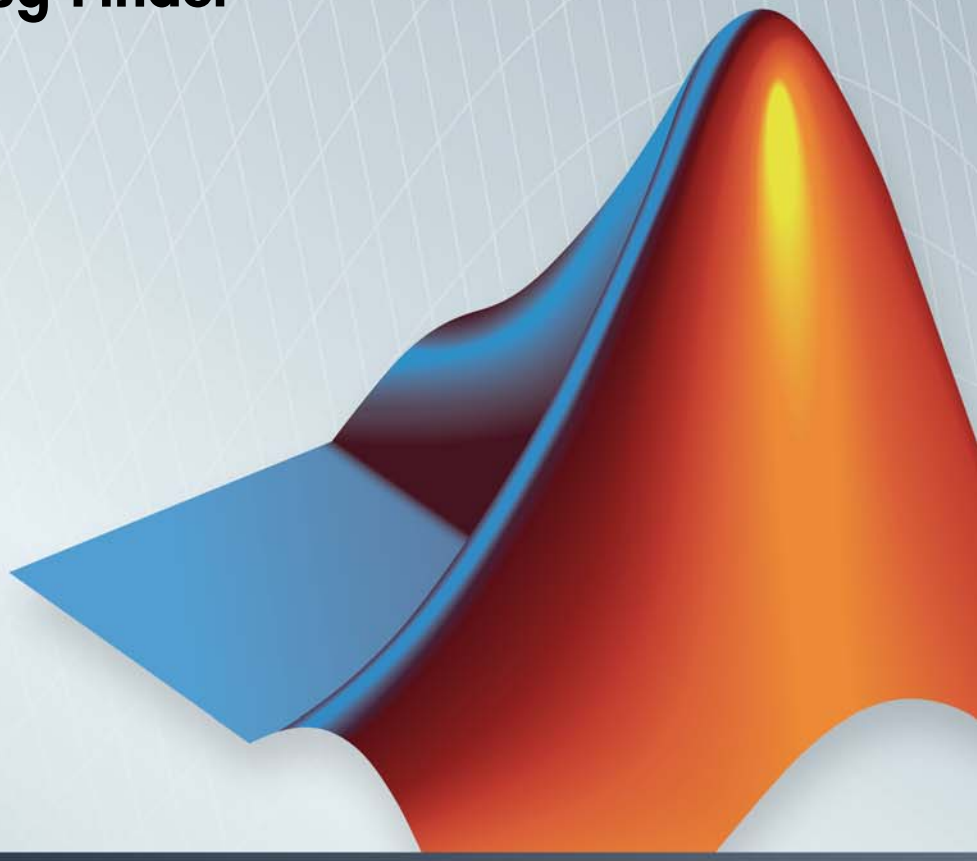
Polyspace® Bug Finder™

Reference

R2013**b**



MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Polyspace® Bug Finder™ Reference

Trademarks

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

Patents

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

Revision History

September 2013	Online only	New for Version 1.0 (Release 2013b)

Contents

1 Reference Concepts

2 Option Descriptions for C Code

3

4 Command Line Only Options

5 Checks

6 Functions

1

Reference Concepts

Bug Finder Defect Categories

In this section...

Numerical

These defects are errors relating to variables in your code; their values, data types, and usage. The defects include:

- Mathematical operations
- Conversion overflow
- Operational overflow

For specific defects, see "Numerical Defects".

Static Memory

These defects are errors relating to memory usage when the memory is statically allocated. The defects include:

- Accessing arrays outside their bounds
- Null pointers
- Casting of pointers

For specific defects, see "Static Memory Defects".

Dynamic Memory

These defects are errors relating to memory usage when the memory is dynamically allocated. The defects include:

- Freeing dynamically allocated memory
- Unprotected memory allocations

For specific defects, see “Dynamic Memory Defects”.

Programming

These defects are errors relating to programming syntax. These defects include:

- Assignment vs. equality operators
- Mismatches between variable qualifiers or declarations
- Badly formatted strings

For specific defects, see “Programming Defects”

Data-flow

These defects are errors relating to how information moves throughout your code. The defects include:

- Dead or unreachable code
- Unused code
- Non-initialized information

For the specific defects, see “Data-flow Defects”.

Other

These defects are those that do not fit into any of the other categories. They can be anything from race conditions to pass-by-value errors.

For specific defects, see “Other Defects”.

2

Option Descriptions for C Code

Batch

Specify remote analysis.

Settings

Default: Off

On, select the check box
: Run analysis remotely.

Off, clear the check box
: Run analysis locally.

Command-Line Information

At the command line, use with the `-scheduler` option

Parameter: `-batch`
Value: *`analysis_options`*
Example: `polyspace-bug-finder-nodesktop -batch` *`analysis_options`* `...`

Add to results repository

Specify addition of analysis results to the Polyspace® Metrics results repository, which allows Web-based reporting of results and code metrics.

Settings

Default: Off

On, select the check box
: Analysis results are stored in the Polyspace Metrics results repository. This allows you to use a Web browser to view results and code metrics.

Off, clear the check box
: Analysis results are not stored in the results repository.

Dependency

- This option is available only for remote analyses.

Command-Line Information

Parameter: `-add-to-results-repository`
Example: `polyspace-code-prover-nodesktop -batch -add-to-results-repository`

Other

In this section...

-extra-flags

This dialog box is for adding nonofficial or expert options to the analyzer. Each word of the option (even the parameters) must be preceded by *-extra-flags*.

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

Default:

No extra flags.

Example Shell Script Entry:

```
 polyspace-bug-finder-nodesktop -extra-flags -param1 -extra-flags
-param2 \

  -extra-flags 10 ...
```

-c-extra-flags

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

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

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -c-extra-flags -param1
-c-extra-flags -param2 -c-extra-flags 10
```

-cfe-extra-flags

This option is used to specify an expert option for an analysis.

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

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -cfe-extra-flags -param1
-cfe-extra-flags -param2
```

-il-extra-flags

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

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

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -il-extra-flags -param1
-il-extra-flags -param2 -il-extra-flags 10
```

Target operating system

This option specifies the operating system target for your application.

Possible values are:

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

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

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

Default:

`no-predefined-OS`

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 operating system. For instance, in order to analyze a VxWorks® project, use the option `-I` *`path_to_the_VxWorks_include_folder`*

Example shell script entry:

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

Target processor type

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

Possible values are:

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

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

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

For information on specifying a generic target, or modifying the `mcpu` target, see “Generic target options” on page 2-10.

Default:

`i386`

Example shell script entry:

```
polyspace-bug-finder-nodesktop -target m68k ...
```

Generic target options

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

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

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

- *char [8, 8, char [16,16]]*
- *short [8,8], short [16, 16]*
- *int [16, 16]*
- *long [32, 32], long long [32, 32]*
- *float [32, 32], double [32, 32], long double [32, 32]*
- *pointer [16, 16]*
- *char is signed*
- *little-endian*

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

```
polyspace-bug-finder-nodesktop -target MyTarget
```

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

```
polyspace-bug-finder-nodesktop -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-bug-finder-nodesktop -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-bug-finder-nodesktop -target mcpu -big-endian
```

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

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

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

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

Example Shell Script Entry

```
   polyspace-bug-finder-nodesktop -default-sign-of-char unsigned
-target mcpu ...
```

-char-is-16bits

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

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

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

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

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

Without the option char for *mcpu* are 8 bits

Example shell script entry:

```
polyspace-bug-finder-nodesktop -target mcpu -char-is-16bits
```

-short-is-8bits

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

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

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

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

Example shell script entry

```
polyspace-bug-finder-nodesktop -target mcpu -short-is-8bits
```

-int-is-32bits

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

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

Example shell script entry

```
polyspace-bug-finder-nodesktop -target mcpu -int-is-32bits
```

-long-long-is-64bits

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

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

Example shell script entry

```
polyspace-bug-finder-nodesktop -target mcpu -long-long-is-64bits
```

-double-is-64bits

The default configuration of a generic target defines a double as 32 bits. This option, changes both double and *long double* to 64 bits. When a double or

long double is used as a struct member or array component, its alignment is set to 4 bytes.

See also -align option.

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

- Computation of `sizeof` objects referencing double type
- Detection of floating point underflow/overflow

This option is available for the following targets:

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

Example

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

Using the default configuration of sharc21x62, Polyspace analysis 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-bug-finder-nodesktop -target mcpu
-double-is-64bits
```

-pointer-is-32bits

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

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

Example shell script entry

```
polyspace-bug-finder-nodesktop -target mcpu -pointer-is-32bits
```

-align [8|16|32]

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

-align 32 (Default)

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

Example shell script entry with a 32 bits default alignment

```
polyspace-bug-finder-nodesktop -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-bug-finder-nodesktop -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-bug-finder-nodesktop -target mcpu -align 8
```

Dialect

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

Settings

Default: `none`

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

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

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

Tips

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

  Example of source code with Keil dialect:

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

  Example with IAR dialect:

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

Command-Line Information

Parameter: `-dialect`
Type: string
Value: `none | keil | iar`
Default: `none`
Example: `polyspace-bug-finder-nodesktop -dialect keil`

See Also

“Analyze Keil or IAR Dialects”.

Sfr type support

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

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

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

Default:

No dialect used.

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -dialect iar -sfr-types
sfr=8,sfr32=32,sfrb=16
```

Division round down

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

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

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

Default:

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

Example:

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

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

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

Example:

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

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -div-round-down ...
```

Enum type definition

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

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

Possible values are:

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

Signed right shift

Choose between arithmetical and logical computation.

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

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

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

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

Example shell script entry

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

```
polyspace-bug-finder-nodesktop -logical-signed-right-shift
```

Preprocessor definitions

Define macro compiler flags to be used during compilation phase.

You can specify only one flag with each `-D` option. However, you can specify the option multiple times.

Default:

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

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -D HAVE_MYLIB -D USE_COM1 ...
```

Undefined preprocessor definitions

Undefine macro compiler flags.

You can specify only one flag with each `-U` option. However, you can specify the option multiple times.

Default:

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

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -U HAVE_MYLIB -U USE_COM1 ...
```

Code from DOS or Windows file system

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

The affected files are:

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

For example, with this option,

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

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

resolves to:

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

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

Default:

Enabled

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -I /usr/include -dos -I
./my_copied_include_dir -D test=1
```

Command/script to apply to preprocessed files

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

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

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

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

Default:

No command.

Example Shell Script Entry – file name:

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

```
polyspace-bug-finder-nodesktop -post-preprocessing-command
`pwd`/replace_keywords
```

where `replace_keywords` is the following script:

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

# Process every line from STDIN until EOF
```

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

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:

```
matlabroot\matlab\polyspace\bin\polyspace-bug-finder-nodesktop.exe
-post-preprocessing-command
matlabroot\sys\perl\win32\bin\perl.exe
<absolute_path>\replace_keywords
```

Include

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

Default:

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

Example Shell Script Entry:

```
 polyspace-bug-finder-nodesktop -include `pwd`/sources/a_file.h
-include /inc/inc_file.h ...

 polyspace-bug-finder-nodesktop -include
/the_complete_path/my_defines.h ...
```

Multitasking

Select to analyze multitasking code

Entry points

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

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

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

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -entry-points proc1,proc2,proc3
...
```

Critical section details

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

and

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

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

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

Default:

no critical sections.

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -critical-section-begin
"start_my_semaphore:cs" \

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

Temporally exclusive tasks

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

The format of this file is :

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

Default:

No temporal exclusions.

Example Task Specification file

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

```
task1_group1 task2_group1

task1_group2 task2_group2 task3_group2
```

Example Shell Script Entry :

```
polyspace-bug-finder-nodesktop -temporal-exclusions-file
sources/exclusions \

     -entry-points task1_group1,task2_group1,task1_group2,\

     task2_group2,task3_group2 ...
```

MISRA C rules configuration

Specifies set of coding rules to check using the `-misra2` option.

Available options are:

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

  Format of the custom file:

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

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

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

Default:

```
all-rules
```

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -misra2 all-rules ...
```

```
  polyspace-bug-finder-nodesktop -misra2 SQO-subset1 ...

  polyspace-bug-finder-nodesktop -misra2 -custom myrules.txt ...

  polyspace-bug-finder-nodesktop -disable-checkers all -misra2
all-rules ...
```

MISRA AC AGC rules configuration

Specifies set of coding rules to check.

Available options are:

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

  Format of the custom file:

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

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

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

Default:

Disabled

Example Shell Script Entry

```
polyspace-bug-finder-nodesktop -misra-ac-agc all-rules ...

polyspace-bug-finder-nodesktop -misra-ac-agc OBL-rules ...

polyspace-bug-finder-nodesktop -misra-ac-agc SQO-subset1 ...

polyspace-bug-finder-nodesktop -misra-ac-agc -custom myrules.txt
...

  polyspace-bug-finder-nodesktop -disable-checkers all
-misra-ac-agc all-rules ...
```

Check custom rules

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

Default:

Disabled

Example Shell Script Entry

```
polyspace-bug-finder-nodesktop -custom-rules myrules.txt
```

Files and folders to ignore

Specify files or folders that the coding rules checker should ignore. For example, you can specify this option if you use headers that do not conform to the MISRA C standard. You can specify the following values with this option:

- `all-headers` (default) — Exclude folders specified by the `-I` option that contain only header files, that is, folders with no source files.
- `all` — Exclude all include folders specified by the `-I` option. For example, if you are checking a large code base with standard or Visual headers, excluding all include folders can significantly improve the speed of code analysis.
- `custom` — Exclude files and folders that you specify.

The software displays a warning if:

- A specified file or folder does not exist.
- All source code is ignored.

You can specify this option only if you specify the `-misra2`, `-misra-ac-agc`, or `-custom-rules` option.

Example shell script entry :

```
   polyspace-bug-finder-nodesktop -misra2 misra.txt
-includes-to-ignore all

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

Effective boolean types

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

The command line syntax for this option is

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

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

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

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

```
typedef signed char boolean_t;
```

Default:

No data types specified as Boolean.

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -misra2 all-rules -boolean-types
bool_type1,bool_type2,bool_type3
```

Allowed pragmas

Use this option with the `-misra2` option to specify undocumented pragma directives for which MISRA C rule 3.4 should not be applied. MISRA C rule 3.4 requires checking that all pragma directives are documented within the documentation of the compiler.

The command line syntax for this option is

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

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

Default:

No undocumented pragma directives specified

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -misra2 AC-AGC-OBL-subset
-allowed-pragmas pragma01,pragma02,pragma03
```

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

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

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

Note Depending of the architecture used (notably when using batch analysis), the script can be executed locally or remotely.

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 analysis has been ended. So the command looks like:

```
polyspace-bug-finder-nodesktop -post-analysis-command
`pwd`/end_email
```

where `end_email` is your Perl script.

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

```
matlabroot\matlab\polyspace\bin\polyspace-bug-finder-nodesktop.exe
-post-analysis-command
matlabroot\matlab\sys\perl\win32\bin\perl.exe
<absolute_path>\end_email
```

Generate report

Specify whether to create analysis report using report generation options

Settings

Default: Off

Select the check box to generate a report.

Report template name

Specify template for generating analysis report

Settings

Default:

Polyspace_Install\polyspace\toolbox\psrptgen\templates\Developer.rpt.
Polyspace_Install is the installation folder for your Polyspace product.

Report templates provided with the software include:

- `BugFinderSummary.rpt`
- `BugFinder.rpt`
- `CodeMetrics.rpt`

Tip

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

Command-Line Information

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

Output format

Specify output format of report

Settings

Default: `RTF`

`RTF`
: Generate an .rtf format report.

`HTML`
: Generate an .html format report.

`PDF`
: Generate a .pdf format report.

`Word`
: Generate a .doc format report.

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

`XML`
: Generate and .xml format report.

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

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

Command-Line Information

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

Shell script example:

```
polyspace-bug-finder-nodesktop -report-template my_temp -report-output-format pdf
```

Find defects

Enable or disable defect checking.

Select checkbox to enable defect checking, clear to disable defect checking. Use the settings to enable different sets of checkers

Default: `On`

Settings

Default: `default`

`default`
: A list of default defects defined by the software. For information on which defects are default, refer to the individual defect reference pages.

`all`
: All defects.

`custom`
: Choose the defects you want to find by selecting categories of checkers or specific defects.

Command-Line Information

The shell script always processes the `-checkers` option, and then `-disable-checkers` option. Command-line parameters for the defects can be found on the defect reference pages.

Parameter: `-checkers`
Type: strings
Value: category | defect parameter | `all` | `default`
Default: `default`

Parameter: `-disable-checkers`
Type: strings
Value: category | defect parameter

Shell script example:

```
polyspace-bug-finder-nodesktop -checkers numerical -disable-checkers FLOAT_ZERO_DIV
```

Runs an analysis with all numerical checkers except *Float division by zero*.

Concepts

- “Bug Finder Defect Categories” on page 1-2
- “Polyspace Bug Finder™ Defects”

3

Option Descriptions for C++ Code

Other

This dialog box is for adding nonofficial or expert options to the analyzer. Each word of the option (even the parameters) must be preceded by *-extra-flags*.

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

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -extra-flags -param1 -extra-flags
-param2
```

-cpp-extra-flags flag

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

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

Default:

no extra flags.

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -cpp-extra-flags
-stubbed-new-may-return-null
```

-il-extra-flags flag

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

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

Default:

no extra flags.

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -il-extra-flags flag
```

Target processor type

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

Possible values are:

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

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

You can analyze code intended for an unlisted processor type using one of the listed processor types, if they share common data properties. Refer to "Modify Predefined Target Processor Attributes" for more details.

For information on specifying a generic target, or modifying the mcpu target, see "Generic target options" on page 3-6.

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

Default:

`i386`

Example shell script entry:

```
polyspace-bug-finder-nodesktop -target m68k ...
```

Generic target options

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

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

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

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

- *char [8, 8], char [16,16]*
- *short [16, 16]*
- *int [16, 16]*
- *long [32, 32], long long [32, 32]*
- *float [32, 32], double [32, 32], long double [32, 32]*
- *pointer [16, 16]*
- *char is signed*
- *little-endian*

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

```
polyspace-bug-finder-nodesktop -target MyTarget
```

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

```
polyspace-bug-finder-nodesktop -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-bug-finder-nodesktop -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-bug-finder-nodesktop -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-bug-finder-nodesktop -default-sign-of-char unsigned
-target mcpu ...
```

-char-is-16bits

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

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

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

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

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

Without the option char for *mcpu* are 8 bits

Example shell script entry:

```
polyspace-bug-finder-nodesktop -target mcpu -char-is-16bits
```

-short-is-8bits

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

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

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

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

Example shell script entry

```
polyspace-bug-finder-nodesktop -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-bug-finder-nodesktop -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-bug-finder-nodesktop -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 `sizeof` objects 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-bug-finder-nodesktop -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-bug-finder-nodesktop -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-bug-finder-nodesktop -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-bug-finder-nodesktop -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-bug-finder-nodesktop -target mcpu -align 8
```

Dialect

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

- `gnu` (default if `-OS-target` is set to `Linux`)
- `cfront2`
- `cfront3`
- `iso`
- `visual`
- `visual6`
- `visual7.0`
- `visual7.1`
- `visual8`
- `visual9.0`

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

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

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

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

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

Default:

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

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

`none` otherwise

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -dialect visual8 ...
```

Pack alignment value

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

The argument value must be: 1, 2, 4, 8, or 16. Analysis 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-bug-finder-nodesktop -dialect visual
-pack-alignment-value 4 ...
```

Import folder

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

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -dialect visual8 -import-dir
/com1/inc ...
```

Ignore pragma pack directives

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

Polyspace analysis stops execution and displays an error message if this option is used in non visual mode or without dialect gnu (without `-OS-target visual` or `dialect visual*`).

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -dialect visual
-ignore-pragma-pack ...
```

Support managed extensions

Visual C++ /FX option allows the partial translation of sources making use of managed extensions to Visual C++ sources without managed extensions. Theses extensions are currently not taken into account by Polyspace analysis 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 analysis of a project containing translated sources obtained by compilation of a Visual project using the /FX Visual option. Managed files need to be located in the same folder as the original ones and Polyspace software will analyze managed files instead of the original ones without intrusion, and will permit you to remove part of the limitations due to specific extensions.

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

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -dialect visual -
support-FX-option-results
```

Enum type definition

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

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

Possible values are:

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

Management of scope of 'for loop' variable index

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

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

You can specify one of the following values:

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

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

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

Default:

`defined-by-dialect`

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -for-loop-index-scope in
```

Management of w_char_t

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

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

You can specify one of the following values':

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

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

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

Default:

```
defined-by-dialect
```

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -wchar-t-is typedef
```

Set wchar_t to unsigned long

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

For example, `sizeof(L'W')` will have the value of `sizeof(unsigned long)` and the wchar_t field will be aligned in the same way as the unsigned long field. Note that wchar_t will remain a different type from unsigned long unless “-wchar-t-is typedef” is set or implied by the current dialect. The default underlying type of wchar_t is unsigned short.

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -wchar-t-is-unsigned-long ...
```

Set size_t to unsigned long

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

Example Shell Script Entry: `polyspace-bug-finder-nodesktop -size-t-is-unsigned-long ...`

Overcome link error

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

Applying this option will cause Polyspace to ignore this error.

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

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -no-extern-C ...
```

Main entry point

The option specifies the name of the main subprogram when you select a visual `-OS-target`. This procedure will be analyzed after class elaboration, and before tasks in case of a multitasking application or in case of the `-entry-points` usage.

Possible values are:

- `_tmain` (default)
- `wmain`
- `_tWinMain`
- `wWinMain`
- `WinMain`
- `DllMain`.

However, if the main subprogram does not exist, Polyspace analysis stops with an error message.

Default:

```
_tmain
```

Example Shell script entry:

```
polyspace-bug-finder-nodesktop -main WinMain -OS-target visual
```

Entry points

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

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

Format:

- All tasks must have the prototype "`void` *`any_name`*`()` .
- It is possible to declare a member function as an entry point of a analysis, only and only if the function is declared "`static void task_name()`".

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -entry-points
class::task_name,taskname,proc1,proc2
```

Critical section details

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

and

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

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

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

Limitation:

- Name of procedure accept only void any_name() as prototype.
- The beginning and the end of the critical section need to be defined in same block of code.

Default:

no critical sections.

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -critical-section-begin
"start_my_semaphore:cs" \

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

Check MISRA C++ rules

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

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

For more information, see “Activate Coding Rules Checker”.

MISRA C++ rules configuration

Specifies set of coding rules to check.

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

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

Format of the file:

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

Example:

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

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

```
0-1-10 warning
0-1-11 off
0-1-12 off
1-0-1 error
1-0-2 off # Not implemented
1-0-3 off # Not implemented
2-2-1 off # Not implemented
2-3-1 warning
2-5-1 warning
2-7-1 warning

# End of file
```

Default:

Disabled

Example shell script entry:

```
polyspace-bug-finder-nodesktop -misra-cpp all-rules

polyspace-bug-finder-nodesktop -misra-cpp misra.txt

polyspace-bug-finder-nodesktop -disable-checkers all -misra-cpp
all-rules
```

Check JSF C++ rules

Specifies that Polyspace software checks for compliance with the Joint Strike Fighter® Air Vehicle C++ coding standards (JSF++:2005).

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

For more information, see “Activate Coding Rules Checker”.

JSF C++ rules configuration

Specifies which JSF C++ coding rules to check.

- `shall-rules` — Check all **Shall** rules, which are mandatory rules that require analysis.
- `shall-will-rules` — Check all **Shall** and **Will** rules. **Will** rules are mandatory rules that do not require analysis.
- `all-rules` — Check all **Shall**, **Will**, and **Should** rules. **Should** rules are advisory rules.
- `custom` — Check a specified set of JSF C++ coding rules. When you select this option, you must provide a rules file that specifies the JSF C++ rules to check and whether to report an error or warning for violations of each rule. For more information, see "Select Specific Coding Rules".

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-bug-finder-nodesktop -jsf-coding-rules all-rules

polyspace-bug-finder-nodesktop -jsf-coding-rules jsf.txt

polyspace-bug-finder-nodesktop -disable-checkers all
-jsf-coding-rules all-rules
```

Files and folders to ignore

Specify files or folders that the coding rules checker should ignore. For example, you can specify this option if you use headers that do not conform to the JSF++ or MISRA C++ standard. You can specify the following values with this option:

- `all-headers` (default) — Exclude folders specified by the `-I` option that contain only header files, that is, folders with no source files.
- `all` — Exclude all include folders specified by the `-I` option. For example, if you are checking a large code base with standard or Visual headers, excluding all include folders can significantly improve the speed of code analysis.
- `custom` — Exclude files and folders that you specify.

The software displays a warning if:

- A specified file or folder does not exist
- All source code is ignored

You can specify this option only if you specify the `-jsf-coding-rules`, `-misra-cpp`, or `-custom-rules` option.

Example shell script entry :

```
 polyspace-bug-finder-nodesktop -jsf-coding-rules jsf.txt
-includes-to-ignore all

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

4

Command Line Only Options

-sources-list-file

This option is only available at the command line. The syntax of *file_name* is the following:

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

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

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

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

```
  polyspace-bug-finder-nodesktop -sources-list-file
"C:\Analysis\files.txt"

  polyspace-bug-finder-nodesktop -sources-list-file "files.txt"
```

-v | -version

Display the Polyspace version number.

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop  v
```

It will show a result similar to:

```
Polyspace r2007a+

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

-h[elp]

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

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop  h
```

-prog

Specify a name for the project.

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

Settings

Default: `New_Project`

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

Command-Line Information

Parameter: `-prog`
Value: any valid value
Example: `polyspace-bug-finder-nodesktop -prog myApp ...`

-date

Specify a date stamp for the analysis.

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

Settings

Default: Date the analysis is launched

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

Tip

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

Command-Line Information

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

-lang

Specify the code language for the project.

Note In the Polyspace interface, specify the project language when you create a new project. For more information, see “Create New Projects”.

Settings

Specify either C or C++ as the language.

Command-Line Information

Parameter: `-lang`
Value: `c | cpp`
Example: `polyspace-bug-finder-nodesktop -lang c ...`

-author

Specify the name of the person performing the analysis.

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

Settings

Default: `username` of the current user.

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

Command-Line Information

Parameter: `-author`
Value: any valid value
Example: `polyspace-bug-finder-nodesktop -author "John Tester"`

-results-dir

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

Default:

Shell Script: The folder in which tool is launched.

From Graphical User Interface: `C:\Polyspace_Results`

Example Shell Script Entry:

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

-sources

Specifies a list of source files to be analyzed.

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

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

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

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

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

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

Defaults:

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

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

```
polyspace-bug-finder-nodesktop -sources "my_directory/*.cpp" ...
polyspace-bug-finder-nodesktop -sources "my_directory/file1.cc
other_dir/file2.cpp" ...
```

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

```
polyspace-bug-finder-nodesktop -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 at the command line

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

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

-I

Specify the name of a folder that must be included when compiling C sources. You can specify only one folder for each `-I` instance. However, you can specify this option multiple times.

Polyspace software implicitly includes the `./sources` folder (if it exists) after any include folders that you specify.

Example Shell Script Entry-1:

```
polyspace-bug-finder-nodesktop -I /com1/inc -I /com1/sys/inc
```

is equivalent to

```
polyspace-bug-finder-nodesktop -I /com1/inc -I /com1/sys/inc
-I ./sources
```

Example Shell Script Entry-2:

```
polyspace-bug-finder-nodesktop
```

is equivalent to

```
polyspace-bug-finder-nodesktop -I ./sources
```

-import-comments

Removing

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

Default:

Disabled

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -version 1.3 -import-comments
C:\PolyspaceResults\1.2
```

-tmp-dir-in-results-dir

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

Default:

Disabled

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -tmp-dir-in-results-dir
-results-dir C:\Polyspace\Results
```

-less-range-information

Limits the amount of range information displayed in analysis results.

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

In addition, selecting this option enables the `no-pointer-information` option. See "`-no-pointer-information`" on page 4-17.

Computing range information for reads and operators may take a long time, and can reduce the precision of the analysis. Selecting this option can reduce analysis time significantly, and improve the precision of the analysis. Consider the following example:

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

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

Default:

Disabled.

Example Shell Script Entry :

```
polyspace-bug-finder-nodesktop -less-range-information
```

-no-pointer-information

Stops the display of pointer information in analysis results.

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

Consider the following example:

```
x = *p;
```

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

Default:

Disabled.

Example Shell Script Entry :

```
polyspace-bug-finder-nodesktop -no-pointer-information
```

-asm-begin -asm-end

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

and

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

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

Consider the following example.

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

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

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

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -discard-asm -asm-begin
"asm_begin_1,asm_begin_2" -asm-end "asm_end_1,asm_end_2" ...
```

-permissive

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

- `-ignore-constant-overflows`
- `-allow-negative operand-in-shift`

-Wall

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

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

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

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

Default:

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

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -Wall ...
```

-report-output-name

Specify name of analysis report file

Settings

Default: *Prog_TemplateName.Format* where:

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

Command-Line Information

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

Shell script example:

```
polyspace-bug-finder-nodesktop -report-template my_temp -report-output-name Air.rtf
```

-max-processes

This option determines the number of processors used in during the analysis. By default, Polyspace will take advantage of a multiprocessor to speed up analysis. If you want to specify a specific maximum number of processors use this option at the command line.

Command-Line Information

Parameter: `-max-processes`
Type: integer
Value: an integer between 1 and 128

Shell script example:

```
polyspace-bug-finder-nodesktop -max-processes 4 ...
```

-scheduler

This option calls the job scheduler to run your analysis remotely. Use this option with the `-batch` option.

Command-Line Information

Parameter: `-scheduler`
Type: hostname or MATLAB® Job Scheduler
Value: `hostname` or `MJSname@host`

Shell script example:

```
polyspace-bug-finder-nodesktop -batch -scheduler MJSname@host
```

```
polyspace-bug-finder-nodesktop -batch -scheduler hostname
```

5

Checks

Assertion

Purpose

Failed assertion statement

Description

Assertion occurs when the asserted expression is or might be false.

Examples

Check Assertion on Unsigned Integer

```
void asserting_x(unsigned int theta) {

    theta =+ 5;
    assert(theta < 0);
}
```

In this example, the `assert` function checks if the input variable, `theta`, is less than or equal to zero. The assertion fails because `theta` is an unsigned integer, so the value at the beginning of the function is at least zero. This positive value is increased by five. Therefore, the range of `theta` is `[5..MAX_INT]`. `theta` is always greater than zero.

Correction — Change Assert Expression

One possible correction is to change the assertion expression. By changing the *less-than-or-equal-to* sign to a *greater-than-or-equal-to* sign, the assertion no longer fails.

```
void asserting_x(unsigned int theta) {

    theta =+ 5;
    assert(theta > 0);
}
```

Correction — Fix Code

One possible correction is to fix the code related to the assertion expression. If the assertion expression is true, fix your code so the assertion passes.

```
void asserting_x(int theta) {
```

```
    theta = -abs(theta);
    assert(theta < 0);
}
```

Command-Line Information

Argument: `assert`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers assert`

Related Examples

- "Review and Comment Results"

Concepts

- "Other Defects"

Illegal delete

Purpose

Pointer deallocation using `delete` without corresponding allocation using `new`

Description

Illegal delete occurs when a block of memory released using the `delete` operator was not previously allocated with the `new` operator.

This defect applies only if the code language for the project is C++.

Examples

Illegal delete error

```
void Assign_Ones(void)
{
  int p[10];

  for(int i=0;i<10;i++)
     *(p+i)=1;

  delete[] p;
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer `p` is released using the `delete` operator. However, `p` points to a memory location that was not dynamically allocated.

Corrected Code: Remove Pointer Deallocation

If the number of elements of the array `p` is known at compile time, one possible correction is to remove the deallocation of the pointer `p`.

```
void Assign_Ones(void)
{
  int p[10];

  for(int i=0;i<10;i++)
     *(p+i)=1;

  /* Fix: Remove deallocation of p */
}
```

Correction — Introduce Pointer Allocation

If the number of elements of the array `p` is not known at compile time, one possible correction is to dynamically allocate memory to the array `p` using the `new` operator.

```
void Assign_Ones(int num)
   {
     /* Fix: Allocate memory dynamically to p */
     int *p = new int[10];

     for(int i=0;i<10;i++)
        *(p+i)=1;

     delete[] p;
   }
```

Command-Line Information

Argument: `bad_delete`
Type: string
Default: `'off'`
Example: `polyspace-bug-finder-nodesktop -checkers bad_delete`

See Also

`Invalid free of pointer` |

Related Examples

- "Review and Comment Results"

Invalid use of == operator

Purpose

Equality operation in assignment statement

Description

Invalid use of == operator occurs when an equality operator instead of an assignment operator is used in a simple statement. A common correction is removing one of the equal signs (=).

Examples

Equality Evaluation in for-loop

```
void populate_array(void)
{
    int i = 0;
    int j = 0;
    int array[4];

    for (j == 5; j < 9; j++) {
        array[i] = j;
        i++;
    }
}
```

Inside the `for-loop`, the statement `j == 5` tests whether `j` is equal to 5 instead of setting `j` to 5. The `for-loop` iterates from 0 to 8 because `j` starts with a value of 0, not 5. A by-product of the invalid equality operator is an out-of-bounds array access in the next line.

Correction — Change to Assignment Operator

One possible correction is to change the == operator to a single equals sign (=), Changing the == sign resolves both defects because the `for-loop` iterates the correct number of times.

```
void populate_array(void)
{
    int i = 0;
    int j = 0;
    int array[4];

    for (j = 5; j < 9; j++) {
```

```
            array[i] = j;
            i++;
        }
    }
```

Command-Line Information

Argument: `bad_equal_equal_use`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers bad_equal_equal_use`

See Also

`Invalid use of = operator` |

Related Examples

- "Review and Comment Results"

Concepts

- "Programming Defects"

Invalid use of = operator

Purpose

Assignment in control statement

Description

Invalid use of = operator occurs when an assignment is made inside a logical statement, such as `if` or `while`. Use the equals operator as an assignment operator, not to determine equality. A common correction for this defect is adding a second equal sign (==).

Examples

Assignment in an if-statement

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if(alpha = beta){
        printf("Equal\n");
    }
}
```

The equal sign is flagged as a defect because the assignment operator is used within the if-statement. Due to the single equals sign, the statement assigns the value `beta` to `alpha`, then determines the logical value of `alpha`.

Correction — Equality operator in if-statement

One possible correction is adding an additional equal sign. This correction changes the assignment operator to an equality operator. The if-statement evaluates the equality between `alpha` and `beta`.

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if(alpha == beta){
        printf("Equal\n");
    }
}
```

Correction — Assignment Inside an if-statement

If an assignment must be made inside a control statement, one possible correction is clarifying the control statement. This correction assigns the value of `beta` to `alpha`, and determines if `alpha` is nonzero.

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if((alpha = beta) != 0){
        printf("Equal\n");
    }
}
```

Command-Line Information

Argument: `bad_equal_use`
Type: string
Default: `'off'`
Example: `polyspace-bug-finder-nodesktop -checkers bad_equal_use`

See Also

`Invalid use of == operator` |

Related Examples

- "Review and Comment Results"

Concepts

- "Programming Defects"

Invalid use of floating point operation

Purpose

Imprecise comparison of floating point variables

Description

Invalid use of floating point operation occurs when you use an equality (==) or inequality (!=) operation with floating point numbers. It is possible that the equality or inequality of two floating point values is not exact because floating point representation might be imprecise.

Examples

Two Equal Floats

```
float onePointOne(void) {

    float flt = 1.0;
    if (flt == 1.1)
        return flt;
    return 0;
}
```

In this function, the if-statement tests the equality of `flt` and the number 1.1. Even though the equality in this function is obvious (1.0 is not equal to 1.1), longer floating point values are not quite so simple. Do not use equality with floating points because it can produce unexpected behavior.

Correction — Change the Operator

One possible correction is to use a different operator that is not as strict. For example, an inequality like > or <.

```
float onePointOne(void) {

    float flt = 1.0;
    if (fabs(flt-1.1) < Epilson)
        return flt;
    return 0;
}
```

Correction — Change the Operands

One possible correction is to change the operands to more precise data types. In this example, using integers instead of floats corrects the error.

```
int onePointOne(void) {

    int flt = 1;
    if (flt == 1)
        return flt;
    return 0;
}
```

Command-Line Information

Argument: `bad_float_op`
Type: string
Default: `'off'`
Example: `polyspace-bug-finder-nodesktop -checkers bad_float_op`

Related Examples

- "Review and Comment Results"

Concepts

- "Other Defects"

Invalid free of pointer

Purpose

Pointer deallocation without a corresponding dynamic allocation

Description

Invalid free of pointer occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

Examples

Invalid free of pointer error

```
#include <stdlib.h>

void Assign_Ones(void)
{
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;

  free(p);
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer `p` is deallocated using the `free` function. However, `p` points to a memory location that was not dynamically allocated.

Correction — Remove Pointer Deallocation

If the number of elements of the array `p` is known at compile time, one possible correction is to remove the deallocation of the pointer `p`.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

Correction — Introduce Pointer Allocation

If the number of elements of the array `p` is not known at compile time, one possible correction is to dynamically allocate memory to the array `p`.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

Command-Line Information

Argument: `bad_free`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers bad_free`

See Also

`Illegal delete` |

Related Examples

- "Review and Comment Results"

Code deactivated by constant false condition

Purpose

Code segment deactivated by `#if 0` directive or `if(0)` condition

Description

Code deactivated by constant false condition occurs when a block of code is deactivated using a `#if 0` directive or `if(0)` condition.

Examples

Code deactivated by constant false condition error

```
#include<stdio.h>
int Trim_Value(int* Arr,int Size,int Cutoff)
{
 int Count=0;

 for(int i=0;i < Size;i++)
     {
      if(Arr[i]>Cutoff)
            {
             Arr[i]=Cutoff;
             Count++;
            }
     }


  #if 0
  /* Defect: Code Segment Deactivated */

      if(Count==0)
           {
            printf("All values less than cutoff.");
           }
 #endif

 return Count;
}
```

In the preceding code, the `printf` statement is placed within a `#if #endif` directive. The portion within the directive is treated as a code comment and not compiled.

Correction — Change `#if 0` to `#if 1`

Unless you intended to deactivate the `printf` statement, one possible correction is to reactivate the block of code in the `#if #endif` directive. To reactivate the block, change `#if 0` to `#if 1`.

```
#include<stdio.h>
int Trim_Value(int* Arr,int Size,int Cutoff)
{
 int Count=0;

 for(int i=0;i < Size;i++)
     {
      if(Arr[i]>Cutoff)
            {
             Arr[i]=Cutoff;
             Count++;
            }
     }


 /* Fix: Replace #if 0 by #if 1 */
 #if 1
      if(Count==0)
           {
            printf("All values less than cutoff.");
           }
 #endif

 return Count;
}
```

Command-Line Information

Argument: `deactivated_code`
Type: string
Default: `'off'`
Example: `polyspace-bug-finder-nodesktop -checkers deactivated_code`

Code deactivated by constant false condition

See Also `Dead code` |

Related Examples

- “Review and Comment Results”

Purpose

Code cannot be reached along any execution path

Description

Dead code occurs when a block of code cannot be reached along any execution path. This error excludes directives such as `#if 0`, which you can deliberately use to deactivate a code segment.

Examples

Dead code error

```
#include <stdio.h>

int Return_From_Table(int ch)
{

 int table[5];

 /* Create a table */
 for(int i=0;i<=4;i++)
   table[i]=i^2+i+1;

 if(table[ch]>100) return 0;
 /*Defect: Condition always false */

 return table[ch];
}
```

The maximum value in the array `table` is `4^2+4+1=21`, so the test expression `table[ch]>100` always evaluates to false. The `return 0` in the `if` statement is never executed.

Correction — Remove Dead Code

One possible correction is to remove the `if` condition from the code.

```
#include <stdio.h>

int Return_From_Table(int ch)
  {
   int table[5];
```

```
  /* Create a table */
  for(int i=0;i<=4;i++)
    table[i]=i^2+i+1;

  /* Fix: Remove dead code */
  return table[ch];
}
```

Command-Line Information

Argument: `dead_code`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers dead_code`

See Also

`Code deactivated by constant false condition` |

Related Examples

- "Review and Comment Results"

Purpose

Mismatch between function or variable declarations

Description

Declaration mismatch occurs when a function or variable declaration does not match other instances of the function or variable.

Examples

Inconsistency Between Files

file1.c

```
int foo(void) {
    return 1;
}
```

file2.c

```
double foo(void);

int bar(void) {
    return (int)foo();
}
```

Correction — Align the Function Declarations

One possible correction is to change the function declarations so they match. In this example, by changing the declaration of `foo` in file2.c to match file1.c, the defect is fixed.

file1.c

```
int foo(void) {
    return 1;
}
```

file2.c

```
int foo(void);

int bar(void) {
    return foo();
}
```

Declaration mismatch

Command-Line Information

Argument: `decl_mismatch`
Type: string
Default: `'off'`
Example: `polyspace-bug-finder-nodesktop -checkers decl_mismatch`

Related Examples

- "Review and Comment Results"

Concepts

- "Programming Defects"

Deallocation of previously deallocated pointer

Purpose

Memory freed more than once without allocation

Description

Deallocation of previously deallocated pointer occurs when a block of memory is freed more than once using the `free` function without any intermediate allocation.

Examples

Deallocation of previously deallocated pointer error

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    free (pi);
    /* Defect: pi has already been freed */
}
```

The first `free` statement releases the block of memory that `pi` refers to. The second `free` statement on `pi` releases a block of memory that has been freed already.

Correction — Remove Duplicate Deallocation

One possible correction is to remove the second `free` statement.

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;
```

```
    *pi = 2;
    free(pi);
    /* Fix: remove second deallocation */
}
```

Command-Line Information

Argument: `double_deallocation`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers double_deallocation`

See Also

`Use of previously freed pointer` |

Related Examples

- "Review and Comment Results"

Purpose

Overflow when converting between floating point data types

Description

Float conversion overflow occurs when converting a floating point number to a smaller floating point data type. If there is not enough memory to represent the original number, the conversion overflows.

The exact storage allocation for different floating point types depends on your target operating system. See "Predefined Target Processor Specifications".

Examples

Converting from `double` to `float`

```
float convert(void) {

    double diam = 1e100;
    return (float)diam;
}
```

In the return statement, the variable `diam` of type double is converted to a variable of type float. However, the value 1^100 requires more than the 32-bits of a float to be accurately represented.

Command-Line Information

Argument: `float_conv_ovfl`
Type: string
Default: `'off'`
Example: `polyspace-bug-finder-nodesktop -checkers float_conv_ovfl`

See Also

`Integer conversion overflow` | `Unsigned integer conversion overflow` | `Sign change integer conversion overflow` |

Related Examples

- "Review and Comment Results"

Concepts

- "Numerical Defects"

Float overflow

Purpose

Overflow from operation between floating points

Description

Float overflow occurs when an operation on floating point variables exceeds the space available to represent the resulting value.

The exact storage allocation for different floating point types depends on your target operating system. See "Predefined Target Processor Specifications".

Examples

Multiplication of Floats

```
float square(void) {

   float val = FLT_MAX;
   return val * val;
}
```

In the return statement, the variable `val` is multiplied by itself. The square of the maximum float value cannot be represented by a float (the return type for this function) because the value of `val` is the maximum float value.

Correction — Different storage type

One possible correction is to store the operation's result in a larger data type. In this example, by returning a `double` instead of a `float`, the overflow defect is fixed.

```
double square(void) {
    float val = FLT_MAX;

    return val * val;
}
```

Command-Line Information

Argument: `float_ovfl`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers float_ovfl`

See Also `Integer overflow` | `Unsigned integer overflow` |

Related Examples

- "Review and Comment Results"

Concepts

- "Numerical Defects"

Invalid use of standard library floating point routine

Purpose

Wrong arguments to standard library function

Description

Invalid use of standard library floating point routine occurs when you use invalid arguments with a floating point function from the standard library. This defect picks up:

- Rounding and absolute value routines

  `ceil, fabs, floor, fmod`

- Fractions and division routines

  `fmod, modf`

- Exponents and log routines

  `frexp, ldexp, sqrt, pow, exp, log, log10`

- Trigonometry function routines

  `cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, acosh, asinh, atanh`

Examples

Arc Cosine Operation

```
double arccosine(void) {

    double degree = 5.0;
    return acos(degree);
}
```

The input value to `acos` must be in the interval `[-1,1]`. This input argument, `degree`, is outside this range.

Correction — Change Input Argument

One possible correction is to change the input value to fit the specified range. In this example, change the input value from degrees to radians to fix this defect.

```
double arccosine(void) {
```

```
    double degree = 5.0;
    double radian = degree*180/(3.14159);
    return acos(radian);
}
```

Command-Line Information

Argument: `float_std_lib`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers float_std_lib`

See Also

`Invalid use of standard library integer routine` | `Invalid use of standard library memory routine` | `Invalid use of standard library string routine` | `Invalid use of standard library routine` |

Related Examples

- "Review and Comment Results"

Concepts

- "Numerical Defects"

Float division by zero

Purpose

Dividing floating point number by zero

Description

Float division by zero occurs when the denominator of a division operation is a zero and a floating point number.

Examples

Dividing an Integer by Zero

```
float fraction(float num)
{
    float denom = 0.0;
    float result = 0.0;

    result = num/denom;

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

Correction — Check Before Division

```
float fraction(float num)
{
    float denom = 0.0;
    float result = 0.0;

    if( ((int)denom) != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, ensuring that no division by zero defects occur. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
float fraction(float num)
{
    float denom = 2.0;
    float result = 0.0;

    result = num/denom;

    return result;
}
```

Command-Line Information

Argument: `float_zero_div`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers float_zero_div`

See Also

`Integer division by zero` |

Related Examples

- "Review and Comment Results"

Concepts

- "Numerical Defects"

Use of previously freed pointer

Purpose

Memory accessed after deallocation

Description

Use of previously freed pointer occurs when a block of memory is accessed after it is freed using the `free` function.

Examples

Use of previously freed pointer error

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   {
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);

    j = *pi + shift;
    /* Defect: Reading a freed pointer */

    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, the dereference of `pi` after the `free` statement is not valid.

Correction — Free Pointer After Use

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;
```

```
    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

Command-Line Information

Argument: `freed_ptr`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers freed_ptr`

See Also

`Deallocation of previously deallocated pointer` |

Related Examples

- "Review and Comment Results"

Unreliable cast of function pointer

Purpose

Function pointer cast to another function pointer with different argument or return type

Description

Unreliable cast of function pointer occurs when a function pointer is cast to another function pointer that has different argument or return type.

This defect applies only if the code language for the project is C.

Examples

Unreliable cast of function pointer error

```
#include <math.h>
#include <stdio.h>
#define PI 3.142

double Calculate_Sum(int (*fptr)(double))
{
    double  sum = 0.0;
    double  y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;

    fp = sin;
    sum = Calculate_Sum(fp);
    /* Defect: fp implicitly cast to int(*) (double) */
```

```
    printf("sum(sin): %f\n", sum);
    return 0;
}
```

The function pointer `fp` is declared as `double (*)(double)`. However in passing it to function `Calculate_Sum`, `fp` is implicitly cast to `int (*)(double)`.

Correction — Avoid Function Pointer Cast

One possible correction is to ensure that the function pointer in the definition of `Calculate_Sum` has the same argument and return type as `fp`. This step ensures that `fp` is not implicitly cast to a different argument or return type.

```
#include <math.h>
#include <stdio.h>
# define PI 3.142

/*Fix: Ensure fptr has same argument and return type everywhere*/
double Calculate_Sum(double (*fptr)(double))
{
    double  sum = 0.0;
    double y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;
```

```
    fp = sin;
    sum = Calculate_Sum(fp);
    printf("sum(sin): %f\n", sum);

    return 0;
}
```

Command-Line Information

Argument: `func_cast`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers func_cast`

Related Examples

- "Review and Comment Results"

Purpose

Overflow when converting between integer types

Description

Integer conversion overflow occurs when converting an integer to a smaller integer type. If there are not enough bytes to represent the original constant, the conversion overflows.

The exact storage allocation for different integer types depends on your operating system. See "Predefined Target Processor Specifications".

Examples

Converting from `int` to `char`

```
char convert(void) {

    int num = 1000000;

    return (char)num;
}
```

In the return statement, the integer variable `num` is converted to a `char`. However, 1000000 cannot be represented by an 8-bit or 16-bit character because it requires at least 20 bits. So the conversion operation overflows.

Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number.

```
long convert(void) {

    int num = 1000000;

    return (long)num;
}
```

Command-Line Information

Argument: `int_conv_ovfl`
Type: string
Default: `'on'`

Example: `polyspace-bug-finder-nodesktop -checkers int_conv_ovfl`

See Also

`Float conversion overflow` | `Unsigned integer conversion overflow` | `Sign change integer conversion overflow` |

Related Examples

- "Review and Comment Results"

Concepts

- "Numerical Defects"

Purpose

Overflow from operation between integers

Description

Integer overflow occurs when an operation on integer variables exceeds the space available to represent the resulting value.

The exact storage allocation for different integer types depends on your operating system. See “Predefined Target Processor Specifications”.

Examples

Addition of Maximum Integer

```
int plusplus(void) {

    int var = INT_MAX;
    var++;
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so one plus the maximum integer value cannot be represented by an `int`.

Correction — Different storage type

One possible correction is to change data types. Store the operation's result in a larger data type. In this example, by returning a `long` instead of an `int`, the overflow error is fixed.

```
long plusplus(void) {

    long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

Command-Line Information

Argument: `int_ovfl`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers int_ovfl`

Integer overflow

See Also `Unsigned integer overflow` | `Float overflow` |

Related Examples
- "Review and Comment Results"

Concepts
- "Numerical Defects"

Invalid use of standard library integer routine

Purpose

Wrong arguments to standard library function

Description

Invalid use of standard library integer routine occurs when you use invalid arguments with an integer function from the standard library. This defect picks up:

- Character Conversion

  `toupper, tolower`

- Character Checks

  `isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit`

- Integer Division

  `div, ldiv`

- Absolute Values

  `abs, labs`

Examples

Absolute Value of Large Negative

```
int absoluteValue(void) {

    int neg = INT_MIN;
    return abs(neg);
}
```

The input value to `abs` is `INT_MIN`. The absolute value of `INT_MIN` is `INT_MAX+1`. This number cannot be represented by the type `int`.

Correction — Change Input Argument

One possible correction is to change the input value to fit returned data type. In this example, change the input value to `INT_MIN+1`.

```
int absoluteValue(void) {

    int neg = INT_MIN+1;
```

```
    return abs(neg);
}
```

Command-Line Information

Argument: `int_std_lib`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers int_std_lib`

See Also

`Invalid use of standard library floating point routine` | `Invalid use of standard library memory routine` | `Invalid use of standard library string routine` | `Invalid use of standard library routine` |

Related Examples

- "Review and Comment Results"

Concepts

- "Numerical Defects"

Purpose

Dividing integer number by zero

Description

Integer division by zero occurs when the denominator of a division operation is a zero.

Examples

Dividing an Integer by Zero

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    result = num/denom;

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

Correction — Check Before Division

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    if (denom != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, ensuring that no division by zero defects occur. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
int fraction(int num)
{
    int denom = 2
    int result = 0;

    result = num/denom;

    return result;
}
```

Command-Line Information

Argument: `int_zero_div`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers int_zero_div`

See Also

`Integer division by zero` | `Float division by zero` |

Related Examples

- "Review and Comment Results"

Concepts

- "Numerical Defects"

Purpose

Memory allocated dynamically not freed

Description

Memory leak occurs when you do not free a block of memory allocated through `malloc`, `calloc`, or `realloc`. If the memory is allocated in a function `func`, the defect does not occur if:

- Within `func`, you free the memory using the `free` function.
- `func` returns the pointer assigned by `malloc`, `calloc`, or `realloc`.

Examples

Memory leak error

The memory allocated through `malloc` and referenced by `pi` is neither freed nor returned by the function `assign_memory`.

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }


    *pi = 42;
    /* Defect: pi is not freed */
  }
```

Correction — Free Memory

One possible correction is to free the memory referenced by `pi` using the `free` function. The `free` function must be called before the function `assign_memory` terminates

```
#include<stdlib.h>
```

```
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }
    *pi = 42;

    /* Fix: Free the pointer pi*/
    free(pi);
}
```

Correction — Return Pointer from Dynamic Allocation

Another possible correction is to return the pointer `pi`. Returning `pi` allows the function calling `assign_memory` to free the memory block using `pi`.

```
#include<stdlib.h>
#include<stdio.h>

int* assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return(pi);
        }
    *pi = 42;

    /* Fix: Return the pointer pi*/
    return(pi);
}
```

Command-Line Information

Argument: `mem_leak`
Type: string
Default: `off`
Example: `polyspace-bug-finder-nodesktop -checkers mem_leak`

Related Examples

- "Review and Comment Results"

Invalid use of standard library memory routine

Purpose

Standard library memory function called with invalid arguments

Description

Invalid use of standard library memory routine occurs when a memory library function is called with invalid arguments.

Examples

Invalid use of standard library memory routine error

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string `str2` is 5, but 6 characters of string `str1` are copied into `str2` using the `memcpy` function.

Correction — Call Function with Valid Arguments

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];
```

```
 printf("Enter string:\n");
 scanf("%s",str1);

 memcpy(str2,str1,6);
 return str2;
}
```

Command-Line Information

Argument: `mem_std_lib`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers mem_std_lib`

See Also

`Invalid use of standard library string routine` |

Related Examples

- "Review and Comment Results"

Missing null in string array

Purpose

String does not terminate with null character

Description

Missing null in string array occurs when a string does not have enough space to terminate with a null character `'\0'`. This defect can cause various memory errors in your code, so is important to fix it.

This defect applies only for projects in C.

Examples

Array size is too small

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[5] = "THREE";
}
```

The character array `three` has a size of 5 and 5 characters `'T'`, `'H'`, `'R'`, `'E'`, and `'E'`. There is no room for the null character at the end because `three` is only five bytes large.

Correction — Increase array size

One possible correction is to change the array size to allow for all five characters plus a null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[6] = "THREE";
}
```

Correction — Change initialization method

One possible correction is to initialize the string by leaving the array size blank. This initialization method correctly allocates enough memory for all charaters and a terminating-null character.

```
void countdown(int i)
```

```
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[]  = "THREE";
}
```

Command-Line Information

Argument: `missing_null_char`
Type: string
Default: `'off'`
Example: `polyspace-bug-finder-nodesktop -checkers missing_null_char`

Related Examples

- "Review and Comment Results"

Concepts

- "Programming Defects"

Missing or invalid return statement

Purpose

Function does not return value though return type is not `void`.

Description

Missing or invalid return statement occurs when a function does not return a value along at least one execution path. If the return type of the function is `void`, this error does not occur.

Examples

Missing or invalid return statement error

```
int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }
 }
/* Defect: No return value if n is not 0*/
```

If `n` is equal to 0, the code does not enter the `if` statement. Therefore, the function `AddSquares` does not return any value if `n` is 0.

Correction — Place Return Statement on All Execution Paths

One possible correction is to return a value in all branches of the `if...else` statement.

```
 int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
```

```
  {
   for(i=1;i<=n;i++)
      {
       sum+=i^2;
      }
   return(sum);
  }

 /*Fix: Place a return statement on all branches of if-else */
 else
   return 0;
}
```

Command-Line Information

Argument: `missing_return`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers missing_return`

Related Examples

- "Review and Comment Results"

Non-initialized pointer

Purpose

Pointer not initialized before dereference

Description

Non-initialized pointer occurs when a pointer is not assigned an address before dereference.

Examples

Non-initialized pointer error

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
      {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
      }

    *pi = j;
    /* Defect: Writing to uninitialized pointer */

    return pi;
}
```

If `prev` is not `NULL`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on all execution paths, irrespective of whether `prev` is `NULL` or not.

Correction — Initialize Pointer on All Execution Paths

One possible correction is to assign an address to `pi` when `prev` is not `NULL`.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
```

```
{
    int j = 42;
    int* pi;

    if (prev == NULL)
       {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
       }
    /* Fix: Initialize pi in all branches of if statement  */
    else
        pi = prev;


    *pi = j;

    return pi;
}
```

Command-Line Information

Argument: `non_init_ptr`
Type: string
Default: `'On'`
Example: `polyspace-bug-finder-nodesktop -checkers non_init_ptr`

See Also

`Non-initialized variable` |

Related Examples

- "Review and Comment Results"

Pointer to non-initialized value converted to const pointer

Purpose

Pointer to constant assigned address that does not contain a value

Description

Pointer to non initialized value converted to const pointer occurs when a pointer to a constant is assigned an address that does not yet contain a value.

Examples

Pointer to non initialized value converted to const pointer error

```
#include<stdio.h>

void Display_Parity()
 {
  int num,parity;
  const int* num_ptr = &num;
  /* Defect: Address &num does not store any value */

  printf("Enter a number\n:");
  scanf("%d",&num);

  parity=((*num_ptr)%2);
  if(parity==0)
    printf("The number is even.");
  else
    printf("The number is odd.");

 }
```

`num_ptr` is declared as a pointer to a constant. However the variable `num` does not contain any value when `num_ptr` is assigned the address `&num`.

Correction — Store Value in Address Before Assignment to Pointer

One possible correction is to obtain the value of `num` from the user before `&num` is assigned to `num_ptr`.

```
#include<stdio.h>

void Display_Parity()
 {
  int num,parity;
  const int* num_ptr;

  printf("Enter a number\n:");
  scanf("%d",&num);

 /* Fix: Assign &num to pointer after it receives a value */
  num_ptr=&num;
  parity=((*num_ptr)%2);
  if(parity==0)
    printf("The number is even.");
  else
    printf("The number is odd.");
 }
```

The `scanf` statement stores a value in `&num`. Once the value is stored, it is legitimate to assign `&num` to `num_ptr`.

Command-Line Information

Argument: `non_init_ptr_conv`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers non_init_ptr_conv`

Related Examples

- "Review and Comment Results"

Non-initialized variable

Purpose

Variable not initialized before use

Description

Non-initialized variable occurs when a variable is not initialized before its value is read.

Examples

Non-initialized variable error

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
    /* Defect: val does not have a value if command is not 2 */
}
```

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

Correction — Initialize During Declaration

One possible correction is to initialize `val` during declaration so that its value is determined on all execution paths.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    /* Fix: Initialize val */
    int val=0;
```

```
    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
 }
```

`val` is assigned an initial value of 0. When `command` is not equal to 2, the function `get_sensor_value` returns this value.

Command-Line Information

Argument: `non_init_var`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers non_init_var`

See Also

`Non-initialized pointer` |

Related Examples

- "Review and Comment Results"

Null pointer

Purpose

`NULL` pointer derefenced

Description

Null pointer occurs when you use a pointer with a value of `NULL` as if it points to a valid memory location.

Examples

Null pointer error

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 int* p=NULL;

 *p=arr[0];
 /* Defect: Null pointer dereference */

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

The pointer `p` is initialized with value of `NULL`. However, when the value `arr[0]` is written to `*p`, `p` is assumed to point to a valid memory location.

Correction — Assign Address to Null Pointer Before Dereference

One possible correction is to initialize `p` with a valid memory address before dereference.

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
```

```
 /* Fix: Assign address to null pointer */
 int* p=&arr[0];

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

Command-Line Information

Argument: `null_ptr`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers null_ptr`

See Also

`Arithmetic operation with NULL pointer` | `Non-initialized pointer` |

Related Examples

- "Review and Comment Results"

Arithmetic operation with NULL pointer

Purpose

Arithmetic operation performed on `NULL` pointer

Description

Arithmetic operation with NULL pointer occurs when an arithmetic operation involves a pointer whose value is `NULL`.

Examples

Arithmetic operation with NULL pointer error

```
#include<stdlib.h>

int Check_Next_Value(int *loc, int val)
 {
  int *ptr= *loc, found = 0;

  if (ptr==NULL)
   {
      ptr++;
      /* Defect: NULL pointer shifted */

      if (*ptr==val) found=1;
   }

  return(found);
 }
```

When `ptr` is a `NULL` pointer, the code enters the `if` statement body. Therefore, a `NULL` pointer is shifted in the statement `ptr++`.

Correction — Avoid NULL Pointer Arithmetic

One possible correction is to perform the arithmetic operation when `ptr` is not `NULL`.

```
#include<stdlib.h>

int Check_Next_Value(int *loc, int val)
 {
  int *ptr= *loc, found = 0;
```

```
 /* Fix: Perform operation when ptr is not NULL */
 if (ptr!=NULL)
  {
     ptr++;

     if (*ptr==val) found=1;
  }

 return(found);
}
```

Command-Line Information

Argument: `null_ptr_arith`
Type: string
Default: `'off'`
Example: `polyspace-bug-finder-nodesktop -checkers null_ptr_arith`

See Also

`Null pointer` |

Related Examples

- "Review and Comment Results"

Invalid use of standard library routine

Purpose

Wrong arguments to standard library function

Description

Invalid use of standard library routine occurs when you use invalid arguments with a function from the standard library. This defect picks up errors related to any other functions not covered by float, integer, memory, or string standard library routines.

Examples

Calling `printf` Without a String

```
void print_null(void) {

  printf(NULL);
}
```

The function `printf` takes only string input arguments or format specifiers. In this function, the input value is NULL, which is not a valid string.

Correction — Use Correct Input Arguments

One possible correction is to change the input arguments to fit the requirements of the standard library routine. In this example, the input argument was changed to a character.

```
void print_null(void) {
    char zero_val = '0';
    printf(zero_val);
}
```

Command-Line Information

Argument: `other_std_lib`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers other_std_lib`

See Also

`Invalid use of standard library integer routine` | `Invalid use of standard library floating point routine` | `Invalid use`

`of standard library memory routine | Invalid use of standard library string routine |`

Related Examples

- “Review and Comment Results”

Concepts

- “Other Defects”

Array access out of bounds

Purpose

Array index outside bounds during array access

Description

Array access out of bounds occurs when an array index falls outside the range `[0...array_size-1]` during array access.

Examples

Array access out of bounds error

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

Correction — Keep Array Index Within Array Bounds

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>
```

```
void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

Command-Line Information

Argument: `out_bound_array`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers out_bound_array`

See Also

`Pointer access out of bounds` |

Related Examples

- "Review and Comment Results"

Pointer access out of bounds

Purpose

Pointer dereferenced outside its bounds

Description

Pointer access out of bounds occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

Examples

Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

Correction — Ensure Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
```

```
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced any more.

Command-Line Information

Argument: `out_bound_ptr`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers out_bound_ptr`

See Also

`Array access out of bounds` |

Related Examples

- "Review and Comment Results"

Partially access array

Purpose

Array partly read or written before end of scope

Description

Partially access array occurs when an array is partially read or written before the end of array scope. For arrays local to a function, the end of scope occurs when the function ends.

Examples

Partially access array error

```
int Calc_Sum(void)
{
  int tab[5]={0,1,2,3,4},sum=0;
  /* Defect: tab[4] is not read */

  for (int i=0; i<4;i++) sum+=tab[i];

  return(sum);

 }
```

The array `tab` is only partially read before end of function `Calc_Sum`. While calculating `sum`, `tab[4]` is not included.

Correction — Access All Elements of Array

One possible correction is to read all elements of array `tab`.

```
int Calc_Sum(void)
{
  int tab[5]={0,1,2,3,4},sum=0;

  /* Fix: Include tab[4] in calculating sum */
  for (int i=0; i<5;i++) sum+=tab[i];

  return(sum);

 }
```

Command-Line Information

Argument: `partially_access_array`
Type: string
Default: `'off'`
Example: `polyspace-bug-finder-nodesktop -checkers partially_access_array`

Related Examples

- "Review and Comment Results"

Large pass-by-value argument

Purpose

Large argument passed between functions by value

Description

Large pass-by-value argument occurs when a large input argument or return value is passed between functions by its value. For variables larger than 64 bytes, pass the value by pointer or by reference to save stack space and copy time.

Examples

Passing a Large `struct` Between Functions

```
typedef struct s_userid {
    char name[2];
    int idnumber[100];
} userid;

char username(userid first) {
    return first.name[0];
}
```

The large structure, `userid`, is passed to the function `username`. Because `userid` is larger than 64 bytes, this function produces a large pass-by-value defect.

Correction — Pass-By-Reference

One possible correction is to pass the argument by reference instead of by value. In this example, the pointer to a `userid` structure is passed instead of the actual structure.

```
typedef struct s_userid {
    char name[2];
    int idnumber[100];
} userid;

char username(userid *first) {
    return (*first).name[0];
}
```

Command-Line Information

Argument: `pass_by_value`
Type: string
Default: `'off'`
Example: `polyspace-bug-finder-nodesktop -checkers pass_by_value`

Related Examples

- "Review and Comment Results"

Concepts

- "Other Defects"

Unreliable cast of pointer

Purpose

Pointer implicitly cast to different data type

Description

Unreliable cast of pointer occurs when a pointer is implicitly cast to a data type different from its declaration type. Such an implicit casting can take place, for instance, when a pointer to data type `char` is assigned the address of an integer.

This defect applies only if the code language for the project is C.

Examples

Unreliable cast of pointer error

```
#include <string.h>

void Copy_Integer_To_String()
{
 int src[]={1,2,3,4,5,6,7,8,9,10};
 char buffer[]="Buffer_Text";
 strcpy(buffer,src);
 /* Defect: Implicit cast of (int*) to (char*) */
}
```

`src` is declared as an `int*` pointer. The `strcpy` statement, while copying to `buffer`, implicitly casts `src` to `char*`.

Correction — Avoid Pointer Cast

One possible correction is to declare the pointer `src` with the same data type as `buffer`.

```
#include <string.h>
 void Copy_Integer_To_String()
{
 /* Fix: Declare src with same type as buffer */
 char *src[10]={"1","2","3","4","5","6","7","8","9","10"};
 char *buffer[10];

 for(int i=0;i<10;i++)
   buffer[i]="Buffer_Text";
```

```
for(int i=0;i<10;i++)
  buffer[i]= src[i];
}
```

Command-Line Information

Argument: `ptr_cast`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers ptr_cast`

See Also

`Unreliable cast of function pointer` |

Related Examples

- "Review and Comment Results"

Wrong type used in sizeof

Purpose

`sizeof` argument does not match pointer type

Description

Wrong type used in sizeof occurs when the size specified for the block of memory does not match the pointer type being initialized.

Examples

Allocate a Char Array With `sizeof`

```
void test_case_1(void) {
    char* str;

    str = malloc(sizeof(char*) * 5);
    free(str);

}
```

In this example, memory is allocated for the character pointer `str` using a `malloc` of five char pointers. However, `str` is a pointer to a character, not a pointer to a character pointer. Therefore the `sizeof` argument, `char*`, is incorrect.

Correction — Match Pointer Type to `sizeof` Argument

One possible correction is to match the argument to the pointer type. In this example, `str` is a character pointer, therefore the argument must also be a character.

```
void test_case_1(void) {
    char* str;

    str = malloc(sizeof(char) * 5);
    free(str);

}
```

Command-Line Information

Argument: `ptr_sizeof_mismatch`
Type: string
Default: `'on'`

Example: `polyspace-bug-finder-nodesktop -checkers ptr_sizeof_mismatch`

Related Examples

- "Review and Comment Results"

Concepts

- "Programming Defects"

Qualifier removed in conversion

Purpose

Variable qualifier is lost during conversion

Description

Qualifier removed in conversion occurs during a conversion when one variable has a qualifier and the other does not. For example, when converting from a `const int` to an `int`, the conversion removes the `const` qualifier.

This defect applies only for projects in C.

Examples

Cast of Character Pointers

```
void implicit_cast(void) {
    const char  cc, *pcc = &cc;
    char * quo;

    quo = &cc;
    quo = pcc;

    read(quo);
}
```

During the assignment to the character `q`, the variables, `cc` and `pcc`, are converted from `const char` to `char`. The `const` qualifier is removed during the conversion causing a defect.

Correction — Add Qualifiers

One possible correction is to add the same qualifiers to the new variables. In this example, changing `q` to a `const char` fixes the defect.

```
void implicit_cast(void) {
    const char  cc, *pcc = &cc;
    const char * quo;

    quo = &cc;
    quo = pcc;

    read(quo);
}
```

Correction — Remove Qualifiers

One possible correction is to remove the qualifiers in the converted variable. In this example, removing the `const` qualifier from the `cc` and `pcc` initialization fixes the defect.

```
void implicit_basic_cast(void) {
    char  cc, *pcc = &cc;
    char * quo;

    quo = &cc;
    quo = pcc;

    read(quo);
}
```

Command-Line Information

Argument: `qualifier_mismatch`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers qualifier_mismatch`

Related Examples

- "Review and Comment Results"

Concepts

- "Programming Defects"

Race conditions

Purpose

Race conditions between multiple instances of the same variable

Description

Race conditions occur in multitasking code when two parallel task change the same variable. A race condition occurs because both tasks are racing to be the first to use the variable.

This defect is associated with the multitasking and entry point options.

Examples

Simple Function Race

```
int var_for_rc;
void race_condition(void) {
    var_for_rc++;
}
void task1(void) { race_condition(); }
void task2(void) { race_condition(); }
```

In this example, the tasks `task1` and `task2` were specified as the entry points to the multitasking code. Both tasks call the same function which uses external variable `var_for_rc`. A race condition occurs because`var_for_rc` is changing in two parallel tasks.

Correction — Exclusive Task

One possible correction is to change which tasks are parallel and which are exclusive. Change `task1` and `task2` to be nonparallel multitasking tasks. The code is the same, but how the code is built changes.

Command-Line Information

Argument: `race_cond`
Type: string
Default: `'off'`
Example: `polyspace-bug-finder-nodesktop -entry-points task1,task2 -checkers race_cond`

See Also

"Multitasking" |

Related Examples

- "Review and Comment Results"

Concepts

- “Other Defects”

Shift of a negative value

Purpose

Shift operator on negative value

Description

Shift of a negative value occurs when a bit-wise shift is used on a negative number. Shifts can overwrite the sign bit that identifies a number as negative.

Examples

Shifting a negative variable

```
int shifting(int val)
{
    int res = -1;
    return res << val;
}
```

In the return statement, the variable `res` is shifted a certain number of bits to the left. However, because `res` is negative, the shift might overwrite the sign bit.

Correction — Change the Data Type

One possible correction is to change the data type of the shifted variable to unsigned. This correction eliminates the sign bit, so left shifting does not affect the value.

```
int shifting(int val)
{
    unsigned int res = -1;
    return res << val
}
```

Command-Line Information

Argument: `shift_neg`
Type: string
Default: `'off'`
Example: `polyspace-bug-finder-nodesktop -checkers shift_neg`

See Also

`Shift operation overflow` |

Related Examples

- "Review and Comment Results"

Concepts

- "Numerical Defects"

Shift operation overflow

Purpose

Overflow from shifting operation

Description

Shift operation overflow occurs when a shift operation exceeds the space available to represent the resulting value.

The exact storage allocation for different data types depends on your operating system. See “Predefined Target Processor Specifications”.

Examples

Left Shift of Integer

```
int left_shift(void) {

    int foo = 33;
    return 1 << foo;
}
```

In the return statement of this function, bit-wise shift operation is performed shifting 1 `foo` bits to the left. However, an `int` has only 32 bits, so the range of the shift must be between 0 and 31. Therfore, this shift operation causes an overflow.

Correction — Different storage type

One possible correction is to store the shift operation result in a larger data type. In this example, by returning a `long` instead of an `int`, the overflow defect is fixed.

```
long left_shift(void) {

    int foo = 33;
    return 1 << foo;
}
```

Command-Line Information

Argument: `shift_ovfl`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers shift_ovfl`

Related Examples

- "Review and Comment Results"

Concepts

- "Numerical Defects"

Sign change integer conversion overflow

Purpose

Overflow when converting between signed and unsigned integers

Description

Sign change integer conversion overflow occurs when converting an unsigned integer to a signed integer. If there are not enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different integer types depends on your operating system. See "Predefined Target Processor Specifications".

Examples

Convert from `unsigned char` to `char`

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count;
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

Command-Line Information

Argument: `sign_change`
Type: string
Default: `'on'`

Example: `polyspace-bug-finder-nodesktop -checkers sign_change`

See Also

`Float conversion overflow` | `Unsigned integer conversion overflow` | `Integer conversion overflow` |

Related Examples

- "Review and Comment Results"

Concepts

- "Numerical Defects"

Invalid use of standard library string routine

Purpose

Standard library string function called with invalid arguments

Description

Invalid use of standard library string routine occurs when a string library function is called with invalid arguments.

Examples

Invalid use of standard library string routine error

```
#include <string.h>
#include <stdio.h>

char* Copy_String(void)
{
 char *res;
 char gbuffer[5],text[20]="ABCDEFGHIJKL";

 res=strcpy(gbuffer,text);
 /* Error: Size of text is less than gbuffer */

 return(res);
}
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot successfully copy `text` into `gbuffer`.

Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: Ensure that gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";
```

```
 res=strcpy(gbuffer,text);

 return(res);
}
```

Command-Line Information

Argument: `str_std_lib`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers str_std_lib`

See Also

`Invalid use of standard library memory routine` |

Related Examples

- "Review and Comment Results"

Format string specifiers and arguments mismatch

Purpose

String specifiers do not match corresponding arguments

Description

Format string specifiers and arguments mismatch occurs when the parameters in the format specification do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

Examples

Printing a Float

```
void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", fst);
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
void string_format(void) {

    unsigned long fst = 1;
```

```
    printf("%d\n", (int)fst);
}
```

Command-Line Information

Argument: `string_format`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers string_format`

See Also

`Invalid use of standard library string routine` |

Related Examples

- "Review and Comment Results"

Concepts

- "Other Defects"

External Web Sites

- http://en.cppreference.com/w/cpp/io/c/fprintf

Unsigned integer conversion overflow

Purpose

Overflow when converting between unsigned integer types

Description

Unsigned integer conversion overflow occurs when converting an unsigned integer to a smaller unsigned integer type. If there are not enough bytes to represent the original constant, the conversion overflows.

The exact storage allocation for different integer types depends on your operating system. See "Predefined Target Processor Specifications".

Examples

Converting from `int` to `char`

```
unsigned char convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned char)unum;
}
```

In the return statement, the unsigned integer variable `unum` is converted to an unsigned character type. However, the conversion overflows because 1000000 requires at least 20 bits. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum possible value plus 1. In this example, `unum` is reduced by modulo `2^8` because a character data type can only represent `2^8-1`.

Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number. For example, `long`.

```
unsigned long convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned long)unum;
}
```

Command-Line Information

Argument: `uint_conv_ovfl`
Type: string

Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers uint_conv_ovfl`

See Also

`Float conversion overflow` | `Integer conversion overflow` | `Sign change integer conversion overflow` |

Related Examples

- "Review and Comment Results"

Concepts

- "Numerical Defects"

Unsigned integer overflow

Purpose

Overflow from operation between unsigned integers

Description

Unsigned integer overflow occurs when an operation on unsigned integer variables exceeds the space available to represent the resulting value.The exact storage allocation for different integer types depends on your operating system. See “Predefined Target Processor Specifications”.

Examples

Add One to Maximum Unsigned Integer

```
unsigned int plusplus(void) {

    unsigned uvar = UINT_MAX;
    uvar++;
    return uvar;
}
```

In the third statement of this function, the variable `uvar` is increased by 1. However, the value of `uvar` is the maximum unsigned integer value, so 1 plus the maximum integer value cannot be represented by an `unsigned int`. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum possible value plus 1. In this example, `uvar` is reduced by modulo `UINT_MAX`. The result is `uvar = 1`.

Correction — Different Storage Type

One possible correction is to store the operation result in a larger data type. In this example, by returning an `unsigned long` instead of an `unsigned int`, the overflow error is fixed.

```
unsigned long plusplus(void) {

    unsigned uvar = UINT_MAX;
    unsigned long ulvar = uvar++;
    return ulvar;
}
```

Command-Line Information

Argument: `uint_ovfl`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers uint_ovfl`

See Also

`Integer overflow` | `Float overflow` |

Related Examples

- "Review and Comment Results"

Concepts

- "Numerical Defects"

Uncalled function

Purpose

Function with static scope never called in file

Description

Uncalled function occurs when a `static` function is not called in the same file where it is defined.

Examples

Uncalled function error

Save the following code in the file `Initialize_Value.c`

```
#include <stdlib.h>
#include <stdio.h>

static int Initialize(void)
/* Defect: Function never called */
  {
   int input;
   printf("Enter an integer:");
   scanf("%d",&input);
   return(input);
  }

 void main()
  {
   int num;

   num=0;

   printf("The value of num is %d",num);
  }
```

The `static` function `Initialize` is never called in the file `Initialize_Value.c`.

Correction — Call Function at Least Once

One possible correction is to call `Initialize` at least once in the file `Initialize_Value.c`.

```
#include <stdlib.h>
```

```
#include <stdio.h>

static int Initialize(void)
  {
   int input;
   printf("Enter an integer:");
   scanf("%d",&input);
   return(input);
  }

 void main()
  {
   int num;

   /* Fix: Call static function Initialize */
   num=Initialize();

   printf("The value of num is %d",num);
  }
```

Command-Line Information

Argument: `uncalled_func`
Type: string
Default: `'off'`
Example: `polyspace-bug-finder-nodesktop -checkers uncalled_func`

Related Examples

- "Review and Comment Results"

Unprotected dynamic memory allocation

Purpose

Pointer returned from dynamic allocation not checked for `NULL` value

Description

Unprotected dynamic memory allocation occurs when the code does not check for the success of a dynamic memory allocation.

When memory is dynamically allocated using `malloc`, `calloc`, or `realloc`, it returns a value `NULL` if the requested memory is not available. If the code following the allocation accesses the memory block without checking for the `NULL` value, this access is not protected from failures.

Examples

Unprotected dynamic memory allocation error

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));

  *p = 2;
  /* Defect: p is not checked for NULL value */

  free(p);
}
```

If the memory allocation is not successful, the function `calloc` returns `NULL` to `p`. Before accessing the memory through `p`, the code does not check whether memory allocation has been successful.

Correction — Check for NULL Value

One possible correction is to check whether `p` has value `NULL` before dereference.

```
#include <stdlib.h>

void Assign_Value(void)
 {
   int* p = (int*)calloc(5, sizeof(int));
```

```
  /* Fix: Check if p is NULL */
  if(p!=NULL) *p = 2;

  free(p);
}
```

Command-Line Information

Argument: `unprotected_memory_allocation`
Type: string
Default: `off`
Example: `polyspace-bug-finder-nodesktop -checkers unprotected_memory_allocation`

Related Examples

- "Review and Comment Results"

Write without further read

Purpose

Variable never read after assignment

Description

Write without further read occurs when a value assigned to a variable is never read.

Examples

Write without further read error

```
void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();
    /* Defect: Useless write */
}
```

After the variable `level` gets assigned the value `4 * getsensor()`, it is never read.

Correction — Use Value After Assignment

One possible correction is to use the variable `level` after the assignment.

```
void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();

    /* Fix: Use level after assingment */
    printf('The value is %d', level)

}
```

The variable `level` is printed, reading the new value.

Command-Line Information

Argument: `useless_write`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers useless_write`

Related Examples

- "Review and Comment Results"

Variable shadowing

Purpose

Variable hides another variable of same name with nested scope

Description

Variable shadowing occurs when a variable hides another variable of the same name with nested scope.

Examples

Variable Shadowing error

```
#include <stdio.h>

int fact[5]={1,2,6,24,120};

int factorial(int n)
 {
  int fact=1;
  /*Defect: Local variable hides global array with same name */

  for(int i=1;i<=n;i++)
    fact*=i;

  return(fact);
 }
```

Inside the `factorial` function, the integer variable `fact` hides the global integer array `fact`.

Correction — Change Variable Name

One possible correction is to change the name of one of the variables, preferably the one with more local scope.

```
#include <stdio.h>

int fact[5]={1,2,6,24,120};

int factorial(int n)
 {
  /* Fix: Change name of local variable */
  int f=1;
```

```
 for(int i=1;i<=n;i++)
   f*=i;

 return(f);
}
```

Command-Line Information

Argument: `var_shadowing`
Type: string
Default: `'on'`
Example: `polyspace-bug-finder-nodesktop -checkers var_shadowing`

Related Examples

- "Review and Comment Results"

Variable shadowing

6

Functions

PolyspaceAnnotation

Purpose

Annotate Simulink blocks with known Polyspace results

Syntax

```
PolyspaceAnnotation('type',typeValue,'kind',kindValue,Name,
   Value)
```

Description

`PolyspaceAnnotation('type',typeValue,'kind',kindValue,Name,Value)` adds an annotation of type `typeValue` and kind `kindValue` to the currently selected block in the model. You can also specify a different block using a `Name,Value` pair argument. You can also add notes about a priority classification, an action status, or other comments using `Name,Value` pairs.

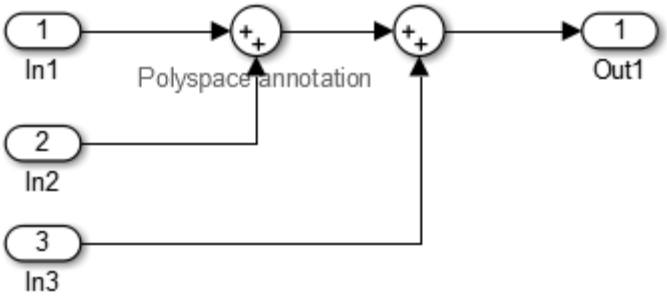
In the generated code associated with the annotated block, code comments are added before and after the lines of code. Polyspace reads these comments and marks any Polyspace results of the specified `kind` with the annotated information.

When you add annotations, you can identify known errors and coding rule violations to focus on new results.

Limitations

- You can have only one annotation per block. If a block produces both a rule violation and an error, only one type can be annotation.
- Even though you apply annotations to individual blocks, the scope of the annotation may be larger. The generated code from one block can overlap with another causing the annotation to also overlap.

  For example, consider this model.



The first summation block has a Polyspace annotation, but the second does not. However, the associated generated code adds all three inputs in one line of code. Therefore, the annotation justifies both summations:

```
/*
* polyspace:begin<RTE:OVFL:Medium:Fix>
*/
annotate_y.Out1 = (annotate_u.In1 + annotate_U.In2) + annotate_U.In3

/* polyspace:end<RTE:OVFL:Medium:Fix> */
```

Input Arguments

typeValue - type of result

`'MISRA-C' | 'MISRA-CPP' | 'JSF'`

The type of result with which to annotate the block, specified as:

- `` `MISRA-C' `` for MISRA C coding rule violations (C code only).
- `` `MISRA-CPP' `` for MISRA C++ coding rule violations (C++ code only).
- `` `JSF' `` for JSF C++ coding rule violations (C++ code only).

Example: `` `type','MISRA-C' ``

kindValue - specific check or coding rule

check acronym | rule number

The specific check or coding rule specified by the acronym of the check or the coding rule number. For the specific input for each type of annotation, see the following table.

Type Value	Kind Values
`'MISRA-C'`	Use the rule number you want to annotate. For example, `'2.2'`. For the list of supported MISRA C rules and their numbers, see "MISRA C Coding Rules".
`'MISRA-CPP'`	Use the rule number you want to annotate. For example, `'0-1-1'`. For the list of supported MISRA C++ rules and their numbers, see "MISRA C++ Coding Rules".
`'JSF'`	Use the rule number you want to annotate. For example, `'3'`. For the list of supported JSF C++ rules and their numbers, see "JSF C++ Coding Rules".

Example:

```
PolyspaceAnnotation('type','MISRA-CPP','kind','1-2-3')
```

Data Types
`char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'block','MyModel\Sum', 'status','fix'

'block' - block to be annotated

`gcb` (default) | block name

Block to be annotated specified by the block name. If you do not use this option, the block returned by the function `gcb` is annotated.

Example: `'block','MyModel\Sum'`

'class' - classification of the check

`'high' | 'medium' | 'low' | 'not a defect' | 'unset'`

Classification of the check specified as `high`, `medium`, `low`, `not a defect`, or `unset`.

Example: `'class','high'`

'status' - action status

`'undecided' | 'investigate' | 'fix' | 'improve' | 'restart with different options' | 'justify with annotation' | 'no action planned' | 'other'`

Action status of the check specified as `undecided`, `investigate`, `fix`, `improve`, `restart with different options`, `justify with annotation`, `no action planned`, or `other`.

Example: `'status','no action planned'`

'comment' - additional comments

string

Additional comments specified as a string. The comments provide more information about why the results are justified.

Example: `'comment','defensive code'`

Examples

Annotate a Block and Run a Polyspace Bug Finder Analysis

Use the Polyspace annotation function to annotate a block and see the annotation in the analysis results.

At the MATLAB command line, load and open the example model WhereAreTheErrors_v2:

```
WhereAreTheErrors_v2
```

Add an annotation to the switch block to annotate any violations to MISRA C rule 13.7. Also, add to the annotation a comment, a classification, and a status.

```
PolyspaceAnnotation('type','Misra-C', 'kind', '13.7','block',...
'WhereAreTheErrors_v2/Switch1','status','improve','comment','look into la
```

In the WhereAreTheErrors_v2 model in Simulink®, you can see a Polyspace annotation added to the switch block.

At the MATLAB command line, generate code for the model:

```
slbuild('WhereAreTheErrors_v2');
```

Run an analysis on your model:

```
pslinkrun('WhereAreTheErrors_v2');
```

After the analysis is finished, open the results in the Polyspace environment:

```
PolySpaceViewer('WhereAreTheErrors_v2');
```

Results 10–14 are all MISRA C 13.7 rule violations. The annotation information that you added to the switch block appears in all four results, because all four results are from the switch block.

See Also

`pslinkoptions | PolySpaceViewer | pslinkrun | gcb`

Concepts

- "MATLAB Functions for Polyspace Batch Runs"

Purpose

Create options object to customize Polyspace runs from MATLAB command line

Syntax

```
opts = pslinkoptions(codegen)
opts = pslinkoptions(model)
```

Description

`opts = pslinkoptions(codegen)` returns an options object with the configuration options for code generated by `codegen`.

`opts = pslinkoptions(model)` returns an options object with the configuration options for the Simulink model `model`.

Input Arguments

codegen - Code generator

`'ec' | 'tl'`

Code generator, specified as either `'ec'` for Embedded Coder® or `'tl'` for TargetLink®. Each argument creates a Polyspace options object with configuration options specific to that code generator.

For a description of all configuration options and their values, see .

Example: `embedded_coder_opt = pslinkoptions('ec')`

Example: `target_link_opt = pslinkoptions('tl')`

Data Types
`char`

model - Simulink model

`model name`

Simulink model, specified by the model name. Creates a Polyspace options object with the configuration options of that model. If no options have been set, the object has all default configuration options. If a code generator has been set, the object has the default options for that code generator.

For a description of all configuration options and their values, see .

Example: `model_opt = pslinkoptions('my_model')`

Data Types
`char`

Output Arguments

opts - Polyspace configuration options

options object

Polyspace configuration options, returned as an options object. The object is used with `pslinkrun` to run a Polyspace from the MATLAB command line.

The following table provides possible values and a description for each configuration option. Depending on the code generator, the object will have different configuration options. The value in curly brackets `{}` is the default.

Configuration Option	Values	Description
`ResultDir`	`{'C:\Polyspace_Results\results_$`*`ModelName`*`$'}`	Specify location of results folder. Can be either an absolute path or a path relative to the current folder.
`VerificationSettings`	`{'PrjConfig'} | 'PrjConfigAndMisraAGC' | 'PrjConfigAndMisra' | 'MisraAGC' | 'Misra'`	Specify checking of coding rules for C: **`'PrjConfig'`** – Inherit all options from project configuration and run complete analysis. **`'PrjConfigAndMisraAGC'`** – Inherit all options from project configuration, enable MISRA AC AGC rule checking, and run complete analysis. **`'PrjConfigAndMisra'`** – Inherit all options from project configuration, enable MISRA C rule

Configuration Option	Values	Description
		checking, and run complete analysis. `'MisraAGC'` – Enable MISRA AC AGC rule checking, and run compilation phase only. `'Misra'` – Enable MISRA C rule checking, and run compilation phase only.
`OpenProjectManager`	`{false} | true`	Open Polyspace Metrics or Project Manager to monitor the progress. Afterward, you can switch to the Results Manager perspective to review the results.
`AddSuffixToResultDir`	`{false} | true`	Modify location of results folder by appending a unique number to the folder name instead of overwriting an existing folder.
`EnableAdditionalFileList`	`{false} | true`	Specify whether additional files must be analyzed. You can specify these additional files with the `AdditionalFileList` option
`AdditionalFileList`	`{0x1 cell}`	List additional files to analyze.

Configuration Option	Values	Description
`InputRangeMode`	`{'DesignMinMax'}	'FullRange'`	Specify whether to use data ranges defined in blocks and workspace or treat inputs as full-range values.
`ParamRangeMode`	`{'None'}	'DesignMinMax'`	Specify whether to use constant values of parameters specified in the code, or use a range defined in blocks and workspace.
`OutputRangeMode`	`{'None'}	'DesignMinMax'`	Specify whether to apply assertions to outputs (using a range defined in blocks and workspace).
`VerificationMode`	`{'BugFinder'}	'CodeProver'`	Specify whether to run a Bug Finder analysis or Code Prover verification.
`AutoStubLUT` *Only for TargetLink*	`{false}	true`	Specify whether to include Lookup Table code in the analysis.

Configuration Option	Values	Description
`ModelRefVerifDepth` *Only for Embedded Coder*	`{'Current model only'} | '1' | '2' | '3' | 'All'`	Specify analysis of generated code with respect to model reference hierarchy levels.
`ModelRefByModelRefVerif` *Only for Embedded Coder*	`{false} | true`	Specify whether to analyze code from models within model reference hierarchies jointly or separately.
`CxxVerificationSettings` *Only for Embedded Coder*	`{'PrjConfig'} | 'PrjConfigAndMisraCxx' | 'PrjConfigAndJSF' | 'MisraCxx' | 'JSF'`	Specify checking of coding rules for C++: **`'PrjConfig'`** – Inherit all options from project configuration and run complete analysis. **`'PrjConfigAndMisraCxx'`** – Inherit all options from project configuration, enable MISRA C++ rule checking, and run complete analysis. **`'PrjConfigAndJSF'`** – Inherit all options from project configuration, enable JSF rule checking, and run complete analysis. **`'MisraCxx'`** – Enable MISRA C++ rule checking, and run compilation phase only.

Configuration Option	Values	Description
		`'JSF'` – Enable JSF rule checking, and run compilation phase only.

Examples

Use a Simulink model to create and edit an options objects

Load the Simulink model `psdemo_model_link_sl`:

```
load_system('psdemo_model_link_sl_v2')
```

From the MATLAB command line, create a Polyspace options object from the model:

```
model_opt = pslinkoptions('psdemo_model_link_sl_v2')
```

```
model_opt =

                  ResultDir: 'results_$ModelName$'
        VerificationSettings: 'PrjConfig'
          OpenProjectManager: 0
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {0x1 cell}
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
            VerificationMode: 'BugFinder'
          ModelRefVerifDepth: 'Current model only'
     ModelRefByModelRefVerif: 0
     CxxVerificationSettings: 'PrjConfig'
```

The model is already configured for Embedded Coder, so only the Embedded Coder configuration options appear.

Change the results folder name option:

```
model_opt.ResultDir = 'results_v1_$ModelName$';

model_opt =

                  ResultDir: 'results_v1_$ModelName$'
        VerificationSettings: 'PrjConfig'
          OpenProjectManager: 0
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {0x1 cell}
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
            VerificationMode: 'BugFinder'
          ModelRefVerifDepth: 'Current model only'
     ModelRefByModelRefVerif: 0
     CxxVerificationSettings: 'PrjConfig'
```

Set the `OpenProjectManager` to true, to monitor progress in the Polyspace interface.

```
model_opt.OpenProjectManager = true

model_opt =

                  ResultDir: 'results_v1_$ModelName$'
        VerificationSettings: 'PrjConfig'
          OpenProjectManager: 1
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {0x1 cell}
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
            VerificationMode: 'BugFinder'
          ModelRefVerifDepth: 'Current model only'
     ModelRefByModelRefVerif: 0
```

```
     CxxVerificationSettings: 'PrjConfig'
```

Create and edit an options object for Embedded Coder at the command line

Create a Polyspace options object called `new_opt` with Embedded Coder parameters:

```
new_opt = pslinkoptions('ec')
```

```
new_opt =

                  ResultDir: 'results_$ModelName$'
        VerificationSettings: 'PrjConfig'
          OpenProjectManager: 0
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {0x1 cell}
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
            VerificationMode: 'BugFinder'
          ModelRefVerifDepth: 'Current model only'
     ModelRefByModelRefVerif: 0
     CxxVerificationSettings: 'PrjConfig'
```

Set the `OpenProjectManager` option to true to follow the progress in the Polyspace interface:

```
new_opt.OpenProjectManager = true
```

```
new_opt =

                  ResultDir: 'results_$ModelName$'
        VerificationSettings: 'PrjConfig'
          OpenProjectManager: 1
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {0x1 cell}
```

```
                InputRangeMode: 'DesignMinMax'
                ParamRangeMode: 'None'
               OutputRangeMode: 'None'
              VerificationMode: 'BugFinder'
            ModelRefVerifDepth: 'Current model only'
      ModelRefByModelRefVerif: 0
      CxxVerificationSettings: 'PrjConfig'
```

Change the configuration to check for both run-time errors and MISRA C coding rule violations:

```
new_opt.VerificationSettings = 'PrjConfigAndMisra'
```

```
new_opt =

                    ResultDir: 'results_$ModelName$'
          VerificationSettings: 'PrjConfigAndMisra'
            OpenProjectManager: 1
          AddSuffixToResultDir: 0
     EnableAdditionalFileList: 0
            AdditionalFileList: {0x1 cell}
                InputRangeMode: 'DesignMinMax'
                ParamRangeMode: 'None'
               OutputRangeMode: 'None'
              VerificationMode: 'BugFinder'
            ModelRefVerifDepth: 'Current model only'
      ModelRefByModelRefVerif: 0
      CxxVerificationSettings: 'PrjConfig'
```

See Also

`PolySpaceViewer | pslinkrun | PolyspaceAnnotation`

Concepts

- “MATLAB Functions for Polyspace Batch Runs”

pslinkrun

Purpose

Run Polyspace analysis on generated code from MATLAB command line

Syntax

```
resultsFolder = pslinkrun
resultsFolder = pslinkrun(system)
resultsFolder = pslinkrun(system,opts)
resultsFolder = pslinkrun(system,opts,asModelRef)
```

Description

`resultsFolder = pslinkrun` on generated code from the current system and returns the location of the results folder. It uses the analysis options associated with the current system. The current system, or model, is the system returned by the command `bdroot`.

`resultsFolder = pslinkrun(system)` runs Polyspace on the code generated from the model or subsystem specified by `system`. It uses the analysis options associated with `system`.

`resultsFolder = pslinkrun(system,opts)` analyzes `system` using the analysis options from the options object `opts`.

`resultsFolder = pslinkrun(system,opts,asModelRef)` uses `asModelRef` to specify which type of generated code to analyze, standalone code or model reference code. This option is useful when you want to analyze only a referenced model instead of an entire model hierarchy.

Input Arguments

system - Model or system

`bdroot` (default) | model or system name

Model or system that you want to analyze, specified as a string, with the model or system name in single quotes. The default value is the system returned by `bdroot`.

Example: `resultsFolder = pslinkrun('demo')` where `demo` is the name of a model.

Data Types
`char`

opts - Analysis options

options associated with `system` (default) | Polyspace options object

Analysis options for the analysis, specified as an options object or the options already associated with the model or system. The function `pslinkoptions` creates an options object. You can customize the options object by changing the

Example: `pslinkrun('demo', opts_demo)` where `demo` is the name of a model and `opts_demo` is an options object.

asModelRef - Indicator for model reference analysis

`false` (default) | `true`

Indicator for model reference analysis, specified as true or false.

- If `asModelRef` is false (default), Polyspace analyzes code generated as standalone code. This option is equivalent to choosing **Verify Code Generated For > Model** in the Simulink Polyspace options.
- If `asModelRef` is true, Polyspace analyzes code generated as model referenced code. This option is equivalent to choosing **Verify Code Generated For > Referenced Model** in the Simulink Polyspace options.

Data Types
`logical`

Output Arguments

resultsFolder - Variable for location of the results folder

string

Variable for location of the results folder, specified as a string. The default value of this variable is `results_$ModelName$`. You can change this value in the configuration options using `pslinkoptions`.

Data Types
`char`

Examples

Run Polyspace from the Command Line

Use a Simulink model to generate code, set configuration options, and then run an analysis from the command line.

Create a variable `model` to store the name of the Polyspace example model, `WhereAreTheErrors_v2`:

```
model = 'WhereAreTheErrors_v2';
```

This step is not necessary to use the function, but will make the rest of the example easier.

Load the model:

```
load_system(model);
```

From the MATLAB command line, build the model to generate code:

```
slbuild(model);
```

Create a Polyspace options object from the model:

```
opts = pslinkoptions(model)
```

```
opts =

                  ResultDir: 'results_$ModelName$'
        VerificationSettings: 'PrjConfig'
          OpenProjectManager: 0
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {0x1 cell}
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
            VerificationMode: 'CodeProver'
          ModelRefVerifDepth: 'Current model only'
     ModelRefByModelRefVerif: 0
```

```
     CxxVerificationSettings: 'PrjConfig'
```

Change the configuration to run a Bug Finder analysis instead of a Code Prover verification:

```
opts.VerificationMode = 'BugFinder'
```

```
opts =

                  ResultDir: 'results_$ModelName$'
        VerificationSettings: 'PrjConfig'
          OpenProjectManager: 0
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {0x1 cell}
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
            VerificationMode: 'BugFinder'
          ModelRefVerifDepth: 'Current model only'
     ModelRefByModelRefVerif: 0
     CxxVerificationSettings: 'PrjConfig'
```

Run Polyspace using your options object:

```
results = pslinkrun(model,opts)
```

The results are saved to the folder `results_WhereAreTheErrors_v2`.

Build and Analyze Referenced Model Code from the Command Line

Use a Simulink model to generate reference code, set configuration options, and then run an analysis from the command line.

Create a variable `model` to store the name of the Polyspace example model, `WhereAreTheErrors_v2`:

```
model = 'WhereAreTheErrors_v2';
```

This step is not necessary to use the function, but will make the rest of the example easier.

Load the model:

```
load_system(model)
```

From the MATLAB command line, build the model to generate code as if it is referenced by another model:

```
slbuild(model,'ModelReferenceRTWTargetOnly')
```

Create a Polyspace options object from the model:

```
opts = pslinkoptions(model)
```

```
opts =

                  ResultDir: 'results_$ModelName$'
        VerificationSettings: 'PrjConfig'
          OpenProjectManager: 0
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {0x1 cell}
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
            VerificationMode: 'CodeProver'
          ModelRefVerifDepth: 'Current model only'
     ModelRefByModelRefVerif: 0
     CxxVerificationSettings: 'PrjConfig'
```

Change the configuration to run a Bug Finder analysis instead of a Code Prover verification:

```
opts.VerificationMode = 'BugFinder'
```

```
opts =
```

```
                    ResultDir: 'results_$ModelName$'
          VerificationSettings: 'PrjConfig'
            OpenProjectManager: 0
          AddSuffixToResultDir: 0
       EnableAdditionalFileList: 0
            AdditionalFileList: {0x1 cell}
                InputRangeMode: 'DesignMinMax'
                ParamRangeMode: 'None'
               OutputRangeMode: 'None'
              VerificationMode: 'BugFinder'
            ModelRefVerifDepth: 'Current model only'
     ModelRefByModelRefVerif: 0
     CxxVerificationSettings: 'PrjConfig'
```

Run Polyspace software:

```
results = pslinkrun(model,opts,true)
```

The results are saved to the folder `results_mr_WhereAreTheErrors_v2`.

See Also

`pslinkoptions` | `PolySpaceViewer` | `PolyspaceAnnotation` | `bdroot`

Concepts

- "MATLAB Functions for Polyspace Batch Runs"

PolySpaceViewer

Purpose

Open analysis results in the Polyspace environment

Syntax

```
PolySpaceViewer(system)
```

Description

`PolySpaceViewer(system)` opens the Polyspace results associated with the model or subsystem `system` in the Polyspace environment. If `system` has not been analyzed, Polyspace opens to the Project Manager perspective.

Input Arguments

system - Simulink model

system | subsystem

Simulink model specified by the system or subsystem name.

Example: `PolySpaceViewer(`myModel')`

Examples

Open Results in the Polyspace environment from the Command Line

Use the preconfigured model `WhereAreTheErrors_v2` to run a Polyspace analysis and open the results in the Polyspace environment.

Load the model `WhereAreTheErrors_v2`:

```
load_system('WhereAreTheErrors_v2')
```

Open the Polyspace Viewer:

```
PolySpaceViewer('WhereAreTheErrors_v2')
```

The Polyspace environment opens to the Project Manager page because the model does not yet have Polyspace results.

Build the model to generate C code:

```
slbuild('WhereAreTheErrors_v2');
```

Create a Polyspace options object to set the configuration options:

```
config = pslinkoptions('WhereAreTheErrors_v2')

config =

                  ResultDir: 'results_$ModelName$'
        VerificationSettings: 'PrjConfig'
          OpenProjectManager: 0
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {0x1 cell}
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
            VerificationMode: 'CodeProver'
          ModelRefVerifDepth: 'Current model only'
     ModelRefByModelRefVerif: 0
     CxxVerificationSettings: 'PrjConfig'
```

Change the analysis options to also check for MISRA coding rule violations:

```
config.VerificationSettings = 'PrjConfigAndMisra';
```

Change the analysis options to run a Bug Finder analysis:

```
config.VerificationMode = 'BugFinder';

config =

                  ResultDir: 'results_$ModelName$'
        VerificationSettings: 'PrjConfigAndMisra'
          OpenProjectManager: 0
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {0x1 cell}
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
```

```
          VerificationMode: 'BugFinder'
       ModelRefVerifDepth: 'Current model only'
ModelRefByModelRefVerif: 0
CxxVerificationSettings: 'PrjConfig'
```

Run Polyspace on `WhereAreTheErrors_v2` using the configuration options object that you created:

```
pslinkrun('WhereAreTheErrors_v2', config);
```

Open the results in the Polyspace environment:

```
PolySpaceViewer('WhereAreTheErrors_v2');
```

The analysis results of `WhereAreTheErrors_v2` appear in the Polyspace Results Manager.

See Also

`pslinkoptions | pslinkrun | PolyspaceAnnotation`

Concepts

- "MATLAB Functions for Polyspace Batch Runs"