PolySpace® Products for C++ 7
Reference

The MathWorks™

Accelerating the pace of engineering and science
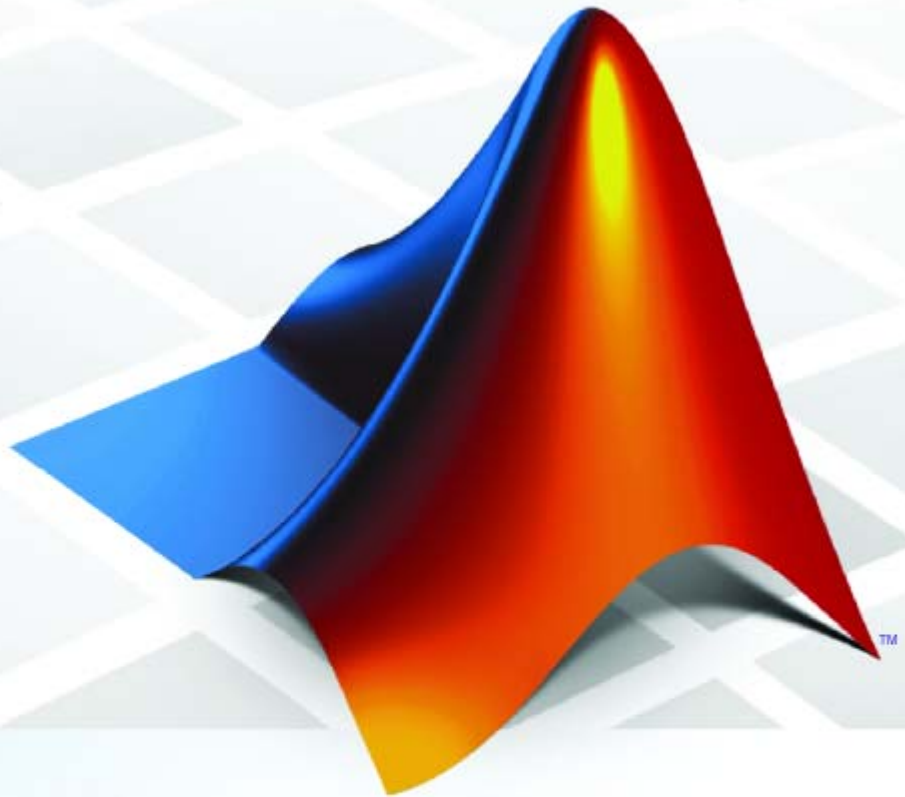
How to Contact The MathWorks

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++ 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

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

Revision History

Contents

Options Description

1

Check Descriptions

2

Approximations Used During Verification

3

1

Options Description

- "Overview" on page 1-2
- "Sources/Includes" on page 1-3
- "General" on page 1-6
- "Targets/Compilers" on page 1-12
- "Compliance with Standards" on page 1-24
- "PolySpace Inner Settings" on page 1-36
- "Precision/Scaling" on page 1-52
- "MultiTasking (PolySpace Server for C/C++ Only)" on page 1-63
- "Specific Batch Options" on page 1-66

Overview

This chapter describes all options available within PolySpace® software. All of these options with the exception of the multitasking options are accessible through the graphical user interface of the PolySpace Launcher.

They can also be accessed with the associated batch command `polyspace-cpp`.

Sources/Includes

-results-dir Results_Directory

This option specifies the directory 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 directory 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 ...
```

-sources files or -sources-list-file file_name

-sources "file1[file2[...]]" (Linux® and Solaris™)

or

-sources "file1[,file2[, ...]]" (Windows®, Linux and Solaris)

or

-sources-list-file file_name (not a graphical option)

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

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

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

-I directory

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:

- When no directory is specified using this option, the ./sources directory (if it exists) is automatically included

- If several include-dir are mentioned, the ./sources directory (if it exists), is implicitly added at the end of the "-I" list

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
```

General

Overview

This section collates all options relating to the identification of the verification, including the destination directory for the results and sources.

-prog Session identifier

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 *Session Identifier*.

Default:

Shell Script: polyspace

GUI: New_Project

Example shell script entry:

```
polyspace-cpp -prog myApp ...
```

-date 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.

Default:

  Day of launching the verification

Example shell script entry:

```
polyspace-cpp -date "02/01/2002"...
```

-author Author

This option is used to specify the name of the author of the verification.

Default:

  the name of the author is the result of the *whoami* command

Example shell script entry:

```
polyspace-cpp -author "John Tester"
```

-verif-version 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*.

Default:

  1.0.

Example shell script entry:

```
polyspace-cpp -verif-version 1.3 ...
```

-keep-all-files

When this option is set, all intermediate results and associated working files are retained. Consequently, it is possible to 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.

By default, intermediate results and associated working files are erased when they are no longer needed by the software.

This option is applicable only to client verifications. Intermediate results are always removed before results are downloaded from the PolySpace server.

> **Note** To cleanup intermediate files at a later time, you can select **Tools > Clean Results** in the Launcher.
>
> This options deletes the preliminary results files from the results directory.

Default:

  Disabled.

Example shell script entry:

```
polyspace-cpp -keep-all-files
```

-continue-with-existing-host

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.

Default:

  PolySpace verification stops when the host configuration is incorrect or the system is under specified.

Example Shell Script Entry:

```
polyspace-cpp -continue-with-existing-host ...
```

-allow-unsupported-linux

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" in the *PolySpace Installation Guide*.

For all other Linux distributions, you may be able to verify code using the -allow-unsupported-linux option, but a warning will be displayed in the log file informing you of possible incorrect behaviors:

```
*************************************************** ***
***                        ***
***          WARNING        ***
***                        ***
***   You are running PolySpace on an   ***
***   unsupported Linux distribution. It may lead  ***
***   to incorrect behaviour of the product. Please ***
***   note that no support will be available for  ***
***   this operating system.          ***
***                        ***
*************************************************** ** ***
```

Default:

Disabled

Example Shell Script Entry:

```
polyspace-cpp  allow-unsupported-linux ...
```

Report Generation

When this option is selected, PolySpace software creates a report for the verification, using the following options:

-report-template Report_Template_Name

Generates a report for the verification, using the specified report template name. The report is generated at the end of the verification process, before any `post-analysis-command` is executed.

Default:

```
C:\PolySpace\PolySpace_Common\ReportGenerator\templates\Developer.rpt
```

Example Shell Script Entry:

```
polyspace-cpp -report-template c:/polyspace/my_template
```

-report-output-format Output_Format

Specifies the output format for the report specified by the `report-template` option. The argument is not case sensitive.

Valid options are:

- HTML
- PDF
- RTF
- WORD
- XML

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

Default:

If you do not specify an output format, RTF is used by default.

Example Shell Script Entry:

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

-report-output-name Name

Specifies the name of the report file that is generated for the verification.

Default:

If you do not specify a name, the following name is used by default:

```
  -Prog_TemplateName.Format
```

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

Example Shell Script Entry:

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

Targets/Compilers

-target TargetProcessorType

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: sparc, m68k, powerpc, i386, c-167, mcpu, or PST Generic target.

mcpu is a reconfigurable Micro Controller/Processor Unit target. One or more generic targets can also be specified and saved. In addition, 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 "Setting Up a Target" in the *PolySpace Products for C++ User's Guide* for more details.

For information on specifying a generic target, or modifying the mcpu target, see "GENERIC ADVANCED TARGET OPTIONS" on page 1-13.

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

Default:

sparc

Example shell script entry:

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

GENERIC ADVANCED 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, it is necessary to use the GUI 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,]*
- *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
```

1-13

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-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 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 ...
```

-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
```

-OS-target OperatingSystemTarget

This option specifies the operating system target for PolySpace stubs.

Possible values are 'Solaris', 'Linux', 'VxWorks', 'Visual' and 'no-predefined-OS'.

This information allows the appropriate system definitions to be used during preprocessing in order to analyze the included files properly. *-OS-target* no-predefined-OS may be used 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:

Solaris

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 it is necessary to 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  ...
```

-D compiler-flag

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 ...
```

-U compiler-flag

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 ...
```

-include file1[,file2[,...]]

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 ...
```

-post-preprocessing-command "command"

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 on each preprocessed c files. The command should be designed to process the standard output from pre-processing and produce its results in accordance with that standard output.

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 :

```perl
#!/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
```

-post-analysis-command <file_name> or "command"

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 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 an appropriate 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_emails
```

Compliance with Standards

-dos

This option must be used when the contents of the **include** or **source** directory comes from a DOS or Windows file system. It deals with upper/lower case sensitivity and control characters issues.

Concerned files are:

• header files: all include dir specified (-I option)

- source files: all sources files selected for the verification (-sources option)

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

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

into

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

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

Default:

     disabled by default

Example Shell Script Entry:

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

Embedded Assembler

PolySpace stops the execution when detecting assembler code and displays an error message. It can continue the execution if it is requested by the user with the option –discard-asm.

PolySpace ignores the assembler code by assuming that the assembler code does not have any side effect on global variables. Delimiters for assembler code to ignore can be recognized by PolySpace following C++ standard specified asm declarations: __asm and __asm__.

-discard-asm

This option instructs the PolySpace verification to discard assembler code. If this option is used, the assembler code should be modelled in c.

Default:

Embedded assembler is treated as an error.

Example Shell Script Entry:

```
polyspace-cpp -discard-asm ...
```

-wchar-t-is-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 ...
```

-size-t-is-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 ...`

-no-extern-C

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 ...
```

-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 ...
```

-dialect DialectName

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

`default`, `cfront2`, `cfront3`, `iso`, `gnu`, `visual`, `visual6`, `visual7.0`, `visual7.1`, and `visual8`.

`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` and `visual8`) the -OS-target option must be set to `Visual`.

If the dialect is visual, the options `-dos`, `-OS-target Visual` and `-discard-asm` are 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:

default

Example Shell Script Entry:

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

-wchar-t-is

This option forces wchar_t to be treated as a keyword as per the C++ standard or as a typedef as with Microsoft Visual C++ 6.0/7.x dialects.

Possible values are 'keyword' or 'typedef':

- typedef is the default behavior when using -dialect option associated to visual6, visual7.0 and visual7.1.

- keyword is the 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++ /Zc:wchar and /Zc:wchar- options.

Default:

default (depends on -dialect value).

Example in shell script:

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

-for-loop-index-scope

This option changes the scope of the index variable declared within a for loop.

Example:

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

Possible values are 'in' and 'out':

- out is the default for the -dialect option associated with values cfront2, crfront3, visual6, visual7 and visual 7.1.

- in is the default for all other dialects, including visual8.

The C++ ANSI standard specifies the index be 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:

default (depends on –dialect value)

Example in shell script:

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

-ignore-pragma-pack

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 will stop the execution and display 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" in the *PolySpace Products for C++ User's Guide*.

Example Shell Script Entry:

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

Visual Specific Options

- "-import-dir directory" on page 1-30

- "-pack-alignment-value value" on page 1-30

- "-support-FX-option-results" on page 1-30

-import-dir directory

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 ...
```

-pack-alignment-value 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 ...
```

-support-FX-option-results

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 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 directory as the original ones and PolySpace 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 will stop the execution and display 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
```

JSF-C++: 2005 Rules

- "-jsf-coding-rules [all-rules | file_name]" on page 1-31
- "-includes-to-ignore "dir_or_file_path1[,dir_or_file_path2[,...]]"" on page 1-32

-jsf-coding-rules [all-rules | file_name]

This option allows PolySpace sofware to check 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.

- Keyword *all-rules* – Checks all available JSF++ rules using the default configuration. Any violation of JSF++ rules is considered a warning.
- Option *file_name* – The name of an ASCII text file containing a list of JSF++ rules to check.

> **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
```

-includes-to-ignore "dir_or_file_path1[,dir_or_file_path2[,...]]"

This option prevents JSF++ rules checking in a given list of files or directories (all files and subdirectories under selected directory). This option is useful when non-JSF++ conforming include headers are used. A warning is displayed if one of the files does not exist.

This option is enabled only when you specify `-jsf-coding-rules`.

Example shell script entry :

```
 polyspace-cpp -jsf-coding-rules jsf.txt  includes-to-ignore
"c:\usr\include"
```

-ignore-constant-overflows

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-undef-variables

When this option is used, PolySpace will continue in case of linkage errors due to undefined global variables. For instance when this option is used, PolySpace will tolerate a variable always being declared as extern

Default:

Undefined variables causes PolySpace to stop.

Example Shell Script Entry:

```
polyspace-cpp -allow-undef-variables ...
```

-allow-negative-operand-in-shift

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 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 ...
```

-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 ..
```

PolySpace Inner Settings

In this section...
"-unit-by-unit" on page 1-36
"-unit-by-unit-common-source *filename*" on page 1-37
"-main sub_program_name" on page 1-37
"Generate a Main Using a Given Class" on page 1-38
"-main-generator-calls" on page 1-42
"General options for the generation of mains" on page 1-43
"-data-range-specifications file_name" on page 1-45
"-no-automatic-stubbing" on page 1-45
"-ignore-float-rounding" on page 1-46
"-detect-unsigned-overflows" on page 1-48
"-machine-architecture" on page 1-49
"-max-processes" on page 1-49
"-extra-flags option-extra-flag" on page 1-50
"-cpp-extra-flags flag" on page 1-50
"-il-extra-flags flag" on page 1-51

-unit-by-unit

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.

> **Note** When using this option in class-analyzer mode, out of class functions are not analyzed if they are not called by some method in the analyzed class.

Default:

Not selected

Example Shell Script Entry:

```
polyspace-cpp -unit-by-unit
```

-unit-by-unit-common-source *filename*

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
```

-main sub_program_name

The option specifies the qualified name of the main subprogram when a visual –OS-target is selected. 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:

main, _tmain, wmain, _tWinMain, wWinMain, WinMain and DllMain.

However, if the main subprogram does not exist and the option -main-generator is not set, PolySpace will stop the verification with an error message.

Default:

  main

Example Shell script entry:

```
polyspace-cpp -main WinMain  OS-target visual
```

Generate a Main Using a Given Class

- "-class-analyzer" on page 1-38
- "-class-only" on page 1-39
- "-class-analyzer-calls" on page 1-39
- "-no-constructors-init-check" on page 1-40

-class-analyzer

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.

If *MyclassName* does not exist in the application, the verification will come to a halt. 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"in the *PolySpace Products for C++ User's Guide*).

> **Note** This option cannot be used with the option
> `-function-called-before-main`.

Example shell script entry:

```
polyspace-cpp  class-analyzer MyClass
polyspace-cpp  class-analyzer MyNamespace::MyClass
```

-class-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 MyClass
are verified. All functions out of class scope are automatically stubbed even
though they are defined in the source code.

Default:

disable

Example Shell Script Entry:

```
polyspace-cpp  class-analyzer MyClass  class-only...
```

-class-analyzer-calls

Specifies which functions are called by the generated main.

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.

There are three options:

- **Default** – The generated main calls all public and protected function
  members declared within the class, whether called in the code or not. Every
  eligible function is called directly by the generated main.

- **Unused** – The generated main calls only those functions not called by another eligible function. Members inherited from a base class are not verified.

- **Inherited** – The generated main calls the functions of the named class as well as functions inherited from the base class that are not called within the analyzed class. Inherited methods are called in the child context, meaning that the generated main does not make explicit calls to the parent's constructor and destructor.

Note If the hierarchy contains the same class more than once, only the first instance of the class is analyzed, and the software displays a warning message.

Eligible functions are:

- Public and protected methods of analyzed class

- Existing constructors of analyzed class

- Destructor of analyzed class

- Existing copy constructors of analyzed class

Default:

Default

Example Shell Script Entry:

```
polyspace-cpp  class-analyzer MyClass  class-analyzer-calls
unused ...
```

-no-constructors-init-check

By default, PolySpace 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
                // grey 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 — grey

- `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
...
```

-main-generator-calls

This option is used with the -main-generator option, to specify the functions to be called.

Note that this option is protected by a license.

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 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 stops and displays an error message. This error message is put in the log file.

Default 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-cpp -main-generator -main-generator-calls
custom=function_1,function_2
```

General options for the generation of mains

- "-function-called-before-main" on page 1-43
- "-main-generator-writes-variables" on page 1-44

-function-called-before-main

This option is used with the main generator option –main-generator-calls to specify a function which will be called before all selected functions in the main.

Note This option cannot be used with the option class-analyzer.

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 stops and displays an error message.

> **Note** If the function name you provide is ambiguous (there is another
> function of the same name in another class) the verification stops and displays
> an error message listing the specific names of all possible functions. You can
> avoid this error by copying the correct name from the error message and
> enclosing it with double quotes..

Example:

```
polyspace-cpp -main-generator-calls unused
-function-called-before-main MyFunction
```

-main-generator-writes-variables

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
```

-data-range-specifications file_name

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

For more information, see "Applying Data Ranges to External Variables and Stub Functions (DRS)"in the *PolySpace Products for C++ User's Guide*.

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 concern external linkage, const variables and not necessary a defined variable (i.e. 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-c -data-range-specifications range.txt ...
```

-no-automatic-stubbing

By default, PolySpace 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

-ignore-float-rounding

Without this option, PolySpace 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 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/UNFL 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 ...
```

-detect-unsigned-overflows

When this option is selected, verification is more strict with overflowing computations on unsigned integers than the ANSI® C standard requires.

The ANSI C standard states that promotion occurs for logic, bitwise and arithmetic operators. For `char`, `short`, and `int` types, variables are implicitly cast into integers before the operation. Then, after the operation, the variables are downcast into the original type.

Consider the examples below.

Example 1

Using this option, the following example generates an error:

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

Without this 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 this 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: 000000001

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 -detect-unsigned-overflows ...
```

-machine-architecture

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
```

-max-processes

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
```

-extra-flags option-extra-flag

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 PolySpace Support as necessary for your verifications.

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 PolySpace 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 PolySpace support as necessary.

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 PolySpace 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 PolySpace support as necessary.

Default:

no extra flags.

Example Shell Script Entry:

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

Precision/Scaling

-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 .

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

-O(0-3)

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.

The MathWorks recommends you begin with the lowest precision level. Red errors and gray code can then be addressed before relaunching PolySpace verification using 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 ...
```

-from verification-phase

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 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 ...`

-to verification-phase

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" (Reaches the compilation phase)

- `cpp-normalize` or "C++ source normalization" (Reaches the normalization phase)

- `cpp-link` or "C++ Link" (Reaches the link phase)

- `cpp-to-il` or "C++ to Intermediate Language" (Reaches the transformation to intermediate language)

- `pass0` or "Software Safety Analysis level 0"

- `pass1` or "Software Safety Analysis level 1"

- `pass2` or "Software Safety Analysis level 2"

- `pass3` or "Software Safety Analysis level 3"

- pass4 or "Software Safety Analysis level 4"

- other (stop verification after level 20)

Note If you use *-to other* then PolySpace will continue until you stop it manually (via *"PolySpace Install Directory"/bin/kill-rte-kernel "Results directory"/"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 ...

-context-sensitivity "proc1[,proc2[,...]]"

This option allows the precise verification of a procedure with regards to the discrete calls to it in the analyzed code.

Each check inside the procedure is split into several sub-checks depending on the context of call. Therefore if a check is red for one call to the procedure and green for another, both colors will be revealed.

This option is especially useful is a problem function is called from a multitude of places.

-context-sensitivity-auto

This option is similar to the -context-sensitivity option, except that the system automatically chooses the procedures to be considered.

Usually, the ten functions which are the most called are automatically selected.

-path-sensitivity-delta number

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 ...
```

-k-limiting number

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 PolySpace 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.

-inline "proc1[,proc2[,...]]"

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  ...
```

-respect-types-in-globals

This is a scaling option, designed to help process complex code. When it is applied, PolySpace 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 will not take care that x contains the address of y resulting a green assert.

Default:

PolySpace assumes that global variables may contain pointer values.

Example Shell Script Entry:

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

-respect-types-in-fields

This is a scaling option, designed to help process complex code. When it is applied, PolySpace 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 will not take care that S1.x contains the address of y resulting a green assert.

Default:

PolySpace assumes that structure fields may contain pointer values.

Example Shell Script Entry:

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

-less-range-information

Limits the amount of range information displayed in verification results.

When you select this option, range information is provided on assignments, but not on reads and operators.

Scaling problems can occur when computing range information on reads and operators. In these cases, using this option can significantly increase the speed of the verification.

Default:

Disabled.

Example Shell Script Entry :

```
polyspace-cpp -less-range-information
```

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 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 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 might retain very detailed information about the data at a particular point in the code, so that

1-61

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 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 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 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.

MultiTasking (PolySpace Server for C/C++ Only)

Note Concurrency options are not compatible with -main-generator options.

-entry-points *str1[,str2[,...]]*

This option is used to specify the tasks/entry points to be analyzed by PolySpace, 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-[begin or end] "proc1:cs1[,proc2:cs2]"

```
-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.

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"
```

-temporal-exclusions-file file_name

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' directory 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 ...
```

Specific Batch Options

-server server_name_or_ip[:port_number]

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

Note If the option –server is not specified, the default server referenced in the PolySpace-Launcher.prf configuration file will be used as 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 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-list-file file_name

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"
```

-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
```

Check Descriptions

- "Check Categories" on page 2-2
- "Colored Source Code for C++" on page 2-9

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 to C++ specific constructions:" on page 2-2
"Acronym Not Related to C++ Constructions (Also Used for C Code):" on page 2-6

Acronyms associated to C++ specific 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
	INF		
	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
	OOP	Internal PolySpace error: please contact support	
	INF		
	INF		
	INF		

Category	Acronym	Red	Orange
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
	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

Category	Acronym	Green	Gray
Zero division	ZDV		Unreachable check:
Non-initialized variable	NIV local/other	[local] variable is initialized	Unreachable check:
scalar or float overflows	OVFL/UNFL		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
	COR		
	COR		
	COR		
Shift amount out of bounds	SHF	Scalar shift amount is within its bounds	Unreachable check: shift error
	SHF		
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

Category	Acronym	Red	Orange
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/UNFL		
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		wrong type for argument of call to function
	COR		wrong number of arguments for call to 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	
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		

Colored Source Code for C++

Function Returns a value: FRV

Check to establish whether on every value-returning function there is no flowing off the end the function.

C++ Example

```
1     static volatile int rand;
2
3     class function {
4     public:
5       function() { rep = 0; }
6       int reply(int msg) {     // FRV Verified: [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);  // NTC ERROR: propagation of FRV ERROR
32      else
33        f.reply3(rand);
34    }
```

Explanation

Variables are often initialized using the return value of functions. However it may occur that, as in the above example, the return value is not initialized for all input parameter values (which should always be the case). In this case, the target variable will not be properly initialized with a valid return.

Non Null this-pointer: NNT

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.

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

```cpp
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]; // PAS 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); // NTC ERROR:
propagation of PAS ERROR
28    }
```

Explanation

The property, the non-negative value of an array size, is checked at line 14, where the *array* is defined with the *[numberUser][1]* dimension. Unfortunately the *numberUser* variable is always negative as an opposite of an *unsigned short int* type. PolySpace detects a red error and displays a message.

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

```cpp
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

```cpp
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();
// NTC ERROR: propagation of dynamic_cast reference error
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 20, because *Object* is not a subclass of *Item*.

The verification stops at line 20 and the error is propagated to a NTC error at line 28. The behavior is different with a dynamic_cast on a pointer.

Invalid Pointer to Member: OOP

PolySpace 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 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 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 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.

Potential Call to: INF

[potential call to] are informative checks that help to understand reasoning of PolySpace 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() {}
12      virtual void log(const char* msg);
13      valueKind getVal() {return val;};
14      void undef();
15    };
16
17    SubVal::SubVal(valueKind v) : val(v) {
18      init();
19    }
20
21    void SubVal::init() {
22      log("SubVal creation");             // INF informative: [call
of log during construction]
23    }
24
25    void SubVal::log(const char* msg) {
26      cout << msg;
27    }
28
29    void SubVal::undef() {
30      log("ArithVal destruction");        // INF informative: [call
of log during destruction]
```

```
31      }
32
33      class ArithVal : SubVal {
34      public:
35        ArithVal(double d) : SubVal(GET) {}
36        ~ArithVal();
37        void ArithAdd(double d) {};
38        virtual void log(const char* msg) {
39          cout << getVal();
40        };
41      };
42
43      ArithVal::~ArithVal() {
44        undef();
45      }
46
47      void main(void){
48        ArithVal *xVal = new ArithVal(10.0);
49        xVal->ArithAdd(1.0);   // INF informative: [call to
        X::function()]
50
51        SubVal *eVal = new SubVal(AOP);
52        eVal->log("new");      // INF informative: [potential
        call to X::function()]
53
54        delete xVal;
55        delete eVal;
56      }
```

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.

Non-Initialized Variable: NIV/NIVL

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 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; };
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; // Unproven NIV:
index 0 is not 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 prompts the user 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*).

Non-Initialized Pointer: NIP

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 "*").

User Assertion Failure: ASRT

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 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 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.

Overflows and Underflows

- "Scalar and Float Overflows: OVFL" on page 2-29
- "Scalar and Float Underflows: UNFL" on page 2-32
- "Float Underflow and Overflow: UOVFL (Deprecated)" on page 2-34

Scalar and Float Overflows: OVFL

Check to establish whether an arithmetic expression overflows. 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: [scalar
variable overflows on [+] ...]
10          // 2^31 is an overflow value for int32
11        }
12        float overflow (float value){
13          return  2 * value + 1.0;   // OVFL ERROR: [float
variable overflows on [conversion from ...]]
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);           // NTC ERROR:
propagation of OVFL ERROR
28
29      if (random_int())
30         fvalue = c.overflow(fvalue);     // NTC ERROR:
propagation of OVFL ERROR
31    }
```

Explanation. On a 32-bits 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.

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 type given to a constant is the first type which can accommodate its value, from the appropriate sequence shown below. (See "Predefined Target Processor Specifications (size of char, int, float, double...)" in the *PolySpace Products for C++ User's Guide* for information about the size of a type depending on the target.)

Decimals	`int` , `long` , `unsigned long`
Hexadecimals	`Int`, `unsigned int`, `long`, `unsigned long`
Floats	`double`

For example (assuming 16-bits 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 (from -32768 to 32767).

Scalar and Float Underflows: UNFL

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

C++ Example.

```
1       #include <float.h>
2
3       extern int random_int(void);
4
5       class Calcul
6       {
7       public:
8         int makeUnderflow(int i){
9           return i - 1;              // UNFL ERROR: scalar
variable is underflow
10        }
11        float underflow (float value){
12          return - 2 * value;        // UNFL ERROR: float
variable is underflow
13        }
14      };
15
16
17      void main(void)
18      {
19        Calcul c;
20        int i = 1;
21        float fval = FLT_MAX;
```

```
22
23      i = -2 * (i << 30);           // i = -2**31
24
25      if (random_int())
26        i = c.makeUnderflow(i);   // NTC ERROR: propagation
of UNFL ERROR
27
28      if (random_int())
29        fval = c.underflow(fval); // NTC ERROR: propagation
of UNFL ERROR
30     }
```

Explanation. The minimum integer value on a 32 bit architecture platform is represented by *-2**31*, thus adding *(-1)* will raise an underflow.

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.

```
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 underflow
near zero
11  min_float = min_float * min_float; // UNFL ERROR: underflow
checked by verifier
12
13 }
```

Float Underflow and Overflow: 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 shows an UOVFL
when both overflow and underflow can occur on the same operation.

Scalar or Float Division by zero: ZDV

Check to establish whether the right operand of a division (denominator)
is different from 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
8        }
9        float zdvf(float p){
10         float j = 1.0;
11         return (1024.0 / (j-p)); // ZDV ERROR: float Division by Zero
12       }
13     };
14
15     int main(void)
```

```
16    {
17      Operation op;
18
19      if (random_value())
20        op.zdvs(1);           // NTC ERROR: propagation of ZDV ERROR.
21
22      if (random_value())
23        op.zdvf(1.0);         // NTC ERROR: propagation of ZDV ERROR.
24    }
```

Shift Amount is Outside its Bounds: SHF

Check to establish that a shift (left or right) is not bigger than the size of integral type (int and long int). The range of allowed shift depends on the target processor: 16 bits on *c-167*, 32 bits on *i386* for int, etc.

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 );        // NTC
ERROR: propagation of SHF ERROR
29      if (random_value()) s.opShiftUnsigned(u ,32 ); // NTC
ERROR: propagation of SHF ERROR
30      if (random_value()) s.opShiftSup(l ,32 );     // NTC
ERROR: propagation of SHF ERROR
31
32    }
```

Explanation

In this example, we just show that shift amount is 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); // NTC ERROR: propagation
of SHF ERROR
18    }
```

Explanation

As signed number representation is stored in the higher order bit, you can not left-shift a signed number without loosing sign information.

As an aside, note that the -allow-negative-operand-in-shift option used at launching time instructs PolySpace to allow explicitly signed numbers on shift operations. Using the option in the current example, the red check at line 8 is transformed in a green one.

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)

Array Index is Outside its Bounds: OBAI

Check to establish whether an index is compatible with the length of the array being accessed.

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 = O; index < TAILLE_TAB ; index++){
19         table[index] = 10;
20       }
21       table[index] = 1;  // OBAI ERROR: [out of bounds array index]
22     };
23
24
25     void main(void)
26     {
27       Array* test = new Array();
28       test->initArray();    // NTC ERROR: propagation of OBAI ERROR
29     }
```

Explanation

Just after the loop, *index* equals *SIZE_TAB*. Thus *tab[index] = 1* overwrites
the memory cell just after the last array element.

Note The message associated with the check OBAI gives always the range of the array: out of bounds array index [0..1023].

Function Pointer Must Point to a Valid Function: COR

Check to establish whether a function pointer points to a valid function, or to function with a valid 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); // COR ERROR: function pointer must point to a
valid function
16     }
```

Explanation

In the example, *func* has a prototype in conformance with *CallBack*'s declaration. Therefore *func* is initialized to point to the NULL function through the global declaration of *funcS*.

Wrong Number of Arguments: COR

Check to establish whether the number of arguments passed to a function matches the number of argument in its prototype.

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); // COR ERROR: [function pointer
must point on a valid function]
22       // COR Warning: [wrong number of arguments for call
to function foo_nb(int): got 2 instead of 1]
23     }
```

Explanation

In this example, *ptr_func* is a pointer to a function that takes two arguments but it has been initialized to point to a function that only takes one.

In this case this is the associated COR warning which explains the COR ERROR: *[wrong number of arguments for call to function <name<: got <N< instead of <M<]*, where <N< is the number of argument used and <M< the number of argument waited.

Wrong Type of Argument: COR

Check to establish whether each argument passed to a function matches the prototype of that 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) ;   // COR ERROR: [function pointer must
point on a valid function]
12       // COR Warning: [wrong type for argument #1 of call
to function f(float)]
13     }
14
15     int main()
16     {
17       int idx = 0;
18
19       for (int i = 9; i < 10; ++ i) {
20         if (random)
21           badTab(++idx); // NTC ERROR: propagation of COR ERROR
22       }
23     }
```

Explanation

In this example, *tab* is an function pointer to functions which expects a float as input argument. However, the parameter used is an *int*. So PolySpace Viewer prompts the user to check the validity oh the code.

In this case, this is the associated COR warning which explains the COR ERROR: *[wrong type for argument #<N> of call to function <name>]*, where <N> gives the location of the wrong argument in the function.

Pointer is Outside its Bounds: IDP

Check to establish whether a reference refers to a valid object (whether the dereferenced pointer is still inbound 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: reference refers to an
invalid object
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); // NTC ERROR:
propagation of IDP ERROR
32     }
```

Explanation

The pointer *p* is initialized to point to the first element of *tab* at line 4. When the loop exits, *p*.

For more information, refer to the following sections:

- "Understanding Addressing" on page 2-43
- "Understanding Pointers" on page 2-47

Understanding Addressing
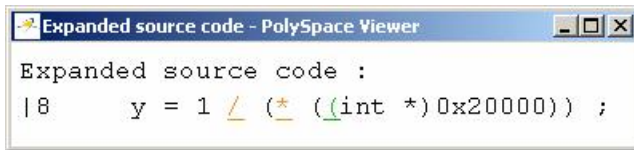
- "Hardware Registers" on page 2-43
- "NULL pointer" on page 2-45
- "Comparing addresses" on page 2-46

Hardware Registers. Many code verifications exhibit **orange** out of bound checks with respect to accesses to absolute addresses and/or hardware registers.
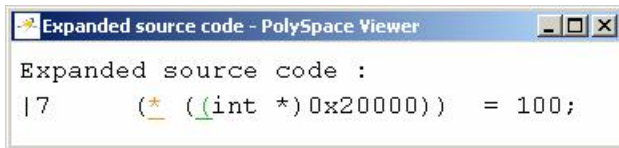
(Also refer to the discussion on Absolute Addressing)

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
```



```
Expanded source code - PolySpace Viewer     _ |□| x|
Expanded source code :
|8      y = 1 / (* ((int *)0x20000)) ;
```



```
Expanded source code - PolySpace Viewer     _ |□| x|
Expanded source code :
|7      (* ((int *)0x20000))  = 100;
```

```
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;`

Replace	With

`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, 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 essential to ensure the assignment of the correct initialization values to them.

Comparing addresses. PolySpace 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 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 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?" on page 2-48

- "Structure Handling — Array Conversions: COR" on page 2-48

- "Structure Handling — Mapping a Small Structure into a Bigger One" on page 2-49

How does malloc work for PolySpace?. PolySpace verification accurately 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 prompts user 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 }
```

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 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 throw and not catch in a destructor or during a delete.

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        };
```

```cpp
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;          // NTC ERROR: propagation of throw during
destructor
53      }
54      catch (error){
55        //raised when an error occurs in a destructor
56      }
57      catch (...){}
58    }
```

Explanation

In the 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, it is possible that the exception is raised or not in the destructor, and an orange warning is displayed instead.

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 displays a green message (line 6).

- The second one, *returnTabInOrderBad*, uses a C function pointer which always raises an exception. PolySpace displays a red message on the C function (line 9).

Limitation: even if *c_compare_bad* function always raise an exception, PolySpace 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        >_&).COR.0.error.html" name="L10-C2">NotPossible(const string & s)
: Error_Message(>_&).NIP.1.error.html" name="L10-C48">s)>_&).COR.2.error.html"
name="L10-C50">{};
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);                // NTC ERROR:
propagation of not specified exception
36        long total = James -> getAccount();
37      }
38      catch (NotPossible&){}
39      catch (...){};
40    }
41
```

Explanation

In the above example, the *Account* class is defined with the *debit* function which allows to throw the specified exception. This function can only catch the *int* and *char* exceptions. The bank authorized an overdraft of 300 euros. The James's account is created with an initial balance of 12000 Euros. So, at line 35, his account is debited with 13000. In the *debit* function, the *if* condition

(line 27) is true, thus 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. So PolySpace 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 = O;
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 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;*

Unreachable Code: UNR

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

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) {            // Unreachable
code for inter == Wait
28            inter = End;
29          }
30          // use of i not initialized
31          if (random_int()) {
32            inter = (enumState)intermediate_state(i); // NIV ERROR:
[non initialized variable]
33            if (inter == Intermediate) {            // Unreachable
code after runtime error
34              inter = End;
35            }
36          }
37      } else {
38        i = 1;                                   // Unreachable code
39        inter = (enumState)intermediate_state(i);  // UNR check
40      }
41      return res_end;
42    }
43
```

Explanation

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

- At line 30 a conditional part of a conditional branch is always true and the other part never evaluated because of the standard definition of logical operator "||".

- The piece of code after a red error is never evaluated by PolySpace. The call to the function and the following line after line 35 are considered to be lines of dead code. Correcting the red error and re-launching would allow the color to be revised.

- At line 27, the first branch is always evaluated to true (*if-{* part) and the other branch is never executed (*else* part at lines 41 to 42).

Non Terminations: Calls and Loops

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.

Check	Description
	• 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 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 mathematic 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 known-NTC facility in the Viewer, 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.

C++ Example.

```
1
2      static  volatile int _x = 1;
3
4      void foo(int x)
5      {
6        int y = 1 / x;          // ZDV Warning: depends on context
7        while(1) {         // NTL ERROR:  loop never terminates
8          if ( y != x) {
9             y = 1 / (y-x);     // ZDV Verified
10         }
11       }
12     }
13
14     void main(void) {
15
16       if (_x)
17         foo(0); // NTC ERROR: Zero DiVision (ZDV) in foo
18       if (_x)
19         foo(2); // NTC ERROR: Non Termination Loop (NTL) in foo
20     }
21
```

Explanation. In this example, the function *foo* is called twice in *main* and neither of these 2 calls ever terminates:

- The first never returns because a division by zero occurs at line 6 (bad argument value), and propagation of this error is propagated to caller at line 17.

- The second never terminates because of an infinite loop (red NTL) at line 7. This error is propagated to caller at line 19.

As an inside, note that by using either the *-context-sensitivity "foo"* option or the *-contex-sensitivity-auto* option at launch time, it is possible for PolySpace to show explicitly that a ZDV error comes from the **first** call of *foo* in *main*.

Non Termination of Loop: NTL

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

C++ Example.

```
1
2      // prototypes fo functions
3      void send_data(double data);
4      void update_alpha(double *a);
5      static volatile double _acq =0.0;
6      static volatile int start_ = 0;
7      typedef void (*pfvoid)(void);
8
9      extern void launch (pfvoid);
10
11     void task(void)
12     {
13       double acq, filtered_acq, alpha;
14
15       // Init
16       filtered_acq = 0.0;
17       alpha = 0.85;
18
19       while (1) {      // NTL ERROR: [non termination of loop]
20         //  Acquisition
21         acq = _acq;
22         // Treatment
23         filtered_acq = acq + (1.0 - alpha) * filtered_acq;
```

```
24        // Action
25        send_data(filtered_acq);
26        update_alpha(&alpha);
27      }
28    }
29
30    void rte_loop(void)
31    {
32      int i;
33      double twentyFloat[20];
34
35      for (i = 0; i <= 20; i++) {  // NTL ERROR: propagation
of OBAI ERROR
36        twentyFloat[i] = 0.0;      // OBAI Warning: 20
verification with i in [0,19]
37                                   // and one ERROR with i = 20
38      }
39    }
40
41    void main()
42    {
43      if (start_)
44        launch(task);
45
46      rte_loop();          // NTC ERROR: propagation of NTL error
47    }
```

Explanation. In the example at line 19, the "continuation condition" is always true and the loop will never exit. Thus PolySpace will raise an error. In some case, the condition is not trivial and may depend on some program variables. Nevertheless PolySpace is still able to 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 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; }`.

Approximations Used During Verification

Why PolySpace Verification Uses Approximations

What is Static Verification

PolySpace software uses *static verification* to prove the absence of runtime errors. Static verification derives the dynamic properties of a program without actually executing it. This differs significantly from other techniques, such as runtime debugging, in that the verification it provides is not based on a given test case or set of test cases. The dynamic properties obtained in the PolySpace verification are true for all executions of the software.

PolySpace verification works by approximating the software under verification, using safe and 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 needed.

Using the static verification approach, the variable 'i' is modelled by its variation domain. For instance the model of 'i' is that it belongs to the [0..999] static interval. (Depending on the complexity of the data, convex polyhedrons, integer lattices and more elaborated models are also used for this purpose).

Any approximation leads by definition to information loss. For instance, the information that 'i' is incremented by one every cycle in the loop is lost. However the important fact is that this information is not required to ensure that no range error will occur; it is only necessary to prove that the variation domain of 'i' is smaller than the range of 'tab'. Only one check is required

to establish that – and hence the gain in efficiency compared to traditional approaches.

Static code verification does have an exact solution, but that solution is generally not practical, as it would generally require the enumeration of all possible test cases. As a result, approximation is required.

Exhaustiveness

Nothing is lost in terms of exhaustiveness. The reason is that PolySpace works by performing upper approximations. In other words, the computed variation domain of any program variable is always a superset of its actual variation domain. The direct consequence is that no runtime error (RTE) item to be checked can be missed by PolySpace.

Approximations Made by PolySpace Verification

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 is a static tool analyzing source code. Memory management concerns dynamic considerations, and the characteristics of particular compilers and targets. PolySpace 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" point to real variables

```
6  i = & one_variable;
7  j = & another_one;
9  if ( i < j) // the condition can still be true or false
```

Left Shift on Negative Variables

Consider the example below.

- When the option -**allow-negative-operand-in- shift** is not used. PolySpace gives a red error on the SHF check because behavior is compiler-dependant.

- When the option -**allow-negative-operand-in- shift is**used, *y* is always
  full range even if the signed value of *x* is known.

```
4  char x, y;
5  x = 0x8F;
6  y = x << 3 ; // OVFL and UNFL Warnings
```

Bitfields

PolySpace considers a bitfield to be a permanently full range variable.

```
4 typedef struct _x
5 {  unsigned int a:1;
7  unsigned int b:1; } bit;

12 int main(void)
13 { bit z;
14  z.b = 0;
15  z.a = 1;
16  assert(z.a == 1); // orange ASRT
```

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, even it should be gray
18   assert (Y == 12); // Warning assert, even 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; // Verifier considers [X==1] or [X==13]
28  if (Y == 13)
29   Y = 14;
30  else
31   Y = X - 1 ; // Verified checks even it should be gray
32  end_cs();
33 }
```

Trigonometric Functions

With all trigonometric functions such as cosines, sines etc., PolySpace always assumes that the return value is bound between the limits of that function - irrespective 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 = asin(0.0);
13 assert(res == 0.0);  // Always in [-pi/2..pi/2]
14
15 res = acos(0.0);
16 assert(res == 0.0);  // Always in [0..pi]
```

Unions

It is recognized nonetheless that there are situations in which the careful use of unions is desirable in constructing an efficient implementation. Nevertheless, the kinds of implementation behavior that might relevant are:

- Padding: padding could be inserted at the end of an union.

- Alignment: members of any structures within union could 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.

This why PolySpace can lose precision when structure unions are considered. Indeed this kind of implementation is compiler dependant. Conversions from one type a union to another will cause a loss of precision on the following check:

Is the other field initialized? Orange NIV

```
typedef union _u {
int a;
char b[4]; } my_union;
my_union X;

X.b[O] = 1; X.b[1] = 1; X.b[2] = 1; X.b[1] = 1;
if (X.A == Ox1111)
else // both branches are reachable
```

Constant Pointer

To increase PolySpace precision where pointers are analyzed, replace

```
const int *p = &y;
```

with:

```
#define p (&y)
```