Polyspace® Products for Ada
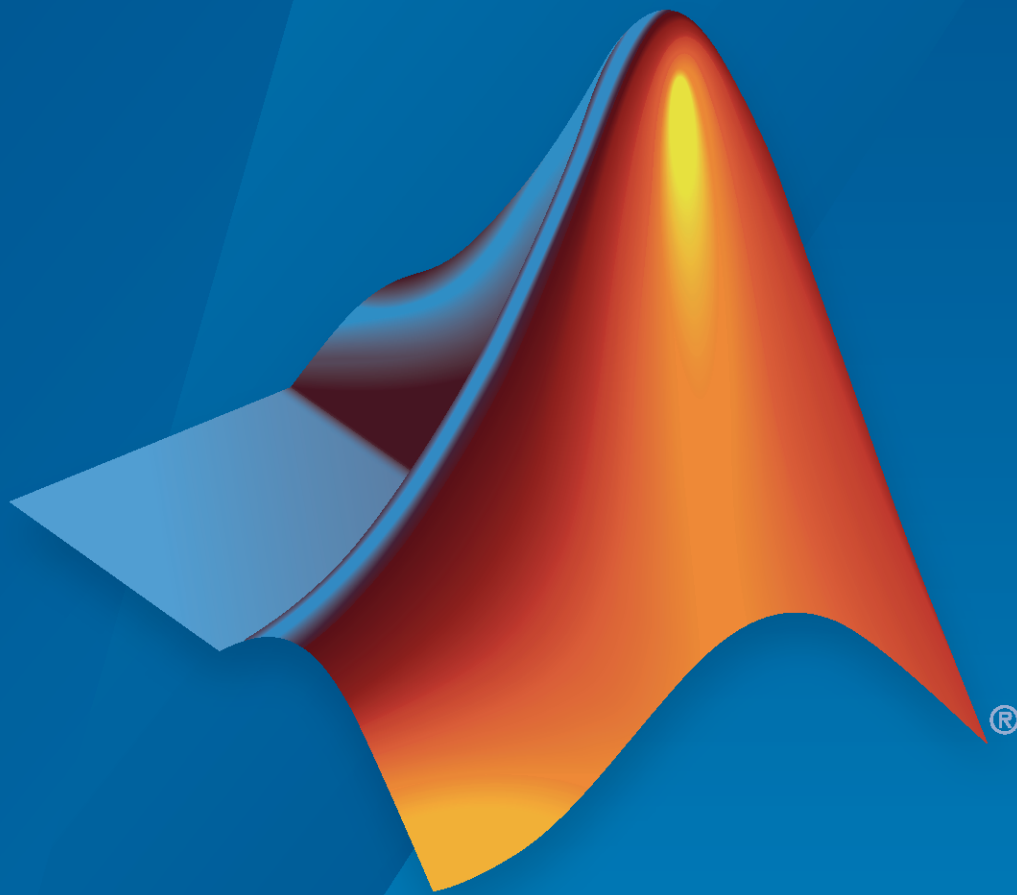
Reference

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Contents

Code Metrics

4

Global Variables

5

Report Components

6

Option Descriptions

Send to Polyspace Server

Specify whether verification runs on the server or client system

Description

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 user interface 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-remote-ada -server 192.168.1.124:12400`

See Also
"Specify Analysis Options" | `Add to results repository`

Add to results repository

Specify upload of analysis results to the Polyspace Metrics results repository

Description

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-remote-ada -server` *IPaddress* `-add-to-results-repository`

See Also
"Specify Analysis Options" | `Send to Polyspace Server`

Source code language

Specify the language of your source files

Description

Specify the language of your source files. If your code is written to Ada83 standards, choose this option before specifying any other configuration options.

This option is available on the **Target & Compiler** node in the **Configuration** pane.

Settings

Default: Ada95

Ada95

> This value restricts the verification to Ada95 language conventions. All files are interpreted and compiled as Ada95 files.

Ada83

> This value restricts the verification to Ada83 language conventions. All files are interpreted and compiled as Ada83 files.

Command-Line Information

Parameter: `-lang`
Value: `ada83` | `ada95`
Default: `ada95`

Target operating system

Specify operating system target

Description

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 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

"Specifying Target & Compiler Parameters"

-shared-variables-mode

Compute global variable sharing and usage without running full analysis

Description

Specify this option to run a less extensive analysis that computes the global variable sharing and usage in your entire application. The analysis does not verify your code for run-time errors. The analysis results also include code metrics if you enable code metrics computation.

Set Option

User interface: In your project configuration, specify the option `-shared-variables-mode` in the **Other** on page 1-51 field on the **Advanced Settings** node.

Command line: Use the option `-shared-variables-mode`. See "Command-Line Information" on page 1-6.

Why Use This Option

You can see global variable sharing and usage without running a full analysis on your entire application that includes run-time error detection. Run-time error detection on an entire application can take a long time.

Settings

☑ On

Polyspace computes global variable sharing and usage but does not verify your code for run-time errors.

◉ Off (default)

Polyspace runs a full analysis on your code, including run-time error detection.

Dependencies

- **User interface**: Select `Verify whole application` and specify `Main entry point`.
- **Command Line**: Specify the `-main` option.

Tips

- After you analyze your complete application to see global variable sharing and usage, run a component-by-component analysis to detect run-time errors.
- In the desktop product, you can see all read and write operations on global variables in the "Variable Access" pane.
- In this less extensive analysis mode, the analysis computes most but not all code metrics.

Command-Line Information
Parameter: `-shared-variables-mode`

Default: Off
Example: `polyspace-ada -sources` *file_name* `-main mainpackage.init -shared-variables-mode`

See Also

Topics
"Analysis Options"

Introduced in R2019b

Target processor type

Specify the target processor type

Description

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

"Specifying Target & Compiler Parameters"

Files extensions

Specify extensions used by package specification files

Description

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

Description

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

"Common Compile Errors"

Value of the constant Storage_Unit

Specify a positive value for System.Storage_Unit.

Description

Specify a positive value for Storage_Unit constant in System package.

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

"Common Compile Errors"

Preprocessor definitions

Define compiler flags for compilation of preprocessor macros

Description

Define compiler flags for compilation of preprocessor macros.

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

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

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

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

expression can be one of the following:

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

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

Settings

No Default

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

  compiler_flag="*value*"

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

  compiler flag. Then, click the button .

- Consider the following example.

```
with Apex_Processes;
with Apex_Types;

package Lift_Load_Control_Process_P is

    procedure Start_S;

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

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

If you specify `VEROCODE="True"`, then Polyspace does not verify code associated with the `#else` and `#end if` parts of the `if` statement. You will still see this code when you view results in the Polyspace user interface. 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-ada -lang ada95 -D HAVE_MYLIB -D No_debug="Yes" -D USE_COM1="true" ...
```

See Also

- `Disable preprocessor definitions`
- "Specifying Target & Compiler Parameters"

Disable preprocessor definitions

Undefine macro compiler flags during compilation phase

Description

Nullify (undefine) macro compiler flags during compilation phase

Settings

No Default

- In the Undefined Preprocessor Macros dialog box, enter *compiler_flag*. Then click the **Adds this item to the list** button .

- Nullifying a macro compiler flag is equivalent to specifying in **Defined Preprocessor Macros** *compiler_flag*="False".

- To remove a compiler flag from the list, in the Undefined Preprocessor Macros dialog box, select the compiler flag. Then, click the button .

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

Command-Line Information

Parameter: -U
Type: string
Shell script example:

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

See Also

- Preprocessor definitions
- "Specifying Target & Compiler Parameters"

Command/script to apply before start of the code verification

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

Description

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

You can run scripts on preprocessed files to work around compilation errors or imprecisions of the analysis while keeping your original source files untouched. For instance, suppose Polyspace does not recognize a compiler-specific keyword. If you are certain that the keyword is not relevant for the analysis, you can run a Perl script to remove all instances of the keyword. When you use this option, the software removes the keyword from your preprocessed code but keeps your original code untouched.

Settings

No Default

Enter full path to the command or script or click ▢ to navigate to the location of the command or script. This script is executed before verification.

For instance, use this Perl script as template. The script replaces all instances of the `Volatile` keyword with the `Import` keyword.

```
#!/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 run a Perl script in Windows®, specify the full path to the Perl executable followed by the full path to the script. For instance, if your Perl script is named `replace_keyword.pl` and you use the Perl executable that comes with MATLAB®, enter the following:

matlabroot`\sys\perl\win32\bin\perl.exe` *absolute_path*`\replace_keyword.pl`

Here, *matlabroot* is the location of the current MATLAB installation such as `C:\Program Files\MATLAB\R2015b\` and *absolute_path* is the location of the Perl script. If the paths contain spaces, use quotes to enclose the full path names.

You can use Perl regular expressions to perform substitutions. For instance, you can use the following expressions.

Expression	Meaning
.	Matches any single character except newline
[a-z0-9]	Matches any single letter in the set a-z, or digit in the set 0-9
[^a-e]	Matches any single letter not in the set a-e
\d	Matches any single digit
\w	Matches any single alphanumeric character or _
x?	Matches 0 or 1 occurrence of x
x*	Matches 0 or more occurrences of x
x+	Matches 1 or more occurrences of x

For complete list of regular expressions, see Perl documentation.

Command-Line Information

Parameter: -pre-analysis-command
Type: string
Value: Script file name or command
Example:: polyspace-ada.exe -pre-analysis-command*matlabroot*\sys\perl\win32\bin
\perl.exe *absolute-path*\replace_keyword

See Also

"Specifying Target & Compiler Parameters"

Include folders

View the include folders used for verification

Description

View the include folders used for verification.

- To add include folders, on the **Project Browser**, right-click your project. Select **Add Source**.
- To view the include folders you used, with your results open, select **Window > Show/Hide View > Configuration**. Under the node **Environment Settings**, you see the folders listed under **Include folders**.

Settings

This is a read-only option available only for your result configuration. Unlike other options, 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

Description

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`

Topics
"Specify Analysis Options"
"Automatically Generating a Main"
"Main Generator Overview"

Main entry point

Specify the procedure that Polyspace verification must use as the `main` subprogram

Description

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`

Topics
"Specify Analysis Options"
"Automatically Generating a Main"
"Main Generator Overview"

Multitasking

Specify whether the code is intended for a multitasking application

Description

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`

Topics
"Specify Analysis Options"
"Modelling Synchronous Tasks"
"Interruptions and Asynchronous Events/Tasks"
"Priorities"
"Polyspace Software Assumptions"

Entry points

Specify the procedures that Polyspace must consider as entry points in a multitasking application

Description

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

Settings

No Default

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

Topics
"Specify Analysis Options"
"Modelling Synchronous Tasks"
"Interruptions and Asynchronous Events/Tasks"
"Priorities"
"Polyspace Software Assumptions"

Critical section details

Specify the procedures that begin and end critical sections

Description

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

No Default

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`

Topics
"Specify Analysis Options"
"Modelling Synchronous Tasks"
"Shared Variables" on page 3-9

Temporally exclusive tasks

Specify the tasks that do not execute concurrently.

Description

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

Settings

No Default

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`

Topics
"Specify Analysis Options"
"Modelling Synchronous Tasks"
"Shared Variables" on page 3-9

Verify module

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

Description

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 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`

Topics
"Specify Analysis Options"
"Modelling Synchronous Tasks"

Verify files independently

Specify that each source file must be verified independently of other source files

Description

Specify that each source file must be verified independently of other source files. Each file is verified individually, independent of other files in the module. Verification results can be viewed for the entire project, or for individual units.

After you open the verification result for one file, you can see a summary of results for all files on the **Dashboard** pane. You can open the results for each file directly from this summary table. For more information, see "Run File-by-File Local Verification".

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.

Tips

- If you perform a file by file verification, you cannot specify multitasking options.
- If your verification for the entire project takes very long, perform a file by file verification. After the verification is complete for a file, you can view the results while other files are still being verified.

Command-Line Information
Parameter: `-unit-by-unit`
Example: `polyspace-ada -sources` *`filename`* `-unit-by-unit`

See Also
`Common source files`

Topics
"Specify Analysis Options"
"Run File-by-File Local Verification"

1-27

Common source files

Specify files that you want to include with each source file verification

Description

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

No Default

Click ![add icon] to add a field. Enter full path to file. Alternatively, you can use ![folder icon] 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`

Topics
"Specify Analysis Options"
"Run File-by-File Local Verification"

Constraint setup

Specify range for global variables or `in` and `in out` parameters of procedures and functions

Description

Specify range for global variables or `in` and `in out` parameters of procedures and functions using a text file.

Settings

No Default

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 "Constraint File Format".

Command-Line Information

Parameter: `-data-range-specifications`
Value: Full path to text file with constraints
Example: `polyspace-ada -sources` *filename* `-data-range-specifications`
`"C:\Polyspace\drs.txt"`

See Also
`Initialization of uninitialized global variables`

Topics
"Specify Analysis Options"
"Specifying Constraints Using Text Files"
"Performing Efficient Module Testing with Constraints"
"Reducing Orange Checks with External Constraints"

Initialization of uninitialized global variables

Specify how Polyspace treats global variables that are not initialized

Description

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:

• **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`

Topics
"Specify Analysis Options"
"Choosing Contextual Verification Options"

Continue after noninitialized variables

Specify that verification must continue past a red noninitialized variable

Description

Specify that verification must continue past a red noninitialized variable.

Settings

Default: Off

☑ On

Polyspace continues verification even after it detects a red noninitialized 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 noninitialized 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 | Verify module`

Topics
"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

Description

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 noninitialized 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 noninitialized 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` | `Constraint setup`

Topics
"Specify Analysis Options"

Treat import as nonvolatile

Specify that Polyspace must not consider variables imported through a `pragma Import` as volatile variables

Description

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`

Topics
"Specify Analysis Options"
"Volatile Variables" on page 3-8
"Stubbing Overview"

Treat export as nonvolatile

Specify that Polyspace should not consider variables exported through a `pragma Export` as volatile variables

Description

Specify that Polyspace should 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`

Topics
"Specify Analysis Options"
"Volatile Variables" on page 3-8
"Stubbing Overview"

Precision level

Specify the precision level that the verification must use

Description

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: -0
Value: 0 | 1 | 2
Default: -02
Example: `polyspace-ada -sources file_name -01`

See Also
`Verification level`

Topics
"Specify Analysis Options"
"Improve Verification Precision"

Verification level

Specify the number of times the Polyspace verification process runs on your source code

Description

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`

Topics
"Improve Verification Precision"

Verification time limit

Specify a time limit for the verification

Description

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`

See Also

Topics
"Specify Analysis Options"
"Improve Verification Precision"

Sensitivity context

Specify that the software must store call context information during verification

Description

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`

See Also

Topics
"Specify Analysis Options"
"Improve Verification Precision"

Improve precision of interprocedural analysis

Propagate greater information about function arguments into the called function

Description

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`

See Also

Topics
"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

Description

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`

Topics
"Specify Analysis Options"
"Improve Verification Precision"

Max size of global array variables

Specify a threshold for global array size

Description

Specify a threshold for global array size.

- For array sizes less than or equal to this threshold, Polyspace treats each array element as a separate variable. Each array element appears as an individual variable on the **Variable Access** pane.
- For array sizes greater than this threshold, Polyspace treats the array as one variable.

Increasing the threshold increases the 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`

Topics
"Specify Analysis Options"
"Expansion of Sizes" on page 3-6

Variables to expand

Specify names of record variables that Polyspace must split into its components during verification

Description

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`

Topics
"Specify Analysis Options"
"Expansion of Sizes" on page 3-6

Expansion limit for a structured variable

Specify a limit to the depth of analysis for nested records

Description

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`

Topics
"Specify Analysis Options"
"Expansion of Sizes" on page 3-6

Generate report

Specify whether to generate a report during the analysis

Description

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.

See Also

Topics
"Specify Analysis Options"
"Generate Report"

Report template

Specify template for generating analysis report

Description

Specify template for generating analysis report.

The report templates are available in the folder *MATLAB_Install*\toolbox\polyspace\psrptgen \templates\. Here, *MATLAB_Install* is the MATLAB installation folder.

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 severity 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: Full path to *template*`.rpt`
Example: `polyspace-ada -sources` *file_name* `-report-template -report-template`
matlabroot`\toolbox\polyspace\psrptgen\templates\Developer.rpt`

See Also

Topics
"Specify Analysis Options"
"Generate Report"
"Customize Report Templates"

Output format

Specify output format of generated report

Description

Specify output format of generated report.

Settings

Default: Word

HTML

 Generate report in `.html` format

PDF

 Generate report in `.pdf` format

Word

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

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

See Also

Topics
"Specify Analysis Options"
"Generate Report"

Disable code metrics generation

Specify that Polyspace must not generate code complexity metrics for your source code

Description

Specify that Polyspace must not generate code complexity metrics for your source code. For more information, see "Code Metrics".

Settings

Default: Off

☑ On

Polyspace does not generate code complexity metrics for your source code.

☐ Off

Polyspace generates code complexity metrics for your source code. After verification, your results contain these code complexity metrics.

Command-Line Information
Parameter: `-no-code-metrics`
Example: `polyspace-ada -sources` *file_name* `-no-code-metrics`

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

Specify a command or script to be executed after the verification

Description

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

Settings

No Default

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`

See Also

Topics
"Specify Analysis Options"

Other

Specify extra Polyspace options

Description

Specify extra Polyspace options.

Settings

No Default

- 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

"Polyspace Software Administration"

-ada-include-dir

Specify a folder that contains include files required for compiling your sources

Description

`-ada-include-dir` *folder* specifies a folder that contains include files required for compiling your sources. You can specify only one folder for each instance of `-ada-include-dir`. However, you can specify this option multiple times.

For instance, to specify two include folders `com/inc` and `com/sys/inc`, use the following syntax:

```
polyspace-ada -ada-include-dir com/inc -ada-include-dir com/sys/inc
```

-author

Specify author of verification

Description

`-author` *authorName* specifies author of verification.

Examples

Default: user ID

Example Shell Script Entry:

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

-server

Specify server name and port number

Description

`-server` *server_name_or_ip*[:*port_number*] specifies a server name and port number for remote analysis.

Using `polyspace-remote-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.

Examples

`polyspace-remote-ada -desktop -server 192.168.1.124:12400 ...`

`polyspace-remote-ada ...`

`polyspace-remote-ada -server Bergeron ...`

-help

Display list of analysis options

Description

-h or -help displays simple help in the shell window that provides information on the analysis options.

Examples

polyspace-ada -h

polyspace-ada -help

-version

Display Polyspace version number

Description

`-v` or `-version` displays the Polyspace version number.

Examples

```
polyspace-ada -v
```

produces an output like the following:

```
Polyspace r2011b
Copyright (c) 1999-2011 The Mathworks, Inc.
```

-sources

Specify list of source files to analyze

Description

`-sources` *file1*`[,`*file2*`[,...]]` specifies the list of source files you want to analyze. You can use standard UNIX® wildcards with this option to specify your sources. The source files are compiled in the order in which they are specified.

Examples

To analyze the files `source1.ada` and `source2.ada`, use the following syntax:

```
polyspace-ada -sources source1.ada,source2.ada
```

-sources-list-file

Specify a text file with names of sources to analyze

Description

`-sources-list-file` *file_name* is available only in batch mode.

Here, *file_name* specifies:

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

Examples

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

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

-from

Specify starting point of verification

Description

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.

Examples

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

-report-output-name

Specify name of verification report file

Description

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.

Examples

Parameter: `report-output-name`
Type: string
Default: *Prog_TemplateName.Format*

-import-comments

Import coding rule and run-time check comments and justifications from specified folder at the end of verification

Description

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

Examples

Default:

Disabled

Shell script examples: :

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

-tmp-dir-in-results-dir

Use a subfolder of the results folder to store temporary files

Description

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.

Examples

Default:

Disabled

Example Shell Script Entry:

polyspace-ada -tmp-dir-in-results-dir -results-dir C:\Polyspace\Results

-max-processes

Specify the maximum number of processors that you want the verification to use

Description

`-max-processes` *num* specifies the maximum number of processes that you want the analysis to use. On a multicore system, the software parallelizes the analysis and creates the specified number of processes to speed up the analysis. The valid range of *num* is 1 to 128.

Unless you specify this option, the verification uses up to four processors. If you have fewer than four processors, the verification uses the maximum available number. To increase or restrict the number of processors, use this option.

To use this option effectively, determine the number of processors available for use. If the number of processes you create is greater than the number of processors available, the analysis does not benefit from the parallelization. Check the system information in your operating system. When you start a verification, a message states the number of logical processors detected on your system.

Settings

Default: 4

- Valid range is 1 to 128
- To disable parallel processing, set to 1.

Tips

You must have at least 4 GB of RAM per processor for analysis. For instance, if your machine has 16 GB of RAM, do not use this option to specify more than four processors.

Examples

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

-generate-launching-script-for

Extracts information from Polyspace project file created in user interface for verification from the command line.

Description

Extracts information from the project file so that you can run a verification from the command-line. When you run this command, for each configuration and each module in your project, a folder is created containing the following files:

- `source_command.txt` — List of source files for the `-source-files` option.
- `options_command.txt` — List of the analysis options for the `-options-file` option.
- `temporal_exclusions.txt` — List of temporal exclusions, generated only if you specify the `Temporally exclusive tasks` option.
- `.polyspace_conf.psprj` — A copy of the project file used to generate the scripting files.
- `launchingCommand.sh` (UNIX) or `launchingCommand.bat` (DOS) — shell script that calls the correct commands. The script also contains any options that cannot be given to the `-options-file` command, such as `-server` or `-add-to-results-repository`. You can give this file additional analysis options as parameters.

Examples

This option is used with the desktop binary `polyspace`.

Parameter: `-generate-launching-script-for`
Value: `psprj` project file
Shell script example: `polyspace -generate-launching-script-for Demo_Ada.psprj`

-list-all-values

Display valid option arguments for a given command-line option

Syntax

`-list-all-values` *option*

Description

`-list-all-values` *option* displays all the valid option arguments for the command-line option *option*.

Examples

Display the valid option arguments for option `-sources-encoding`:
Parameter: `-list-all-values`
Value: *option*
Shell script example: `polyspace-ada -list-all-values -sources-encoding`

See Also

Topics
"Run Verification"

`-xml-annotations-description`

Apply custom code annotations to your Polyspace analysis results

Description

`-xml-annotations-description` *file_path* uses the annotation syntax defined in the XML file located in *file_path* to interpret code comments in your source files. You can use the XML file to specify an annotation syntax and map it to the Polyspace annotation syntax. When you run an analysis using this option, you can justify and hide results with annotations that use your syntax. If you run Polyspace from the command line, *file_path* is the absolute path or path relative to the folder from which you run the command. If you run Polyspace from the user interface, *file_path* is the absolute path.

If you are running an analysis from the user interface, you can enter this option in the **Other** field, under the **Advanced Settings** node on the **Configuration** pane. See `Other`.

If you have existing annotations from previous code reviews, you can import these annotations to Polyspace. You do not have to review and justify results you have already annotated. Similarly, if your code comments must adhere to a specific format, you can map and import that format to Polyspace.

Examples

Parameter: `-xml-annotations-description`
Value: *path\to\file1.xml*
Shell script example: `polyspace-ada -sources` *source_path* `-xml-annotations-description` *C:\path\to\file1.xml*

See Also

Topics
"Justify Results Through Code Annotations"
"Define Custom Annotation Format"

polyspace-access

(DOS/UNIX) Manage upload or export of Polyspace results from the Polyspace Access web interface

Syntax

```
polyspace-access -host hostname [configuration options] -create-project
projectFolder
```

```
polyspace-access -host hostname [configuration options] -upload
pathToFolderOrZipFile [upload options]
polyspace-access -host hostname [configuration options] -export
findingsToExport -output filePath [export options]
```

```
polyspace-access -host hostname [configuration options] -set-unassigned-
findings findingsToAssign -owner userToAssign -source-contains pattern [set
unassigned findings options]
```

```
polyspace-access -host hostname [configuration options] -list-project [
findingsPath]
```

```
polyspace-access -host hostname [configuration options] -set-role role -user
username -project-path projectFolderOrFindingsPath
polyspace-access -host hostname [configuration options] -unset-role -user
username -project-path projectFolderOrFindingsPath
```

```
polyspace-access -encrypt-password
```

```
polyspace-access -generate-migration-commands metrics_dir -output-folder-path
dir [generate migration commands options]
polyspace-access -host hostname [configuration options] -migrate -option-
file-path dir [-dryrun]
```

Description

`polyspace-access -host hostname [configuration options] -create-project projectFolder` creates a project folder in the Polyspace Access web interface. The folder can be at the top of the project hierarchy or a subfolder under an existing project folder.

`polyspace-access -host hostname [configuration options] -upload pathToFolderOrZipFile [upload options]` uploads Polyspace results from a folder or a zipped file to the Polyspace Access database. Use the `upload options` to specify a project folder other than `public`.

`polyspace-access -host hostname [configuration options] -export findingsToExport -output filePath [export options]` exports project results from a project in the Polyspace Access database to a text file whose location you specify with `filePath`. You specify the project using either the full path in Polyspace Access or the run ID. Use this command to export findings to other tools that you use for custom reports. To get the paths to projects and their last run IDs, use the `polyspace-access` command with the `-list-project` command.

`polyspace-access -host hostname [configuration options] -set-unassigned-findings findingsToAssign -owner userToAssign -source-contains pattern [set unassigned findings options]` assigns owners to unassigned results in a project in the Polyspace Access database. You specify the project using either the full path in Polyspace Access or the run ID. Use the `set unassigned findings options` to assign findings from different source files or different groups of source files to different owners. To get the paths to projects and their last run IDs, use the `polyspace-access` command with the `-list-project` command.

`polyspace-access -host hostname [configuration options] -list-project [findingsPath]` without the optional argument `findingsPath` lists the paths to all projects in the Polyspace Access database and their last run IDs. If you specify the full path to a project with the argument `findingsPath`, the command lists the last run ID.

`polyspace-access -host hostname [configuration options] -set-role role -user username -project-path projectFolderOrFindingsPath` assigns a role `role` to the user specified by `username` for the specified project or project folder. A user role set for a project folder applies to all project findings under that folder. You specify the project using either the full path in Polyspace Access or the last run ID. To get the paths to projects and their last run IDs, use the `polyspace-access` command with the `-list-project` command.

`polyspace-access -host hostname [configuration options] -unset-role -user username -project-path projectFolderOrFindingsPath` removes any role previously assigned to `username` for the specified project or project folder. You specify the project using either the full path in Polyspace Access or the last run ID. To get the paths to projects and their last run IDs, use the `polyspace-access` command with the `-list-project` command.

`polyspace-access -encrypt-password` encrypts the password you use to log into Polyspace Access. Use the output of this command as the argument of the `-encrypted-password` option when you write automation scripts to interact with Polyspace Access.

`polyspace-access -generate-migration-commands metrics_dir -output-folder-path dir [generate migration commands options]` generates scripts to migrate projects from the path `metrics_dir` in Polyspace Metrics to Polyspace Access. The command stores the scripts in `dir`. To specify which project findings to migrate, use `generate migration commands options`.

`polyspace-access -host hostname [configuration options] -migrate -option-file-path dir [-dryrun]` migrates projects from Polyspace Metrics to Polyspace Access using the scripts generated with the `-generate-migration-commands` command. To view which projects are migrated without actually migrating the projects, use the `-dryrun` option.

Examples

Encrypt Password and Set Configuration Options

Polyspace Access requires login credentials. You can enter them at the command line when you execute a command, or you can generate an encrypted password that you use in automation scripts.

To encrypt your password, use the `-encrypt-password` command and enter your Polyspace Access credentials.

```
polyspace-access -encrypt-password
```

The command uses the user name and password you enter to generate an encrypted password.

```
login: jsmith
password:
CRYPTED_PASSWORD KEAGKAMJMCOPLFKPKOHOJNDJCBACFJBL
Command Completed
```

If you manage your analysis findings through automated scripts, create a variable to store the connection configuration and login credentials. Use this variable in your script, or at the command line to avoid entering your credentials when you execute a command.

```
set LOGIN=-host my-company-server -port 1234 ^
-protocol https -login jsmith ^
-encrypted-password KEAGKAMJMCOPLFKPKOHOJNDJCBACFJBL
polyspace-access %LOGIN% -create-project myProject
```

Create a Project Folder with Restricted Access and Upload to Folder

Suppose that you want to upload a set of findings to Polyspace Access and authorize only some team members to view these findings.

Create a project folder `Restricted` at the top of the project hierarchy.

```
polyspace-access -host my-company-server -port 1234 ^
-create-project Restricted
```

Set user roles for users `aUser` and `bUser`, authorizing them to access the project folder as contributors.

```
polyspace-access -host my-company-server ^
-port 1234 -set-role contributor ^
-user aUser -user bUser -project Restricted
```

Aside from the creator of the project folder and the previous two users, no other user can view or access any findings uploaded to `Restricted`.

Upload project findings under `Restricted`.

```
polyspace-access -host my-company-server -port 1234 ^
-upload C:\Polyspace_Workspace\projectName\Module_1 ^
-parent-project Restricted
```

The uploaded findings are stored under `Restricted/projectName`.

Assign Results to Component Owners and Export Assigned Results

If you follow a component-based development approach, you can assign analysis findings by component to their respective owners.

Get a list of projects currently stored on the Polyspace Access database.

```
polyspace-access -host my-company-server ^
-list-project
```

The command outputs a list of project findings paths and their last run ID.

```
Connecting to https://my-company-server:9443
Connecting as jsmith
Get project list with the last Run Id
Restricted/Code_Prover_Example (Code Prover) RUN_ID 14
multimodule/vxWorks_demo (Code Prover) RUN_ID 16
public/Bug_Finder_Example (Bug Finder) RUN_ID 24
public/CP/Code_Prover_Example (Polyspace Code Prover) RUN_ID 8
public/Polyspace (Code Prover) RUN_ID 28
Command Completed
```

Assign all red and orange run-time error findings to the owner of all the files in `Component_A` of project `vxWorks_demo`. Perform the same assignment for the owner of `Component_B`. To specify the `vxWorks_demo` project, use the run ID.

```
polyspace-access -host my-company-server ^
-set-unassigned-findings 16 ^
-owner A_owner -source-contains Component_A ^
-owner B_owner -source-contains Component_B ^
-rte Red -rte Orange
```

`-source-contains Component_A` matches all files with a file path that contains `Component_A`.

`-source-contains Component_B` matches all files with a file path that contains `Component_B`, but excludes files with a file path that contains `Component_A`.

After you assign findings, export the findings and generate `.csv` files for each owner containing the findings assigned to them.

```
polyspace-access -host my-company-server ^
-export 16 ^
-output C:\Polyspace_Workspace\myResults.csv ^
-output-per-owner
```

The command generates file `myResults.csv` containing all findings from the project with run ID 16. The command also generates files `myResutls.csv.A_owner.csv` and `myResults.csv.B_owner.csv` on the same file path.

Migrate Projects from Metrics to Polyspace Access

If you have projects stored on a Polyspace Metrics server, you can migrate them to the Polyspace Access database. Log in to your Metrics server to complete this operation.

Generate migration scripts for the projects you want to migrate. Specify the folder path of the location where the projects are stored, for example `C:\Users\jsmith\AppData\Roaming\Polyspace_RLDatas\results-repository`

```
polyspace-access -generate-migration-commands ^
C:\Users\jsmith\AppData\Roaming\Polyspace_RLDatas\results-repository ^
-output-folder-path C:\Polyspace_Workspace\toMigrate -project-date-after 2017-06
```

The command generates migration scripts for all projects in the specified metrics folder that were uploaded on or after June 2017. The scripts are stored in folder `C:\Polyspace_Workspace\toMigrate`.

Use the `-dryrun` option to check which projects will be migrated.

```
polyspace-access -host my-company-server ^
-migrate -option-file-path ^
C:\Polyspace_Workspace\toMigrate -dryrun
```

The command output contains a list of projects. Inspect it to ensure that you are migrating the correct projects.

To perform the migration, rerun the last command without the `-dryrun` option.

Input Arguments

Connect and Login

`hostname` — Machine host name
string

Fully qualified host name of the machine hosting the Polyspace Access **Gateway** service. You must specify a host name with all `polyspace-access` commands, except the `-generate-migration-commands` and `-encrypt-password` commands .

Example: `-host my-company-server`

`configuration options` — Options to configure connection to Polyspace Access
string

Options to specify connection configuration and login credentials.

Configuration Options

Option	Description
`-port portNumber`	Port number of the Polyspace Access **Gateway** service. The default port number is `9443`.
`-protocol http | https`	HTTP protocol used to access Polyspace Access. The default protocol is `https`.

`-login username`
 	
`-encryted-password ENCRYPTED_PASSWD`
	Login credentials you use to interact with Polyspace Access. The argument of `-encrypted-password` is the output of the `-encrypt-password` command.
	
	If you do not use these two options, you are prompted to enter your credentials at the command line.

Miscellaneous Options

Option	Description
`-output file_path`	File path where you store command outputs
`-tmp-dir folder_path`	Folder path where you store temporary files generated by the `polyspace-access` commands. The default folder path is `tmp/ps_results_server` on Linux and `C:/Users/%username%/AppData/Local/Temp/ps_results_server` on Windows.

Option	Description
`-log`	File path where you store the command output log. By default the command does not generate a log file.
`-h`	Display the help information for `polyspace-access` or one of its commands.

Create New Folder

`projectFolder` — Name of project folder
string

Project folder path specified as a string. If the name includes spaces, use double quotes. Specify the full path to folders nested under a parent folder.

If your folder path involves a folder that does not already exist, the folder is created.

Example: `-create-project topFolder`

Example: `-create-project "topFolder/subFolder/subSubFolder"`

Upload Results

`pathToFolderOrZipFile` — Path to folder or zipped file containing analysis results
string

Folder or zipped file path specified as a string. The folder or zipped file contains analysis results you want to upload to Polyspace Access. Specify the path of the folder containing the `*.psbf`, `*.pscp`, or `*.rte` file, or the path of the parent of this folder to upload multiple analysis runs.

For instance, for the Bug Finder results stored in `C:\Polyspace_Workspace\myProject\Module_1\BF_results\ps_results.psbf`, specify the path to `BF_results` or to `Module_1`. If the path name includes spaces, use double quotes.

Example: `-upload C:\Polyspace_Workspace\myProject\Module_1\BF_results`

Example: `-upload C:\Polyspace_Workspace\myProject\Module_1\ -project projectFolder`

`upload options` — Options to specify where to upload results
string

Options to specify path to project folder where you upload results.

Option	Description
`-parent-project` *`projectFolder`*	Path of the parent project folder under which you upload project findings. If you do not specify a parent project folder, projects are upload to the **public** folder.

Option	Description
`-project` *projectFolderOrFindi ngsPath*	If the `FOLDER` you specify for `-upload` contains only one analysis run, for instance `ps_results.psbf`, this option is optional. Use `-project` to rename project findings, or omit it to use the project name from your Polyspace analysis. If the `FOLDER` you specify for `-upload` contains more than one analysis run, or if you specify the parent folder of the results folder, this option is mandatory. Use `-project` to create a project folder under which all the analysis runs are stored.

Export Results

findingsToExport — Project findings path or run ID
string

Path or run ID of the project findings that you export. Polyspace assigns a unique run ID to each analysis run you upload. If the path name includes spaces, use double quotes. To get the project findings path or last run ID, use `-list-project`.

Example: `-export "public/Examples/Bug_Finder_Example (Bug Finder)"`

Example: `-export 4`

filePath — Path to file containing command output
string

Path to the file that stores the output of the command when you specify the `-output` option. This option is mandatory with the `-export` command.

Example: `-output C:\Polyspace_Workspace\myResults.txt`

export options — Options to specify which findings to export
string

Options to specify where to export findings, and which subset of findings you export. Use these options to export findings to other tools you use to create custom reports or other custom review templates.

Option	Description
`-output` *file_path*	File path where you export the findings. This option is mandatory with the `-export` command.
`-new-findings`	Export only new findings compared to the previous analysis (previous upload with the same project name).
`-output-per-owner`	Use this option to generate files that only contain findings assigned to a particular user. The files are stored on the path you specify with `-output`.
`-rte` *color*	Type of RTE finding to export. Specify `All`, `Red`, `Gray`, `Orange`, or `Green`. To specify more than one argument, call the option for each argument. For example, `-rte Red -rte Orange`.

Option	Description
`-defects` *impact*	Impact of DEFECTS findings to export. Specify `All`, `High`, `Medium`, or `Low`. To specify more than one argument, call the option for each argument. For example, `-defects Medium -defects Low`.
`-custom-coding-rules`	Export all custom coding rules findings.
`-coding-rules`	Export all coding rules findings.
`-code-metrics`	Export all code metrics findings.
`-global-variables`	Export all global variables findings.
`-review-status` *status*	Review status of the findings to export. Specify `New`, `Unreviewed`, `Unassigned`, `Toinvestigate`, `Tofix`, `Justified`, `Noactionplanned`, `Notadefect`, `Other`, or `Annotated`. To specify more than one argument, call the option for each argument. For example, `-review-status Tofix -review-status Toinvestigate`.
`-severity` *severity*	Severity of the findings to export. Specify `All`, `High`, `Medium`, or `Low`. To specify more than one argument, call the option for each argument. For example, `-severity High -severity Low`.
`-open-findings-for-sqo` *sqo_level*	Software quality objective or SQO level that must be satisfied. Specify a number from 1 to 6 for *sqo_level*. If you specify an SQO level, the `polyspace-access` command exports only open findings that must be fixed or justified to satisfy the requirements of this level. For more information on the SQO levels, see . The SQO levels 1 to 6 specify an increasingly stricter set of requirements defined in terms of Polyspace results. The requirements are predefined but you can customize them in the Polyspace Access web interface. For instance, SQO level 2 in Code Prover requires that you must not have unjustified red checks. This specification means that if you use `-open-findings-for-sqo` with a level higher than 2, all red checks are exported and must be subsequently fixed or justified. If you want to impose this requirement in the earlier SQO level 1, you can customize level 1 in the Polyspace Access web interface.

You can also use a combination of options. For instance, `-coding-rules -severity High` exports coding rule violations that have been assigned a status of `High` in the Polyspace Access web interface.

Assign Findings

findingsToAssign — Project findings path or run ID
string

Path or run ID of the project findings that you assign to a user. Polyspace assigns a unique run ID to each analysis run you upload. If the path name includes spaces, use double quotes. To get the project findings path or last run ID, use `-list-project`.

Example: `-set-unassigned-findings "public/Examples/Bug_Finder_Example (Bug Finder)"`

Example: `-set-unassigned-findings 4`

userToAssign — Polyspace Access user name
string

User name of user you assign as owner of unassigned findings. To assign multiple owners, call the option for each user.

Each call to `-owner` must be paired with a call to `-source-contains`.

Example: `-user jsmith`

pattern — Pattern to match against file path
string

Pattern to match against file path of project source files. To match file paths for all source files, use `-source-contains /`.

Enter a substring from the file path. You cannot use regular expressions.

When you call this option more than once, each instance excludes patterns from previous instances. For example, `-source-contains foo -source-contains bar` matches all file paths that contain `foo`, then all file paths that contain `bar` excluding paths that contain `foo`.

When you assign findings to multiple owners, call this option for each call to `-owner`.

Example: `-source-contains main`

set unassigned findings options — Options to specify which findings to assign
string

Options to assign all findings or only a subset based on component or individual source files. To make an assignment, specify a pattern to match against the folder or file paths to assign.

Option	Description
`-rte color`	Type of RTE finding to assign. Specify `All`, `Red`, `Gray`, `Orange`, or `Green`. To specify more than one argument, call the option for each argument. For example, `-rte Red -rte Orange`.
`-defects impact`	Impact of DEFECTS findings to assign. Specify `All`, `High`, `Medium`, or `Low`. To specify more than one argument, call the option for each argument. For example, `-defects Medium -defects Low`.
`-custom-coding-rules`	Assign all custom coding rules findings.
`-coding-rules`	Assign all coding rules findings.
`-code-metrics`	Assign all code metrics findings.
`-global-variables`	Assign all global variables findings.

Option	Description
`-review-status` *`status`*	Review status of the findings to assign. Specify `New`, `Unreviewed`, `Unassigned`, `Toinvestigate`, `Tofix`, `Justified`, `Noactionplanned`, `Notadefect`, `Other`, or `Annotated`. To specify more than one argument, call the option for each argument. For example, `-review-status Tofix -review-status Toinvestigate`.
`-severity` *`severity`*	Severity of the findings to assign. Specify `All`, `High`, `Medium`, or `Low`. To specify more than one argument, call the option for each argument. For example, `-severity High -severity Low`.
`-dryrun`	Display command output without making any assignment. Use this option to check that your assignments are correct.

List Projects

findingsPath — Project findings path
string

Path of the project findings. Specify this optional argument with `-list-project` to get the path and the last run ID of the corresponding project findings. If the path name includes spaces, use double quotes.

Example: `-list-project "public/Examples/Bug_Finder_Example (Bug Finder)"`

Set and Unset User Roles

role — Level of access permissions for project folder or findings
owner | contributor | forbidden

Level of access to project folder and findings for a user.

- **owner**: User can move, rename, or delete specified project folders or findings and review their content.
- **contributor**: User can review content of specified project folder or findings.
- **forbidden**: User cannot access specified project folder or findings. Set this role to restrict the access of a user to a set of project findings inside a project folder that is accessible to the user.

Example: `-set-role contributor`

username — Polyspace Access user name
string

Polyspace Access user name.

Example: `-user jsmith`

projectFolderOrFindingsPath — Project folder or findings path
string

Path of a project folder or project findings. When `projectFolderOrFindingsPath` is the path to a project folder, the user role you set applies to all subfolders and project findings under that folder. If

the path name includes spaces, use double quotes. To get the project folder or findings path, use `-list-project`.

Example: `-project-path "public/Examples/Bug_Finder_Example (Bug Finder)"`

Example: `-project-path public`

Migrate Results from Metrics to Polyspace Access

metrics_dir — Folder path of Polyspace Metrics projects
string

Path of folder containing the Polyspace Metrics projects you want to migrate to Polyspace Access.

Example: `-generate-migration-commands C:\Users\%username%\AppData\Roaming\Polyspace_RLDatas\results-repository`

dir — Output folder for migration scripts
string

Path to folder that stores the output of `-generate-migration-commands`. Do not specify an existing folder.

Example: `local/Polyspace_Workspace/migration_scripts`

generate migration commands options — Options to specify which projects to migrate
string

Option	Description
`-output-folder-path` *dir*	Folder path where you want to store the generated command files. Do not specify an existing folder.
`-max-project-runs` *int*	Number of most recent analysis runs you want to migrate for each project. For instance, to migrate only the last two analysis runs of a project, specify 2.
`-project-date-after` *YYYY[-MM[-DD]]*	Only migrate results that were uploaded to Polyspace Metrics on or after the specified date.
`-product` *productName*	Product used to analyze and produce project findings, specified as `bug-finder`, `code-prover`, or `polyspace-ada`.
`-analysis-mode` *mode*	Analysis mode use to generate project findings, specified as `integration` or `unit-by-unit`.

See Also

Introduced in R2018b

Check Descriptions

Function Not Reachable

Function is called from unreachable part of code

Description

This check determines if a procedure is reachable from the main entry point procedure through the program call tree. See also `Main entry point`.

Check Information
Group: Data Flow
Language: Ada
Acronym: FNR

Non-Initialized Local Variable

Local variable is not initialized before being read

Description

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

Examples

Ada Example

```
1        package NIVL 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 NIVL;
10
11    package body NIVL 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; -- NIVL 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 Warning: because of NIVL 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); -- NIVL Warning because
40    B(odd) not initialized
41         end MAIN;
42
43    end NIVL;
```

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 NIVL warning on *B(I)*) when all fields have not been initialized.

NIVL 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 NIVLIO is
2          procedure MAIN;
3          function Random_Boolean return Boolean;
4      end NIVLIO;
5
6      package body NIVLIO is
7
8          Y : Integer := 3;
9          procedure NIVL_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 NIVL_Not_Dangerous;
16
17          procedure NIVL_Dangerous(X : in out integer) is
18          begin
19              if (Y /= 3) then
20                  Y := X + 3;
21              end if ;
22          end NIVL_Dangerous;
23
24          procedure MAIN is
25              X : Integer;
26          begin
27              if (Random_Boolean) then
28                  NIVL_Dangerous(X); -- NIVL ERROR: very significant
29              end if ;
30              if (Random_Boolean) then
31                  NIVL_Not_dangerous(X); -- NIVL ERROR: not significant
32              End if ;
33          end MAIN;
34
35      end NIVLIO;
```

Explanation

In the previous example, as shown at line 28, Polyspace highlights a non-initialized variable that could be a significant error. In the *NIVL_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 NIVL • Permanent random value	• No NIVL 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 NIVL.

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 NIVL • Permanent random value	• No NIVL 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 NIVL • Permanent random value	• No NIVL 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 anything within its whole range between one read access and the next.

Check Information
Group: Data Flow
Language: Ada
Acronym: NIVL

Non-Initialized Variable

Variable other than local variable is not initialized before being read

Description

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;
```

Check Information

Group: Data Flow
Language: Ada
Acronym: NIV

Division by Zero

Division by zero occurs

Description

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
```

Check Information
Group: Numerical
Language: Ada
Acronym: ZDV

Arithmetic Exceptions

Input argument outside domain of mathematical function

Description

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

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

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 math function.

Check Information
Group: Other
Language: Ada
Acronym: EXCP

Scalar and Float Overflow

Arithmetic operation causes overflow

Description

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 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 maximum floating point value, *2*FValue* cannot be represented with the same type and so raises an overflow at line 30.

Check Information
Group: Numerical
Language: Ada
Acronym: OVFL

Correctness Condition

Mismatch occurs when access type is used

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)));
34  -- COR ERROR: Aggregate is not compatible with definition
35              end if;
36              IF OBJ3 = ((DISC => 7), (DISC => 7)) then
37  -- COR ERROR: Aggregate is not compatible with definition
38                  COMMENT ("PREVENTING DEAD VARIABLE OPTIMIZATION");
39              END IF;
40          END AGGDEF_INIT;
41
42          PROCEDURE MAIN IS
43          BEGIN
44              AGGDEF_INIT; -- NTC ERROR: propagation of COR ERROR
45          END MAIN;
45      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.

Check Information

Group: Other
Language: Ada
Acronym: COR

Power Arithmetic

Power interger or float function used with negative argument

Description

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*.

Check Information
Group: Numerical
Language: Ada
Acronym: POW

User Assertion

Assertion statement fails

Description

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.

Check Information
Group: Other
Language: Ada
Acronym: ASRT

Non Terminating Call

Called function does not return to calling context

Description

- 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 List** 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 on page 2-7 warning) at line 12 when *X = 0*.

- The second does not terminate because of an infinite loop (red `Non Terminating Loop`) 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-ada -lang 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-ada -lang95 -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 accepted meeting in the task body from the main (using `-main main.main`):

```
    main main.main):
    --package body main is
        procedure main is
        begin
        depend.control.cancel; -- 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 control is
        date : Integer := 0;
        init_date: Integer;
        begin
        loop
        select
        accept cancel;
        if (date = 0) then
        init_date := 10;
        end if ;
        date := init_date ;
    Text_Io.Put_Line ("Hello ....");
        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 math 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.

Check Information
Group: Control Flow
Language: Ada
Acronym: NTC

Non Terminating Loop

Loop does not terminate or contains an error

Description

- 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 List** 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.

Check Information
Group: Control Flow
Language: Ada
Acronym: NTL

Unreachable Code

Code cannot be reached during execution

Description

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.

Check Information
Group: Data Flow
Language: Ada
Acronym: UNR

Assumptions Used During Verification

Why Polyspace Verification Uses Approximations

In this section...
"What is Static Verification" on page 3-2
"Exhaustiveness" on page 3-2

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.

Procedure Calls with Default Parameters

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

Example

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

Explanation

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

COR checks at line 21 and 38 on `MessageFailure` represent verification made by Polyspace on the default assignment of a null string value on the input parameter.

Note Checks remain on the procedure definition except for the following basic types and values:

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

_INIT_PROC Procedures

In the Polyspace results, you might find nodes _INIT_PROC. As your compiler, Polyspace generates a function _INIT_PROC for each record where initialization occurs. When a package defines many records, each _INIT_PROC is differentiated by $I (I in 1.n).

Example

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

In the previous example, an unproven OVFL on the field a of record my_rec1 has been detected when initializing the global variable V1. It initializes record of global variable V1 at line 17. A random procedure could return any integer and lead to an overflow by adding to 2. The check is located in the _INIT_PROC node in the **Results List** view.

Expansion of Sizes

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

Example 3.1. Example

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

The `-variable-to-expand` option is used to specify aggregate variables (record, etc.) that will be split into independent variables for the purpose of verification. This option has an impact on the Global Data Dictionary results:

- Each variable specified in this option will have its fields verified separately;
- The data dictionary will distinguish fields accessed by different tasks.

The depth of the variable to expand is controlled by the `-variable-to-expand`.

Note Expansion options have an impact on the duration of a verification.

Inline Assemblers

Using Asm procedures, you inline assembly language instructions in your source code. You use the Asm_Output attribute to define the Outputs parameter of the Asm procedure. The general format of the attribute is

Type'Asm_Output (*constraint_string*, *var*)

var is variable that stores the result of the assembly language instruction in the Asm statement. *constraint_string* is a string such as "=r" that defines what kind of register is required for storing the variable.

Irrespective of the value of *constraint_string*, Polyspace for Ada assumes that after the Asm statement, *var* is initialized and has a full range of values allowed by its type.

For instance, after the following statement, Polyspace for Ada assumes that the variable Result is initialized and has full range of values.

```
Asm("and %0, %1, %2",
    Outputs => Integer'Asm_Output("=r", Result),
    Inputs => (Integer'Asm_Input("r", Key),
    Integer'Asm_Input("r", Mask)));
```

If a later instruction reads Result:

- The **Non-initialized variable** check on Result is green.
- On the **Source** pane, if you place your cursor on Result, the tooltip shows that Result has full range of values allowed by its type.

Volatile Variables

Problem

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

This axiom is:

"If I write a value V in the variable X and if I read X's value before another writing to X occurs, I will get V."

Explanation

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

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

```
function test return integer is
        random: Integer;
        pragma volatile (random);
        y: Integer;           -- random ~ [-2^31, 2^31-1] ,
                                   -- although random is not initialized
begin
        y:= 1 /random;        -- division and init orange
                                   -- because random
~     [-2^31, 2^31-1]
        random:= 100;
        y:= 1 /random;       -- division and init orange
                                   -- because random~ [-2^31,2^31-1]
        return random;       -- random ~ [-2^31, 2^31-1]
end;
```

Shared Variables

Abstract

My shared variables appear in orange in the variable dictionary.

Explanation

Polyspace Server™ for Ada does not make prior assumptions about the execution sequence of tasks. Specifically, shared variables are considered as unprotected.

Solution

You can use the following mechanisms to protect your variables.

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

Critical Sections

These are the most common protection mechanism in applications and they are simple to use in Polyspace Server for Ada:

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

Also refer to "Atomicity"

package my_tasking

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

package body my_tasking

```
    with pkutil; use pkutil;
package body my_tasking is
    procedure proc1 is
    begin
        begin_cs;
            X := 12; -- X is protected
```

```
            Y := 100;
        end_cs;
    end;
    procedure proc2 is
    begin
        begin_cs;
            X := 11; -- X is protected
        end_cs;
        Y := 101; -- Y is not protected
    end;
    procedure my_main is
    begin
        X := 0;
        Y := 0;
    end
end my_tasking;
```

package pkutil

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

package body pkutil

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

Launching command

```
polyspace-ada \
-main my_tasking.my_main \
-entry-points my_tasking.proc1,my_tasking.proc2 \
-critical-section-begin "pkutil.begin_cs:CS1" \
-critical-section-end "pkutil.end_cs:CS1"
```

Mutual Exclusion

Mutual exclusion between tasks or interrupts can be implemented while preparing Polyspace Server for Ada for launch setting.

Suppose there are entry-points which do not overlap with each other, and that variables are shared by nature.

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

These entry-points cannot overlap:

- t1 and t3

- t2, t3 and t4

These entry-points can overlap:

- t1 and t2
- t1 and t4

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

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

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

```
t1 t3
```

```
t2 t3 t4
```

Rendezvous

Polyspace Server for Ada takes the specified rendezvous into account. When the rendezvous are explicitly specified in the code, the software overrides other synchronization mechanisms specified through the -entry-points option.

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

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

```
polyspace-ada -main other_task.main
```

Semaphores

Although it is possible to implement in ada, it is not possible to take into account a semaphore system call in Polyspace Server for Ada. Nevertheless, Critical sections on page 3-9 may be used to model the behavior.

Pointers to Explicit Tasks

If a task type is used through a pointer, then Polyspace automatically adds two instances of this task type to the Polyspace execution model of your application. Task pointer objects that are used in your application are represented by these two instances. Polyspace uses these instances to simulate tasks associated with the execution of your application.

Consider the following example.

```
package Test is
   task type Ressource_T is
      entry Get (X : out integer);
      entry Set (X : in integer);
   end Ressource_T;
   type Ressource_Ptr_T is access Ressource_T;
   Ressource_Ptr : Ressource_Ptr_T;
   V : Integer := 0;
   function Alloc_Ressource return Ressource_Ptr_T;
   procedure Test_Ressource;
private
end Test;

package body Test is
   task body Ressource_T is
      Random : Boolean;
      pragma Volatile (Random);
   begin
      while Random loop
         select
            accept Get (X : out Integer) do
               X := V + 2;
            end Get;
         or
            accept Set(X : in Integer) do
               V := X - 2;
            end Set;
         end select;
      end loop;
   end Ressource_T;

   function Alloc_Ressource return Ressource_Ptr_T is
   begin
      return new Ressource_T;
   end Alloc_Ressource;

   procedure Test_Ressource is
     X : Integer;
     Tp : Ressource_Ptr_T;
   begin
      Tp := Alloc_Ressource;
      Tp.Get(X);
   end Test_Ressource;
end Test;
```

At the end of verification, in the **Results List** view, you see two instances of the task type Ressource_T, that is, PST_Ressource_T_1 and PST_Ressource_T_2.

3-13

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\ada_limitations.pdf

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

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

Code Metrics

Number of Files

Number of source files

Description

This metric calculates the number of source files in your project.

Metric Information
Group: Project
Acronym: ADA_FILES

See Also
Number of Packages

Number of Lines Without Comment

Number of code lines excluding comments

Description

This metric calculates the number of lines in a file. When calculating the value of this metric, Polyspace excludes comments and blank lines.

Metric Information

Group: Project
Acronym: ADA_LINES_WITHOUT_CMT

See Also

Number of Files

Number of Packages

Number of Ada packages

Description

This metric calculates the number of packages in a project.

Metric Information
Group: Project
Acronym: PACKAGES

See Also
Number of Packages in With Statements | Number of Files

Number of Packages in With Statements

Number of packages in `with` statement

Description

This metric calculates the number of packages included using `with` statements.

Metric Information
Group: Project
Acronym: PACKWITH

See Also
`Number of Subprograms in With Statements`|`Number of Packages`

Number of Potentially Unprotected Shared Variables

Number of unprotected shared variables

Description

This metric measures the number of variables with the following properties:

- The variable is used in more than one task.
- At least one operation on the variable is not protected from interruption by operations in other tasks.

For examples of potentially unprotected shared variables, see `Shared unprotected global variable`.

Metric Information
Group: Project
Acronym: UNPSHV

See Also
`Shared unprotected global variable`

Introduced in R2018b

Number of Protected Shared Variables

Number of protected shared variables

Description

This metric measures the number of variables with the following properties:

- The variable is used in more than one task.
- All operations on the variable are protected from interruption through critical sections or temporal exclusions.

For examples of protected shared variables, see `Shared protected global variable`.

Metric Information
Group: Project
Acronym: PSHV

See Also
`Shared protected global variable`

Introduced in R2018b

Number of Subprograms in With Statements

Number of subprograms in `with` statement

Description

This metric calculates the number of subprograms included in `with` statements.

Metric Information
Group: Project
Acronym: SUBPWITH

See Also
`Number of Packages in With Statements`

Global Variables

Shared protected global variable

Global variables shared between multiple tasks and protected from concurrent access by the tasks

Description

A **shared protected global variable** has the following properties:

- The variable is used in more than one task.
- Operations on the variable are protected from interruption through critical sections or temporal exclusion. The calls to functions beginning and ending a critical section must be reachable.

In code that is not intended for multitasking, all global variables are non-shared.

In your verification results, these variables are colored green on the **Source**, **Results List** and **Variable Access** panes.

Examples

Shared Variables Protected Through Temporal Exclusion

```
package example is

    -- Task declaration
    procedure Task1;
    procedure Task2;

    -- Public variables in package
    Shr1, Shr2 : INTEGER :=0;

    -- Public procedure in package
    procedure main;

end example;

package body example is

procedure Task1 is
  begin
    Shr1 := 1;
    Shr2 := 1;
  end Task1;

procedure Task2 is
  Tmp: INTEGER;
  begin
    Tmp := Shr1;
    Tmp := Shr2;
  end Task2;

procedure main is
    begin
    null;
```

```
    end main;

end example;
```

In this example, `Shr1` and `Shr2` are protected shared variables if you specify the following options:

Option	Value	
Main entry point	`example.main`	

Entry points	`example.Task1`
	
	`example.Task2`	

Temporally exclusive tasks	`example.Task1 example.Task2`	

Both variables are shared between `example.Task1` and `example.Task2`. However, because the two tasks are temporally exclusive, the variables are protected from concurrent access.

Shared Variables Protected Through Critical Sections

```
package example is

    -- Task declaration
    procedure Task1;
    procedure Task2;


    -- Public variables in package
    Shr1, Shr2 : INTEGER :=0;


    -- Public procedure in package
    procedure main;

    -- Critical sections
    procedure Begin_CS;
    procedure End_CS;

end example;

package body example is

 -- Critical sections
    procedure Begin_CS is
    begin
        null;
    end Begin_CS;

    procedure End_CS is
    begin
        null;
    end;

procedure Task1 is
  Tmp: INTEGER;
  begin
    Shr1 := 1;
    Begin_CS;
```

```
      Shr2 := 1;
    End_CS;
  end Task1;

procedure Task2 is
  Tmp: INTEGER;
  begin
    Tmp := Shr1;
    Begin_CS;
      Tmp := Shr2;
    End_CS;
  end Task2;

 procedure main is
   begin
    null;
   end main;

end example;
```

In this example, `Shr2` is a protected shared variable if you specify the following:

Option	Value	
Main entry point	**example.main**	
Entry points	example.Task1 example.Task2	
Critical section details	**Starting procedure**	**Ending procedure**
	example.Begin_CS	example.End_CS

The variable `Shr2` is shared between `example.Task1` and `example.Task2`. However, because operations on the variable are between calls to the starting and ending procedure of the same critical section, they cannot interrupt each other.

However, the variable `Shr1` is potentially unprotected.

See Also
Main entry point | Entry points | Critical section details | Temporally exclusive tasks

Shared unprotected global variable

Global variables shared between multiple tasks but not protected from concurrent access by the tasks

Description

A **shared unprotected global variable** has the following properties:

- The variable is used in more than one task.
- At least one operation on the variable is not protected from interruption by operations in other tasks.

In code that is not intended for multitasking, global variables are non-shared.

In your verification results, these variables are colored orange on the **Source**, **Results List** and **Variable Access** panes.

Examples

Unprotected Shared Variables

```
package example is

    -- Task declaration
    procedure Task1;
    procedure Task2;

    -- Public variables in package
    Shr1, Shr2 : INTEGER :=0;

    -- Public procedure in package
    procedure main;

end example;

package body example is

procedure Task1 is
  begin
    Shr1 := 1;
    Shr2 := 1;
  end Task1;

procedure Task2 is
  Tmp: INTEGER;
  begin
    Tmp := Shr1;
    Tmp := Shr2;
  end Task2;

procedure main is
    begin
     null;
```

```
    end main;

end example;
```

In this example, if you specify:

- `example.main` as main entry point,
- `example.Task1` and `example.Task2` as other entry points,

and do not specify protection mechanisms, `Shr1` and `Shr2` are potentially unprotected shared variables

See Also

Main entry point | Entry points | Critical section details | Temporally exclusive tasks

Non-shared used global variable

Global variables used in a single task

Description

A **non-shared used** global variable has the following properties:

- The variable is used only in a single task.
- At least one read or write operation is performed on it.

When determining how a variable is used, the software considers all operations, both reachable and unreachable. In code that is not intended for multitasking, all global variables are non-shared.

In your verification results, these variables are colored black on the **Results List** and **Variable Access** panes.

Non-shared unused global variable

Global variables declared but not used

Description

A **non-shared unused** global variable has the following properties:

- The variable is declared in the code.
- In code that can be reached at run time, the code does not perform read or write operations on it.

In your verification results, these variables are colored gray on the **Source**, **Results List** and **Variable Access** panes.

Examples

Variable Read in Unreachable Code

```
package example is

   procedure Task3;

   -- Public variables in package
   var: Integer;

   -- Public procedure in package
   procedure main;

end example;

package body example is

procedure Task3 is
  Tmp: INTEGER;
  begin
    if false then
        Tmp := var;
    end if;
  end Task3;

 procedure main is
   begin
    null;
   end main;

end example;
```

In this example, `var` is declared but an operation on `var` occurs only in deactivated code. Therefore, Polyspace considers that `var` is an unused variable.

Report Components

Acronym Definitions

Create table of Polyspace acronyms used in report and their full forms

Description

This component creates a table containing the acronyms used in the report and their full forms. Acronyms are used for Polyspace checks and result status.

See Also

Call Hierarchy

Create table showing call graph in source code

Description

This component creates a table showing the call hierarchy in your source code. For each function call in your source code, the table displays the following information:

- Level of call hierarchy, where the function is called.

  Each level is denoted by |. If a function call appears in the table as `|||->` *file_name*.*function_name*, the function call occurs at the third level of the hierarchy. Beginning from `main` or an entry point, there are three function calls leading to the current call.
- File containing the function call.

  In Code Prover, the line and column is also displayed.
- File containing the function definition.

  In Code Prover, the line and column where the function definition begins is also displayed.

In addition, the table also displays uncalled functions.

This table captures the information available on the **Call Hierarchy** pane in the Polyspace user interface.

See Also

Code and Verification Information

Create table of verification times and code characteristics

Description

This component creates tables containing verification times and code characteristics such as number of lines.

Properties

Include Verification Time Information

If you select this option, the report contains verification times broken down by phase.

- For Polyspace Bug Finder™, the phases are `compilation`, `pass0`, `pass1`, etc.
- For Polyspace Code Prover™, the phases are `compilation`, `global`, `function`, etc.

Include Code Details

If you select this option, the report contains the following code characteristics:

- Number of files
- Number of lines
- Number of lines without comment

See Also

Code Metrics Details

Create table of Polyspace metrics broken down by file and function

Description

This component creates a table containing metrics from a Polyspace project. The metrics appear broken down by file and function.

Properties

Project Metrics

If you select this option, the report contains the following metrics about the project:

- Number of direct recursions
- Number of files
- Number of headers
- Number of protected and unprotected shared variables

File Metrics

If you select this option, the report contains the following metrics about each file in the project:

- Estimated function coupling
- Lines without comment
- Comment density
- Total lines

Function Metrics

If you select this option, the report contains the following metrics about each function in the project:

- Cyclomatic complexity
- Language scope
- Lower and higher estimates of local variable size
- Number of lines within body
- Number of executable lines
- Number of `goto` statements
- Number of call levels
- Number of called functions
- Number of call occurrences
- Number of function parameters
- Number of paths
- Number of `return` statements

- Number of instructions
- Number of calling functions

See Also

Code Metrics Summary

Create table of Polyspace metrics

Description

This component creates a table containing metrics from a Polyspace project. The metrics are the same as those displayed under . However, the file and function metrics are not broken down by individual files and functions. Instead, the table provides the minimum and maximum value of a file metric over all files and a function metric over all functions.

See Also

Code Verification Summary

Create table of Polyspace analysis results

Description

This component creates tables containing the following results:

- Number of results
- Number of coding rule violations for each coding rule type such as MISRA C®
- Number of defects, for Polyspace Bug Finder results
- Number of checks of each color, for Polyspace Code Prover results
- Whether the project passed or failed the software quality objective

Properties

Include Checks from Polyspace Standard Library Stub Functions

Unless you deselect this option, the tables contain Polyspace Code Prover checks that appear in Polyspace stubs for the standard library functions.

See Also

Configuration Parameters

Create table of analysis options, assumptions and coding rules configuration

Description

This component creates the following tables:

- *Polyspace settings*: The analysis options that you used to obtain your results. The table lists command-line version of the options along with their values.
- *Analysis assumptions*: The assumptions used to obtain your Code Prover results. The table lists only the modifiable assumptions. For assumptions that you cannot change, see the Polyspace documentation.
- *Coding rules configuration*: The coding rules whose violations you checked for. The table lists the rule number, rule description and other information about the rules.
- *Files with compilation errors*: If your project has source files with compilation errors, these files are listed.

See Also

Global Variable Checks

Create table of global variables (Code Prover only)

Description

This component creates a table of Polyspace Code Prover global variables. From this table, you can see the number of global variables of each type.

See Also

Recursive Functions

Create table of recursive functions

Description

This component creates a table containing the recursive functions in your source code (along with the files containing the functions).

- For each direct recursion (function calling itself directly), the table lists the recursive function.
- For each indirect recursion cycle (function calling itself through other functions), the table lists one function in the cycle.

For instance, the following code contains two indirect recursion cycles.

```
volatile int signal;

void operation1() {
    int stop = signal%2;
    if(!stop)
        operation1_1();
}


void operation1_1() {
    operation1();
}

void operation2() {
    int stop = signal%2;
    if(!stop)
        operation2_1();
}

void operation2_1() {
    operation2();
}
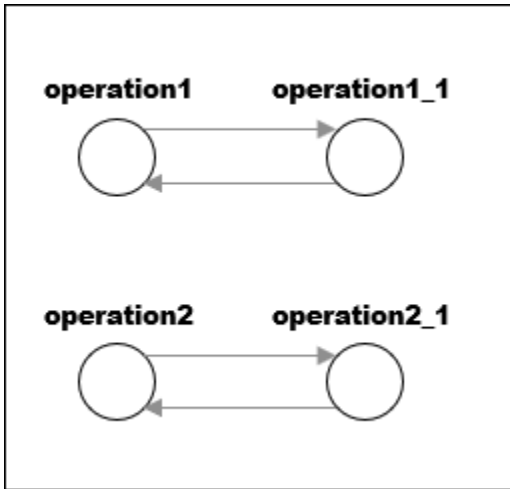
void main(){
    operation1();
    operation2();
}
```

The two call graph cycles are:

- operation1 → operation1_1 → operation1
- operation2 → operation2_1 → operation2

This report component shows one function from each of the two cycles: `operation1` and `operation2`. To see the full cycle, open the results in the Polyspace user interface.

See Also

Report Customization (Filtering)

Create filters that apply to your Polyspace reports

Description

This component allows you to filter unwanted information from existing Polyspace report templates. To apply global filters, place this component immediately below the node representing the report name.

Properties

Code Metrics Filters

The properties in table below apply to the inclusion of code metrics in your report.

Property	Purpose	User Action
Include Project Metrics	Choose whether to include metrics about your Polyspace project.	Select the check box to include project metrics.
Project metrics to include	Specify project metrics to include or exclude from report.	Enter a MATLAB regular expression.
Include File Metrics	Choose whether to include per file metrics in report.	Select the check box to include per file metrics.
File Metrics > Files to include	Specify files to include or exclude when reporting file metrics.	Enter a MATLAB regular expression.
File metrics to include	Specify file metrics to include or exclude from report.	Enter a MATLAB regular expression.
Include Function Metrics	Choose whether to include per function metrics in report.	Select the check box to include per function metrics.
Function Metrics > Files to include	Specify files to include or exclude when reporting function metrics.	Enter a MATLAB regular expression.
Functions to include	Specify functions to include or exclude when reporting function metrics.	Enter a MATLAB regular expression.
Function metrics to include	Specify function metrics to include or exclude from report.	Enter a MATLAB regular expression.

Coding Rules Filters

The properties in table below apply to the inclusion of coding rule violations in your report.

Property	Purpose	User Action
Files to include	Specify files to include or exclude when reporting coding rule violations.	Enter a MATLAB regular expression.
Coding rule numbers to include	Specify coding rules to include or exclude when reporting coding rule violations.	Enter a MATLAB regular expression.
Classifications to include	Specify classifications to include or exclude when reporting coding rule violations.	Enter a MATLAB regular expression.
Status types to include	Specify statuses to include or exclude when reporting coding rule violations.	Enter a MATLAB regular expression.

Run-time Check Filters

The properties in table below apply to the inclusion of Polyspace Code Prover checks in your report.

Property	Purpose
Red Checks	Specify whether to include red checks in your report. Red checks indicate proven run-time errors.
Gray Checks	Specify whether to include gray checks in your report. Gray checks indicate unreachable code.
Orange Checks	Specify whether to include orange checks in your report. Orange checks indicate possible run-time errors.
Green Checks	Specify whether to include green checks in your report. Green checks indicate that an operation does not contain a specific run-time error.
Inspection Point Checks	Specify whether to include inspection point checks in your report. These checks allow an user to find the values that a variable can take at a certain point in the code.
Unreachable Functions	Specify whether to include unreachable functions in your report.

Advanced Filters

The properties in table below apply to the inclusion of metrics, coding rule violations and Polyspace Code Prover checks in your report.

Property	Purpose	User Action
Justification status	Choose whether to report only justified checks, only unjustified checks or all checks.	Choose an option from the dropdown list.

Property	Purpose	User Action
Files to include	Specify files to include or exclude from your report.	Enter a MATLAB regular expression.
Check types to include	Specify Polyspace Code Prover checks to include in your report.	Enter a MATLAB regular expression.
Function names to include	Specify functions to include or exclude from your report.	Enter a MATLAB regular expression.
Classification types to include	Specify classifications to include or exclude from your report.	Enter a MATLAB regular expression.
Status types to include	Specify statuses to include or exclude from your report.	Enter a MATLAB regular expression.
Comments to include	Specify comments to include or exclude from your report.	Enter a MATLAB regular expression.

See Also

Topics
"Regular Expressions" (MATLAB)

Run-time Checks Details Ordered by Color/File

Create overrides for global filters in Polyspace reports (Code Prover only)

Description

This component adds detailed information about the run-time checks to your report. This component can also be used to override global filters in specific chapters of your report. Use the following workflow when using filters in your report:

1   To create filters that apply to all chapters of your report, use the **Report Customization (Filtering)** component. For more information, see `Report Customization (Filtering)`.
2   To override some of the filters in individual chapters, use the **Run-time Checks Details Ordered by Color/File** component. Select the **Override Global Report filter** box.

Properties

Categories To Include

The properties in table below apply to the inclusion of Polyspace Code Prover checks in your report.

Property	Purpose
Red Checks	Specify whether to include red checks in your report. Red checks indicate proven run-time errors.
Gray Checks	Specify whether to include gray checks in your report. Gray checks indicate unreachable code.
Orange Checks	Specify whether to include orange checks in your report. Orange checks indicate possible run-time errors.
Green Checks	Specify whether to include green checks in your report. Green checks indicate that an operation does not contain a specific run-time error.
Inspection Point Checks	Specify whether to include inspection point checks in your report. These checks allow an user to find the values that a variable can take at a certain point in the code.
Unreachable Functions	Specify whether to include unreachable functions in your report.

Advanced Filters

The properties in table below apply to the inclusion of metrics, coding rule violations and Polyspace Code Prover checks in your report.

Property	Purpose	User Action
Justification status	Choose whether to report only justified checks, only unjustified checks or all checks.	Choose an option from the dropdown list.
Files to include	Specify files to include or exclude from your report.	Enter a regular MATLAB expression.
Check types to include	Specify Polyspace Code Prover checks to include in your report.	Enter a regular MATLAB expression.
Function names to include	Specify functions to include or exclude from your report.	Enter a regular MATLAB expression.
Classification types to include	Specify classifications to include or exclude from your report.	Enter a regular MATLAB expression.
Status types to include	Specify statuses to include or exclude from your report.	Enter a regular MATLAB expression.
Comments to include	Specify comments to include or exclude from your report.	Enter a regular MATLAB expression.

See Also

Run-time Checks Details Ordered by Review Information

Create table with run-time checks ordered by review information (Code Prover only)

Description

This component creates tables displaying the Polyspace Code Prover checks in your code. All checks with same combination of **Severity** and **Status** appear in the same table.

See Also

Run-time Checks Summary Ordered by File

Create table with run-time checks ordered by file (Code Prover only)

Description

This component creates a table displaying the number of Polyspace Code Prover checks per file in your code.

Properties

Sort the data

Use this option to sort the rows in the table alphabetically by filename or by percentage of unproven code.

Display as

Use this option to display the number of checks in a table or in bar charts.

Display ratio of checks in a file

Select this option to display the number of checks of a certain color as a ratio of total number of checks in the file.

Include checks from Polyspace standard library stub functions

Select this option to include the checks from Polyspace standard library stub functions in your display.

See Also

Software Quality Objectives - Run-time Checks Details

Create table of result details for results downloaded from Polyspace Metrics

Description

This component creates tables showing results downloaded from Polyspace Metrics.

The component `Software Quality Objectives - Run-time Checks Summary` shows the distribution of results. This component shows individual instances of results. Each file has a dedicated table showing the findings in the file.

See Also

Software Quality Objectives - Run-time Checks Summary

Create table of results summary for results downloaded from Polyspace Metrics

Description

This component creates a table showing the distribution of run-time checks in results downloaded from Polyspace Metrics.

This component shows the distribution of run-time checks. The component `Software Quality Objectives - Run-time Checks Details` shows the individual instances of run-time checks.

See Also

Summary By File

Create table showing summary of Polyspace results by file

Description

This component creates a table showing a breakdown of Polyspace results by file.

See Also

Variable Access

Create table showing global variable access in source code (Code Prover only)

Description

This component creates a table showing the global variable access in your source code. For each global variable, the table displays the following information:

- Variable name.

  The entry for each variable is denoted by |.
- Type of the variable.
- Number of read and write operations on the variable.
- Details of read and write operations. For each read or write operation, the table displays the following information:

  - File and function containing the operation in the form *file_name.function_name*.

    The entry for each read or write operation is denoted by ||. Write operations are denoted by < and read operations by >.
  - Line and column number of the operation.

This table captures the information available on the **Variable Access** pane in the Polyspace user interface.

The table showing variable access contains only the names of files. Below this table, a second table shows the full paths to files (in two columns, **Filename** and **Full filename**). If a variable access occurs in a Standard library function, the two columns contain this information:

- **Filename**: `__polyspace__stdstubs.c` (the file containing Polyspace implementation of Standard Library functions)
- **Full filename**: `Std library`

See Also

Variable Checks Details Ordered By Review Information

Create table with global variable results ordered by review information (Code Prover only)

Description

This component creates tables displaying the Polyspace Code Prover global variable results in your code. All checks with same combination of **Severity** and **Status** appear in the same table.

See Also