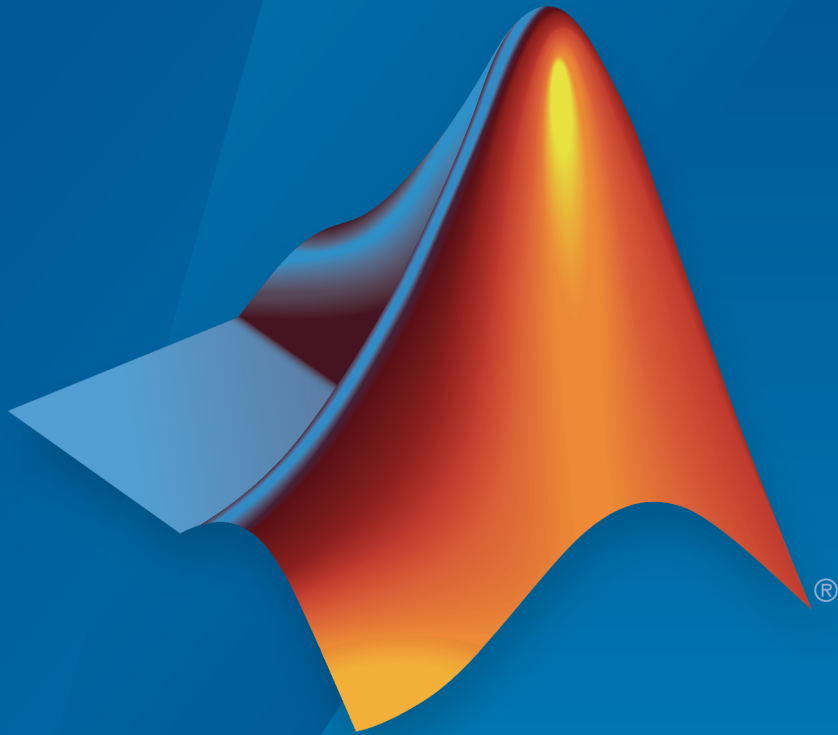


# Polyspace<sup>®</sup> Products for Ada Reference



®

R2016a

 MathWorks<sup>®</sup>

## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*Polyspace<sup>®</sup> Products for Ada Reference*

© COPYRIGHT 1999–2016 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

**FEDERAL ACQUISITION:** This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

### **Trademarks**

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://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](http://www.mathworks.com/patents) for more information.

### **Revision History**

March 2009	Online Only	Revised for Version 5.3 (Release 2009a)
September 2009	Online Only	Revised for Version 5.4 (Release 2009b)
March 2010	Online Only	Revised for Version 5.5 (Release 2010a)
September 2010	Online Only	Revised for Version 6.0 (Release 2010b)
April 2011	Online Only	Revised for Version 6.1 (Release 2011a)
September 2011	Online Only	Revised for Version 6.2 (Release 2011b)
March 2012	Online Only	Revised for Version 6.3 (Release 2012a)
September 2012	Online Only	Revised for Version 6.4 (Release 2012b)
March 2013	Online Only	Revised for Version 6.5 (Release 2013a)
September 2013	Online Only	Revised for Version 6.6 (Release 2013b)
March 2014	Online Only	Revised for Version 6.7 (Release 2014a)
October 2014	Online Only	Revised for Version 6.8 (Release 2014b)
March 2015	Online Only	Revised for Version 6.9 (Release 2015a)
September 2015	Online Only	Revised for Version 6.10 (Release 2015b)
March 2016	Online Only	Revised for Version 6.11 (Release 2016a)



## Option Descriptions

<b>Send to Polyspace Server</b> .....	1-4
Settings .....	1-4
Tips .....	1-4
Command-Line Information .....	1-4
<b>Add to results repository</b> .....	1-5
Settings .....	1-5
Dependency .....	1-5
Command-Line Information .....	1-5
<b>Source code language</b> .....	1-6
Settings .....	1-6
Command-Line Information .....	1-6
<b>Target operating system</b> .....	1-7
Settings .....	1-7
Command-Line Information .....	1-7
See Also .....	1-8
<b>Target processor type</b> .....	1-9
Settings .....	1-9
Command-Line Information .....	1-9
See Also .....	1-9
<b>Files extensions</b> .....	1-10
Settings .....	1-10
Command-Line Information .....	1-10
<b>Remove ambiguities in comparison operators</b> .....	1-11
Settings .....	1-11
Command-Line Information .....	1-12
See Also .....	1-12

<b>Value of the constant Storage_Unit</b> .....	<b>1-13</b>
Settings .....	1-13
Command-Line Information .....	1-13
See Also .....	1-13
<b>Preprocessor definitions</b> .....	<b>1-14</b>
Settings .....	1-14
Command-Line Information .....	1-15
See Also .....	1-15
<b>Disable preprocessor definitions</b> .....	<b>1-17</b>
Settings .....	1-17
Command-Line Information .....	1-17
See Also .....	1-17
<b>Command/script to apply before start of the code</b>	
<b>verification</b> .....	<b>1-18</b>
Settings .....	1-18
Command-Line Information .....	1-19
See Also .....	1-19
<b>Include folders</b> .....	<b>1-20</b>
Settings .....	1-20
Command-Line Information .....	1-20
<b>Verify whole application</b> .....	<b>1-21</b>
Settings .....	1-21
Command-Line Information .....	1-21
<b>Main entry point</b> .....	<b>1-22</b>
Settings .....	1-22
Dependencies .....	1-22
Command-Line Information .....	1-22
<b>Multitasking</b> .....	<b>1-23</b>
Settings .....	1-23
Command-Line Information .....	1-23
<b>Entry points</b> .....	<b>1-24</b>
Settings .....	1-24
Dependencies .....	1-24
Tips .....	1-24
Command-Line Information .....	1-24

<b>Critical section details</b> .....	<b>1-26</b>
Settings .....	1-26
Dependencies .....	1-26
Command-Line Information .....	1-26
<b>Temporally exclusive tasks</b> .....	<b>1-28</b>
Settings .....	1-28
Dependencies .....	1-28
Command-Line Information .....	1-28
<b>Verify module</b> .....	<b>1-29</b>
Settings .....	1-29
Tips .....	1-30
Command-Line Information .....	1-30
<b>Verify files independently</b> .....	<b>1-31</b>
Settings .....	1-31
Dependencies .....	1-31
Tips .....	1-31
Command-Line Information .....	1-31
<b>Common source files</b> .....	<b>1-33</b>
Settings .....	1-33
Command-Line Information .....	1-33
<b>Variable/function range setup</b> .....	<b>1-34</b>
Settings .....	1-34
Command-Line Information .....	1-34
<b>No automatic stubbing</b> .....	<b>1-35</b>
Settings .....	1-35
Tips .....	1-35
Dependencies .....	1-35
Command-Line Information .....	1-35
<b>Initialization of uninitialized global variables</b> .....	<b>1-37</b>
Settings .....	1-37
Dependencies .....	1-37
Command-Line Information .....	1-37
<b>Continue after noninitialized variables</b> .....	<b>1-39</b>
Settings .....	1-39
Tips .....	1-39

Command-Line Information .....	1-39
<b>Continue with noninitialized in/out parameters .....</b>	<b>1-41</b>
Settings .....	1-41
Command-Line Information .....	1-41
<b>Treat import as nonvolatile .....</b>	<b>1-42</b>
Settings .....	1-42
Command-Line Information .....	1-42
<b>Treat export as nonvolatile .....</b>	<b>1-43</b>
Settings .....	1-43
Command-Line Information .....	1-43
<b>Precision level .....</b>	<b>1-44</b>
Settings .....	1-44
Command-Line Information .....	1-44
<b>Verification level .....</b>	<b>1-45</b>
Settings .....	1-45
Command-Line Information .....	1-45
<b>Verification time limit .....</b>	<b>1-47</b>
Settings .....	1-47
Command-Line Information .....	1-47
<b>Sensitivity context .....</b>	<b>1-48</b>
Settings .....	1-48
Command-Line Information .....	1-48
<b>Improve precision of interprocedural analysis .....</b>	<b>1-49</b>
Settings .....	1-49
Tips .....	1-49
Command-Line Information .....	1-49
<b>Specific precision .....</b>	<b>1-50</b>
Settings .....	1-50
Command-Line Information .....	1-50
<b>Max size of global array variables .....</b>	<b>1-51</b>
Settings .....	1-51
Command-Line Information .....	1-51



<b>Variables to expand</b> .....	<b>1-52</b>
Settings .....	<b>1-52</b>
Dependencies .....	<b>1-52</b>
Command-Line Information .....	<b>1-52</b>
<b>Expansion limit for a structured variable</b> .....	<b>1-53</b>
Settings .....	<b>1-53</b>
Dependencies .....	<b>1-53</b>
Command-Line Information .....	<b>1-53</b>
<b>Generate report</b> .....	<b>1-55</b>
Settings .....	<b>1-55</b>
Tips .....	<b>1-55</b>
Command-Line Information .....	<b>1-55</b>
<b>Report template</b> .....	<b>1-56</b>
Settings .....	<b>1-56</b>
Dependencies .....	<b>1-57</b>
Command-Line Information .....	<b>1-57</b>
<b>Output format</b> .....	<b>1-58</b>
Settings .....	<b>1-58</b>
Tips .....	<b>1-58</b>
Dependencies .....	<b>1-58</b>
Command-Line Information .....	<b>1-58</b>
<b>Disable code metrics generation</b> .....	<b>1-59</b>
Settings .....	<b>1-59</b>
Command-Line Information .....	<b>1-59</b>
<b>Command/script to apply after the end of the code</b>	
<b>verification</b> .....	<b>1-60</b>
Settings .....	<b>1-60</b>
Command-Line Information .....	<b>1-60</b>
<b>Other</b> .....	<b>1-61</b>
Settings .....	<b>1-61</b>
Command-Line Information .....	<b>1-61</b>
See Also .....	<b>1-61</b>
<b>-ada-include-dir</b> .....	<b>1-62</b>
<b>-author name</b> .....	<b>1-63</b>

<b>-server</b> <i>server_name_or_ip[:port_number]</i> .....	1-64
<b>-h[elp]</b> .....	1-65
<b>-v   -version</b> .....	1-66
<b>-sources</b> .....	1-67
<b>-sources-list-file</b> .....	1-68
<b>-from</b> .....	1-69
Settings .....	1-69
Command-Line Information .....	1-69
<b>-report-output-name</b> .....	1-70
Settings .....	1-70
Command-Line Information .....	1-70
<b>-less-range-information</b> .....	1-71
Settings .....	1-71
Command-Line Information .....	1-71
<b>-import-comments</b> .....	1-72
Command-Line Information .....	1-72
<b>-tmp-dir-in-results-dir</b> .....	1-73
<b>-max-processes</b> .....	1-74
Settings .....	1-74
Command-Line Information .....	1-74

## Check Descriptions

# 2

<b>Function Not Reachable: FNR</b> .....	2-2
<b>Non-Initialized Local Variable: NIVL</b> .....	2-3
Examples .....	2-3
Pragma Interface/Import .....	2-5
Type Access Variables .....	2-6

Address Clauses .....	2-6
<b>Non-Initialized Variable: NIV</b> .....	2-7
Example .....	2-7
<b>Division by Zero: ZDV</b> .....	2-8
<b>Arithmetic Exceptions: EXCP</b> .....	2-9
Ada Example .....	2-9
Explanation .....	2-11
<b>Scalar and Float Overflow: OVFL</b> .....	2-12
Ada Example .....	2-12
Explanation .....	2-13
<b>Correctness Condition: COR</b> .....	2-14
Attributes Check .....	2-14
Array Length Check .....	2-16
DIGITS Value Check .....	2-18
DELTA Value Length Check .....	2-18
Static Range and Values Check .....	2-19
Discriminant Check .....	2-21
Component Check .....	2-22
Dimension Versus Definition Check .....	2-23
Aggregate Versus Definition Check .....	2-24
Aggregate Array Length Check .....	2-25
Sub-Aggregates Dimension Check .....	2-26
Characters Check .....	2-28
Accessibility Level on Access Type .....	2-29
Accessibility of a Tagged Type .....	2-30
Explicit Dereference of a Null Pointer .....	2-32
<b>Power Arithmetic: POW</b> .....	2-33
Ada Example .....	2-33
Explanation .....	2-34
<b>User Assertion: ASRT</b> .....	2-35
Ada Example .....	2-35
Explanation .....	2-36
<b>Non Terminating Call: NTC</b> .....	2-37
Solution .....	2-37
Non Termination of Call: NTC .....	2-38

Non Termination of Call Due to Entry in Tasks . . . . .	2-39
Sqrt, Sin, Cos, and Generic Elementary Functions . . . . .	2-41
<b>Non Terminating Loop: NTL</b> . . . . .	2-43
Non Termination of Loop: NTL . . . . .	2-43
<b>Unreachable Code: UNR</b> . . . . .	2-46
Ada Example . . . . .	2-46
Explanation . . . . .	2-47

## Assumptions Used During Verification

# 3

<b>Why Polyspace Verification Uses Approximations</b> . . . . .	3-2
What is Static Verification . . . . .	3-2
Exhaustiveness . . . . .	3-3
<b>Procedure Calls with Default Parameters</b> . . . . .	3-4
Example . . . . .	3-4
Explanation . . . . .	3-5
<b>_INIT_PROC Procedures</b> . . . . .	3-6
Example . . . . .	3-6
<b>Expansion of Sizes</b> . . . . .	3-7
<b>Inline Assemblers</b> . . . . .	3-8
<b>Volatile Variables</b> . . . . .	3-9
Problem . . . . .	3-9
Explanation . . . . .	3-9
<b>Shared Variables</b> . . . . .	3-10
Abstract . . . . .	3-10
Explanation . . . . .	3-10
Solution . . . . .	3-10
Critical Sections . . . . .	3-10
Mutual Exclusion . . . . .	3-12
Rendezvous . . . . .	3-13
Semaphores . . . . .	3-13

<b>Pointers to Explicit Tasks</b> .....	<b>3-14</b>
<b>Limitations of Polyspace Verification</b> .....	<b>3-16</b>

**4** | Code Metrics — Alphabetical List

**5** | Global Variables— Alphabetical List

**6** | Polyspace Report Components — Alphabetical List



# Option Descriptions

---

- “Send to Polyspace Server” on page 1-4
- “Add to results repository” on page 1-5
- “Source code language” on page 1-6
- “Target operating system” on page 1-7
- “Target processor type” on page 1-9
- “Files extensions” on page 1-10
- “Remove ambiguities in comparison operators” on page 1-11
- “Value of the constant Storage\_Unit” on page 1-13
- “Preprocessor definitions” on page 1-14
- “Disable preprocessor definitions” on page 1-17
- “Command/script to apply before start of the code verification” on page 1-18
- “Include folders” on page 1-20
- “Verify whole application” on page 1-21
- “Main entry point” on page 1-22
- “Multitasking” on page 1-23
- “Entry points” on page 1-24
- “Critical section details” on page 1-26
- “Temporally exclusive tasks” on page 1-28
- “Verify module” on page 1-29
- “Verify files independently” on page 1-31
- “Common source files” on page 1-33
- “Variable/function range setup” on page 1-34
- “No automatic stubbing” on page 1-35
- “Initialization of uninitialized global variables” on page 1-37
- “Continue after noninitialized variables” on page 1-39

- “Continue with noninitialized in/out parameters” on page 1-41
- “Treat import as nonvolatile” on page 1-42
- “Treat export as nonvolatile” on page 1-43
- “Precision level” on page 1-44
- “Verification level” on page 1-45
- “Verification time limit” on page 1-47
- “Sensitivity context” on page 1-48
- “Improve precision of interprocedural analysis” on page 1-49
- “Specific precision” on page 1-50
- “Max size of global array variables” on page 1-51
- “Variables to expand” on page 1-52
- “Expansion limit for a structured variable” on page 1-53
- “Generate report” on page 1-55
- “Report template” on page 1-56
- “Output format” on page 1-58
- “Disable code metrics generation” on page 1-59
- “Command/script to apply after the end of the code verification” on page 1-60
- “Other” on page 1-61
- “-ada-include-dir” on page 1-62
- “-author name” on page 1-63
- “-server *server\_name\_or\_ip[:port\_number]*” on page 1-64
- “-h[elp]” on page 1-65
- “-v | -version” on page 1-66
- “-sources” on page 1-67
- “-sources-list-file” on page 1-68
- “-from” on page 1-69
- “-report-output-name” on page 1-70
- “-less-range-information” on page 1-71
- “-import-comments” on page 1-72
- “-tmp-dir-in-results-dir” on page 1-73



- “-max-processes” on page 1-74

## Send to Polyspace Server

Specify whether verification runs on the server or client system

### Settings

**Default:** On

On

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

Off

Run verification on the client system

### Tips

- Specifying this option in the 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” on page 1-5

## Add to results repository

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

### Settings

**Default:** Off

On

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

Off

Analysis results are stored locally.

### Dependency

This option is available only for remote verifications.

### Command-Line Information

**Parameter:** `-add-to-results-repository`

**Default:** `off`

**Example:** `polyspace-remote-ada -server IPaddress -add-to-results-repository`

### See Also

“Specify Analysis Options” | “Send to Polyspace Server” on page 1-4

## Source code language

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 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 Sources and Includes”.

#### Default: `no-predefined-OS`

`no-predefined-OS`

No operating system (with implementation-specific declarations in Ada Standard Libraries) specified

`gnat`

GCC Ada95

`greenhills`

Greenhills<sup>®</sup> Software real-time operating system (RTOS)

`rational`

IBM<sup>®</sup> Rational<sup>®</sup> Apex compiler

`aonix`

Aonix<sup>®</sup> 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”

## Target processor type

Specify the target processor type.

### Settings

**Default:** i386

i386

Intel<sup>®</sup> 80386 (i386) processor

sparc

Sun<sup>®</sup> Microsystems SPARC<sup>®</sup> processor

m68k

Freescale<sup>™</sup> ColdFire<sup>®</sup> m68k processor

1750a

MIL-STD-1750A 16-bit instruction set architecture

powerpc64bit

PowerPC<sup>®</sup> 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 in the **Include** folder of your project. Package specification files contain definitions and declarations referenced by your Ada body files. The software assumes that body files and the corresponding package specification files have the same names except for the extensions.

### Settings

**Default:** \*.ad[sa]

### Command-Line Information

**Parameter:** -extensions-for-spec-files

**Type:** string

**Value:** Valid file extensions

**Default:** \*.ad[sa]



## Remove ambiguities in comparison operators

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

In the following code:

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

Package OTHER_IABC_ADA_4 is
  procedure Main;
end OTHER_IABC_ADA_4;

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

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

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

### Settings

**Default:** Off

On

Remove ambiguities.

Off

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

### **Command-Line Information**

Parameter: `-base-type-directly-visible`

### **See Also**

“Common Compile Errors”

## Value of the constant `Storage_Unit`

Specify a positive value for `SYSTEM.Storage_Unit`.

### Settings

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

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

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

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

### Command-Line Information

**Parameter:** `-storage-unit`

**Value:** Integer

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

### See Also

“Common Compile Errors”

## Preprocessor definitions

Define compiler flags for compilation of preprocessor macros.

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

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

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

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


*expression* can be one of the following:


- *compiler\_flag*
- *compiler\_flag*="value"
- not (*expression*)
- *expression* and *expression*
- *expression* or *expression*
- *expression* and then *expression*
- *expression* or else *expression*

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

## Settings

### No Default

- To define a compiler flag, in the Defined Preprocessor Macros dialog box, enter:  
*compiler\_flag*="value"  
Then, click the **Adds this item to the list** button .
- Omitting the flag value is equivalent to specifying *compiler\_flag*="True".

- Flag values are case-insensitive strings.
- To remove a compiler flag from the list, in the Defined Preprocessor Macros dialog box, select the compiler flag. Then, click the button .
- Consider the following example.

```
with Apex_Processes;
with Apex_Types;

package Lift_Load_Control_Process_P is

    procedure Start_S;

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

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

If you specify `VEROCODE="True"`, then Polyspace does not verify code associated with the `#else` and `#end if` parts of the `if` statement. You will still see this code when you view results in the 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” on page 1-17



- “Specifying Target & Compiler Parameters”

## Disable preprocessor definitions

Nullify (undefine) macro compiler flags during compilation phase

### Settings

#### No Default

- In the Undefined Preprocessor Macros dialog box, enter *compiler\_flag*. Then click the **Adds this item to the list** button .
- Nullifying a macro compiler flag is equivalent to specifying in **Defined Preprocessor Macros** *compiler\_flag="False"*.
- To remove a compiler flag from the list, in the Undefined Preprocessor Macros dialog box, select the compiler flag. Then, click the button .
- As in the command line with compilers, you must specify only one flag for each `-U` option. However, you can use this option several times.

### Command-Line Information

**Parameter:** `-U`

**Type:** string

**Shell script example:**

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

### See Also

- “Preprocessor definitions” on page 1-14
- “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.

### Settings

#### No Default

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

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

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

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

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

- If you are running Polyspace software Version 5.1 (r2008a) or later on a Windows<sup>®</sup> system, you cannot use Cygwin<sup>™</sup> shell scripts. Cygwin is not with Polyspace software, so all files must be executable by Windows. To support scripting, the Polyspace installation includes Perl:

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

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

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



## **Command-Line Information**

**Parameter:** -pre-analysis-command

**Type:** string

**Value:** Script file name or command

## **See Also**

“Specifying Target & Compiler Parameters”

## Include folders

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.

### 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” on page 1-22 | “Entry points” on page 1-24

### Related Examples

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

### More About

- “Main Generator Overview”

## Main entry point

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

### Settings

Enter procedure name.

### Dependencies

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

### Command-Line Information

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

**Parameter:** `-main`

**Value:** Procedure name

**Example:** `polyspace-ada -sources filename -main mainpackage.init`

### See Also

“Verify whole application” on page 1-21 | “Entry points” on page 1-24

### Related Examples

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

### More About

- “Main Generator Overview”

# Multitasking

Specify whether the code is intended for a multitasking application.

## Settings

**Default:** Off

On

The code is intended for a multitasking application.

Off

The code is not intended for a multitasking application. Polyspace verifies only those functions that are called by the “Main entry point” on page 1-22.

## 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” on page 1-24 | “Critical section details” on page 1-26 |  
“Temporally exclusive tasks” on page 1-28

## Related Examples

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

## More About


- “Priorities”
- “Polyspace Software Assumptions”

## Entry points

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

### Settings

#### 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” on page 1-22 | “Critical section details” on page 1-26 |  
“Temporally exclusive tasks” on page 1-28

### Related Examples

- “Specify Analysis Options”

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

### **More About**


- “Priorities”
- “Polyspace Software Assumptions”

## Critical section details

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

### Settings

#### 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” on page 1-24 | “Temporally exclusive tasks” on page 1-28

### Related Examples

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



## **More About**


- “Shared Variables” on page 3-10

## Temporally exclusive tasks

Specify the tasks that do not execute simultaneously. 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” on page 1-24 | “Critical section details” on page 1-26

### Related Examples

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

### More About

- “Shared Variables” on page 3-10

## Verify module

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

### Settings

Default: On

On

Polyspace generates a `main` subprogram.

Entities	Action of generated <code>main</code>
Procedures and functions	<p>The generated <code>main</code> calls procedures and functions that are:</p> <ul style="list-style-type: none"> <li>Specified in package declarations in the code. If a package declaration is inside a procedure or another package, the generated <code>main</code> does not call procedures in the declaration.</li> <li>Specified outside a package.</li> </ul> <p>For these procedures and functions, Polyspace initializes the <code>in</code> and <code>out</code> parameters with random values.</p> <p>The generated <code>main</code> does not call procedures and functions that are already called in the code.</p>
Global variables	<p>The generated <code>main</code> assigns a random value to global variables that are specified in the :</p> <ul style="list-style-type: none"> <li>Package declaration.</li> </ul> <p>If you do not initialize a global variable before reading it in the package body, Polyspace generates an orange <b>Non-initialized variable</b> 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.</p> <ul style="list-style-type: none"> <li>Package body.</li> </ul> <p>If you do not initialize a global variable before reading it in the package body, Polyspace generates a red <b>Non-initialized variable</b> check if you read the variable in the package body.</p>

Entities	Action of generated main
	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” on page 1-21 | “Main entry point” on page 1-22 | “Initialization of uninitialized global variables” on page 1-37

## Related Examples

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

## Verify files independently

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” on page 1-33

## **Related Examples**

- “Specify Analysis Options”
- “Run File-by-File Local Verification”

## **More About**



- “Multiple File Error in File by File Verification”

## Common source files

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

### Settings

#### No Default

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

### Command-Line Information

**Parameter:** `-unit-by-unit-common-source`

**Value:** Full path to file

**Example:** `polyspace-ada -sources filename -unit-by-unit -unit-by-unit-common-source "C:/polyspace/function.adb"`

### See Also

“Verify files independently” on page 1-31

### Related Examples


- “Specify Analysis Options”
- “Run File-by-File Local Verification”

## Variable/function range setup

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

### Settings

#### 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 “DRS Text File Format”.

### Command-Line Information

**Parameter:** `-data-range-specifications`

**Value:** Full path to Data Range Specifications template file

**Example:** `polyspace-ada -sources filename -data-range-specifications "C:\Polyspace\drs.txt"`

### See Also

“Initialization of uninitialized global variables” on page 1-37 | “No automatic stubbing” on page 1-35

### Related Examples

- “Specify Analysis Options”
- “Specifying Constraints Using Text Files”
- “Performing Efficient Module Testing with DRS”
- “Reducing Orange Checks with DRS”



## No automatic stubbing

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

### Settings

**Default:** Off

On

Polyspace displays a list of undefined procedures and stops verification.

Off

Polyspace stubs undefined procedures only.

### Tips

Use this option when:

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

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

### Dependencies

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

### Command-Line Information

Parameter: `-no-automatic-stubbing`

### Related Examples

- “Specify Analysis Options”
- “Manual vs. Automatic Stubbing”

- “Automatic Stubbing”

## **More About**

- “Stubbing Overview”

## Initialization of uninitialized global variables

Specify how Polyspace treats global variables that are not initialized.

### Settings

**Default:** No initialization

No initialization

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

With random value

Polyspace initializes the global variables with random values.

With zero or random value

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

### Dependencies

You cannot use this option if you select:

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

### Command-Line Information

**Parameter:** `-init-stubbing-vars-random` | `-init-stubbing-vars-zero-or-random`

**Example:** `polyspace-ada -sources filename -init-stubbing-vars-random`

### See Also

“Verify module” on page 1-29

### Related Examples

- “Specify Analysis Options”

- “Choosing Contextual Verification Options”

## Continue after noninitialized variables

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” on page 1-41 | “Initialization of uninitialized global variables” on page 1-37 | “Non-Initialized Local Variable: NIVL” on page 2-3 | “Verify module” on page 1-29

## **Related Examples**

- “Specify Analysis Options”

## Continue with noninitialized in/out parameters

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

### Settings

**Default:** Off

On

Polyspace continues verification even after it detects a red 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” on page 1-39 | “Non-Initialized Local Variable: NIVL” on page 2-3 | “Variable/function range setup” on page 1-34

### Related Examples

- “Specify Analysis Options”

## Treat import as nonvolatile

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

### Settings

**Default:** Off

On

Polyspace considers the imported variables as volatile.

Off

Polyspace does not consider the imported variables as volatile.

### Command-Line Information

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

### See Also

“Treat export as nonvolatile” on page 1-43

### Related Examples

- “Specify Analysis Options”
- “Volatile Variables” on page 3-9
- “Stubbing Overview”



## Treat export as nonvolatile

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” on page 1-42

### Related Examples

- “Specify Analysis Options”
- “Volatile Variables” on page 3-9
- “Stubbing Overview”

## Precision level

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

### Settings

**Default:** 2

0

This option corresponds to a static interval verification.

1

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

2

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

### Command-Line Information

**Parameter:** -0

**Value:** 0 | 1 | 2

**Default:** -02

**Example:** `polyspace-ada -sources file_name -01`

### See Also

“Verification level” on page 1-45

### Related Examples

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

## Verification level

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

### Settings

**Default:** Software Safety Analysis level 2

#### Source Compliance Checking

The verification process checks for compliance of source code.

**Software Safety Analysis level 0**

The verification process runs once on your code.

**Software Safety Analysis level 1**

The verification process runs twice on your code.

**Software Safety Analysis level 2**

The verification process runs thrice on your code.

**Software Safety Analysis level 3**

The verification process runs four times on your code.

**Software Safety Analysis level 4**

The verification process runs five times on your code.

**other**

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

### Command-Line Information

**Parameter:** -to

**Value:** compile | pass0 | pass1 | pass2 | pass3 | pass4 | other

**Example:** polyspace-ada -sources *filename* -to pass2

### See Also

“Precision level” on page 1-44

## **Related Examples**

- “Improve Verification Precision”

## Verification time limit

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

### Settings

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

### Command-Line Information

**Parameter:** `-timeout`

**Value:** Time in hours

**Example:** `polyspace-ada -sources file_name -timeout 5.75`

### Related Examples

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

## Sensitivity context

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

### Settings

**Default:** auto

none

The software does not store call context information for procedures.

auto

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

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

custom

The software stores call context information for procedures that you specify. Click



to enter the name of a procedure.

### Command-Line Information

**Parameter:** `-context-sensitivity`

**Value:** `auto` | `none` | `-custom procedure_name`

**Example:** `polyspace-ada -sources file_name -context-sensitivity auto`

### Related Examples

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

## Improve precision of interprocedural analysis

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

### Settings

**Default:** Off

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

### Tips

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

### Command-Line Information

**Parameter:** `-path-sensitivity-delta`

**Value:** 0 | 1

**Example:** `polyspace-ada -sources filename -path-sensitivity-delta 1`

### Related Examples


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

## Specific precision

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

### Settings

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

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

### Command-Line Information

**Parameter:** `-modules-precision`

**Value:** File name and corresponding precision separated by :

**Example:** `polyspace-ada -sources file_name -01 -modules-precision My_File.c:02`

### See Also

“Precision level” on page 1-44

### Related Examples

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



## Max size of global array variables

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 -01 -array-expansion-size 8`

### See Also

“Expansion limit for a structured variable” on page 1-53 | “Variables to expand” on page 1-52

### Related Examples

- “Specify Analysis Options”

### More About

- “Expansion of Sizes” on page 3-7

## Variables to expand

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

### Settings

**Default** None

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

### Dependencies

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

### Command-Line Information

**Parameter:** -variables-to-expand

**Value:** Variable name

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

### See Also

“Expansion limit for a structured variable” on page 1-53 | “Max size of global array variables” on page 1-51

### Related Examples

- “Specify Analysis Options”

### More About

- “Expansion of Sizes” on page 3-7

## Expansion limit for a structured variable

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

### Settings

**Default:** 1

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

For instance, consider the following code:

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

In this code, if you specify the limit as:

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

### Dependencies

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

### Command-Line Information

**Parameter:** `-variable-expansion-depth`

**Value:** Integer

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

## See Also

“Variables to expand” on page 1-52 | “Max size of global array variables” on page 1-51

## Related Examples

- “Specify Analysis Options”

## More About

- “Expansion of Sizes” on page 3-7

## Generate report

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

### Settings

**Default:** Off

On

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

Off

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

### Tips

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

### Command-Line Information

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

### Related Examples

- “Specify Analysis Options”
- “Generate Report”

## Report template

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

### Settings

**Default:** Developer

#### CodeMetrics

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

#### Developer

The report lists information useful to developers, including:

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

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

#### DeveloperReview

The report lists the same information as the **Developer** report. However, the reviewed results are sorted by 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:** Name of template with extension `.rpt`

**Example:** `polyspace-ada -sources file_name -report-template Developer.rpt`

## Related Examples

- “Specify Analysis Options”
- “Generate Report”
- “Customize Report Templates”

## Output format

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`

### Related Examples

- “Specify Analysis Options”
- “Generate Report”



## Disable code metrics generation

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.

### 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

### Related Examples

- “Specify Analysis Options”

## Other

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**

`-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 *name***

Specify author of verification.

**Default:** user ID

**Example Shell Script Entry:**

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

## **-server *server\_name\_or\_ip[:port\_number]***

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.

### **Option Example Shell Script Entry:**

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

```
polyspace-remote-ada ...
```

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

## **-h[elp]**

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

### **Example Shell Script Entry:**

```
polyspace-ada -h
```

**-v | -version**

Displays the Polyspace version number.

**Example Shell Script Entry:**

```
polyspace-ada -v
```

produces an output like the following:

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



## **-sources**

`-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.

For instance, to analyze the files `source1.ada` and `source2.ada`, use the following syntax:

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

## **-sources-list-file**

`-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

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

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

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

## **-from**

Specify starting point of verification

### **Settings**

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

### **Command-Line Information**

**Parameter:** from

**Type:** string

**Value:** scratch | compile | pass0 | pass1 | pass2 | pass3 | pass4 | other

**Default:** scratch

**Shell script example:** polyspace-ada -from pass0

## **-report-output-name**

Specify name of verification report file

### **Settings**

**Default:** *Prog\_TemplateName.Format* where:

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

### **Command-Line Information**

**Parameter:** `report-output-name`

**Type:** `string`

**Default:** *Prog\_TemplateName.Format*

#### **Shell script example:**

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

## **-less-range-information**

Limit amount of range information displayed in verification results

### **Settings**

**Default:** Off

On

Provide range information on assignments, but not read operations.

Consider the following example:

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

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

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

Off

Range information available on assignments and read operations.

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

### **Command-Line Information**

**Parameter:** `less-range-information`

**Shell script examples:**

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

## **-import-comments**

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

### **Command-Line Information**

**Default:**

Disabled

**Shell script examples: :**

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

## **-tmp-dir-in-results-dir**

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

**Default:**

Disabled

**Example Shell Script Entry:**

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

## **-max-processes**

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

### **Settings**

**Default:** 4

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

### **Command-Line Information**

**Parameter:** max-processes

**Value:** Integer between 1 and 128

**Default:** 4

**Shell script example:** polyspace-ada -max-processes 1



# Check Descriptions

---

- “Function Not Reachable: FNR” on page 2-2
- “Non-Initialized Local Variable: NIVL” on page 2-3
- “Non-Initialized Variable: NIV” on page 2-7
- “Division by Zero: ZDV” on page 2-8
- “Arithmetic Exceptions: EXCP” on page 2-9
- “Scalar and Float Overflow: OVFL” on page 2-12
- “Correctness Condition: COR” on page 2-14
- “Power Arithmetic: POW” on page 2-33
- “User Assertion: ASRT” on page 2-35
- “Non Terminating Call: NTC” on page 2-37
- “Non Terminating Loop: NTL” on page 2-43
- “Unreachable Code: UNR” on page 2-46

### **Function Not Reachable: FNR**

This check determines if a procedure is reachable from the main entry point procedure through the program call tree. See also “Main entry point” on page 1-22.

## Non-Initialized Local Variable: NIVL

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
<pre>pragma interface (C,variable_name) pragma import (C, variable_name)</pre>	<ul style="list-style-type: none"> <li>• green NIVL</li> <li>• Permanent random value</li> </ul>	<ul style="list-style-type: none"> <li>• No NIVL check</li> <li>• Permanent random value</li> </ul>	<ul style="list-style-type: none"> <li>• same behavior as -automatic-stubbing</li> <li>• in/out and out variables are written within their entire type range</li> </ul>

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;	<ul style="list-style-type: none"> <li>• orange NIVL</li> <li>• Permanent random value</li> </ul>	<ul style="list-style-type: none"> <li>• No NIVL check</li> <li>• Permanent random value</li> </ul>

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;	<ul style="list-style-type: none"> <li>• orange NIVL</li> <li>• Permanent random value</li> </ul>	<ul style="list-style-type: none"> <li>• No NIVL check</li> <li>• Permanent random value</li> </ul>

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.

## Non-Initialized Variable: NIV

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

### Example

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

package body Example is
  Var : Integer;

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

## Division by Zero: ZDV

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

### Ada Example:

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



## Arithmetic Exceptions: EXCP

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

- Argument of *sqrt* must be positive
- Argument of *tan* must be different from  $\pi/2$  modulo  $\pi$
- Argument of *log* must be strictly positive
- Argument of *acos* and *asin* must be within  $[-1..1]$
- Argument of *exp* must be less than or equal to a specific value which depends on the processor target: 709 for 64/32 bit targets and 88 for 16 bit targets

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

### Ada Example

```

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

```

26         Fres : Long_Float;
27     begin
28         -- assert is used to redefine range values of a variable.
29         pragma assert(SP > 0.0);
30         pragma assert(P >= 0.0);
31         pragma assert(SN < 0.0);
32         pragma assert(N <= 0.0);
33         pragma assert(NO_TRIG_VAL < -1.0 or NO_TRIG_VAL > 1.0);
34
35         if (bool_random) then
36             res := sqrt(sn); -- EXCP ERROR: argument of SQRT must be
positive.
37         end if ;
38         if (bool_random) then
39             res := tan(M_PI_2);
40                                     -- EXCP Warning: Float argument of
-- 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.

## Scalar and Float Overflow: OVFL

Check to establish whether an arithmetic expression overflows. This is a scalar check with integer types and a float check for floating point expressions.

An overflow is also detected should an array `index_ada.htm` be out of bounds.

### Ada Example

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

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

## Explanation

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

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

# Correctness Condition: COR

### In this section...

“Attributes Check” on page 2-14  
“Array Length Check” on page 2-16  
“DIGITS Value Check” on page 2-18  
“DELTA Value Length Check” on page 2-18  
“Static Range and Values Check” on page 2-19  
“Discriminant Check” on page 2-21  
“Component Check” on page 2-22  
“Dimension Versus Definition Check” on page 2-23  
“Aggregate Versus Definition Check” on page 2-24  
“Aggregate Array Length Check” on page 2-25  
“Sub-Aggregates Dimension Check” on page 2-26  
“Characters Check” on page 2-28  
“Accessibility Level on Access Type” on page 2-29  
“Accessibility of a Tagged Type” on page 2-30  
“Explicit Dereference of a Null Pointer” on page 2-32

## 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); -- Unreachable 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;
46 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
11 RANGE <>) OF INTEGER;
12      TYPE CHOICE_INDEX IS (H, I);
13      TYPE CHOICE_CNTR IS ARRAY(CHOICE_INDEX) OF INTEGER;
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
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 *AI* 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 B'Rand then
36                 Proc2(Y'Access); -- NTC ERROR: propagation of error
at line 22
37             elsif B'Rand then
38                 Proc3(Y'Access); -- NTC ERROR: propagation of error
at line 27
39             end if;
40         end;
41         Proc3(X'Access);
42     end main;
43 end CORACCESS;
44
```

### Explanation

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

On the other hand, *y* is not defined in a block deeper or inside the one in which *Ref* is defined. So, at the end of block, *y* does not exist and *Ref* is supposed to points to on *y*. It is prohibited and Polyspace checks at lines 21 and 26.

---

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

---

### Accessibility of a Tagged Type

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

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

### Ada Example

```

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

```

### Explanation

In the previous example *W* represents any object in the hierarchy rooted at *Tag\_Type*.

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

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

### Explicit Dereference of a Null Pointer

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

#### Ada Example

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

#### Explanation

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

These checks are currently colored orange.



## Power Arithmetic: POW

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

### Ada Example

```

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

```

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

### Explanation

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

---

**Note:** As recognized by the Standard, Polyspace places a green check on the instruction *left\*\*right* with *left:=right:=0*.

---

## User Assertion: ASRT

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

### Ada Example

```

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

```

```
is verified
33           pragma assert(ten_Positive <= 5); -- ASRT ERROR:
Failure User Assert
34           end if;
35
36           end MAIN;
37
38     end ASRT; -- End Package
```

### Explanation

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

- To establish whether the values *flip\_flop* and *var\_flip* in the program are inside the domain which that the program is designed to handle. If the values were outside the range implied by the assert, then the program wouldn't be able to run properly. Thus they are flagged as runtime errors.
- To redefine the range of variables as shown at line 32 where *ASRT.Ten\_positive* is restrained to only a few values. Polyspace makes the assumption that if the program is executed without a runtime error at line 32, *Ten\_positive* can only have a value greater than 5 after the line.

## Non Terminating Call: NTC

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

Check	Description
NTC	<p>Your function called "test" calls f;. And “f;” is flagged as a red NTC. Why? There could be five distinct explanations for this NTC:</p> <ul style="list-style-type: none"> <li>• “f” contains a red error;</li> <li>• “f” contains an NTL ;</li> <li>• “f” contains an NTC;</li> <li>• “f” contains an orange which is context dependant : it is either red or green: for this call, it makes the function crash.</li> </ul> <hr/> <p><b>Note:</b> Some information can be given when clicking on the NTC</p>

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

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

### Solution

Carefully check the reasons with relation to your situation.

---

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

---

### Non Termination of Call: NTC

Check to establish whether a procedure call returns. It is not the case when the procedure contains an endless loop or a certain error, or if the procedure calls another procedure which does not terminate. In the latter instance, the status of this check is propagated to the caller.

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

#### Ada Example

```
1  package NTC is
2      procedure MAIN;
3      -- Stubbed function
4      function Random_Boolean return Boolean;
5  end NTC;
6
7  package body NTC is
8
9      procedure FOO (X : Integer) is
10         Y : Integer;
11     begin
12         Y := 1 / X; -- ZDV Warning: Scalar division
13 by zero may occur
14         while (X >= 0) loop -- NTL ERROR: Loop does not terminate
15             if ( Y /= X) then
16                 Y := 1 / (Y-X);
17             end if;
18         end loop;
19     end FOO;
20
21     procedure MAIN is
22     begin
23         if (Random_Boolean) then
24             FOO(0); --NTC ERROR: Division by zero in NTC.FOO (ZDV)
25         end if ;
26         if (Random_Boolean) then
27             FOO(2); --NTC ERROR: Non Termination Loop in NTC.FOO (NTL)
28         end if;
29     end MAIN;
30 end NTC;
```

#### Explanation

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

- The first does not return because of a division by zero (ZDV warning) at line 12 when  $X = 0$ .

- The second does not terminate because of an infinite loop (red “Non Terminating Loop: NTL” on page 2-43) 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.

---

## Non Terminating Loop: NTL

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

Check	Description
NTL	<p>A NTL is a loop for which the break condition cannot be met. Among NTLs, you will find the following examples:</p> <ul style="list-style-type: none"> <li>• while( 1=1 ) loop function_call; end loop; // informative NTL</li> <li>• while( x &gt;= 0 ) loop x := x+1; end loop; // with x as an unsigned int could reveal a bug, or not (an unsigned is always positive)</li> <li>• 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</li> </ul>

## Non Termination of Loop: NTL

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

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

### Ada Example

```

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

```

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

### Explanation

In the above example, the "continuation condition" of the while is always true and the loop does not exit. Thus Polyspace will raise an error.

In some case, the condition is not trivial and may depend on some program variables. Nevertheless, Polyspace is still able to treat those cases.

### Another NTL Example: Error Propagation

As opposed to other red errors, Polyspace does not continue with the verification in the current branch. Due to the inside error, the (for, do-while, while) loop does not terminate.

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

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

---

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

---

## Unreachable Code: UNR

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

### Ada Example

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

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

## Explanation

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

- As shown at line 26, the first branch is always true (*if-then part*) and so the other branch is not executed (*else part* at lines 40 to 42).
- At line 29 a conditional part of a conditional branch is always true and the other part not evaluated because of the standard definition of logical operator *or else*.
- The piece of code after a red error is not evaluated by Polyspace Server. The call to the function and the lines following line 34 are considered to be dead code. Correcting the red error and relaunching would allow the color to be revised.





# Assumptions Used During Verification

---

- “Why Polyspace Verification Uses Approximations” on page 3-2
- “Procedure Calls with Default Parameters” on page 3-4
- “\_INIT\_PROC Procedures” on page 3-6
- “Expansion of Sizes” on page 3-7
- “Inline Assemblers” on page 3-8
- “Volatile Variables” on page 3-9
- “Shared Variables” on page 3-10
- “Pointers to Explicit Tasks” on page 3-14
- “Limitations of Polyspace Verification” on page 3-16

## Why Polyspace Verification Uses Approximations

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

### 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
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
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 `N error` (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
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 Summary** 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

```
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;
- this doesn't mean the code between two critical sections is atomic;
- It is a binary semaphore: you only have one token per label (in the example below CS1). Unlike many implementations of semaphores, it is not a decrementing counter that can keep track of a number of attempted accesses.

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
<pre>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;</pre>	<pre>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;</pre>

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 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;
```

```
    end Test_Ressource;  
end Test;
```

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

## Limitations of Polyspace Verification

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

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

*Polyspace\_Install*\polyspace\verifier\ada\_limitations.pdf

---

**Note:** By default, the *Polyspace\_Install* folder refers to the following location:

- **Windows systems** – C:\Program Files\Polyspace  
  \PolyspaceForADA\_Release
  - **UNIX<sup>®</sup> systems** – /usr/local/Polyspace/PolyspaceForADA\_Release
-



# Code Metrics — Alphabetical List

---

## 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 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 — Alphabetical List**

---

# 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 Summary** 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	
<b>Main entry point</b>	example.main	
<b>Entry points</b>	example.Task1	
	example.Task2	
<b>Temporally exclusive tasks</b>	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	
<b>Main entry point</b>	<code>example.main</code>	
<b>Entry points</b>	<code>example.Task1</code> <code>example.Task2</code>	
<b>Critical section details</b>	<b>Starting procedure</b>	<b>Ending procedure</b>
	<code>example.Begin_CS</code>	<code>example.End_CS</code>

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” on page 1-22 | “Entry points” on page 1-24 | “Critical section details” on page 1-26 | “Temporally exclusive tasks” on page 1-28

# 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 Summary** 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” on page 1-22 | “Entry points” on page 1-24 | “Critical section details” on page 1-26 | “Temporally exclusive tasks” on page 1-28

## 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 Summary** 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 Summary** 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;
end body example;
```

```
        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.



# Polyspace Report Components — Alphabetical List

---

## **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. Aronyms are used for Polyspace Code Prover™ checks and Polyspace result status.

# 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 addition, the line and column is also displayed.

- File containing the function definition.

In addition, 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.

## 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

# 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:

- 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
- Cyclomatic complexity
- Language scope
- Number of calling functions

# 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.

## 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<sup>®</sup>
- 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.



# Coding Rules Details

Create table of coding rule violations broken down by file

## Description

This component creates tables containing coding rule violations broken down by each file in the Polyspace project. For each rule violation, the table contains the following information:

- Rule number
- Rule description
- Function containing the violation
- Line and column number
- Review information such as classification, status and comments

## Properties

### Select Coding Rules Type

Using this option, you can choose which coding rule violations to display. You can display violations for the following set of coding rules:

- MISRA C rules
- MISRA<sup>®</sup> AC AGC rules
- MISRA C++ rules
- JSF<sup>®</sup> C++ rules
- Custom coding rules

### Display by

Using this option, you can break down the display of coding rule violations by file.

## Coding Rules Summary

Create table with number of coding rule violations

### Description

This component creates a table containing the number of coding rule violations. You can choose whether to break this information down by rule number or file.

### Properties

#### Select Coding Rules Type

Using this option, you can choose which coding rule violations to display. You can display violations for the following set of coding rules:

- MISRA C rules
- MISRA AC AGC rules
- MISRA C++ rules
- JSF C++ rules
- Custom coding rules

#### Include Files/Rules with No Problems Detected

If you select this option, the table displays:

- Files that do not contain coding rule violations
- Rules that your code does not violate

#### Display by

Using this option, you can break down the display of coding rule violations by:

- Rule number

- File

## Configuration Parameters

Create table of analysis options and coding rules

### Description

This component creates two 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.
- 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.

# Defects Summary

Create table of Polyspace Bug Finder defects

## Description

This component creates a table of Polyspace Bug Finder defects. From this table, you can see the number of defects of each type.

## Properties

### Include Checkers with No Defects Detected

If you select this option, the table includes all defect types that Polyspace Bug Finder can detect, including those that do not occur in your code.

## **Recursive Functions**

Create table of recursive functions

### **Description**

This component creates a table containing the recursive functions in your source code. For each recursive function, the table lists its immediate caller.

# 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
<b>Include Project Metrics</b>	Choose whether to include metrics about your Polyspace project.	Select the check box to include project metrics.
<b>Project metrics to include</b>	Specify project metrics to include or exclude from report.	Enter a regular MATLAB <sup>®</sup> expression.
<b>Include File Metrics</b>	Choose whether to include per file metrics in report.	Select the check box to include per file metrics.
<b>File Metrics &gt; Files to include</b>	Specify files to include or exclude when reporting file metrics.	Enter a regular MATLAB expression.
<b>File metrics to include</b>	Specify file metrics to include or exclude from report.	Enter a regular MATLAB expression.
<b>Include Function Metrics</b>	Choose whether to include per function metrics in report.	Select the check box to include per function metrics.

Property	Purpose	User Action
<b>Function Metrics &gt; Files to include</b>	Specify files to include or exclude when reporting function metrics.	Enter a regular MATLAB expression.
<b>Functions to include</b>	Specify functions to include or exclude when reporting function metrics.	Enter a regular MATLAB expression.
<b>Function metrics to include</b>	Specify function metrics to include or exclude from report.	Enter a regular MATLAB expression.

## Coding Rules Filters

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

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

## Run-time Check Filters

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



Property	Purpose
<b>Red Checks</b>	Specify whether to include red checks in your report. Red checks indicate proven run-time errors.
<b>Gray Checks</b>	Specify whether to include gray checks in your report. Gray checks indicate unreachable code.
<b>Orange Checks</b>	Specify whether to include orange checks in your report. Orange checks indicate possible run-time errors.
<b>Green Checks</b>	Specify whether to include green checks in your report. Green checks indicate that an operation does not contain a specific run-time error.
<b>Inspection Point Checks</b>	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.
<b>Unreachable Functions</b>	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
<b>Justification status</b>	Choose whether to report only justified checks, only unjustified checks or all checks.	Choose an option from the dropdown list.
<b>Files to include</b>	Specify files to include or exclude from your report.	Enter a regular MATLAB expression.
<b>Check types to include</b>	Specify Polyspace Code Prover checks to include in your report.	Enter a regular MATLAB expression.

<b>Property</b>	<b>Purpose</b>	<b>User Action</b>
<b>Function names to include</b>	Specify functions to include or exclude from your report.	Enter a regular MATLAB expression.
<b>Classification types to include</b>	Specify classifications to include or exclude from your report.	Enter a regular MATLAB expression.
<b>Status types to include</b>	Specify statuses to include or exclude from your report.	Enter a regular MATLAB expression.
<b>Comments to include</b>	Specify comments to include or exclude from your report.	Enter a regular MATLAB expression.

### More About

- “Regular Expressions”

# Run-time Checks Details Ordered by Color/File

Create overrides for global filters in Polyspace reports

## 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
<b>Red Checks</b>	Specify whether to include red checks in your report. Red checks indicate proven run-time errors.
<b>Gray Checks</b>	Specify whether to include gray checks in your report. Gray checks indicate unreachable code.
<b>Orange Checks</b>	Specify whether to include orange checks in your report. Orange checks indicate possible run-time errors.

Property	Purpose
<b>Green Checks</b>	Specify whether to include green checks in your report. Green checks indicate that an operation does not contain a specific run-time error.
<b>Inspection Point Checks</b>	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.
<b>Unreachable Functions</b>	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
<b>Justification status</b>	Choose whether to report only justified checks, only unjustified checks or all checks.	Choose an option from the dropdown list.
<b>Files to include</b>	Specify files to include or exclude from your report.	Enter a regular MATLAB expression.
<b>Check types to include</b>	Specify Polyspace Code Prover checks to include in your report.	Enter a regular MATLAB expression.
<b>Function names to include</b>	Specify functions to include or exclude from your report.	Enter a regular MATLAB expression.
<b>Classification types to include</b>	Specify classifications to include or exclude from your report.	Enter a regular MATLAB expression.
<b>Status types to include</b>	Specify statuses to include or exclude from your report.	Enter a regular MATLAB expression.
<b>Comments to include</b>	Specify comments to include or exclude from your report.	Enter a regular MATLAB expression.

## Run-time Checks Details Ordered by Review Information

Create table with Polyspace Code Prover checks ordered by review information

### Description

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

## Run-time Checks Summary Ordered by File

Create table with Polyspace Code Prover checks ordered by file

### 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.

# Software Quality Objectives - Coding Rules Summary

Create table of coding rule violations in results downloaded from Polyspace Metrics

## Description

This component creates a table containing coding rule violations in results downloaded from Polyspace Metrics.

## Software Quality Objectives - Run-time Checks Details

Create table of run-time checks in results downloaded from Polyspace Metrics

### Description

This component creates a table containing run-time checks in results downloaded from Polyspace Metrics.



# Software Quality Objectives - Run-time Checks Summary

Create table of run-time check distribution in results downloaded from Polyspace Metrics

## Description

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

## Variable Access

Create table showing global variable access in source code

### 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.