis level 2`. Therefore, you can save on the number of passes that the verification takes on your code.
- Using this option, you can increase the verification time enormously within a certain pass. Therefore, use this option only when you have less than 1000 lines of code.

Command-Line Information

Parameter: `-path-sensitivity-delta`
Value: `0 | 1`
Example: `polyspace-ada -sources` *`filename`* `-path-sensitivity-delta 1`

Related Examples

- “Specify Analysis Options”
- “Improve Verification Precision”

Specific precision

Specify source files that you want to verify at a **Precision level** higher than that for the entire verification.

Settings

Default: All files are verified with the precision you specified using **Precision** > **Precision level**.

Click  to enter the name of a file and the corresponding precision level.

Command-Line Information

Parameter: `-modules-precision`
Value: File name and corresponding precision separated by `:`
Example: `polyspace-ada -sources` *`file_name`* `-O1 -modules-precision My_File.c:O2`

See Also

"Precision level"

Related Examples

- "Specify Analysis Options"
- "Improve Verification Precision"

Max size of global array variables

Specify a threshold for global array size above which Polyspace must treat each array element as a separate variable. Above the threshold value, each array element appears as an individual variable on the **Variable Access** pane. Reducing the threshold increases verification time.

Settings

Default: 3

Enter an integer in the field provided.

Command-Line Information

Parameter: `-array-expansion-size`
Value: Threshold value
Example: `polyspace-ada -sources` *`filename`* `-O1 -array-expansion-size 8`

See Also

"Expansion limit for a structured variable" | "Variables to expand"

Related Examples

- "Specify Analysis Options"

More About

- "Expansion of Sizes"

Variables to expand

Specify names of record variables that Polyspace must split into its components during verification. Each component appears as an individual variable on the **Variable Access** pane.

Settings

Default None

Click  to add a field. Enter the record variable name.

Dependencies

Specify a value for the option **Expansion limit for a structured variable.** This value applies to record variables named by the option **Variables to expand**.

Command-Line Information

Parameter: `-variables-to-expand`
Value: Variable name
Example: `polyspace-ada -sources` *`filename`* `-variables-to-expand pkg.rec,pkg2.recF -variable-expansion-depth 4`

See Also

“Expansion limit for a structured variable” | “Max size of global array variables”

Related Examples

- “Specify Analysis Options”

More About

- “Expansion of Sizes”

Expansion limit for a structured variable

Specify a limit to the depth of analysis for nested records.

Settings

Default: 1

Enter an integer. This integer specifies a limit to the depth of analysis in nested records.

For instance, consider the following code:

```
Package foo is
 Type Internal is
 Record
  FieldI : Integer;
  FieldII : Integer;
 End Record ;
 Type External is
 Record
  Data : Internal ;
  FieldE : Integer;
 End Record ;
 myVar : External ;
End foo;
```

In this code, if you specify the limit as:

- 1: `foo.myVar.FieldE` and `foo.myVar.Data` are treated as individual variables
- 2: `foo.myVar.FieldE`, `foo.myVar.Data.FieldI` and `foo.myVar.Data.FieldII` are treated as individual variables.

Dependencies

Specify record names using the option **Variables to expand**. The value specified using **Expansion limit for a structured variable** applies to these records.

Command-Line Information

Parameter: `-variable-expansion-depth`

Value: Integer
Example: `polyspace-ada -sources` *`filename`* `-variables-to-expand pkg.rec,pkg2.recF -variable-expansion-depth 4`

See Also

"Variables to expand" | "Max size of global array variables"

Related Examples

- "Specify Analysis Options"

More About

- "Expansion of Sizes"

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

Specify a command or script to be executed after the verification.

Settings

Default: none

Enter full path to the command or script, or click  to navigate to the location of the command or script. For example, you can enter the path to a script that sends an email. After the verification, this script will be executed.

Command-Line Information

Parameter: `-post-analysis`
Value: Full path to script
Example: `polyspace-ada -sources` *`file_name`* `-post-analysis-command` `` `pwd`/send_email ``

Related Examples

- "Specify Analysis Options"

Other

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

See Also

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

Generate report

Specify whether to generate a report during the analysis. Depending on the format you specify, you can view this report using an external software. For example, if you specify the format `PDF`, you can view the report in a `pdf` reader.

Settings

Default: Off

☑ On

Polyspace generates an analysis report using the template and format you specify.

☐ Off

Polyspace does not generate an analysis report. You can still view your results in the Polyspace interface.

Tips

- To generate a report *after* an analysis is complete, select **Reporting** > **Run Report**. Alternatively, at the command line, use the command `polyspace-report-generator` with the options `-template` and `-format`.

Command-Line Information

There is no command-line option to solely turn on the report generator. However, using the options `-report-template` for template and `-report-output-format` for output format automatically turns on the report generator.

Related Examples

- "Specify Analysis Options"
- "Generate Report from User Interface"
- "Generate Report from Command Line"
- "Open Report"

Report template

Specify template for generating analysis report. The report templates are available in the folder *MATLAB_Install*`\polyspace\toolbox\psrptgen\templates\`.

Settings

Default: `Developer`

`CodeMetrics`

The report contains a summary of code metrics, followed by the complete metrics for an application.

`Developer`

The report lists information useful to developers, including:

- Summary of results
- Coding rule violations
- List of proven run-time errors or red checks
- List of unproven run-time errors or orange checks
- List of unreachable procedures or gray checks

The report also contains the Polyspace configuration settings for the analysis.

`DeveloperReview`

The report lists the same information as the `Developer` report. However, the reviewed results are sorted by review classification and status, and unreviewed results are sorted by file location.

`Developer_withGreenChecks`

The report lists the same information as the `Developer` report. In addition, the report lists code proven to be error-free or green checks.

`Quality`

The report lists information useful to quality engineers, including:

- Summary of results
- Statistics about the code
- Graphs showing distributions of checks per file

The report also contains the Polyspace configuration settings for the analysis.

Dependencies

This option is enabled only if you select the **Generate report** box.

Command-Line Information

Parameter: `-report-template`
Value: Name of template with extension `.rpt`
Example: `polyspace-ada -sources` *`file_name`* `-report-template Developer.rpt`

Related Examples

- "Specify Analysis Options"
- "Generate Report from User Interface"
- "Generate Report from Command Line"
- "Open Report"
- "Customize Report Templates"

Output format

Specify output format of generated report.

Settings

Default: `RTF`

`RTF`

Generate report in `.rtf` format

`HTML`

Generate report in `.html` format

`PDF`

Generate report in `.pdf` format

`Word`

Generate report in `.doc` format. Not available on `UNIX` platforms.

`XML`

Generate report in `.xml` format.

Tips

- You must have Microsoft® Office installed to view `RTF` format reports containing graphics, such as the `Quality` report.

Dependencies

This option is enabled only if you select the **Generate report** box.

Command-Line Information

Parameter: `-report-output-format`
Value: `RTF | HTML | PDF | Word | XML`
Default: `RTF`
Example: `polyspace-ada -sources` *`file_name`* `-report-output-format pdf`

Related Examples

- “Specify Analysis Options”
- “Generate Report from User Interface”
- “Generate Report from Command Line”
- “Open Report”

-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.

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 the analysis 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 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

"Overview: Reducing Orange Checks"

-keep-all-files

Specify whether to retain intermediate results and associated working files.

Settings

Default: Off

On

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

Off

Erase 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 one of the complete passes (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 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
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"

-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 reduce 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-ada -tmp-dir-in-results-dir -results-dir C:\Polyspace
\Results
```

-max-processes

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: Integer between 1 and 128
Default: 4
Shell script example: `polyspace-ada -max-processes 1`

Check Descriptions

Non-Initialized Local Variable: NIVL

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

Examples

Ada Example

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

```
36      B(I) := 0.0;
37     end if;
38    end loop;
39    B(2) := B(4) + B(5); -- NIV Warning because
40 B(odd) not initialized
41   end MAIN;
42
43  end NIV;
```

Explanation

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

NIV Check vs. IN OUT Parameter Mode

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

Clearly, in out parameters necessitate initialization before call.

Ada Example

```
1  package NIVIO is
2   procedure MAIN;
3   function Random_Boolean return Boolean;
4  end NIVIO;
5
6  package body NIVIO is
7
8   Y : Integer := 3;
9   procedure Niv_Not_Dangerous(X : in out integer) is
10   begin
11    X := 2;
12    if (Y > 2) then
13     Y := X + 3;
14    end if ;
15   end Niv_Not_Dangerous;
16
17   procedure Niv_Dangerous(X : in out integer) is
18   begin
19    if (Y /= 3) then
```

```
20     Y := X + 3;
21    end if ;
22   end Niv_Dangerous;
23
24   procedure MAIN is
25    X : Integer;
26   begin
27    if (Random_Boolean) then
28     Niv_Dangerous(X); -- NIV ERROR: 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 assumed to have the full range of values allowed by 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.

Non-Initialized Variable: NIV

For variables other than local variables, this check occurs on every variable read. It determines whether the variable being read is initialized.

Example

```
package Example is
   procedure Main;
end Example;

package body Example is
   Var : Integer;

   procedure Main is
      I : Integer;
   begin
      I := Var;
   end Main;
end Example;
```

Division by Zero: ZDV

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

Ada Example:

```
1  package ZDV is
2   function Random_Bool return Boolean;
3   procedure ZDVS (X : Integer);
4   procedure ZDVF (Z : Float);
5   procedure MAIN;
6  end ZDV;
7
8  package body ZDV is
9
10   procedure ZDVS(X : Integer) is
11    I : Integer;
12    J : Integer := 1;
13   begin
14    I := 1024 / (J-X); -- ZDV ERROR: Scalar Division by Zero
15   end ZDVS;
16
17   procedure ZDVF(Z : Float) is
18    I : Float;
19    J : Float := 1.0;
20   begin
21    I := 1024.0 / (J-Z); -- ZDV ERROR: float Division by Zero
22   end ZDVF;
23
24   procedure MAIN is
25   begin
26    if (random_bool) then
27     ZDVS(1); -- NTC ERROR: ZDV.ZDVS call does not terminate
28    end if ;
29    if (Random_Bool) then
30     ZDVF(1.0); -- NTC ERROR: ZDV.ZDVF call does not terminate
31    end if;
32   end MAIN;
33
34  end ZDV;
35
36
37
```

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

```
1
2  With Ada.Numerics; Use Ada.Numerics;
3  With Ada.Numerics.Aux; Use Ada.Numerics.Aux;
4
5  package EXCP is
6   function Bool_Random return Boolean;
7   procedure MAIN;
8  end EXCP;
9
10  package body EXCP is
11
12  -- implementation dependant in Ada.Numerics.Aux: subtype
Double is Long_Float;
13   M_PI_2 : constant Double := Pi/2.0; -- pi/2
14
15   procedure MAIN is
16    IRes, ILeft, IRight : Integer;
17    Dbl_Random : Double;
18    pragma Volatile_ada.htm (dbl_Random);
19
20    SP : Double := Dbl_Random;
21    P : Double := Dbl_Random;
22    SN : Double := Dbl_Random;
23    N : Double := Dbl_Random;
24    NO_TRIG_VAL : Double := Dbl_Random;
25    res : Double;
```

```
26    Fres : Long_Float;
27   begin
28    -- assert is used to redefine range values of a variable.
29    pragma assert(SP > 0.0);
30    pragma assert(P >= 0.0);
31    pragma assert(SN < 0.0);
32    pragma assert(N <= 0.0);
33    pragma assert(NO_TRIG_VAL < -1.0 or NO_TRIG_VAL > 1.0);
34
35    if (bool_random) then
36     res := sqrt(sn); -- EXCP ERROR: argument of SQRT must be
positive.
37    end if ;
38    if (bool_random) then
39     res := tan(M_PI_2);
             -- EXCP Warning: Float argument of TAN
40            -- may be different than pi/2 modulo pi.
41    end if;
42    if (bool_random) then
43     res := asin(no_trig_val); --EXCP ERROR: float argument of
ASIN is not in -1..1
44    end if;
45    if (bool_random) then
46     res := acos(no_trig_val); --EXCP ERROR: float argument of
ACOS is not in -1..1
47    end if;
48    if (bool_random) then
49    res := log(n); -- EXCP ERROR: float argument of LOG is not
strictly positive
50    end if;
51   if (bool_random) then
52     res := exp(710.0); -- EXCP ERROR: float argument of EXP
is not less than or equal to 709 or 88
53    end if;
54
55    -- range results on trigonometric functions
56    if (Bool_Random) then
57     Res := Sin (dbl_random); -- -1 <= Res <= 1
58     Res := Cos (dbl_random); -- -1 <= Res <= 1
59     Res := atan(dbl_random); -- -pi/2 <= Res <= pi/2
60    end if;
61
62    -- Arithmetic functions where a check is not currently
implemented
```

```
63    if (Bool_Random) then
64     Res := cosh(dbl_random);
65     Res := tanh(dbl_random);
66    end if;
67   end MAIN;
68  end EXCP;
```

Explanation

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

Standard *cosh* and *tanh* hyperbolic functions are currently assumed to return the full range of values mathematically possible, 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

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

```
33
34   procedure MAIN is
35   begin
36    if (Bool_Random) then OVFL_ARRAY; end if; -- NTC
propagation because of OVFL ERROR
37    if (Bool_Random) then OVFL_ARITHMETIC; end if;
38   end MAIN;
39
40  end OVFL;
41
42
```

Explanation

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

A different example is shown by the overflow on line 26, where adding 1 to *Integer'Last* (the maximum integer value being *2**31-1* on a 32 bit architecture platform). Similarly, if *OVFL_ARITHMETIC.FValue* represents the max floating value, *2*FValue* cannot be represented with the same type and so raises an overflow at line 30.

Correctness Condition: COR

In this section...

Attributes Check

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

Ada Example

```
1
2  package CORS is
3   function Bool_Random return Boolean;
4   procedure MAIN;
5   function INT_VALUE (S : String) return Integer;
6   type PSTCOLORS is (ORANGE, RED, gray, GREEN);
7   type ADCFUZZY is (LOW, MEDIUM, HIGH);
8  end CORS;
9
10  package body CORS is
```

```
11
12   type STR_ENUM is (AA,BB);
13
14   function INT_VALUE (S : String) return Integer is
15    X : Integer;
16   begin
17    X := Integer'Value (S); -- COR Warning: Value parameter
might not be in range integer
18    return X;
19   end INT_VALUE;
20
21   procedure MAIN is
22    E : PSTCOLORS := GREEN;
23    F : PSTCOLORS;
24    ADCVAL : ADCFUZZY := ADCFUZZY'First;
25    StrVal : STR_ENUM;
26    X : Integer;
27   begin
28    if (Bool_Random) then
29     F := PSTCOLORS'PRED(E); -- COR Verified: Pred attribute
is not used on the first element of pstcolors
30     E := PSTCOLORS'SUCC(E); -- COR ERROR: Succ attribute is
used on the last element of pstcolors
31    end if;
32    if (Bool_Random) then
33     ADCVAL := ADCFUZZY'PRED(ADCVAL); -- COR ERROR: Pred
attribute is used on the first element of adcfuzzy
34    end if ;
35
36    StrVal := STR_ENUM'Value ("AA"); -- COR Warning: Value
parameter might not be in range str_enum
37    StrVal := STR_ENUM'Value ("AC"); -- COR Warning: Value
parameter might not be in range str_enum
38    X := INT_VALUE ("123"); --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

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

Explanation

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

Array Length Check

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

Ada Example

```
1
2  with Dname;
3  package CORL is
4   function Bool_Random return Boolean;
5   type Name_Type is array (1 .. 6) of Character;
```

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

Explanation

At lines 21 and 24, Polyspace gives the exact value 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

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

Checks the length of *DELTA* constructions.

Ada Example

```
1
```

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

Explanation

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

Static Range and Values Check

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

Ada Example

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

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

Explanation

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

Example is shown on the record types *REC2* and *REC3*. 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

Checks the usage of a discriminant in a record declaration.

Ada Example

```
1
2  package DISC is
3   PROCEDURE MAIN;
4
5   TYPE T_Record(A: Integer) is record -- COR Verified: Value
is in range of 1 .. 16#7FFF_FFFF#
6    Sa: String(1..A);
7   END RECORD;
8  end DISC;
9
10  package body DISC is
11
12   PROCEDURE MAIN is
13   begin
```

```
14    declare
15     T_STRING6 : T_RECORD(6) := (6, "abcdef"); --COR Verified:
Discriminant is compatible
16     T_StringOther : T_RECORD(6); -- COR Verified:
Discriminant is compatible
17     T_STRING5 : T_RECORD(5) := (5, "abcde"); -- COR Verified:
Discriminant is compatible
18    begin
19     T_StringOther := T_STRING6; -- COR Verified: Discriminant
is compatible
20     T_string5 := T_Record(T_STRING6); -- COR ERROR:
Discriminant is not compatible
21    end;
22   END Main;
23
24  END DISC;
```

Explanation

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

Component Check

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

Ada Example

```
1  package COMP is
2
3   PROCEDURE MAIN;
4   SUBTYPE DINT IS INTEGER RANGE 0..1;
5   TYPE COMP_RECORD ( D : DINT := 0) is record
6    X : INTEGER;
7    CASE D IS
8     WHEN 0 => ZERO : BOOLEAN;
9     WHEN 1 => UN : INTEGER;
10    END CASE;
11   END RECORD;
12
13  end COMP;
14
15  package body COMP is
16
```

```
17   PROCEDURE MAIN is
18    CZERO : COMP_RECORD(0);
19   BEGIN
20    CZERO.X := 0;
21    CZERO.ZERO := FALSE; -- COR Verified: zero is a component
of the variable
22    CZERO.UN := CZERO.X; -- COR ERROR: un is not a component
of the variable
23   END MAIN;
24  END COMP;
25
```

Explanation

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

Dimension Versus Definition Check

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

Ada Example

```
1  package DIMDEF is
2   PROCEDURE MAIN;
3   FUNCTION Random RETURN boolean;
4  end DIMDEF;
5
6  package body DIMDEF is
7
8   SUBTYPE ST IS INTEGER RANGE 4 .. 8;
9   TYPE BASE IS ARRAY(ST RANGE <>, ST RANGE <>) OF INTEGER;
10   SUBTYPE TBASE IS BASE(5 .. 7, 5 .. 7);
11
12   FUNCTION IDENT_INT(VAL : INTEGER) RETURN INTEGER IS
13   BEGIN
14    RETURN VAL;
15   END IDENT_INT;
16
17   PROCEDURE MAIN IS
18    NEWARRAY : TBASE;
19   BEGIN
20    IF RANDOM THEN
21     NEWARRAY := (7 .. IDENT_INT(9) => (5 .. 7 => 4)); --
```

```
COR Error: Dimension is not compatible with definition
22    END IF;
23    IF Random THEN
24     NEWARRAY := (5 .. 7 => (IDENT_INT(3) .. 5 => 5)); --
COR Error: Dimension is not compatible with definition
25    END IF;
26   END MAIN;
27
28  END DIMDEF;
```

Explanation

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

Aggregate Versus Definition Check

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

Ada Example

```
1
2  package AGGDEF is
3   PROCEDURE MAIN;
4   PROCEDURE COMMENT (A: STRING);
5   function RANDOM return BOOLEAN;
6  end AGGDEF;
7
8  package body AGGDEF is
9
10   TYPE REC1 (DISC : INTEGER := 5) IS RECORD
11    NULL;
12   END RECORD;
13
14   TYPE REC2 (DISC : INTEGER) IS RECORD
15    NULL;
16   END RECORD;
17
18   TYPE REC3 is RECORD
19    COMP1 : REC1(6);
20    COMP2 : REC2(6);
21   END RECORD;
22
23   FUNCTION IDENT_INT(VAL : INTEGER) RETURN INTEGER IS
```

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

Explanation

At 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

Checks the length for array aggregate.

Ada Example

```
1  package AGGLEN is
2   PROCEDURE MAIN;
3   PROCEDURE COMMENT(A: STRING);
4  end AGGLEN;
5
6  package body AGGLEN is
7
8   SUBTYPE SLENGTH IS INTEGER RANGE 1..5;
```

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

Explanation

At line 33, Polyspace shows an error. The static aggregate length is not compatible with the definition of the component FIELD1 at line 21.

Sub-Aggregates Dimension Check

Checks the dimension of sub-aggregates.

Ada Example

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

Explanation

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

Sub-aggregates must be singular.

Characters Check

Checks the construction using the *character* type.

Ada Example

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

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

Ada Example

```
1
2  package CORACCESS is
3   procedure main;
4   function Brand return Boolean;
5  end CORACCESS;
6
7  package body CORACCESS is
8   procedure main is
9
10    type T is new Integer;
11    type A is access all T;
12    Ref : A;
13
14    procedure Proc1(Ptr : access T) is
15    begin
16    Ref := A(Ptr); -- COR Verified: Accessibility level deeper
than that of access type
17    end;
18
19    procedure Proc2(Ptr : access T) is
20    begin
21     Ref := A(Ptr); -- COR ERROR: Accessibility level not
deeper than that of access type
22    end;
23
```

```
24    procedure Proc3(Ptr : access T) is
25    begin
26     Ref := A(Ptr); -- COR Warning: Accessibility level might
be deeper than that of access type
27    end;
28
29    X : aliased T := 1;
30   begin
31    declare
32     Y : aliased T := 2;
33    begin
34     Proc1(X'Access);
35     if BRand then
36      Proc2(Y'Access); -- NTC ERROR: propagation of error
at line 22
37     elsif BRand then
38      Proc3(Y'Access); -- NTC ERROR: propagation of error
at line 27
39     end if;
40    end;
41    Proc3(X'Access);
42   end main;
43  end CORACCESS;
44
```

Explanation

In the example above at line 16: *Ref* is set to *x'access* and *Ref* is defined in same block or in a deeper one. This is authorized.

On the other hand, *y* is not defined in a block deeper or inside the one in which *Ref* is defined. So, at the end of block, *y* does not exist 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.

Accessibility of a Tagged Type

Checks if a tag belongs to a tagged type hierarchy. This check is defined in Ada Standard at chapter 4.6 (paragraph 42).

It detects errors when a Tag of an operand does not refer to class-wide inheritance hierarchy.

Ada Example

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

Explanation

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

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

Polyspace is not precise on tagged types and currently flags each one with a COR warning.

Explicit Dereference of a Null Pointer

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

Ada Example

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

Explanation

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

These checks are currently colored orange.

Power Arithmetic: POW

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

Ada Example

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

```
34   end MAIN;
35  end POWF;
```

Explanation

An error occurs on the power function on integer values "**" with respect to the values of the left and right parameters when *left <= 0 and right < 0*. Otherwise, Polyspace prompts the user to investigate further by means of an orange check.

Note: As 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

```
1
2  package ASRT is
3   function Bool_Random return Boolean;
4   procedure MAIN;
5  end ASRT;
6
7  package body ASRT is
8
9   subtype Intpos is Integer range 0..Integer'Last;
10   subtype TenInt is Integer range 1..10;
11
12   Val_Constant : constant Boolean := True;
13   procedure MAIN is
14    -- Init variables
15    Flip_Flop, Flip_Or_val : Boolean;
16    Ten_Random, Ten_Positive : TenInt;
17    pragma Volatile_ada.htm (ten_random);
18   begin
19
20    if (Bool_Random) then
21     -- Flip_Flop is randomly be True or False
22     Flip_Flop := bool_random;
23
24     -- Flip_Or_Val is 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 without a runtime error at line 32, *Ten_positive* can only have a value greater than 5 after the line.

Non Terminating Call: NTC

- NTC and NTL 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
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: To exclude from verification non-terminating procedures that you want to retain, use the option **Verification Assumptions > Procedures known to cause NTC**.

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

```
1  package NTC is
2   procedure MAIN;
3   -- Stubbed function
4   function Random_Boolean return Boolean;
5  end NTC;
6
7  package body NTC is
8
9   procedure FOO (X : Integer) is
10    Y : Integer;
11   begin
12    Y := 1 / X; -- ZDV Warning: Scalar division
by zero may occur
13    while (X >= 0) loop -- NTL ERROR: Loop does not 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 does not return because of a division by zero (ZDV warning) at line 12 when *X = 0*.

- The second does not terminate because of an infinite loop (red "Non Terminating Loop: NTL" on page 2-44) 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 another task is started.

Example

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

```
21    accept V842;
22    A := A - 1; -- Gray code
23   end MyTask;
24
25   procedure Main is
26    T1 : MyTask;
27   begin
28    A := 0;
29    T1.Start;     -- NTC ERROR: entry task in the main
30    T1.V842;
31    T1.V842;
32    T1.V842;
33    pragma Assert(A=0); -- Gray code
34   end Main;
35  end NTC_entry;
```

Using the launching command `polyspace-ada95 -main NTC_entry.main` on the previous example leads to a red NTC in the main procedure and 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;
```

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 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 does not have an 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;

3  package Main is
4   procedure Main;
5  end Main;
6
7
8  with New_Math; use New_Math;
9  with Types; use Types;
10
11  package body Main is
12   procedure Main is
13   X : My_float;
```

```
14   begin
15    X := Cos(12.3); --range [-1.0 .. 1.0]
16    X := Sin(12.3); --range [-1.0 .. 1.0]
17    X ::= Sqrt(-1.5); --is red: NTC Error
18   end;
19  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 sometimes does not 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.

Known Non-Terminating Call: K_NTC

Description

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

```
1  package KNTC is
2   procedure Put_io (X : Integer);
3   procedure get_data(Data : out Float; Status : out Integer);
4   procedure store_data(Data : in Float);
5   procedure SysHalt(Value : Integer);
6   procedure MAIN;
7  end KNTC;
8
9  package body KNTC is
10
11   -- known NTC function
12   procedure SysHalt(Value : Integer) is
13   begin
14    Put_io(Value);
15    loop -- Non terminating loop
16     null;
17    end loop;
18   end SysHalt;
19
20   procedure MAIN is
21    Status : Integer := 1;
22    Data : Float;
23   begin
24
25    while(Status = 1) loop
26     -- get data
27     get_data(Data, Status);
28     if (status = 1) then
29      store_data(data);
30     end if;
31     if (Status = 0) then
```

```
32      SysHalt(1); -- k-NTC check: Non terminating call
33     end if;
34    end loop;
35   end MAIN;
36  end KNTC;
```

Explanation

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

Non Terminating Loop: NTL

- NTC and NTL 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 cannot be 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

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

```
1
2  package NTL is
3   procedure MAIN;
4   -- Prototypes stubbed as pure functions
```

```
5   procedure Send_Data (Data : in Float);
6   procedure Update_Alpha (A : in Float);
7   end NTL;
8
9  package body NTL is
10
11   procedure MAIN is
12    Acq, Vacq : Float;
13    pragma Volatile_ada.htm (Vacq);
14    -- Init variables
15    Alpha : Float := 0.85;
16    Filtered : Float := 0.0;
17   begin
18    loop    -- NTL information: Non terminating looop
19     -- Acquisition
20     Acq := Vacq;
21     -- Treatment
22     Filtered := Alpha * Acq + (1.0 - Alpha) * Filtered;
23     -- Action
24     Send_Data(Filtered);
25     Update_Alpha(Alpha);
26    end loop;
27   end MAIN;
28  end NTL;
29
```

Explanation

In the above example, the "continuation condition" of the while is always true and the loop does not 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 does not terminate.

```
1  package NTLDO is
2   procedure MAIN;
3  end NTLDO;
4
5  package body NTLDO is
```

```
6   procedure MAIN is
7    A : array(1..20) of Float;
8    J : Integer;
9   begin
10    for I in A'First .. 21 loop -- NTL ERROR: propagation of
OVFL ERROR
11     A(I) := 0.0 ; -- OVFL Warning: 20 verification with
I in [1,20] and one ERROR with I = 21
12     J := I + 1;
13    end loop;
14   end MAIN;
15  end NTLDO;
```

Note: A NTL check can only be red.

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

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

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

Explanation

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

- As shown at line 26, the first branch is always true (*if-then part*) and so the other branch is not 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 not evaluated because of the standard definition of logical operator *or else*.
- The piece of code after a red error is not 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.

3

Approximations Used During Verification

Why Polyspace Verification Uses Approximations

In this section...

What is Static Verification

Polyspace software uses *static verification* to prove the absence of runtime errors. Static verification derives the dynamic properties of a program without actually executing it. This differs significantly from other techniques, such as runtime debugging, in that the verification it provides is not based on a given test case or set of test cases. The dynamic properties proven 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' does not overflow 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).

An 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 a range error will not 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 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 a program variable is a superset of its actual variation domain. The direct consequence is that a runtime error (RTE) item to be checked cannot 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_Install\polyspace\verifier\ada_limitations.pdf

Note: By default, the *Polyspace_Install* folder refers to the following location:

- **Windows systems** – C:\Program Files\Polyspace\PolyspaceForADA_R2013b
- **UNIX® systems** – /usr/local/Polyspace/PolyspaceForADA_R2013b

---