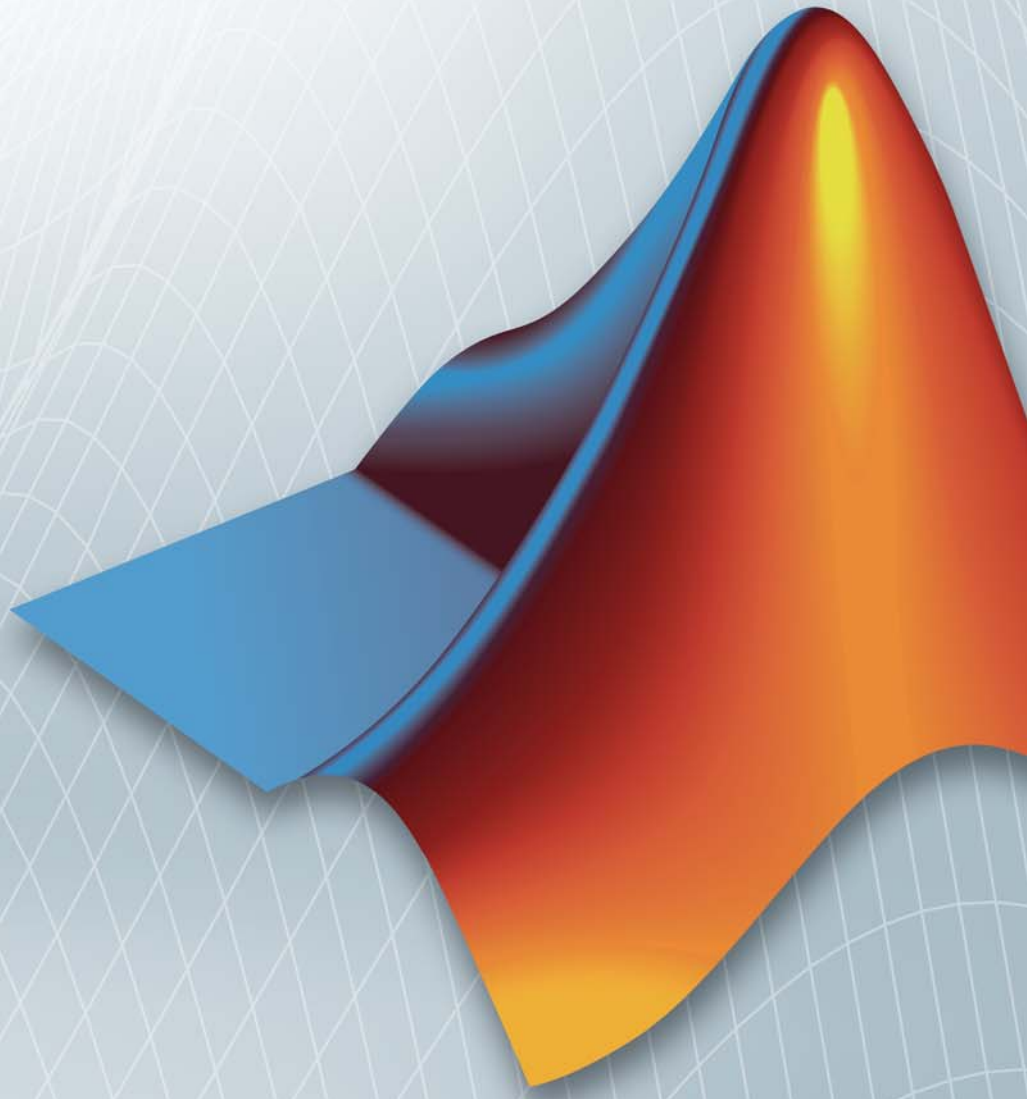
Polyspace® Products for C/C++

Reference

R2012b



MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Polyspace® Products for C/C++ Reference

Trademarks

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

Patents

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

Revision History

March 2009	Online Only	New for Version 7.0 (Release 2009a)
September 2009	Online Only	Revised for Version 7.1 (Release 2009b)
March 2010	Online Only	Revised for Version 7.2 (Release 2010a)
September 2010	Online Only	Revised for Version 8.0 (Release 2010b)
April 2011	Online Only	Revised for Version 8.1 (Release 2011a)
September 2011	Online Only	Revised for Version 8.2 (Release 2011b)
March 2012	Online Only	Revised for Version 8.3 (Release 2012a)
September 2012	Online Only	Revised for Version 8.4 (Release 2012b)

Contents

1 Option Descriptions for C Code

2 Option Descriptions for C++ Code

3 Check Descriptions for C Code

4 Check Descriptions for C++ Code

5 Approximations Used During Verification

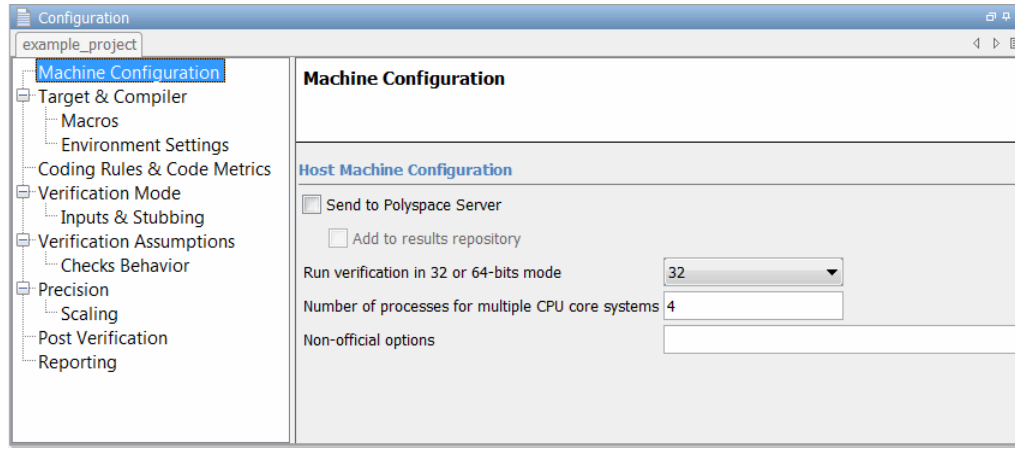
6 Examples

1

Option Descriptions for C Code

Polyspace Analysis Options Overview

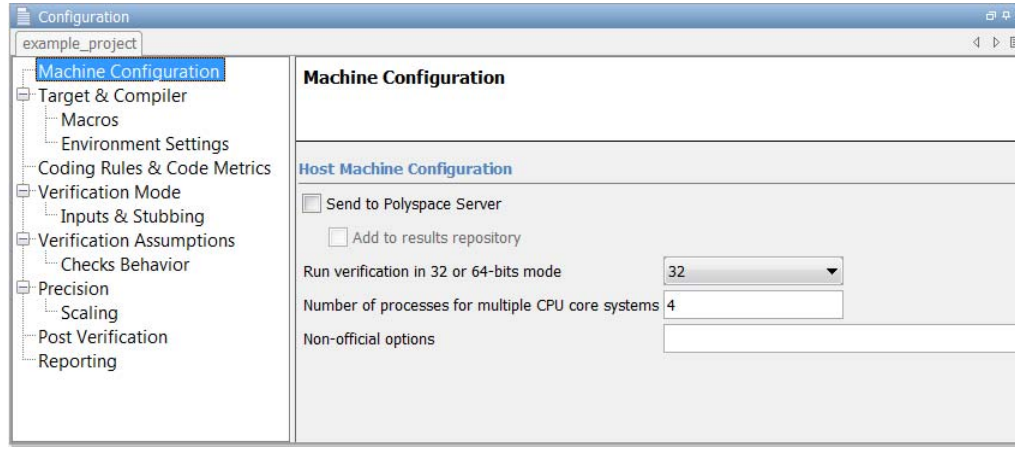
In the Project Manager perspective, on the **Configuration** pane, you can specify the analysis options that Polyspace® software uses for code verification.



Polyspace software groups the analysis options into various categories. To display the parameters for a specific category, from the **Configuration** tree, select the category.

Note From the command line, you can use the `polyspace-c` command to specify parameters. The description for each parameter includes command line information.

Machine Configuration



In this section...

Machine Configuration Overview

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

Send to Polyspace Server

Specify whether verification runs on the server or client system.

Settings

Default: On

 On

Run verification on the Polyspace server. The server to use is specified in the Polyspace preferences.

 Off

Run verification on the client system.

Tips

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

Command-Line Information

Parameter: `-server`
Value: *`name or IP address:port number`*
Example: `polyspace-remote-c  server 192.168.1.124:12400`

Add to results repository

Specify whether verification results are added to the Polyspace Metrics results database, allowing Web-based reporting of results and code metrics.

Settings

Default: Off

 On

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

 Off

Verification results are not added to the database.

Dependency

- This option is available only for server verifications.

Command-Line Information

Parameter: `-add-to-results-repository`
Example: `polyspace-c -server -add-to-results-repository`

Run verification in 32 or 64-bits mode

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

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

Available options are:

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

Default:

```
auto
```

Example Shell Script Entry:

```
polyspace-c -machine-architecture auto
```

Number of processes for multiple CPU core systems

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

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

Default:

```
4
```

Example Shell Script Entry:

```
polyspace-c -max-processes 1
```

Non-official options

`-extra-flags`

This option specifies an expert option to be added to the analyzer. Each word of the option (even the parameters) must be preceded by *-extra-flags*.

These flags will be given to you by MathWorks if required.

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-c -extra-flags -param1 -extra-flags -param2 \
 -extra-flags 10 ...
```

`-c-extra-flags`

This option is used to specify an expert option to be added to a verification. Each word of the option (even the parameters) must be preceded by *-c-extra-flags*.

These flags will be given to you by MathWorks if required.

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-c -c-extra-flags -param1 -c-extra-flags -param2
-c-extra-flags 10
```

`-cfe-extra-flags`

This option is used to specify an expert option for a verification.

These flags will be given to you by MathWorks if required.

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-c -cfe-extra-flags -param1 -cfe-extra-flags -param2
```

`-il-extra-flags`

This option is used to specify an expert option to be added to a verification. Each word of the option (even the parameters) must be preceded by *-il-extra-flags*.

These flags will be given to you by MathWorks if required.

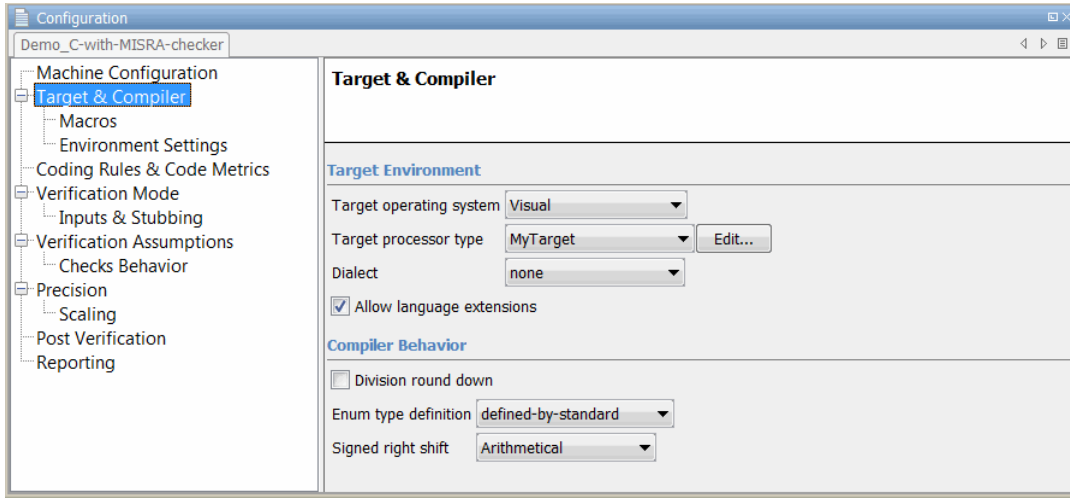
Default:

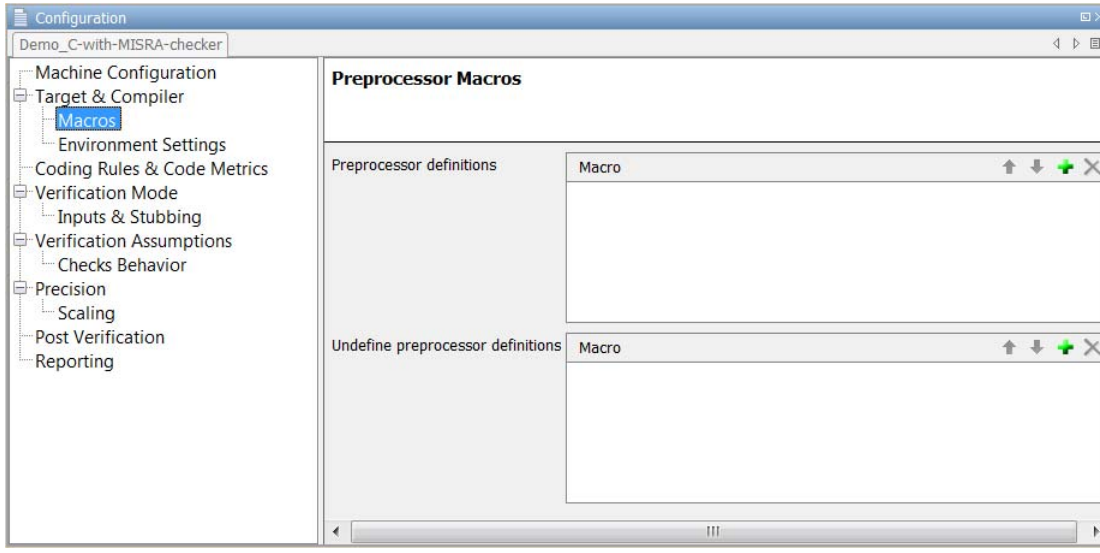
No extra flags.

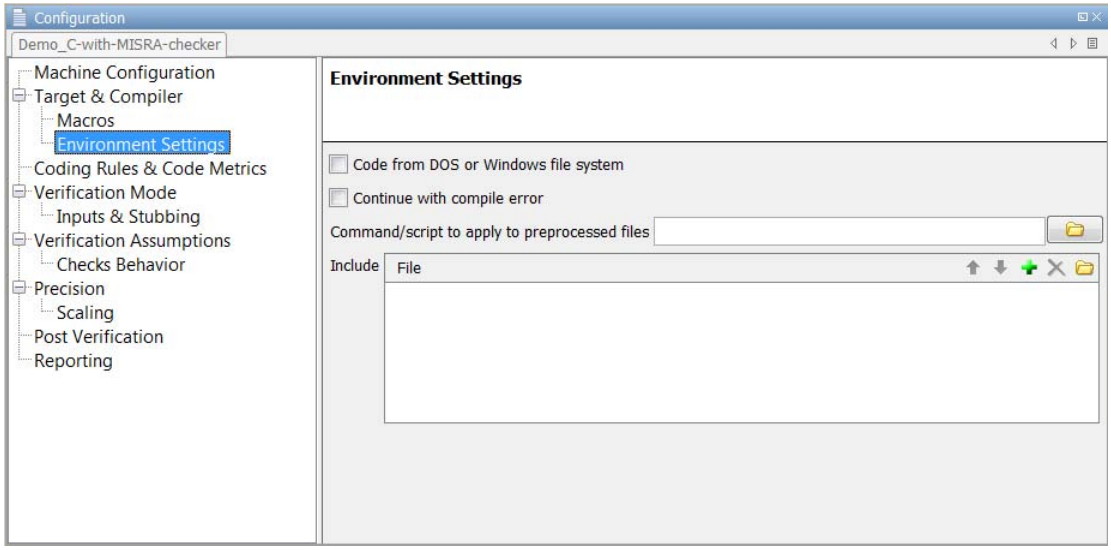
Example Shell Script Entry:

```
polyspace-c -il-extra-flags -param1 -il-extra-flags -param2
-il-extra-flags 10
```

Target & Compiler







In this section...

In this section...

Target & Compiler Overview

Use **Target & Compiler**, **Target & Compiler > Macros**, and **Target & Compiler > Environment Settings** to specify the target environment and compiler behavior.

Target operating system

This option specifies the operating system target for your application.

Possible values are:

- `Linux`
- `Solaris`
- `VxWorks`
- `Visual`
- `no-predefined-OS`

This information allows the corresponding system definitions to be used during preprocessing — to analyze the included files properly.

You can use the target `no-predefined-OS` in conjunction with -include or/and -D to give all of the system preprocessor flags to be used at execution time. Details of these may be found by executing the compiler for the project in verbose mode.

Default:

`Linux`

Note Only the Linux® include files are provided with Polyspace software (see the include folder in the installation directory). Projects developed for use with other operating systems may be analyzed by using the corresponding include files for that OS. For instance, in order to analyze a VxWorks® project, use the option `-I` *`path_to_the_VxWorks_include_folder`*

Example shell script entry:

```
polyspace-c -OS-target linux
polyspace-c  -OS-target no-predefined-OS -D GCC_MAJOR=2     /
      -include /complete_path/inc/gn.h  ...
```

Target processor type

This option specifies the target processor type, and in doing so informs the verification of the size of fundamental data types and of the endianess of the target machine.

Possible values are:

- `i386` (default)
- `sparc`
- `m68k`
- `powerpc`
- `c-167`
- `tms320c3x`
- `sharc21x61`
- `necv850`
- `hc08`
- `hc12`
- `mpc5xx`
- `c18`
- `x86_64`

- `mcpu...(Advanced)`

`mcpu` is a reconfigurable Micro Controller/Processor Unit target. You can use this type to configure one or more generic targets.

You can analyze code intended for an unlisted processor type using one of the other processor types, they share common data properties.

For information on specifying a generic target, or modifying the `mcpu` target, see “Generic target options” on page 1-13.

Default:

`i386`

Example shell script entry:

```
polyspace-c -target m68k ...
```

Generic target options

The *Generic target options* dialog box is only available when you select a *mcpu* target.

Allows the specification of a generic *"Micro Controller/Processor Unit"* or *mcpu* target name. Initially, use the dialog box to specify the name of a new *mcpu* target — say, “MyTarget”.

That new target is added to the -target options list. The default characteristics of the new target are as follows (using the *type [size, alignment]* format)

- *char [8, 8, char [16,16]]*
- *short [8,8], short [16, 16]*
- *int [16, 16]*
- *long [32, 32], long long [32, 32]*
- *float [32, 32], double [32, 32], long double [32, 32]*
- *pointer [16, 16]*

- *char is signed*
- *little-endian*

When using the command line, *MyTarget* is specified with all the options for modification:

```
polyspace-c -target MyTarget
```

For example, a specific target uses 8 bit alignment (see also -align), for which the command line would read:

```
polyspace-c -target mcpu -align 8
```

-little-endian

This option is only available when a -*mcpu* generic target has been chosen.

The endianness defines the byte order within a word (and the word order within a long integer). Little-endian architectures are Less Significant byte First (LSF), for example: i386.

For a little endian target, the less significant byte of a short integer (for example 0x00FF) is stored at the first byte (0xFF) and the most significant byte (0x00) at the second byte.

Example shell script entry:

```
polyspace-c -target mcpu -little-endian
```

-big-endian

This option is only available when a -*mcpu* generic target has been chosen.

The endianness defines the byte order within a word (and the word order within a long integer). Big-endian architectures are Most Significant byte First (MSF), for example: SPARC, m68k.

For a big endian target, the most significant byte of a short integer (for example 0x00FF) is stored at the first byte (0x00) and the less significant byte (0xFF) at the second byte.

Example shell script entry:

```
polyspace-c -target mcpu -big-endian
```

-default-sign-of-char [signed|unsigned]

This option is available for all targets. It allows a char to be defined as "signed", "unsigned", or left to assume the mcpu target's default behavior

- **default mode** – The sign of char is left to assume the target's default behavior. By default all targets are considered as signed except for hc08 and powerpc targets.
- **signed** – Disregards the target's default char definition, and specifies that a "signed char" should be used.
- **unsigned** – Disregards the target's default char definition, and specifies that a "unsigned char" should be used.

Example Shell Script Entry

```
polyspace-c -default-sign-of-char unsigned -target mcpu ...
```

-char-is-16bits

This option is only available when a -*mcpu* generic target has been chosen.

The default configuration of a generic target defines a char as 8 bits. This option changes it to 16 bits, regardless of sign.

the minimum alignment of objects is also set to 16 bits and so, incompatible with the options -short-is-8bits and -align 8.

Setting the char type to 16 bits has consequences on the following:

- computation of size of for objects
- detection of underflow and overflow on chars

Without the option char for *mcpu* are 8 bits

Example shell script entry:

```
polyspace-c -target mcpu -char-is-16bits
```

-short-is-8bits

This option is only available when a *mcpu* generic target has been chosen.

The default configuration of a generic target defines a short as 16 bits. This option changes it to 8 bits, regardless of sign.

It sets a short type as 8-bit without specific alignment. That has consequences for the following:

- computation of size of objects referencing short type
- detection of short underflow/overflow

Example shell script entry

```
polyspace-c -target mcpu -short-is-8bits
```

-int-is-32bits

This option is available with a *mcpu* generic target, hc08, hc12 and mpc5xx target has been chosen.

The default configuration of a generic target defines an int as 16 bits. This option changes it to 32 bits, regardless of sign. Its alignment, when an int is used as struct member or array component, is also set to 32 bits. See also -align option.

Example shell script entry

```
polyspace-c -target mcpu -int-is-32bits
```

-long-long-is-64bits

This option is only available when a *mcpu* generic target has been chosen.

The default configuration of a generic target defines a long long as 32 bits. This option changes it to 64 bits, regardless of sign. When a long long is used

as struct member or array component, its alignment is also set to 64 bits. See also -align option.

Example shell script entry

```
polyspace-c -target mcpu -long-long-is-64bits
```

-double-is-64bits

The default configuration of a generic target defines a double as 32 bits. This option, changes both double and *long double* to 64 bits. When a double or long double is used as a struct member or array component, its alignment is set to 4 bytes.

See also -align option.

Defining the double type as a 64 bit double precision float impacts the following:

- Computation of sizeofobjects referencing double type
- Detection of floating point underflow/overflow

This option is available for the following targets:

- *mcpu* generic target
- sharc21x61
- hc08
- hc12
- mpc5xx

Example

```
int main(void)
{
 struct S {char x; double f;};
 double x;
 unsigned s1, s2;
 s1 = sizeof (double);
```

```
 s2 = sizeof(struct S);
 x = 3.402823466E+38; /* IEEE 32 bits float point maximum value */
 x = x * 2;
 return 0;
}
```

Using the default configuration of sharc21x62, Polyspace verification assumes that a value of 1 is assigned to s1, 2 is assigned to s2, and there is a consequential float overflow in the multiplication x * 2. Using the –double-is-64bits option, a value of 2 is assigned to s1, and no overflow occurs in the multiplication (because the result is in the range of the 64-bit floating point type)

Example shell script entry

```
polyspace-c -target mcpu -double-is-64bits
```

-pointer-is-32bits

This option is only available when a *mcpu* generic target has been chosen.

The default configuration of a generic target defines a pointer as 16 bits. This option changes it to 32 bits. When a pointer is used as struct member or array component, its alignment is also set also to 32 bits (see -align option).

Example shell script entry

```
polyspace-c -target mcpu -pointer-is-32bits
```

-align [8|16|32]

This option is available with a *mcpu* generic target and some other specific targets (with hc08, hc12 or mpc5xx available values are 16 and 32). It is used to set the largest alignment of all data objects to 4/2/1 byte(s), meaning a 32, 16 or 8 bit boundary respectively.

-align 32 (Default). The default alignment of a generic target is 32 bits. This means that when objects with a size of more than 4 bytes are used as struct members or array components, they are aligned at 4 byte boundaries.

Example shell script entry with a 32 bits default alignment

```
polyspace-c -target mcpu
```

-align 16. If the `-align 16` option is used, when objects with a size of more than 2 bytes are used as struct members or array components, they are aligned at 2 bytes boundaries.

Example shell script entry with a 16 bits specific alignment:

```
polyspace-c -target mcpu -align 16
```

-align 8. If the `-align 8` option is used, when objects with a size of more than 1 byte are used as struct members or array components, are aligned at 1 byte boundaries. Consequently the storage assigned to the arrays and structures is strictly determined by the size of the individual data objects without member and end padding.

Example shell script entry with a 8 bits specific alignment:

```
polyspace-c -target mcpu -align 8
```

Dialect

Specify whether verification allows syntax associated with the IAR and Keil dialects.

Settings

Default: `none`

`none`
: Verification does not allow non-ANSI® C dialects.

`keil`
: Verification allows non-ANSI C syntax and semantics associated with the Keil dialect.

`iar`
: Verification allows non-ANSI C syntax and semantics associated with the IAR dialect.

Tips

- IAR refers to the compilers from IAR Systems (www.iar.com).
- Keil refers to the Keil™ products from ARM (www.keil.com).
- Using this option allows verification to tolerate additional structure types as keywords of the language, such as `sfr`, `sbit`, and `bit`. These structures and associated semantics are part of the compiler that has integrated it with the ANSI C language as an extension.

  Example of source code with Keil dialect:

  ```
  unsigned char bdata Status[4];
  sfr AU = 0xF0;
  sbit OCmd = Status[0]^2;
  s^2 = 1; s^6 = 0;
  ```

  Example with IAR dialect:

  ```
  unsigned char bdata Status[4];
  sfr OCmd @ 0x4FFE;
  OCmd.2 = 1; s.6 = 0;
  ```

Command-Line Information

Parameter: `-dialect`
Type: string
Value: `none | keil | iar`
Default: `none`
Example: `polyspace-c  dialect keil`

See Also

"Verify Keil or IAR Dialects".

Allow language extensions

This option allows the verification to accept a subset of common C language constructs and extended keywords, as defined by the C99 standard or supported by many compilers.

When you select this option, the following constructs are supported:

- Designated initializers (labeling initialized elements)
- Compound literals (structs or arrays as values)
- Boolean type (`_Bool`)
- Statement expressions (statements and declarations inside expressions)
- `typeof` constructs
- Case ranges
- Empty structures
- Cast to union
- Local labels (`__label__`)
- Hexadecimal floating-point constants
- Extended keywords, operators, and identifiers (`_Pragma`, `__func__`, `__const__`, `__asm__`)

In addition, when you use this option, the software ignores the following extended keywords:

- `near`
- `far`
- `restrict`
- `_attribute_(X)`
- `rom`

Note This option is set automatically when you select the `-permissive` option.

You cannot use this option with the `-strict` option.

Default:

Selected

Example Shell Script Entry:

```
polyspace-c -allow-language-extensions
```

Sfr type support

Associated to the option `-dialect`, if the code uses specific `sfr` type keyword, it is **mandatory** to declare using `sfr-types` option. It gives the name of the `sfr` type and its size in bits. The syntax is:

`-sfr-types <sfr_name>=<size_in_bits>,`

where `<sfr_name>` could be any name, but most of the time we encounter `sfr`, `sfr16` and `sfr32` . `<size in bits>` could be one of the values `8`, `16` and `32`.

Default:

No dialect used.

Example Shell Script Entry:

```
polyspace-c  dialect iar  sfr-types sfr=8,sfr32=32,sfrb=16
```

Division round down

This option concerns the division and modulus of a negative number.

The ANSI standard stipulates that "*if either operand of / or % is negative, whether the result of the / operator, is the largest integer less or equal than the algebraic quotient or the smallest integer greater or equal than the quotient, is implementation defined, same for the sign of the % operator*".

Note `a = (a / b) * b + a % b` is always true.

Default:

Without the option (default mode), if either operand of / *or* % is negative, the result of the / operator is the smallest integer greater or equal than the algebraic quotient. The result of the % operator is deduced from `a % b = a - (a / b) * b`

Example:

`assert(-5/3 == -1 && -5%3 == -2);` is **true** .

With the *-div-round-down* option:

If either operand / *or* % is negative, the result of the / operator is the largest integer less or equal than the algebraic quotient. The result of the % operator is deduced from `a % b = a - (a / b) * b` .

Example:

`assert(-5/3 == -2 && -5%3 == 1);` is **true** .

Example Shell Script Entry:

polyspace-c -div-round-down ...

Enum type definition

Allows the verification to use different base types to represent an enumerated type, depending on the enumerator values and the selected definition.

When using this option, each enum type is represented by the smallest integral type that can hold all its enumeration values.

Possible values are:

- **`defined-by-standard`** – Uses the integer type (`signed int`).
- **`auto-signed-first`** - Uses the first type that can hold all of the enumerator values from the following list: `signed char`, `unsigned char`, `signed`

`short`, `unsigned short`, `signed int`, `unsigned int`, `signed long`, `unsigned long`, `signed long long`, `unsigned long long`.

- **`auto-unsigned-first`** - Uses the first type that can hold all of the enumerator values from the following lists:
  - If enumerator values are all positive: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`.
  - If one or more enumerator values are negative: `signed char`, `signed short`, `signed int`, `signed long`, `signed long long`.

Signed right shift

Choose between arithmetical and logical computation.

- **- Arithmetic:** the sign bit remains:

```
(-4) >> 1 = -2
(-7) >> 1 = -4
   7 >> 1 = 3
```

- **- Logical:** 0 replaces the sign bit

```
(-4) >> 1 = (-4U) >> 1 = 2147483646
(-7) >> 1 = (-7U) >> 1 = 2147483644
 7 >> 1 = 3
```

Example shell script entry

When using the command line, arithmetic is the default computation mode. When this option is set, logical computation will be performed.

```
polyspace-c -logical-signed-right-shift
```

Defined Preprocessor Macros

This option is used to define macro compiler flags to be used during compilation phase.

Only one flag can be used with each –D as for compilers, but the option can be used several times as shown in the example below.

Default:

Some defines are applied by default, depending on your -OS-target option.

Example Shell Script Entry:

```
polyspace-c -D HAVE_MYLIB -D USE_COM1 ...
```

Undefined Preprocessor Macros

This option is used to undefine a macro compiler flags

As for compilers, only one flag can be used with each –U, but the option can be used several times as shown in the example below.

Default:

Some undefines may be set by default, depending on your -OS-target option.

Example Shell Script Entry:

```
polyspace-c -U HAVE_MYLIB -U USE_COM1 ...
```

Code from DOS or Windows file system

Use this option when the contents of the **include** or **source** folder comes from a DOS or Windows® file system. It deals with upper/lower case sensitivity and control character issues.

The affected files are:

- Header files in all include folders specified through the `-I` option.
- All source files selected for the verification through the `-sources` option.

For example, with this option,

```
#include "..\mY_TEst.h"^M

#include "..\mY_other_FILE.H"^M
```

resolves to:

```
#include "../my_test.h"

#include "../my_other_file.h"
```

Default:

Enabled

Example Shell Script Entry:

```
      polyspace-c -I /usr/include -dos -I ./my_copied_include_dir
-D test=1
```

Continue with compile error

Specifies that verification continues even if some source files do not compile. Functions that are used but not specified are stubbed automatically.

If a source file contains global variables, you may also need to select the option `-allow-undef-variables` to enable verification.

Example Shell Script Entry :

```
 polyspace-c -continue-with-compile-error ...
```

Command/script to apply to preprocessed files

When this option is used, the specified script file or command is run just after the preprocessing phase on each source file. The script executes on each preprocessed c file. The command should be designed to process the standard output from preprocessing and produce its results in accordance with that standard output.

Note The Compilation Assistant is automatically disabled when you specify this option.

You can find each preprocessed file in the results directory in the zipped file ci.zip located in <results/ALL/SRC/MACROS. The extension of the preprocessed file is .ci.

It is important to preserve the number of lines in the preprocessed .ci file. Adding a line or removing one could result in some unpredictable behavior on the location of checks and MACROS in the Polyspace viewer.

Default:

No command.

Example Shell Script Entry – file name:

To replace the keyword “Volatile” by “Import”, you can type the following command on a Linux workstation:

```
polyspace-c -post-preprocessing-command `pwd`/replace_keywords
```

where `replace_keywords` is the following script:

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

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

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

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

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

```
%POLYSPACE_C%\Verifier\bin\polyspace-c.exe
-post-preprocessing-command
%POLYSPACE_C%\Verifier\tools\perl\win32\bin\perl.exe
<absolute_path>\replace_keywords
```

Include

This option is used to specify files to be included by each C file involved in the verification.

Default:

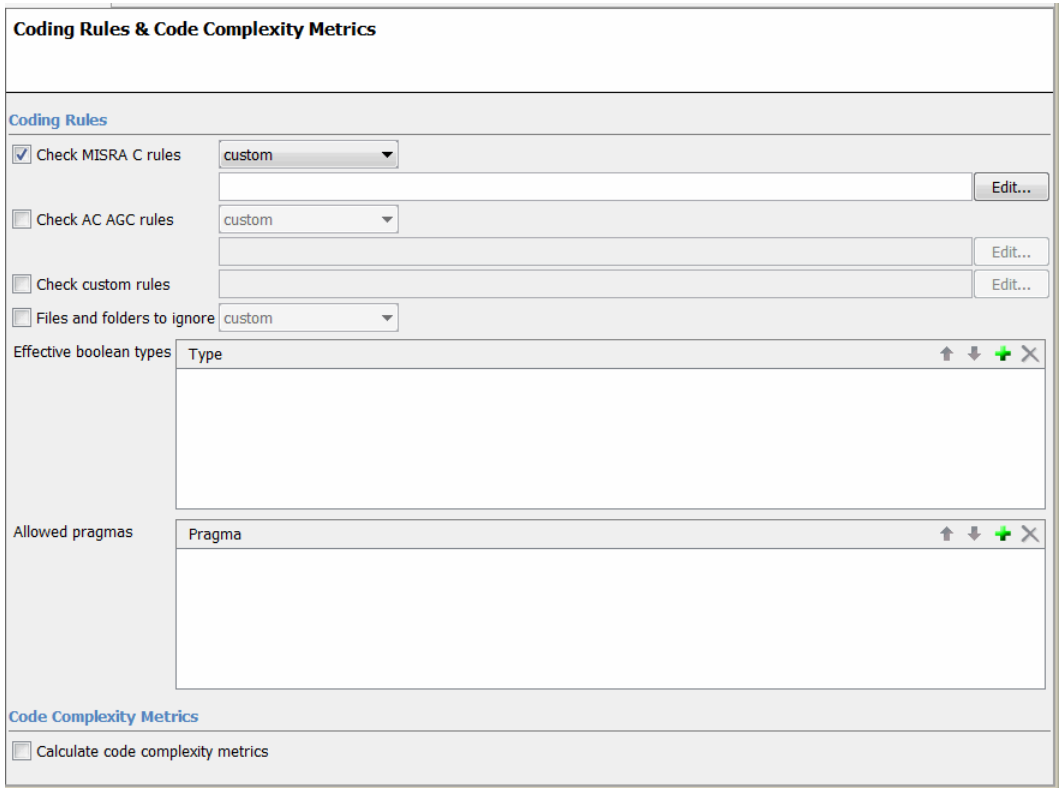
No file is universally included by default, but directives such as "#include <*include_file.h*>" are acted upon.

Example Shell Script Entry:

```
 polyspace-c -include `pwd`/sources/a_file.h -include
/inc/inc_file.h ...

 polyspace-c -include /the_complete_path/my_defines.h ...
```

Coding Rules & Code Complexity Metrics



In this section...

In this section...

Check MISRA C rules

This option allows you to check the code against a set of MISRA-C:2004 rules. All MISRA checks are included in the log file of the verification.

Note This option requires a Polyspace Client™ for C/C++ license.

MISRA C rules configuration

Specifies set of coding rules to check.

Available options are:

- `required-rules` — Check *required* MISRA C® coding rules. All violations are reported as warnings.
- `all-rules` — Check all (*required* and *advisory*) MISRA C coding rules. All violations are reported as warnings.
- `SQO-subset1` — Check a subset of MISRA C rules that have a direct impact on the selectivity of verification. All violations are reported as warnings. For more information, see "SQO Subset 1 – Direct Impact on Selectivity".
- `SQO-subset2` — Check a second subset of MISRA C rules that have an indirect impact on the selectivity of verification, as well as the rules contained in SQO-subset1. All violations are reported as warnings. For more information, see "SQO Subset 2 – Indirect Impact on Selectivity".
- `custom` — Check a specified set of coding rules. You must provide the name of an ASCII file containing a list of MISRA® rules to check.

  Format of the custom file:

  ```
  <rule number> off|error|warning
  ```

Use the character # at the start of a comment. For example:

```
# MISRA configuration file for my_project
10.5 off # disable misra rule number 10.5
17.2 error # violation misra rule 17.2 is an error
17.3 warning # violation of misra rule 17.3 is a warning
```

Default:

```
all-rules
```

Example Shell Script Entry:

```
polyspace-c -misra2 all-rules ...

polyspace-c -misra2 SQO-subset1 ...

polyspace-c -misra2 -custom myrules.txt ...
```

Check AC AGC rules

This option allows you to check the code against rules specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*. All MISRA AC AGC checks are included in the log file of the verification.

Note This option requires a Polyspace Client for C/C++ license.

MISRA AC AGC rules configuration

Specifies set of coding rules to check.

Available options are:

- `OBL-rules` — Check coding rules that belong to the OBL (obligatory) category specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*

- `OBL-REC-rules` — Check coding rules that belong to the the OBL (obligatory) and REC (recommended) categories specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*
- `all-rules` — Check all MISRA C coding rules. All violations are reported as warnings.
- `SQO-subset1` — Check a subset of MISRA C rules that have a direct impact on the selectivity of verification. All violations are reported as warnings. For more information, see "SQO Subset 1 – Direct Impact on Selectivity"
- `SQO-subset2` — Check a second subset of MISRA C rules that have an indirect impact on the selectivity of verification, as well as the rules contained in SQO-subset1. All violations are reported as warnings. For more information, see "SQO Subset 2 – Indirect Impact on Selectivity".
- `custom` — Check a specified set of coding rules. You must provide the name of an ASCII file containing a list of MISRA rules to check.

  Format of the custom file:

  ```
  <rule number> off|error|warning
  ```

  Use the character # at the start of a comment. For example:

  ```
  # MISRA configuration file for my_project
  10.5 off # disable misra rule number 10.5
  17.2 error # violation misra rule 17.2 is an error
  17.3 warning # violation of misra rule 17.3 is a warning
  ```

Default:

Disabled

Example Shell Script Entry

```
polyspace-c -misra-ac-agc all-rules ...

polyspace-c -misra-ac-agc OBL-rules ...

polyspace-c -misra-ac-agc SQO-subset1 ...
```

```
polyspace-c -misra-ac-agc -custom myrules.txt ...
```

Check custom rules

Check names or text patterns in source code with reference to custom rules in specified text file. Each rule defines a check of a specified pattern against a source code identifier. For more information, see “Create a Custom Coding Rules File”.

Default:

Disabled

Example Shell Script Entry

```
polyspace-c -custom-rules myrules.txt
```

Files and folders to ignore

This option prevents MISRA rules checking in a given list of files or folders (all files and subfolders under selected folder). This option is useful when non-MISRA C conforming include headers are used.

The software displays a warning if:

- Any file or folder on the list does not exist.
- All source code is ignored.

This option is allowed only when `-misra2` is used.

Note You can exclude all include folders from MISRA C checking by specifying `"all"`. All files and folders included using the `-I` option are automatically added to `-includes-to-ignore`, and therefore excluded from the coding rules checker.

Example shell script entry :

```
   polyspace-c -misra2 misra.txt  includes-to-ignore
"c:\usr\include"

   polyspace-c -misra2 misra.txt  includes-to-ignore "all"
```

Effective boolean types

Use this option with the `-misra2` option to specify data types that you want Polyspace to treat as Boolean. The use of this option may affect the checking of MISRA-C rules 12.6, 13.2, and 15.4.

The command line syntax for this option is

```
-boolean-types type1,type2, ...
```

where `type1,type2, ...` are names of the data types that you want Polyspace to treat as Boolean.

Polyspace applies this treatment to the named data types in *all* source files. For example, if two different data types share a name that is passed to the option, then Polyspace considers both data types to be Boolean.

This option supports only integer data types (`char`, `signed` and `unsigned` integer types, and enumerated types). For example, the data type `boolean_t` defined as follows:

```
typedef signed char boolean_t;
```

Default:

No data types specified as Boolean.

Example Shell Script Entry:

```
polyspace-c -misra2 all-rules -boolean-types
bool_type1,bool_type2,bool_type3
```

Allowed pragmas

Use this option with the `-misra2` option to specify undocumented pragma directives for which MISRA C rule 3.4 should not be applied. MISRA C rule

3.4 requires checking that all pragma directives are documented within the documentation of the compiler.

The command line syntax for this option is

```
-allowed-pragmas pragma1,pragma2,pragma3 ...
```

where *pragma1*,*pragma2*, ... are undocumented pragma directives.

Default:

No undocumented pragma directives specified

Example Shell Script Entry:

```
polyspace-c -misra2 AC-AGC-OBL-subset -allowed-pragmas
pragma01,pragma02,pragma03
```

Calculate code complexity metrics

Specify whether to calculate software quality metrics, such as cyclomatic number, during verification.

Note This option requires a Polyspace Client for C/C++license.

Settings

Default: On

 On

Calculate software quality metrics, including project metrics, file metrics, and function metrics.

 Off

Do not calculate software quality metrics.

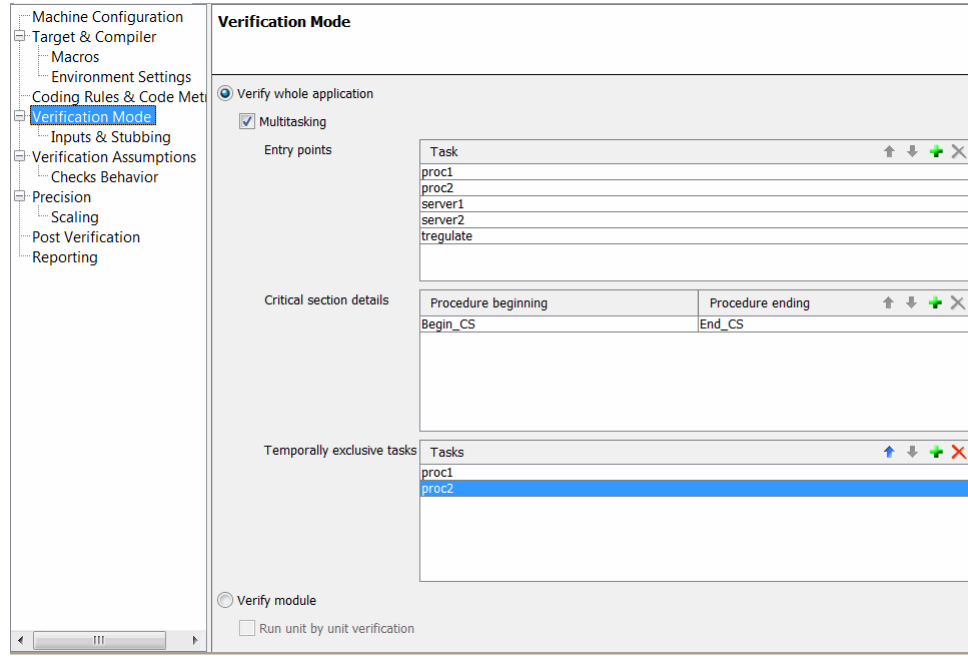
Tips

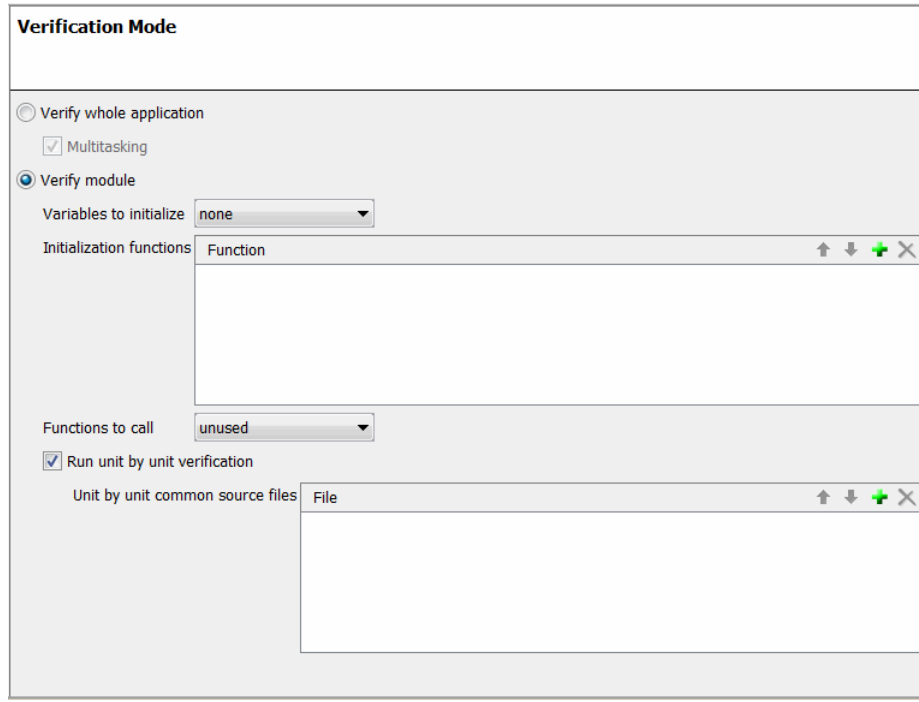
- You can view software quality metrics data in the Polyspace Metrics Web interface, or by running a Software Quality Objectives report from the Polyspace verification environment.
- Project metrics include number of recursions, number of include headers, and number of files.
- File metrics include comment density, and number of lines.
- Function metrics include cyclomatic number, number of static paths, number of calls, and Language scope.

Command-Line Information

Parameter: `-code-metrics`
Example: `polyspace-c -code-metrics`

Verification Mode




Verification Mode
Verify whole application
Multitasking
Verify module
Variables to initialize
none
Initialization functions
Function
Functions to call
unused
Run unit by unit verification
Unit by unit common source files
File


Inputs & Stubbing
Inputs
Variable/function range setup
Edit...
Ignore default initialization of global variables
Stubbing
No automatic stubbing
Functions to stub
Function

In this section...

Verify whole application

Use the function main from the given set of files to verify the whole application

Multitasking

Verify multitasking code

Entry points

This option is used to specify the tasks/entry points to be analyzed by the verification, using a Comma-separated list with no spaces.

These entry points must not take parameters. If the task entry points are functions with parameters they should be encapsulated in functions with no parameters, with parameters passed through global variables instead.

Using Polyspace verification, c tasks must have the prototype "void *task_name*(void);".

Example Shell Script Entry:

```
polyspace-c -entry-points proc1,proc2,proc3 ...
```

Critical section details

```
-critical-section-begin "proc1:cs1[,proc2:cs2]"
```

and

```
-critical-section-end "proc3:cs1[,proc4:cs2]"
```

These options specify the procedures beginning and ending critical sections, respectively. Each uses a list enclosed within double speech marks, with list entries separated by commas, and no spaces. Entries in the lists take the form of the procedure name followed by the name of the critical section, with a colon separating them.

These critical sections can be used to model protection of shared resources, or to model interruption enabling and disabling.

Note This option cannot be used with the `main-generator` option.

Default:

no critical sections.

Example Shell Script Entry:

```
polyspace-c -critical-section-begin "start_my_semaphore:cs" \

-critical-section-end "end_my_semaphore:cs"
```

Temporally exclusive tasks

This option specifies the name of a file. That file lists the sets of tasks which never execute at the same time (temporal exclusion).

The format of this file is :

- one line for each group of temporally excluded tasks,
- on each line, tasks are separated by spaces.

Note This option cannot be used with the `main-generator` option.

Default:

No temporal exclusions.

Example Task Specification file

File named 'exclusions' (say) in the 'sources' directory and containing:

```
task1_group1 task2_group1

task1_group2 task2_group2 task3_group2
```

Example Shell Script Entry :

```
polyspace-c -temporal-exclusions-file sources/exclusions \

    -entry-points task1_group1,task2_group1,task1_group2,\

    task2_group2,task3_group2 ...
```

Generate a main

This option activates the Polyspace main generator.

When you select this option, the main generator automatically generates a main unless a main already exists in your verification.

The generated main has the following behavior.

1. It initializes any variables identified by the option `-variables written-before-loop`.
2. It calls any functions specified by the option `-functions-called-before-loop`. This could be considered an initialization function.
3. It initializes any variables identified by the option `-variables written-in-loop`.
4. It calls any functions specified by the option `-functions-called-in-loop`.
5. It calls any functions specified by the option `-functions-called-after-loop`. This could be a terminate function for a cyclic program.

The following sections describe each of these options in detail.

Note The main generator is always active for client verifications.

Main for Generated Code

The following example shows how to use the main generator options to generate a main for a cyclic program, such as code generated from a Simulink® model.

```
init parameters  \\ -variables-written-before-loop
init_fct()    \\ -functions-called-before-loop
 while(1){   \\ start main loop
 init inputs   \\ -variables-written-in-loop
```

```
 step_fct()   \\ -functions-called-in-loop
}
terminate_fct()  \\ -functions-called-after-loop
```

Main Generator Behavior for Polyspace Software

This same options can be used for both Polyspace Client for C/C++ and Polyspace Server™ for C/C++ verifications, but the default behavior differs between the two:

- **Server Verification** – You have the choice whether to activate the main generator.
- **Client Verification** – The main generator is always activated.

Polyspace Client for C/C++ Main Generator

For client verifications, you do not need to determine whether the code contains a "main" or not. Polyspace Client for C/C++ product automatically checks your code for a main.

- If a main exists in the set of files, the verification uses that main.
- If a main does not exist, the tool generates a main using the options you specify.

Polyspace Server for C/C++ Main Generator

If you do not select the `-main-generator` option, a Polyspace Server for C/C++ verification stops if it does not detect a main. This behavior can help isolate files missing from the verification.

When you select the `-main-generator` option, the Polyspace Server for C/C++ product checks your code for a main.

- If a main exists in the set of files, the verification uses that main.
- If a main does not exist, the tool generates a main using the options you specify.

Variables to initialize

Specifies how generated main initializes global variables. Use with `-main-generator`.

Settings available:

- `none` — No global variable will be written by the main.
- `public` — Every variable except `static` and `const` variables are assigned a random value, representing the full range of possible values.
- `all` — Every variable is assigned a random value, representing the full range of possible values.
- `custom` — Only variables present in the list are assigned a random value, representing the full range of possible values.

Command-Line Information

Parameter: `-main-generator-writes-variables`

Example

```
polyspace-c  main-generator -main-generator-writes-variables all

polyspace-c -main-generator -main-generator-writes-variables
custom=variable_a,variable_b
```

Initialization functions (`-function-called-before-main`)

This option is used with `-main-generator-calls` to specify a function, or list of functions, that are called before all selected functions in `main`.

Eligible functions:

Every function defined in the source code is considered eligible.

If the function is not defined in the source code, the verification continues with a warning message.

Example:

```
polyspace-c -main-generator-calls unused
-function-called-before-main MyFunction1,MyFunction2
```

Functions to call

Use this option with the `-main-generator` option to specify the functions to be called.

Set this option to `unused` (default) when you run a unit-by-unit verification.

Eligible functions:

Every function declared and defined in the source code is considered eligible.

The list of functions is a list of short names (names without signature) separated by comas. If the name of a function from the list is associated with a function that is not defined in the source code, the Polyspace verification stops and displays an error message. If the name of a function from the list is ambiguous, all the functions with the same short name are called. If a function from the list is not eligible, Polyspace verification stops and displays an error message. This error message is also in the log file.

Values:

- **`none`** – No function is called. This can be used with a multitasking application without a main, for instance.
- **`unused` (default)** – Call all functions not already called within the code. Inline functions will not be called by the generated main.
- **`all`** – all functions except inline will be called by the generated main.
- **`custom`** – Only functions present in the list are called from the main. Inline functions can be specified in the list and will be called by the generated main.

An inline (static or extern) function is not called by the generated main program with values all or unused. An inline function can only be called with custom value:

```
-main-generator-calls custom=my_inlined_func
```

Example:

```
polyspace-c -main-generator -main-generator-calls
custom=function_1,function_2
```

Run unit by unit verification

This option creates a separate verification job for each source file in the project.

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

Note Unit by unit verification is available only for server verifications. It is not compatible with multitasking options such as `-entry-points`.

Default:

Not selected

Example Shell Script Entry:

```
polyspace-c -unit-by-unit
```

Unit by unit common source files

Specifies a list of files to include with each unit verification. These files are compiled once, and then linked to each unit before verification. Functions not included in this list are stubbed.

Default:

None

Example Shell Script Entry:

```
      polyspace-c -unit-by-unit-common-source
c:/polyspace/function.c
```

Calibration variables

Specify how the generated main initializes variables (parameters) of cyclic system. Initialization occurs before cyclic loop.

Settings available:

- `none` (default) — No variable will be written by the main.
- `public` — Every variable except static and const variables are assigned a "random" value, representing the full range of possible values
- `all` — Every variable except const variables are assigned a "random" value, representing the full range of possible values
- `custom` — Only variables present in the list are assigned a "random" value, representing the full range of possible values

Example

```
polyspace-c -main-generator -variables-written-before-loop none
polyspace-c -main-generator -variables-written-before-loop
custom=variable_a,variable_b
```

Input variables

Specifies how the generated main initializes variables that are inputs to a cyclic system. The generated main resets variables at each iteration of the cyclic loop.

Settings available:

- `none` (default) — No variable will be written by the main.
- `public` — Every variable except static and const variables are assigned a "random" value, representing the full range of possible values
- `all` — Every variable except const variables are assigned a "random" value, representing the full range of possible values

- `custom` — Only variables present in the list are assigned a "random" value, representing the full range of possible values

Example

```
polyspace-c -main-generator -variables-written-in-loop none
polyspace-c -main-generator -variables-written-in-loop
custom=variable_a,variable_b
```

Initialization functions (`-functions-called-before-loop`)

Specify an initialization function, or list of functions that are called by the generated main before the cyclic loop.

The generated main does the following:

1. Initialize variables
2. Call initialization functions that you specify *`MyInitFunction1`*, *`MyInitFunction2`*
3. Execute cyclic loop with calls to functions that you specify using the option `-functions-called-in-loop`.
4. Call termination functions that you specify using the option `-functions-called-after-loop`.

Example shell script entry:

```
polyspace-c -main-generator -function-called-before-loop
MyInitFunction,MyInitFunction2
```

Cyclic functions

Specify the functions to be called within the cyclic loop of the generated main.

Possible values:

- `none` — No function called. Use this when verifying a multitasking application without a main.

- `unused` (default) — Every function is called by the generated main unless the function is called elsewhere by the code being verified.
- `all` — Except for inline functions, all functions are called by the generated main.
- `custom` — Only functions in your specified list are called by the generated main. You can specify inline functions in your list.

An inline (`static` or `extern`) function is not called by the generated main program if you specify `all` or `unused`.

Note If you specify the `unused` option, the generated main might call functions that are also called by a function pointer, that is, these functions might be called twice.

Example:

```
polyspace-c -main-generator -functions-called-in-loop public

polyspace-c -main-generator -functions-called-in-loop
custom=function_1,function_2
```

Termination functions

Specify a function or list of functions that are called by the generated main after the cyclic loop.

The generated main does the following:

1. Initialize variables
2. Call initialization functions that you specify using the option `-functions-called-before-loop`
3. Execute cyclic loop with calls to functions that you specify using the option `-functions-called-in-loop`

4 Call the termination functions that you specify *MyTermFunction1*, *MyTermFunction2*

Example shell script entry:

```
polyspace-c -main-generator -function-called-after-loop
MyTermFunction1,MyTermFunction2
```

Variable/function range setup

This option permits the setting of specific data ranges for a list of given global variables.

For more information, see "Specify Data Ranges for Variables and Functions (Contextual Verification)".

File format:

The file filename contains a list of global variables with the below format:

```
variable_name val_min val_max <init|permanent|globalassert>
```

Variables scope:

Variables do not have to be defined variables.

Note Only one mode can be applied to a global variable.

No checks are added with this option except for `globalassert` mode.

Some warning can be displayed in log file concerning variables when format or type is not in the scope.

Default:

Disable.

Example shell script entry:

```
polyspace-c -data-range-specifications range.txt ...
```

Do not consider all global variables to be initialized

This option specifies that Polyspace verification should not take into account default initialization defined by ANSI C. When this option is not used, default initialization are

- 0 for integers
- 0 for characters
- 0.0 for floats

With the option in use, all global variable will be treated as non initialized, and therefore cause a red `NIV` error if they are read before being written to.

NIV Example 1:

```
1  int var;  // line 1: -no-def-init-glob does not follow ANSI C
2             // initialization behavior on global variables
3  int main(void)
4  {
5  int res;
6  res = var; // line 6: red NIV using -no-def-init-glob
7  return res;
8  }
```

To change the red `NIV` to green using the `-no-def-init-glob` option, change line 1 to:

```
int var = 0;
```

NIV Example 2 — With Tasks:

```
 // -entry-points t1, t2,
 // -no-def-init-glob

1  int var;          // line 1
2  void main(void) { var = 1;}   // line 2
3  void t1(void)
```

```
4  {
5   int res;
6   while(1) {
7    res = var;   // line 7: green NIV using -no-def-init-glob
8         // because var has been initialized before read
9   }
10 }
11 static var2;        // line 11
12 void t2(void)
13 {
14  int res;
15  while(1) {
16  res = var2;  // line 16: red NIV using -no-def-init-glob because
17              // var2 is not initialized before first read
18  }
19 }
```

At line 7, the variable has a green `NIV` because the variable has been written before first read in task `t1()`. At line 16, the verification reports a red `NIV` because a global variable has not been explicitly written before first read.

To workaround the red `NIV` at line 16, change line 11 to:

```
static var2 = 0;
```

Example Shell Script Entry :

```
polyspace-c -no-def-init-glob ...
```

No automatic stubbing

By default, Polyspace verification automatically stubs all functions. When this option is used, the list of functions to be stubbed is displayed and the verification is stopped.

Benefits:

This option may be used where

- The entire code is to be provided, which may be the case when verifying a large piece of code. When the verification stops, it means the code is not complete.
- Manual stubbing is preferred to improve the selectivity and speed of the verification.

Note This option cannot be used with the `permissive-stubber` option.

Default:

All functions are stubbed automatically

Functions to stub

Specifies functions that you want the software to stub.

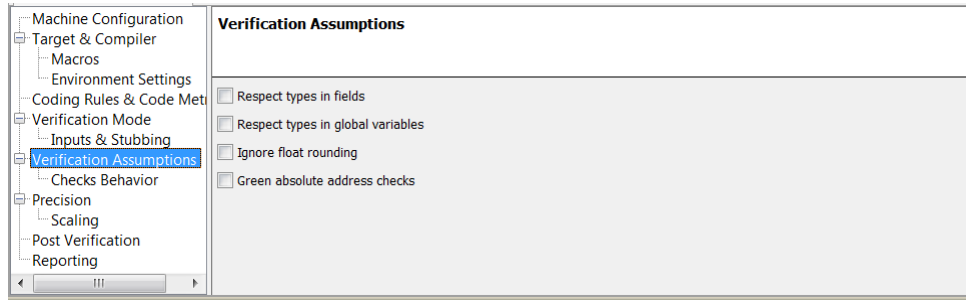
Enter a comma-separated list of functions. For example, `function_1,function_2`.

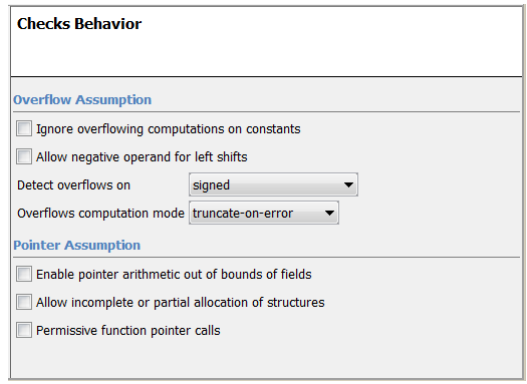
Spaces are not allowed for C functions.

Example Shell Script Entry:

```
polyspace-c -functions-to-stub function_1,function_2 ...
```

Verification Assumptions





In this section...

In this section...

Respect types in fields

This is a scaling option, designed to help process complex code. When it is applied, Polyspace verification assumes that structure fields not declared as containing pointers are never used for holding pointer values. This option should only be used with Type-safe code, when it does not cause a loss of precision. See also -respect-types-in-globals.

In the following example, we will lose precision using option –respect-types-in-fields option:

```
struct {
 unsigned x;
 int f1;
 int *z[2];
} S1;

void funct2(void) {
 int *tmp;
 int y;
 ((int**)&S1)[0] = &y;  /* S1.x points on y */
 tmp = (int*)S1.x;
 y=0;
 *tmp = 1;      /* write 1 into y */
 assert(y==0);
}
```

Polyspace verification will not take care that S1.x contains the address of y resulting a green assert.

Default:

Polyspace verification assumes that structure fields may contain pointer values.

Example Shell Script Entry:

```
polyspace-c -respect-types-in-fields ...
```

Respect types in global variables

This is a scaling option, designed to help process complex code. When it is applied, Polyspace verification assumes that global variables not declared as containing pointers are never used for holding pointer values. This option should only be used with Type-safe code, when it does not cause a loss of precision. See also -respect-types-in-fields.

In the following example, we will lose precision using option –respect-types-in-globals option:

```
int x;
void t1(void) {
 int y;
 int *tmp = &x;
 *tmp = (int)&y;
 y=0;
 *(int*)x = 1; // x contains address of y
 assert (y == 0); // green with the option
}
```

Polyspace verification will not take care that x contains the address of y resulting a green assert.

Default:

Polyspace verification assumes that global variables may contain pointer values.

Example Shell Script Entry:

```
polyspace-c -respect-types-in-globals ...
```

Ignore float rounding

Without this option, Polyspace verification rounds floats according to the IEEE® 754 standard: simple precision on 32-bits targets and double precision on target which define double as 64-bits.

With the option, **exact** computation is performed.

Example:

```
void ifr(float f)
{
 double a,b;
  a = 0.2;
  b = 0.2;

  if ( a + b == 0.4) {
  // reached whether -ignore-float-rounding is used or not
  assert (1);
  f = 1.0F*f;
  }
 else {
  assert (1);
  f = 1.0F * f;
  // reached only when -ignore-float-rounding is not used
 }
}
```

Using this option can lead to different results compared to the "real life" (compiler and target dependent): Some paths will be reachable or not for Polyspace verification while they are not (or are) depending of the compiler and target. So it can potentially give approximate results (green should be unproven). This option has an impact on OVFL checks on floats.

However, this option allows reducing the number of unproven checks because of the “delta” approximation.

For example:

- FLT_MAX (with option set) = 3.40282347e+38F

- FLT_MAX (following IEEE 754 standard) = 3.40282347e+38F ± Δ

```
void ifr(float f)
{
 double a,b;
 a = 0.2;
 b = 0.2;

 if ( a + b == 0.4) {
  assert (1);
  f = 1.0F*f;   // Overflow never occurs because f <= FLT_MAX.
        // reached when -ignore-float-rounding is used
 }
 else {
  assert (1);
  f = 1.0F * f;   // OVFL could occur when f = (FLT_MAX + D)
        // reached when -ignore-float-rounding is not used
 }
}
```

Default:

IEEE 754 rounding under 32 bits and 64 bits.

Example Shell Script Entry:

```
polyspace-c -ignore-float-rounding ...
```

Green absolute address checks

If you know that the absolute addresses in your code are valid, you can specify this option to make all ABS_ADDR checks green. Otherwise, the software generates an orange ABS_ADDR check when an absolute address is assigned to a pointer. This is because the software has no information about the absolute address and therefore cannot verify, for example, the address, availability of memory, and initialization of memory.

The software permits memory access to the absolute address after generating the orange ABS_ADDR check for the first assignment operation. IDP and

NIV checks for memory access operations after the first assignment operation are green.

Default:

Orange ABS_ADDR check generated when an absolute address is assigned to a pointer.

Example Shell Script Entry:

```
polyspace-c -green-absolute-address-checks ...
```

Ignore overflowing computations on constants

This option specifies that the verification should be permissive with regards to overflowing computations on constants. Note that it deviates from the ANSI C standard.

For example,

```
char x = 0xff;
```

causes an overflow according to the standard, but if it is analyzed using this option it becomes effectively the same as

```
char x = -1;
```

With this second example, a red overflow will result regardless of the use of the option.

char x = (rnd?0xFF:0xFE);

Default:

`char x = 0xff;` causes an overflow

Example Shell Script Entry:

```
polyspace-c -ignore-constant-overflows ...
```

Allow negative operand for left shifts

This option permits a shift operation on a negative number.

According to the ANSI C standard, such a shift operation on a negative number is illegal. For example:

```
-2 << 2
```

With this option in use, Polyspace verification considers the operation to be valid. In the above example, the result would be

```
-2 << 2 = -8
```

Note According to the ANSI-C standard, if the value of the right operand is negative, or is greater than or equal to the width in bits of the promoted left operand, the behavior is undefined. Therefore, the verification flags it as illegal shift operation.

Default:

A shift operation on a negative number causes a red error.

Example Shell Script Entry:

```
polyspace-c -allow-negative-operand-in-shift ...
```

Detect overflows on

Specifies how verification handles overflowing computations on integers.

Possible settings are:

- **`none`** – The verification does not check for integer computation overflows. If a computation value exceeds the range of the result type, the result is wrapped, and no OVFL check is reported.

  For example, `MAX_INT + 1` wraps to `MIN_INT`.

- **`signed`** (default) – Verification checks all signed integer computations and signed integer to signed integer conversions. The option `-scalar-overflows-behavior` specifies whether results for signed integers are restricted to an acceptable value or wrapped.

  For unsigned integer operations, the results are wrapped, and no OVFL check is reported.

  This behavior conforms to the ANSI C (ISO C++) standard.

- **`signed-and-unsigned`** – Verification checks all integer computations and integer conversions, including conversions that cause a change of signs.

  The option `-scalar-overflows-behavior` specifies whether operation results are restricted to an acceptable value or wrapped.

  This behavior is more strict than the ANSI C (ISO C++) standard requires.

Consider the examples below.

Example 1

When you use the `signed-and-unsigned` option, the following example generates an error:

```
unsigned char x;
x = 255;
x = x+1;   //overflow due to this option
```

Using the `none` option, however, the example does not generate an error.

```
unsigned char x;
x = 255;
x = x+1;  // turns x into 0 (wrap around)
```

Example 2

When you use the `signed-and-unsigned` option, the following example does not generate an error:

```
unsigned char Y=1;
Y = ~Y;  //no overflow and GREEN OVFL check
```

As the ~ operator applies directly to an `unsigned char` variable, there is no risk of an overflow. The outcome of the operation is a wrapped-around `unsigned char` variable. In this example, the Polyspace verification continues with the value 254 assigned to the `unsigned char` variable `Y`.

Example Shell Script Entry:

```
polyspace-c -scalar-overflows-checks signed ...
```

Overflows computation mode

Specifies how verification computes the results of overflowing integer computations or integer conversions.

Possible settings are:

- **`truncate-on-error`** (default) — Result of an overflowing operation is restricted to an acceptable value. If the check is red, verification stops (in the current code). If the check is orange, verification continues with restricted value.
- **`wrap-around`** — Result of an overflowing operation wraps around the type range. The check has no impact on values for the rest of the verification.

  For example, `MAX_INT + 1` wraps to `MIN_INT`.

Example Shell Script Entry:

```
polyspace-c -scalar-overflows-behavior wrap-around ...
```

Enable pointer arithmetic out of bounds of fields

This option enables navigation within a structure or union from one field to another, within the rules defined below. It automatically sets the -size-in-bytes option.

Default

By default, when a pointer points to a variable then the size of the objected pointed to is that of that variable - regardless of whether it is contained within a bigger object, like a structure. Therefore, going out of the scope of

this variable leads to a red IDP check (Illegal Dereference Pointer). This is illustrated below.

```
struct S {char a; char b; int c;} x;
 char *ptr = &x.b;
 ptr ++;
 *ptr = 1; // red on the dereference, because the pointed
object was "b"
```

Using this option

When this option is used in the above option, Polyspace verification considers that the object pointed to is now the host object "x". The "ptr" pointer is in fact pointing to &x, with the correct offset to the field “b” within the structure of type S (inter-fields and end-padding included). Therefore, the dereference becomes green

Consider a second example:

```
int main()
{

 struct S {
  char a;
  /* 3 bytes of padding between 'a', 'b' */
  int b;
  int c;
  char d[3];
  unsigned char e:7;
  char f;
  /* 3 bytes of end padding */
 } x;
 char *ptr;
 struct Nesting_S {
  struct S s;
  int c;
  char buf[8];
  int d;
 } z

 struct S *ptr0;
```

```
 char *ptr;

 ptr = &z.s.f;
 ptr += 4;
 *(int *)ptr = 10; /* access to z.c, Green IDP */

 ptr0 = &z.s;
 ptr = &ptr0->f;
 ptr += 4;

 *(int *) ptr = 10; /* access to z.c, Green IDP */

 ptr = &z.buf[0];
 ptr += 8;
 *(int *)ptr = 10; /* access to z.d, Green IDP */

 return (0);
}
```

In the third example below, the *ptr access is red regardless of whether the option is set or not.

With the option set, the ptr pointer points to the structure+offset z.s, and ptr can safely navigate within this structure z.s, but z.c is outside it.

Without the option, the ptr pointer points to z.s.f, which is only 1 byte long. So no navigation is allowed, not even within z.s.

```
ptr = (char *)z.s.f; ptr += 4; *ptr = 10; // ptr points to the
first byte of c:
```

Allows incomplete or partial allocation of structures

This option allows incomplete or partial allocation of structures. This allocation can be made by malloc or cast .

The example below shows an example using malloc. Further explanation can be found in the section describing the partial and incomplete allocation of structures. Also refer to the -allow-ptr-arith-on-struct section.

```
typedef struct _little { int a; int b; } LITTLE;
```

```
typedef struct _big { int a; int b; int c; } BIG;
BIG *p = malloc(sizeof(LITTLE));
```

Default results

```
 p->a = 0 ;   // red pointer out of its bounds
or p->b = 0 ;  // red pointer out of its bounds
or p->c = 0 ;  // red pointer out of its bounds
```

Results using this option

```
if (p!= ((void *) 0) ) {
 p->a = 0 ;   // green pointer within bounds
or p->b = 0 ;  // green pointer within bounds
or p->c = 0 ;  // red pointer out of its bounds
}
```

Permissive function pointer calls

By default, Polyspace allows a function pointer to call a function only if both the function pointer and function types are identical. For example, a function with type

```
int f(int*)
```

cannot be called by a function pointer of type

```
int fptr(void*)
```

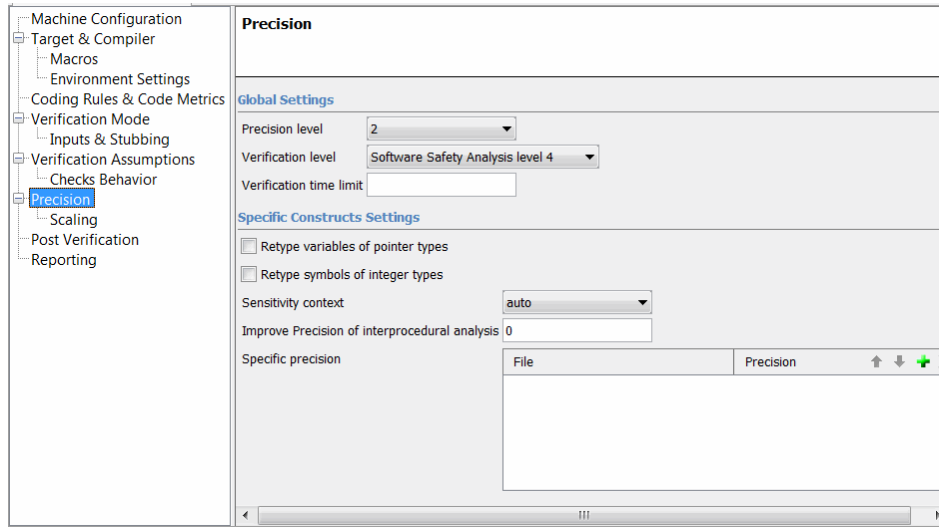
If this option is set, Polyspace allows such calls.

Caution With applications that use function pointers extensively, this option can cause a significant loss in performance and a higher number of orange checks as Polyspace has to consider more execution paths.

Example Shell Script Entry:

```
polyspace-c -permissive-function-pointer ...
```

Precision



In this section...

Precision level

This option specifies the precision level to be used. It provides higher selectivity in exchange for more verification time, therefore making results review more efficient and hence making bugs in the code easier to isolate. It does so by specifying the algorithms used to model the program state space during verification.

Begin with the lowest precision level. Red errors and gray code can then be addressed before rerunning the Polyspace verification with higher precision levels.

Benefits:

- A higher precision level contributes to a higher selectivity rate, making results review more efficient and hence making bugs in the code easier to isolate.
- A higher precision level also means higher verification time
  - -O0 corresponds to static interval verification.
  - -O1 corresponds to complex polyhedron model of domain values.
  - -O2 corresponds to more complex algorithms to closely model domain values (a mixed approach with integer lattices and complex polyhedrons).
  - -O3 is only suitable for code smaller than 1000 lines of code. For such codes, the resulting selectivity might reach high values such as 98%, resulting in a very long verification time, such as an hour per 1000 lines of code.

Default:

-O2

Example Shell Script Entry:

```
polyspace-c -O1 -to pass4 ...
```

Verification level

This option specifies the phase after which the verification will stop.

Benefits:

This option provides improved selectivity, making results review more efficient and making bugs in the code easier to isolate.

- A higher integration level contributes to a higher selectivity rate, leading to "finding more bugs" with the code.
- A higher integration level also means higher verification time

---

Note Begin by running `-to pass0` (`Software Safety Analysis level 0`) You can then address red errors and gray code before rerunning the verification with higher integration levels.

Possible values:

- `c-compile` or `"C Source Compliance Checking"`
- `c-to-il` or `C to Intermediate Language` — Not available from the Polyspace verification environment.
- `pass0` or `"Software Safety Analysis level 0"`
- `pass1` or `"Software Safety Analysis level 1"`
- `pass2` or `"Software Safety Analysis level 2"`
- `pass3` or `"Software Safety Analysis level 3"`
- `pass4` or `"Software Safety Analysis level 4"` — Default
- `other`

Note If you use *-to other* then Polyspace verification will continue until you stop it manually (via `kill-rte-kernel`) or stops until it has reached *pass20*.

Default:

pass4

Example Shell Script Entry:

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

polyspace-c -to pass0 ...
```

Verification time limit

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

If the verification does not complete within the specified time, the verification fails.

You can specify fractions of an hour in decimal form. For example:

- `-timeout 5.75` – Five hours, 45 minutes.
- `-timeout 3,5` – Three hours, 30 minutes.

Example Shell Script Entry :

```
polyspace-c -timeout 5.75 ...
```

Retype variables of pointer types

Retype variables of pointer types to improve precision of pointer conversions chain

The software replaces the original type by the aliased object type when a symbol of pointer type aliases to a single type of objects.

For example, the `assert` statement in the following code can be proved using `-retype-pointer` option:

```
struct A {int a; char b;} s = {1,2};
char *tmp = (char *)&s;
struct A *pa = (struct A*)tmp;
assert((pa->a == 1) && (pa->b == 2));
```

This principle can be applied to fields of struct/unions of a pointer type. However, this option sets the `-size-in-bytes` option.

See also `-retype-int-pointer`.

Default:

Disable by default

Example Shell ScriptEntry:

```
polyspace-c -retype-pointer ...
```

Retype symbols of integer types

Retype variables of pointer to signed or unsigned integer types in order to improve precision of pointer conversions chain.

The software replaces the original type by the aliased object type when a symbol of pointer type aliases to a single type of objects. It applies only on symbols of signed or unsigned integer types.

For example, the following `assert` statement can be proved using this option:

```
void function(void)
 {
  struct S1 {
   int x;
   int y;
   int z;
   char t;
  } s1 = {1,2,3,4};
 struct S2 {
  int first;
  void *p;
 } s2;
 int addr;
 addr = (int)(&s1);
 assert(((struct S1 *)addr)->y == 2); // ASRT is verified
 s2.first = (int)(&s1);
 assert(((struct S1 *)s2.first)->y == 2); // ASRT is verified
}
```

However, this option sets `-size-in-bytes` and has no effect on:

- Global symbols of integer types if `-respect-types-in-globals` is set
- Fields of structure or unions of integer types if `-respect-types-in-fields` is set

Some sides effects can be noticed on Polyspace checks concerning initialization on variables which can be stated as initialization on pointer check (NIP).

This option sets the `-retype-pointer` option internally, so `-retype-pointer` and `-retype-int-pointer` are exclusive in the Polyspace verification environment.

This option should be used on:

- **Code with memory mapping** – When constant big structures (global variable) are declared with a pointer and points to const structure, setting the option will consider that the pointer and the pointer structure are synonyms (aliased) and precision of the result will increase. Option to set: `-retype-pointer`.
- **Code close to the communication layer API** (code with lots of cast in (void *)) – When code contains low level drivers, generic pointer (void *) can be used. It is recommended to use this with an `-inline` of the functions containing these casts. Options to set: `-retype-pointer -inline`.
- **Code in which MISRA rule 11.2 is violated** – When integers contains pointers, precision can be improved when setting an option. Option to set: `-retype-int-pointer`.

These options are not set by default because they all change the option `-size-in-bytes`. Therefore, precision can decrease and some red IDP checks may be affected. In addition, using these options will consider "x" (previously int) as a pointer. This results in checks changing category (NIV to NIP).

Default:

Disable by default

Example Shell ScriptEntry:

```
polyspace-c -retype-int-pointer...
```

Sensitivity context

Add call context information for checks contained in given functions. For example, if one call of the function results in a red check, and another call results in a green check, the call information and color for both calls is kept in the function check.

- `none` — No context sensitivity.
- `auto` — Automatically select functions for which context sensitivity is applied.
- `custom` — Apply context sensitivity to functions that you specify.

Example Shell Script Entry:

```
polyspace-c -context-sensitivity -auto ...
```

Improve precision of interprocedural analysis

This option is used to improve interprocedural verification precision within a particular pass (see -to *pass1*, *pass2*, *pass3* or *pass4*). The propagation of information within procedures is done earlier than usual when this option is specified. That results in improved selectivity and a longer verification time.

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

- a level 1 verification in (with) (pass1) will provide results equivalent to level 1 or 2 in the (without) verification
- a level 1 verification in (with) can last x times more than a cumulated level 1+2 verification from (without). "x" might be exponential.
- the same applies to level 2 in (with) equivalent to level 3 or 4 in (without), with potentially exponential verification time for (a)

Gains using the option

- (+) highest selectivity obtained in level 2. no need to wait until level 4

- (-) This parameter increases exponentially the verification time and might be even bigger than a cumulated verification in level 1+2+3+4
- (-) This option can only be used with less than 1000 lines of code

Default:

0

Example Shell Script Entry:

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

Specific Precision

This option is used to specify the list of .c files to be analyzed with a different precision from that specified generally -O(0-3) for this verification.

In batch mode, each specified module is followed by a colon and the desired precision level for it. Any number of modules can be specified in this way, to form a comma-separated list with no spaces.

Default:

All modules are treated with the same precision.

Example Shell Script Entry:

```
polyspace-c -O1 \
-modules-precision myMath:O2,myText:O1, ...
```

Optimize huge static initializers

When variables are defined with huge static initialization, scaling problems may occur during the compilation phase. This option approximates the initialization of array types of integer, floating point, and char types (included string) if required.

It can speed up the verification, but may decrease precision for some applications

Default:

Option not set.

Example Shell Script Entry:

```
polyspace-c -no-fold ...
```

Reduce task complexity

This scaling option can be used to reduce task complexity (see also -entry-points).

It uses a slightly less precise model of pointer/thread interaction compared to that used by default, and is likely to prove helpful when there are a lot of pointers in an application. See “Reduce Verification Time” for more explanation of when to use it.

It causes a loss of precision:

- more orange checks
- loss of precision when shared variables are reads via pointers.

Default:

disabled by default.

Example Shell Script Entry :

```
polyspace-c -lightweight-thread-model ...
polyspace-c -lwtm ...
```

Inline

A scaling option that creates a clone of a each specified procedure for each call to it.

Cloned procedures follow a naming convention viz:

```
procedure1_pst_cloned_nb,
```

where nb is a unique number giving the total number of cloned procedures.

Such an inlining allows the number of aliases in a given procedure to be reduced, and may also improve precision.

Restrictions :

- Extensive use of this option may duplicate too much code and may lead to other scaling problems. Carefully choose procedures to inline.
- This option should be used in response to the inlining hints provided by the alias verification
- This option should not be used on main, task entry points and critical section entry points

Depth of analysis inside structures

This is a scaling option to limits the depth of verification into nested structures during pointer verification.

This option is only available for C and C++.

Default:

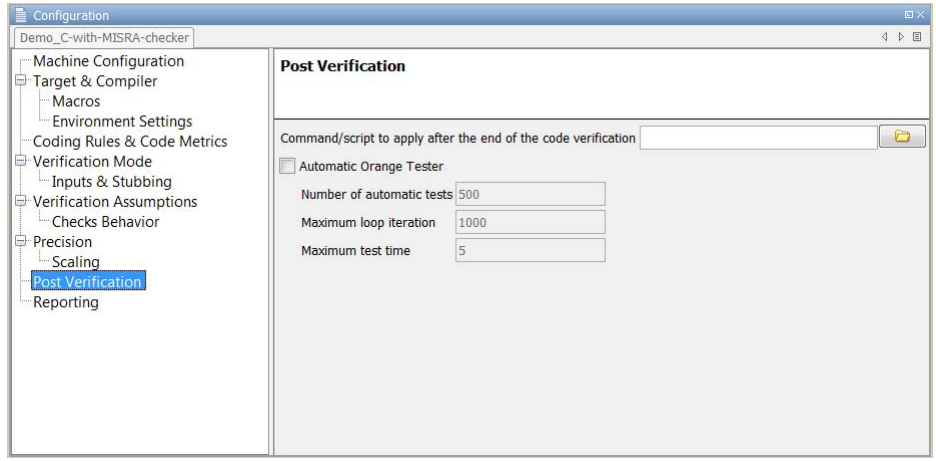
There is no fixed limit.

Example Shell Script Entry:

```
polyspace-c -k-limiting 1 ...
```

In this example above, verification will be precise to only one level of nesting.

Post Verification



In this section...

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

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

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

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

Note Depending of the architecture used (notably when using remote launcher), the script can be executed on the client side or the server side.

Default:

No command.

Example Shell Script Entry – file name:

This example shows how to send an email to tip the client side off that his verification has been ended. So the command looks like:

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

where `end_email` is your Perl script.

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

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

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

```
%POLYSPACE_C%\Verifier\bin\polyspace-c.exe -post-analysis-command
%POLYSPACE_C%\Verifier\tools\perl\win32\bin\perl.exe
<absolute_path>\end_email
```

Automatic Orange Tester

Activates the Automatic Orange Tester at the end of static verification. By running dynamic stress tests on unproven code, the Automatic Orange

Test can identify orange checks that are potential run-time errors. See "Automatically Test Orange Code".

Note The software still supports the option `-prepare-automatic-tests`. However, support for this option will cease in a future release.

With `-automatic-orange-tester`, the software does not support the following:

- `-div-round-down`
- `-char-is-16bits`
- `-short-is-8bits`
- Global asserts in the code of the form `Pst_Global_Assert(A,B)`

In addition, there are restrictions on other options. You must not specify the following with `-automatic-orange-tester`:

- `-target [c18 | tms320c3c | x86_64 | sharc21x61]`
- `-data-range-specification` (in global assert mode)

You must use the `-target mcpu` option together with `-pointer-is-32bits`.

Default :

Disabled

Example Shell Script Entry :

```
polyspace-c -automatic-orange-tester ...
```

Number of automatic tests

Specify total number of test cases that you want to the Automatic Orange Tester to run. Running more tests increases the chances of finding a run-time error, but takes more time to complete.

Option is available only if you select Automatic Orange Tester (`-automatic-orange-tester`). The maximum value that the software supports is 100,000.

Default:

500

Example Shell Script Entry:

```
polyspace-c -automatic-orange-tester
-automatic-orange-tester-tests-number 550 ...
```

Maximum loop iterations

Specify maximum number of iterations for a loop after which the Automatic Orange Tester considers the loop to be an infinite loop. A larger number of iterations decreases the chances of incorrectly identifying an infinite loop, but takes more time to complete.

Option is available only if you select Automatic Orange Tester (`-automatic-orange-tester`). The maximum value that the software supports is 1000.

Default:

1000

Example Shell Script Entry:

```
polyspace-c -automatic-orange-tester
-automatic-orange-tester-loop-max-iteration 800 ...
```

Maximum test time

Maximum time (in seconds) allowed for a test before Automatic Orange Tester moves on to next test. Increasing test time reduces number of tests that time out, but increases total verification time.

Option is available only if you select Automatic Orange Tester (`-automatic-orange-tester`). The maximum value that the software supports is 60.

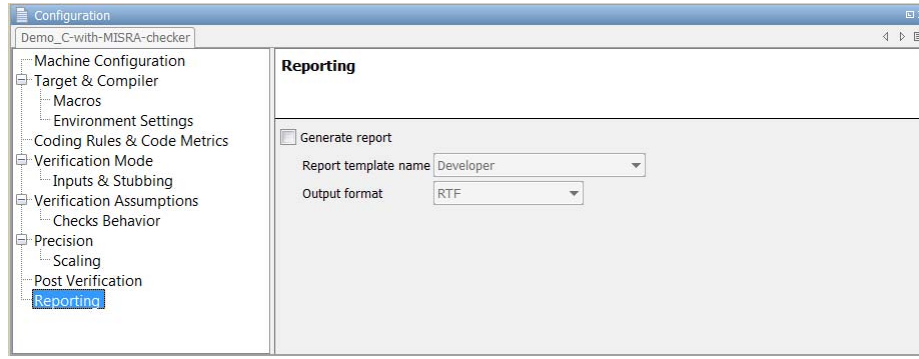
Default:

5 seconds

Example Shell Script Entry:

```
polyspace-c -automatic-orange-tester
-automatic-orange-tester-timeout 10 ...
```

Reporting



In this section...

Generate report

Specify whether to create verification report using report generation options

Settings

Default: Off

☑ On
Create report.

☐ Off
No report created.

Report template name

Specify template for generating verification report

Settings

Default:

```
C:\Polyspace\Polyspace_Common\ReportGenerator\templates\Developer.rpt
```

Report templates provided with the software include:

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

Tip

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

Command-Line Information

Parameter: `report-template`
Type: string
Value: any valid script file name
Example: `polyspace-c -report-template` *`filepath`*`\my_template`

Output format

Specify output format of report

Settings

Default: `RTF`

`RTF`
: Generate an .rtf format report.

`HTML`
: Generate an .html format report.

`PDF`
: Generate a .pdf format report.

`Word`
: Generate a .doc format report.

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

`XML`
: Generate and .xml format report.

Note WORD format is not available on UNIX platforms, RTF format is used instead.

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

Command-Line Information

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

Shell script example:

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

Batch Options

In this section...

-server

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

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

When a `server` option is associated to the batch launching command, the name or IP address and a port number need to be specified. If the port number does not exist, the `12427` value will be used by default.

Note also that `polyspace-remote-` accepts all other options.

Option Example Shell Script Entry:

```
polyspace-remote-c  server 192.168.1.124:12400

polyspace-remote-c

polyspace-remote-c  server Bergeron
```

-sources-list-file

This option is only available in batch mode. The syntax of *file_name* is the following:

- One file per line.
- Each file name includes its absolute or relative path.

The source files are compiled in the order in which they are specified.

Note If you do not specify any files, the software verifies all files in the source directory in alphabetical order.

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

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

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

-v | -version

Display the Polyspace version number.

Example Shell Script Entry:

```
polyspace-c  v
```

It will show a result similar to:

```
Polyspace r2007a+

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

-h[elp]

Display in the shell window a simple help in a textual format giving information on all options.

Example Shell Script Entry:

```
polyspace-c  h
```

-prog

Specify a name for the project.

Note The Session identifier option no longer appears in the General section of the Analysis options GUI. You specify the Project name, Version, and Author parameters in the Polyspace Project – Properties dialog box. For more information, see “Create Verification Project”.

Settings

Default: `New_Project`

- The Session identifier cannot contain spaces.
- Use only characters that are valid for UNIX file names.

Command-Line Information

Parameter: `-prog`
Value: any valid value
Example: `polyspace-c -prog myApp ...`

`-date`

Specify a date stamp for the verification.

Note The Date option no longer appears in the General section of the Analysis options GUI. The date is set automatically when you launch a verification.

Settings

Default: Date the verification is launched

By default, the date stamp uses the `dd/mm/yyyy` format.

Tip

You can specify an alternative date format by selecting **Edit > Preferences > Miscellaneous** in the Launcher.

Command-Line Information

Parameter: `-date`
Value: any valid value
Example: `polyspace-c -date "02/01/2002"...`

-author

Specify the name of the person performing the verification.

Note The Author option no longer appears in the General section of the Analysis options GUI. You specify the Project name, Version, and Author parameters in the Polyspace Project – Properties dialog box. For more information, see "Create Verification Project" .

Settings

Default: `username` of the current user.

Note The default username is obtained with the *whoami* command.

Command-Line Information

Parameter: `-author`
Value: any valid value
Example: `polyspace-c -author "John Tester"`

-verif-version

Specify a version identifier for the verification.

Note The Project Version option no longer appears in the General section of the Analysis options GUI. You specify the Project name, Version, and Author parameters in the Polyspace Project – Properties dialog box. For more information, see "Create Verification Project".

Settings

Default: `1.0`

Tip

This option can be used to identify different verifications.

Command-Line Information

Parameter: `-verif-version`
Value: any valid value
Example: `polyspace-c -verif-version 1.3`

-results-dir

This option specifies the directory in which Polyspace software will write the results of the verification. Note that although relative directories may be specified, particular care should be taken with their use especially where the tool is to be launched remotely over a network, and/or where a project configuration file is to be copied using the "Save as" option.

Default:
Shell Script: The directory in which tool is launched.
From Graphical User Interface: C:\Polyspace_Results

Example Shell Script Entry:

```
polyspace-c -results-dir RESULTS ...
export RESULTS=results_`date +%d%B_%HH%M_%A`
polyspace-c -results-dir `pwd`/$RESULTS ...
```

-sources

Specifies a list of source files to be verified.

The list of source files must be double-quoted and separated by commas.

- `-sources "`*`file1`*`[ `*`file2`*`[ ...]]"` (Linux and Solaris™)
- `-sources "`*`file1`*`[,`*`file2`*`[, ...]]"` (Windows, Linux and Solaris)
- `-sources-list-file `*`file_name`* (not a graphical option)

Note UNIX standard wild cards are available to specify a number of files.

The source files are compiled in the order in which they are specified.

Note If you do not specify any files, the software verifies all files in the source directory in alphabetical order.

Note The specified files must have valid extensions:
*.(c|C|cc|cpp|CPP|cxx|CXX)

Defaults:

```
sources/*.(c|C|cc|cpp|CPP|cxx|CXX)
```

Example Shell Script Entry under linux or solaris (*files are separated with a white space)*:

```
polyspace-c -sources "my_directory/*.cpp" ...
polyspace-c -sources "my_directory/file1.cc other_dir/file2.cpp"
...
```

Example Shell Script Entry under windows (*files are separated with a comma):*

```
polyspace-c -sources "my_directory/file1.cpp,other_dir/file2.cc"
...
```

Using `-sources-list-file`, each file *name* need to be given with an absolute path. Moreover, the syntax of the file is the following:

- One file by line.
- Each file name is given with its absolute path.

Note This option is only available in batch mode

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

```
polyspace-c -sources-list-file "C:\Analysis\files.txt"
polyspace-c -sources-list-file "/home/poly/files.txt"
```

-I

This option is used to specify the name of a directory to be included when compiling C sources. Only one directory may be specified for each –I, but the option can be used multiple times.

Default:

- Polyspace software automatically adds the following standard include folders after any includes you specify:
  - *Polyspace_Install*`/Verifier/include/include-gnu`
  - *Polyspace_Install*`/Verifier/include/include-gnu/next`
- The `./sources` directory (if it exists), is implicitly added at the end of the "`-I`" list
- If you do not specify any include folders, the two standard include folders and the `./sources` directory (if it exists) are automatically included

Example Shell Script Entry-1:

```
polyspace-c -I /com1/inc -I /com1/sys/inc
```

is equivalent to

```
polyspace-c -I /com1/inc -I /com1/sys/inc -I ./sources
```

Example Shell Script Entry-2:

```
polyspace-c
```

is equivalent to

```
polyspace-c -I ./sources
```

-from

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

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

The option is usually used in a verification after one run with the `-to` option, although it can also be used to recover after power failure.

Possible values are as described in the `-to` *`verification-phase`* section, with the addition of the `scratch` option.

Note

- This option can be used only for client verifications. All server verifications start from `scratch`.
- Unless the `scratch` option is used, this option can be used only if the previous verification was launched using the option `-keep-all-files` .
- This option cannot be used if you modify the source code between verifications.

Default :

scratch

Example Shell Script Entry :

```
polyspace-c -from c-to-il ...
```

-import-comments

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

Default:

Disabled

Example Shell Script Entry:

```
polyspace-c -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. This action may affect processing speed if the results folder is mounted on a network drive. Use this option only when the temporary folder partition is not large enough and troubleshooting is required.

Default:

Disabled

Example Shell Script Entry:

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

-less-range-information

Limits the amount of range information displayed in verification results.

When you select this option, the software provides range information on assignments, but not on reads and operators.

In addition, selecting this option enables the `no-pointer-information` option. See "`-no-pointer-information`" on page 1-94.

Computing range information for reads and operators may take a long time, and can reduce the precision of the verification (causing more orange checks). Selecting this option can reduce verification time significantly, and improve the precision of the verification. Consider the following example:

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

If you do not select this option (the default), the software displays range information when you place the cursor over `x`, `y`, `z`, or `+`. However, if you select this option, the software displays range information only when you place the cursor over `x`.

Default:

Disabled.

Example Shell Script Entry :

```
polyspace-c -less-range-information
```

-no-pointer-information

Stops the display of pointer information in verification results.

When you select this option, the software does not provide pointer information through tooltips. As computing pointer information may take a long time, selecting this option can significantly reduce verification time.

Consider the following example:

```
x = *p;
```

If you do not select this option (the default), the software displays pointer information when you place the cursor on `p` or `*`. If you select this option, the software does not display pointer information.

Default:

Disabled.

Example Shell Script Entry :

```
polyspace-c -no-pointer-information
```

-keep-all-files

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

"-keep-all-files" on page 1-95

Settings

Default: Off

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

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

Tips

- When you select this option you can restart Polyspace verification from the end of any complete pass (provided the source code remains entirely unchanged). If this option is not used, you must restart the verification from scratch.
- This option is applicable only to client verifications. Intermediate results are always removed before results are downloaded from the Polyspace server.
- To cleanup intermediate files at a later time, you can select **Tools > Clean Results** in the Launcher. This option deletes the preliminary result files from the results folder.

Command-Line Information

Parameter: `-keep-all-files`
Example: `polyspace-c -keep-all-files`

-known-NTC

After a few verifications, you may discover that a few functions "never terminate". Some functions such as tasks and threads contain infinite loops by design, while functions that exit the program such as *kill_task* , *exit* or *Terminate_Thread* are often stubbed by means of an infinite loop. If these functions are used very often or if the results are for presentation to a third party, it may be desirable to filter all NTC of that kind in the Viewer.

This option is provided to allow that filtering to be applied. All NTC specified at launch will appear in the viewer in the known-NTC category, and filtering will be possible.

Default :

All checks for deliberate Non Terminating Calls appear as red errors, listed in the same category as any problem NTC checks.

Example Shell Script Entry :

```
polyspace-c -known-NTC "kill_task,exit"

polyspace-c -known-NTC "Exit,Terminate_Thread"
```

-asm-begin -asm-end

```
-asm-begin "mark1[mark2[...]]  "
```

and

```
-asm-end "mark1[mark2[...]]"
```

These options are used to allow compiler specific asm functions to be excluded from the verification, with the offending code block delimited by two #pragma directives.

Consider the following example.

```
#pragma asm_begin_1
int foo_1(void) { /* asm code to be ignored by Polyspace */ }
#pragma asm_end_1
#pragma asm_begin_2
void foo_2(void) { /* asm code to be ignored by Polyspace */ }
#pragma asm_end_2
```

Where "asm_begin_1" and "asm_begin_2" marks the beginning of asm sections which will be discarded and “asm_end_1”, respectively "asm_end_2" mark the end of those sections.

Note The asm-begin and asm-end options must be used together.

Example Shell Script Entry:

```
polyspace-c -discard-asm -asm-begin "asm_begin_1,asm_begin_2"
-asm-end "asm_end_1,asm_end_2" ...
```

-strict

This option selects the Strict mode of Polyspace verification. It is equivalent to using the following options:

- -wall
- -no-automatic-stubbing

This option is not compatible with the options -asm-begin and -asm-end.

-permissive

This option selects the Polyspace permissive mode, which is equivalent to using all of the following options:

- -ignore-constant-overflows

- `-allow-negative operand-in-shift`

-Wall

Specifies that the software display all possible warnings during the C compliance phase.

Using this option can be an effective way to detect problems in the code without using the MISRA checker.

For example, when you specify this option, the software adds the following warning to the log file when trying to write into a `const` variable:

```
warning:  assignment of read-only member <var>
```

Default:

By default, only warnings about compliance across different files are printed.

Example Shell Script Entry:

```
polyspace-c -Wall ...
```

-report-output-name

"-report-output-name" on page 1-98

Specify name of verification report file

Settings

Default: *Prog_TemplateName.Format* where:

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

Command-Line Information

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

Shell script example:

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

Deprecated Options

In this section...

-continue-with-red-error (Deprecated)

Note This option is deprecated in R2009a and later releases, and no longer exists in the user interface. Verification now continues to the next integration pass even if a red errors is encountered.

This option allows Polyspace verification to continue even if one of these red errors is encountered. In most cases, this will mean that the dynamic behavior of the code beyond the point where red errors are identified will be undefined, unless the red code is actually inaccessible.

-continue-with-existing-host (Deprecated)

Note This option is deprecated in R2010a and later releases, and no longer exists in the user interface. Polyspace verification now continues regardless of the system configuration. The software still checks the hardware configuration, and issues a warning if it does not satisfy requirements.

When this option is set, the verification will continue even if the system is under specified or its configuration is not as preferred by Polyspace software. Verified system parameters include the amount of RAM, the amount of swap space, and the ratio of RAM to swap.

-allow-unsupported-linux (Deprecated)

Note This option is deprecated in R2010a and later releases, and no longer exists in the user interface. Polyspace verification now continues regardless of the Linux distribution. If the Linux distribution is not officially supported, the software displays a warning in the log file.

This option specifies that Polyspace verification will be launched on an unsupported OS Linux distribution.

-quick (Deprecated)

Note This option is deprecated in R2009a and later releases.

`quick` mode is obsolete and has been replaced with verification PASS0. PASS0 takes somewhat longer to run, but the results are more complete. The limitations of `quick` mode, (no NTL or NTC checks, no float checks, no variable dictionary) no longer apply. Unlike `quick` mode, PASS0 also provides full navigation in the Viewer.

This option is used to select a very fast mode for Polyspace .

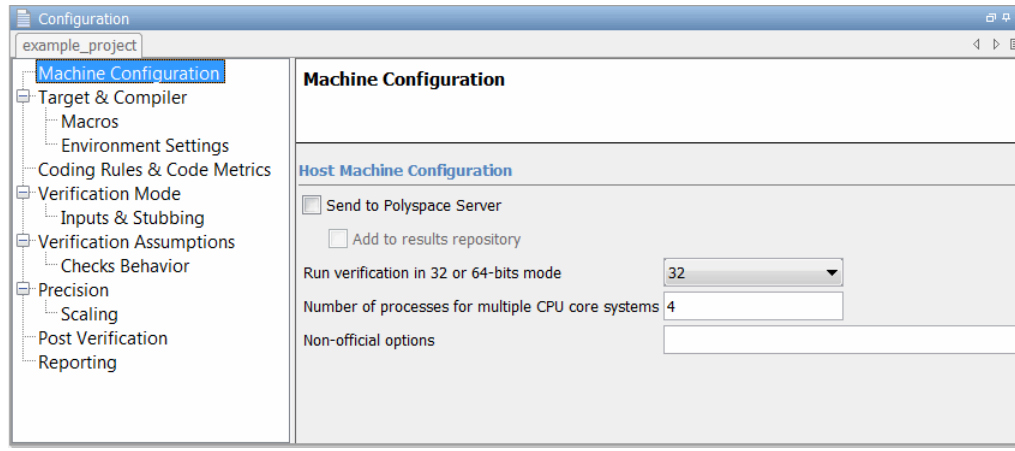
This option allows results to be generated very quickly. These are suitable for initial verification of red and gray errors only, as orange checks are too plentiful to be relevant using this option.

2

Option Descriptions for C++ Code

Overview

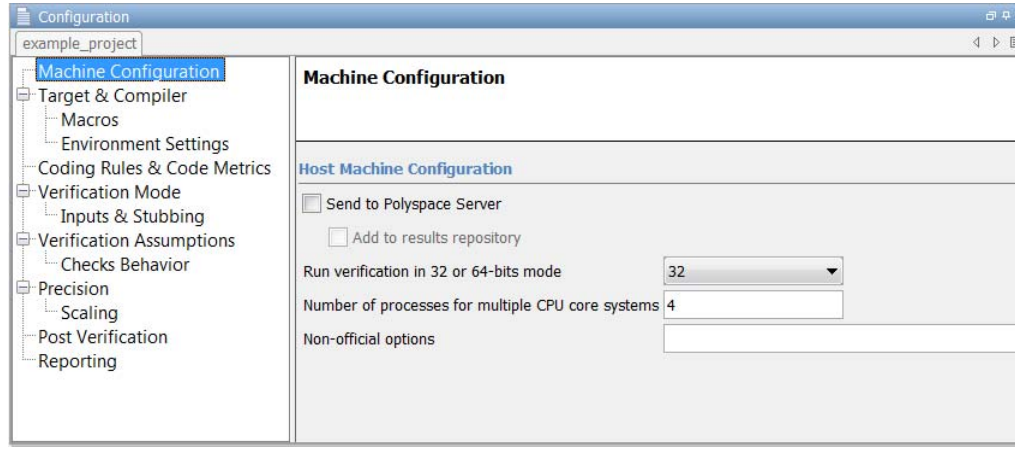
In the Project Manager perspective, on the **Configuration** pane, you can specify the analysis options (identification information and parameters) that software uses for code verification.



Polyspace software groups the analysis options into various categories. To display the parameters for a specific category, from the **Configuration** tree, select the category.

Note From the command line, you can use the `polyspace-cpp` command to specify parameters. The description for each parameter includes command line information.

Machine Configuration



In this section...

Machine Configuration Overview

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

Send to Polyspace Server

Specify whether verification runs on the server or client system.

Settings

Default: On

 On

Run verification on the Polyspace server. The server to use is specified in the Polyspace preferences.

 Off

Run verification on the client system.

Tips

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

Command-Line Information

Parameter: `-server`
Value: *`name or IP address:port number`*
Example: `polyspace-remote-desktop-cpp  server 192.168.1.124:12400`

Add to results repository

Specify whether verification results are added to the Polyspace Metrics results database, allowing Web-based reporting of results and code metrics.

Settings

Default: Off

 On

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

 Off

Verification results are not added to the database.

Dependency

- This option is available only for server verifications.

Command-Line Information

Parameter: `-add-to-results-repository`
Example: `polyspace-cpp -server -add-to-results-repository`

Run verification in 32 or 64-bits mode

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

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

Available options are:

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

Default:

```
auto
```

Example Shell Script Entry:

```
polyspace-cpp -machine-architecture auto
```

Number of processes for multiple CPU core systems

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

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

Default:

```
4
```

Example Shell Script Entry:

```
polyspace-cpp -max-processes 1
```

Non-official options

This option specifies an expert option to be added to the analyzer. Each word of the option (even the parameters) must be preceded by *-extra-flags*.

These flags will be given to you by MathWorks if required.

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-cpp -extra-flags -param1 -extra-flags -param2
```

-cpp-extra-flags flag

It specifies an expert option to be added to a C++ verification. Each word of the option (even the parameters) must be preceded by *-cpp-extra-flags*.

These flags will be given to you by MathWorks if required.

Default:

no extra flags.

Example Shell Script Entry:

```
polyspace-cpp -cpp-extra-flags -stubbed-new-may-return-null
```

-il-extra-flags flag

It specifies an expert option to be added to a C++ verification. Each word of the option (even the parameters) must be preceded by *-il-extra-flags*.

These flags will be given to you by MathWorks if required.

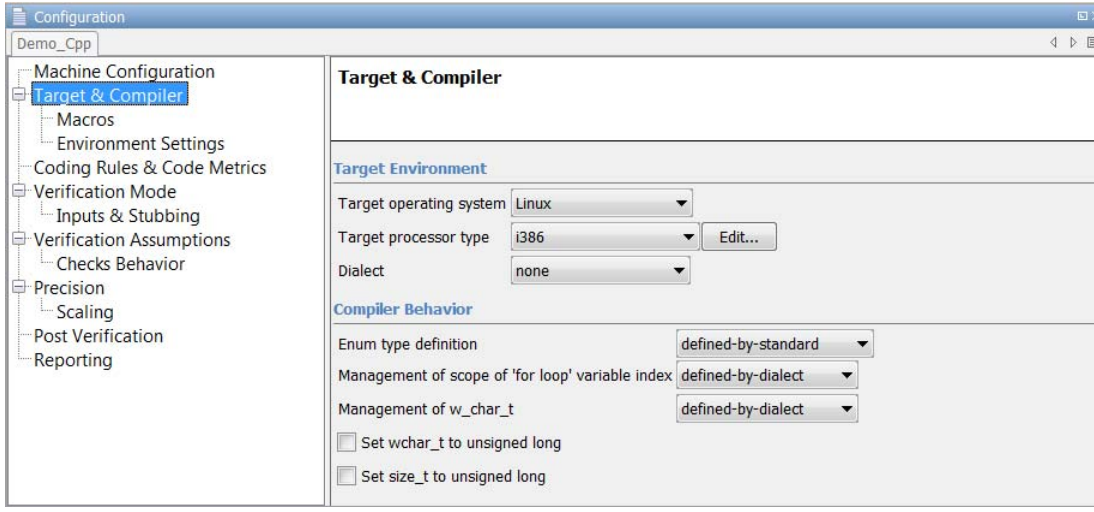
Default:

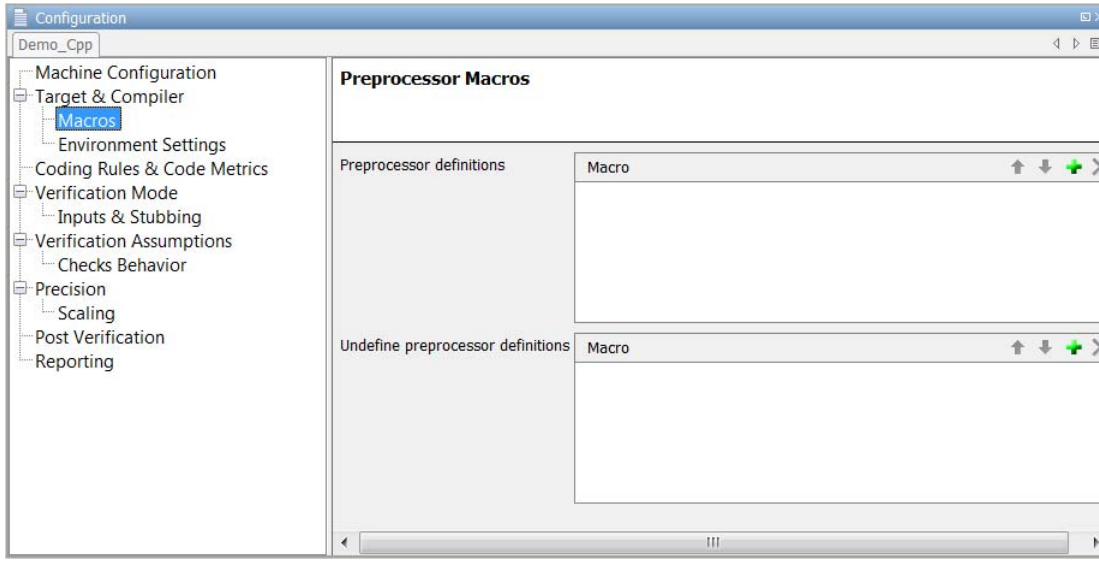
no extra flags.

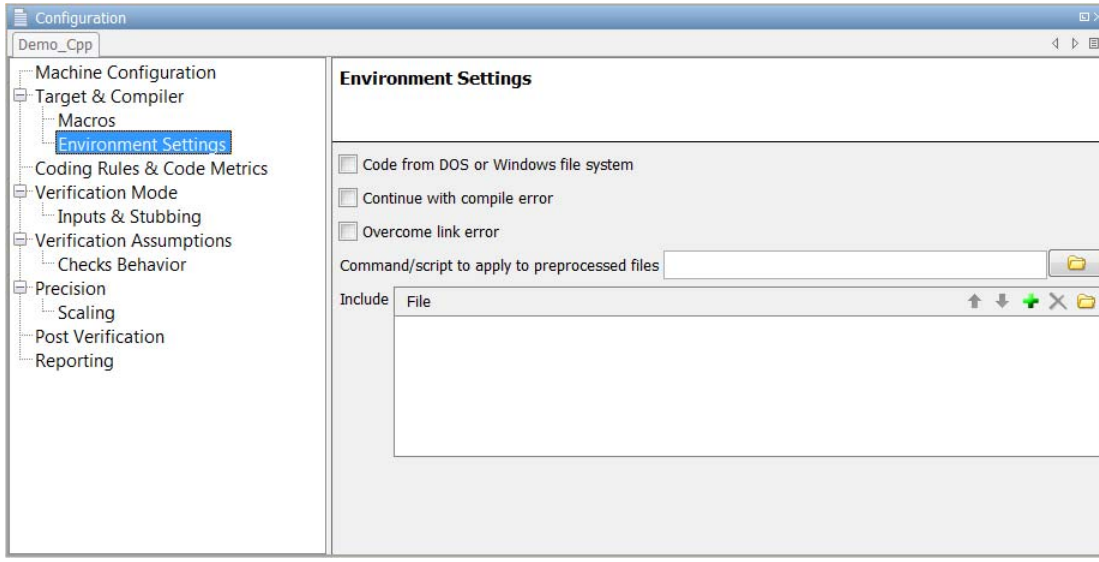
Example Shell Script Entry:

```
polyspace-cpp -il-extra-flags flag
```

Target & Compiler







In this section...

In this section...

Target operating system

This option specifies the operating system target for Polyspace stubs.

Possible values are

- `Linux` (default)
- `Solaris`
- `VxWorks`
- `Visual`
- `no-predefined-OS`

This information allows the corresponding system definitions to be used during preprocessing — to analyze the included files properly.

You can use the target *`-OS-target`* `no-predefined-OS` in conjunction with -include and/or -D to give all of the system preprocessor flags to be used at execution time. Details of these may be found by executing the compiler for the project in verbose mode.

Default:

`Linux`

Note Only the Linux include files are provided with Polyspace software. If you use Linux header files, you must set this option to `Linux`. Otherwise, you see compilation errors.

Projects developed for use with other operating systems may be analyzed using include files for the corresponding operating systems. For instance, in order to analyze a VxWorks project, use the option -I *`path_to_the_VxWorks_include_folder`*.

Example shell script entry:

```
polyspace-cpp -OS-target linux
polyspace-cpp  -OS-target no-predefined-OS -D  GCC_MAJOR=2     /
      -include /complete_path/inc/gn.h  ...
```

Target processor type

This option specifies the target processor type, and by doing so informs Polyspace of the size of fundamental data types and of the endianess of the target machine.

Possible values are:

- `i386` (default)
- `sparc`
- `m68k`
- `powerpc`
- `c-167`
- `x86_64`
- `mcpu...(Advanced)`

`mcpu` is a reconfigurable Micro Controller/Processor Unit target. You can use this type to configure one or more generic targets.

You can analyze code intended for an unlisted processor type using one of the listed processor types, if they share common data properties. Refer to “Set Up a Target” for more details.

For information on specifying a generic target, or modifying the mcpu target, see “Generic target options” on page 2-12.

Note The generic target option is incompatible with any visual dialect.

Default:

`i386`

Example shell script entry:

```
polyspace-cpp -target m68k ...
```

Generic target options

The *Generic target options* dialog box opens when you select an *mcpu* target, or a *generic* target.

This dialog box allows you to specify a generic *"Micro Controller/Processor Unit"* or *mcpu* target name. Initially, use the dialog box to specify the name of a new *mcpu* target – say, “MyTarget”.

Note The generic target option is incompatible with any visual dialect.

That new target is added to the -target options list. The new target’s default characteristics are as follows, using the *type [size, alignment]* format.

- *char [8, 8], char [16,16]*
- *short [16, 16]*
- *int [16, 16]*

- *long [32, 32], long long [32, 32]*
- *float [32, 32], double [32, 32], long double [32, 32]*
- *pointer [16, 16]*
- *char is signed*
- *little-endian*

When using the command line, *MyTarget* is specified with all the options for modification:

```
polyspace-cpp -target MyTarget
```

For example, a specific target uses 8 bit alignment (see also -align), for which the command line would read:

```
polyspace-cpp -target mcpu -align 8
```

-little-endian

This option is only available when a *-mcpu* generic target has been chosen.

The endianness defines the byte order within a word (and the word order within a long integer). Little-endian architectures are Less Significant byte First (LSF), for example: i386.

For a little endian target, the less significant byte of a short integer (for example 0x00FF) is stored at the first byte (0xFF) and the most significant byte (0x00) at the second byte.

Example shell script entry:

```
polyspace-cpp -target mcpu -little-endian
```

-big-endian

This option is only available when a *-mcpu* generic target has been chosen.

The endianness defines the byte order within a word (and the word order within a long integer). Big-endian architectures are Most Significant byte First (MSF), for example: SPARC, m68k.

For a big endian target, the most significant byte of a short integer (for example 0x00FF) is stored at the first byte (0x00) and the less significant byte (0xFF) at the second byte.

Example shell script entry:

```
polyspace-cpp -target mcpu -big-endian
```

-default-sign-of-char [signed|unsigned]

This option is available for all targets. It allows a char to be defined as "signed", "unsigned", or left to assume the mcpu target's default behavior

Default mode:

The sign of char is left to assume the target's default behavior. By default all targets are considered as signed except for powerpc targets.

Signed:

Disregards the target's default char definition, and specifies that a "signed char" should be used.

Unsigned:

Disregards the target's default char definition, and specifies that a "unsigned char" should be used.

Example Shell Script Entry

```
polyspace-cpp -default-sign-of-char unsigned -target mcpu ...
```

-char-is-16bits

This option is available only when you select a mcpu generic target.

The default configuration of a generic target defines a char as 8 bits. This option changes it to 16 bits, regardless of sign.

the minimum alignment of objects is also set to 16 bits and so, incompatible with the options `-short-is-8 bits` and `-align 8`.

Setting the char type to 16 bits has consequences on the following:

- computation of size of for objects
- detection of underflow and overflow on chars

Without the option char for *mcpu* are 8 bits

Example shell script entry:

```
polyspace-cpp -target mcpu -char-is-16bits
```

-short-is-8bits

This option is only available when a generic target has been chosen.

The default configuration of a generic target defines a short as 16 bits. This option changes it to 8 bits, irrespective of sign.

It sets a short type as 8-bit without specific alignment. That has consequences for the following:

- computation of size of objects referencing short type
- detection of short underflow/overflow

Example shell script entry

```
polyspace-cpp -target mcpu -short-is-8bits
```

-int-is-32bits

This option is available with a generic target has been chosen.

The default configuration of a generic target defines an int as 16 bits. This option changes it to 32 bits, irrespective of sign. Its alignment, when an int is used as struct member or array component, is also set to 32 bits. See also -align option.

Example shell script entry

```
polyspace-cpp -target mcpu -int-is-32bits
```

-long-long-is-64bits

This option is only available when a generic target has been chosen.

The default configuration of a generic target defines a long long as 32 bits. This option changes it to 64 bits, irrespective of sign. When a long long is used as struct member or array component, its alignment is also set to 64 bits. See also -align option.

Example shell script entry

```
polyspace-cpp -target mcpu -long-long-is-64bits
```

-double-is-64bits

This option is available when either a generic target has been chosen.

The default configuration of a generic target defines a double as 32 bits. This option, changes both double and *long double* to 64 bits. When a double or long double is used as a struct member or array component, its alignment is set to 4 bytes.

See also -align option.

Defining the double type as a 64 bit double precision float impacts the following:

- Computation of sizeofobjects referencing double type
- Detection of floating point underflow/overflow

Example

```
int main(void)
{
 struct S {char x; double f;};
 double x;
 unsigned s1, s2;
 s1 = sizeof (double);
 s2 = sizeof(struct S);
 x = 3.402823466E+38; /* IEEE 32 bits float point maximum value */
 x = x * 2;
 return 0;
}
```

Using the default configuration of sharc21x62, C Polyspace assumes that a value of 1 is assigned to s1, 2 is assigned to s2, and there is a consequential float overflow in the multiplication x * 2. Using the –double-is-64bits option, a value of 2 is assigned to s1, and no overflow occurs in the multiplication (because the result is in the range of the 64-bit floating point type)

Example shell script entry

```
polyspace-cpp -target mcpu -double-is-64bits
```

-pointer-is-32bits

This option is only available when a *generic* target has been chosen.

The default configuration of a generic target defines a pointer as 16 bits. This option changes it to 32 bits. When a pointer is used as struct member or array component, its alignment is also set also to 32 bits (see -align option).

Example shell script entry

```
polyspace-cpp -target mcpu -pointer-is-32bits
```

-align [8|16|32]

This option is available with an *mcpu* generic target and some other specific targets. It is used to set the largest alignment of all data objects to 4/2/1 byte(s), meaning a 32, 16 or 8 bit boundary respectively.

The default alignment of a generic target is 32 bits. This means that when objects with a size of more than 4 bytes are used as struct members or array components, they are aligned at 4 byte boundaries.

Example shell script entry with a 32 bits default alignment

```
polyspace-cpp -target mcpu
```

-align 16. If the -align 16 option is used, when objects with a size of more than 2 bytes are used as struct members or array components, they are aligned at 2 bytes boundaries.

Example shell script entry with a 16 bits specific alignment:

```
polyspace-cpp -target mcpu -align 16
```

-align 8. If the -align 8 option is used, when objects with a size of more than 1 byte are used as struct members or array components, are aligned at 1 byte boundaries. Consequently the storage assigned to the arrays and structures is strictly determined by the size of the individual data objects without member and end padding.

Example shell script entry with a 8 bits specific alignment:

```
polyspace-cpp -target mcpu -align 8
```

Dialect

Specifies the dialect in which the code is written. Possible values are:

- `gnu` (default if `-OS-target` is set to `Linux`)
- `cfront2`
- `cfront3`
- `iso`
- `visual`
- `visual6`
- `visual7.0`

- `visual7.1`
- `visual8`
- `visual9.0`

`visual6` activates dialect associated with code used for Microsoft Visual 6.0 compiler and visual activates dialect associated with Microsoft Visual 7.1 and subsequent.

If the dialect is visual (`visual`, `visual6`, `visual7.0`, `visual7.1` `visual8`, and `visual9.0`) the -OS-target option must be set to `Visual`.

If the dialect is visual, the option `-dos`, `-OS-target Visual` is set by default.

`visual8` dialect activates support for Visual 2005 .NET specific compiler. All Visual 2005 .NET given include files can compile both with the `-no-stl-stubs` option and without it (recommended).

Note If you select the `-jsf-coding-rules` option and a dialect other than `iso` or `default`, some JSF++ coding rules may not be completely checked. For example, AV Rule 8: "All code shall conform to ISO/IEC 14882:2002(E) standard C++."

Default:

`gnu` if `-OS-target` is set to `Linux`

`visual7.1` if `-OS-target` is set to `visual`

`none` otherwise

Example Shell Script Entry:

```
polyspace-cpp -dialect visual8 ...
```

Pack alignment value

Visual C++ /Zp option specifies the default packing alignment for a project. Option -pack-alignment-value transfers the default alignment value to Polyspace verification.

The argument value must be: 1, 2, 4, 8, or 16. Verification will halt and display an error message with a bad value or if this option is used in non visual mode (-OS-target visual or -dialect visual* (6, 7.0 or 7.1)).

Default:

8

Example Shell Script Entry:

```
polyspace-cpp  dialect visual  pack-alignment-value 4 ...
```

Import folder

One directory to be included by #*import*directive. This option must be used with -OS-target visual or -dialect visual* (6, 7.0, 7.1 and 8). It gives the location of *.tlh files generated by a Visual Studio compiler when encounter #import directive on *.tlb files.

Example Shell Script Entry:

```
polyspace-cpp -dialect visual8 -import-dir /com1/inc ...
```

Ignore pragma pack directives

C++ #pragma directives specify packing alignment for structure, union, and class members. The -ignore-pragma-pack option allows these directives to be ignored in order to prevent link errors.

Polyspace verification stops execution and displays an error message if this option is used in non visual mode or without dialect gnu (without -OS-target visual or –dialect visual*). See also “Link messages” .

Example Shell Script Entry:

```
polyspace-cpp  dialect visual  ignore-pragma-pack ...
```

Support managed extensions

Visual C++ /FX option allows the partial translation of sources making use of managed extensions to Visual C++ sources without managed extensions. Theses extensions are currently not taken into account by Polyspace verification and can be considered as a limitation to analyze this kind of code.

Using /FX, the translated files are generated in place of the original ones in the project, but the names are changed from foo.ext to foo.mrg.ext.

Option – support-FX-option-results allows the verification of a project containing translated sources obtained by compilation of a Visual project using the /FX Visual option. Managed files need to be located in the same folder as the original ones and Polyspace software will verify managed files instead of the original ones without intrusion, and will permit you to remove part of the limitations due to specific extensions.

Polyspace verification stops execution and displays an error message if this option is used in non visual mode (-OS-target visual or -dialect visual* (6, 7.0 or 7.1)).

Example Shell Script Entry:

```
polyspace-cpp  dialect visual - support-FX-option-results
```

Enum type definition

Allows the verification to use different base types to represent an enumerated type, depending on the enumerator values and the selected definition.

When using this option, each enum type is represented by the smallest integral type that can hold all its enumeration values.

Possible values are:

- **`defined-by-standard`** – Uses the first type that can hold all of the enumerator values from the following list: `signed int`, `unsigned int`, `signed long`, `unsigned long`, `signed long long`, `unsigned long long`

- **`auto-signed-first`** - Uses the first type that can hold all of the enumerator values from the following list: `signed char`, `unsigned char`, `signed short`, `unsigned short`, `signed int`, `unsigned int`, `signed long`, `unsigned long`, `signed long long`, `unsigned long long`.
- **`auto-unsigned-first`** - Uses the first type that can hold all of the enumerator values from the following lists:
  - If enumerator values are all positive: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`.
  - If one or more enumerator values are negative: `signed char`, `signed short`, `signed int`, `signed long`, `signed long long`.

Management of scope of 'for loop' variable index

This option changes the scope of the index variable declared within a `for` loop. For example:

```
for (int index=0; ...){};
index++; // At this point, index variable is usable (out) or not (in)
```

You can specify one of the following values:

- `defined-by-dialect` — Default behavior specified by selected dialect.
- `out` — Default behavior for the `-dialect` options `cfront2`, `crfront3`, `visual6`, `visual7` and `visual 7.1`.
- `in` — Default behavior for all other dialects, including `visual8`. The C++ standard specifies that the index is treated as `in`.

This option allows the default behavior implied by the Polyspace `-dialect` option to be overridden.

This option is equivalent to the Visual C++® options `/Zc:forScope` and `Zc:forScope-`.

Default:

`defined-by-dialect`

Example Shell Script Entry:

```
polyspace-cpp -for-loop-index-scope in
```

Management of w_char_t

With this option, you can force `wchar_t` to be treated as a:

- Keyword as given by the C++ standard
- `typedef` statement specified by Microsoft Visual C++ `6.0/7.x` dialects.

You can specify one of the following values':

- `defined-by-dialect` — Default behavior specified by selected dialect.
- `typedef` — Default behavior for `-dialect` options `visual6`, `visual7.0` and `visual7.1`.
- `keyword` — Default behavior for all others dialects including `visual8`.

This option allows the default behavior implied by the Polyspace `-dialect` option to be overridden.

This option is equivalent to the Visual C++ options `/Zc:wchar` and `/Zc:wchar-`.

Default:

```
defined-by-dialect
```

Example Shell Script Entry:

```
polyspace-cpp -wchar-t-is typedef
```

Set wchar_t to unsigned long

This option forces the "underlying type" as defined in the C++ standard to be unsigned long.

For example, sizeof(L'W') will have the value of sizeof(unsigned long) and the wchar_t field will be aligned in the same way as the unsigned long field. Note that wchar_t will remain a different type from unsigned long unless

"-wchar-t-is typedef" is set or implied by the current dialect. The default underlying type of wchar_t is unsigned short.

Example Shell Script Entry:

```
polyspace-cpp -wchar-t-is-unsigned-long ...
```

Set size_t to unsigned long

Indicates the expected typedef of size_t to the software; forces the size_t type to be unsigned long. The default type of size_t is unsigned int.

Example Shell Script Entry: `polyspace-cpp -size-t-is-unsigned-long ...`

Preprocessor definitions

This option is used to define macro compiler flags to be used during compilation phase.

Only one flag can be used with each –D as for compilers, but the option can be used several times as shown in the example below.

Default:

Some defines are applied by default, depending on your -OS-target option.

Example Shell Script Entry:

```
polyspace-cpp -D HAVE_MYLIB -D USE_COM1 ...
```

Undefine preprocessor definitions

This option is used to undefine a macro compiler flags

As for compilers, only one flag can be used with each –U, but the option can be used several times as shown in the example below.

Default:

Some undefines may be set by default, depending on your -OS-target option.

Example Shell Script Entry:

```
polyspace-cpp -U HAVE_MYLIB -U USE_COM1 ...
```

Code from DOS or Windows file system

Use this option when the contents of the **include** or **source** folder comes from a DOS or Windows file system. It deals with upper/lower case sensitivity and control character issues.

The affected files are:

- Header files in all include folders specified through the `-I` option.
- All source files selected for the verification through the `-sources` option.

For example, with this option,

```
#include "..\mY_TEst.h"^M

#include "..\mY_other_FILE.H"^M
```

resolves to:

```
#include "../my_test.h"

#include "../my_other_file.h"
```

Default:

Enabled

Example Shell Script Entry:

```
          polyspace-cpp -I /usr/include -dos -I
./my_copied_include_dir -D test=1
```

Continue with compile error

Specifies that verification continues even if some source files do not compile. Functions that are used but not specified are stubbed automatically.

If a source file contains global variables, you may also need to select the option `-allow-undef-variables` to enable verification.

Example Shell Script Entry :

```
polyspace-cpp -continue-with-compile-error ...
```

Overcome link error

Some functions may be declared inside an extern "C" { } bloc in some files and not in others. Then, their linkage is not the same and it causes a link error according to the ANSI standard.

Applying this option will cause Polyspace to ignore this error.

This permissive option may not solve all the extern C linkage errors.

Example Shell Script Entry:

```
polyspace-cpp -no-extern-C ...
```

Command/script to apply to preprocessed files

When this option is used, the specified script file or command is run just after the pre-processing phase on each source file. The script executes for each preprocessed c file. The command should be designed to process the standard output from preprocessing and produce its results in accordance with that standard output.

Note The Compilation Assistant is automatically disabled when you specify this option.

Default:

No command.

Example Shell Script Entry – file name:

To replace the keyword “Volatile” with “Import”, you can type the following command on a Linux workstation:

```
polyspace-cpp -post-preprocessing-command `pwd`/replace_keywords
```

where `replace_keywords` is the following script :

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

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

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

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

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

```
%POLYSPACE_C%\Verifier\bin\polyspace-cpp.exe
-post-preprocessing-command
```

```
%POLYSPACE_C%\Verifier\tools\perl\win32\bin\perl.exe
<absolute_path>\replace_keywords
```

Include

This option is used to specify files to be included by each C++ file involved in the verification.

Default:

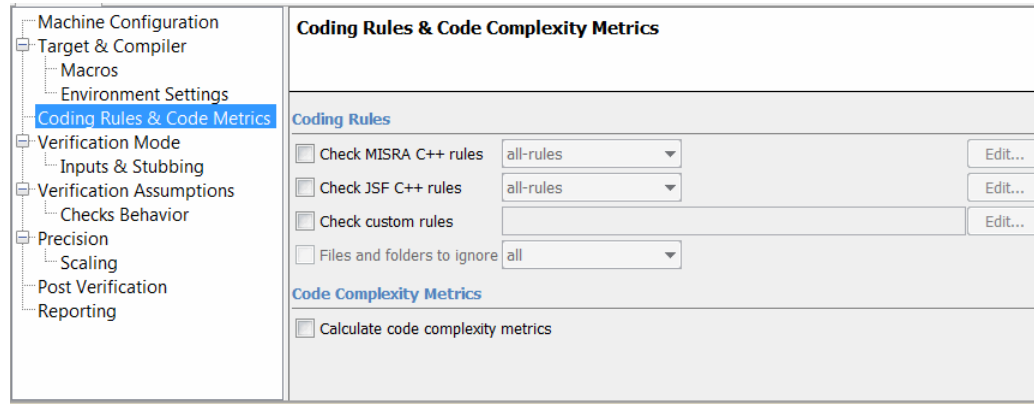
No file is universally included by default, but directives such as "#include <*include_file.h*>" are acted upon.

Example Shell Script Entry:

```
 polyspace-cpp -include `pwd`/sources/a_file.h -include
/inc/inc_file.h ...

 polyspace-cpp -include /the_complete_path/my_defines.h ...
```

Coding Rules & Code Complexity Metrics



In this section...

Check MISRA C++ rules

Specifies that Polyspace software checks for compliance with the MISRA C++ coding standards (MISRA C++:2008).

The results are included in the log file of the verification.

For more information, see “Set Up Coding Rules Checking”.

MISRA C++ rules configuration

Specifies set of coding rules to check.

- `required-rules` — Check all *required* MISRA C++ coding rules. All violations are reported as warnings.
- `all-rules` — Check all *required* and *advisory* coding rules. All violations are reported as warnings.
- `SQO-subset1` — Check a subset of MISRA C++ rules that have a direct impact on the selectivity of verification. All violations are reported as warnings. For more information, see "SQO Subset 1 – Direct Impact on Selectivity".
- `SQO-subset2` — Check a second subset of MISRA C++ rules that have an indirect impact on the selectivity of verification, as well as the rules contained in `SQO-subset1`. All violations are reported as warnings. For more information, see "SQO Subset 2 – Indirect Impact on Selectivity".
- `custom` — Check a specified set of MISRA C++ coding rules. You must provide the name of a file containing a list of MISRA C++ rules to check.

Note If you specify `-misra-cpp`, the `-Wall` option is disabled.

Format of the file:

```
<rule number> off|error|warning
# is considered a comment.
```

Example:

```
# MISRA-C++ rules configuration file
# Generated by Polyspace

0-1-1 warning
0-1-2 warning
0-1-7 warning
0-1-8 off
0-1-9 off
0-1-10 warning
```

```
0-1-11 off
0-1-12 off
1-0-1 error
1-0-2 off # Not implemented
1-0-3 off # Not implemented
2-2-1 off # Not implemented
2-3-1 warning
2-5-1 warning
2-7-1 warning

# End of file
```

Default:

Disabled

Example shell script entry:

```
polyspace-cpp -misra-cpp all-rules

polyspace-cpp -misra-cpp misra.txt
```

Check JSF C++ rules

Specifies that Polyspace software checks for compliance with the Joint Strike Fighter Air Vehicle C++ coding standards (JSF++:2005).

The results are included in the log file of the verification.

For more information, see “Set Up Coding Rules Checking”.

JSF C++ rules configuration

Specifies which JSF++ coding rules to check.

- `shall-rules` — Check all **Shall** rules, which are mandatory rules that require verification.
- `shall-will-rules` — Check all **Shall** and **Will** rules. **Will** rules are mandatory rules that do not require verification.

- `all-rules` — Check all **Shall**, **Will**, and **Should** rules. **Should** rules are advisory rules.
- `custom` — Check a specified set of JSF C++ coding rules. When you select this option, you must provide a rules file that specifies the JSF C++ rules to check and whether to report an error or warning for violations of each rule. For more information, see "Create a MISRA or JSF C++ Coding Rules File".

Note If you specify `-jsf-coding-rules`, the `-Wall` option is disabled.

Note If your project uses a dialect other than ISO, some JSF++ coding rules may not be completely checked. For example, AV Rule 8: "All code shall conform to ISO/IEC 14882:2002(E) standard C++."

Format of the file:

```
<rule number> off|error|warning
# is considered a comment.
```

Example:

```
# JSF-CPP rules configuration file
1 off # disable AV Rule number 1
2 off # Not implemented
3 off # disable AV Rule 3
8 error # violation AV Rule 8 is error
9 warning # violation AV Rule 9 is only a warning
# End of file
```

Default:

Disabled

Example shell script entry:

```
polyspace-cpp -jsf-coding-rules all-rules
```

```
polyspace-cpp -jsf-coding-rules jsf.txt
```

Check custom rules

Check names or text patterns in source code with reference to custom rules in specified text file. Each rule defines a check of a specified pattern against a source code identifier. For more information, see “Create a Custom Coding Rules File”.

Default:

Disabled

Example Shell Script Entry

```
polyspace-cpp -custom-rules myrules.txt
```

Files and folders to ignore

This option prevents the coding rules checker from checking a given list of files or folders (all files and subfolders under selected folder). This option is useful if you use headers that do not conform with JSF++ or MISRA C++ standards.

The software displays a warning if:

- Any file or folder on the list does not exist.
- All source code is ignored

This option is enabled only when you specify `-jsf-coding-rules` or `-misra-cpp`.

Note You can exclude all include folders from coding rules checking by specifying `"all"`. All files and folders included using the `-I` option are automatically added to `-includes-to-ignore`, and therefore excluded from the coding rules checker.

Example shell script entry :

```
 polyspace-cpp -jsf-coding-rules jsf.txt  includes-to-ignore
"c:\usr\include"

 polyspace-cpp -jsf-coding-rules jsf.txt  includes-to-ignore all
```

Calculate code complexity metrics

"Calculate code complexity metrics" on page 2-34

Specify whether to calculate software quality metrics, such as cyclomatic number, during verification.

Note This option requires a Polyspace Client for C/C++ license.

Settings

Default: On

 On

Calculate software quality metrics, including project metrics, file metrics, and function metrics.

 Off

Do not calculate software quality metrics.

Tips

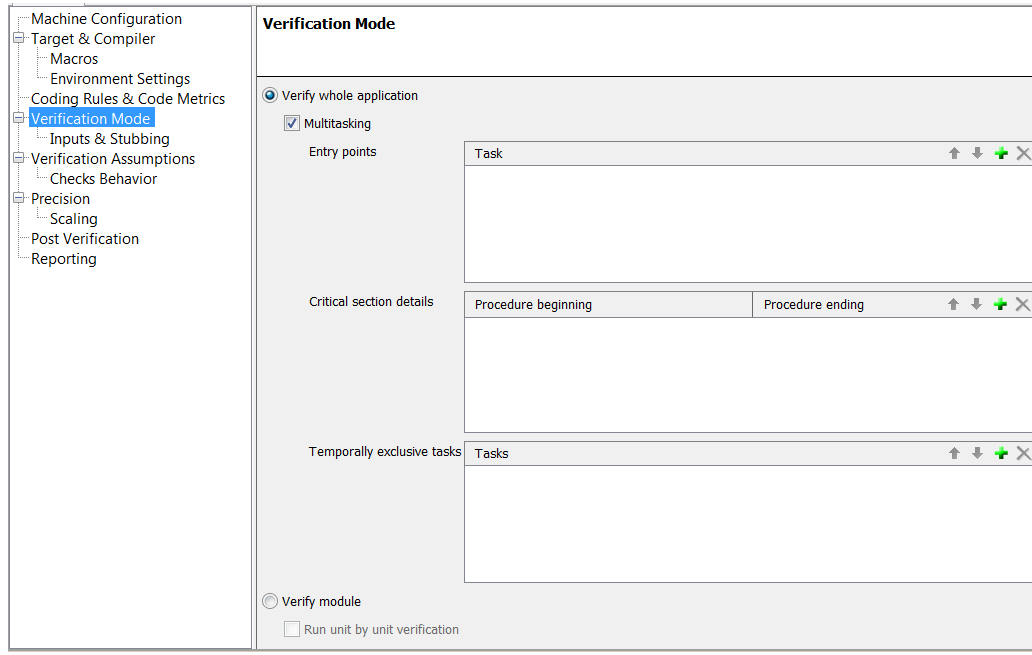
- You can view software quality metrics data in the Polyspace Metrics Web interface, or by running a Software Quality Objectives report from the Polyspace verification environment.
- Project metrics include number of recursions, number of include headers, and number of files.
- File metrics include comment density, and number of lines.

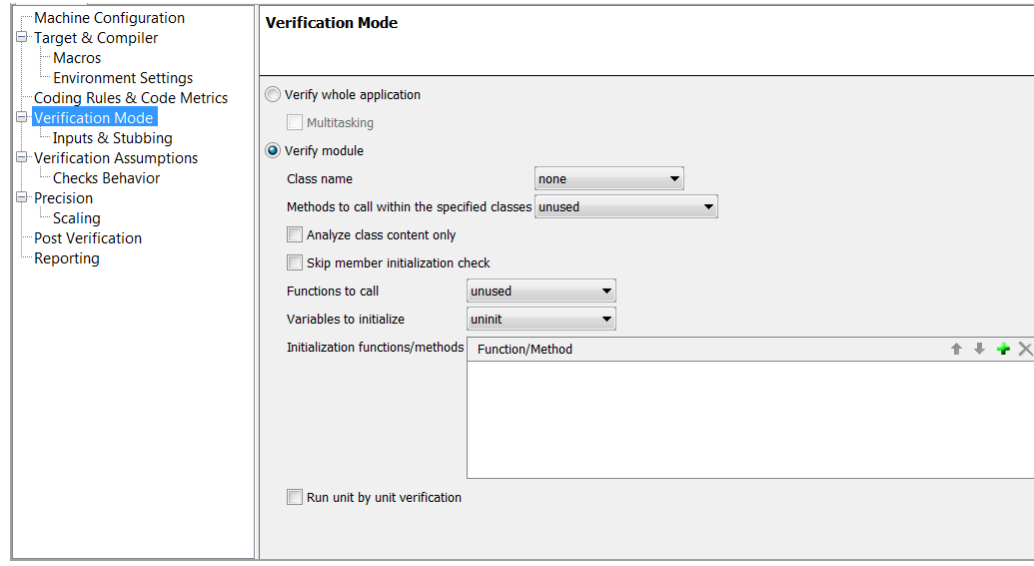
- Function metrics include cyclomatic number, number of static paths, number of calls, and Language scope.

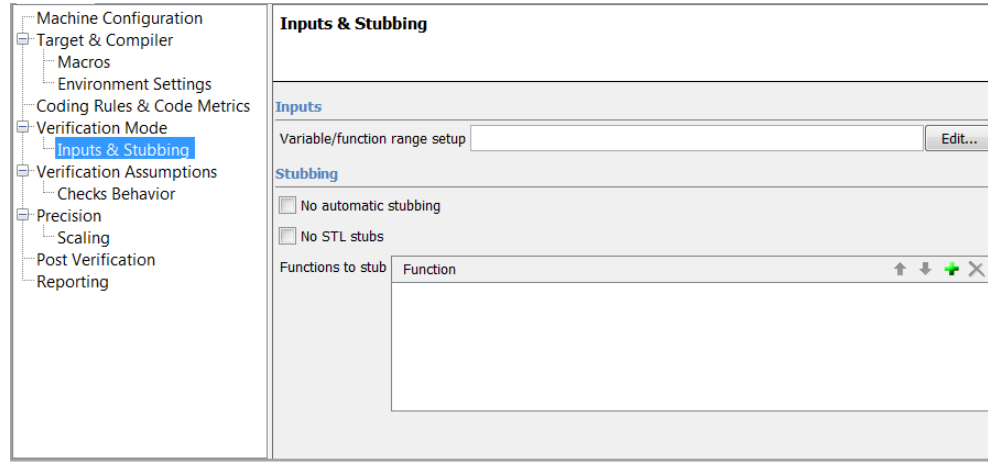
Command-Line Information

Parameter: `-code-metrics`
Example: `polyspace-cpp -code-metrics`

Verification Mode







In this section...

In this section...

Note Concurrency options are not compatible with Main Generator options.

Main entry point

The option specifies the name of the main subprogram when you select a visual –OS-target. This procedure will be analyzed after class elaboration, and before tasks in case of a multitask application or in case of the -entry-points usage.

Possible values are:

- `_tmain` (default)
- `main`

- `wmain`
- `_tWinMain`
- `wWinMain`
- `WinMain`
- `DllMain`.

However, if the main subprogram does not exist and the option -main-generator is not set, Polyspace verification stops with an error message.

Default:

```
_tmain
```

Example Shell script entry:

```
polyspace-cpp -main WinMain  OS-target visual
```

Entry points

This option is used to specify the tasks/entry points to be analyzed by Polyspace software, using a Comma-separated list with no spaces.

These entry points must not take parameters. If the task entry points are functions with parameters they should be encapsulated in functions with no parameters, with parameters passed through global variables instead.

Format:

- All tasks must have the prototype "`void` *`any_name`*`()` .
- It is possible to declare a member function as an entry point of a verification, only and only if the function is declared "`static void task_name()`".

Example Shell Script Entry:

```
polyspace-cpp -entry-points class::task_name,taskname,proc1,proc2
```

Critical section details

```
-critical-section-begin "proc1:cs1[,proc2:cs2]"
```

and

```
-critical-section-end "proc3:cs1[,proc4:cs2]"
```

These options specify the procedures beginning and ending critical sections, respectively. Each uses a list enclosed within double quotation marks ( ), with list entries separated by commas, and no spaces. Entries in the lists take the form of the procedure name followed by the name of the critical section, with a colon separating them.

These critical sections can be used to model protection of shared resources, or to model interruption enabling and disabling.

Limitation:

- Name of procedure accept only void any_name() as prototype.
- The beginning and the end of the critical section need to be defined in same block of code.

Default:

no critical sections.

Example Shell Script Entry:

```
polyspace-cpp -critical-section-begin "start_my_semaphore:cs" \

-critical-section-end "end_my_semaphore:cs"
```

Temporally exclusive tasks

This option specifies the name of a file. That file lists the sets of tasks which never execute at the same time (temporal exclusion).

The format of this file is :

- one line for each group of temporally excluded tasks,
- on each line, tasks are separated by spaces.

Default:

No temporal exclusions.

Example Task Specification file

File named 'exclusions' (say) in the 'sources' folder and containing:

```
task1_group1 task2_group1

task1_group2 task2_group2 task3_group2
```

Example Shell Script Entry :

```
polyspace-cpp -temporal-exclusions-file sources/exclusions \

    -entry-points task1_group1,task2_group1,task1_group2,\

    task2_group2,task3_group2 ...
```

Verify module

The option specifies that Polyspace software generate a main subprogram, if it does not find a main.

Selecting this option allows you to specify the following options:

- Generate a Main Using a Given Class
- Function calls
- First functions to call
- Write accesses to global variables

Default:

Disabled

Example Shell script entry:

```
polyspace-cpp -main-generator
```

Class name

Polyspace for C++ is a class analyzer. If a main program is present in the set of files that you submit for verification then the verification proceeds with that main program. Otherwise, you can choose not to provide a main program and select a single class instead.

The **Class name** option specifies the class for the generated main.

Valid options are:

- **`custom` (default)** – The specified classes are used to generate the main. You must provide a list of classes.
- **`all`** — Every class is used to generate the main.
- **`none`** — No classes are used to generate the main.

Selected member functions (defined by the option `-class-analyzer-calls`) of the specified classes are called by the generated main.

If you select `all` or `custom`, and set `-class-analyzer-calls`, then the option `-main-generator-calls` is automatically set to `unused`, unless you explicitly set another value for `-main-generator-calls`.

If any class name you specify does not exist in the application, the verification will stop. All public and protected function members declared within the class, whether they are called within the code or not, will be analyzed separately and called by a generated main.

This generated main is not code compliant but is visible in the graphical user interface within the `_polyspace_main.cpp` file. Note that it initializes all global variables to random (see “How the Class Analyzer Works”.

Note This option can be specified only if you specify the option `-class-analyzer`.

Example shell script entry:

```
polyspace-cpp -main-generator  class-analyzer MyClass
polyspace-cpp  class-analyzer MyNamespace::MyClass
```

Methods to call within the specified classes

Use this option to verify eligible methods of the classes specified by the option `-class-analyzer`. Eligible methods are static, public, and protected methods of the specified classes.

Values:

- `all` — Default. The generated main calls all public and protected methods of the specified classes. Members inherited from a parent class are not called.
- `all-public` — The generated main calls all public methods of the specified classes but does not call protected methods.
- `inherited-all` — The generated main calls all public and protected methods of the specified classes and methods inherited from the parents of these classes.
- `inherited-all-public` — The generated main calls all public methods of the specified classes and the parents of these classes.
- `unused` — The generated main calls all public and protected methods that are not called within the specified classes.
- `unused-public` — With the exception of protected methods, the generated main calls all methods that are not called within the specified classes.
- `inherited-unused` — The generated main calls all public and protected methods that belong to the specified classes and their parents **and** are not called by another method.

- `inherited-unused-public` — The generated main calls all public methods that belong to the specified classes and their parents **and** are not called by another method.
- `custom` — The generated main calls the methods that you provide in a list.

Default:

`all`

Example Shell Script Entry:

```
polyspace-cpp  class-analyzer MyClass  class-analyzer-calls
unused ...
```

Analyze class contents only

This option can only be used with the option –class-analyzer. If the -class-analyzer option is not used, verification stops and displays an error message.

With the option `class-only`, only functions associated to the specified classes are verified. All functions out of class scope are automatically stubbed even if they are defined in the source code.

Default:

disable

Example Shell Script Entry:

```
polyspace-cpp  class-analyzer MyClass  class-only...
```

Skip member initialization check

By default, Polyspace verification checks for member initialization just after object construction and initialization with `-function-called-before-main` when using `-class-analyzer`.

This option can only be used with `class-analyzer`. If the option `-class-analyzer` is not used, verification stops and displays an error message.

Without this option, in the generated main in `__polyspace_main.cpp` file, you will find some added code checks like on the simple example below using `-class-analyzer` A options:

```
class A {
public: int i ; int *j ; int k; int l;
  A() : i(0), j(0), k(0) { ; }
  A(int a) : i(a), k(0) { ; }
  void foo() {
    i = 1; i++;
    j = (int *) 0x0100; j++;
    l = 1; l++;}
};
```

In `__polyspace_main.cpp` after a call to the constructor(s) and function called before main:

```
check_NIV( __polyspace__A__this->i );  // green NIV
check_NIP( __polyspace__A__this->j );  // orange NIP
check_NIV( __polyspace__A__this->k );/* member has been detected as  \
  never read */
              // gray NIV
check_NIV( __polyspace__A__this->l );  // red NIV
```

- `i` is always initialized, read and written in foo — green NIV
- `j` is initialized in one constructor only, read and written in foo — orange NIP
- `k` is always initialized, but never read and written outside the constructors — gray
- `l` is never initialized in the constructors — red NIV

When this option is applied, **no further check** of member variables' initialization is made.

Default:

Check is made for member scalars, floats and pointer member variables.

Example Shell Script Entry:

```
polyspace-cpp  class-analyzer MyClass no-constructors-init-check
...
```

Functions to call

Use this option with the `-main-generator` option to specify the functions to be called.

Set this option to `unused` (default) when you run a unit-by-unit verification.

Eligible functions:

Every function declared outside a class and defined in the source code to analyze, is considered as eligible when using the option.

The list of functions contains a list of short name (name without signature) separated by comas. If the name of a function from the list is associated to a function not defined in the source code, Polyspace verification stops and displays an error message. If the name of a function from the list is ambiguous, all the functions with the same short name are called. If a function from the list does not belong or is not eligible, Polyspace verification stops and displays an error message. This error message is put in the log file.

Values:

- **`none`** – No function is called. This can be used with a multitasking application without a main, for instance.
- **`unused` (default)** – Call all functions not already called within the code. Inline functions will not be called by the generated main.
- **`all`** – all functions except inline will be called by the generated main.
- **`custom`** – Only functions present in the list are called from the main. Inline functions can be specified in the list and will be called by the generated main.

Selecting `unused`, `all`, or `custom` automatically sets `-class-analyzer none`, unless you explicitly set the `-class-analyzer` option.

An inline (static or extern) function is not called by the generated main program with values all or unused. An inline function can only be called with custom value:

`-main-generator-calls custom=`*`my_inlined_func`*

Example:

```
polyspace-cpp -main-generator -main-generator-calls
custom=function_1,function_2
```

Variables to initialize

This option is used with the main generator options –class-analyzer and –main-generator-calls to dictate how the generated main will initialize global variables.

Settings available:

- *uninit* – main generator writes random on not initialized global variables.
- *none* – no global variable will be written by the main.
- *public* – every variable except static and const variables are assigned a "random" value, representing the full range of possible values
- *all* – every variable is assigned a "random" value, representing the full range of possible values
- *custom* – only variables present in the list are assigned a "random" value, representing the full range of possible values

Example

```
polyspace-cpp  class-analyzer MyClass
-main-generator-writes-variables uninit

polyspace-cpp -main-generator -main-generator-writes-variables
custom=variable_a,variable_b
```

Initialization functions/methods

This option is used with the main generator option –main-generator-calls to specify a function, or list of functions, which will be called before all selected functions in the main.

Eligible functions:

Every function or method defined in the source code to analyze is considered as eligible when using the option.

If the function or method is not overloaded, simply specify the name of the function. If the function or method is overloaded, you must specify the full prototype, including the type of argument (but not the name of argument).

If the function is not defined in the source code, the verification continues with a warning message.

If the function is an out of class function, it is called before any other call.

If the function is a method, it is called after call of its constructor.

If the function is not qualified (if it is not preceeded by its class name, if it has no signature) and if it is ambiguous, every matching function will be called. For example, if a coding convention is to have a function "`init`" in each class, putting `-function-called-before-main init` implies that in each class, the `init` function will be called after the constructor call.

If two `init` functions in a class have different parameter types, the two functions will be called.

Unit-by-unit verification:

When performing unit-by-unit verification (using use the option `-unit-by-unit`) the behavior of `-function-called-before-main` changes depending on the type of init function you specify.

When you set the option `-function-called-before-main` in unit-by-unit mode:

- If the `init` function is an out of class function, it is called at the beginning of the generated main (before the "if random" block of classes).
- If the `init` function is a method (function member of a class), it is called after all constructor calls of the corresponding class. If several classes are present in the unit, the software displays a warning explaining that the function called before main will be called only with the concerned class.

Example:

```
polyspace-cpp -main-generator-calls unused
-function-called-before-main MyFunction1,MyFunction2
```

Run unit by unit verification

This option creates a separate verification job for each source file in the project.

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

This option is only available for server verifications. It is not compatible with multitasking options such as `-entry-points`.

A unit-by-unit verification calls functions and classes as follows:

1. All UNUSED functions are called.
2. All UNUSED public and protected member functions of a class are called.

For example, in the following class `Base`, only `reqOn` is called by the generated main:

```
class Base
{
public:
void reqOn(void);
virtual bool actOn(void);
};

#include "Base.h"
```

```
void Base::reqOn ()
{
bool a;
a = actOn();
}

bool Base::actOn ()
{
bool a;
return a;
}
```

Default:

Not selected

Example Shell Script Entry:

```
polyspace-cpp -unit-by-unit
```

Unit by unit common source files

Specifies a list of files to include with each unit verification. These files are compiled once, and then linked to each unit before verification. Functions not included in this list are stubbed.

Default:

None

Example Shell Script Entry:

```
polyspace-cpp -unit-by-unit-common-source
c:/polyspace/function.cpp
```

Variable/function range setup

This option permits the setting of specific data ranges for a list of given global variables.

For more information, see “Specify Data Ranges for Variables and Functions (Contextual Verification)” .

File format:

The file filename contains a list of global variables with the below format:

```
variable_name val_min val_max <init|permanent|globalassert>
```

Variables scope:

Variables do not have to be defined variables, that is, could be `extern` with option `-allow-undef-variables`.

Note Only one mode can be applied to a global variable.

No checks are added with this option except for `globalassert` mode.

Some warning can be displayed in log file concerning variables when format or type is not in the scope.

Default:

Disable.

Example shell script entry:

```
polyspace-cpp -data-range-specifications range.txt ...
```

No automatic stubbing

By default, Polyspace verification automatically stubs all functions. When this option is used, the list of functions to be stubbed is displayed and the verification is stopped.

Benefits:

This option may be used where

- The entire code is to be provided, which may be the case when analyzing a large piece of code. When the verification stops, it means the code is not complete.
- Manual stubbing is preferred to improve the selectivity and speed of the verification.

Default:

All functions are stubbed automatically

No STL stubs

Polyspace provide an efficient implementation of part of the Standard library (STL). This implementation may not be compatible with includes files of the applications. In that case some linking errors could arise.

With this option Polyspace does not use this implementation of the STL.

Example Shell Script Entry:

```
polyspace-cpp -no-stl-stubs ...
```

Functions to stub

Specifies functions that you want the software to stub.

Enter a comma-separated list of functions. For example, `function_1,function_2`.

When entering function names, two syntaxes are supported for C++:

- Basic syntax, with extensions for classes and templates:

Function Type	Syntax
Simple function	`test`
Class method	`A::test`
Template method	`A<T>::test`

- Syntax with function arguments, to differentiate overloaded functions. Function arguments are separated with semicolons:

Function Type	Syntax
Simple function	`test()`
Class method	`A::test(int;int)`
Template method	`A<T>::test(T;T)`

Note All overloaded versions of the function will be discarded.

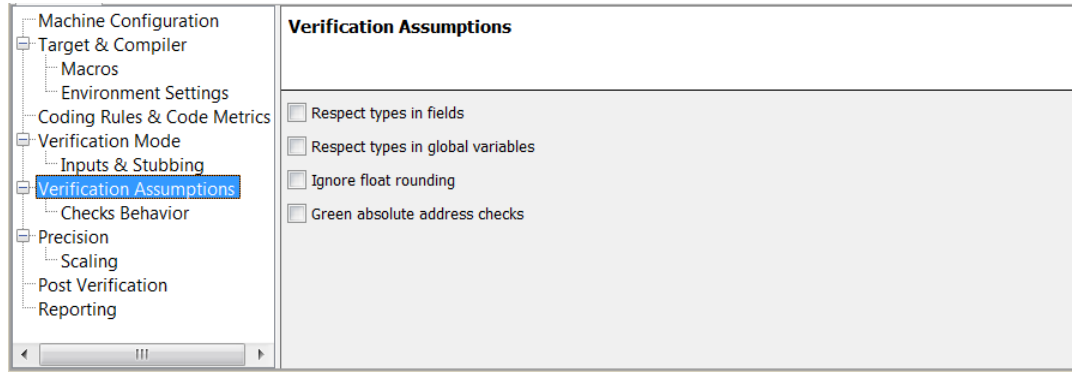
The following special characters are allowed for C++:

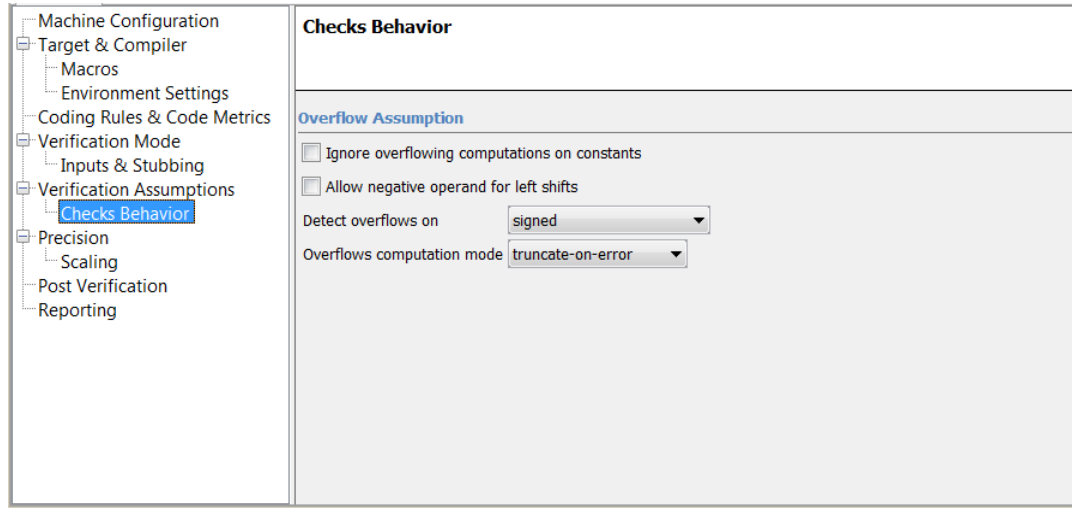
`( ) < > ; _ * & [ ]` and space.

Example Shell Script Entry:

```
polyspace-cpp -functions-to-stub function_1,function_2 ...
```

Verification Assumptions





In this section...

In this section...

Respect types in fields

This is a scaling option, designed to help process complex code. When it is applied, Polyspace verification assumes that structure fields not declared as containing pointers are never used for holding pointer values. This option should only be used with Type-safe code, when it does not cause a loss of precision. See also -respect-types-in-globals .

In the following example, we will lose precision using option `respect-types-in-fields` option:

```
struct {
 unsigned x;
 int f1;
 int *z[2];
} S1;

void funct2(void) {
 int *tmp;
 int y;
 ((int**)&S1)[0] = &y;  /* S1.x points on y */
 tmp = (int*)S1.x;
 y=0;
 *tmp = 1;      /* write 1 into y */
 assert(y==0);
}
```

Polyspace verification will not take care that S1.x contains the address of y resulting a green assert.

Default:

Polyspace verification assumes that structure fields may contain pointer values.

Example Shell Script Entry:

```
polyspace-cpp -respect-types-in-fields ...
```

Respect types in global variables

This is a scaling option, designed to help process complex code. When it is applied, Polyspace verification assumes that global variables not declared as containing pointers are never used for holding pointer values. This option should only be used with Type-safe code, when it does not cause a loss of precision. See also -respect-types-in-fields.

In the following example, we will lose precision using the –respect-types-in-globals option:

```
int x;
void t1(void) {
 int y;
 int *tmp = &x;
 *tmp = (int)&y;
 y=0;
 *(int*)x = 1; // x contains address of y
 assert (y == 0); // green with the option
}
```

Polyspace verification will not take care that x contains the address of y resulting a green assert.

Default:

Polyspace verification assumes that global variables may contain pointer values.

Example Shell Script Entry:

```
polyspace-cpp -respect-types-in-globals ...
```

Ignore float rounding

Without this option, Polyspace verification rounds floats according to the IEEE 754 standard: simple precision on 32-bits targets and double precision on target which define double as 64-bits. With the option, **exact** computation is performed.

Example

```
1
2 void ifr(float f)
3 {
4 double a = 1.27;
5  if ((double)1.27F == a) {
6   assert (1);
7    f = 1.0F * f;
8     // reached when -ignore-float-rounding is used or not
9  }
10 else {
11   assert (1);
12   f = 1.0F * f;
13   // reached when compiled under Visual and when
-ignore-floatrounding is not used
14  }
15 }
```

Using this option can lead to different results compared to the "real life" (compiler and target dependent): Some paths will be reachable or not for Polyspace verification while they are not (or are) depending of the compiler and target. So it can potentially give approximate results (green should be unproven). This option has an impact on OVFL checks on floats.

However, this option allows reducing the number of unproven checks because of the “delta” approximation.

For example:

- FLT_MAX (with option set) = 3.40282347e+38F
- FLT_MAX (following IEEE 754 standard) = 3.40282347e+38F ± Δ

```
1
2 void ifr(float f)
3 {
4  double a = 1.27;
5  if ((double)1.27F == a) {
6   assert (1);
7   f = 1.0F * f;  // Overflow never occurs because f <= FLT_MAX.
8        // reached when -ignore-float-rounding is used
9   }
10  else {
11   assert (1);
12   f = 1.0F * f;  // OVFL could occur when f = (FLT_MAX + D)
13        // reached when -ignore-float-rounding is not used
14  }
15 }
```

Default:

IEEE 754 rounding under 32 bits and 64 bits.

Example Shell Script Entry:

```
polyspace-cpp -ignore-float-rounding ...
```

Green absolute address checks

If you know that the absolute addresses in your code are valid, you can specify this option to make all ABS_ADDR checks green. Otherwise, the software generates an orange ABS_ADDR check when an absolute address is assigned to a pointer. This is because the software has no information about the absolute address and therefore cannot verify, for example, the address, availability of memory, and initialization of memory.

The software permits memory access to the absolute address after generating the orange ABS_ADDR check for the first assignment operation. IDP and NIV checks for memory access operations after the first assignment operation are green.

Default:

Orange ABS_ADDR check generated when an absolute address is assigned to a pointer.

Example Shell Script Entry

```
polyspace-cpp -green-absolute-address-checks ...
```

Ignore overflowing computations on constants

This option specifies that the verification should be permissive with regards to overflowing computations on constants. Note that it deviates from the ANSI C standard.

For example,

```
char x = 0xff;
```

causes an overflow according to the standard, but if it is analyzed using this option it becomes effectively the same as

```
char x = -1;
```

With this second example, a red overflow will result irrespective of the use of the option.

char x = (rnd?0xFF:0xFE);

Default:

`char x = 0xff;` causes an overflow

Example Shell Script Entry:

```
polyspace-cpp -ignore-constant-overflows ...
```

Allow negative operand for left shifts

This option allows a shift operation on a negative number.

According to the ANSI standard, such a shift operation on a negative number is illegal – for example,

-2 << 2

With this option in use, Polyspace verification considers the operation to be valid. In the previous example, the result would be
-2 << 2 = -8

Default:

A shift operation on a negative number causes a red error.

Example Shell Script Entry:

```
polyspace-cpp -allow-negative-operand-in-shift ...
```

Detect overflows on

Specifies how verification handles overflowing computations on integers.

Possible settings are:

- **`none`** – The verification does not check for integer computation overflows. If a computation value exceeds the range of the result type, the result is wrapped, and no OVFL check is reported.

  For example, `MAX_INT + 1` wraps to `MIN_INT`.

- **`signed`** (default) – Verification checks all signed integer computations and signed integer to signed integer conversions. The option `-scalar-overflows-behavior` specifies whether results for signed integers are restricted to an acceptable value or wrapped.

  For unsigned integer operations, the results are wrapped, and no OVFL check is reported.

  This behavior conforms to the ANSI C (ISO C++) standard.

- **`signed-and-unsigned`** – Verification checks all integer computations and integer conversions, including conversions that cause a change of signs.

The option `-scalar-overflows-behavior` specifies whether operation results are restricted to an acceptable value or wrapped.

This behavior is more strict than the ANSI C (ISO C++) standard requires.

Consider the examples below.

Example 1

Using the `signed-and-unsigned` option, the following example generates an error:

```
unsigned char x;
x = 255;
x = x+1;   //overflow due to this option
```

Using the `none` option, however, the example does not generate an error.

```
unsigned char x;
x = 255;
x = x+1;  // turns x into 0 (wrap around)
```

Example 2

Using the `signed-and-unsigned` option, the following example generates an error:

```
unsigned char Y=1;
Y = ~Y;  //overflow because of type promotion
```

In this example:

1. `Y` is coded as an unsigned char: `00000001`
2. `Y` is promoted to an integer: `00000000 00000000 00000000 00000001`
3. The operation "~" is performed, making `Y`: `11111111 11111111 11111111 11111110`
4. The integer is downcast to an unsigned char, causing an overflow.

Example Shell Script Entry:

```
polyspace-cpp -scalar-overflow-checks signed...
```

Overflows computation mode

Specifies how verification computes the results of overflowing integer computations or integer conversions.

Possible settings are:

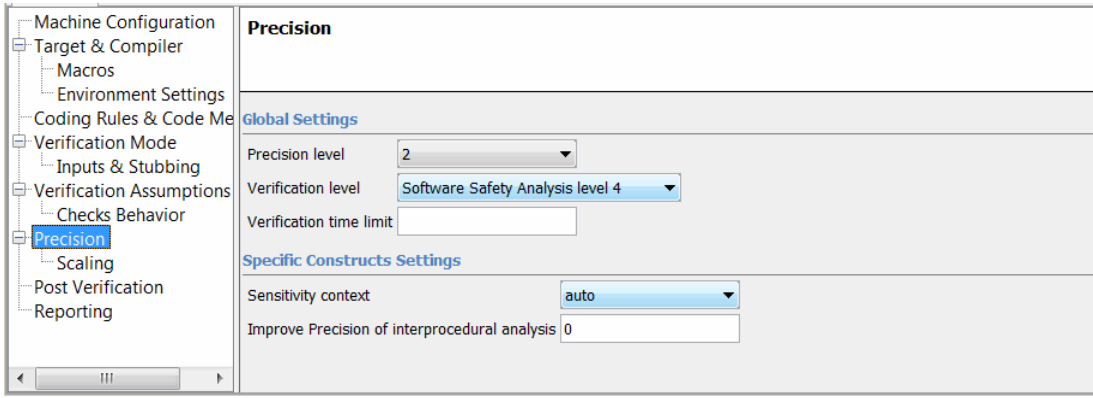
- **`truncate-on-error`** (default) — Result of an overflowing operation is restricted to an acceptable value. If the check is red, verification stops (in the current code). If the check is orange, verification continues with restricted value.
- **`wrap-around`** — Result of an overflowing operation wraps around the type range. The check has no impact on values for the rest of the verification.

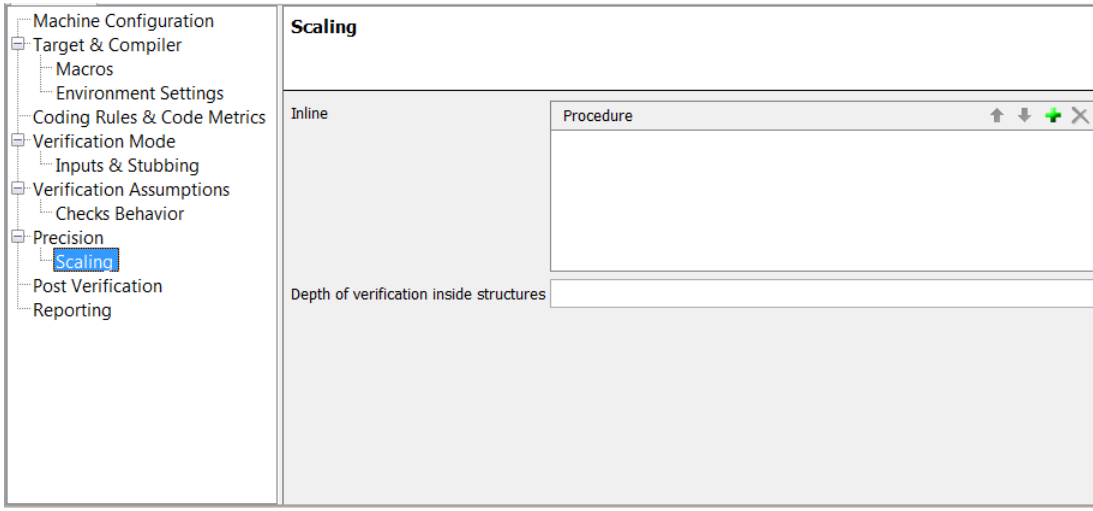
  For example, `MAX_INT + 1` wraps to `MIN_INT`.

Example Shell Script Entry:

```
polyspace-cpp -scalar-overlows-behavior wrap-around ...
```

Precision





In this section...

In this section...

Tuning Precision and Scaling Parameters

Precision versus Time of Verification

There is a compromise to be made to balance the time required to obtain results, and the precision of those results. Consequently, launching Polyspace verification with the following options will allow the time taken for verification to be reduced but will compromise the precision of the results. It is suggested that the parameters should be used in the sequence shown - that is, if the first suggestion does not increase the speed of verification sufficiently then introduce the second, and so on.

- switch from -O2 to a lower precision;
- set the -respect-types-in-globals and -respect-types-in-fields options;
- set the -k-limiting option to 2, then 1, or 0;
- stub manually missing functions which write into their arguments.

Precision versus Code Size

Polyspace verification can make approximations when computing the possible values of the variables, at any point in the program. Such an approximation will always use a superset of the actual possible values.

For instance, in a relatively small application, Polyspace verification might retain very detailed information about the data at a particular point in the code, so that for example the variable VAR can take the values { -2; 1; 2; 10; 15; 16; 17; 25 }. If VAR is used to divide, the division is green (because 0 is not a possible value). If the program being analyzed is large, Polyspace verification would simplify the internal data representation by using a less

precise approximation, such as [-2; 2] U {10} U [15 ; 17] U {25} . Here, the same division appears as an orange check.

If the complexity of the internal data becomes even greater later in the verification, Polyspace verification might further simplify the VAR range to (say) [-2; 20].

This phenomenon leads to the increase or the number of orange warnings when the size of the program becomes large.

Note The amount of simplification applied to the data representations also depends on the required precision level (O0, O2), Polyspace verification will adjust the level of simplification:

- `-O0`: shorter computation time. You only need to focus on red and gray checks.
- `-O2`: less orange warnings.
- `-O3`: less orange warnings and bigger computation time.

Precision level

This option specifies the precision level to be used. It provides higher selectivity in exchange for more verification time, therefore making results review more efficient and hence making bugs in the code easier to isolate. It does so by specifying the algorithms used to model the program state space during verification.

Begin with the lowest precision level. Red errors and gray code can then be addressed before rerunning the Polyspace verification with higher precision levels.

Benefits:

- A higher precision level contributes to a higher selectivity rate, making results review more efficient and hence making bugs in the code easier to isolate.

- A higher precision level also means higher verification time
  - -O0 corresponds to static interval verification.
  - -O1 corresponds to complex polyhedron model of domain values.
  - -O2 corresponds to more complex algorithms to closely model domain values (a mixed approach with integer lattices and complex polyhedrons).
  - -O3 is only suitable for code smaller than 1000 lines of code. For such codes, the resulting selectivity might reach high values such as 98%, resulting in a very long verification time, such as an hour per 1000 lines of code.

Default:

-O2

Example Shell Script Entry:

```
polyspace-cpp -O1 -to pass4 ...
```

Verification level

This option specifies the verification phase after which the verification will stop.

Benefits:

This option provides improved selectivity, making results review more efficient and making bugs in the code easier to isolate.

- A higher integration level contributes to a higher selectivity rate, leading to "finding more bugs" with the code.
- A higher integration level also means longer verification time

Possible values:

- `cpp-compliance` or "`C++ source compliance checking`"
- `cpp-normalize` or "`C++ source normalization`" — Not available from the Polyspace verification environment

- `cpp-link` or "`C++ Link`" — Not available from the Polyspace verification environment
- `cpp-to-il` or "`C++ to Intermediate Language`" — Not available from the Polyspace verification environment
- `pass0` or "`Software Safety Analysis level 0`"
- `pass1` or "`Software Safety Analysis level 1`"
- `pass2` or "`Software Safety Analysis level 2`"
- `pass3` or "`Software Safety Analysis level 3`"
- `pass4` or "`Software Safety Analysis level 4`" — Default
- `other` (stop verification after level 20)

Note If you use `-to other`, then verification will continue until you stop it manually (via *`Polyspace_Install`*`/bin/kill-rte-kernel "`*`Results folder`*`"/"`*`log file name`*`"`) or stops until it has reached `pass20`.

Default:

pass4

Example Shell Script Entry:

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

```
polyspace-cpp -to pass0 ...
```

Verification time limit

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

If the verification does not complete within the specified time, the verification fails.

You can specify fractions of an hour in decimal form. For example:

- `-timeout 5.75` – Five hours, 45 minutes.
- `-timeout 3,5` – Three hours, 30 minutes.

Example Shell Script Entry :

```
polyspace-cpp -timeout 5.75 ...
```

Sensitivity context

Add call context information for checks contained in given functions. For example, if one call of the function results in a red check, and another call results in a green check, the call information and color for both calls is kept in the function check.

- `none` — No context sensitivity.
- `auto` — Automatically select functions for which context sensitivity is applied.
- `custom` — Apply context sensitivity to functions that you specify.

Example Shell Script Entry:

```
polyspace-cpp -context-sensitivity -auto ...
```

Improve precision of interprocedural analysis

This option is used to improve interprocedural verification precision within a particular pass (see -to *pass1*, *pass2*, *pass3* or *pass4*). The propagation of information within procedures is done earlier than usual when this option is specified. That results in improved selectivity and a longer verification time.

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

- a level 1 analysis in (with) (pass1) will provide results equivalent to level 1 or 2 in the (without) analysis
- a level 1 analysis in (with) can last x times more than a cumulated level 1+2 analysis from (without). "x" might be exponential.

- the same applies to level 2 in (with) equivalent to level 3 or 4 in (without), with potentially exponential analysis time for (a)

Gains using the option

- (+) highest selectivity obtained in level 2. no need to wait until level 4
- (-) This parameter increases exponentially the analysis time and might be even bigger than a cumulated analysis in level 1+2+3+4
- (-) This option can only be used with less than 1000 lines of code

Default:

0

Example Shell Script Entry:

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

Inline

A scaling option that creates a clone of a each specified procedure for each call to it.

Cloned procedures follow a naming convention:

```
procedure1_pst_inlined_nb
```

where nb is a unique number giving the total number of inlined procedures.

Inlining allows the number of aliases in a given procedure to be reduced, and it may also improve precision.

It can also allow you to more easily locate run-time errors that relate the copy or set of a large structure to a smaller one (NTC, for instance).

Restrictions :

- **Extensive use** of this option may duplicate too much code and may lead to other scaling problems. Carefully choose procedures to inline.

- This option should be used in response to the inlining hints provided by the alias verification (the log file can sometimes provide this kind of information).
- This option should not be used on main, task entry points and critical section entry points.
- When using this option with a method of a class, all overload of the method will apply to the inline.

Example Shell Script Entry:

```
polyspace-cpp  inline  myclass::myfunc  ...
```

Depth of analysis inside structures

This is a scaling option to limit the depth of verification into nested structures during pointer verification (see Tuning Precision and Scaling Parameters).

This option is only available for C and C++.

Default:

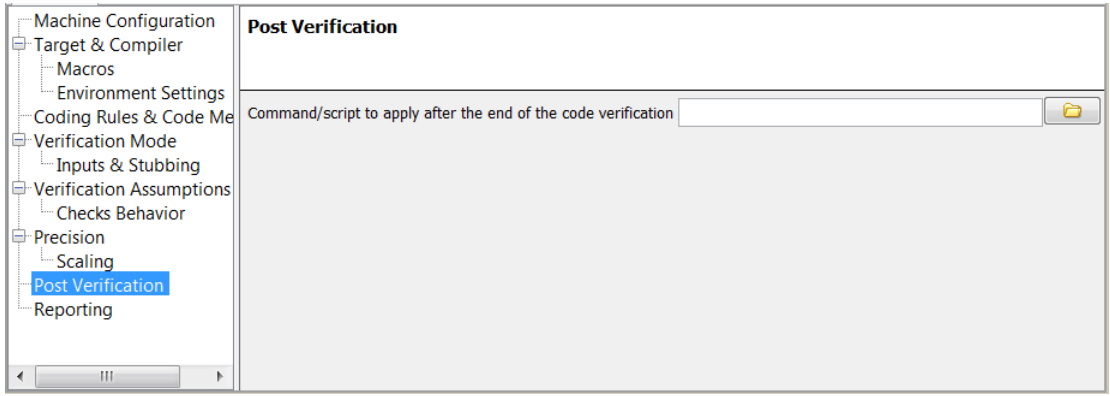
There is no fixed limit.

Example Shell Script Entry:

```
polyspace-cpp -k-limiting 1 ...
```

In this example above, verification will be precise to only one level of nesting.

Post Verification



Command/script to apply after the end of the code

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

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

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

Note Depending on the architecture used (notably when performing a server verification), the script can be executed on the client side or the server side.

Default:

No command.

Example Shell Script Entry – file name:

This example shows how to send an email to tip the client side off that his verification has been ended. This example supposes that the mailx command is available on the machine. So the command looks like:

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

where `end_email` is your Perl script.

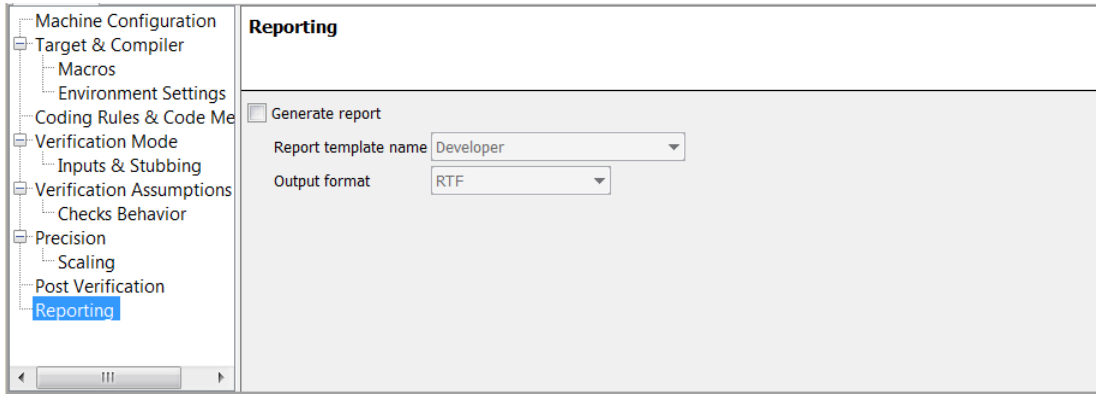
Note If you are running Polyspace software version 5.1 (r2008a) or later on a Windows system, you cannot use Cygwin shell scripts. Since Cygwin is no longer included with Polyspace software, all files must be executable by Windows. To support scripting, the Polyspace installation now includes Perl. You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

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

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

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

Reporting



In this section...

Generate report

Specify whether to create verification report using report generation options

Settings

Default: Off

 On

Create report.

 Off

No report created.

Report template name

Specify template for generating verification report

Settings

Default:

```
C:\Polyspace\Polyspace_Common\ReportGenerator\templates\Developer.rpt
```

Report templates provided with the software include:

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

Tip

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

Command-Line Information

Parameter: `report-template`
Type: string
Value: any valid script file name
Example: `polyspace-cpp -report-template` *`filepath`*`\my_template`

Output format

Specify output format of report

Settings

Default: `RTF`

`RTF`
: Generate an .rtf format report.

`HTML`
: Generate an .html format report.

`PDF`
: Generate a .pdf format report.

`Word`
: Generate a .doc format report.

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

`XML`
: Generate and .xml format report.

Note WORD format is not available on UNIX platforms, RTF format is used instead.

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

Command-Line Information

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

Shell script example:

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

Batch Options

In this section...

-server

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

Note If you do not specify a server name or IP address, the default server referenced in the Polyspace Preferences is used.

When a `server` option is associated to the batch launching command, the name or IP address and a port number need to be specified. If the port number does not exist, the 12427 value will be used by default.

Note polyspace-remote- accepts all other options.

Option Example Shell Script Entry:

```
polyspace-remote-desktop-cpp  server 192.168.1.124:12400

polyspace-remote-cpp

polyspace-remote-cpp  server Bergeron
```

`-sources`

`-sources "file1[ file2[ ...]]"` (Linux and Solaris)

or

`-sources "file1[,file2[, ...]]"` (Windows, Linux and Solaris)

List of source files to be analyzed, double-quoted and separated by commas. Note that UNIX standard wild cards are available to specify a number of files.

Note The specified files must have valid extensions. The extensions are not case-sensitive: `*.(c|C|cc|cpp|cPp|CPP|cxx|CxX|CXX)`

Defaults:

```
sources/*.(c|C|cc|cpp|cPp|CPP|cxx|CxX|CXX)
```

Example Shell Script Entry under linux or solaris *(files are separated with a white space)*:

```
polyspace-cpp -sources "my_folder/*.cpp"
polyspace-cpp -sources "my_folder/file1.cc other_dir/file2.cpp"
```

Example Shell Script Entry under windows (*files are separated with a comma):*

```
polyspace-cpp -sources "my_folder/file1.cpp,other_dir/file2.cc"
```

Using `-sources-list-file`, each file *name* need to be given with an absolute path. Moreover, the syntax of the file is the following:

- One file by line.
- Each file name is given with its absolute path.

Note This option is only available in batch mode.

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

```
polyspace-cpp -sources-list-file "C:\Analysis\files.txt"
polyspace-cpp -sources-list-file "/home/poly/files.txt"
```

-sources-list-file

This option is only available in batch mode. The syntax of *file_name* is the following:

- One file per line.
- Each file name includes its absolute or relative path.

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

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

polyspace-cpp -sources-list-file "/home/poly/files.txt"
```

-main-generator-files-to-ignore

Specifies source files containing functions not called by the automatically generated main.

Enter a comma-separated list of files or folders for which defined functions are not called by the automatically generated main.

If you specify a folder, all files in the folder and its subfolders are ignored by the main generator.

Use this option for files containing function bodies, so that the verification looks for the function body only when the function is called by a primary source file and no body is found.

This option applies only to the automatically generated main. Therefore, you must also set the option `-main-generator` for this option to take effect.

Note You can set this option in the Polyspace Verification Environment by right-clicking a source file in the Project Browser Source tree.

Example Shell Script Entries:

```
polyspace-cpp -main-generator  main-generator-files-to-ignore
"my_folder/file.c"

polyspace-cpp -main-generator  main-generator-files-to-ignore
"my_folder/file1.c,my_folder2/file2.c"

polyspace-cpp -main-generator  main-generator-files-to-ignore
"my_folder"
```

-v | -version

Display the Polyspace version number.

Example Shell Script Entry:

```
polyspace-cpp  v
```

It will show a result similar to:

```
Polyspace r2008a

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

-h[elp]

Display in the shell window a simple help in a textual format giving information on all options.

Example Shell Script Entry:

```
polyspace-cpp  h
```

-prog

This option specifies the application name, using only the characters which are valid for Unix file names. This information is labelled in the GUI as the *Project Name*.

Note The Project Name (Session Identifier) option no longer appears in the General section of the Analysis options GUI. You specify the Project name, Version, and Author parameters in the Polyspace Project – Properties dialog box. For more information, see "Create Verification Project".

Default:

Shell Script: polyspace

GUI: New_Project

Example shell script entry:

```
polyspace-cpp -prog myApp ...
```

-date

This option specifies a date stamp for the verification in dd/mm/yyyy format. This information is labelled in the GUI as the *Date*. The GUI also allows alternative default date formats, via the Edit/Preferences window.

Note The Date option no longer appears in the General section of the Analysis options GUI. The date is now set automatically in the GUI.

Default:

Day of launching the verification

Example shell script entry:

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

-author

This option is used to specify the name of the author of the verification.

Note The Author option no longer appears in the General section of the Analysis options GUI. You specify the Project name, Version, and Author parameters in the Polyspace Project – Properties dialog box. For more information, see “Create Verification Project”.

Default:

the name of the author is the result of the *whoami* command

Example shell script entry:

```
polyspace-cpp -author "John Tester"
```

-verif-version

Specifies the version identifier of the verification. This option can be used to identify different verifications. This information is identified in the GUI as the *Version*.

Note The Version option no longer appears in the General section of the Analysis options GUI. You specify the Project name, Version, and Author parameters in the Polyspace Project – Properties dialog box. For more information, see "Create Verification Project".

Default:

1.0.

Example shell script entry:

```
polyspace-cpp -verif-version 1.3 ...
```

-results-dir

This option specifies the folder in which Polyspace will write the results of the verification. Note that although relative directories may be specified, particular care should be taken with their use especially where the tool is to be launched remotely over a network, and/or where a project configuration file is to be copied using the "Save as" option.

Default:

Shell Script: The folder in which tool is launched.

From Graphical User Interface: C:\Polyspace_Results

Example Shell Script Entry:

```
polyspace-cpp -results-dir RESULTS ...

export RESULTS=results_`date +%d%B_%HH%M_%A`
```

```
polyspace-cpp -results-dir `pwd`/$RESULTS ...
```

-I

Specify the name of a folder to be included when compiling C++ sources. Only one folder may be specified for each –I, but the option can be used multiple times.

Polyspace software automatically adds the following standard include folders after any includes you specify:

- *Polyspace_Install*`/Verifier/include/include-gnu`
- *Polyspace_Install*`/Verifier/include/include-gnu/next`

Default:

- The `./sources` folder (if it exists), is implicitly added at the end of the "`-I`" list
- If you do not specify any include folders, the two standard include folders and the `./sources` folder (if it exists) are automatically included

Example Shell Script Entry-1:

```
polyspace-cpp -I /com1/inc -I /com1/sys/inc
```

is equivalent to

```
polyspace-cpp -I /com1/inc -I /com1/sys/inc -I ./sources
```

Example Shell Script Entry-2:

```
polyspace-cpp
```

is equivalent to

```
polyspace-cpp -I ./sources
```

-from

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

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

The option is usually used in a verification after one run with the `-to` option, although it can also be used to recover after power failure.

Possible values are as described in the `-to` *`verification-phase`* section, with the addition of the `scratch` option.

Note

- This option can only be used for client verifications. All server verifications start from `scratch`.
- Unless the `scratch` option is used, this option can be used only if the previous verification was launched using the option `-keep-all-files`.
- This option cannot be used if you modify the source code between verifications.

Default :

From `scratch`

Example Shell Script Entry :

```
polyspace-cpp -from c-to-il ...
```

-import-comments

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

Default:

Disabled

Example Shell Script Entry:

```
polyspace-c -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. This action may affect processing speed if the results folder is mounted on a network drive. Use this option only when the temporary folder partition is not large enough and troubleshooting is required.

Default:

Disabled

Example Shell Script Entry:

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

-less-range-information

Limits the amount of range information displayed in verification results.

When you select this option, the software provides range information on assignments, but not on reads and operators.

In addition, selecting this option enables the `no-pointer-information` option. See "`-no-pointer-information`" on page 2-86.

Computing range information for reads and operators may take a long time, and can reduce the precision of the verification (causing more orange checks). Selecting this option can reduce verification time significantly, and improve the precision of the verification. Consider the following example:

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

If you do not select this option (the default), the software displays range information when you place the cursor over `x`, `y`, `z`, or `+`. However, if you select this option, the software displays range information only when you place the cursor over `x`.

Default:

Disabled.

Example Shell Script Entry :

```
polyspace-cpp -less-range-information
```

-no-pointer-information

Stops the display of pointer information in verification results.

When you select this option, the software does not provide pointer information through tooltips. As computing pointer information may take a long time, selecting this option can significantly reduce verification time.

Consider the following example:

```
x = *p;
```

If you do not select this option (the default), the software displays pointer information when you place the cursor on `p` or `*`. If you select this option, the software does not display pointer information.

Default:

Disabled.

Example Shell Script Entry :

```
polyspace-cpp -no-pointer-information
```

-keep-all-files

"-keep-all-files" on page 2-87

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

Settings

Default: Off

 On

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

 Off

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

Tips

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

Command-Line Information

Parameter: `-keep-all-files`
Example: `polyspace-cpp -keep-all-files`

-permissive

This option selects the permissive verification mode, which produces results quickly but may not detect all run-time errors.

The permissive mode is equivalent to using the following options:

- `-ignore-constant-overflows`
- `-allow-negative-operand-in-shift`

Example Shell Script Entry

```
polyspace-cpp -permissive ...
```

`-Wall`

Force the C++ compliance phase to print all warnings.

Note If you specify `-jsf-coding-rules`, this option is disabled.

Default:

By default, only warnings about compliance across different files are printed.

Example Shell Script Entry:

```
polyspace-cpp -Wall ..
```

`-report-output-name`

"`-report-output-name`" on page 2-88

Specify name of verification report file

Settings

Default: *`Prog_TemplateName.Format`* where:

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

Command-Line Information

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

Shell script example:

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

Deprecated Options

In this section...

-continue-with-existing-host (Deprecated)

Note This option is deprecated in R2010a and later releases, and no longer exists in the user interface. Polyspace verification now continues regardless of the system configuration. The software still checks the hardware configuration, and issues a warning if it does not satisfy requirements.

When this option is set, the verification will continue even if the system is under specified or its configuration is not as preferred by Polyspace software. Verified system parameters include the amount of RAM, the amount of swap space, and the ratio of RAM to swap.

-allow-unsupported-linux (Deprecated)

Note This option is deprecated in R2010a and later releases, and no longer exists in the user interface. Polyspace verification now continues regardless of the Linux distribution. If the Linux distribution is not officially supported, the software displays a warning in the log file.

This option specifies that Polyspace verification will be launched on an unsupported OS Linux distribution.

Polyspace software supports the Linux distributions listed in "Hardware and Software Requirements".

-quick (Deprecated)

Note This option is deprecated in R2009a and later releases.

`quick` mode is obsolete and has been replaced with verification PASS0. PASS0 takes somewhat longer to run, but the results are more complete. The limitations of `quick` mode, (no NTL or NTC checks, no float checks, no variable dictionary) no longer apply. Unlike `quick` mode, PASS0 also provides full navigation in the results.

This option is used to select a very fast mode for Polyspace .

Benefits

This option allows results to be generated very quickly. These are suitable for initial verification of red and gray errors only, as orange checks are too plentiful to be relevant using this option.

Limitations

- No NTL or NTC are displayed (non termination of loop/call)
- The variable dictionary is not available
- No check is performed on floats
- The call tree is available but navigation is not possible
- Orange checks are too plentiful to be relevant

3

Check Descriptions for C Code

UNR – Unreachable Code

This is a check to establish whether different code snippets (assignments, returns, conditional branches and function calls) are dead, such that they can never be accessed during the normal execution of the software. Dead, or Unreachable, code is represented by means of a gray coding on every check, with supplementary UNR checks also being added.

Consider the following example.

```
1
2 #define True 1
3 #define False 0
4
5 typedef enum {
6  Intermediate, End, Wait, Init
7 } enumState;
8
9 // pure stub
10 int intermediate_state(int);
11 int random_int(void);
12
13 int State (enumState stateval)
14 {
15  volatile int random;
16  int i;
17  if (stateval == Init) return False;
18  return True;
19 }
20
21 int main (void)
22 {
23  int i, res_end;
24  enumState inter;
25
26  res_end = State(Init);
27  if (res_end == False) {
28   res_end = State(End);
29   inter = (enumState)intermediate_state(0);
30   if (res_end || inter == Wait) {    // UNR code on inter
```

```
== Wait
31    inter = End;
32   }
33   // use of I not initialized
34   if (random_int()) {
35    inter = (enumState)intermediate_state(i); // NIV ERROR
36    if (inter == Intermediate) {    // UNR code because
of NIV ERROR
37     inter = End;
38    }
39   }
40  } else {
41   i = 1; // UNR code
42   inter = (enumState)intermediate_state(i);  // UNR code
43  }
44  if (res_end) { // UNR code always reached, but no else
45   inter = End;
46  }
47
48  return res_end;
49 }
50
```

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

- At line 30, the first part of a two part test is always true. The other part is never evaluated, following the standard definition of logical operator "||".
- The piece of code after a red error is never evaluated by Polyspace software. The call to the function on line 35 and the line following it are considered to be dead code. Correcting the red error and relaunching would allow the color to be revised.
- At line 27, the test is always true (`if {` part), and the first branch is always executed. Consequently there is dead code in the other branch (that is, in the `else {` part at lines 41 to 42).

In addition, at line 44, there is an `if` statement without an `else` clause. In this instance, because there is no `else` clause and `res_end` is *always* true, the `if` keyword is colored **gray**.

OBAI – Out of Bounds Array Index

This is a check to establish whether an index accessing an array is compatible with the length of that array.

The message associated with an OBAI check provides the range of the array. For example: Array index out of bounds [0..1023].

Consider the following example.

```
1
2 #define TAILLE_TAB 1024
3 int tab[TAILLE_TAB];
4
5 void main(void)
6 {
7  int index;
8
9  for (index = 0; index < TAILLE_TAB ; index++)
10   {
11    tab[index] = 0;
12   }
13  tab[index] = 1;
14 // OBAI ERROR: Array index out of bounds [0..1023]
15 }
```

Just after the loop, *index* equals *SIZE_TAB*. Thus *tab[index] = 1* overwrites the memory cell just after the last array element.

An OBAI check can also be localized on a + operator, as another example illustrates.

```
1 int tab[10];
2
3 void main(void)
4 {
5   int index;
6  for (index = 0; index < 10 ; index++)
7   *(tab + index) = 0;
8
```

```
9  *(tab + index) = 1; // OBAI ERROR: Array index out of bounds
10 }
```

ZDV – Division by Zero

This is a check to establish whether the right operand of a division (that is, the denominator) is different from 0[.0]. Consider the following example.

```
1 extern int random_value(void);
2
3 void zdvs(int p)
4 {
5  int i, j = 1;
6  i = 1024 / (j-p); // ZDV ERROR: Scalar Division by Zero
7 }
8
9 void zdvf(float p)
10 {
11  float i,j = 1.0;
12  i = 1024.0 / (j-p); // ZDV ERROR: float Division by Zero
13 }
14
15 int main(void)
16 {
17  volatile int random;
18  if (random_value()) zdvs(1);
// NTC ERROR: because of ZDV ERROR
in ZDVS.
19  if (random_value()) zdvf(1.0);
// NTC ERROR: because of ZDV ERROR
in ZDVF.
20 }
```

NIV (NIVL) – Non-Initialized Variable

This is a check to establish whether a variable is initialized before being read. Consider the following example.

```
1
2 extern int random_int(void);
3
4 void main(void)
5 {
6  int x,i;
7  double twentyFloat[20];
8  int y = 0;
9
10  if (random_int()) {
11   y += x; // NIV ERROR: Non
Initialized Variable (type: int 32)
12  }
13  if (random_int()) {
14   for (i = 1; i < 20; i++) {
15    if (i % 2) twentyFloat[i] = 0.0;
16   }
17   twentyFloat[2] = twentyFloat[4] + 5.0; // NIV Warning.
Only odd indexes are initialized.
18  }
19 }
```

The result of the addition is unknown at line 11 because *x* is not initialized (UNR unreachable code on "+" operator).

In addition, line 17 shows how Polyspace software prompts the user to investigate further (by means of an orange check) when all cells have not been initialized.

Note Associated to each message which concerns a NIV check, Polyspace software gives the type of the variable like the following examples: `(type: volatile int32)`, `(type:  int 16)`, `(type:  unsigned int 8)`, etc.

OVFL – Scalar and Float Overflow

In this section...

OVFL Checks

These are checks to establish whether arithmetic expressions overflow or underflow. This is a scalar check with integer type and float check for floating point expression. Consider the following example.

```
1 #include <float.h>
2 extern int random_int(void);
3
4 void main(void)
5 {
6  int i = 1;
7  float fvalue = FLT_MAX;
8
9  i = i << 30; // i = 2**30
10  if (random_int())
11   i = 2 * (i - 1) + 2; // OVFL ERROR: 2**31 is an overflow
value for int32
12  if (random_int())
13   fvalue = 2 * fvalue + 1.0; // OVFL ERROR: float variable is
overflow
14 }
```

On a 32 bit architecture platform, the maximum integer value is *2**31-1*, thus *2**31*will raise an overflow.

In the same manner, if *fvalue* represents the biggest float its double cannot be represented with same type and raises an overflow.

What is the Biggest Float in C?

There are occasions when it is important to understand when overflow may occur on a float value approaching its maximum value. Consider the following example.

```
void main(void)
{
 float x, y;
 x = 3.40282347e+38f; // is green
 y = (float) 3.40282347e+38; // OVFL red
}
```

There is a red error on the second assignment, but not the first. The real "biggest" value for a float is: `340282346638528859811704183484516925440.0 - MAXFLOAT -.`

Now, rounding is not the same when casting a constant to a float, or a constant to a double:

- floats are rounded to the nearest lower value;
- doubles are rounded to the nearest higher value;
- 3.40282347e+38 is strictly bigger than `340282346638528859811704183484516925440` (named `MAXFLOAT`).
- In the case of the second assignment, the value is cast to a double first - by your compiler, using a temporary variable D1 -, then into a float - another temporary variable -, because of the cast. Float value is greater than `MAXFLOAT`, so the check is red.
- In the case of the first assignment, 3.40282347e+38f is directly cast into a float, which is less than `MAXFLOAT`

The solution to this problem is to use the "f" suffix to specify the variable directly as a float, rather than casting.

What is the Type of Constants/What is a Constant Overflow?

Consider the following example, which would cause an overflow.

```
int x = 0xFFFF; /* OVFL */
```

The table that follows shows three types of constants with corresponding lists of data types. The data type given to a constant is the first data type from the corresponding list that can accommodate the constant value. (See "Predefined Target Processor Specifications" for information about the size of a type depending on the target.)

Decimal	`int`, `long`, `unsigned long`
Hexadecimal	`int`, `unsigned int`, `long`, `unsigned long`
Float	`double`

For example (assuming 16-bit target):

5.8	`double`
6	`int`
65536	`long`
0x6	`int`
0xFFFF	`unsigned int`
5.8F	`float`
65536U	`unsigned int`

The option "Ignore overflowing computations on constants" on page 1-59 allows you to bypass this limitation and consider the line

```
int x = 0xFFFF; /* OVFL */
```

as

```
int x = -1;
```

instead of 65535, which does not fit into a 16-bit integer (–32768 to 32767).

Left shift overflow on signed variables: OVFL

Overflows can be also be encountered in the case of left shifts on signed variables. In the following example, the higher order bit of *0x41021011* (hexadecimal value of *1090654225*) has been lost, highlighting an overflow (integer promotion).

```
1
2 void main(void)
3 {
4  int i;
5
6  i = 1090654225 << 1; // OVFL ERROR: on left shift range
7 }
```

Float Underflow Versus Values Near Zero: OVFL

The definition of the word "underflow" differs between the ANSI standard and the ANSI/IEEE 754-1985 standard. According to the former definition, underflow occurs when a number is sufficiently negative for its type not to be capable of representing it. According to the latter, underflow describes the erroneous representation of a value close to zero due to the limits of its representation.

Polyspace verifications apply the former definition. The latter definition does not impose the raising of an exception as a result of an underflow. By default, processors supporting this standard permit the deactivation of such exceptions.

Consider the following example.

```
2 #define FLT_MAX 3.40282347e+38F // maximum representable
float found in <float.h>
3 #define FLT_MIN 1.17549435e-38F // minimum normalised
float found in <float.h>
4
5 void main(void)
6 {
7  float zer_float = FLT_MIN;
8  float min_float = -(FLT_MAX);
9
10  zer_float = zer_float * zer_float; // No check overflow
```

```
near zero
11  min_float = min_float * min_float; // OVFL ERROR:
underflow checked by verifier
12
13 }
```

IRV – Initialized Return Value

This is a check to establish whether a function returns an initialized value. Consider the following example.

```
1
2 extern int random_int(void);
3
4 int reply(int msg)
5 {
6  int rep = 0;
7  if (msg > 0) return rep;
8 }
9
10 void main(void)
11 {
12  int ans;
13
14  if (random_int())
15   ans = reply(1); // IRV verified: function returns an
initialised value
16  else if (random_int())
17   ans = reply(0); // IRV ERROR: function does not return an
initialised value
18  else
19   reply(0); // No IRV checks because the return value
is not used
20
21 }
22
23
```

Variables are often initialized using the return value of functions. However, in the above example the return value is not initialized for all input parameter values. In this case, the target variable will not be always be properly initialized with a valid return value.

SHF – Shift Operations

In this section...

Shift Amount in 0..31 (0..63): SHF

This is a check to establish whether a shift (left or right) is bigger than the size of the integral type operated upon (int or long int). The range of allowed shift depends on the target processor: 16 bits on *c-167*, 32 bits on *i386* for int, etc. Consider the following example.

```
1 extern int random_value(void);
2
3 void main(void)
4 {
5  volatile int x;
6  int k, l = 1024; // 32 bits on i386
7  unsigned int v, u = 1024;
8
9   if (x) k = l << 16;
10  if (x) k = l >> 16;
11
12  if (x) k = l << 32; // SHF ERROR
13  if (x) k = l >> 32; // SHF ERROR
14
15  if (x) v = u >> 32; // SHF ERROR
16  if (x) k = u << 32; // SHF ERROR
17
18 }
```

In this example, it is shown that the shift amount is greater than the integer size.

Left Operand of Left Shift is Negative: SHF

This is a check to establish whether the operand of a left shift is a signed number. Consider the following example.

```
1
2
3 void main(void)
4 {
5  int x = -200;
6  int y;
7
8  y = x << 1; // SHF ERROR: left operand must be positive
9
10 }
```

As an aside, note that the "Allow negative operand for left shifts" on page 1-60 option used at launching time instructs Polyspace software to permit explicitly signed numbers on shift operations. Using the option in the example above would see the red check at line 8 transformed in a green one. Similarly, if the verification had included the expression *-2 << 2* at line 9, then that line would have been given a green check and y would assume a values of *-8*.

IDP – Illegal Dereferenced Pointer

In this section...

Illegal Pointer Access to Variable or Structure Field: IDP

Illegal Pointer Access to Variable or Structure Field

This is a check to establish whether in the dereferencing of an expression of the form *ptr+i*, the variable/structure field initially pointed to by *ptr* is still the one accessed. See ANSI C standard ISO/IEC 9899 section 6.3.6.

Consider the following example.

```
1 int a;
2
3 struct {
4  int f1;
5  int f2;
6  int f3;
7 } S;
8
9 void main(void)
10 {
11  volatile int x;
12
13  if (x)
14   *(&a+1) = 2;
15  // IDP ERROR: &a +1 doesn't point to a any longer
16  if (x)
17   *(&S.f1 +1) = 2;
18  // IDP ERROR: you are not allowed to access f2 like this
```

```
19 }
```

According to the ANSI C standard, it is not permissible to access a variable (or a structure field) from a pointer to another variable. That is, *ptr+i* may only be dereferenced if *ptr+i* is the address of a subpart of the object pointed to by *ptr* (such as an element of the array pointed to by *ptr*, or a field of the structure pointed to by *ptr*).

For instance, the following code is correct because the length of the entity pointed to by *ptr_s* reflects the full structure length of *My_struct* (at line 11):

```
1 typedef struct {
2  int f1;
3  int f2;
4  int f3;
5 } My_Struct;
6
7 My_Struct s = {1,2,3};
8
9 int main(void)
10 {
11 My_Struct *ptr_s = &s;
12
13  // change to f2 field
14  *((int *)&s +1) = 2; // Correct evaluation
15
16  return 0;
17 }
```

Pointer Within Bounds: IDP

Check to establish whether a reference refers to a valid object (whether a dereference pointer is still within the bounds of the object it intended to point to).

Consider the following example.

```
1
2 #define TAILLE_TAB 1024
3 int tab[TAILLE_TAB];
4 int *p = tab;
```

```
5
6 void main(void)
7 {
8
9  int index;
10
11  for (index = 0; index < TAILLE_TAB ; index++, p++)
12  {
13   *p = 0;
14  }
15
16  *p = 1; // IDP ERROR: reference refers to an invalid object
17 }
```

In the example, the pointer *p* is initialized to point to the first element of the *tab* array at line 4. When the loop is exited, *p* points beyond the last element of the array.

Thus line 16 overwrites memory illegally.

Understanding Addressing

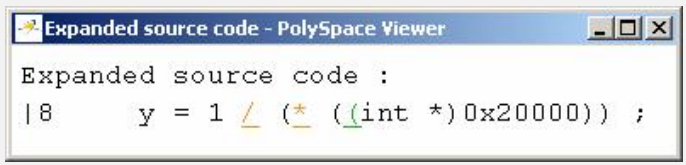
I Systematically Have an Orange Out of Bounds Access On My Hardware Register

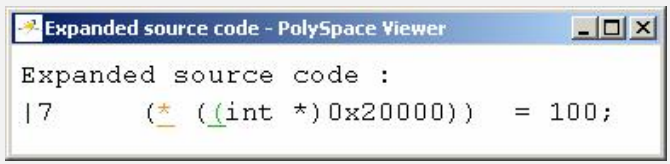
Many code verifications exhibit **orange** out of bound checks with respect to accesses to absolute addresses and/or hardware registers.

(Also refer to the discussion on Absolute Addressing)

Here is an example of what such code might look like:

```
#define X (* ((int *)0x20000))
X = 100;
y = 1 / X;  // ZDV check is orange because
     // X ~ [-2^31, 2^31-1] permanently.
     // The pointer out of bounds check is orange because 0x20000
     // may address anything of any length
     // NIV check is orange on X as a consequence
```






```
3 void main (void)
4 {
5 int y;
6
7 X = 100;
8 y = 1 / X;
9
10  }
```

```
int *p = (int *)0x20000;
*p = 100;
y = 1 / *p;  // ZDV check is orange because
     // *p ~ [-2^31, 2^31-1] permanently
     // The pointer out of bounds is orange because 0x20000
     // may address anything of any length
     // NIV check on *p is orange as a consequence
```

This can be addressed by defining registers as regular variables:

Replace	By
`#define X ....`	`int X;`
`int *p;`	`int _p;#define p (&_p)` **Note** Check that the chosen variable name (p in this example) doesn't already exist
`int *p;`	`volatile int _p;int *p = &_p;`

See "Volatile Variables" for a discussion of an approach which will help avoid the orange check on the pointer dereference, but retains the representation of a "full range" variable.

The NULL Pointer Case

Consider the NULL address:

```
#define NULL ((void *)0)
```

- It is illegal to dereference this NULL address;
- 0 **is not** treated as an absolute address.

Therefore, the following code produces a red (IDP) check:

```
*NULL = 100;  // IDP: pointer is outside its bounds
```

Assuming these declarations:-

```
int *p = 0x5;
volatile int y;
```

and these definitions:-

```
#define NULL ((void *) 0)
#define RAM_MAX ((int *)0xffffffff)
```

consider the code snippets below.

```
While (p != (void *)0x1)
 p--; // terminates
```

`0x1` is an absolute address, it can be reached and the loop terminates

```
for (p = NULL; p <= RAM_MAX; p++)
{
 *p = 0; // Red IDP: pointer is outside its bounds
}
```

At the first iteration of the loop `p` is a NULL pointer. Dereferencing a NULL pointer is forbidden.

```
While (p != NULL)
{
 p--;
 *p = 0; // Orange IDP: pointer may be outside its bounds
}
```

When `p` reaches the address `0x0`, there is an attempt to considered it as an absolute address

In effect, it is an attempt to dereference a NULL pointer - which is forbidden.

Note In this case, the **check** is **orange** because the execution of the code here is OK (green) until `0x0` is reached (red)

The best way to address this issue depends on the purpose of the function.

- Thanks to the default behavior of Polyspace software, it is easy to automatically stub a function whose purpose is to copy data from/to RAM or to compute a checksum on RAM.
- If a function is supposed to copy calibration data, it should also be stubbed automatically.
- If the purpose of a function is to map EEPROM data to global variables, then a manually written stub is required to assign initial values to the global variables.

Comparing Address

Polyspace software only deals with the information referred to by a pointer, and not the physical location of a variable. Consequently it does not compare addresses of variables, and makes no assumption regarding where they are located in memory.

Consider the following two examples of Polyspace verification behavior:

```
int a,b;
if (&a > &b) // condition can be true and/or false
{ } // both branches are reachable
else
{ } // both branches are reachable
```

and

```
int x,z;
void main(void)
{ int i;
 x = 12;
```

```
 for (i=1; i<= 0xffffffff; i++)
 {
  *((int *)i) = 0;
 }
 z = 1 / x;  // ZDV green check because Polyspace doesn't consider
     // any relationship between x and its address
}
```

“x” is aliased by no other variable. No pointer points to “x” in this example, so as far as the Polyspace verification is concerned, “x” remains constantly equal to 12.

Understanding Pointers

Pointers and Verification

Polyspace software does not analyze anything which would require the physical address of a variable to be taken into account.

- Consider two variables x and y. Polyspace verification will not make a meaningful comparison of “&x” (address of x) and “&y”
- So, the Boolean (&x < &y) can be true or false as far as Polyspace verification is concerned.

However, Polyspace verification does keep track of the pointers that point to a particular variable.

- So, if ptr points to X, *ptr and X will be synonyms.

Address Alignment: the bitfield Example

Structure size depends on bit alignment.

Consider the following example, where an attempt is made to map a character to a bitfield.

```
struct reg {
 unsigned int a: 5;
 unsigned int b: 3;
};
int main()
{
 volatile unsigned char c;
 struct reg *r;
 r = (struct reg *) &c;
 if (r-> a == 10)
  return 1;
 return 0;
}
```

Consider a 32 bit target architecture (so int are 32 bits, i.e. 4 bytes). The size of a bit field is the size of the type of its elements. In the example above, the elements in the bit field are unsigned int, hence the size is 4 bytes. Since this is greater than 1, the structure reg cannot be contained in the char c.

This can be solved by using the unsigned char type for the elements in the bit field. The size of the bit field is then 1 byte and there is therefore no red error.

```
struct reg {
 unsigned char a: 5;
 unsigned char b: 3;
};
int main()
{
 volatile unsigned char c;
 struct reg *r;
 r = (struct reg *) &c;
 if (r-> a == 10)
```

```
  return 1;
 return 0;
}
```

Note You must also use the option -allow-non-int-bitfield to implement this solution, since this is an extension to the ANSI standard.

How Does malloc Work for Polyspace Verification?

Polyspace verification models malloc, such that both the possible return values of a null pointer and the requested amount of memory are taken into account.

Consider the following example.

```
void main(void)
{
 char *p;
 char *q;
 p = malloc(120);
 q = p;
 *q = 'a'; // results in an orange dereference check
}
```

This code will avoid the orange dereference:

```
void main(void)
{
 char *p;
 char *q;
 p = malloc(120);
 q = p;
 if (p!= NULL)
  *q = 'a'; // results in a green dereference check
}
```

Data Mapping into a Structure

It often happens that structured data are read as a char array. Before manipulating them it might be desirable to map such data into a structure

that reflects their organization. In the following example an IDP warning (orange check) at line 22 suggests that the correctness of the code needs to be confirmed.

```
1
2
3 typedef struct
4 {
5  unsigned int MsgId;
6  union {
7   float fltv;
8   unsigned int intv;
9  } Msgbody;
10 } Message;
11
12 int random_int(void);
13 Message *get_msg(void);
14 void wait_idl(void);
15
16 void treatment_msg(char *msg)
17 {
18  Message *ptrMsg;
19
20  ptrMsg = (Message *)msg;
21  if (ptrMsg != NULL) {
22   if (ptrMsg->MsgId) { // IDP Warning: reference may not
refer to a valid object
23    // ...
24   }
25  }
26 }
27
28 int main (void) {
29
30  Message *msg;
31
32  while(random_int()) {
33   msg = get_msg();
34   if (msg) treatment_msg((char *)msg);
35   wait_idl();
```

```
36  }
37  return 0;
38 }
```

Mapping of a small structure into a bigger one

For example, suppose that *p* is a pointer to an object of type *t_struct* and it is initialized to point to an object of type *t_struct_bis*.

Now suppose that the size of *t_struct_bis* is less than the size of *t_struct*. Under these circumstances, it would be illegal to dereference *p* because it would be possible to access memory outside of *t_struct_bis*.

Consider the following example.

```
1 #include <malloc.h>
2
3 typedef struct {
4  int a;
5 union {
6   char c;
7   float f;
8  } b;
9 } t_struct;
10
11 void main(void)
12 {
13  t_struct *p;
14
15  // optimize memory usage
16  p = (t_struct *)malloc(sizeof(int)+sizeof(char));
17
18  p->a = 1; // IDP ERROR: not allowed to deference p
19
20 }
```

Partially allocated pointer (-size-in-bytes)

According to the ANSI standard, the whole of a structure must be populated for that structure to be valid. In this case, the pointer is said to be fully allocated. A pointer is said to be partly allocated when only the first part of a

structure is populated. In some development environments, that approach is tolerated despite the ANSI stance.

By default, Polyspace verification strictly conforms to the standard and checks for adherence to it. A more tolerant approach can be specified by using the -size-in-bytes option. So, depending on the -size-in-bytes option, when a partially allocated pointer is encountered during a Polyspace verification, the first elements of the allocated object may or may not be considered as valid.

First consider the following example. (A second example follows it to illustrate how this might apply to pointer arithmetic within a structure)

```
1  typedef struct _little { int a; int b; } LITTLE;
2  typedef struct _big { int a; int b; int c; } BIG;
3
4 int main(void)
5  {
6   BIG *p = malloc(sizeof(LITTLE));
7   volatile int y;
```

With -size-in-bytes option

```
9   if (p==((void *)0)) return 0;
10   if(y) { p->a = 0; } // green
11   if(y) { p->b = 0; } // green
12   if(y) { p->c = 0; } // red
}
```

Default launching option

```
9   if(y) { p->a = 0 ; } // red
10   if(y) { p->b = 0 ; } // red
11   if(y) { p->c = 0 ; } // red
12
13   if (p==((void *)0))
14     return 0;
15   else
16     return 1; // dead code
17   return 1;
18  }
```

With the standard launching option, a pointer that has not been allocated to a complete structure is considered invalid, or NULL (as shown in the dead code).

Pointer to a structure field

According to the ANSI C standard, pointer arithmetic is to be independent of the size of the object (structure or array) to which the pointer points. By default, Polyspace verification strictly conforms to the standard and checks for adherence to it.

In some development environments an approach that does not recognize that requirement is tolerated, despite the ANSI stance. Under those circumstances, results are likely to include **red** pointer out of bounds **checks** unexpectedly.

A more tolerant approach can be specified at launch time. Consider the following examples.

```
char *p; // the size of the object pointed to is unknown,
// but arithmetic on this pointer is well defined.
// p = p + 5; will increment the location pointed to by
5 bytes (if the
size of a char is 1 byte)
int x;  // assuming that an int is 4 bytes
p = &x; *p = 0; // the first byte of x
p++; *p = 0;  // the second byte of x
p++; *p = 0;  // the third byte of x
p++; *p = 0;  // the fourth byte of x
p++; *p = 0;  // an out of bound access
```

For structures, the same behavior can be applied.

```
struct { int a; int b; } x;
char *p = &x.a; // the pointed object is not the structure
but the field
*p = 0; // it is the first byte of x.a
p++; *p = 0;  // it is the second byte of x.a
p++; *p = 0;  // it is the third byte of x.a
p++; *p = 0;  // it is the fourth byte of x.a
p++; *p = 0;  // here is an out of bound access because
we are out of the field
```

If you wish to tolerate an approach which allows a pointer to go from one field to another, you can do so by using the -size-in-bytes option **together with** the -allow-ptr-arith-on-struct option . When a pointer points to a field in a structure, you will then be allowed to access other fields from this pointer. Note that as a consequence, any other "out of bound" accesses in the code will be ignored.

An alternative solution is to make your variable point to the structure rather than to the field, as follows:

```
struct { int a; int b; } x;
char *p = &x;  // the pointed object is the structure
*p = 0;    // we are modifying x.a (first byte)
p++; *p = 0;  // we are modifying x.a (second byte)
p++; *p = 0;   // we are modifying x.a (third byte)
p++; *p = 0;  // we are modifying x.a (fourth byte)
p++; *p = 0;  // we are modifying x.b (fifth byte)
```

A further alternative is to follow the ANSI C recommendation to use the “offsetof()” function, which jumps to the corresponding offset within the structure:-

```
#include <stddef.h>
typedef struct _m { int a; int b; } S;
S x;
char *p = (char *) &x + offsetof(S,b); // points to field b
```

I have a red when reading a field of one structure

Consider the following example.

```
5 typedef struct {
6  unsigned char c1;
7  unsigned char c2;
8 } my_struct;
9
10 int main(void)
11 {
12  my_struct v;
13  unsigned short x=0,y=0;
14
```

```
15  v.c1=9;
16  v.c2=15;
17  x = *((unsigned short *)&v.c1);
```

Just like the example in "Pointer to a structure field" on page 3-30, the object pointed to is the field in the structure, not the structure itself. Therefore, it is only possible to navigate inside this field. A short variable occupies more memory than a char, so it is a red pointer out of bounds.

This can be addressed by replacing

```
x = * ((unsigned short *) &v.c1);
```

with

```
y = (v.c1 << sizeof(v.c2)*8 ) | v.c2;
```

This solution also ensures that the code is no longer target dependent.

COR – Correctness Condition

In this section...

Array Conversion Must Not Extend Range: COR

This is a check to establish whether a small array is mapped onto a bigger one through a pointer cast. Consider the following example.

```
1
2  typedef int Big[100];
3  typedef int Small[10];
4  typedef short EquivBig[200];
5
6  Small smalltab;
7  Big bigtab;
8
9  void main(void)
10  {
11    volatile int random;
12
13   Big * ptr_big = &bigtab;
14   Small * ptr_small = &smalltab;
15
16   if (random) {
17     Big *new_ptr_big = (Big*)ptr_small;
18     // COR ERROR: array conversion must not extend range
19   }
20   if (random) {
21     EquivBig *ptr_equivbig = (EquivBig*)ptr_big;
22     Small *ptr_new_small = (Small*)ptr_big;
23      // Conversion verified
22   }
23  }
```

In this example, a pointer is initialized to the `Big` array with the address of the `Small` array. This is not legal since it would be possible to dereference this

pointer outside the `Small` array. Line 21 shows that the mapping of arrays of the same size but with different prototypes is acceptable.

Function Pointer Does Not Point to a Valid Function: COR

This is a check to establish whether a function pointer points to a valid function or a function with a valid prototype. The software checks, for example, whether:

- The pointer points to a function.
- Each argument passed to a function matches the corresponding argument in the function prototype.
- The number of arguments passed to a function matches the number of arguments in the function prototype.
- The return type passed to a function pointer matches the return type declared in the function prototype.

Pointer Does Not Point To Any Function

Consider the following example.

```
1
2  typedef void (*CallBack)(float *data);
3  typedef struct {
4   int a;
5   char name[20];
6   CallBack func;
7  } funcS;
8
9  funcS myvar;
10  CallBack cb;
11
12  void My_function(float *data)
13  {
14   *data = 2;
15  }
16
17  static void Struct_not_init_art_ptr_def(void)
```

```
18  {
19   cb = &My_function;
20  }
21
22  int main(void)
23  {
24   float fval=0;
25
26    cb = myvar.func;
27        cb(&fval);
28       // Red COR: function pointer does not point to a valid function
29  return 0;
30  }
```

In this example, `func` has a prototype that conforms to the declaration of `CallBack`. Therefore, `func` is initialized to point to the `NULL` function through the global declaration of `funcS`. So a `NULL` pointer is assigned to the `cb` local variable.

Consider a second example.

```
1
2 #define MAX_MEMSEG 32764
3 typedef void (*ptrFunc)(int memseg);
4 ptrFunc initFlash = (ptrFunc)(0x003c);
5
6 void main(void)
7 {
8  int i;
9
10  for (i = 0 ; i < MAX_MEMSEG; i++)
11  // In Source view, for statement has red, dashed underlining to highlight issue with initFla
12  {
13   initFlash(i);
14     // Red COR: function pointer does not point to a valid function
15  }
16 }
```

Polyspace verification does not take the memory mapping of programs into account, and cannot determine whether `0x003` is the address of a function code segment or a data segment. Therefore, the verification generates a red check.

Function Arguments Do Not Match Prototype Arguments

Consider the flowing example.

```
1
2 typedef struct {
3  float r;
4  float i;
5 } complex;
6
7 typedef int (*t_func)(complex*);
8
9 int foo_type(int *x)
10 {
11  if (*x%2 == 0) return 0;
12  else return 1;
13 }
14
15 void main(void)
16 {
17  t_func ptr_func;
18  int j,i = 0;
19
20  ptr_func = foo_type;
21  j = ptr_func(&i);
22  // Red COR: function pointer does not point to a valid function
23 }
24
```

In this example, `ptr_func` is a pointer to a function that expects the input argument to be a pointer to a `complex` structure. However, the input argument is a pointer to an `int`.

Wrong Number of Arguments

Consider the following example.

```
1
2 typedef int (*t_func_2)(int);
3 typedef int (*t_func_2b)(int,int);
4
5 int foo_nb(int x)
6 {
7  if (x%2 == 0)
8   return 0;
9  else
10   return 1;
11 }
12
13
14 void main(void)
15 {
16  t_func_2b ptr_func;
17  int i = 0;
18
19  ptr_func = (t_func_2b)foo_nb;
20  i = ptr_func(1,2);
21  // Red COR: function pointer does not point to a valid function
22 }
23
```

In this example, `ptr_func` is a pointer to a function that takes two arguments. However, the function pointer has been initialized to point to a function that takes only one argument.

Wrong Return Type

Consider the following example.

```
1
2 typedef int (*t_func_2)(int);
3 typedef double (*t_func_2b)(int);
4
5 int foo_nb(int x)
6 {
7  if (x%2 == 0)
8   return 0;
9  else
```

```
10   return 1;
11 }
12
13
14 void main(void)
15 {
16  t_func_2b ptr_func;
17  int i = 0;
18
19  ptr_func = (t_func_2b)foo_nb;
20  i = ptr_func(2);
21  // Red COR ERROR: function pointer does not point to a valid function
22 }
23
```

In this example, `ptr_func` is a pointer to a function that returns a `double`. However, the function pointer is initialized to point to a function that returns an `int`.

Consider return types for arithmetic functions. You might be able use arithmetic functions in your code without including the `<math.h>` file because your compiler could associate integral return types with implicit functions. However, Polyspace software has built-in arithmetic functions. If you do not include `<math.h>` for code with the relevant arithmetic functions, you might see results that are not consistent.

The following example does not include `<math.h>` and verification generates a red COR check for the `cos` function.

```
1
2 int main(void) {
3
4 double x;
5 x = cos(2*3.1415);
6 // Red COR: function pointer does not point to a valid function
7 }
```

Including `<math.h>` resolves the issue.

```
1 #include <math.h>
2 int main(void) {
```

```
3
4 double x;
5 x = cos (2*3.1415);
6 }
```

NIP – Non-Initialized Pointer

Check to establish whether a reference is initialized before being dereferenced. Consider the following example.

```
2
3 void main(void)
4 {
5  int* p;
6  *p = 0; // NIP ERROR: reference is not initialized
7 }
```

As *p* is not initialized, an undefined memory cell would be overwritten at line 6 (**p = 0*) (also leading to the unreachable gray check on "*").

ASRT – User Assertion

This is a check to establish whether a user assertion is valid. If the assumption implied by an assertion is invalid, then the standard behavior of the assert macro is to abort the program. Polyspace verification therefore considers a failed assertion to be a runtime error. Consider the following example.

```
1 #include <assert.h>
2
3 typedef enum
4 {
5  monday=1, tuesday,
6  wensday, thursday,
7  friday, saturday,
8  sunday
9 } dayofweek ;
10
11 // stubbed function
12 dayofweek random_day(void);
13 int random_value(void);
14
15 void main(void)
16 {
17  unsigned int var_flip;
18  unsigned int flip_flop;
19  dayofweek curDay;
20  unsigned int constant = 1;
21
22  if (random_value()) flip_flop=1; else flip_flop=0;
 // flip_flop can randomly be 1 or 0
23  var_flip = (constant | random_value());
// var_flip is always > 0
24
25  if(random_value()) {
26   assert(flip_flop==0 || flip_flop==1); // User Assertion is
verified
27   assert(var_flip>0); // User Assertion is
verified
28   assert(var_flip==0); // ASRT ERROR: Failure User
Assert
```

```
29  }
30
31  if (random_value()) {
32   curDay = random_day(); // Random day of the week
33   assert( curDay > thursday); // ASRT Warning: User
assertion may fails
34   assert( curDay > thursday); // User assertion is
verified
35   assert( curDay <= thursday); // ASRT ERROR: Failure User
Assertion
36  }
37 }
```

In the *main*, the *assert* function is used in two different manners:

- To establish whether the values *flip_flop* and *var_flip* in the program are inside the domain which the program is designed to handle. If the values were outside the range implied by the *assert* (see line 28), then the program would not be able to run properly. Thus they are flagged as runtime errors.
- To redefine the range of variables as shown at line 34 where *curDay* is restricted to just a few days. Indeed, Polyspace verification makes the assumption that if the program is executed without a runtime error at line 33, *curDay* can only have a value greater than *thursday* after this line.

NTC – Non-Termination of Call

In this section...

Non-Termination of Calls and Loops: Informative Checks

NTC and NTL are informative red (or orange) checks.

- They are the only red checks which can be filtered out as shown below
- They don't stop the verification
- As for other red checks, code found after them are gray (unreachable)
- These checks may only be red. There are no "orange" NTL or NTC checks.
- They can reveal a bug, or can simply just be informative

NTL	In a Non Terminating Loop, the break condition is never met. Here are some examples.
	`while(1)  { function_call(); } // informative NTL`
	The following may reveal a bug:
	`while(x>=0) {x++; } // where x is an unsigned int.`
	The following red NTL reveals a bug in the array access, flagged in orange:
	`for(i=0; i<=10; i++) my_array[i]  = 10; // where int my_array[10];  applies.`
	In the following example, the first iteration of the loop is red, and therefore it is flagged as an NTL. The "i++" will be gray, because the first iteration crashed.

	`ptr = NULL; for(i=0; i<=100;  i++) *ptr=0; //`
NTC	Suppose that a function calls f(), and that function call is flagged with a red NTC check. There could be five distinct explanations: • "f" contains a red error; • "f" contains an NTL ; • "f" contains an NTC; • "f" contains an orange which is context dependant; that is, it is either red or green. For this particular call, it makes the function "f" crash. • "f" is a mathematical function, such as sqrt, acos which has always an invalid input parameter Remember, additional information can be found when clicking on the NTC

Note A sqrt check is only colored if the input parameter is **never** valid. For instance, if the variable x may take any value between -5 and 5, then sqrt(x) has no color.

The list of constraints which cannot be satisfied (found by clicking on the NTC check) represents the variables that cause the red error inside the function. The (potentially) long list of variables can help to understand the cause of the red NTC, as it shows each condition causing the NTC

- where the variable has a given value; and
- where the variable is not initialized. (Perhaps the variable is initialized outside the set of files under verification?).

If a function is identified which is not expected to terminate (such as a loop or an exit procedure) then the -known-NTC function is an option. You will find all the NTCs and their consequences in the k-NTC facility in the Viewer, allowing you to filter them.

Non Termination of a Call: NTC

This is a 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 caller.

```
1
2
3 void foo(int x)
4 {
5  int y;
6  y = 1 / x; // Warning ZDV: its depends of the context
7  while(1) { // NTL ERROR: loop never terminates
8   if ( y != x) {
9    y = 1 / (y-x);
10   }
11  }
12 }
13
14 void main(void) {
15  volatile int _x;
16
17  if (_x)
18   foo(0); // NTC ERROR: Zero DiVision (ZDV) in foo
19  if (_x)
20   foo(2); // NTC ERROR: Non Termination Loop (NTL) in foo
21
22 }
23
```

In this example, the function *foo* is called twice in *main* and neither of these 2 calls ever terminates.

1. The first never returns because a division by zero occurs at line 6 (bad argument value),
2. The second never terminates because of an infinite loop (red NTL) at line 7.

Also with reference to the example and as an aside, note that by using the -context-sensitivity "foo" option at launch time it would be possible for

Polyspace verification to show explicitly that a ZDV error comes from the **first** call of *foo* in *main*.

An NTC check can only be red or uncolored, unless you use the -context-sensitivity option. If you use the -context-sensitivity option, NTC checks can also be orange.

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

Arithmetic Expressions: NTC

This is a check to establish whether standard arithmetic functions are used with valid arguments, as defined in the following:

- Argument of *sqrt* must be positive (ISO®/IEC 9899 section 7.5.5.2)
- Argument of *tan* must be different from *pi/2* modulo *pi* (ISO/IEC 9899 section 7.5.2.7)
- Argument of *log* must be strictly positive (ISO/IEC 9899 section 7.5.4.4)
- Argument of a*cos* and a*sin* must be within [-1..1] (ISO/IEC 9899 sections 7.5.2.1 and 7.5.2.2)
- Argument of *exp* must be less than or equal to 709 (ISO/IEC 9899 section 7.5.4.1)
- Argument of *atanh* must be within]-1..1[(ISO/IEC 9899 section 7.12.5.3)
- Argument of *acosh* must be greater or equal to 1 (ISO/IEC 9899 section 7.12.5.1)

A domain error (such that *errno* returns *EDOM*) occurs if an input argument is outside the domain over which the mathematical function is defined. A range error occurs (such that *errno* returns *ERANGE*) if the result cannot be represented as a double value. In the latter case, the function returns *0* if the result is too small, or *HUGE_VAL* with the sign if it is too big.

Consider the following example

```
1
2 #include <math.h>
```

```
3 #include <assert.h>
4
5 extern int random_int(void);
6
7 int main(void)
8 {
9
10  volatile double dbl_random;
11  const double dbl_one = 1.0;
12  const double dbl_mone = -1.0;
13
14  double sp      = dbl_random;
15  double p      = dbl_random;
16  double sn      = dbl_random;
17  double n      = dbl_random;
18  double no_trig_val_neg = dbl_random;
19  double no_trig_val_pos = dbl_random;
20  double pun      = dbl_random;
21  double res;
22
23  // assert is used here to redefine range values of variables
24  assert(sp > 0.0);
25  assert(p >= 0.0);
26  assert(sn < 0.0);
27  assert(n <= 0.0);
28  assert(pun < 1.0);
29  assert(no_trig_val_neg < -1.0); assert(no_trig_val_pos > 1.0);
30
31  if (random_int()) res = sqrt(sn);       // NTC ERROR:
need argument positive
32  if (random_int()) res = asin(no_trig_val_neg);  // NTC ERROR:
need argument in range [-1..1]
33  if (random_int()) res = asin(no_trig_val_pos);  // NTC ERROR:
need argument in range [-1..1]
34  if (random_int()) res = acos(no_trig_val_pos);  // NTC ERROR:
need argument in range [-1..1]
35  if (random_int()) res = acos(no_trig_val_neg);  // NTC ERROR:
need argument in range [-1..1]
36  if (random_int()) res = tan(1.5707963267948966); // NTC ERROR:
need argument in range ]-pi/2..pi/2[
```

```
37  if (random_int()) res = log(n);       // NTC ERROR:
need argument strictly positive
38  if (random_int()) res = exp(710);       // NTC ERROR:
need argument less or equal to 709
39
40  // No information about asin or acos because of random value
41  if (random_int()) {
42   res = asin(dbl_random);
43   res = acos(dbl_random);
44  }
45
46  // hyperbolic functions are available in the float range
47  if (random_int()) {
48   res = cosh(710);
49   res = cosh(10.0);
50   assert (res < 1.0);
51  }
52  if (random_int()) res = sinh(710);
53  if (random_int()) {
54   res = tanh(1.0);
55   assert (res > -1.0 && res < 1.0);
56  }
57
58  // inverted hyperbolic functions
59  if (random_int()) res = acosh(pun);      // NTC ERROR:
Need argument >= 1
60  else res = acosh(1.0);
61  if (random_int()) res = atanh(no_trig_val_neg);  // NTC ERROR:
Need argument in ]-1..1[
62  if (random_int()) res = atanh(no_trig_val_pos);  // NTC ERROR:
Need argument in ]-1..1[
63  if (random_int()) res = atanh(dbl_mone);    // NTC ERROR:
Need argument in ]-1..1[
64  if (random_int()) res = atanh(dbl_one);    // NTC ERROR:
Need argument in ]-1..1[
65
66  return 0;
67 }
68
```

sqrt, *tan*, *asin*, *acos*, *exp* and *log* errors are derived directly from the mathematical definition of functions. Polyspace verification highlights any definite problems by means of an NTC to show that this is where execution would terminate. No NTC information is delivered when Polyspace cannot determine the exact value of the argument, (for *asin* and *acos* at lines 42 and 43). No range restriction is currently made for hyperbolic functions.

Caution Due to a lack of precision in some areas, Polyspace verification is not always able to indicate a red NTC check on mathematical functions even where a problem exists. In the following example involving a *sqrt* function, neither an orange nor a red check is shown on line16 even though the variable *val2* is negative.

By default it is important to consider each call to any mathematical functions as though it had been highlighted by an **orange check**, and could therefore lead to a runtime error.

```
1
2 #include <math.h>
3
4 extern int random_int(void);
5
6 int main(void)
7 {
8
9  double val1, val2;
10
11  int i;
12  val2 = 5.0;
13  for (i = 0 ; i < 10 ; i++) {
14   val2 = val2 - 1.0;
15  }
16  val1 = sqrt(val2); // No check on sqrt
17  return ((int)val1);
18 }
19
```

K_NTC – Known Non-Termination of Call

By using the -known-NTC option with a specified function in a verification, it is possible to change a "NTC – Non-Termination of Call" on page 3-43 check to a k-NTC check. Like NTC checks, k-NTC checks are propagated to their callers. In the Results Manager perspective, you can filter out known functions that do not terminate by applying the K_NTC filter.

Consider the following example, supposing that **-know-NTC "SysHalt"** option has been applied in a verification.

```
1
2 /* external get data function */
3 extern int get_data(int *ptr,void *data);
4 extern int printf (const char *, ...);
5
6 // known NTC function
7 void SysHalt(int value)
8 {
9  printf("Halt value %d",value);
10  while (1) ; // NTL ERROR: Loop Never Terminate
11 }
12
13 #define OK 1
14 int main(void)
15 {
16  int data, *ptr = NULL;
17  int status = OK;
18
19  // get next store
20  status = get_data(ptr,(void *)&data);
21  if (status != OK)
22   SysHalt(status); // k-NTC check: Call never
terminate
23
24  return(0);
25 }
```

In the example, the relevant NTC check is converted to a k-NTC one.

NTL – Non-Termination of Loop

In this section...

Non Termination of Loop: NTL

This is a check to establish whether a loop (`for`, `do-while` or `while`) terminates.

An `NTL` check can only be **red** or uncolored, unless you use the `-context-sensitivity` option. If you use the `-context-sensitivity` option, `NTL` checks can also be orange.

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

Tooltips for NTL Checks

Tooltips provide range information in the viewer, including the number of iterations for loops.

There are 2 possible situations:

- **Loops that terminate** – A tooltip gives the number of iterations of the loop. For example, `for (i=0; i<10;` i++), a tooltip on the `for` keyword says `Number of iteration(s): 10`.
- **Non–terminating loops** — The NTL check contains information about the maximum number of iterations that can be done. This number is an overset of the real number of iterations (which may be lower).

  For example:

  - **Failure at a given iteration**, `for (i=0; i<10; i++) y = 2 / (i - 5);` — The NTL check on the `for` keyword says: `Number of iteration(s): 6`

This means that the loop fails at the 6th iteration, which can help you find the orange check that contains the failure.

- **Infinite loop** `x = 0; while (x >= 0) y = 2;` — The NTL check on the `for` keyword says: `Number of iteration(s):  0..?`

  This means that the loop has an unknown number of iterations (up to an infinite number). It does not mean that the loop *is* an infinite loop, but that it *may* be an infinite loop. You would also get `0..?` on the loop `while (1) { if (random) break; }`.

NTL Check Examples

The following examples show conditions leading to an NTL check.

NTL Example 1:

Consider the following example:

```
1
2 // Function prototypes
3 void send_data(double data);
4 void update_alpha(double *a);
5
6 void main(void)
7 {
8  volatile double _acq;
9  double acq, filtered_acq, alpha;
10
11  // Init
12  filtered_acq = 0.0;
13  alpha = 0.85;
14
15  while (1) { //NTL ERROR: Non Termination Loop
16   // Acquisition
17   acq = _acq;
18   // Treatment
19   filtered_acq = acq + (1.0 - alpha) * filtered_acq;
20   // Action
21   send_data(filtered_acq);
22   update_alpha(&alpha);
23  }
```

```
24 }
```

In the example, the continuation condition is always true and the loop will never exit. Polyspace verification will raise an error in trivial examples such as this, and in much more complex circumstances.

NTL Example 2:

Consider this second verification. When an error is found inside a `for`, `do-while`, or `while` loop, Polyspace will not continue to propagate it.

```
1
2  void main(void)
3  {
4  int i;
5  double twentyFloat[20];
6
7  for (i = 0; i <= 20; i++) {  // NTL ERROR: propagation of
OBAI ERROR
8   twentyFloat[i] = 0.0;   // OBAI Warning: 20 verification
with i in [0,19] and one ERROR with i = 20
9  }
10  }
```

At line 8 in this example, the **red** OBAI related to the **21st** execution of the loop has yielded the **orange check**. The 20 first executions would be no problem, so this **orange** warning represents a combination of **red** and **green** checks.

NTL Example 3:

In the following example, there is a red `NTL` on the `for`.

```
1 int tab[4];
2
3 int g(int i) {
4  tab[i] = i;       // Orange OBAI
5  return 1;
6 }
7
```

```
8 void f(void)
9 {
10  int i;
11  int x;
12
13  for (i=0; i <5; i++) {   // Red NTL
14     x = g(i);
15  }
16 }
```

In this example, the `for` loop is well-bounded, but there is an `NTL` check on it even though the body is green.

The cause of the `NTL` is located in the function `g()`. There is a contextual orange `OBAI` in this function since the array is only 4 `ints` big. The first call to `g()` will be OK, but the last one will lead to an `OBAI`. There is no `NTC` on `g()` because there are valid calls to this function. And NTC are always red, or they are not present.

NTL Example 4:

In the following example, verification flags an `NTL` on the `while`.

```
1
2 int random(void);
3 unsigned int x;
4 void foo(void)
5 {
6  x = random();
7  while (x!=0) {    // red NTL
 8  }
 9 x--;
10 }
```

If `x=0`, the `while` loop is not executed and the program jumps to `x--;`. If `x` is not `0`, the `while` loop is executed, and `x` is not modified in the loop, therefore the loop never terminates.

Note An NTL check can only be **red** or uncolored, unless you use the -context-sensitivity option. If you use the -context-sensitivity option, NTL checks can also be orange.

NTL Example 5:

In the following example, there is a red `NTL` on the `while`.

```
1
2 extern in var;
3 unsigned char getstatus(void)
4 {
5  return (!(var == 0));
6 }
7
8 void func(void)
9 {
10  while(getstatus()) {  // red NTL: if we enter the loop we will never ex
11   /* do something without change value of var */
12  }
13  assert(1); // green ASRT
14 }
15 int main(void) { func(); return 0;}
```

In this example, we have a red because:

- At the first evaluation, `getstatus()` returns `TRUE` or `FALSE`. If it returns `FALSE`, we do not enter in the `while()` loop and jump to `assert()` line
- As soon as we enter in the `while()` loop `var` value does not change anymore and so `getstatus()` continues to return same value: `1`.

NTL Example 6:

In the following example, verification uses the options:

- `-scalar-overflows-checks signed_and_unsigned`
- `-scalar-overflows-behavior truncate-on-error`

There is a red NTL on the while.

```
1 extern unsigned char ucfullrange(void);
2
3 void foo(void)
4 {
5  unsigned char a = ucfullrange();
6
7  while (a--) {     // red NTL
8   assert(1);
9  }
10  assert(1);
11 }
```

In this example, the variable becomes 0 at the end of the loop, the loop terminates and the variable is decremented one more time.

With the two options specified above, since the variable is unsigned, a wrap-around occurs leading to an unsigned overflow. The verification reports this as an orange OVFL on the "a--", and propagates it to the while loop with the message:
"The Loop is infinite or contains a run-time error."

In this case, the loop contains a run-time error, causing the red NTL.

STD_LIB – Standard Library Function Call

This is a check to determine whether the arguments of a call to a function from the C standard library are valid.

Consider the following example with an invalid argument in standard library function call.

```
1 #include <assert.h>
2 #include <string.h>
3 volatile int rd;
4 const char *str = "test";
5 char gbuffer[10] = "is ";
6 int main(void)
7 {
8   if ( rd ) strcat((char*)0, str);
9   if ( rd ) strcat((char*)0x12345678, str);
10  if ( rd ) strcat(gbuffer, str);
11  return 1;
12 }
```

There are three calls to the standard library function `strcat`. For each call, the second argument `str` is a valid string. However, the validity of the first argument varies:

- In the first call, `(char*)0` does not point to sufficient allocated memory. The software generates a red `STD_LIB` check for `strcat`.
- In the second call, `(char*)0x12345678` *may* point to sufficient allocated memory. The software generates an orange `STD_LIB` check for `strcat`.
- In the third call, there is sufficient memory in `gbuffer` for the concatenated string `"is test"`. As both arguments are valid, the software generates a green `STD_LIB` check for `strcat`.

ABS_ADDR – Absolute Address

The software generates an orange ABS_ADDR check when an absolute address is assigned to a pointer. The check is colored orange because the software has no information about the absolute address and cannot verify, for example, the address, availability of memory, and initialization of memory.

The software permits memory access to the absolute address after generating the orange ABS_ADDR check for the first assignment operation. IDP and NIV checks for memory access operations after the first assignment operation are green.

Consider the following code.

```
27   int *p;
28   int x;
29
30   p = (int *)0x32;   // Orange ABS_ADDR
31   x = *p;            // Green IDP and NIV
32
33   p++;
34   y = *p;            // Orange IDP and NIV
35
```

On line 30, the first assignment of the absolute address to a pointer produces an orange ABS_ADDR check. The next memory access operation produces green IDP and and NIV checks.

On line 34, the memory access operation produces orange IDP and NIV checks. The checks are orange because the accessed memory location is not covered by an orange ABS_ADDR check.

Note By default, the software displays ABS_ADDR checks on the **Results Explorer** or **Results Summary** tab only if you set the Review Level slider to `All`.

If you know that the absolute addresses in your code are valid, you can specify the option `-green-absolute-address-checks`, which makes all ABS_ADDR checks green. See “Green absolute address checks” on page 1-58.

IPT – Inspection Points

You can create inspection points in the code that provide range information.

For example:

```
#pragma Inspection_Point <var1> <var2>
```

where *var1* and *var2* are scalar variables, instructs the verification to provide range information for `var1` and `var2` at that point in the code.

You see this information in a tooltip message when you place your cursor over `var1` or `var2`.

Note Use the **Run-time Check Details Ordered by Color/File** component of the MATLAB® Report Generator™ to display IPT checks in generated reports. Under **Categories To Include**, select the **Inspection Point Checks (Informational Checks)** check box. For more information, see "Customize Verification Reports".

Consider the following example:

```
1
2  typedef struct {
3   unsigned char msb;
4   unsigned char lsb;
5  } int16;
6
7  int main(void)
8  {
9   volatile unsigned char var_uc;
10   float var_float;
11   int i;
12   int16 val;
13
14   #pragma Inspection_Point var_uc
15   i = 3;
16   #pragma Inspection_Point i
```

```
17   val.msb = 12;
18   val.lsb = var_uc;
19   #pragma Inspection_Point val
20   var_float = 10.0;
21   #pragma Inspection_Point var_float
22
23  }
24
```

The software provides tooltips with range information for the scalar variables `var_uc` and `i` at lines 14 and 16 respectively. However, tooltips are not provided for inspection points at lines 19 and 21.

POW (Deprecated)

Note The POW check is deprecated in R2009a and later. The POW check no longer appears in Polyspace results.

Check to establish whether the standard **pow** function from *math.h* library is used with an acceptable (positive) argument.

UNFL (Deprecated)

Note The UNFL check is deprecated in R2010a and later. The UNFL check no longer appears in Polyspace results. Instead of two separate UNFL and OVFL checks, a single OVFL check now appears.

These are checks to establish whether arithmetic expressions underflow. A scalar check is used with integer type, and a float check for floating point expressions. Consider the following example.

UOVFL (Deprecated)

Note The UOVFL check is deprecated in R2009a and later. The UOVFL check no longer appears in Polyspace results. Instead of a single UOVFL check, the results now display two checks, a UNFL and an OVFL.

The check UOVFL only concerns float variables. Polyspace verification shows an UOVFL when both overflow and underflow can occur on the same operation.

4

Check Descriptions for C++ Code

C++ Check Categories

This section presents all categories of checks that Polyspace software verifies. These checks are classified into acronyms. Each acronym represents one or more verifications made by Polyspace software. The list of acronyms, checks and associated colored messages are listed in the following tables.

In this section...

Acronyms Associated with Specific C++ Constructions

Category	Acronym	Green	Gray
function returns a value	FRV	function returns a value	Unreachable check: function returns a value
non null this-pointer	NNT	this-pointer [of f] is not null	Unreachable check: this-pointer [of f] is not null
C++ related instructions	CPP	array size is strictly positive	Unreachable check: array size is strictly positive
	CPP	typeid argument is correct	Unreachable check: typeid argument is correct
	CPP	dynamic_cast on pointer is correct	Unreachable check: dynamic_cast on pointer is correct
	CPP	dynamic_cast on reference is correct	Unreachable check: dynamic_cast on reference is correct
	INF	Informative check: f is implicitly called	Informative check: implicit call of f is unreachable

Category	Acronym	Green	Gray
Display of errors that relate to Object Oriented Programming and inheritance	OOP	call of virtual function [f] is not pure	Unreachable check: call of pure virtual function [f]
	OOP	this-pointer type [of f] is correct	Unreachable check: this-pointer type [of f] is correct
	INF	Informative check: f is called if this-pointer is of type T	Informative check: call of f depending on this type is unreachable
	OOP	pointer to member function points to a valid member function	Unreachable check: pointer to member function points to a valid member function
	OOP		Unreachable check: call to no function Information
	INF	Informative check: f is potentially called through pointer to member function	Informative check: potential call to f through pointer to member function is unreachable
	INF	Informative check: f is called during construction of T	Informative check: call of f during construction of T is unreachable
	INF	Informative check: f is called during destruction of T	Informative check: call of f during destruction of T is unreachable
Display of errors that relate to exception handling	EXC	exception raised as specified in the throw list	Unreachable check: exception raised as specified in the throw list
	EXC	catch parameter construction does not throw	Unreachable check: catch parameter construction does not throw
	EXC	dynamic initialization does not throw	Unreachable check: dynamic initialization does not throw

Category	Acronym	Green	Gray
	EXC	destructor or delete does not throw	Unreachable check: destructor or delete does not throw
	EXC	main, task or C library function does not throw	Unreachable check: main, task or C library function does not throw
	EXC	call [to f] does not throw	Unreachable check: call [to f] does not throw
	EXC	function does not throw	Unreachable check: function does not throw
	EXC	expression value is not EXCEPTION_CONTINUE_EXECUTION	Unreachable check: expression value is not EXCEPTION_CONTINUE_EXECUTION
	EXC		Unreachable check: throw is not allowed with option -no-exception

Category	Acronym	Red	Orange
function returns a value	FRV	Error: function does not return a value	Warning: function may not return a value
non null this-pointer	NNT	Error: this-pointer [of f] is null	Warning: this-pointer [of f] may be null

Category	Acronym	Red	Orange
C++ related instructions	CPP	Error: array size is not strictly positive	Warning: array size may not be strictly positive
	CPP	Error: incorrect typeid argument	Warning: typeid argument may be incorrect
	CPP	Error: incorrect dynamic_cast on pointer (verification continues using a null pointer)	Warning: dynamic_cast on pointer may be incorrect
	CPP	Error: incorrect dynamic cast on reference	Warning: dynamic_cast on reference may be incorrect
	INF		
Display of errors that relate to Object Oriented Programming and inheritance	OOP	Error: call of pure virtual function [f]	Warning: call of virtual function [f] may be pure
	OOP	Error: incorrect this-pointer type [of f]	Warning: this-pointer type of [f] may be incorrect
	OOP	Error: pointer to member function is null or points to an invalid member function	Warning: pointer to member function may be null or point to an invalid member function
	INF		
Display of errors that relate to exception handling	EXC	Error: exception raised is not specified in the throw list	Warning: exception raised may not be specified in the throw list
	EXC	Error: throw during catch parameter construction	Warning: possible throw during catch parameter construction
	EXC	Error: throw during dynamic initialization	Warning: possible throw during dynamic initialization
	EXC	Error: throw during destructor or delete	Warning: possible throw during destructor or delete

Category	Acronym	Red	Orange
	EXC	Error: main, task or C library function throws	Warning: main, task or C library function may throw
	EXC	Error: call [to f] throws (verification jumps to enclosing handler)	Warning: call [to f] may throw
	EXC	Error: function throws (verification jumps to enclosing handler)	Warning: function may throw
	EXC	Error: expression value is EXCEPTION_CONTINUE_ EXECUTION (limitation)	Warning: expression value may be EXCEPTION_CONTINUE_ EXECUTION (limitation)
	EXC	Error: throw is not allowed with option -no-exception	

Acronym Not Related to C++ Constructions (Also Used for C Code):

Category	Acronym	Green	Gray
Out of bound array index	OBAI	Array index is within its bounds	Unreachable check: out of bounds array index error
Zero division	ZDV		Unreachable check:
Non-initialized variable	NIV local/other	[local] variable is initialized	Unreachable check:
scalar or float overflows	OVFL		Unreachable check: variable overflow error
Illegal dereference pointer	IDP	Reference refers to a valid object	Unreachable check: invalid reference
Correctness condition	COR	Function pointer must point to a valid function	Unreachable check: Function pointer must point to a valid function

Category	Acronym	Green	Gray
Shift amount out of bounds	SHF	Scalar shift amount is within its bounds	Unreachable check: shift error
Non initialized pointer	NIP	Reference is initialized	Unreachable check: non-initialized reference
user assertion failures	ASRT	User assertion is verified	Unreachable check: user assertion error
non termination of call	NTC		
non termination of loop	NTL		
Unreachable check	UNR		Unreachable code

Category	Acronym	Red	Orange
Out of bound array index	OBAI	Out of bound array	Array index may be outside its bounds
Zero division	ZDV	[scalar | float] division by zero occurs	[scalar | float] division by zero may occur
Non-initialized variable	NIV local/other	[local] variable is not initialized	[local] variable may not initialized
scalar or float overflows	OVFL		
Illegal dereference pointer	IDP	Reference refers to an invalid object	Reference may not refer to a valid object
Correctness condition	COR	Function pointer must point to a valid function	Function pointer may point to a valid function
	COR	Array conversion must not extend range	
Shift amount out of bounds	SHF	Scalar shift amount is outside its bounds	
	SHF	Left operand of left shift is negative	

Category	Acronym	Red	Orange
Non initialized pointer	NIP	Reference is not initialized	Reference may be non-initialized
user assertion failures	ASRT	User assertion fails	User assertion may fail
non termination of call	NTC	[f] call never terminates	
non termination of loop	NTL	non termination of loop	
Unreachable check	UNR		

UNR – Unreachable Code

Check to establish whether different code snippets (assignments, returns, conditional branches and function calls) are reached. Unreachable code is referred to as *dead code*, which is indicated by UNR checks and code colored gray.

C++ Example

```
1
2     typedef enum {
3       Intermediate,   End,   Wait,   Init
4     }  enumState;
5
6     // automatic stubs
7     int intermediate_state(int);
8     int random_int(void);
9
10    bool State (enumState stateval)
11    {
12      int i;
13      if (stateval == Init) return false;
14      return true;
15    }
16
17    int main (void)
18    {
19      int i;
20      bool res_end;
21      enumState inter;
22
23      res_end = State(Init);
24      if (res_end == false) {
25        res_end = State(End);
26        inter = (enumState)intermediate_state(0);
27        if (res_end || inter == Wait) {             //  UNR Unreachable code
28          inter = End;
29        }
30        // use of i not initialized
```

```
31        if (random_int()) {
32          inter = (enumState)intermediate_state(i); // NIV Error: local variable is not initia
33          if (inter == Intermediate) {
34            inter = End;
35          }
36        }
37      } else {       // UNR Unreachable code
38        i = 1;
39        inter = (enumState)intermediate_state(i);
40      }
41      return res_end;
42    }
```

Explanation

The example illustrates why code might be unreachable and therefore colored gray:

- At line 27, `res_end`, a conditional part of a conditional branch is always true. Because of the standard definition of the logical operator `||`, the other part `inter == Wait` is never evaluated. The software generates two UNR checks — for `if` and `inter`.
- At line 24, the first branch of the `if` statement is always evaluated to be true. Therefore, at line 37, the other branch `else` is never executed. The software generates a UNR check.

Lines 33 and 34 after the red check on line 32 are not evaluated by Polyspace verification. To verify these lines of code, you must fix the red check and rerun the verification.

OBAI – Out of Bounds Array Index

Check to establish whether an index is within the bounds of array size during array access.

C++ Example

```
1     #define TAILLE_TAB  1024
2     typedef int tab[TAILLE_TAB];
3
4     class Array
5     {
6     public:
7       Array(){};
8       void initArray();
9     private:
10      tab table;
11    };
12
13
14    void Array::initArray()
15    {
16      int index;
17
18      for (index = 0; index < TAILLE_TAB ; index++){
19        table[index] = 10;
20      }
21      table[index] = 1;  // OBAI Error: array index is outside its bounds: [0..1023]
22    };
23
24
25    void main(void)
26    {
27      Array* test = new Array();
28      test->initArray();    // initArray has dashed, red underlining to indicate propagation o
29    }
```

Explanation

Just after the loop on line 18, `index` equals 1024. Therefore, on line 21, the assignment statement writes to a memory location that lies outside the array.

ZDV – Division by Zero

Check to establish whether the divisor of a division operator is not 0 (or 0.0).

C++ Example

```
1     extern int random_value(void);
2
3     class Operation {
4     public:
5       int zdvs(int p){
6         int j = 1;
7         return (1024 / (j-p));    // ZDV Error: scalar division by zero occurs
8       }
9       float zdvf(float p){
10       float j = 1.0;
11       return (1024.0 / (j-p)); // ZDV Error: float division by zero occurs
12      }
13    };
14
15    int main(void)
16    {
17      Operation op;
18
19      if (random_value())
20        op.zdvs(1);          // zdvs has dashed, red underlining to indicate propagation of ZD
21
22      if (random_value())
23        op.zdvf(1.0);        // zdvf has dashed, red underlining to indicate propagation of ZD
24    }
```

NIV (NIVL) – Non-Initialized Variable

Check to establish whether a variable local or not is initialized before being read. We make a distinction between local variables (including parameters of functions) and others. So Polyspace verification checks for same problems into two categories.

C++ Example

```
1     extern int random_int(void);
2     typedef double tab[20];
3
4
5     class operation
6     {
7     public:
8       int addI(int x, int y) { return y+=x; };  //*@@UNP-GRAY@@*/
9
10      void initTab(){
11        for (int i = 1; i < 20; i++) {
12          twentyFloat[i] = 0.0;
13        }
14      };
15
16      void addD(int x, int y){
17        twentyFloat[x] = twentyFloat[y] + 5.0;  // NIV Green: index 0
is not initialized, but addD method is called with parameters 2 and 4,
never 0. So index 2 and 4 are initialized
18      };
19
20    protected:
21      tab twentyFloat;
22    };
23
24
25    void main(void)
26    {
27      operation calculate;
28      int x, y = 0;
29
```

```
30      if (random_int()) {
31        calculate.addI(x,y);   // NIV ERROR: Non Initialized Variable
32      }
33
34      calculate.initTab();
35      calculate.addD(2,4);
36
37    }
```

Explanation

The result of the addition is unknown at line 28 because *x* is not initialized, (UNR unreachable code on "+" operator).

In addition, line 16 shows how Polyspace software prompts you to investigate further (by means of an orange check) when all cells have not been initialized.

A local variable is notified with a NIVL acronym.

Note The message associated with the check NIV or NIVL can give the type of the variable if it concerns a basic type: *"variable may be non initialized (type unsigned int32)"*. The modifier *volatile* can also be notified: *(type : volatile unsigned int 8*).

OVFL – Scalar and Float Overflow

In this section...

Scalar and Float Overflows: OVFL

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

C++ Example

```
1     #include <float.h>
2
3     extern int random_int(void);
4
5     class Calcul
6     {
7     public:
8       int makeOverflow(int i){
9         return  2 * (i - 1) + 2;      // OVFL Error: operation [+] on sca
10        // 2^31 is an overflow value for int32
11      }
12      float overflow (float value){
13        return  2 * value + 1.0;   // OVFL Error: operation [*] on float
14      }
15    };
16
17
18    void main(void)
19    {
20      Calcul c;
21      int i = 1;
```

```
22      float fvalue = FLT_MAX;
23
24      i = i << 30;                            // i = 2**30
25
26      if (random_int())
27        i = c.makeOverflow(i);
           // makeOverflow has dashed, red underlining to indicate propagat
           // OVFL ERROR from line 9
28
29      if (random_int())
30        fvalue = c.overflow(fvalue);
           // overflow has dashed, red underlining to indicate propagation
           // OVFL ERROR from line 13
31    }
```

Explanation

On a platform with a 32-bit architecture, the maximum integer value is 2^31-1. Therefore, on line 9, 2^31 causes an overflow. This OVFL error propagates to line 27.

In the same way, if `value` represents the biggest `float`, a number that is twice `value` cannot be represented by the same data type and causes an overflow on line 13. This OVFL error propagates to line 30.

Overflow on the Biggest Float

There are occasions when it is important to understand when overflow may occur on a float value approaching its maximum value. Consider the following example.

```
void main(void)
{
 float x, y;
 x = 3.40282347e+38f;    // is green
 y = (float) 3.40282347e+38;  // OVFL red
}
```

There is a red error on the second assignment, but not the first. The real "biggest" value for a float is: `340282346638528859811704183484516925440.0 - MAXFLOAT -`.

Now, rounding is not the same when casting a constant to a float, or a constant to a double:

- floats are rounded to the nearest lower value;
- doubles are rounded to the nearest higher value;
- 3.40282347e+38 is strictly bigger than 340282346638528859811704183484516925440 (named MAXFLOAT).
- In the case of the second assignment, the value is cast to a double first - by your compiler, using a temporary variable D1 -, then into a float - another temporary variable -, because of the cast. Float value is greater than MAXFLOAT, so the check is red.
- In the case of the first assignment, 3.40282347e+38f is directly cast into a float, which is less than MAXFLOAT

The solution to this problem is to use the "f" suffix to specify the variable directly as a float, rather than casting.

Constant Overflow

Consider the following example, which would cause an overflow.

```
int x = 0xFFFF; /* OVFL */
```

The table that follows shows three types of constants with corresponding lists of data types. The data type given to a constant is the first data type from the corresponding list that can accommodate the constant value. (See “Predefined Target Processor Specifications” for information about the size of a type depending on the target.)

Decimal	`int` , `long` , `unsigned long`
Hexadecimal	`int`, `unsigned int`, `long`, `unsigned long`
Float	`double`

For example (assuming 16-bit target):

5.8	`double`
6	`int`
65536	`long`
0x6	`int`
0xFFFF	`unsigned int`
5.8F	`float`
65536U	`unsigned int`

The option `-ignore-constant-overflows` allows the user to bypass this limitation and consider the line

```
int x = 0xFFFF; /* OVFL */
```

as

```
int x = -1;
```

instead of 65535, which does not fit into a 16-bit integer (–32768 to 32767).

Float Underflow Versus Values Near Zero

The definition of the word "underflow" differs between the ANSI standard and the ANSI/IEEE 754-1985 standard. According to the former definition, underflow occurs when a number is sufficiently negative for its type not to be capable of representing it. According to the latter, underflow describes the erroneous representation of a value close to zero due to the limits of its representation.

Polyspace verifications apply the former definition.

(The latter definition does not impose the raising of an exception as a result of an underflow. By default, processors supporting this standard permit the deactivation of such exceptions.)

Consider the following example.

```
1 #define FLT_MAX 3.40282347e+38F // maximum representable    \
float found in <float.h>
2 #define FLT_MIN 1.17549435e-38F // minimum normalised     \
float found in <float.h>
3
4 void main(void)
5 {
6  float zer_float = FLT_MIN;
7  float min_float = -(FLT_MAX);
8
9  zer_float = zer_float * zer_float; // No check underflow    \
near zero. VOA says {[expr] = 0.0}
10  min_float = -min_float * min_float; // OVFL ERROR: underflow  \
checked by verifier
11
12 }
```

SHF – Shift Operations

In this section...

Shift Amount is Outside its Bounds: SHF

Check to establish that a shift (left or right) is not greater than the size of integer type (`int` and `long int`). The range of allowed shift depends on the target processor, for example, 16 bits on `c-167` and 32 bits on `i386` for `int`.

C++ Example

```
1     extern int random_value(void);
2
3     class Shift {
4     public:
5       Shift(int val) : k(val){};
6       void opShift(int x, int l){
7        k  = x << l;         // SHF error: scalar shift amount is outside its bounds[0..31]
8       }
9       void opShiftSup(int x, int l){
10        k = x >> l;         // SHF error: scalar shift amount is outside its bounds[0..31]
11      }
12      void opShiftUnsigned(unsigned int x, int l){
13        unsigned int v = 1024;
14        v = x >> l;         // SHF error: scalar shift amount is outside its bounds[0..31]
15      }
16    protected:
17      int k;
18    };
19
20
21    void main(void)
22    {
23      int m, l = 1024;          // 32 bits on i386
24      unsigned u = 1024;
```

```
25
26      Shift s(1024);
27
28      if (random_value()) s.opShift(l ,32 );
          // opShift has dashed, red underlining to indicate propagation of SHF error
29      if (random_value()) s.opShiftUnsigned(u ,32 );
          // opShiftUnsigned has dashed, red underlining to indicate propagation of SHF error
30      if (random_value()) s.opShiftSup(l ,32 );
          // opShiftUnsigned has dashed, red underlining to indicate propagation of SHF error
31
32    }
```

Explanation

On lines 7, 10, and 14, the shifts are greater than the integer size.

Left Operand of Left Shift is Negative: SHF

Check to establish whether the operand of a left shift is a signed number.

C++ Example

```
1     extern int random_value(void);
2
3     class Shift {
4     public:
5       Shift(){};
6       int operationShift(int x, int y){
7         return x << 1;  // SHF Error: left operand of left shift is negative
8       }
9     };
10
11
12    void main(void)
13    {
14      Shift* s = new Shift();
15
16      if (random_value())
17        s->operationShift(-200,1);
```

```
            // operationShift has dashed, red underlining to indicate propagation of SHF error
18    }
```

Explanation

As signed number representation is stored in the higher order bit, you cannot left-shift a signed number without losing sign information.

Note The option `-allow-negative-operand-in-shift` allows explicitly signed numbers on shift operations. If you use this option when verifying the example, the red check at line 7 becomes a green check.

NNT – Pointer of function Not Null

This check verifies that the *this* pointer is null during call of a member function.

C++ Example

```
1     #include <new>
2     static volatile int random_int = 0;
3
4     class Company
5     {
6     public:
7       Company(int numbClients):numberClients(numbClients){};
8       void newClients (int numb) {
9         numberClients = numberClients + numb;
10      }
11    protected:
12      int numberClients;
13    };
14
15    void main (void)
16    {
17      Company *Tech = 0;
18
19      if (random_int)
20        Tech->newClients(2); // NNT ERROR: [this-pointer of
newClients is null]
21
22      Company *newTech = new Company(2);
23      newTech->newClients(1); // NNT Verified: [this-pointer
of newClients is not null]
24
25    }
26
```

Explanation

Polyspace verifies that all functions, virtual or not virtual, by a direct calling, and through pointer calling are never called with a null this-pointer.

In the above example, a pointer to a *Company* object is declared and initialized to null. When the *newClients* member function of the *Company* class is called (line 20), Polyspace detects that the class object is a null pointer.

On the new allocation at line 22, as standard *new* operator returns an initialized pointer or raises an exception, the *this-pointer* is considered as correctly allocated at line 23.

CPP – C++ Specific Checks

In this section...

Positive Array Size: CPP

This check verifies that the array size is always a non-negative value. In the following example, the array is defined with a negative value by a function call.

C++ Example

```
1     static volatile int random_int = 1;
2     static volatile unsigned short int random_user;
3
4     class Licence {
5     public:
6       Licence(int nUser);
7       void initArray();
8     protected:
9       int numberUser;
10      int (*array)[2];
11    };
12
13    Licence::Licence(int nUser) : numberUser(nUser) {
14      array = new int [numberUser][2]; // CPP error: array size is not strictly positive
15      initArray();
16    }
17
18    void Licence::initArray() {
19      for (int i = 0; i < numberUser; i++) {
20        array[i][2]=0;
21      }
```

```
22    };
23
24    void main (void)
25    {
26      if (random_int && random_user != 0)
27        Licence FirmUnknown (-random_user);
            // FirmUnknown has dashed, red underlining to indicate propagation of CPP error
28    }
```

Explanation

At line 14, where the dimension of `array` is defined by `[numberUser][2]`, the value of the array size is checked. However, the `numberUser` variable is always negative. Polyspace verification detects this error and displays the following message:

```
CPP error: array size is not strictly positive

Code sequence for probable cause:
2           volatile variable declaration random_user
27         formal argument number 1 (nuUser) of call to function Licence::Licence(int)
13         assignment
14         ! PAS_doc.Licence::Licence(int).CPP
```

Incorrect typeid Argument: CPP

Check to establish whether a *typeid* argument is not a null pointer dereference. This check only occurs using typeid function declared in stl library <typeinfo>.

C++ Example

```
1     #include <typeinfo>
2
3     static volatile int random_int=1;
4
5     class Form
6     {
7     public:
8       Form (){};
9       virtual void trace(){};
```

```
10    };
11
12    class Circle : public Form
13    {
14    public:
15      Circle() : Form () {};
16      void trace(){};
17    };
18
19
20    int main ()
21    {
22
23      Form* pForm = 0 ;
24      Circle *pCircle = new Circle();
25
26      if (random_int)
27        return (typeid(Form) == typeid(*pForm));    // CPP ERROR:
[incorrect typeid argument]
28      if (random_int)
29        return (typeid(Form) == typeid(*pCircle));  // CPP Verified:
[typeid argument is correct]
30    }
31
32
33
34
```

Explanation

In this example, the *pForm* variable is a pointer to a *Form* object and initialized to a null pointer. Using the *typeid* standard function, an exception is raised. In fact here, the typeid parameter of an expression obtained by applying the unary "*" operator is a null pointer leading to this red error.

At line 29, **pCircle* is not null and typeid can be applied.

Incorrect dynamic_cast on Pointer: CPP

Check to establish when only valid pointer casts are performed through *dynamic_cast* operator. +

C++ Example

```
1     #include <new>
2     static volatile int random = 1;
3
4     class Object {
5     protected:
6       static Object* obj;
7     public:
8       virtual ~Object() {}
9     };
10
11    class Item : Object {
12    private:
13      static Item* item;
14    public:
15      Item();
16    };
17
18    Object* Object::obj = new Object;
19
20    Item::Item() {
21      if (obj != 0) {
22        item = dynamic_cast<Item*>(obj); // CPP ERROR: [incorrect
dynamic_cast on pointer (verification continue using a null pointer)]
23        if (item == 0) { // here analyzed and reachable code
24          item = this;
25        }
26      }
27    }
28
29    void main()
30    {
31      Item *first= new Item();
32    }
```

Explanation

Only the dynamic casting between a subclass and its upclass is authorized. So, the casting of *Object* object to a *Item* object is an error on *dynamic_cast* at line 21, because *Object* is not a subclass of *Item*.

Behavior follows ANSI C++ standard, in sense that even if dynamic_cast is forbidden, verification continue using null pointer. So at line 22, *item* is considered as null and assigned to *this* at line 23.

Note **This is only check where we can have another color after a red**. It is not the case for a dynamic_cast on a reference.

Incorrect dynamic_cast on Reference: CPP

Check to establish when only valid reference casts are performed through *dynamic_cast* operator.

C++ Example

```
1     #include <new>
2     static volatile int random = 1;
3     class Object {
4     protected:
5       static Object* obj;
6     public:
7       virtual ~Object() {}
8     };
9
10    class Item : public Object {
11    private:
12      static Item* item;
13    public:
14      Item& get_item();
15      Item& other_item();
16    };
```

```
17
18    Object* Object::obj = new Object;
19
20    Item& Item::get_item() {
21      Item& ref = dynamic_cast<Item&>(*Object::obj);
// CPP Error: incorrect dynamic_cast on reference
22      *item  =  ref;
// unreachable code
23    }
24
25    void main ()
26    {
27      Item * first= new Item();
28      if (random)
29        first->get_item();
// get_item has dashed, red underlining to indicate propagation of dynamic_
30      Object &refo = dynamic_cast<Object&>(first->other_item());
// CPP Verified: [dynamic_cast on reference is correct]
31    }
```

Explanation

Only the dynamic casting between a subclass and its upclass is authorized. So, the casting of reference `Object` object to a reference `Item` object is an error on `dynamic_cast` at line 21, because `Object` is not a subclass of `Item`.

The verification stops at line 21 and the error is propagated to line 29. The tooltip for `get_item` shows:

```
A problem occurs during the execution of call to function DCTR_doc.Item::get_item().
See check CPP at DCTR_doc.cpp line 21 ...
...
```

The behavior is different with a dynamic_cast on a pointer.

FRV – Function Returns a Value

Check to establish whether function returns a value when a value is expected.

C++ Example

```
1     static volatile int rand;
2
3     class function {
4     public:
5       function() { rep = 0; }
6       int reply(int msg) {     // FRV green: function returns a value
7         if (msg > 0) return rep;
8       };
9
10      int reply2(int msg) {    // FRV Error: function does not return a value
11        if (msg > 0) return rep;
12      };
13
14      int reply3(int msg) {    // FRV Warning: function may not return a value
15        if (msg > 0) return rep;
16      };
17
18    protected:
19      int rep ;
20    };
21
22    void main (void){
23
24      int ans;
25      function f;
26
27      if (rand)
28        ans = f.reply(1);
29
30      else if (rand)
31        ans = f.reply2( 0);  // reply2 has dashed, red underlining to indicate propagation of
32      else
33        f.reply3(rand);
```

```
34      }
```

Explanation

Variables are often initialized using the return value of functions. However, in the example, the return value is not initialized for all input parameter values. Therefore, the target variable is not properly initialized with a valid return value.

IDP – Illegal Dereferenced Pointer

In this section...

Pointer is Outside its Bounds: IDP

Check to establish whether a reference refers to a valid object (whether the dereferenced pointer is still within the bounds of the pointed object).

C++ Example

```
1     #define TAILLE_TAB  1024
2
3     typedef int tab[TAILLE_TAB];
4
5     class Array {
6     public:
7       Array(tab a){
8         p = a;
9         initArray();
10      }
11      void initArray(){
12        int index;
13        for (index = 0; index < TAILLE_TAB ; index++, p++) {
14          *p = 0;
15        }
16      }
17      void changeNextElementWithValue(int i){
18        *p = i;       // IDP Error: pointer is outside its bounds
19      }
20
21    private:
22      int *p;
23    };
```

```
24
25
26    void main(void)
27    {
28      tab t;
29
30      Array a(t);
31      a.changeNextElementWithValue(1);
          // changeNextElementWithValue has dashed, red underling to indicate propagation of IDP
32    }
```

Explanation

The pointer `p` is initialized to point to the first element of `tab` at line 8. When the loop ends, `p`:

```
...
points to 4 bytes at offset 4096 in buffer of 4096 bytes, so is outside bounds
...
```

For more information, see:

- “Understanding Addressing” on page 4-36
- “Understanding Pointers” on page 4-40

Understanding Addressing

- “Hardware Registers” on page 4-36
- “NULL pointer” on page 4-38
- “Comparing addresses” on page 4-39

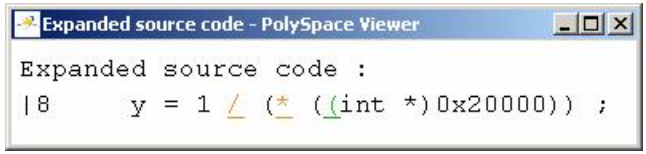
Hardware Registers

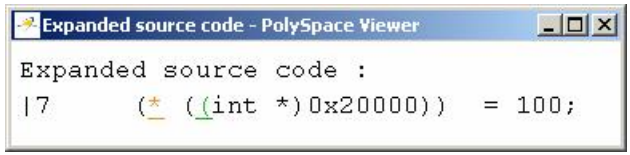
Many code verifications exhibit **orange** out of bound checks with respect to accesses to absolute addresses and/or hardware registers.

(Also refer to the discussion on Absolute Addressing)

Here is an example of what such code might look like:

```
#define X (* ((int *)0x20000))
X = 100;
y = 1 / X;  // ZDV check is orange because X ~ [-2^31, 2^31-1] permanently.
     // The pointer out of bounds check is orange because 0x20000
     // may address anything of any length
     // NIV check is orange on X as a consequence
```






```
3 void main (void)
4 {
5 int y;
6
7 X = 100;
8 y = 1 / X;
9
10 }
```

```
int *p = (int *)0x20000;
*p = 100;
y = 1 / *p;  // ZDV check is orange because *p ~ [-2^31, 2^31-1] permanently
     // The pointer out of bounds is orange because 0x20000
     // may address anything of any length
     // NIV check on *p is orange as a consequence
```

This can be addressed by defining registers as regular variables:

Replace	With
`#define X ....`	`int X;`

`int *p;`	`int _p;`
	`#define p (&_p)`
	
	Note Check that the chosen variable name (p in this example) does not already exist

`int *p;`	`volatile int _p;` `int *p = &_p;`

NULL pointer

Consider the following NULL address:

```
#define NULL 0
```

- It is illegal to dereference this 0 value
- 0 **is not** treated as an absolute address.

```
*NULL = 100; // produces a red   Illegal Dereference
Pointer (IDP)
```

Assuming these declarations:

```
int *p = 0x5;
volatile int y;
```

and these definitions:

```
#define NULL 0
#define RAM_MAX ((int *)0xffffffff)
```

consider the code snippets below:

```
While (p != (void *)0x1)
```

```
 p--; // terminates
```

0x1 is an absolute address, it can be reached and the loop terminates

```
for (p = NULL; p <= RAM_MAX; p++)
{
 *p = 0; // illegal dereference of pointer
}
```

At the first iteration of the loop p is a NULL pointer. Dereferencing a NULL pointer is forbidden.

```
While (p != NULL)
{
 p--;
 *p = 0; // Orange dereference of a pointer
}
```

When p reaches the address 0x0, there is an attempt to considered it as an absolute address In effect, it is an attempt to dereference a NULL pointer – which is forbidden. Note that in this case, the check is orange because the execution of the code here is ok (green) until 0x0 is reached (red)

The best way to address this issue depends on the purpose of the function.

- Thanks to the default behavior of Polyspace verification, it is easy to automatically stub a function whose purpose is to copy data from/to RAM or to compute a checksum on RAM.
- If a function is supposed to copy calibration data, it should also be stubbed automatically.
- If the purpose of a function is to map EEPROM data to global variables, then a manually written stub is required to assign initial values to the global variables.

Comparing addresses

Polyspace verification only deals with the information referred to by a pointer, and not the physical location of a variable. Consequently it does not compare addresses of variables, and makes no assumption regarding where they are located in memory.

Consider the following two examples of Polyspace verification behavior:

```
int a,b;
if (&a > &b) // condition can be true and/or false
{ } // both branches are reachable
else
{ } // both branches are reachable
```

and

```
int x,z;
void main(void)
{ int i;
 x = 12;
 for (i=1; i<= 0xffffffff; i++)
 {
  *((int *)i) = 0;
 }
 z = 1 / x;  // ZDV green check because Polyspace doesn't consider any
     // relationship between x and its address
}
```

"x" is aliased by no other variable. No pointer points to "x" in this example, so as far as the Polyspace verification is concerned, "x" remains constantly equal to 12.

Understanding Pointers

Polyspace verification doesn't analyze anything which would require the physical address of a variable to be taken into account.

- Consider two variables x and y. Polyspace verification will not make a meaningful comparison of "&x" (address of x) and "&y"
- So, the Boolean (&x < &y) can be true or false as far as Polyspace verification is concerned.

However, Polyspace verification does keep track of the pointers that point to a particular variable.

- So, if ptr points to X, *ptr and X will be synonyms.

- "How does malloc work for Polyspace verification?" on page 4-41
- "Structure Handling — Array Conversions: COR" on page 4-41
- "Structure Handling — Mapping a Small Structure into a Bigger One" on page 4-42

How does malloc work for Polyspace verification?

Polyspace verification models `malloc`, such that both the possible return values of a null pointer and the requested amount of memory are taken into account.

Consider the following example.

```
void main(void)
{
 char *p;
 char *q;
 p = malloc(120);
 q = p;
 *q = 'a'; // results in an orange dereference check
}
```

This code will avoid the orange dereference:

```
void main(void)
{
 char *p;
 char *q;
 p = malloc(120);
 q = p;
 if (p!= NULL)
  *q = 'a'; // results in a green dereference check
}
```

Structure Handling — Array Conversions: COR

Check to establish whether a small array is mapped onto a bigger one through pointer cast.

C++ Example.

```
1     typedef int Big[100];
2     typedef int Small[10];
3     typedef short EquivBig[200];
4
5     Small smalltab;
6     Big bigtab;
7
8     extern int random_val();
9
10    void main(void)
11    {
12
13      Big   * ptr_big   = &bigtab;
14      Small * ptr_small = &smalltab;
15
16      if (random_val()){
17        Big *new_ptr_big = (Big*)ptr_small;     // COR ERROR:
array conversion must not extend range
18      }
19
20      if (random_val()){
21        EquivBig *ptr_equivbig = (EquivBig*)ptr_big;
22        Small *ptr_new_small = (Small*)ptr_big; // COR Verified
23      }
24    }
```

Explanation. In the example above, a pointer is initialized to the *Big* array with the address of a the *Small* array. This is not legal since it would be possible to dereference this pointer outside of the *Small* array. Line 22 shows that the mapping of arrays with same length and different prototypes is authorized.

Structure Handling — Mapping a Small Structure into a Bigger One

For example, if *p* is a pointer to an object of type *t_struct* and it is initialized to point to an object of type *t_struct_bis* whose size is less than the size of *t_struct*, it is illegal to dereference *p* because it would be possible to access

memory outside of *t_struct_bis*. Polyspace software prompts you to investigate further by means of an orange check. See the following example.

```
1 #include <malloc.h>
2
3 typedef struct {
4  int a;
5  union {
6   char c;
7   float f;
8  } b;
9 } t_struct;
10
11 void main(void)
12 {
13  t_struct *p;
14
15  // optimize memory usage
16  p = (t_struct *)malloc(sizeof(int)+sizeof(char));
17
18  p->a = 1; // IDP Warning: reference may not refer to a
valid object
19
20 }
```

COR – Correctness Condition

In this section...

Function Pointer Does Not Point to a Valid Function: COR

This is a check to establish whether a function pointer points to a valid function or a function with a valid prototype. The software checks, for example, whether:

- The pointer points to a function.
- Each argument passed to a function matches the corresponding argument in the function prototype.
- The number of arguments passed to a function matches the number of arguments in the function prototype.
- The return type passed to a function pointer matches the return type declared in the function prototype.

C++ Example

```
1     typedef void (*CallBack)(void *data);
2
3     struct {
4       int ID;
5       char name[20];
6       CallBack func;
7     } funcS;
8
9     float fval;
10
11    void main(void)
```

```
12    {
13      CallBack cb =(CallBack)((char*)&funcS + 24 * sizeof(char));
14
15      cb(&fval);
16     // Red COR: function pointer does not point to a valid function
17    }
```

Explanation

In the example, `func` has a prototype that conforms to the declaration for `CallBack`. Therefore, `func` is initialized to point to the NULL function through the global declaration of `funcS`.

Verification generates a red COR check for `cb` (line 15). The **Check Details** pane gives the reason:

```
...
pointer does not point to any function
...
```

C++ Example

```
1     static volatile int random = 1;
2
3     int f(float f) { return 0; }
4     int g(int i) { return i; }
5
6     typedef int (*func_int)(int);
7
8     func_int ftab = (func_int)f;
9
10    void badTab(int i) {
11      ftab(++i) ;
12      // Red COR: function pointer does not point to a valid function
13
14    }
15
16   int main()
17    {
```

```
18      int idx  = 0;
19
20      for (int i = 9; i < 10; ++ i) {
21        if (random)
22          badTab(++idx);
23          // In Source view, badTab displayed with red, dash-underlining to highlight problem
24      }
25    }
```

Explanation

In this example, `ftab` is a pointer to a function that expects a `float` input argument. However, the input argument is an `int`.

Verification generates a red COR check for `ftab` (line 11). The **Check Details** pane gives the reason:

```
...
pointer is not null
pointer points to badly-typed function: f
...
```

C++ Example

```
1     extern int random_value(void);
2
3     typedef int (*t_func_2)(int);
4     typedef int (*t_func_2b)(int,int);
5
6     int foo_nb(int x)
7     {
8       if (x%2 == 0)
9         return 0;
10      else
11        return 1;
12    }
13
14    void main(void)
15    {
```

```
16      t_func_2b ptr_func;
17      int i = 0;
18
19      ptr_func = (t_func_2b)foo_nb;
20      if (random_value())
21        i = ptr_func(1,2);
22        // Red COR: function pointer does not point to a valid function
23    }
```

Explanation

In this example, `ptr_func` is a pointer to a function that takes two arguments. However, the function pointer is initialized to point to a function that takes only one argument.

Verification generates a red COR check for `ptr_func` (line 21). The **Check Details** pane gives the reason:

```
...
- error when calling function foo_nb: wrong number of arguments (call has 2 but
  function expects 1)
...
```

Scalar Overflow on Division (/) Operation: COR

This is a check to establish whether the value returned from a division operation (/) overflows its declaration.

C++ Example

```
1 #define MAX_INT 2147483647
2 #define MIN_INT (-MAX_INT-1)
3
4 void foo(void)
5 {
6  int a, b, c;
7  a = MIN_INT;
8  b = -1;
9  c = a/b;   // COR error:  operation [/] on scalar overflows
(result is always strictly greater than MAX_INT32)
```

```
10 }
```

Explanation

In this example, `a` is `MIN_INT`, so `MIN_INT / -1` is equal to `MAX_INT+1`. This causes an overflow when `c` is assigned to an `int32`.

This error occurs on a division, so it is assigned a `COR` check.

NIP – Non-Initialized Pointer

Check to establish whether a reference is initialized before being dereferenced.

C++ Example

```
1     class declare
2     {
3     public:
4       declare(int* p):pointer(p){};
5       int changeValue(int val){*pointer = 0;};
6     protected:
7       int* pointer;
8     };
9
10    void main(void)
11    {
12      int* p;
13      declare newPointer(p);          // NIP ERROR:
reference is not initialized

14      newPointer.changeValue(0);
15    }
```

Explanation

As *p* is not initialized, the line 5 (**pointer = 0*) would overwrite an unknown memory cell (corresponding to the unreachable gray code on "*").

EXC – Exception Handling

In this section...

Function throws: EXC

Check to verify that a function never raises an exception for every returned values.

C++ Example

```
1     #include <vector>
2
3     static volatile int random_int = 1;
4     class error{};
5
6     class InitVector
7     {
8     public:
9       InitVector (int size) {
10        sizeVector = size;
11        table.resize(sizeVector);
12        Initialisation();
13      };
14      void Initialisation ();
15      void reSize(int size);
16      int getValue(int number) throw (error);
17      int returnSize();
18    private:
```

```
19      int sizeVector;
20      vector<int> table;
21    };
22
23    void InitVector::Initialisation() { // EXC Warning: [functions
may throw]
24      int i;
25      for (i = 0; i < table.size(); i++){
26        table[i] = 0;
27      }
28      if (random_int) throw i;
29    }
30
31    void InitVector::reSize(int sizeT) {
32      table.resize(sizeT);
33      sizeVector = table.size();
34    }
35
36    int InitVector::getValue(int number) throw (error) { // EXC ERROR:
[function throws (verification jumps to enclosing handler)]
37      if (number >= 0 && number < sizeVector)
38        return table[number];
39      else throw error();
40    }
41
42    int InitVector::returnSize() { // EXC Verified: [function
does not throw]
43      return table.size();
44    }
45
46    void main (void)
47    {
48      InitVector *vectorTest = new InitVector(5);
49
50      if (random_int)
51        vectorTest->returnSize();
52
53      if (random_int)
54        vectorTest->getValue(5); // EXC ERROR: [call to getValue
throws (verification jumps to enclosing handler)]
```

```
55    }
```

Explanation

The class *InitVector* allows to create a new vector with a defined size. The *resize* member function allows to change the size, without any size limit. *returnSize* returns the vector's size, and no exception can be thrown. A green check is displayed for this function: *[function does not throw]*.

The *getValue* function returns the array's value for a given index. If the parameter is outside vector bounds, an exception is raised. For a vector'size of 5 elements, valid index are [0..4]. At line 53, the programmers tries to access the fifth element *table[5]*. An exception is raised and Polyspace displays a red message.

Polyspace Verfier tests functions that raises exception or no, with void or no-void type:

- always: function throws (verification jumps to enclosing handler)
- never: function does not throw
- sometimes: function may throw

When this check happens, a propagation to caller is made with another exception check [call to <name> throws] (see line 53).

Call to Throws: EXC

Check to verify that a function call raises or not an exception.

C++ Example

```
1     static volatile int random_int =1 ;
2
3     class error{};
4
5     class A
6     {
7     public:
```

```
8       A() {value=9;};
9       int badReturn() throw (int);
10      int goodReturn() throw (error);
11    protected:
12      int value;
13    };
14
15    int A::badReturn() throw (int) { // EXC ERROR: [function
throws (verification jumps to enclosing handler)]
16      if(!value)
17        return value;
18      else
19        throw 2;
20    };
21
22    int A::goodReturn() throw (error) { // EXC Verified: [function
does not throws]
23      int p = 7;
24      if (p>0)
25        return value;
26      else
27        throw error();
28    };
29
30    void main (void)
31    {
32      A* a = new A();
33      if(random_int)
34        a->badReturn(); // EXC ERROR: [call to badRetrun throws
(verification jumps to enclosing handler)]
35      if(random_int)
36        a->goodReturn(); // EXC Verified: [call to goodRetrun
does not throw]
37    }
```

Explanation

In the first call, Polyspace proposes to caller that the function always raises an exception because member variable value is always different from 0.

In the second call, Polyspace verification checks that no throw has been made in the function because the conditional test at line 24 is always true.

Most of the time, the *[call to <name> throws]* is associated to [function throws] check.

Destructor or Delete Throws: EXC

Check to establish whenever an exception is “thrown” and not “caught” in a destructor or during a delete operation.

C++ Example

```
1     #include <math.h>
2     using namespace std;
3     volatile unsigned int random_int = 1 ;
4
5     class error{};
6
7     class Rectangle
8     {
9     public:
10      Rectangle(){};
11      Rectangle (unsigned int longueur, unsigned int large):
longueurRect(longueur),largeRect(large){};
12
13      virtual ~Rectangle(){     // EXC Warning: possible throw during destructor or delete
14        if (!random_int)
15          throw error();
16      };
17
18      virtual double calculArea() {
19        return longueurRect * largeRect;
20      };
21
22    protected:
23      unsigned int longueurRect;
24      unsigned int largeRect;
25    };
26
```

```
27  class Cube : public Rectangle
28  {
29  public:
30    Cube():cote(3){};
31    ~Cube(){                    // EXC ERROR: throw during destructor or delete

32      if(random_int>=0)
33        throw error();
34    };
35    double calculArea(){
36      return pow(cote,cote);
37    };
38  protected:
39    int cote ;
40  };
41
42  void main (void)
43  {
44    try {
45      Rectangle* form1 = new Rectangle(10,2);
46      double k = form1->calculArea();
47
48      Cube* form2 = new Cube;
49      double l = form2->calculArea();
50
51      delete form1;
52      delete form2;
              // delete on line 52 has red, dashed underlining to
              // indicate propagation of NTC error
53    }
54    catch (error){
55      //raised when an error occurs in a destructor
56    }
57    catch (...){}
58  }
```

Explanation

In class `Cube`'s destructor at line 31, an error is raised when `random_int` is greater than 0. As `random_int` was declared as a `volatile unsigned int`, this condition is always true.

At line 13, in the destructor of class `Rectangle`, the test on the `random_int` value may be true when it is different from 0. Thus, an exception might be raised in the destructor, and an orange warning is displayed.

Destructors are called during stack unwinding when an exception is thrown. In this case any exception thrown by a destructor would cause the program to terminate. Therefore it is better programming to catch exceptions in destructors.

Main, Tasks or C Library Function Throws: EXC

Check that functions used at C level, in a task or in main do not raise exceptions.

C++ Example

```
1     #include <cstdlib>
2     #include <iostream>
3     static volatile int random_int = 1;
4
5     extern "C" {
6       int compare (const void * a, const void * b) {
// EXC Verifeid:
[main, task or C library function does not throw]
7         return ( *(int*)a - *(int*)b );
8       }
9       int c_compare_bad (const void *k, const void *e) {
// EXC ERROR:
[main, task or C library function throws]
10        throw 1;
11      }
12    };
13
14    typedef int arrayT[5];
```

```
15
16    class arrayToRange
17    {
18    public:
19      arrayToRange(arrayT* a) :tab(a) {};
20      arrayT* returnTabInOrder() {
21        qsort(*tab, 5, sizeof(int), compare);
22        return tab;
23      };
24      arrayT* returnTabInOrderBad() {
25        qsort(*tab, 5, sizeof(int), c_compare_bad);
26        return tab;
27      };
28    protected:
29      arrayT* tab;
30    };
31
32    void main(void) // EXC Verified: [main, task or C library
function does not throw]
33    {
34     try
35     {
36      arrayT tabInit = {1,3,4,2,5};
37      arrayT* table = &tabInit;
38      arrayToRange ArrayTest(table);
39      ArrayTest.returnTabInOrderBad(); // No jump to enclosing
handler
40      ArrayTest.returnTabInOrder();
41     }
42     catch (...) {                          // gray code
43       cout << "error raised:" <<  "bye"; // gray code
44     }
45    }
```

Explanation

In this example, we called a C stubbed function, *qsort* defined in the include file cstlib, which returns a sorted array of integers. Two functions, defined in a class called *arrayToRange*, call this *qsort* function:

- The first one, *returnTabInOrder*, calls *qsort*, with a C function pointer as third parameter, which can not raise an exception. So Polyspace software displays a green message (line 6).
- The second one, *returnTabInOrderBad*, uses a C function pointer which always raises an exception. Polyspace software displays a red message on the C function (line 9).

Limitation: even if *c_compare_bad* function always raise an exception, Polyspace verification does not propagate to enclosing handler. Indeed at line 39, all is green and the verification continue even if call is surrounded by a *try/catch* leading to gray code in catch block.

Exception Raised is Not Specified in the Throw List: EXC

Check to determine whether a function has thrown a non-authorized exception.

C++ Example

```
1     #include <string>
2
3     using namespace std;
4
5     int negative_balance = -300;
6
7     class NotPossible
8     {
9     public:
10      NotPossible(const string & s) : Error_Message(s) { };
11      ~NotPossible(){};
12      string Error_Message;
13    };
14
15    class Account
16    {
17    public:
18      Account(long accountInit):account(accountInit) {}
19      void debit (long amount) throw (int, char);
```

```
20      long getAccount () { return account; };
21    protected:
22      long account;
23    };
24
25    void Account::debit(long amount) throw (int, char) {
                //  EXC Error: exception raised is not specified in the throw list
26      if ((account - amount) < negative_balance)
27        throw NotPossible ("error");
28      account = account - amount;
29    }
30
31    void main (void)
32    {
33      try {
34        Account *James = new Account(12000);
35        James -> debit(13000);
          // debit has red, dashed underlining to indicate propagation of NTC error:
propagation of not specified exception
36        long total = James -> getAccount();
37      }
38      catch (NotPossible&){}
39      catch (...){};
40    }
41
```

Explanation

In the example, the `debit` function of the `Account` class can throw the specified exception. This function can only catch the `int` and `char` exceptions. The bank has authorized an overdraft of 300 euros. The `James` account is created with an initial balance of 12000 Euros. At line 35, his account is debited by 13000. In the `debit` function, the `if` condition (line 26) is true. Therefore, a `NotPossible` exception is raised. Unfortunately, this exception type is not allowed within the throw list at line 25 even if the catch operand allows it. Therefore, Polyspace verification detects an error.

Throw During Catch Parameter Construction: EXC

Check to prevent throw during dynamic initialization in constructors and during initialization of arguments in *catch*.

C++ Example

```
1     #include <string>
2
3     static volatile int random_int = 1;
4     static volatile int random_red = 0;
5
6     class error{};
7
8     class NotPossible
9     {
10    public:
11      NotPossible(const NotPossible&) // EXC ERROR: [function
throws (verification jump to enclosing handler)]
12      {
13        throw error();
14      };
15      NotPossible()                   // NRE ERROR: [function
throws (verification jump to enclosing handler)]
16      {
17        throw NotPossible(7);
18      };
19      NotPossible(int){};
20      ~NotPossible(){};
21    private:
22      string Error_Message;
23    };
24
25    class Test
26    {
27    public:
28      Test(int val) : value(val){};
29      int returnVal(){
30        if (random_int)
31          throw error();
```

```
32        else
33          return value;
34      };
35    private:
36      int value;
37    };
38
39    int main() {
40
41      try {
42        Test* T = new Test(1);
43        if (random_red)
44          throw NotPossible();  // EXC ERROR: [call to
NotPossible throws (verification jumps tp enclosing handler)]
45        else
46          T->returnVal();
47        if (random_red) {
48          NotPossible * Npos = new NotPossible(); // EXC
ERROR: [throw during dynamic initialization]
49        }
50      }
51      catch(NotPossible a) {}   // EXC ERROR: [throw during
catch parameter conctruction]
52      catch(...) {}
53    }
```

Explanation

At line 48 of the previous example, during dynamic initialization of Npos, a call to default constructor NotPossible is made. This constructor raises an exception leading to the EXC error. Indeed, raising an exception during a dynamic initialization is not authorized.

In same example at line 51, an exception is caught by the throw coming from line 44. A variable of type *NotPossible* is created at line 48 using also same default constructor. However, this constructor throws an *integer* exception leading to red error at line 48.

Each catch clause (exception handler) is like a function that takes a single argument of one particular type. The identifier may be used inside the handler, just like a function argument. Moreover, the throw of an exception in a catch block is not authorized.

Continue Execution in __except: EXC

Check to establish whether in a __except catch block the use of MACRO EXCEPTION_CONTINUE_EXECUTION. This check can only occur using a visual dialect.

C++ Example

```
1
2     #include <windows.h>
3     #include <excpt.h>
4
5     void* data;
6     struct No_Data {};
7
8     void* check_glob() {          // EXC ERROR: [function throws
(verification jumps to enclosing handler)]
9       if (!data) throw No_Data(); // EXC ERROR: []
10      return data;
11    }
12
13    int main() {
14      __try {
15        data = 0;
16        check_glob();   // EXC ERROR: [call to check_glob() throws
(verification jumps to enclosing handler)]
17      }
18      __except(data == 0
19               ? EXCEPTION_CONTINUE_EXECUTION  // EXC ERROR:
[expression value is EXCEPTION_CONTINUE_EXECUTION]
20               : EXCEPTION_EXECUTE_HANDLER) {
21        data = new (void*);                    // Gray code
22      }
23    }
```

Explanation

In this example, the call to function check_glob() throws an exception. This exception jumps to enclosing handler, in this case the __*except* block. Using EXCEPTION_CONTINUE_EXECUTION, it could be possible normally to continue verification and comes back at line 9 as if exception never happened. In the example, data is assigned to new value at line 21 in __except block and no more throw will occur.

Polyspace verification cannot handle this kind of behavior and put a red error on the EXCEPTION_CONTINUE_EXECUTION keyword since it has found a path to this instruction. It results gray code at line 21 and at line 10. All other red errors concern management of the exception: function throws and call throws].

Note It is possible to match functional behavior using volatile keyword by replacing code at line 5: *volatile void *data;*

ASRT – User Assertion

Check to establish whether a user assertion is valid. If the assumption implied by an assertion is invalid, then the standard behavior of the assert macro is to abort the program. Polyspace verification therefore considers a failed assertion to be a run-time error.

C++ Example

```
1     #include <assert.h>
2
3     typedef enum
4     {
5       monday=1, tuesday,
6       wensday,  thursday,
7       friday,   saturday,
8       sunday
9     } dayofweek  ;
10
11    // stubbed function
12    dayofweek random_day(void);
13    int random_value(void);
14
15    void main(void)
16    {
17      unsigned int var_flip;
18      unsigned int flip_flop;
19      dayofweek curDay;
20      unsigned int constant = 1;
21
22      if (random_value()) flip_flop=1; else flip_flop=0;
// flip_flop randomly be 1 or 0
23      var_flip = (constant | random_value());
// var_flip is always > 0
24
25      if(random_value()) {
26        assert(flip_flop==0 || flip_flop==1); // ASRT Verified:
user assertion is verified
27        assert(var_flip>0);                   // ASRT Verified
```

```
28        assert(var_flip==0);                    // ASRT ERROR:
user assertion fails
29      }
30
31      if (random_value()) {
32        curDay = random_day();                  // Random day
of the week
33        assert( curDay > thursday);             // ASRT Warning:
 User assertion may fail
34        assert( curDay > thursday);             // ASRT Verified
35        assert( curDay <= thursday);            // ASRT ERROR:
user assertion fails
36      }
37    }
```

Explanation

In the *main*, the *assert* function is used in two different ways:

- To establish whether the values *flip_flop* and *var_flip* in the program are inside the domain which the program is designed to handle. If the values were outside the range implied by the *assert* (see line 28), then the progam would not be able to run properly. Thus they are flagged as run-time errors.
- To redefine the range of variables as shown at line 34 where *curDay*is restricted to just a few days. Indeed, Polyspace verification makes the assumption that if the program is executed without a run-time error at line 33, *curDay* can only have a value greater than *thursday* after this line.

OOP – Object Oriented Programming

In this section...

Invalid Pointer to Member: OOP

Polyspace verification checks that the pointer to a function member is invalid or null.

C++ Example

```
1
2  class A {
3  public:
4     void f() {
5    }
6  };
7
8  int main() {
9
10  void (A::*pf)(void) = &A::f;
11    int (A::*pf2)(void) =  (int (A::*)(void))&A::f;
12
13    volatile int random;
14    A a;
15
16    if (random) {
17      int res = (a.*pf2)() ; // RED OOP ERROR [pf2 points to A::f   \
that does not return a value]
18      res++;
19    }
20
21    pf = 0;
22    if (random) {
```

```
23   (a.*pf)() ; // Red OOP ERROR [pf pointer is null]
24    }
25 }
```

Explanation

When a function pointer operates on a null pointer to a member value, the behavior is undefined. In the above example, the *pf* pointer is declared and initialized to a null member function. When the function is called (at line 23) a red error is raised. In addition, the *pf2* pointer points to `A::f`, that does not return a value, raising another red error at line 17.

Call of Pure Virtual Function: OOP

This check detects a pure virtual function call.

C++ Example

```
1
2     class Form
3     {
4     public:
5       Form(Form* f){};
6       Form(Form* f, char* title){
7         f->draw();  // OOP Error: [call to pure virtual  \
function draw()]
8       };
9       virtual void draw() = 0;
10    };
11
12    class Rectangle : public Form
13    {
14    public:
15      Rectangle(): Form (this, "Rectangle"){} ;
16      void draw();
17    };
18
19    void Rectangle::draw () {
20      Form::draw(); // Draw the rectangle
21    };
```

```
22
23    void main (void)
24    {
25      Rectangle Rect1;
26      Rect1.draw();
27    }
```

Explanation

The effect of making a virtual call to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined (see Standard ANSI ISO/IEC 1998 pp. 199).

One *Rectangle* object is declared: *Rect1* calls the constructor (line 15), and so the *Form* constructor (line 6) whose the *draw()* function member is called. Unfortunately, this function is a pure virtual function. Polyspace verification points out a warning at line 7.

Incorrect Type for this-pointer: OOP

Check to verify that a member function is associated to the right instance of a class.

Three principal causes lead to an incorrect this-pointer type:

- An out of bounds pointer access
- A non initialized variable member
- An inadequate cast.

The following example shows the three possible cases.

C++ Example

```
1     #include <new>
2
3     int get_random_value(void);
4
5     struct A {
```

```
6       virtual int f();
7     };
8
9     struct C {
10      virtual int h() { return 7; }
11    };
12
13    void f(void) {
14      struct T {
15        int m_j;
16        C m_field;
17        T() : m_j(m_field.h()) {} // OOP ERROR (initialisation):   \
[incorrect this-pointer type of T]
18      } badInit;
19      int r;
20
21
22      r = badInit.m_j;
23    }
24
25    class Bad
26    {
27    public:
28      int i;
29      void f();
30      Bad() : i(0) {}
31    };
32
33
34    class Good
35    {
36    public:
37      virtual void g() {}
38      void h() {}
39      static void k() {}
40    };
41
42    int main()
43    {
44
```

```
45      A* a = new A;
46      Good *ptr = (Good *)(void *)(new Bad);
47
48      a->f();             // OOP Verified: [this-pointer type of   \
A is correct]
49
50      if (get_random_value()) {
51        C* c = new C;
52        ++c;
53        c->h();           // OOP ERROR (out of bounds):      \
[incorrect this-pointer type of C]
54      }
55
56      if (get_random_value()) ptr->g(); // OOP ERROR (cast):    \
[incorrect this-pointer type of Bad]
57      if (get_random_value()) ptr->h(); // OOP ERROR (cast):    \
[incorrect this-pointer type of Bad]
58
59      ptr->k();  // correct call to a static function
60
61      f();
62
63    }
```

Explanation

At line 17 of the example, Polyspace verification identifies a this-pointer type problem (OOP category), because of an initialization missing for member field *m_field*.

At line 53 of the example, Polyspace verification points out that even if the function member *h* is part of the c Class, we are outside the structure. It could be compared to IDP for simple class.

Finally, lines 56 and 57 show another this-pointer problems: function members g and h are not part of the *Bad* Class. *Good* does not inherited from *Bad*. Note that there is no problem with static function member *k* because it is only syntaxic.

NTC – Non-Termination of Call

In this section...

Non Termination of Calls and Loops: Informative Checks

NTC and NTL are informative red checks.

- They are the only red checks which can be filtered out, as shown below
- They do not stop the verification
- As with other red checks, code found after them are gray (unreachable)
- These checks can only be red. There are no orange NTL or NTC checks.
- They can reveal a bug, or can simply just be informative

Check	Description
NTL	In a Non Terminating Loop, the break condition is never met. Here are some examples. • `while(1) { function_call(); }` Informative NTL. • `while(x>=0) {x++; }` Where x is an unsigned int. This may reveal a bug. • `for(i=0; i<=10; i++) my_array[i] = 10;` Where “`int my_array[10];`” applies. This red NTL reveals a bug in the array access, flagged in orange. • `ptr = NULL; for(i=0; i<=100; i++)*ptr=0;` The first iteration of the loop is red, and therefore it is flagged as an NTL. The “`i++`” will be gray, because the first iteration crashed.
NTC	Suppose that a function calls `f()`, and one of the following applies to `f`:

Check	Description
	• `f` contains a red error. • `f` contains an NTL. • `f` contains an NTC. • `f` contains an orange check that is context dependent — either red or green. In addition, the call makes `f` fail. • `f` is a mathematical function, for example, `sqrt` or `acos`, that has an invalid input parameter. Verification generates a red NTC check or applies dashed, red underlining to the function call. For the function call with dashed, red underlining, the tooltip indicates the location of the actual error.

Note A `sqrt` check is only colored if the input parameter is **never** valid. For instance, if the variable `x` may take any value between -5 and 5, then `sqrt(x)` has no color.

The list of constraints which cannot be satisfied (found by clicking on the NTC check) represents the variables that cause the red error inside the function. The (potentially) long list of variables can help to understand the cause of the red NTC, as it shows each condition causing the NTC

- where the variable has a given value; and
- where the variable is not initialized. (Perhaps the variable is initialized outside the set of files under verification).

If know a function is not expected to terminate (such as a loop or an exit procedure), you can specify the known-NTC (k-NTC) filter is an option. You will find all the NTCs and their consequences in the k-NTC facility in the Results Manager perspective, allowing you to filter them.

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 caller.

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

C++ Example

```
1     int cte; // Initialised by default to zero
2
3     int foo(int a) {
4       return 2%a;  // Modulo operation is orange, one call of foo is green, and another is red
5     }
6
7     int main(void) {
8
9        return foo(7) + foo(cte); // NTC on rightmost call, it passes cte=0
10
11   }
```

Explanation

In this example, the red NTC check on line 9 is due to the operation on line 4. The call `foo(cte)` passes the value zero, resulting in an undefined expression `2%0`.

NTL – Non Termination of Loop

In this section...

Non Termination of Loop: NTL

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

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

C++ Example

```
1
2  class NTL {
3  public:
4   NTL();
5   void rte_loop(void);
6   void task (void);
7   void update_alpha(double *a);
8   void send_data(double a);
9  };
10
11     static volatile double _acq =0.0;
12  static volatile int start_ = 0;
13
14
15 typedef void (NTL::*ptask) ();
16
17 extern void launch(ptask);
18
19
20  void NTL::task(void)
21   {
```

```
22    double acq, filtered_acq, alpha;
23
24  // Init
25  filtered_acq = 0.0;
26  alpha = 0.85;
27
28  while (1) {      // NTL ERROR: [non termination of loop]
29       //  Acquisition
30   acq = _acq;
31   // Treatment
32   filtered_acq = acq + (1.0 - alpha) * filtered_acq;
33   // Action
34   send_data(filtered_acq);
35   update_alpha(&alpha);
36  }
37 }
38
39 void NTL::rte_loop(void)
40 {
41    int i;
42  double twentyFloat[20];
43
44  for (i = 0; i <= 20; i++) {   // NTL ERROR: propagation  \
 of OBAI ERROR
45   twentyFloat[i] = 0.0;       // OBAI Warning: 20    \
 verification with i in [0,19]
46                                     // and one ERROR with i = 20
47  }
48 }
49
50 NTL::NTL()
51 {
52
53  ptask mytask = &NTL::task;
54  if (start_)
55   launch(mytask);
45 }
```

Explanation

In the example at line 19, the "continuation condition" is always true and the loop will never exit. Thus Polyspace verification will raise an error. In some case, the condition is not trivial and may depend on some program variables. Nevertheless the verification is still able to analyze those cases.

On the other error at line 35, the red OBAI related to the **21th** execution of the loop has been transformed in an orange warning because of the 20 first verified executions.

Tooltips for NTL Checks

Tooltips provide range information in the Results Manager perspective, including the number of iterations for loops.

There are 2 possible situations:

- **Loops that terminate** – A tooltip gives the number of iterations of the loop. For example, `for (i=0; i<10; i++)`, a tooltip on the `for` keyword says `Number of iteration(s): 10`.
- **Non–terminating loops** — The NTL check contains information about the maximum number of iterations that can be done. This number is an overset of the real number of iterations (which may be lower).

  For example:

  - **Failure at a given iteration**, `for (i=0; i<10; i++) y = 2 / (i - 5);` — The NTL check on the `for` keyword says: `Number of iteration(s): 6`

    This means that the loop fails at the 6th iteration, which can help you find the orange check that contains the failure.

  - **Infinite loop** `x = 0; while (x >= 0) y = 2;` — The NTL check on the `for` keyword says: `Number of iteration(s): 0..?`

    This means that the loop has an unknown number of iterations (up to an infinite number). It does not mean that the loop *is* an infinite loop, but that it *may* be an infinite loop. You would also get `0..?` on the loop `while (1) { if (random) break; }`.

ABS_ADDR – Absolute Address

The software generates an orange ABS_ADDR check when an absolute address is assigned to a pointer. The check is colored orange because the software has no information about the absolute address and cannot verify, for example, the address, availability of memory, and initialization of memory.

The software permits memory access to the absolute address after generating the orange ABS_ADDR check for the first assignment operation. IDP and NIV checks for memory access operations after the first assignment operation are green.

Consider the following code.

```
27   int *p;
28   int x;
29
30   p = (int *)0x32;   // Orange ABS_ADDR
31   x = *p;            // Green IDP and NIV
32
33   p++;
34   y = *p;            // Orange IDP and NIV
35
```

On line 30, the first assignment of the absolute address to a pointer produces an orange ABS_ADDR check. The next memory access operation produces green IDP and and NIV checks.

On line 34, the memory access operation produces orange IDP and NIV checks. The checks are orange because the accessed memory location is not covered by an orange ABS_ADDR check.

Note By default, the software displays ABS_ADDR checks on the **Results Explorer** or **Results Summary** tab only if you set the Review Level slider to `All`.

If you know that the absolute addresses in your code are valid, you can specify the option `-green-absolute-address-checks`, which makes all ABS_ADDR checks green. See “Green absolute address checks” on page 2-58.

INF – Potential Call

INF checks (potential call to) are informative checks that help to understand reasoning of Polyspace verification during function calls, constructions and destructions of objects through

C++ Example

```
1     #include <iostream>
2     static volatile int random_int = 1 ;
3
4     typedef enum { AOP, UTC, GET } valueKind;
5
6     class SubVal {
7       valueKind val;
8       void init();
9     public:
10      SubVal(valueKind v);
11      virtual ~SubVal() {}     // INF informative:   \
 [operator_delete(void*) is implicitly called]
12
13      virtual void log(const char* msg);
14      valueKind getVal() {return val;};
15      void undef();
16    };
17
18    SubVal::SubVal(valueKind v) : val(v) {
19      init();
20    }
21
22    void SubVal::init() {
23      log("SubVal creation");          // INF informative:   \
 [SubVal::log(const_char*) is called during construction of SubVal]
24    }
25
26    void SubVal::log(const char* msg) {
27      cout << msg;
28    }
29
```

```
30    void SubVal::undef() {
31      log("ArithVal destruction");   // INF informative:   \
 [ArithVal::log(const_char*) is called if this-pointer is of type  \
 ArithVal]
32    }
33
34    class ArithVal : SubVal {
35    public:
36      ArithVal(double d) : SubVal(GET) {}
37      ~ArithVal();
38      void ArithAdd(double d) {};
39      virtual void log(const char* msg) {
40        cout << getVal();
41      };
42    };
43
44    ArithVal::~ArithVal() {
45      undef();
46    }  // INF informative: [SubVal::~SubVal() is implicitly called]
47
48
49
50    void main(void){
51      ArithVal *xVal = new ArithVal(10.0);
52      xVal->ArithAdd(1.0);
53
54      SubVal *eVal = new SubVal(AOP);
55      eVal->log("new");          // INF informative:   \
 [SubVal::log(const_char*) is called if this-pointer is of type   \
 SubVal]
56
57      delete xVal;          // INF informative:   \
 [ArithVal::~ArithVal() is called if this-pointer is of type    \
 ArithVal]

58      delete eVal;        // INF informative:   \
 [SubVal::~SubVal() is called if this-pointer is of type SubVal]
59
60    }
```

Explanation

In this example, a base and derived classes are described. From main program, we create objects, call member functions and delete them. Associated to each function call, including constructors and destructors, some informative checks are put giving (potential) call of functions, during construction and destruction of objects.

Theses checks can only be green or gray.

POW (Deprecated)

Note The POW check is deprecated in R2009a and later. The POW check no longer appears in Polyspace results.

The pow function is now a standard stub, and the POW check has been replaced by a function call and an NTC error when the power is negative.

Check to establish whether the left operand of the *pow* mathematical function declared in <math.h> is positive (directly or in generated constructors or destructors)

UNFL (Deprecated)

Note The UNFL check is deprecated in R2010a and later. The UNFL check no longer appears in Polyspace results. Instead of two separate UNFL and OVFL checks, a single OVFL check now appears.

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

UOVFL (Deprecated)

Note The UOVFL check is deprecated in R2009a and later. The UOVFL check no longer appears in Polyspace results. Instead of a single UOVFL check, the results now display two checks, a UNFL and an OVFL.

The check UOVFL only concerns float variables. Polyspace verification shows an UOVFL when both overflow and underflow can occur on the same operation.

5

Approximations Used During Verification

Why Polyspace Verification Uses Approximations

In this section...

What is Static Verification

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

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

For example, consider the following code:

```
for (i=0 ; i<1000 ; ++i)
{    tab[i] = foo(i);
}
```

To check that the variable `i` never overflows the range of `tab` a traditional approach would be to enumerate each possible value of `i`. One thousand checks would be required.

Using the static verification approach, the variable `i` is modelled by its variation domain. For instance the model of `i` is that it belongs to the [0..999] static interval. (Depending on the complexity of the data, convex polyhedrons, integer lattices and more elaborated models are also used for this purpose).

Any approximation leads by definition to information loss. For instance, the information that `i` is incremented by one every cycle in the loop is lost. However, the important fact is that this information is not required to ensure that no range error will occur; it is only necessary to prove that the variation domain of `i` is smaller than the range of `tab`. Only one check is required

to establish that – and hence the gain in efficiency compared to traditional approaches.

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

Exhaustiveness

Nothing is lost in terms of exhaustiveness. The reason is that Polyspace verification works by performing upper approximations. In other words, the computed variation domain of any program variable is always a superset of its actual variation domain. The direct consequence is that no runtime error (RTE) item to be checked can be missed by Polyspace verification.

Approximations Made by Polyspace Verification

In this section...

Volatile Variables

Volatile variables are potentially uninitialized and their content is always full range.

```
2 int volatile_test (void)
3 {
4  volatile int tmp;
5  return(tmp); // NIV orange: the variable content is full range
[-2^31;2^31-1]
6 }
```

In the case of a global variable the content would also be full range, but the NIV check would be green.

Structures with Volatile Fields

In this example, although only the b field is declared as volatile, in practice any read access to the "a" field will be full range and orange.

```
2 typedef struct {
3  int a;
4  volatile int b;
5 } Vol_Struct;
```

Absolute Addresses

Both reading from, and writing to, an absolute address leads to warning checks on the pointer dereference. An absolute address is considered as a volatile variable.

```
Val = *((char *) 0x0F00); // NIV and IDP orange: access to an
absolute address
```

Pointer Comparison

Polyspace verification is a static tool verifying source code. Memory management concerns dynamic considerations, and the characteristics of particular compilers and targets. Polyspace verification therefore doesn't consider where objects are actually implanted in memory

```
5  int *i, *j, k;
6  i = (int *) 0x0F00;
7  j = (int *) 0x0FF0;
8
9  if ( i < j) // the condition can be true or false
10   k = 12; // this line is reachable
11  else
12   k = 23; // this line is reachable too.
```

Its the same situation if "i" and "j" points to real variable

```
6  i = & one_variable;
7  j = & another_one;
9  if ( i < j) // the condition can still be true or false
```

Shared Variables

At the minimum, a shared variable contains a union of all ranges it can contain among the application. At the maximum, the variable will be full range.

```
12 void p_task1(void)
13 {
14  begin_cs();
15  X = 0;
16  if (X) {
17   Y = X;      // Verified NIV, although it should be gray
```

```
18   assert (Y == 12);  // Warning assert, although it should be gray
19  }
20  end_cs();
21 }
22
23 void p_task2(void)
24 {
25  begin_cs();
26  X = 12;
27  Y = X + 1;       // Polyspace considers [Y==1] or [Y==13]
28  if (Y == 13)
29   Y = 14;
30  else
31   Y = X - 1 ;     // this line should be gray
32  end_cs();
33 }
```

Trigonometric Functions

With trigonometric functions, such as sines and cosines, verification sometimes assumes that the return value is bound between the limits of that function, regardless of the parameter passed to it. Consider the following example, which uses acos, sin and asin functions.

```
7  double res;
8
9  res = sin(3.141592654);
10 assert(res == 0.0); // Range is [-1..1]
11
12 res = acos(0.0);
13 assert(res == 0.0); // Range always in [0..pi]
14
15 res = asin(0.0);
16 assert(res == 0.0); // Always gives [0.0]
```

Unions

In some situations, unions can help you construct efficient code. However, unions can cause issues for code verification, for example:

- **Padding** – Padding might be inserted at the end of an union.

- **Alignment** – Members of structures within a union might have different alignments.
- **Endianness** – Whether the most significant byte of a word could be stored at the lowest or highest memory address.
- **Bit-order** – Bits within bytes could have both different numbering and allocation to bit fields.

These issues can cause Polyspace verification to lose precision when structure unions are considered. In fact, these kinds of implementation are compiler dependant. Conversions from one type a union to another will cause a loss of precision on two checks:

- Is the other field initialized? Orange NIV
- What is the content of the other field? Full range

```
typedef union _u {
int a;
char b[4]; } my_union;
my_union X;

X.b[0] = 1; X.b[1] = 1; X.b[2] = 1; X.b[1] = 1;
if (X.A == 0x1111)
else // both branches are reachable
```

Constant Pointer

To increase Polyspace precision where pointers are analyzed, replace

```
const int *p = &y;
```

with:

```
#define p (&y)
```

Limitations of Polyspace Verification

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

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

Polyspace_Common\R2012b\help\pdf\polyspace_limitations.pdf

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

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

6

Examples

Complete Examples

In this section...

Simple C Example

```
polyspace-c \
 -prog myCproject \
 -O1 \
 -I /home/user/includes \
 -D SUN4 -D USE_FILES \
```

Apache Example

Here is a script for verifying the code for Apache (after formatting). The source code is in C and the compilation is for a Sun™.

Note The use of O0 to reduce verification time.

```
polyspace-c \ \
 -target sparc \
 -prog Apache \
 -keep-all-files \
 -allow-undef-variables \
 -continue-with-red-error \
 -OO \
 -D PST \
 -D __GNUC_MINOR__=6 -D SOLARIS2=270 -D USE_EXPAT \
 -D NO_DL_NEEDED \
```

```
 -I sources \
 -I /usr/local/pst/include.sparc \
 -I /usr/include \
 -results-dir RESULTS
```

cxref Example

Here is another C launch command. The compilation is for Linux. Note the escape characters, allowing quoted strings to be used as compiler defines.

```
polyspace-c \
 -OS-target linux \
 -prog cxref \
 -OO \
 -I `pwd` \
 -I sources \
 -I <Polyspace_Install>/include/include.linux \
 -D CXREF_CPP='\"/usr/local/gcc/bin/cpp\"' \
 -D PAGE='\"A4\"' \
 -results-dir RESULTS
```

T31 Example

Another simple C launcher. There are a couple of tasks and compilation is for an m68k.

```
polyspace-c \
 -target m68k \
 -entry-points task_callback_main,task_tcp_main,cdtask_depm_main,
task_receiver \
 -to pass1 \
 -prog T31 \
 -OO \
 -results-dir `pwd`/RESULTS_31 \
 -keep-all-files
```

Dishwasher1 Example

Another C example. This one is for the c-167 and has tasks protected by critical section.

```
polyspace-c \
 -target c-167 \
```

```
 -entry-points periodic,pst_main \
 -D PST -D const= -D water= \
 -from scratch \
 -to pass4 \
 -critical-section-begin "critical_enter:cs1" \
 -critical-section-end "critical_exit:cs1" \
 -prog dishwasher1 \
 -I `pwd`/sources \
 -OO \
 -keep-all-files \
 -results-dir RESULTS
```

Satellite Example

A C example with tasks and critical sections.

```
polyspace-c
 -target c-167 \
 -entry-points ctask0,ctask1,ctask2,ctask3,interrupts \
 -O2 \
 -keep-all-files \
 -from scratch \
 -critical-section-begin "DisableInterrupts:sc1" \
 -critical-section-end "EnableInterrupts:sc1" \
 -ignore-constant-overflows \
 -include `pwd`/sources/options.h \
 -to pass4 \
 -prog satellite \
 -I `pwd`/sources \
 -results-dir RESULTS
```