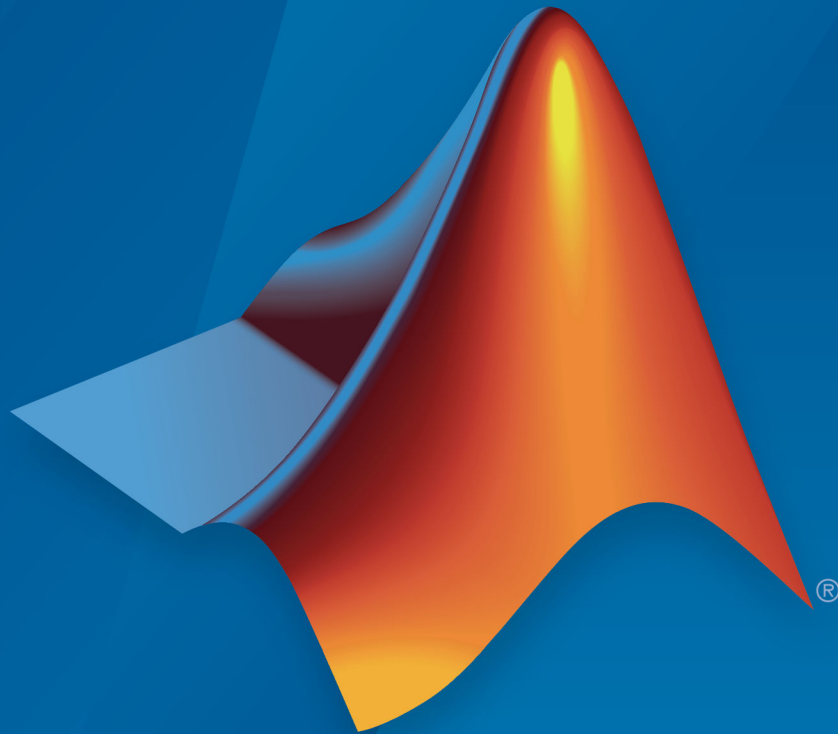


Polyspace® Code Prover™

Reference



MATLAB® & SIMULINK®

R2017b



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Polyspace® Code Prover™ Reference

© COPYRIGHT 2013–2017 by The MathWorks, Inc.

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

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

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks 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	Revised for Version 9.0 (Release 2013b)
March 2014	Online Only	Revised for Version 9.1 (Release 2014a)
October 2014	Online Only	Revised for Version 9.2 (Release 2014b)
March 2015	Online Only	Revised for Version 9.3 (Release 2015a)
September 2015	Online Only	Revised for Version 9.4 (Release 2015b)
March 2016	Online Only	Revised for Version 9.5 (Release 2016a)
September 2016	Online Only	Revised for Version 9.6 (Release 2016b)
March 2017	Online Only	Revised for Version 9.7 (Release 2017a)
September 2017	Online Only	Revised for Version 9.8 (Release 2017b)

1	Option Descriptions	
2	Polyspace Analysis Options — Command Line Only	
3	Functions — Alphabetical List	
4	Approximations Used During Verification	
	Why Polyspace Verification Uses Approximations	4-2
	Variable Ranges	4-3
	Stubbed Functions	4-4
	Function Return Value	4-5
	Function Arguments That are Pointers	4-7
	Global Variables	4-9
	Initialization of Global Variables	4-11
	Volatile Variables	4-13

Definitions and Declarations	4-15
Definition	4-15
Declaration	4-15
Implicit Data Type Conversions	4-16
Implicit Conversion When Operands Have Same Data	
Type	4-16
Implicit Conversion When Operands Have Different Data	
Types	4-17
Using memset and memcpy	4-19
Polyspace Specifications for memcpy	4-19
Polyspace Specifications for memset	4-20
#pragma Directives	4-23
Standard Library Float Routines	4-25
Unions	4-26
Variable Cast as Void Pointer	4-28
Assembly Code	4-29
Recognized Inline Assemblers	4-29
Single Function Containing Assembly Code	4-32
Multiple Functions Containing Assembly Code	4-32
Local Variables in Functions with Assembly Code	4-33
Determination of Program Stack Usage	4-35
Investigate Possible Stack Overflow	4-35
Stage Usage Not Computed	4-37
Stack Usage Assumptions	4-38
Limitations of Polyspace Verification	4-40

5 Functions, Classes, Methods, Properties, and Apps

6 MISRA C 2012

7 Code Metrics

8 Custom Coding Rules

Group 1: Files	8-2
Group 2: Preprocessing	8-3
Group 3: Type definitions	8-4
Group 4: Structures	8-5
Group 5: Classes (C++)	8-6
Group 6: Enumerations	8-7
Group 7: Functions	8-8
Group 8: Constants	8-9
Group 9: Variables	8-10
Group 10: Name spaces (C++)	8-11
Group 11: Class templates (C++)	8-12

Global Variables

9

Polyspace Report Components — Alphabetical List

10

Configuration Parameters

11

Product mode	11-2
Settings	11-2
Dependency	11-2
Command-Line Information	11-2
Settings from (C)	11-3
Settings	11-3
Dependency	11-4
Command-Line Information	11-4
Settings from (C++)	11-5
Settings	11-5
Dependency	11-5
Command-Line Information	11-6
Use custom project file	11-7
Settings	11-7
Dependency	11-7
Command-Line Information	11-7
Project configuration	11-8
Settings	11-8
Dependency	11-8

Command-Line Information	11-8
Enable additional file list	11-9
Settings	11-9
Command-Line Information	11-9
Stub lookup tables	11-10
Settings	11-10
Tips	11-11
Command-Line Information	11-11
Input	11-12
Settings	11-12
Command-Line Information	11-12
Tunable parameters	11-13
Settings	11-13
Command-Line Information	11-13
Output	11-14
Settings	11-14
Command-Line Information	11-14
Model reference verification depth	11-15
Settings	11-15
Command-Line Information	11-15
Model by model verification	11-17
Settings	11-17
Command-Line Information	11-17
Output folder	11-18
Settings	11-18
Command-Line Information	11-18
Make output folder name unique by adding a suffix	11-19
Settings	11-19
Command-Line Information	11-19
Add results to current Simulink project	11-20
Settings	11-20
Dependencies	11-20
Command-Line Information	11-20

Open results automatically after verification	11-21
Settings	11-21
Command-Line Information	11-21
Check configuration before verification	11-22
Settings	11-22
Command-Line Information	11-22
Verify all occurrences	11-23
Settings	11-23
Command-Line Information	11-23

Option Descriptions

Source code language (`-lang`)

Specify language of source files

Description

Specify the language of your source files. Before specifying other configuration options, choose this option because other options change depending on your language selection.

If you add files during project setup, the language selection can change from the default.

Files Added	Source Code Language
Only files with extension <code>.c</code>	C
Only files with extension <code>.cpp</code> or <code>.cc</code>	CPP
Files with extension <code>.c</code> , <code>.cpp</code> , and <code>.cc</code>	CPP

Set Option

User interface: In your project configuration, the option is on the **Target & Compiler** node. See “Dependencies” on page 1-3 for ways in which the source code language can be automatically determined.

Command line: Use the option `-lang`. See “Command-Line Information” on page 1-4.

Settings

Default: `c`

`c`

If your project contains only C files, choose this setting. This value restricts the verification to C language conventions. All files are interpreted as C files, regardless of their file extension.

CPP

If your project contains only C++ files, choose this setting. This value restricts the verification to C++ language conventions. All files are interpreted as C++ files, regardless of their file extension.

Dependencies

- The language option allows and disallows many options and option values. Some options change depending on your language selection. For more information, see the individual analysis option pages.
- If you create a Polyspace project or options file from your build system, the value of this option is determined by:
 - The argument to the `-lang` option. For more information, see “Create Project Automatically” or “Create Project Automatically at Command Line”.
 - If you do not specify the `-lang` option, the source code language is determined by whether your source files are compiled as C or C++ files.

-lang Argument	C or C++	Source Code Language
c		C
cpp		CPP
cpp11		CPP The option C++11 extensions (<code>-cpp11-extension</code>) is also enabled.
auto or no argument	C	C
auto or no argument	C++	CPP
auto or no argument	Both	CPP The option C++11 extensions (<code>-cpp11-extension</code>) is also enabled.

Command-Line Information

Parameter: `-lang`

Value: `c` | `cpp`

Default: `c`

Example: `polyspace-code-prover-nodesktop -lang c -sources "file1.c,file2.c"`

Example: `polyspace-code-prover-nodesktop -lang cpp -sources "file1.cpp,file2.cpp"`

Target processor type (-target)

Specify size of data types and endianness by using predefined target processor list

Description

Specify the processor on which you deploy your code.

The target processor determines the sizes of fundamental data types and the endianness of the target machine. You can analyze code intended for an unlisted processor type by using one of the other processor types, if they share common data properties.

Set Option

User interface: In your project configuration, the option is on the **Target & Compiler** node. To see the sizes of types, click the **Edit** button to the right of the **Target processor type** drop-down list.

If you select `diab`, `tasking` or `greenhills` for `Compiler (-compiler)`, in the user interface, you see only the processors allowed for that compiler. To find the data type sizes for each processor, see `Diab Compiler (-compiler diab)`. Unlike the processors for other compilers, you cannot see the data type sizes in the user interface.

Command line: Use the option `-target`. See “Command-Line Information” on page 1-8.

Why Use This Option

You specify a target processor so that some of the Polyspace run-time checks are tailored to the data type sizes and other properties of that processor.

For instance, a variable can overflow for smaller values on a 32-bit processor such as `i386` compared to a 64-bit processor such as `x86_64`. If you select `x86_64` for your Polyspace analysis, but deploy your code to the `i386` processor, your Polyspace results are not always applicable.

Once you select a target processor, you can specify if the default sign of char is signed or unsigned. To determine which signedness to specify, compile this code using the compiler settings that you typically use:

```
#include <limits.h>
int array[(char)UCHAR_MAX]; /* If char is signed, the array size is -1
```

If the code compiles, the default sign of char is unsigned. For instance, on a GCC compiler, the code compiles with the `-fsigned-char` flag and fails to compile with the `-funsigned-char` flag.

Settings

Default: i386

This table shows the size of each fundamental data type that Polyspace considers. For some targets, you can modify the default size by clicking the **Edit** button to the right of the **Target processor type** drop-down list. The optional values for those targets are shown in [brackets] in the table.

Target	char	short	int	long	long long	float	double	long double ^a	ptr	Default sign of char	endian	Alignment
i386	8	16	32	32	64	32	64	96	32	signed	Little	32
sparc	8	16	32	32	64	32	64	128	32	signed	Big	64
m68k ^b	8	16	32	32	64	32	64	96	32	signed	Big	64
powerpc	8	16	32	32	64	32	64	128	32	unsigned	Big	64
c-167	8	16	16	32	32	32	64	64	16	signed	Little	64
tms320c3x	32	32	32	32	64	32	32	64	32	signed	Little	32
sharc21x61	32	32	32	32	64	32	32 [64]	32 [64]	32	signed	Little	32
necv850	8	16	32	32	32	32	32	64	32	signed	Little	32 [16, 8]
hc08 ^c	8	16	16 [32]	32	32	32	32 [64]	32 [64]	16 ^d	unsigned	Big	32 [16]

Target	char	short	int	long	long long	float	double	long double ^a	ptr	Default sign of char	endian	Alignment
hc12	8	16	16 [32]	32	32	32	32 [64]	32 [64]	32 ⁶	signed	Big	32 [16]
mpc5xx	8	16	32	32	64	32	32 [64]	32 [64]	32	signed	Big	32 [16]
c18	8	16	16	32 [24] ^e	32	32	32	32	16 [24]	signed	Little	8
x86_64	8	16	32	64 [32] ^f	64	32	64	128	64	signed	Little	64 [32]
mcpu... (Advanced) ^g	8 [16]	8 [16]	16 [32]	32	32 [64]	32	32 [64]	32 [64]	16 [32]	signed	Little	32 [16, 8]
Targets for Diab compiler	See Diab Compiler (-compiler diab).											
Targets for TASKING compiler	See TASKING Compiler (-compiler tasking).											
Targets for Green Hills® Compiler	See Green Hills Compiler (-compiler greenhills).											

- a. For targets where the size of long double is greater than 64 bits, the size used for computations is not always the same as the size listed in this table. The exceptions are:
- For targets i386, x86_64 and m68k, 80 bits are used for computations, following the practice in common compilers.
 - For the target tms320c3x, 40 bits are used for computation, following the TMS320C3x specifications.
 - If you use a Visual compiler, the size of long double used for computations is the same as size of double, following the specification of Visual C++® compilers.
- b. The M68k family (68000, 68020, and so on) includes the “ColdFire” processor
- c. Non-ANSI C specified keywords and compiler implementation-dependent pragmas and interrupt facilities are not taken into account by this support
- d. All kinds of pointers (near or far pointer) have 2 bytes (hc08) or 4 bytes (hc12) of width physically.
- e. The c18 target supports the type short long as 24 bits in size.
- f. Use option -long-is-32bits to support Microsoft® C/C++ Win64 target.

- g. `mcpu` is a reconfigurable Micro Controller/Processor Unit target. You can use this type to configure one or more generic targets. For more information, see `Generic target options`.

Tips

If your processor is not listed, use a similar processor that shares the same characteristics, or create an `mcpu` generic target processor. If your target processor does not match the characteristics of a predefined processor, contact MathWorks® technical support.

Command-Line Information

Parameter: `-target`

Value: `i386` | `sparc` | `m68k` | `powerpc` | `c-167` | `x86_64` | `tms320c3x` | `sharc21x61` | `necv850` | `hc08` | `hc12` | `mpc5xx` | `c18` | `mcpu`

Default: `i386`

Example: `polyspace-code-prover-nodesktop -target m68k`

You can override the default values for some targets by using specific command-line options. See the section **Command-Line Options** in `Generic target options`.

See Also

Polyspace Results

`Lower Estimate of Local Variable Size` | `Higher Estimate of Local Variable Size`

Topics

“Specify Analysis Options”

“Specify Target Environment and Compiler Behavior”

Generic target options

Specify size of data types and endianness by creating your own target processor

Description

The **Generic target options** dialog box opens when you set the **Target processor type** to `mcpu`.

Allows the specification of a generic "Micro Controller/Processor Unit" target. Use the dialog box to specify the name of a new `mcpu` target, for example *MyTarget*. That new target is added to the **Target processor type** option list.

Changing the genetic target has consequences for:

- Detection of overflow
- Computation of `sizeof` objects

The **Target processor type** option is available on the **Target & Compiler** node in the **Configuration** pane.

Settings

Default characteristics of a new target: listed as *type* [size]

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

- *pointer* [16]
- *char* is signed
- *endianness* is little-endian

Dependency

A custom target can only be created when `Target processor type (-target)` is set to `mcpu`.

A custom target is not available when `Compiler (-compiler)` is set to one of the `visual*` options.

Command-Line Options

When using the command line, specify your target with the other target specification options.

Option	Description	Available With	Example
<code>-little-endian</code>	<p>Little-endian architectures are Less Significant byte First (LSF). For example: i386.</p> <p>Specifies that the less significant byte of a short integer (e.g. 0x00FF) is stored at the first byte (0xFF) and the most significant byte (0x00) at the second byte.</p>	<code>mcpu</code>	<code>polyspace-code-prover-nodesktop -lang c -target mcpu -little-endian</code>

Option	Description	Available With	Example
<code>-big-endian</code>	<p>Big-endian architectures are Most Significant byte First (MSF). For example: SPARC, m68k.</p> <p>Specifies that the most significant byte of a short integer (e.g. 0x00FF) is stored at the first byte (0x00) and the less significant byte (0xFF) at the second byte.</p>	mcpu	<code>polyspace-code-prover-nodesktop -target mcpu -big-endian</code>
<code>-default-sign-of-char [signed unsigned]</code>	<p>Specify default sign of char.</p> <p>signed: Specifies that char is signed, overriding target's default.</p> <p>unsigned: Specifies that char is unsigned, overriding target's default.</p>	All targets	<code>polyspace-code-prover-nodesktop -default-sign-of-char unsigned -target mcpu</code>
<code>-char-is-16bits</code>	<p>char defined as 16 bits and all objects have a minimum alignment of 16 bits</p> <p>Incompatible with <code>-short-is-8bits</code> and <code>-align 8</code></p>	mcpu	<code>polyspace-code-prover-nodesktop -target mcpu -char-is-16bits</code>

Option	Description	Available With	Example
<code>-short-is-8bits</code>	Define short as 8 bits, regardless of sign	mcpu	<code>polyspace-code-prover-nodesktop -target mcpu -short-is-8bits</code>
<code>-int-is-32bits</code>	Define int as 32 bits, regardless of sign. Alignment is also set to 32 bits.	mcpu, hc08, hc12, mpc5xx	<code>polyspace-code-prover-nodesktop -target mcpu -int-is-32bits</code>
<code>-long-is-32bits</code>	Define long as 32 bits, regardless of sign. Alignment is also set to 32 bits. If your project sets int to 64 bits, you cannot use this option.	All targets	<code>polyspace-code-prover-nodesktop -target mcpu -long-is-32bits</code>
<code>-long-long-is-64bits</code>	Define long long as 64 bits, regardless of sign. Alignment is also set to 64 bits.	mcpu	<code>polyspace-code-prover-nodesktop -target mcpu -long-long-is-64bits</code>
<code>-double-is-64bits</code>	Define double and long double as 64 bits, regardless of sign.	mcpu, sharc21x61, hc08, hc12, mpc5xx	<code>polyspace-code-prover-nodesktop -target mcpu -double-is-64bits</code>
<code>-pointer-is-24bits</code>	Define pointer as 24 bits, regardless of sign.	c18	<code>polyspace-code-prover-nodesktop -target c18 -pointer-is-24bits</code>
<code>-pointer-is-32bits</code>	Define pointer as 32 bits, regardless of sign.	mcpu	<code>polyspace-code-prover-nodesktop -target mcpu -pointer-is-32bits</code>

Option	Description	Available With	Example
<code>-align [32 16 8]</code>	<p>Specifies the largest alignment of struct or array objects to the 32, 16 or 8 bit boundaries.</p> <p>Consequently, the array or struct storage is strictly determined by the size of the individual data objects without member and end padding.</p>	<p>mcpcu,</p> <p>Only 16 or 32 bits for: hc08, hc12, mpc5xx</p>	<pre>polyspace-code-prover- nodesktop -target mcpcu - align 16</pre>

Common Generic Targets

The following tables describe the characteristics of common generic targets.

ST7 (Hiware C compiler : HiCross for ST7)

ST7	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16	32	32	32	32	32	16/32	unsigned	Big
alignment	8	16/8	16/8	32/16/8	32/16/8	32/16/8	32/16/8	32/16/8	32/16/8	N/A	N/A

ST9 (GNU C compiler : gcc9 for ST9)

ST9	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16	32	32	32	64	64	16/64	unsigned	Big
alignment	8	8	8	8	8	8	8	8	8	N/A	N/A

Hitachi H8/300, H8/300L

Hitachi H8/300, H8/300L	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16/32	32	64	32	654	64	16	unsigned	Big
alignment	8	16	16	16	16	16	16	16	16	N/A	N/A

Hitachi H8/300H, H8S, H8C, H8/Tiny

Hitachi H8/300H, H8S, H8C, H8/Tiny	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16/32	32	64	32	64	64	32	unsigned	Big
alignment	8	16	32/16	32/16	32/16	32/16	32/16	32/16	32/16	N/A	N/A

See Also

Target processor type (-target)

Topics

“Specify Target Environment and Compiler Behavior”

Compiler (`-compiler`)

Specify the compiler that you use to build your source code

Description

Specify the compiler that you use to build your source code.

Polyspace fully supports the most common compilers used to develop embedded applications. See the list below. For these compilers, you can run analysis simply by specifying your compiler and target processor. For other compilers, specify `generic` as compiler name. If you face compilation errors, explicitly define compiler-specific extensions to work around the errors.

Set Option

User interface: In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-compiler`. See “Command-Line Information” on page 1-19.

Why Use This Option

Polyspace uses this information to interpret syntax that is not part of the C/C++ Standard, but comes from language extensions. See “Language Extensions Supported by Default”.

For example, the option allows additional language keywords, such as `sfr`, `sbit`, and `bit`. If you do not specify your compiler, these additional keywords can cause compilation errors during Polyspace analysis.

Settings

Default: `generic`

generic

Analysis allows only standard syntax.

For C code, syntax must follow the ANSI® C standard.

For C++ code, syntax must follow ISO®/IEC 14882:2003 C++ (C++ 2003). If you want to allow C++ 11 syntax (ISO/IEC 14882:2011 C++), also select **C++ 11 extensions**.

gnu3.4

Analysis allows GCC 3.4 syntax.

gnu4.6

Analysis allows GCC 4.6 syntax.

gnu4.7

Analysis allows GCC 4.7 syntax.

For more information, see “Limitations” on page 1-18.

gnu4.8

Analysis allows GCC 4.8 syntax.

For more information, see “Limitations” on page 1-18.

gnu4.9

Analysis allows GCC 4.9 syntax.

For more information, see “Limitations” on page 1-18.

clang3.5

Analysis allows Clang 3.5 syntax.

The Clang `__attribute__((vector_size()))` is not supported.

visual9.0

Analysis allows Microsoft Visual C++ 2008 syntax.

visual10.0

Analysis allows Microsoft Visual C++ 2010 syntax.

This option implicitly enables the option `-no-stl-stubs`.

visual11.0

Analysis allows Microsoft Visual C++ 2012 syntax.

This option implicitly enables the option `-no-stl-stubs`.

`visual12.0`

Analysis allows Microsoft Visual C++ 2013 syntax.

This option implicitly enables the option `-no-stl-stubs`.

`visual14.0`

Analysis allows Microsoft Visual C++ 2015 syntax (supports Microsoft Visual Studio@update 2).

This option implicitly enables the option `-no-stl-stubs`.

`keil`

Analysis allows non-ANSI C syntax and semantics associated with the Keilo products from ARM (www.keil.com).

`iar`

Analysis allows non-ANSI C syntax and semantics associated with the compilers from IAR Systems (www.iar.com).

`diab`

Analysis allows non-ANSI C syntax and semantics associated with the Wind River® Diab compiler.

If you select `diab`, the option `Target processor type (-target)` shows only the targets that are allowed for the Diab compiler. See `Diab Compiler (-compiler diab)`.

`tasking`

Analysis allows non-ANSI C syntax and semantics associated with the TASKING compiler.

If you select `tasking`, the option `Target processor type (-target)` shows only the targets that are allowed for the TASKING compiler. See `TASKING Compiler (-compiler tasking)`.

`greenhills`

Analysis allows non-ANSI C syntax and semantics associated with a Green Hills compiler.

If you select `greenhills`, the option `Target processor type (-target)` shows only the targets that are allowed for a Green Hills compiler. See `Green Hills Compiler (-compiler greenhills)`.

Tips

- If you use a Visual Studio compiler, you must use a `Target processor type (-target)` option that sets `long long` to 64 bits. Compatible targets include: `i386`, `sparc`, `m68k`, `powerpc`, `tms320c3x`, `sharc21x61`, `mpc5xx`, `x86_64`, or `mcpu` with `long long` set to 64 (`-long-long-is-64bits` at the command line).
- If you enable `Check JSF C++ rules (-jsf-coding-rules)`, select the compiler `generic`. If you use another compiler, Polyspace cannot check the JSF® coding rules that require conforming to the ISO standard. For example, AV Rule 8: “All code shall conform to ISO/IEC 14882:2002(E) standard C++.”

Limitations

Polyspace does not support certain features of these compilers:

- GNU® compilers (version 4.7 or later):
 - Nested functions.

For instance, the function `bar` is nested in function `foo`:

```
void foo (int a, int b)
{
    void bar (int c) { return c * c; }

    return bar (a) + bar (b);
}
```

- Forward declaration of function parameters.

For instance, the parameter `len` is forward declared:

```
void func (int len; char data[len][len], int len)
{
    /* ... */
}
```

- Complex integer data types.

However, complex floating point data types are supported.

- Structures with flexible array members.

For instance, the structure `S` has a flexible array member `tab`.

```
struct S {
    int x;
    int tab[];           /* flexible array member - not supported */
};
```

- Visual Studio compilers:

- C++ Accelerated Massive Parallelism (AMP).

C++ AMP is a Visual Studio feature that accelerates your C++ code execution for certain types of data-parallel hardware on specific targets. You typically use the `restrict` keyword to enable this feature.

```
void Buffer() restrict(amp)
{
    ...
}
```

- `__assume` statements.

You typically use `__assume` with a condition that is false. The statement indicates that the optimizer must assume the condition to be henceforth true. Code Prover cannot reconcile this contradiction. You get the error:

```
Asked for compulsory presence of absent entity : assert
```

- Managed Extensions for C++ (required for the .NET Framework)
- `__declspec` keyword with attributes other than `noreturn`, `nothrow`, `selectany` or `thread`.

Command-Line Information

Parameter: `-compiler`

Value: `generic` | `gnu3.4` | `gnu4.6` | `gnu4.7` | `gnu4.8` | `gnu4.9` | `clang3.5` | `visual9.0` | `visual10.0` | `visual11.0` | `visual12.0` | `visual14.0` | `keil` | `iar` | `diab` | `tasking`

Default: generic

Example: polyspace-code-prover-nodesktop -lang c -sources
"file1.c,file2.c" -lang c -OS-target Linux -compiler gnu4.6

Example: polyspace-code-prover-nodesktop -lang cpp -sources
"file1.cpp,file2.cpp" -OS-target Visual -compiler visual7.1

See Also

Target processor type (-target) | C++11 extensions (-cpp11-extension)
| Block char16/32_t types (-no-uliterals)

Topics

“Supported Keil or IAR Language Extensions”

“Supported C++ 2011 Language Extensions”

“Troubleshooting in Polyspace Code Prover”

Diab Compiler (-compiler diab)

Specify the Wind River Diab compiler

Description

Specify `diab` for `Compiler` (-compiler) if you compile your code using the Wind River Diab compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `diab` for **Compiler**, in the user interface, you see only the processors allowed for the Diab compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `diab` compiler, you must specify the path to your compiler header files.

- In the user interface, add the folder containing your compiler headers to the project.

For more information, see “Add Source and Include Folders”.

- At the command line, use the flag `-I` with the `polyspace-code-prover-nodesktop` command.

For more information, see `-I`.

The software supports version 5.9.6 and older versions of the Diab compiler.

Settings

The targets use the following default sizes in bits for the fundamental types. Unlike targets for other compilers, you do not see these sizes in the user interface.

Target	char	short	int	long	long long	float	double	long double	ptr	Default sign of char	Endianness	Alignment
i386	8	16	32	32	64	32	64	96	32	signed	Little	32

Target	char	short	int	long	long long	float	double	long double	ptr	Default sign of char	Endianness	Alignment
powerpc	8	16	32	32	64	32	64	64	32	unsigned	Big	64
powerpc64	8	16	32	64	64	32	64	64	64	unsigned	Big	64
arm	8	16	32	32	64	32	64	64	32	unsigned	Big	64
coldfire	8	16	32	32	64	32	64	64	32	signed	Big	64
mips	8	16	32	32	64	32	64	64	32	signed	Big	64
mcore	8	16	32	32	64	32	64	64	32	unsigned	Big	64
rh850	8	16	32	32	64	32	64	64	32	signed	Little	32
superh	8	16	32	32	64	32	64	64	32	signed	Big	64
tricore	8	16	32	32	64	32	64	64	32	signed	Little	64
68k, sparc	Not supported.											

In addition, `wchar_t` is interpreted as `unsigned short` and `size_t` is interpreted as `unsigned int`.

If you use Diab compiler flags to change any of these default specifications and want to emulate these flags, contact Technical Support.

Tips

If you encounter errors during Polyspace analysis, see “Errors Related to Diab Compiler”.

Command-Line Information

Parameter: `-compiler diab -target`

Value: `i386 | powerpc | arm | coldfire | mips | mcore | rh850 | superh | tricore`

Default: `powerpc`

Example: polyspace-code-prover-nodesktop -compiler diab -target tricore

See Also

Compiler (-compiler) | Target processor type (-target)

Topics

“Specify Analysis Options”

Introduced in R2016b

TASKING Compiler (`-compiler tasking`)

Specify the Altium TASKING compiler

Description

Specify `tasking` for `Compiler` (`-compiler`) if you compile your code using the Altium® TASKING compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `tasking` for **Compiler**, in the user interface, you see only the processors allowed for the TASKING compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `tasking` compiler, you must specify the path to your compiler header files.

- In the user interface, add the folder containing your compiler headers to the project.

For more information, see “Add Source and Include Folders”.

- At the command line, use the flag `-I` with the `polyspace-code-prover-nodesktop` command.

For more information, see `-I`.

The software supports different versions of the TASKING compiler, depending on the target:

- TriCore: 6.0 and older versions
- C166: 4.0 and older versions
- ARM: 5.2 and older versions
- RH850: 2.2 and older versions

Settings

The targets use the following default sizes in bits for the fundamental types. Unlike targets for other compilers, you do not see these sizes in the user interface.

Target	char	short	int	long	long long	float	double	long double	ptr	Default sign of char	Endianness	Alignment
tricore	8	16	32	32	64	32	64	64	32	signed	Little	32
c166	8	16	16	32	64	32	64	64	32	signed	Little	16
rh850	8	16	32	32	64	32	64	64	32	signed	Little	64
arm	8	16	32	32	64	32	64	64	32	signed	Big	64

In addition, `wchar_t` is interpreted as `unsigned short` and `size_t` is interpreted as `unsigned int`.

If you use TASKING compiler flags to change any of these default specifications and want to emulate these flags, contact Technical Support.

Tips

- Polyspace does not support some constructs specific to the TASKING compiler.

For the list of unsupported constructs, see `codeprover_limitations.pdf` in `matlabroot\polyspace\verifier\code_prover`. Here, `matlabroot` is the MATLAB® installation folder, for instance, `C:\Program Files\MATLAB\R2017b`.

- The CPU used is TC1793. If you use a different CPU, set the following analysis options in your project:
 - Disabled preprocessor definitions (-U): Undefine the macro `__CPU_TC1793B__`.
 - Preprocessor definitions (-D): Define the macro `__CPU__`. Enter `__CPU__=xxx`, where `xxx` is the name of your CPU.

Additionally, define the equivalent of the macro `__CPU_TC1793B__` for your CPU. For instance, enter `__CPU_TC1793A__`.

Instead of manually specifying your compiler, if you trace your build command (makefile), Polyspace can detect your CPU and add the required definitions in your project. For more information, see:

- “Create Project Automatically”
- “Create Project Automatically at Command Line”
- For some errors related to TASKING compiler-specific constructs, see solutions in “Errors Related to TASKING Compiler”.

Command-Line Information

Parameter: `-compiler tasking -target`

Value: `tricore | c166 | rh850 | arm`

Default: `tricore`

Example: `polyspace-code-prover-nodesktop -compiler tasking -target tricore`

See Also

`Compiler (-compiler) | Target processor type (-target)`

Topics

“Specify Analysis Options”

Introduced in R2017a

Green Hills Compiler (`-compiler greenhills`)

Specify Green Hills compiler

Description

Specify `greenhills` for `Compiler` (`-compiler`) if you compile your code using a Green Hills compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `greenhills` for **Compiler**, in the user interface, you see only the processors allowed for a Green Hills compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `greenhills` compiler, you must specify the path to your compiler header files.

- In the user interface, add the folder containing your compiler headers to the project.

For more information, see “Add Source and Include Folders”.

- At the command line, use the flag `-I` with the `polyspace-code-prover-nodesktop` command.

For more information, see `-I`.

Settings

The targets use the following default sizes in bits for the fundamental types. Unlike targets for other compilers, you do not see these sizes in the user interface.

Target	char	short	int	long	long long	float	double	long double	ptr	Default sign of char	Endianness	Alignment	Default
powerpc	8	16	32	32	64	32	64	64	32	unsigned	Big	64	using general double in next

Target	char	short	int	long	long long	float	double	long double	ptr	Default sign of char	Endianness	Alignment	Default
powerpc64	8	16	32	64	64	32	64	64	64	unsigned	Big	64	us ni \$g in \$e nd ei dn lt o n \$e

Target	char	short	int	long	long long	float	double	long double	ptr	Default sign of char	Endianness	Alignment	Default
arm	8	16	32	32	64	32	64	64	32	unsigned	Little	32	u m s i g n e d i s n h t o r t

Target	char	short	int	long	long long	float	double	long double	ptr	Default sign of char	Endiannes	Alignment	Default
arm64	8	16	32	64	64	32	64	64	64	unsigned	Little	64	us ni \$g in \$e nd ei dn lt o n \$e

Target	char	short	int	long	long long	float	double	long double	ptr	Default sign of char	Endiannes	Alignment	D
tricore	8	16	32	32	64	32	64	64	32	signed	Little	32	ee ff ii nn tt oo nn oo ff sw ic zh ea r t

Target	char	short	int	long	long long	float	double	long double	ptr	Default sign of char	Endiannes	Alignment	D e f i n i t i o n o f f i c z h e a r t
rh850	8	16	32	32	64	32	64	64	32	signed	Little	64	us ni \$g in ge nd el do in ng t

1 Option Descriptions

Target	char	short	int	long	long long	float	double	long double	ptr	Default sign of char	Endiannes	Alignment	Default
i386	8	16	32	32	64	32	64	96	32	signed	Little	32	using general defaulting

Target	char	short	int	long	long long	float	double	long double	ptr	Default sign of char	Endianness	Alignment	Default
x86_64	8	16	32	64	64	32	64	128	64	signed	Little	128	Default

If you use the Green Hills compiler flags to change any of these default specifications and want to emulate these flags, contact Technical Support.

Tips

- Polyspace supports the embedded configuration for the i386 target. If your x86 Green Hills compiler is configured for native Windows® development, you can see compilation errors or incorrect analysis results with Code Prover. Contact Technical Support.

For instance, Green Hills compilers consider a size of 12 bytes for `long double` for embedded targets, but 8 bytes for native Windows. Polyspace considers 12 bytes by default.

- If you create a Polyspace project from a build command that uses a Green Hills compiler, the compiler options `-filetype` and `-os_dir` are not implemented in the project. To emulate the `-os_dir` option, you can explicitly add the path argument of the option as an include folder to your Polyspace project.

Command-Line Information

Parameter: `-compiler greenhills -target`

Value: `powerpc | powerpc64 | arm | arm64 | tricore | rh850 | arm | i386 | x86_64`

Default: `powerpc`

Example: `polyspace-code-prover-nodesktop -compiler greenhills -target arm`

See Also

`Compiler (-compiler) | Target processor type (-target)`

Topics

“Specify Analysis Options”

Introduced in R2017b

Respect C90 standard (-no-language-extensions)

Restrict analysis to C language specified in ANSI C standard

Description

Restrict the analysis to the C language specified in the ANSI C standard (ISO/IEC 9899:1990). Language extensions added in later standards such as C99 generate compilation errors.

Set Option

User interface: In your project configuration, the option is on the **Target & Compiler** node. See “Dependencies” on page 1-38 for other options you must also enable.

Command line: Use the option `-no-language-extensions`. See “Command-Line Information” on page 1-38.

Why Use This Option

Use this option if you compile your code by using the C90 standard.

For instance, if you compile with the GCC option `-ansi` or `-std=c90`, use this option.

Settings

On

Restrict the analysis to the C90 standard. Code must conform to the ANSI C standard (ISO/IEC 9899:1990).

Off (default)

Allow language extensions from the C99 standard (ISO/IEC 9899:1999).

Dependencies

This option is available only when `Source code language (-lang)` is set to C.

If you enable this option, you cannot use `Compiler (-compiler) settings keil and iar`.

Command-Line Information

Parameter: `-no-language-extensions`

Default: `off`

Example: `polyspace-code-prover-nodesktop -lang c -no-language-extensions`

Introduced in R2015b

Sfr type support (`-sfr-types`)

Specify sizes of `sfr` types for code developed with Keil or IAR compilers

Description

Specify sizes of `sfr` types (types that define special function registers).

Set Option

User interface: In your project configuration, the option is on the **Target & Compiler** node. See “Dependency” on page 1-39 for other options you must also enable.

Command line: Use the option `-sfr-types`. See “Command-Line Information” on page 1-40.

Why Use This Option

Use this option if you have statements such as `sfr addr = 0x80;` in your code. `sfr` types are not standard C types. Therefore, you must specify their sizes explicitly for the Polyspace analysis.

Settings

No Default

List each `sfr` name and its size in bits.

Dependency

This option is available only when `Compiler` (`-compiler`) is set to `keil` or `iar`.

Command-Line Information

Syntax: `-sfr-types sfr_name=size_in_bits,...`

No Default

Name Value: an sfr name such as `sfr16`.

Size Value: `8 | 16 | 32`

Example: `polyspace-code-prover-nodesktop -lang c -compiler iar -sfr-types sfr=8,sfr16=16 ...`

See Also

Topics

“Specify Target Environment and Compiler Behavior”

“Supported Keil or IAR Language Extensions”

Division round down (`-div-round-down`)

Round down quotients from division or modulus of negative numbers instead of rounding up

Description

Specify whether quotients from division and modulus of negative numbers are rounded up or down.

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

Set Option

User interface: In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-div-round-down`. See “Command-Line Information” on page 1-42.

Why Use This Option

Use this option to emulate your compiler.

The option is relevant only for compilers following C90 standard (ISO/IEC 9899:1990). The 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"*. The standard allows compilers to choose their own implementation.

For compilers following the C99 standard ((ISO/IEC 9899:1999), this option is not required. The standard enforces division with rounding towards zero (section 6.5.5).

Settings

On

If either operand `/` or `%` is negative, the result of the `/` operator is the largest integer less than or equal to 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.

Off (default)

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

This behavior is also known as rounding towards zero.

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

Command-Line Information

Parameter: `-div-round-down`

Default: Off

Example: `polyspace-code-prover-nodesktop -div-round-down`

Enum type definition (-enum-type-definition)

Specify how to represent an enum with a base type

Description

Allow 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 its enumeration values.

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

Set Option

User interface: In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-enum-type-definition`. See “Command-Line Information” on page 1-45.

Why Use This Option

Your compiler represents enum variables as constants of a base integer type. Use this option so that you can emulate your compiler.

To check your compiler settings, compile this code using the compiler settings that you typically use:

```
#include <assert.h>
#include <limits.h>

enum { MAXSIGNEDBYTE=127 } mysmallenum_t;
int dummy[(int)sizeof(mysmallenum_t) - (int)sizeof(int)]; /* Breakpoint 1 */

enum { MYMAXINT = INT_MAX } myintenum_t;
int main(void) {
```

```
    assert((MYMAXINT + 1) < 0); /* Breakpoint 2 */
    assert((MYMAXINT + 1) >= 0); /* Breakpoint 3 */
    assert(0); /* Breakpoint 4 */

    return 0;
}
```

If compilation does not fail even at breakpoint 4, your `assert` statements do not behave as expected. Check your compiler documentation and change your compiler settings. If compilation fails at:

- Breakpoint 1: Use `defined-by-compiler` for this option.
- Breakpoint 2: Use `auto-signed-first` for this option.
- Breakpoint 3: Use `auto-unsigned-first` for this option.

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

Uses the signed integer type for all compilers except `gnu`.

For the `gnu` compilers, it uses the first type that can hold all of the enumerator values from this list: `signed int`, `unsigned int`, `signed long`, `unsigned long`, `signed long long`, and `unsigned long long`.

`auto-signed-first`

Uses the first type that can hold all of the enumerator values from this list: `signed char`, `unsigned char`, `signed short`, `unsigned short`, `signed int`, `unsigned int`, `signed long`, `unsigned long`, `signed long long`, and `unsigned long long`.

`auto-unsigned-first`

Uses the first type that can hold all of the enumerator values from these lists:

- If enumerator values are positive: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`.
- If one or more enumerator values are negative: `signed char`, `signed short`, `signed int`, `signed long`, and `signed long long`.

Command-Line Information

Parameter: -enum-type-definition

Value: defined-by-compiler | auto-signed-first | auto-unsigned-first

Default: defined-by-compiler

Example: polyspace-code-prover-nodesktop -enum-type-definition auto-signed-first

C++11 extensions (`-cpp11-extension`)

Allow C++11 language extensions

Description

Allow C++11 language extensions.

Set Option

User interface: In your project configuration, the option is on the **Target & Compiler** node. See “Dependencies” on page 1-47 for other options you must also enable.

Command line: Use the option `-cpp11-extension`. See “Command-Line Information” on page 1-47.

Why Use This Option

If your compiler allows C++11 language extensions, enable this option.

To check if your compiler allows the extensions, compile this code using the compiler settings that you typically use:

```
#if defined(__cplusplus) && __cplusplus >= 201103L
    /* C++11 compiler */
#else
    void* ptr = nullptr;
#endif
```

If the code compiles, enable this option.

For instance, on a GCC compiler, the code compiles with the `-std=c++11` flag but fails to compile without the flag. If you typically use the flag, enable this option.

Settings

On

The analysis allows C++11 syntax.

Off (default)

The analysis does not allow C++11 syntax.

Dependencies

This analysis option is available only when both these conditions are true:

- Source code language (-lang) is CPP.
- Compiler (-compiler) is generic, gnu4.6, or gnu4.7.

Command-Line Information

Parameter: -cpp11-extension

Default: off

Example: polyspace-code-prover-nodesktop -lang cpp -cpp11-extension

See Also

Compiler (-compiler) | Block char16/32_t types (-no-uliterals)

Topics

“Supported C++ 2011 Language Extensions”

Block char16/32_t types (`-no-uliterals`)

Disable Polyspace definitions for `char16_t` or `char32_t`

Description

Specify that the analysis must not define `char16_t` or `char32_t` types.

Set Option

User interface: In your project configuration, the option is on the **Target & Compiler** node. See “Dependencies” on page 1-49 for other options you must also enable.

Command line: Use the option `-no-uliterals`. See “Command-Line Information” on page 1-49.

Why Use This Option

If your compiler defines `char16_t` and/or `char32_t` through a `typedef` statement or by using includes, use this option to turn off the standard Polyspace definition of `char16_t` and `char32_t`.

To check if your compiler defines these types, compile this code using the compiler settings that you typically use:

```
typedef unsigned short char16_t;  
typedef unsigned long char32_t;
```

If the file compiles, it means that your compiler has already defined `char16_t` and `char32_t`. Enable this Polyspace option.

Settings

On

The analysis does not allow `char16_t` and `char32_t` types.

Off (default)

The analysis allows `char16_t` and `char32_t` types.

Dependencies

You can select this option only when these conditions are true:

- Source code language (`-lang`) is C++.
- Compiler (`-compiler`) is either generic or a GNU version.

Command-Line Information

Parameter: `-no-uliterals`

Default: off

Example: `polyspace-code-prover-nodesktop -compiler gnu4.7 -lang cpp -cpp11-extension -no-uliterals`

See Also

Compiler (`-compiler`) | C++11 extensions (`-cpp11-extension`)

Topics

“Supported C++ 2011 Language Extensions”

Pack alignment value (`-pack-alignment-value`)

Specify default structure packing alignment for code developed in Visual C++

Description

Specify the default packing alignment (in bytes) for structures, unions, and class members.

Set Option

User interface: In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-pack-alignment-value`. See “Command-Line Information” on page 1-51.

Why Use This Option

If you use compiler options to specify how members of a structure are packed into memory, use this option to emulate your compiler.

For instance, if you use the Visual Studio option `/Zp` to specify an alignment, use this option for your Polyspace analysis.

If you use `#pragma pack` directives in your code to specify alignment, and also specify this option for analysis, the `#pragma pack` directives take precedence. See “`#pragma Directives`” on page 4-23.

Settings

Default: 8

You can enter one of these values:

- 1
- 2
- 4
- 8
- 16

Command-Line Information

Parameter: -pack-alignment-value

Value: 1 | 2 | 4 | 8 | 16

Default: 8

Example: polyspace-code-prover-nodesktop -compiler visual10 -pack-alignment-value 4

Ignore pragma pack directives (`-ignore-pragma-pack`)

Ignore `#pragma pack` directives

Description

Specify that the analysis must ignore `#pragma pack` directives in the code.

Set Option

User interface: In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-ignore-pragma-pack`. See “Command-Line Information” on page 1-53.

Why Use This Option

Use this option if `#pragma pack` directives in your code cause linking errors.

For instance, you have two structures with the same name in your code, but one declaration follows a `#pragma pack(2)` statement. Because the default alignment is 8 bytes, the different packing for the two structures causes a linking error. Use this option to avoid such errors. See also “#pragma Directives” on page 4-23.

Settings

On

The analysis ignores the `#pragma` directives.

Off (default)

The analysis takes into account specifications in the `#pragma` directives.

Command-Line Information

Parameter: -ignore-pragma-pack

Default: Off

Example: polyspace-code-prover-nodesktop -ignore-pragma-pack

See Also

Topics

“#pragma Directives” on page 4-23

Management of `size_t` (`-size-t-type-is`)

Specify the underlying data type of `size_t`

Description

Specify the underlying data type of `size_t` explicitly: `unsigned int`, `unsigned long` or `unsigned long long`. If you do not specify this option, your choice of compiler determines the underlying type.

Set Option

User interface: In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-size-t-type-is`. See “Command-Line Information” on page 1-55.

Why Use This Option

The analysis associates a data type with `size_t` when you specify your compiler. If you use a compiler option that changes this default type, emulate your compiler option by using this analysis option.

If you run into compilation errors during Polyspace analysis and trace the error to the definition of `size_t`, it is possible that you use a compiler option and change your compiler default. To probe further, compile this code with your compiler using the options that you typically use:

```
/* Header defines malloc as void* malloc (size_t size)
#include <stdio.h>

void* malloc (unsigned int size);
```

If the file does not compile, your compiler options cause `size_t` to be defined as `unsigned long` or `unsigned long long`. Replace `unsigned int` with `unsigned long` and try again.

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

Your specification for `Compiler` (`-compiler`) determines the underlying type of `size_t`.

`unsigned-int`

The analysis considers `unsigned int` as the underlying type of `size_t`.

`unsigned-long`

The analysis considers `unsigned long` as the underlying type of `size_t`.

`unsigned-long-long`

The analysis considers `unsigned long long` as the underlying type of `size_t`.

Command-Line Information

Parameter: `-size-t-type-is`

Value: `defined-by-compiler` | `unsigned-int` | `unsigned-long` | `unsigned-long-long`

Default: `defined-by-compiler`

Example: `polyspace-code-prover-nodesktop -size-t-type-is unsigned-long`

Management of `wchar_t` (`-wchar-t-type-is`)

Specify the underlying data type of `wchar_t`

Description

Specify the underlying data type of `wchar_t` explicitly. If you do not specify this option, your choice of compiler determines the underlying type.

Set Option

User interface: In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-wchar-t-type-is`. See “Command-Line Information” on page 1-57.

Why Use This Option

The analysis associates a data type with `wchar_t` when you specify your compiler. If you use a compiler option that changes this default type, emulate your compiler option by using this analysis option.

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

Your specification for `Compiler` (`-compiler`) determines the underlying type of `wchar_t`.

`signed-short`

The analysis considers `signed short` as the underlying type of `wchar_t`.

`unsigned-short`

The analysis considers `unsigned short` as the underlying type of `wchar_t`.

`signed-int`

The analysis considers `signed int` as the underlying type of `wchar_t`.

`unsigned-int`

The analysis considers `unsigned int` as the underlying type of `wchar_t`.

`signed-long`

The analysis considers `signed long` as the underlying type of `wchar_t`.

`unsigned-long`

The analysis considers `unsigned long` as the underlying type of `wchar_t`.

Command-Line Information

Parameter: `-wchar-t-type-is`

Value: `defined-by-compiler` | `signed-short` | `unsigned-short` | `signed-int` | `unsigned-int` | `signed-long` | `unsigned-long`

Default: `defined-by-compiler`

Example: `polyspace-code-prover-nodesktop -wchar-t-type-is signed-int`

Signed right shift (`-logical-signed-right-shift`)

Specify how to treat the sign bit for logical right shifts on signed variables

Description

Choose between arithmetic and logical shift for right shift operations on negative values.

This option does not modify compile-time expressions. For more details, see “Limitation” on page 1-59.

Set Option

User interface: In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-logical-signed-right-shift`. See “Command-Line Information” on page 1-59.

Why Use This Option

The C99 Standard (sec 6.5.7) states that for a right-shift operation $x1 \gg x2$, if $x1$ is signed and has negative values, the behavior is implementation-defined. Different compilers choose between arithmetic and logical shift. Use this option to emulate your compiler.

Settings

Default: `Arithmetical`

`Arithmetical`

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
```

Limitation

In compile-time expressions, this Polyspace option does not change the standard behavior for right shifts.

For example, consider this right shift expression:

```
int arr[ (-4) >> 20 ];
```

The compiler computes array sizes, so the expression `(-4) >> 20` is evaluated at compilation time. Logically, this expression is equivalent to 4095. However, arithmetically, the result is -1. This statement causes a compilation error (arrays cannot have negative size) because the standard right-shift behavior for signed integers is arithmetic.

Command-Line Information

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

Parameter: `-logical-signed-right-shift`

Default: Arithmetic signed right shifts

Example: `polyspace-code-prover-nodesktop -logical-signed-right-shift`

Preprocessor definitions (-D)

Replace macros in preprocessed code

Description

Replace macros with their definitions in preprocessed code.

Set Option

User interface: In your project configuration, the option is on the **Macros** node.

Command line: Use the option `-D`. See “Command-Line Information” on page 1-62.

Why Use This Option

Use this option to emulate your compiler behavior. For instance, if your compiler considers a macro `_WIN32` as defined when you build your code, it executes code in a `#ifdef _WIN32` statement. If Polyspace does not consider that macro as defined, you must use this option to replace the macro with 1.

Depending on your settings for `Compiler (-compiler)`, some macros are defined by default. Use this option to define macros that are not implicitly defined.

Typically, you recognize from compilation errors that a certain macro is not defined. For instance, the following code does not compile if the macro `_WIN32` is not defined.


```
#ifdef _WIN32
    int env_var;
#endif

void set() {
    env_var=1;
}
```

The error message states that `env_var` is undefined. However, the definition of `env_var` is in the `#ifdef _WIN32` statement. The underlying cause for the error is that the macro `_WIN32` is not defined. You must define `_WIN32`.

Settings

No Default

Using the  button, add a row for the macro you want to define. The definition must be in the format *Macro=Value*. If you want Polyspace to ignore the macro, leave the *Value* blank.

For example:

- `name1=name2` replaces all instances of `name1` by `name2`.
- `name=` instructs the software to ignore `name`.
- `name` with no equals sign or value replaces all instances of `name` by 1. To define a macro to execute code in a `#ifdef macro_name` statement, use this syntax.

Tips

- If Polyspace does not support a non-ANSI keyword and shows a compilation error, use this option to replace all occurrences of the keyword with a blank string in preprocessed code. The replacement occurs only for the purposes of the analysis. Your original source code remains intact.

For instance, if your compiler supports the `__far` keyword, to avoid compilation errors:

- In the user interface, enter `__far=`.
- On the command line, use the flag `-D __far`.

The software replaces the `__far` keyword with a blank string during preprocessing. For example:

```
int __far* pValue;
```

is converted to:

```
int * pValue;
```

- Polyspace recognizes keywords such as `restrict` and does not allow their use as identifiers. If you use those keywords as identifiers (because your compiler does not

recognize them as keywords), replace the disallowed name with another name using this option. The replacement occurs only for the purposes of the analysis. Your original source code remains intact.

For instance, to allow use of `restrict` as identifier:

- In the user interface, enter `restrict=my_restrict`.
- On the command line, use the flag `-D restrict=my_restrict`.

Command-Line Information

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

Parameter: `-D`

No Default

Value: *flag=value*

Example: `polyspace-code-prover-nodesktop -D HAVE_MYLIB -D int32_t=int`

See Also

Disabled preprocessor definitions (`-U`)

Disabled preprocessor definitions (-U)

Undefine macros in preprocessed code

Description

Undefine macros in preprocessed code.

Set Option

User interface: In your project configuration, the option is on the **Macros** node.

Command line: Use the option -U. See “Command-Line Information” on page 1-64.

Why Use This Option

Use this option to emulate your compiler behavior. For instance, if your compiler considers a macro `_WIN32` as undefined when you build your code, it executes code in a `#ifndef _WIN32` statement. If Polyspace considers that macro as defined, you must explicitly undefine the macro.

Some settings for `Compiler (-compiler)` enable certain macros by default. This option allows you undefine the macros.

Typically, you recognize from compilation errors that a certain macro must be undefined. For instance, the following code does not compile if the macro `_WIN32` is defined.


```
#ifndef _WIN32
    int env_var;
#endif

void set() {
    env_var=1;
}
```

The error message states that `env_var` is undefined. However, the definition of `env_var` is in the `#ifndef _WIN32` statement. The underlying cause for the error is that the macro `_WIN32` is defined. You must undefine `_WIN32`.

Settings

No Default

Using the  button, add a new row for each macro being undefined.

Command-Line Information

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

Parameter: `-U`

No Default

Value: *macro*

Example: `polyspace-code-prover-nodesktop -U HAVE_MYLIB -U USE_COM1`

See Also

Preprocessor definitions (`-D`)

Code from DOS or Windows file system (-dos)

Consider that file paths are in MS-DOS style

Description

Specify that DOS or Windows files are provided for analysis.

Set Option

User interface: In your project configuration, the option is on the **Environment Settings** node.

Command line: Use the option `-dos`. See “Command-Line Information” on page 1-66.

Why Use This Option

Use this option if the contents of the **Include** or **Source** folder come from a DOS or Windows file system. The option helps you resolve case sensitivity and control character issues.

Settings

On (default)

Analysis understands file names and include paths for Windows/DOS files

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"
```

Off

Characters are not controlled for files names or paths.

Command-Line Information

Parameter: -dos

Default: Off

Example: polyspace-code-prover-nodesktop -dos -I ./
my_copied_include_dir -D test=1

Stop analysis if a file does not compile (`-stop-if-compile-error`)

Specify that a compilation error must stop the analysis

Description

Specify that even a single compilation error must stop the analysis.

Set Option

User interface: In the **Configuration** pane, the option is on the **Environment Settings** node.

Command line: Use the option `-stop-if-compile-error`. See “Command-Line Information” on page 1-68.

Why Use This Option

Use this option to first resolve all compilation errors and then perform the Polyspace analysis. This sequence ensures that all files are analyzed.

Otherwise, only files without compilation errors are fully analyzed. The analysis might return some results for files that do not compile. If a file with compilation errors contains a function definition, the analysis considers the function undefined. This assumption can sometimes make the analysis less precise.







The option is more useful for a Code Prover analysis because the Code Prover run-time checks rely more heavily on range propagation across functions.

Settings

On

The analysis stops even if a single compilation error occurs.

You see the compilation errors on the **Output Summary** pane.

Type	Message	File	Line	Col
	C verification starts at Thu Dec 17 22:26:17 2015			
	6 core(s) detected but the verification uses 4 core(s).			
	identifier "x" is undefined	my_file.c	1	
	Failed compilation.	my_file.c		
	Verifier has detected compilation error(s) in the code.			
	Exiting because of previous error			

For information on how to resolve the errors, see “Troubleshoot Compilation and Linking Errors”.

Despite compilation errors, you can see some analysis results, for instance, coding rule violations.

Off (default)

The analysis does not stop because of compilation errors, but only files without compilation errors are analyzed. The analysis does not consider files that do not compile. If a file with compilation errors contains a function definition, the analysis considers the function undefined. If the analysis needs the definition of such a function, it makes broad assumptions about the function.

- The function return value can take any value in the range allowed by its data type.
- The function can modify arguments passed by reference so that they can take any value in the range allowed by their data types.

If the assumptions are too broad, the analysis can be less precise. For instance, a run-time check can flag an operation in orange even though it does not fail in practice.

If compilation errors occur, the **Dashboard** pane has a link, which shows that some files failed to compile. You can click the link and see the compilation errors on the **Output Summary** pane.

Command-Line Information

Parameter: `-stop-if-compile-error`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources filename -stop-if-compile-error`

Introduced in R2017a

Command/script to apply to preprocessed files (`-post-preprocessing-command`)

Specify command or script to run on source files after preprocessing phase of analysis

Description

Specify a command or script to run on each source file after preprocessing.

Set Option

User interface: In your project configuration, the option is on the **Environment Settings** node.

Command line: Use the option `-post-preprocessing-command`. See “Command-Line Information” on page 1-72.

Why Use This Option

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

Use a script only if the existing analysis options do not meet your requirements. For instance:

- For direct replacement of one keyword with another, use the option `Preprocessor definitions (-D)`.

However, the option does not allow search and replacement involving regular expressions. For regular expressions, use a script.


- For mapping your library function to a standard library function, use the option `-function-behavior-specifications`.

However, the option supports mapping to only a subset of standard library functions. To map to an unsupported function, use a script.

If you are unsure about removing or replacing an unsupported construct, do not use this option. Contact MathWorks Support for guidance.

Settings

No Default

Enter full path to the command or script or click  to navigate to the location of the command or script. After the verification, this script is executed.

Tips

- Your script must be designed to process the standard output from preprocessing and produce its results in accordance with that standard output.
- Your script must preserve the number of lines in the preprocessed file. In other words, it must not add or remove entire lines to or from the file.

Adding a line or removing one can potentially result in some unpredictable behavior on the location of checks and macros in the Polyspace user interface.

- For a Perl script, in Windows, specify the full path to the Perl executable followed by the full path to the script.

For example:

- To specify a Perl command that replaces all instances of the `far` keyword, enter `matlabroot\sys\perl\win32\bin\perl.exe -p -e "s/far//g"`.
- To specify a Perl script `replace_keyword.pl` that replaces all instances of a keyword, enter `matlabroot\sys\perl\win32\bin\perl.exe <absolute_path>\replace_keyword.pl`.

Here, `matlabroot` is the location of the current MATLAB installation such as `C:\Program Files\MATLAB\R2015b\` and `<absolute_path>` is the location of the Perl script.

- Use this Perl script as template. The script removes all instances of the `far` keyword.

```
#!/usr/bin/perl

binmode STDOUT;

# Process every line from STDIN until EOF
while ($line = <STDIN>)
{

    # Remove far keyword
    $line =~ s/far//g;

    # Print the current processed line to STDOUT
    print $line;
}
```

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

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

For complete list of regular expressions, see Perl documentation.

- When you specify this option, the Compilation Assistant is automatically disabled.

Command-Line Information

Parameter: `-post-preprocessing-command`

Value: Path to executable file or command in quotes

No Default

Example in Linux®: `polyspace-code-prover-nodesktop -sources file_name -post-preprocessing-command `pwd`/replace_keyword.pl`

Example in Windows: `polyspace-code-prover-nodesktop -sources file_name -post-preprocessing-command "C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\replace_keyword.pl"`

See Also

Command/script to apply after the end of the code verification (-post-analysis-command)

Topics

“Specify Analysis Options”

“Remove or Replace Keywords Before Compilation”

Include (`-include`)

Specify files to be `#include`-ed by each C file in analysis

Description

Specify files to be `#include`-ed by each C file involved in the analysis. The software enters the `#include` statements in the preprocessed code used for analysis, but does not modify the original source code.

Set Option

User interface: In your project configuration, the option is on the **Environment Settings** node.

Command line: Use the option `-include`. See “Command-Line Information” on page 1-75.

Why Use This Option

There can be many reasons why you want to `#include` a file in all your source files.

For instance, you can collect in one header file all workarounds for compilation errors. Use this option to provide the header file for analysis. Suppose you have compilation issues because Polyspace does not recognize certain compiler-specific keywords. To work around the issues, `#define` the keywords in a header file and provide the header file with this option.

Settings

No Default

Specify the file name to be included in every file involved in the analysis.

Polyspace still acts on other directives such as `#include <include_file.h>`.

Command-Line Information

Parameter: `-include`

Default: None

Value: *file* (Use `-include` multiple times for multiple files)

Example: `polyspace-code-prover-nodesktop -include `pwd`/sources/a_file.h -include /inc/inc_file.h`

See Also

Topics

“Gather Compilation Options Efficiently”

Include folders (-I)

View include folders used for analysis

Description

View the include folders used for analysis.

Set Option

This is not an option that you set in your project configuration. You can only view the include folders in the configuration associated with a result. For instance, in the user interface:

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

Settings

This is a read-only option available only when viewing results. Unlike other options, you do not specify include folders on the **Configuration** pane. Instead, you add your include folders on the **Project Browser** pane.

Command-Line Information

Parameter: -I

Value: Folder name

Example: polyspace-code-prover-nodesktop -I /com1/inc -I /com1/sys/inc

See Also

-I | Include (-include)

Ignore link errors (`-no-extern-c`)

Ignore certain linking errors

Description

Specify that the analysis must ignore certain linking errors.

Set Option

User interface: In your project configuration, the option is on the **Environment Settings** node. See “Dependency” on page 1-79 for other options that you must also enable.

Command line: Use the option `-no-extern-C`. See “Command-Line Information” on page 1-79.

Why Use This Option

Some functions may be declared inside an `extern "C" { }` block 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 resolve all the extern C linkage errors.

Settings

On

Ignore linking errors if possible.

Off (default)

Stop analysis for linkage errors.

Dependency

This option is available only if you set `Source code language (-lang)` to `CPP`.

Command-Line Information

Parameter: `-no-extern-C`

Default: `off`

Example: `polyspace-code-prover-nodesktop -lang cpp -no-extern-C`

Constraint setup (-data-range-specifications)

Constrain global variables, function inputs and return values of stubbed functions

Description

Specify constraints (also known as data range specifications or DRS) for global variables, function inputs and return values of stubbed functions using a **Constraint Specification** template file. The template file is an XML file that you can generate in the Polyspace user interface.

Set Option

User interface: In your project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-data-range-specifications`. See “Command-Line Information” on page 1-81.

Why Use This Option

Use this option for specifying constraints outside your code.

Polyspace uses the code that you provide to make assumptions about items such as variable ranges and allowed buffer size for pointers. Sometimes the assumptions are broader than what you expect because:

- You have not provided the complete code. For example, you did not provide some of the function definitions.
- Some of the information about variables is available only at run time. For example, some variables in your code obtain values from the user at run time.

Because of these broad assumptions, Polyspace can consider more execution paths than those paths that occur at run time. If an operation fails along one of the execution paths,


Polyspace places an orange check on the operation. If that execution path does not occur at run time, the orange check indicates a false positive.

To reduce the number of such false positives, you can specify additional constraints on global variables, function inputs, and return values of stubbed functions.

After you specify your constraints, you can save them as an XML file to use them for subsequent analyses. If your source code changes, you can update the previous constraints. You do not have to create a new constraint template.

Settings

No Default

Enter full path to the template file. Alternately, click  to open a **Constraint Specification** wizard. This wizard allows you to generate a template file or navigate to an existing template file.

For more information, see “Specify External Constraints”.

Command-Line Information

Parameter: -data-range-specifications

Value: *file*

No Default

Example: `polyspace-code-prover-nodesktop -sources file_name -data-range-specifications "C:\DRS\range.xml"`

See Also

Functions to stub (-functions-to-stub) | Ignore default initialization of global variables (-no-def-init-glob)

Topics

“Specify Analysis Options”

“Constrain Global Variable Range”

“Constrain Function Inputs”

“Constrain Stubbed Functions”
“Constraints”
“XML File Format for Constraints”

Ignore default initialization of global variables (`-no-def-init-glob`)

Consider global variables as uninitialized

Description

This option applies to Code Prover only. It does not affect a Bug Finder analysis.

Specify that Polyspace must not consider global and static variables as initialized.

Set Option

User interface: In your project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-no-def-init-glob`. See “Command-Line Information” on page 1-84.

Why Use This Option

The C99 Standard specifies that global variables are implicitly initialized. The default analysis follows the Standard and considers this implicit initialization.

If you want to initialize specific global variables explicitly, use this option to find the instances where global variables are not explicitly initialized.

Settings

On

Polyspace ignores implicit initialization of global and static variables. The verification generates a red **Non-initialized variable** error if your code reads a global or static variable before writing to it.

Off (default)

Polyspace considers global variables and static variables to be initialized according to C99 or ISO C++ standards. For instance, the default values are:

- 0 for `int`
- 0 for `char`
- 0.0 for `float`

Tips

- If you initialize a global variable using the generated `main`:
 - Polyspace does not produce a red **Non-initialized variable** error if your code reads the variable before writing to it. The error is not produced even if you turn on the option **Ignore default initialization of global variables**.
 - Polyspace considers that before the first write operation on the variable in a function, the variable can take any value allowed by its type.

For more information on initializing global variables using the generated `main`, see `Variables to initialize (-main-generator-writes-variables)`.

- Static local variables have the same lifetime as global variables even though their visibility is limited to the function where they are defined. Therefore, the option applies to static local variables.

Command-Line Information

Parameter: `-no-def-init-glob`

Default: Off

See Also

Non-initialized variable

Topics

“Specify Analysis Options”

Functions to stub (-functions-to-stub)

Specify functions to stub during analysis

Description

This option affects a Code Prover analysis only.

Specify functions to stub during analysis.

For specified functions, Polyspace :

- Ignores the function definition even if it exists.
- Assumes that the function inputs and outputs have full range of values allowed by their type.

Set Option

User interface: In your project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-functions-to-stub`. See “Command-Line Information” on page 1-87.

Why Use This Option

If you want the analysis to ignore the code in a function body, you can stub the function.



For instance:

- Suppose you have not completed writing the function and do not want the analysis to consider the function body. You can use this option to stub the function and then specify constraints on its return value and modifiable arguments.
- Suppose the analysis of a function body is imprecise. The analysis assumes that the function returns all possible values that the function return type allows. You can use this option to stub the function and then specify constraints on its return value.

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

When entering function names, use either the basic syntax or, to differentiate overloaded functions, the argument syntax. For the argument syntax, separate function arguments with semicolons. See the following code and table for examples.

```
//simple function
void test(int a, int b);

//C++ template function
Template <class myType>
myType test(myType a, myType b);

//C++ class method
class A {
    public:
    int test(int var1, int var2);
};

//C++ template class method
template <class myType> class A
{
    public:
    myType test(myType var1, myType var2);
};
```

Function Type	Basic Syntax	Argument Syntax
Simple function	test	test(int; int)

Function Type	Basic Syntax	Argument Syntax
C++ template function	test	test(myType; myType)
C++ class method	A::test	A::test(int;int)
C++ template class method	A<myType>::test	A<myType>::test(myType;myType)

Tips

- Polyspace makes assumptions about the arguments and return values of stubbed functions. For example, Polyspace assumes the return values of stubbed functions are full range. These assumptions can affect checks in other sections of the code. See “Stubbed Functions” on page 4-4.
- If you stub a function, you can constrain the range of function arguments and return value. See “Constrain Stubbed Functions”.
- For C functions, these special characters are allowed: () < > ; _

For C++ functions, these special characters are allowed : () < > ; _ * & []

Space characters are allowed for C++, but are not allowed for C functions.

Command-Line Information

Parameter: -functions-to-stub

No Default

Value: *function1*[, *function2*[, ...]]

Example: polyspace-code-prover-nodesktop -sources *file_name* -functions-to-stub *function_1*,*function_2*

See Also

Constraint setup (-data-range-specifications)

Topics

“Specify Analysis Options”

Generate stubs for Embedded Coder lookup tables (`-stub-embedded-coder-lookup-table- functions`)

Stub autogenerated functions that use lookup tables and model them more precisely

Description

This option is available only for model-generated code. The option is relevant only if you generate code from a Simulink® model that uses Lookup Table blocks using MathWorks code generation products.

Specify that the verification must stub autogenerated functions that use certain kinds of lookup tables in their body. The lookup tables in these functions use linear interpolation and do not allow extrapolation. That is, the result of using the lookup table always lies between the lower and upper bounds of the table.

Set Option

If you are running verification from Simulink, use the option “Stub lookup tables” on page 11-10 in Simulink Configuration Parameters, which performs the same task.

User interface: In your Polyspace project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-stub-embedded-coder-lookup-table-functions`. See “Command-Line Information” on page 1-90.

Why Use This Option

If you use this option, the verification is more precise and has fewer orange checks. The verification of lookup table functions is usually imprecise. The software has to make certain assumptions about these functions. To avoid missing a run-time error, the verification assumes that the result of using the lookup table is within the full range allowed by the result data type. This assumption can cause many unproven results

(orange checks) when a lookup table function is called. By using this option, you narrow down the assumption. For functions that use lookup tables with linear interpolation and no extrapolation, the result is at least within the bounds of the table.

The option is relevant only if your model has Lookup Table blocks. In the generated code, the functions corresponding to Lookup Table blocks also use lookup tables. The function names follow specific conventions. The verification uses the naming conventions to identify if the lookup tables in the functions use linear interpolation and no extrapolation. The verification then replaces such functions with stubs for more precise verification.

Settings

On (default)

For autogenerated functions that use lookup tables with linear interpolation and no extrapolation, the verification:

- Does not check for run-time errors in the function body.
- Calls a function stub instead of the actual function at the function call sites. The stub ensures that the result of using the lookup table is within the bounds of the table.

To identify if the lookup table in the function uses linear interpolation and no extrapolation, the verification uses the function name. In your analysis results, you see that the function is not analyzed. If you place your cursor on the function name, you see the following message:

```
Function has been recognized as an Embedded Coder Lookup-Table function.  
It was stubbed by Polyspace to increase precision.  
Unset the -stub-embedded-coder-lookup-table-functions option to analyze  
the code below.
```

Off

The verification does not stub autogenerated functions that use lookup tables.

Tips

- The option applies to only autogenerated functions. If you integrate your own C/C++ S-Function using lookup tables with the model, these functions do not follow the naming conventions for autogenerated functions. The option does not cause them to be stubbed. If you want the same behavior for your handwritten lookup table functions as the autogenerated functions, use the option `-function-behavior-specifications` and map your function to the `__ps_lookup_table_clip` function.
- If you run verification from Simulink, the option is on by default. For certification purposes, if you want your verification tool to be independent of the code generation tool, turn off the option.

Command-Line Information

Parameter: `-stub-embedded-coder-lookup-table-functions`

Default: On

Example: `polyspace-code-prover-nodesktop -sources file_name -stub-embedded-coder-lookup-table-functions`

See Also

Topics

“Specify Analysis Options”

Introduced in R2016b

Generate results for sources and (-generate-results-for)

Specify files on which you want analysis results

Description

Specify files on which you want analysis results.

The option applies only to coding rule violations and code metrics. You cannot suppress run-time checks from select source and header files.

Set Option

User interface: In your project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-generate-results-for`. See “Command-Line Information” on page 1-93.

Why Use This Option

Use this option to see results in header files that are most relevant to you.

For instance, by default, results are generated on header files that are located in the same folder as the source files. Often, other header files belong to a third-party library. Though these header files are required for a precise analysis, you are not interested in reviewing findings in those headers. Therefore, by default, results are not generated for those headers. If you *are interested* in certain headers from third-party libraries, change the default value of this option.

Settings

Default: `source-headers`

source-headers

Results appear on source files and header files in the same folder as the source files or in subfolders of source file folders.

The source files are the files that you add to the **Source** folder of your Polyspace project (or use with the argument `-sources` at the command line).

all-headers


Results appear on source files and all header files. The header files can be in the same folder as source files, in subfolders of source file folders or in include folders.

The source files are the files that you add to the **Source** folder of your Polyspace project (or use with the argument `-sources` at the command line).

The include folders are the folders that you add to the **Include** folder of your Polyspace project (or use with the argument `-I` at the command line).

custom

Results appear on source files and the files that you specify. If you enter a folder name, results appear on header files in that folder.

Click  to add a field. Enter a file or folder name.

Tips

- 1 Use this option in combination with appropriate values for the option `Do not generate results for (-do-not-generate-results-for)`.

If you choose `custom` and the values for the two options conflict, the more specific value determines the display of results. For instance, in the following examples, the value for the option **Generate results for sources and** is more specific.

Generate results for sources and	Do not generate results for	Final Result
custom: C:\Includes \Custom_Library\	custom: C:\Includes	Results are displayed on header files in C:\Includes\Custom_Library\ but not generated for other header files in C:\Includes and its subfolders.
custom: C:\Includes \my_header.h	custom: C:\Includes\	Results are displayed on the header file my_header.h in C:\Includes\ but not generated for other header files in C:\Includes\ and its subfolders.

Using these two options together, you can suppress results from all files in a certain folder but unsuppress select files in those folders.

- 2 If you choose `all-headers` for this option, results are displayed on all header files irrespective of what you specify for the option **Do not generate results for**.

Command-Line Information

Parameter: `-generate-results-for`

Value: `all-headers | custom=file1[, file2[, ...]] | folder1[, folder2[, ...]]`

Example: `polyspace-code-prover-nodesktop -lang c -sources file_name -misra2 required-rules -generate-results-for "C:\usr\include"`

See Also

Topics

“Exclude Files from Rules Checking”

Introduced in R2016a

Do not generate results for (`-do-not-generate-results-for`)

Specify files on which you do not want analysis results

Description

Specify files on which you do not want analysis results.

The option applies only to coding rule violations, code metrics and unused global variables. You cannot suppress run-time checks from source and header files.

Set Option

User interface: In your project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-do-not-generate-results-for`. See “Command-Line Information” on page 1-99.

Why Use This Option

Use this option to see results in header files that are most relevant to you.

For instance, by default, results are generated on header files that are located in the same folder as the source files. If you are not interested in reviewing the findings in those headers, change the default value of this option.

Settings

Default: `include-folders`

`include-folders`

Results are not generated for header files in include folders.

The include folders are the folders that you add to the **Include** folder of your Polyspace project (or use with the argument `-I` at the command line).

`all-headers`


Results are not generated for all header files. The header files can be in the same folder as source files, in subfolders of source file folders or in include folders.

The source files are the files that you add to the **Source** folder of your Polyspace project (or use with the argument `-sources` at the command line).

The include folders are the folders that you add to the **Include** folder of your Polyspace project (or use with the argument `-I` at the command line).

`custom`

Results are not generated for the files that you specify. If you enter a folder name, results are suppressed from files in that folder.

Click  to add a field. Enter a file or folder name.

Tips

- 1 Use this option appropriately in combination with appropriate values for the option `Generate results for sources and` (`-generate-results-for`).

If you choose `custom` and the values for the two options conflict, the more specific value determines the display of results. For instance, in the following examples, the value for the option **Generate results for sources and** is more specific.

Generate results for sources and	Do not generate results for	Final Result
<code>custom:</code> <code>C:\Includes</code> <code>\Custom_Library\</code>	<code>custom:</code> <code>C:\Includes</code>	Results are displayed on header files in <code>C:\Includes</code> but not generated for other header files in <code>C:\Includes</code> and its subfolders.

Generate results for sources and	Do not generate results for	Final Result
custom: C:\Includes \my_header.h	custom: C:\Includes\ 	Results are displayed on the header file my_header.h in C:\Includes\ but not generated for other header files in C:\Includes\ and its subfolders.

Using these two options together, you can suppress results from all files in a certain folder but unsuppress select files in those folders.

- 2 If you choose `all-headers` for this option, results are suppressed from all header files irrespective of what you specify for the option **Generate results for sources and**.
- 3 If a coding rule violation involves two files and you do not generate results for one of the files, the rule violation still appears. For instance, if you define two variables with similar-looking names in files `myFile.cpp` and `myFile.h`, you get a violation of the MISRA@ C++ rule 2-10-1, even if you do not generate results for `myFile.h`. MISRA C++ rule 2-10-1 states that different identifiers must be typographically unambiguous.

The following results can involve more than one file:

MISRA C: 2004 Rules

- MISRA C@: 2004 Rule 5.1 — Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
- MISRA C: 2004 Rule 5.2 — Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
- MISRA C: 2004 Rule 8.8 — An external object or function shall be declared in one file and only one file.
- MISRA C: 2004 Rule 8.9 — An identifier with external linkage shall have exactly one external definition.

MISRA C: 2012 Directives and Rules

- MISRA C: 2012 Directive 4.5 — Identifiers in the same name space with overlapping visibility should be typographically unambiguous.

- MISRA C: 2012 Rule 5.2 — Identifiers declared in the same scope and name space shall be distinct.
- MISRA C: 2012 Rule 5.3 — An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
- MISRA C: 2012 Rule 5.4 — Macro identifiers shall be distinct.
- MISRA C: 2012 Rule 5.5 — Identifiers shall be distinct from macro names.
- MISRA C: 2012 Rule 8.5 — An external object or function shall be declared once in one and only one file.
- MISRA C: 2012 Rule 8.6 — An identifier with external linkage shall have exactly one external definition.

MISRA C++ Rules

- MISRA C++ Rule 2-10-1 — Different identifiers shall be typographically unambiguous.
- MISRA C++ Rule 2-10-2 — Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
- MISRA C++ Rule 3-2-2 — The One Definition Rule shall not be violated.
- MISRA C++ Rule 3-2-3 — A type, object or function that is used in multiple translation units shall be declared in one and only one file.
- MISRA C++ Rule 3-2-4 — An identifier with external linkage shall have exactly one definition.
- MISRA C++ Rule 7-5-4 — Functions should not call themselves, either directly or indirectly.
- MISRA C++ Rule 15-4-1 — If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.

JSF C++ Rules

- JSF C++ Rule 46 — User-specified identifiers (internal and external) will not rely on significance of more than 64 characters.
- JSF C++ Rule 48 — Identifiers will not differ by only a mixture of case, the presence/absence of the underscore character, the interchange of the letter O with the number 0 or the letter D, the interchange of the letter I with the number 1 or the letter l, the interchange of the letter S with the number 5, the interchange of

the letter z with the number 2 and the interchange of the letter n with the letter h.

- JSF C++ Rule 137 — All declarations at file scope should be static where possible.
 - JSF C++ Rule 139 — External objects will not be declared in more than one file.
- 4 If a global variable is never used after declaration, it appears in your verification results as an unused global variable on page 9-15. However, if it is declared in a file for which you do not want results, you do not see the unused variable in your verification results.
 - 5 If a result (coding rule violation or Bug Finder defect) is inside a macro, Polyspace typically shows the result on the macro definition instead of the macro occurrences so that you review the result only once. Even if the macro is used in a suppressed file, the result is still shown on the macro definition, *if the definition occurs in an unsuppressed file*.

Command-Line Information

Parameter: -do-not-generate-results-for

Value: all-headers | custom=*file1*[,*file2*[,...]] |
folder1[,*folder2*[,...]]

Example: polyspace-code-prover-nodesktop -lang c -sources *file_name* -
misra2 required-rules -do-not-generate-results-for "C:\usr\include"

See Also

Topics

“Exclude Files from Rules Checking”

Introduced in R2016a

No STL stubs (`-no-stl-stubs`)

Do not use Polyspace implementations of functions in the Standard Template Library

Description

This option applies to Code Prover only. It does not affect a Bug Finder analysis.

Specify that the verification must not use Polyspace implementations of the Standard Template Library.

Set Option

User interface: In your project configuration, the option is on the **Inputs & Stubbing** node. See “Dependency” on page 1-101 for other options that you must also enable.

Command line: Use the option `-no-stl-stubs`. See “Command-Line Information” on page 1-101.

Why Use This Option

The analysis uses an efficient implementation of all class templates from the Standard Template Library (STL). If your compiler redefines the templates, in some cases, your compiler implementation can conflict with the Polyspace implementation.

Use this option to prevent Polyspace from using its implementations of STL templates. You must also explicitly provide the path to your compiler includes. See “C++ Standard Template Library Stubbing Errors”.

Settings

On

The verification does not use Polyspace implementations of the Standard Template Library.

Off (default)

The verification uses efficient Polyspace implementations of the Standard Template Library.

Dependency

This option is available only if you set `Source code language (-lang)` to `CPP` or `C-CPP`.

Command-Line Information

Parameter: `-no-stl-stubs`

Default: Off

See Also

Topics

“Specify Analysis Options”

Enable automatic concurrency detection for Code Prover (`-enable-concurrency-detection`)

Automatically detect certain families of multithreading functions

Description

This option affects a Code Prover analysis only.

Specify whether to use the automatic concurrency detection for POSIX®, VxWorks®, Windows, and μ C/OS II multithreading functions.

Set Option

User interface: In your project configuration, the option is available on the **Multitasking** node.

Command line: Use the option `-enable-concurrency-detection`. See “Command-Line Information” on page 1-106.

Why Use This Option

If you use this option, Polyspace determines your multitasking model from your use of multithreading functions.

In some cases, using automatic concurrency detection can slow down the Code Prover analysis. In those cases, you can explicitly specify your multitasking model using the option `Configure multitasking` manually.

Settings

On

If you use POSIX, VxWorks, Windows, or μ C/OS II functions for multitasking, the analysis automatically detects your multitasking model from your code.

The supported multitasking functions are the following:

Family	Thread Creation	Critical Section Begins	Critical Section Ends
POSIX	<code>pthread_create</code>	<code>pthread_mutex_lock</code>	<code>pthread_mutex_unlock</code>
VxWorks	<code>taskSpawn</code>	<code>semTake</code>	<code>semGive</code>
Windows	<code>CreateThread</code>	<code>EnterCriticalSection</code>	<code>LeaveCriticalSection</code>
µC/OS II	<code>OSTaskCreate</code>	<code>OSMutexPend</code>	<code>OSMutexPost</code>

To activate automatic detection of concurrency primitives for VxWorks, use the VxWorks template. For more information on templates, see “Create Project Using Configuration Template”.

Note For VxWorks, concurrency detection is possible only if the multitasking functions are created from an entry point named `main`. If the entry point has a different name, such as `vxworks_entry_point`, do the following:

- 1 Verify whole application: Select `verify the whole application`.
 - 2 Preprocessor definitions (-D): In preprocessor definitions, set `vxworks_entry_point=main`.
 - 3 Enable automatic concurrency detection for Code Prover (`-enable-concurrency-detection`): Enable `automatic concurrency detection`.
-

Off (default)

The analysis does not attempt to detect the multitasking model from your code.

If you want to manually configure your multitasking model, see `Configure multitasking manually`.

Limitations

The multitasking model extracted by Polyspace does not include some features. Polyspace cannot model:

- Thread priorities and attributes — Ignored by Polyspace
- Recursive semaphores
- Unbounded thread identifiers — Warning

For example:

```
extern pthread_t ids[]
```

Or

```
pthread_t* ids = (pthread_t* malloc(n*sizeof(pthread_t)))
```

- Calls to concurrency primitive through high-order calls — Warning.
- Termination of threads — Polyspace ignores `pthread_join`, and replaces `pthread_exit` by a standard `exit`.
- Shared local variables — Only global variables are considered shared. If a local variable is accessed by multiple threads, the analysis does not take into account the shared nature of the variable.

Example

In this example, the analysis does not take into account that the local variable `x` can be accessed by both `task1` and `task2` (after the new thread is created).

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

void* task2(void* args) {
    int* x = (int*) args;
    *x = 1;
    return (void*)x;
}

void task1() {
    int x;
    x = 2;
    pthread_t id;
    (void)pthread_create(&id,NULL,task2,(void*) &x);
    /* x (local var) passed to task2 */
    x = 3 ;

    /* Unknown thread priority means x = 1 OR x = 3.*/
}
```



```

    /* However, the analysis considers x = 3 */
    /* Assertion below is green */
    assert(x==3);
}

int main(void) {
    task1();
    return 0;
}

```

- Shared dynamic memory — Only global variables are considered shared. If a dynamically allocated memory region is accessed by multiple threads, the analysis does not take into account its shared nature.

Example

In this example, the analysis does not take into account that `lx` points to a shared memory region. The region can be accessed by both `task1` and `task2` (after the new thread is created). The Code Prover analysis also reports `lx` as a non-shared variable.

```

#include <pthread.h>
#include <stdlib.h>

static int *lx;

void* task2(void* args) {
    int* x = (int*) args;
    *x = 1;
    return (void*)x;
}

void task1() {
    pthread_t id;
    lx = (int *)malloc(sizeof(int));

    if(lx==NULL) exit(1);

    (void)pthread_create(&id, NULL, task2, (void*) lx);

    *lx = 3 ;

    /* Unknown thread priority means *lx = 1 OR *lx = 3.*/
    /* However, the analysis considers *lx = 3 */
    /* Assertion below is green */
    assert(*lx==3);
}

```

```
    }  
  
    int main(void) {  
        task1();  
        return 0;  
    }
```

Command-Line Information

Parameter: `-enable-concurrency-detection`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -enable-concurrency-detection`

See Also

Entry points (`-entry-points`) | Critical section details (`-critical-section-begin` `-critical-section-end`) | Temporally exclusive tasks (`-temporal-exclusions-file`)

Topics

“Manually Model Tasks if main Contains Infinite Loop”

“Manually Model Scheduling of Tasks”

“Verify Multitasking Applications”

OSEK multitasking configuration (-osek-multitasking)

Set up multitasking configuration from OIL file definition

Description

Specify the OIL files that Polyspace parses to set up the multitasking configuration of your OSEK project.

Set Option

User interface: In the **Configuration** pane, the option is available on the **Multitasking** pane.

Command line: Use the option `-osek-multitasking`. See “Command-Line Information” on page 1-111.

Why Use This Option

If your project includes OIL files, Polyspace can parse these files to set up entry points, interrupts, cyclical tasks, and critical sections. You do not have to set them up manually.

Settings

On

Polyspace looks for and parses OIL files to set up your multitasking configuration.
auto

Look for OIL files in your project source and include folders, but not in their subfolders.

custom

Look for OIL files on the specified path and the path subfolders. You can specify a path to the OIL files or to the folder containing the files.

When you select this option, in your source code, Polyspace supports these OSEK multitasking keywords:

- `TASK`
- `DeclareTask`
- `ActivateTask`
- `DeclareResource`
- `GetResource`
- `ReleaseResource`
- `ISR`
- `DeclareEvent`
- `DeclareAlarm`

Polyspace parses the OIL files that you provide for `TASK`, `ISR`, `RESOURCE`, and `ALARM` definitions. The analysis uses these definitions and the supported multitasking keywords to configure entry points, interrupts, cyclical tasks, and critical sections.

Example: Analyze Your OSEK Multitasking Project

This table lists a source code and corresponding OIL file for an OSEK multitasking application.

OIL File	Source Code
<pre> CPU mySystem { OS myOs { EE_OPT = " EXMAPL CPU_DATA = model APP_SRC = " MULTI_STACK ICD2 = TRUE; }; MCU_DATA = model MODEL = 11 A }; TASK init { AUTOSTART = TRUE; PRIORITY = 1; STACK = SHARED ; SCHEDULE = FULL; }; TASK afterinit1 { AUTOSTART = TRUE; PRIORITY = 1; STACK = SHARED ; SCHEDULE = FULL; }; RESOURCE res1 { RESOURCEPROPERTY }; TASK Cyclic_task { AUTOSTART = TRUE; PRIORITY = 1; STACK = SHARED ; SCHEDULE = FULL; </pre>	<pre> #include <assert.h> #include "Header_file" int var1; int var2; CPU var3; DeclareAlarm(Cyclic_task_activate); DeclareResource(res1); DeclareTask(init); TASK(afterinit1); CPU { TASK(init) // entry point 12345678; }; var2++; ActivateTask(afterinit1); var3++; GetResource(res1); // critical section begins var1++; ReleaseResource(res1); // critical section ends }; TASK(afterinit1) // entry point { var3++; var2++; GetResource(res1); // critical section begins var1++; ReleaseResource(res1); // critical section ends }; int STANDARD; void func() { var4++; }; TASK(Cyclic_task) // cyclic task </pre>

OIL File	Source Code
<pre> }; ALARM Cyclic_task_activate { ACTION = ACTIVATETASK { TASK = Cyclic_taskmain() }; AUTOSTART = TRUE { ALARMTIME = 50; CYCLETIME = 100; }; }; }; </pre>	<pre> func(); {} {} </pre>

To set up your multitasking configuration and analyze the code:

- 1 Copy the preceding code examples and save them on your machine as `osek.oil` and `osek.c`, for instance in `C:\Polyspace_workspace\OSEK`.
- 2 Run an analysis on your OSEK project by using the command:

```
polyspace-code-prover-nodesktop -sources C:\Polyspace_workspace\OSEK\osek.c ^
-I Include_Path -osek-multitasking auto
```

Include_Path is the path to the include folder containing *Header_file*, your header files with OSEK function declarations.

Polyspace detects that `var3` is a potentially unprotected global variable on page 9-9 because it is used in tasks `init` and `afterinit1` with no protection from interruption during the read and write operations. The analysis also shows that the cyclic task operation on `var4` can potentially cause an overflow on page 3-93.

```

#include <assert.h>
#include "Header_file"

int var1;
int var2;
int var3;

...
void func()
{
    var4++;
}

```

Variable `var2` is not shared because `afterinit1` goes to an active state (`ActivateTask()`) after `init` increments `var2`. Variable `var1` is a protected variable on page 9-2 through the critical sections from the `GetResource()` and `ReleaseResource()` calls.

To see how Polyspace models the `TASK`, `ISR`, and `RESOURCE` definitions from your OIL files, search the result log file for "OSEK configuration from oil-files". To access the log file from the user interface, select **Window > Show/Hide View > Run Log**. The log file is located inside your project results folder.

Off (default)

Polyspace does not set up a multitasking configuration for your OSEK project.

Additional Considerations

- The analysis ignores `TerminateTask()` declarations in your source code and considers that subsequent code is executed.
- Polyspace ignores syntax elements of your OIL files that do not follow the syntax defined here.

Command-Line Information

Parameter: `-osek-multitasking`

Value: `auto | custom='file1 [,file2, dir1,...]'`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources source_path -I include_path -osek-multitasking custom='path\to\file1.oil, path\to\dir'`

See Also

Topics

“Verify Multitasking Applications”

“Manually Model Tasks if main Contains Infinite Loop”

“Manually Model Scheduling of Tasks”

“Manually Protect Shared Variables from Concurrent Access”

Introduced in R2017b

Configure multitasking manually

Consider that code is intended for multitasking

Description

Specify whether your code is a multitasking application. This option allows you to manually configure the multitasking structure for Polyspace.

Set Option

User interface: In your project configuration, the option is available on the **Multitasking** node.

Command line: See “Command-Line Information” on page 1-114.

Why Use This Option

If you use the option `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`, Polyspace determines your multitasking model from your use of multithreading functions. The automatic detection is supported only for POSIX, VxWorks, Windows, and μ C/OS II multithreading functions. However, in some cases, using automatic concurrency detection can slow down the Code Prover analysis.

In cases where automatic concurrency detection is not supported, you can explicitly specify your multitasking model by using this option. Once you select this option, you can explicitly specify your entry point functions, cyclic tasks, interrupts and protection mechanisms for shared variables, such as critical section details.

The analysis uses your specifications to determine:

- Whether a global variable is shared.
See “Global Variables”.
- Whether a run-time error can occur.

For instance, if the operation `var++` occurs in the body of a cyclic task and you do not impose a limit on `var`, the operation can overflow. The analysis detects the possible overflow.

Settings

On

The code is intended for a multitasking application.

Polyspace verifies functions that are called by the `main` and other entry-point functions.

Off (default)

The code is not intended for a multitasking application.

- If a `main` exists, Polyspace verifies only those functions that are called by the `main`.
- If a `main` does not exist, Polyspace verifies the functions that you specify. To verify the functions, Polyspace generates a `main` function and calls functions from the generated `main` in a sequence that you specify. For more information, see `Verify module or library (-main-generator)`.

Tips

If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.

Command-Line Information

There is no single command-line option to turn on multitasking analysis. By using the `-entry-points` option, you turn on multitasking analysis.

See Also

`Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)` | `Entry points (-entry-points)` | `Critical`

section details (-critical-section-begin -critical-section-end) |
Temporally exclusive tasks (-temporal-exclusions-file)

Topics

“Verify Multitasking Applications”

“Manually Model Tasks if main Contains Infinite Loop”

“Manually Model Scheduling of Tasks”

Entry points (`-entry-points`)

Specify functions that serve as entry points to your multitasking application

Description

Specify functions that serve as entry points to your code. If the function does not exist, the verification warns you and continues the verification.

Set Option

User interface: In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 1-117 for other options you must also enable.

Command line: Use the option `-entry-points`. See “Command-Line Information” on page 1-118.

Why Use This Option

Use this option when your code is intended for multitasking.

To specify cyclic tasks and interrupts, use the options `Cyclic tasks` (`-cyclic-tasks`) and `Interrupts` (`-interrupts`). Use this option to specify other tasks.

The analysis uses your specifications to determine:

- Whether a global variable is shared.

See “Global Variables”.



- Whether a run-time error can occur.

For instance, if the operation `var++` occurs in the body of a cyclic task and you do not impose a limit on `var`, the operation can overflow. The analysis detects the possible overflow.

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option, first select the option `Configure multitasking manually`.

Tips

- The entry point function must have the form


```
void functionName (void)
```
- If a function `func` takes arguments, you cannot use it directly as entry point. To use `func` as entry point:
 - 1 Create a new function `newFunc`. The declaration must be of the form `void newFunc (void)`.
 - 2 Declare arguments to `func` as volatile variables local to `newFunc`. Call `func` inside `newFunc`.
 - 3 Specify `newFunc` as entry point.
- If you specify a function as an entry point, you must provide its definition. Otherwise, the verification stops with the error message:

```
task func_name must be a userdef function without parameters
```

- If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.

Command-Line Information

Parameter: `-entry-points`

No Default

Value: `function1[,function2[,...]]`

Example: `polyspace-code-prover-nodesktop -sources file_name -entry-points func_1,func_2`

See Also

Critical section details (`-critical-section-begin -critical-section-end`) | Temporally exclusive tasks (`-temporal-exclusions-file`)

Topics

“Specify Analysis Options”

“Verify Multitasking Applications”

“Manually Model Tasks if main Contains Infinite Loop”

“Manually Model Scheduling of Tasks”

Cyclic tasks (`-cyclic-tasks`)

Specify functions that represent cyclic tasks

Description

Specify functions that represent cyclic tasks. The analysis assumes that operations in the function body:

- Can execute any number of times.
- Can be interrupted by noncyclic entry points on page 1-116, other cyclic tasks and interrupts on page 1-122.

To model a cyclic task that cannot be interrupted by other cyclic tasks, specify the task as nonpreemptable. See `-non-preemptable-tasks`.

Set Option

User interface: In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 1-120 for other options you must also enable.

Command line: Use the option `-cyclic-tasks`. See “Command-Line Information” on page 1-121.

Why Use This Option

Use this option to specify cyclic tasks in your multitasking code. The functions that you specify must have the prototype:

```
void function_name(void);
```

The analysis uses your specifications to determine:

- Whether a global variable is shared.

See “Global Variables”.



- Whether a run-time error can occur.

For instance, if the operation `var++` occurs in the body of a cyclic task and you do not impose a limit on `var`, the operation can overflow. The analysis detects the possible overflow.

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option, first select the option `Configure multitasking manually`.

Tips

- The functions representing cyclic tasks must have the form

```
void functionName (void)
```
- If a function `func` takes arguments, you cannot use it directly as a cyclic task. To use `func` as cyclic task:
 - 1 Create a new function `newFunc`. The declaration must be of the form `void newFunc (void)`.
 - 2 Declare arguments to `func` as `volatile` variables local to `newFunc`. Call `func` inside `newFunc`.
 - 3 Specify `newFunc` as cyclic task.
- If you specify a function as a cyclic task, you must provide its definition. Otherwise, the verification stops with the error message:

task *func_name* must be a userdef function without parameters

- If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.

Command-Line Information

Parameter: `-cyclic-tasks`

No Default

Value: `function1[,function2[,...]]`

Example: `polyspace-code-prover-nodesktop -sources file_name -cyclic-tasks func_1,func_2`

See Also

`-preemptable-interrupts` | `-non-preemptable-tasks` | `Interrupts (-interrupts)` | `Entry points (-entry-points)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

“Specify Analysis Options”

“Verify Multitasking Applications”

“Manually Model Tasks if main Contains Infinite Loop”

“Manually Model Scheduling of Tasks”

Introduced in R2016b

Interrupts (`-interrupts`)

Specify functions that represent nonpreemptable interrupts

Description

Specify functions that represent nonpreemptable interrupts. The analysis assumes that operations in the function body:

- Can execute any number of times.
- Cannot be interrupted by noncyclic entry points on page 1-116, cyclic tasks on page 1-119 or other interrupts.

To model an interrupt that can be interrupted by other interrupts, specify the interrupt as preemptable. See `-preemptable-interrupts`.

Set Option

User interface: In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 1-123 for other options you must also enable.

Command line: Use the option `-interrupts`. See “Command-Line Information” on page 1-124.

Why Use This Option

Use this option to specify interrupts in your multitasking code. The functions that you specify must have the prototype:

```
void function_name(void);
```

The analysis uses your specifications to determine:

- Whether a global variable is shared.

See “Global Variables”.



- Whether a run-time error can occur.

For instance, if the operation `var=INT_MAX;` occurs in an interrupt and `var++` occurs in the body of a task, an overflow can occur if the interrupt excepts before the operation in the task. The analysis detects the possible overflow.

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option, first select the option `Configure multitasking manually`.

Tips

- The functions representing interrupts must have the form


```
void functionName (void)
```
- If a function `func` takes arguments, you cannot use it directly as an interrupt. To use `func` as interrupt:
 - 1 Create a new function `newFunc`. The declaration must be of the form `void newFunc (void)`.
 - 2 Declare arguments to `func` as `volatile` variables local to `newFunc`. Call `func` inside `newFunc`.
 - 3 Specify `newFunc` as `interrupt`.
- If you specify a function as an interrupt, you must provide its definition. Otherwise, the verification stops with the error message:

task *func_name* must be a userdef function without parameters

- If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.

Command-Line Information

Parameter: `-interrupts`

No Default

Value: `function1[,function2[,...]]`

Example: `polyspace-code-prover-nodesktop -sources file_name -interrupts func_1,func_2`

See Also

`-preemptable-interrupts` | `-non-preemptable-tasks` | Cyclic tasks (`-cyclic-tasks`) | Entry points (`-entry-points`) | Critical section details (`-critical-section-begin -critical-section-end`) | Temporally exclusive tasks (`-temporal-exclusions-file`)

Topics

“Specify Analysis Options”

“Verify Multitasking Applications”

“Manually Model Tasks if main Contains Infinite Loop”

“Manually Model Scheduling of Tasks”

Introduced in R2016b

Critical section details (-critical-section-begin -critical-section-end)

Specify functions that begin and end critical sections

Description

When verifying multitasking code, Polyspace considers that a critical section lies between calls to a lock function and an unlock function.

```
lock();  
/* Critical section code */  
unlock();
```

Specify the lock and unlock function names for your critical sections (for instance, `lock()` and `unlock()` in above example).

Set Option

User interface: In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 1-126 for other options you must also enable.

Command line: Use the option `-critical-section-begin` and `-critical-section-end`. See “Command-Line Information” on page 1-128.

Why Use This Option

When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait till `my_task` calls the corresponding unlock function. Therefore, critical section operations in the other tasks cannot interrupt critical section operations in `my_task`.

For instance, the operation `var++` in `my_task1` and `my_task2` cannot interrupt each other.

```
int var;
```


```
void my_task1() {
    my_lock();
    var++;
    my_unlock();
}

void my_task2() {
    my_lock();
    var++;
    my_unlock();
}
```

Using your specifications, the verification checks if your placement of lock and unlock functions protects all shared variables from concurrent access. When determining values of those variables, the verification accounts for the fact that critical sections in different tasks do not interrupt each other.



Settings

No Default

Click  to add a field.

- In **Starting routine**, enter name of lock function.
- In **Ending routine**, enter name of unlock function.

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option, first select the option `Configure multitasking manually`.

Tips

- For function calls that begin and end critical sections, Polyspace ignores the function arguments.

For instance, Polyspace treats the two code sections below as the same critical section.

Starting routine: func_begin	Starting routine: func_begin
Ending routine: func_end	Ending routine: func_end
<pre>void my_task1() { my_lock(1); /* Critical section code */ my_unlock(1); }</pre>	<pre>void my_task2() { my_lock(2); /* Critical section code */ my_unlock(2); }</pre>

- The functions that begin and end critical sections must be functions. For instance, if you define a function-like macro:

```
#define init() num_locks++
```

You cannot use the macro `init()` to begin or end a critical section.

- When you use multiple critical sections, you can run into issues such as:
 - Deadlock: A sequence of calls to lock functions causes two tasks to block each other.
 - Double lock: A lock function is called twice in a task without an intermediate call to an unlock function.

Use Polyspace Bug Finder™ to detect such issues. Then, use Polyspace Code Prover™ to detect if your placement of lock and unlock functions actually protects all shared variables from concurrent access. For more information, see “Concurrency Defects” (Polyspace Bug Finder) in Polyspace Bug Finder documentation.

- When considering possible values of shared variables, the verification takes into account your specifications for critical sections.

However, if the shared variable is a pointer or array, the software uses the specifications only to determine if the variable is a shared protected global variable on page 9-2. For run-time error checking, the software does not take your specifications into account and considers that the variable can be concurrently accessed.

Command-Line Information

Parameter: `-critical-section-begin` | `-critical-section-end`

No Default

Value: `function1:cs1[, function2:cs2[, ...]]`

Example: `polyspace-code-prover-nodesktop -sources file_name -critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`

See Also

`Configure multitasking manually` | `Entry points (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)` | `Shared protected global variable` | `Shared unprotected global variable`

Topics

“Specify Analysis Options”

“Verify Multitasking Applications”

“Manually Protect Shared Variables from Concurrent Access”

Temporally exclusive tasks (-temporal-exclusions-file)

Specify entry point functions that cannot execute concurrently

Description

Specify entry point functions that cannot execute concurrently. The execution of the functions cannot overlap with each other.

Set Option

User interface: In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 1-130 for other options you must also enable.

Command line: Use the option `-temporal-exclusions-file`. See “Command-Line Information” on page 1-130.


Why Use This Option

Use this option to implement temporal exclusion in multitasking code.



The verification checks if specifying certain tasks as temporally exclusive protects all shared variables from concurrent access. When determining possible values of those shared variables, the verification accounts for the fact that temporally exclusive tasks do not interrupt each other.

Settings

No Default

Click  to add a field. In each field, enter a space-separated list of functions. Polyspace considers that the functions in the list cannot execute concurrently.

Enter the function names manually or choose from a list.

- Click  to add a field and enter the function names.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option, first select the option `Configure multitasking` manually.

Tips

When considering possible values of shared variables, the verification takes into account your specifications for temporally exclusive tasks.

However, if the shared variable is a pointer or array, the software uses the specifications only to determine if the variable is a shared protected global variable on page 9-2. For run-time error checking, the software does not take your specifications into account and considers that the variable can be concurrently accessed.

Command-Line Information

For the command-line option, create a temporal exclusions file in the following format:

- On each line, enter one group of temporally excluded tasks.
- Within a line, the tasks are separated by spaces.

Parameter: `-temporal-exclusions-file`

No Default

Value: Name of temporal exclusions file

Example: `polyspace-code-prover-nodesktop -sources file_name -temporal-exclusions-file "C:\exclusions_file.txt"`

See Also

`Configure multitasking manually` | `Entry points (-entry-points)` | `Critical section details (-critical-section-begin -critical-section-`

end) | Shared protected global variable | Shared unprotected global variable

Topics

“Specify Analysis Options”

“Verify Multitasking Applications”

“Manually Protect Shared Variables from Concurrent Access”

Check MISRA C:2004 (`-misra2`)

Check for violation of MISRA C:2004 rules

Description

Specify whether to check for violation of MISRA C:2004 rules. Each value of the option corresponds to a subset of rules to check.


Set Option

User interface: In your project configuration, the option is on the **Coding Rules & Code Metrics** node. See “Dependencies” on page 1-134 for other options that you must also enable.

Command line: Use the option `-misra2`. See “Command-Line Information” on page 1-134.

Why Use This Option

Use this option to specify the subset of MISRA C:2004 rules to check for.

After analysis, the **Results List** pane lists the coding rule violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a  symbol to the keyword or identifier relevant to the violation.

Settings

Default: `required-rules`

`required-rules`

Check required coding rules.

`all-rules`

Check required and advisory coding rules.


SQA-subset1



Check only a subset of MISRA C rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C:2004)”.

SQA-subset2

Check a subset of rules including SQA-subset1 and some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C:2004)”.

custom

Specify coding rules to check. Click  to create a coding rules file. After creating and saving the file, to reuse it for another project, do one of the following:

- Enter full path to the file in the space provided.
- Click . Click  to load the file.

Format of the custom file:

```
rule number off|on
```

Use # to enter comments in the file. For example:

```
10.5 off # rule 10.5: type conversion
17.2 on # rule 17.2: pointers
```

single-unit-rules

Check a subset of rules that apply only to single translation units. These rules are checked in the compilation phase of the analysis.

system-decidable-rules

Check rules in the single-unit-rules subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. These rules are checked in the compilation and linking phases of the analysis.

Dependencies

- This option is available only if you set `Source code language (-lang)` to C or C-CPP.

For projects with mixed C and C++ code, the MISRA C:2004 checker analyzes only `.c` files.

- If you set `Source code language (-lang)` to C-CPP, you can activate a C coding rule checker **and** a C++ coding rule checker. When you have both C and C++ coding rule checkers active, to avoid duplicate results, Polyspace does not produce the C coding rules found in the linking phase (such as MISRA C:2012 Rule 8.3).

Tips

- To reduce unproven results in Polyspace Code Prover:
 - 1 Find coding rule violations in `SQO-subset1`. Fix your code to address the violations and rerun verification.
 - 2 Find coding rule violations in `SQO-subset2`. Fix your code to address the violations and rerun verification.
- If you select the option `single-unit-rules` or `system-decidable-rules` and choose to detect coding rule violations only, the analysis can complete quicker than checking other rules. For more information, see “Coding Rule Subsets Checked Early in Analysis”.

Command-Line Information

Parameter: `-misra2`

Value: `required-rules | all-rules | SQO-subset1 | SQO-subset2 | single-unit-rules | system-decidable-rules | file`

Default: `required-rules`

Example: `polyspace-code-prover-nodesktop -sources file_name -misra2 all-rules`

See Also

Generate results for sources and `(-generate-results-for)`

Topics

“Specify Analysis Options”

“Set Up Coding Rules Checking”

“Polyspace MISRA C 2004 and MISRA AC AGC Checkers”

“Software Quality Objective Subsets (C:2004)”

Check MISRA AC AGC (`-misra-ac-agc`)

Check for violation of MISRA AC AGC rules

Description

Specify whether to check for violation of rules specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*. Each value of the option corresponds to a subset of rules to check.

Set Option

User interface: In your project configuration, the option is on the **Coding Rules & Code Metrics** node. See “Dependencies” on page 1-138 for other options that you must also enable.

Command line: Use the option `-misra-ac-agc`. See “Command-Line Information” on page 1-138.

Why Use This Option

Use this option to specify the subset of MISRA C:2004 AC AGC rules to check for.

After analysis, the **Results List** pane lists the coding rule violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▼ symbol to the keyword or identifier relevant to the violation.

Settings

Default: `OBL-rules`

`OBL-rules`

Check required coding rules.

`OBL-REC-rules`

Check required and recommended rules.

`all-rules`

Check required, recommended and readability-related rules.

`SQO-subset1`

Check a subset of rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (AC AGC)”.



`SQO-subset2`

Check a subset of rules including `SQO-subset1` and some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (AC AGC)”.

`custom`

Specify coding rules to check. Click  to create a coding rules file.

After creating and saving the file, to reuse it for another project, do one of the following:

- Enter full path to the file in the space provided.
- Click . Click  to load the file.

Format of the custom file:

```
rule number off|on
```

Use # to enter comments in the file. For example:

```
10.5 off # rule 10.5: type conversion
17.2 on # rule 17.2: pointers
```

`single-unit-rules`

Check a subset of rules that apply only to single translation units. These rules are checked in the compilation phase of the analysis.

This setting is not available from the drop-down list in the user interface. To choose this setting, enter the option `-misra-ac-agc single-unit-rules` in the field Other.

`system-decidable-rules`

Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that

apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. These rules are checked in the compilation and linking phases of the analysis.

This setting is not available from the drop-down list in the user interface. To choose this setting, enter the option `-misra-ac-agc system-decidable-rules` in the field `Other`.

Dependencies

- This option is available only if you set `Source code language (-lang)` to C or C-CPP.

For projects with mixed C and C++ code, the MISRA AC AGC checker analyzes only `.c` files.

- If you set `Source code language (-lang)` to C-CPP, you can activate a C coding rule checker **and** a C++ coding rule checker. When you have both C and C++ coding rule checkers active, to avoid duplicate results, Polyspace does not produce the C coding rules found in the linking phase (such as MISRA C:2012 Rule 8.3).

Tips

- To reduce unproven results in Polyspace Code Prover:
 - 1 Find coding rule violations in `SQO-subset1`. Fix your code to address the violations and rerun verification.
 - 2 Find coding rule violations in `SQO-subset2`. Fix your code to address the violations and rerun verification.
- If you select the option `single-unit-rules` or `system-decidable-rules` and choose to detect coding rule violations only, the analysis can complete quicker than checking other rules. For more information, see “Coding Rule Subsets Checked Early in Analysis”.

Command-Line Information

Parameter: `-misra-ac-agc`

Value: OBL-rules | OBL-REC-rules | all-rules | SQO-subset1 | SQO-subset2 | single-unit-rules | system-decidable-rules | *file*

Default: OBL-rules

Example: polyspace-code-prover-nodesktop -sources *file_name* -misra-ac-agc all-rules

See Also

Generate results for sources and (-generate-results-for)

Topics

“Specify Analysis Options”

“Set Up Coding Rules Checking”

“Polyspace MISRA C 2004 and MISRA AC AGC Checkers”

“MISRA C:2004 and MISRA AC AGC Coding Rules”

“Software Quality Objective Subsets (AC AGC)”

Check MISRA C:2012 (`-misra3`)

Check for violations of MISRA C:2012 rules and directives

Description

Specify whether to check for violations of MISRA C:2012 guidelines. Each value of the option corresponds to a subset of guidelines to check.

Set Option

User interface: In your project configuration, the option is on the **Coding Rules & Code Metrics** node. See “Dependencies” on page 1-142 for other options that you must also enable.

Command line: Use the option `-misra3`. See “Command-Line Information” on page 1-143.

Why Use This Option

Use this option to specify the subset of MISRA C:2012 rules to check for.

After analysis, the **Results List** pane lists the coding rule violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▼ symbol to the keyword or identifier relevant to the violation.

Settings


Default: `mandatory-required`

`mandatory-required`

Check for mandatory and required guidelines.

- **Mandatory guidelines:** Your code must comply with these guidelines.
- **Required guidelines:** You may deviate from these these guidelines. However, you must complete a formal deviation record, and your deviation must be authorized.

See Section 5.4 of the MISRA C:2012 guidelines. For an example of a deviation record, see Appendix I of the MISRA C:2012 guidelines.

Note To turn off some required guidelines, instead of mandatory-required select custom. To clear specific guidelines, click . In the **Comment** column, enter your rationale for disabling a guideline. For instance, you can enter the Deviation ID that refers to a deviation record for the guideline. The rationale appears in your generated report.

mandatory

Check for mandatory guidelines.

CERT-rules

Check for a subset of coding rules that corresponds to CERT-C rules.

See “CERT C Coding Standard and Polyspace Results” (Polyspace Bug Finder).

CERT-all

Check for a subset of coding rules that corresponds to CERT-C rules and recommendations.

See “CERT C Coding Standard and Polyspace Results” (Polyspace Bug Finder).

ISO-17961

Check for a subset of coding rules that corresponds to the ISO/IEC TS 17961 coding standard.

all

Check for mandatory, required, and advisory guidelines.


SQO-subset1



Check for only a subset of guidelines. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C:2012)”.

SQO-subset2

Check for the subset SQO-subset1, plus some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C:2012)”.

custom

Specify guidelines to check. Click  to create a coding rules file. Save the file. To reuse it for another project, do one of the following:

- Enter full path to the file in the space provided.
- Click . Click  to load the file.

Custom file format:

```
rule number off|on
```

Use # to enter comments in the file. For example:

```
10.5 off # rule 10.5: essential type model  
17.2 on # rule 17.2: functions
```

single-unit-rules

Check a subset of rules that apply only to single translation units. These rules are checked in the compilation phase of the analysis.

system-decidable-rules

Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. These rules are checked in the compilation and linking phases of the analysis.

Dependencies

- This option is available only if you set `Source code language (-lang)` to C or C-CPP.

For projects with mixed C and C++ code, the MISRA C:2012 checker analyzes only `.c` files.

- If you set `Source code language (-lang)` to C-CPP, you can activate a C coding rule checker **and** a C++ coding rule checker. When you have both C and C++ coding rule checkers active, to avoid duplicate results, Polyspace does not produce the C coding rules found in the linking phase (such as MISRA C:2012 Rule 8.3).

Tips

- To reduce unproven results in Polyspace Code Prover:
 - 1 Find coding rule violations in `SQO-subset1`. Fix your code to address the violations and rerun verification.
 - 2 Find coding rule violations in `SQO-subset2`. Fix your code to address the violations and rerun verification.
- If you select the option `single-unit-rules` or `system-decidable-rules` and choose to detect coding rule violations only, the analysis can complete quicker than checking other rules. For more information, see “Coding Rule Subsets Checked Early in Analysis”.
- Polyspace Code Prover does not support checking of the following:
 - MISRA C:2012 Directive 4.13 and 4.14
 - MISRA C:2012 Rule 21.13, 21.14, and 21.17 - 21.20
 - MISRA C:2012 Rule 22.1 - 22.4 and 22.6 - 22.10

For support of all MISRA C: 2012 rules including the security guidelines in Amendment 1, use Polyspace Bug Finder.

Command-Line Information

Parameter: `-misra3`

Value: `mandatory` | `mandatory-required` | `CERT-rules` | `CERT-all` | `ISO-17961` | `all` | `SQO-subset1` | `SQO-subset2` | `single-unit-rules` | `system-decidable-rules` | `file`

Default: `mandatory-required`

Example: `polyspace-code-prover-nodesktop -lang c -sources file_name -misra3 mandatory-required`

See Also

Generate results for sources and (`-generate-results-for`)

Topics

“Specify Analysis Options”

“Set Up Coding Rules Checking”
“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

Use generated code requirements (-misra3-agc-mode)

Check for violations of MISRA C:2012 rules and directives that apply to generated code

Description

Specify whether to use the MISRA C:2012 categories for automatically generated code. This option changes which rules are mandatory, required, or advisory.

Set Option

User interface: In your project configuration, the option is on the **Coding Rules & Code Metrics** node. See “Dependency” on page 1-147 for other options that you must also enable.

Command line: Use the option `-misra3-agc-mode`. See “Command-Line Information” on page 1-147.

Why Use This Option

Use this option to specify that you are checking for MISRA C:2012 rules in generated code. The option modifies the MISRA C:2012 subsets so that they are tailored for generated code.

Settings

Off (default)

Use the normal categories (mandatory, required, advisory) for MISRA C:2012 coding guideline checking.

On (default for analyses from Simulink)

Use the generated code categories (mandatory, required, advisory, readability) for MISRA C:2012 coding guideline checking.

For analyses started from the Simulink plug-in, this option is the default value.

Category changed to `Advisory`

These rules are changed to advisory:

- 5.3
- 7.1
- 8.4, 8.5, 8.14
- 10.1, 10.2, 10.3, 10.4, 10.6, 10.7, 10.8
- 14.4, 14.4
- 15.2, 15.3
- 16.1, 16.2, 16.3, 16.4, 16.5, 16.6, 16.7
- 20.8

Category changed to `Readability`

These guidelines are changed to readability:

- Dir 4.5
- 2.3, 2.4, 2.5, 2.6, 2.7
- 5.9
- 7.2, 7.3
- 9.2, 9.3, 9.5
- 11.9
- 13.3
- 14.2
- 15.7
- 17.5, 17.7, 17.8
- 18.5
- 20.5

Dependency

To use this option, first select the Check MISRA C:2012 (-misra3) option.

Command-Line Information

Parameter: -misra3-agc-mode

Default: Off

Example: polyspace-code-prover-nodesktop -sources *file_name* -misra3
all -misra3-agc-mode

See Also

Generate results for sources and (-generate-results-for) | Check MISRA C:2012 (-misra3)

Topics

“Specify Analysis Options”

“Set Up Coding Rules Checking”

“Polyspace MISRA C:2012 Checker”

Check custom rules (`-custom-rules`)

Follow naming conventions for identifiers

Description

Define naming conventions for identifiers and check your code against them.

Set Option

User interface: In your project configuration, the option is on the **Coding Rules & Code Metrics** node.

Command line: Use the option `-custom-rules`. See “Command-Line Information” on page 1-151.

Why Use This Option

Use this option to impose naming conventions on identifiers. Using a naming convention allows you to easily determine the nature of an identifier from its name. For instance, if you define a naming convention for structures, you can easily tell whether an identifier represents a structured variable or not.


After analysis, the **Results List** pane lists violations of the naming conventions. On the **Source** pane, for every violation, Polyspace assigns a ▼ symbol to the keyword or identifier relevant to the violation.

Settings

On

Polyspace matches identifiers in your code against text patterns you define. Define the text patterns in a custom coding rules file. To create a coding rules file,

- Use the custom rules wizard:

- 1 Click . The New File window opens.
- 2 From the drop-down list **Set the following state to all Custom C**, select **Off**. Click **Apply**.
- 3 For every custom rule you want to check:
 - a Select **On** .
 - b In the **Convention** column, enter the error message you want to display if the rule is violated.

For example, for rule 4.3, **All struct fields must follow the specified pattern**, you can enter All struct fields must begin with s_. This message appears on the **Result Details** pane if:

- You specify the **Pattern** as s_[A-Za-z0-9_]+.
- A structure field in your code does not begin with s_.

- c In the **Pattern** column, enter the text pattern.

For example, for rule 4.3, **All struct fields must follow the specified pattern**, you can enter s_[A-Za-z0-9_]+. Polyspace reports violation of rule 4.3 if a structure field does not begin with s_.

You can use Perl regular expressions to define patterns. For instance, you can use the following expressions.

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

For frequent patterns, you can use the following regular expressions:

- `(?!__)[a-z0-9_]+(?!__)`, matches a text pattern that does not start and end with two underscores.

```
int __text; //Does not match
int _text_; //Matches
```

- `[a-z0-9_]+_(u8|u16|u32|s8|s16|s32)`, matches a text pattern that ends with a specific suffix.

```
int _text_; //Does not match
int _text_s16; //Matches
int _text_s33; // Does not match
```

- `[a-z0-9_]+_(u8|u16|u32|s8|s16|s32)(_b3|_b8)?`, matches a text pattern that ends with a specific suffix and an optional second suffix.

```
int _text_s16; //Matches
int _text_s16_b8; //Matches
```

For a complete list of regular expressions, see Perl documentation.

- Manually edit an existing custom coding rules file:

- 1 Open the file with a text editor.
- 2 For every custom rule you want to check, enter the following information in adjacent lines.

- a Rule number, followed by `on`. For example:


```
4.3 on
```

- b The error message you want to display starting with `convention=`. For example:

```
convention=All struct fields must begin with s_
```

- c The text pattern starting with `pattern=`. For example:

```
pattern=s_[A-Za-z0-9_]
```

To use an existing coding rules file, enter the full path to the file in the field provided or use  in the New File window to navigate to the file location.

Off (default)

Polyspace does not check your code against custom naming conventions.

Command-Line Information

Parameter: -custom-rules

Value: Name of coding rules file

Default: Off

Example: polyspace-code-prover-nodesktop -sources *file_name* -custom-rules "C:\Rules\myrules.txt"

See Also

Topics

“Specify Analysis Options”

“Set Up Coding Rules Checking”

“Create Custom Coding Rules”

“Format of Custom Coding Rules File”

“Custom Coding Rules”

Effective boolean types (`-boolean-types`)

Specify data types that coding rule checker must treat as effectively Boolean

Description

Specify data types that the coding rule checker must treat as effectively Boolean. You can specify a data type only if you have defined it through a `typedef` statement in your source code.

Set Option

User interface: In your project configuration, the option is on the **Coding Rules & Code Metrics** node. See “Dependencies” on page 1-154 for other options that you must also enable.

Command line: Use the option `-boolean-types`. See “Command-Line Information” on page 1-154.

Why Use This Option

Use this option to allow Polyspace to check the following coding rules:

- MISRA C: 2004 and MISRA AC AGC

Rule Number	Rule Statement
12.6	Operands of logical operators, <code>&&</code> , <code> </code> , and <code>!</code> , should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to other operators.
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
15.4	A <code>switch</code> expression should not represent a value that is effectively Boolean.

- MISRA C: 2012

Rule Number	Rule Statement
10.1 on page 6-133	Operands shall not be of an inappropriate essential type
10.3 on page 6-142	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category
10.5 on page 6-146	The value of an expression should not be cast to an inappropriate essential type
14.4 on page 6-215	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.
16.7 on page 6-253	A switch-expression shall not have essentially Boolean type.

For example, in the following code, unless you specify `myBool` as effectively Boolean, Polyspace detects a violation of MISRA C: 2012 rule 14.4.


```
typedef int myBool;

void func1(void);
void func2(void);

void func(myBool flag) {
    if(flag)
        func1();
    else
        func2();
}
```

Settings

No Default

Click  to add a field. Enter a type name that you want Polyspace to treat as Boolean.

Dependencies

This option is available only if you select Check MISRA AC AGC (`-misra-ac-agc`), Check MISRA C:2004 (`-misra2`), or Check MISRA C:2012 (`-misra3`).

Command-Line Information

Parameter: `-boolean-types`

Value: `type1[, type2[, ...]]`

No Default

Example: `polyspace-code-prover-nodesktop -sources filename -misra2 required-rules -boolean-types boolean1_t,boolean2_t`

See Also

Check MISRA AC AGC (`-misra-ac-agc`) | Check MISRA C:2004 (`-misra2`) | Check MISRA C:2012 (`-misra3`)

Topics

“Set Up Coding Rules Checking”

“Specify Effective Boolean Types”

“MISRA C:2004 and MISRA AC AGC Coding Rules”

Allowed pragmas (-allowed-pragmas)

Specify pragma directives for which MISRA C:2004 rule 3.4 must not be applied

Description

Specify pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ rule 16-6-1 must not be applied.

Set Option

User interface: In your project configuration, the option is on the **Coding Rules & Code Metrics** node. See “Dependencies” on page 1-156 for other options that you must also enable.


Command line: Use the option `-allowed-pragmas`. See “Command-Line Information” on page 1-156.

Why Use This Option

MISRA C:2004/MISRA AC AGC rule 3.4 and MISRA C++ rule 16-6-1 require that all pragma directives are documented within the documentation of the compiler. If you list a pragma as documented using this analysis option, Polyspace does not flag use of the pragma as a violation of these rules.

Settings

No Default

Click  to add a field. Enter the pragma name that you want Polyspace to ignore during coding rule checking .

Dependencies

This option is enabled only if you select one of the following options:

- Check MISRA C:2004 (`-misra2`)
- Check MISRA AC AGC (`-misra-ac-agc`).
- Check MISRA C++ rules (`-misra-cpp`)

Command-Line Information

Parameter: `-allowed-pragmas`

Value: `pragma1[,pragma2[,...]]`

No Default

Example: `polyspace-code-prover-nodesktop -sources filename -misra-cpp
required-rules -allowed-pragmas pragma_01,pragma_02`

Example: `polyspace-code-prover-nodesktop -sources filename -misra2
required-rules -allowed-pragmas pragma_01,pragma_02`

See Also

Check MISRA C:2004 (`-misra2`) | Check MISRA AC AGC (`-misra-ac-agc`) |
Check MISRA C++ rules (`-misra-cpp`)

Topics

“Set Up Coding Rules Checking”

“MISRA C:2004 and MISRA AC AGC Coding Rules”

“MISRA C++ Coding Rules”

Calculate code metrics (-code-metrics)

Compute and display code complexity metrics

Description

Specify that Polyspace must compute and display code complexity metrics for your source code. The metrics include file metrics such as number of lines and function metrics such as cyclomatic complexity and estimated size of local variables.

For more information, see “Code Metrics”.

Set Option

User interface: In your project configuration, the option is on the **Coding Rules & Code Metrics** node.

Command line: Use the option `-code-metrics`. See “Command-Line Information” on page 1-158.

Why Use This Option

By default, Polyspace does not calculate code complexity metrics. If you want these metrics in your analysis results, before running analysis, set this option.

High values of code complexity metrics can lead to obscure code and increase chances of coding errors. Additionally, if you run a Code Prover verification on your source code, you might benefit from checking your code complexity metrics first. If a function is too complex, attempts to verify the function can lead to a lot of unproven code. For information on how to cap your code complexity metrics, see “Review Code Metrics”.

Settings

On

Polyspace computes and displays code complexity metrics on the **Results List** pane.

Off (default)

Polyspace does not compute complexity metrics.

Tips

If you want to compute only the code complexity metrics for your code:

- In Bug Finder, disable checking of defects. See `Find defects (-checkers)`.
- In Code Prover, run verification upto the Source Compliance Checking phase. See `Verification level (-to)`.

Command-Line Information

Parameter: `-code-metrics`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -code-metrics`

Check MISRA C++ rules (-misra-cpp)

Check for violations of MISRA C++ rules

Description

Specify whether to check for violation of MISRA C++ rules. Each value of the option corresponds to a subset of rules to check.

Set Option

User interface: In your project configuration, the option is on the **Coding Rules & Code Metrics** node. See “Dependency” on page 1-160 for other options that you must also enable.

Command line: Use the option `-misra-cpp`. See “Command-Line Information” on page 1-161.

Why Use This Option

Use this option to specify the subset of MISRA C++ rules to check for.

After analysis, the **Results List** pane lists the coding rule violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▼ symbol to the keyword or identifier relevant to the violation.

Settings

Default: `required-rules`

`required-rules`

Check required coding rules.

`all-rules`

Check required and advisory coding rules.

SQO-subset1

Check only a subset of MISRA C++ rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C++)”.



SQO-subset2

Check a subset of rules including SQO-subset1 and some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C++)”

custom

Specify coding rules to check. Click  to create a coding rules file.

After creating and saving the file, to reuse it for another project, do one of the following:

- Enter full path to the file in the space provided.
- Click . Click  to load the file.

Format of the custom file:

```
<rule number> off|on
```

Use # to enter comments in the file. For example:

```
9-5-1 off # rule 9-5-1: classes
15-0-2 on # rule 15-0-2: exception handling
```

Dependency

This option is available only if you set Source code language (-lang) to CPP or C-CPP.

For projects with mixed C and C++ code, the MISRA C++ checker analyzes only .cpp files.

Command-Line Information

Parameter: -misra-cpp

Value: required-rules | all-rules | SQ0-subset1 | SQ0-subset2 | *file*

Default: required-rules

Example: polyspace-code-prover-nodesktop -sources *file_name* -misra-cpp all-rules

See Also

Generate results for sources and (-generate-results-for)

Topics

“Specify Analysis Options”

“Set Up Coding Rules Checking”

“Polyspace MISRA C++ Checker”

“Software Quality Objective Subsets (C++)”

“MISRA C++ Coding Rules”

Check JSF C++ rules (`-jsf-coding-rules`)

Check for violations of JSF C++ rules

Description

Specify whether to check for violation of JSF C++ rules (JSF++:2005). Each value of the option corresponds to a subset of rules to check.


Set Option

User interface: In your project configuration, the option is on the **Coding Rules & Code Metrics** node. See “Dependency” on page 1-163 for other options that you must also enable.

Command line: Use the option `-jsf-coding-rules`. See “Command-Line Information” on page 1-164.

Why Use This Option

Use this option to specify the subset of JSF C++ rules to check for.

After analysis, the **Results List** pane lists the coding rule violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a  symbol to the keyword or identifier relevant to the violation.

Settings

Default: `shall-rules`

`shall-rules`

Check all **Shall** rules. **Shall** rules are mandatory requirements and require verification.

shall-will-rules

Check all **Shall** and **Will** rules. **Will** rules are intended to be mandatory requirements but do not require verification.



all-rules

Check all **Shall**, **Will**, and **Should** rules. **Should** rules are advisory rules.

custom

Specify coding rules to check. Click  to create a coding rules file.

After creating and saving the file, to reuse it for another project, do one of the following:

- Enter full path to the file in the space provided.
- Click . Click  to load the file.

Format of the custom file:

```
<rule number> off|on
```

Use # to enter comments in the file. For example:

```
67 off # rule 67: classes
202 on # rule 202: expressions
```

Tips

- If your project uses a setting other than `iso` for `Compiler (-compiler)`, some rules might not be completely checked. For example, AV Rule 8: “All code shall conform to ISO/IEC 14882:2002(E) standard C++.”

Dependency

This option is available only if you set `Source code language (-lang)` to `CPP` or `C-CPP`.

For projects with mixed C and C++ code, the JSF C++ checker analyzes only `.cpp` files.

Command-Line Information

Parameter: `-jsf-coding-rules`

Value: `shall-rules` | `shall-will-rules` | `all-rules` | *file*

Default: `shall-rules`

Example: `polyspace-code-prover-nodesktop -sources file_name -jsf-coding-rules all-rules`

See Also

Generate results for sources and (`-generate-results-for`)

Topics

“Specify Analysis Options”

“Set Up Coding Rules Checking”

“Polyspace JSF C++ Checker”

“JSF C++ Coding Rules”

Verify whole application

Stop verification if sources files are incomplete and do not contain a main function

Description

This option affects a Code Prover analysis only.

Specify that Polyspace verification must stop if a main function is not present in the source files.

If you select a Visual C++ setting for `Compiler` (`-compiler`), you can specify which function must be considered as main. See `Main entry point` (`-main`).

Set Option

User interface: In your project configuration, the option is on the **Code Prover Verification** node.

Command line: There is no corresponding command-line option. See “Command-Line Information” on page 1-166.

Settings

On

Polyspace verification stops if it does not find a main function in the source files.

Off (default)

Polyspace continues verification even when a main function is not present in the source files. If a main is not present, it generates a file `__polyspace_main.c` that contains a main function.

Command-Line Information

Unlike the user interface, by default, a verification from the command line stops if it does not find a `main` function in the source files. If you specify the option `-main-generator`, Polyspace generates a `main` if it cannot find one in the source files.

See Also

`Verify module or library (-main-generator)`

Topics

“Specify Analysis Options”

“Verify C Application Without main Function”

Verify module or library (`-main-generator`)

Generate a `main` function if source files are modules or libraries that do not contain a `main`

Description

This option affects a Code Prover analysis only.

Specify that Polyspace must generate a `main` function if it does not find one in the source files.

Set Option

User interface: In your project configuration, the option is on the **Code Prover Verification** node.

Command line: Use the option `-main-generator`. See “Command-Line Information” on page 1-169.

For the analogous option for model generated code, see `Verify model generated code (-main-generator)`.

Why Use This Option

Use this option if you are verifying a module or library. A Code Prover analysis requires a `main` function. When verifying a module or library, your code might not have a `main`.

When you use this option, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Settings

On (default)

Polyspace generates a main function if it does not find one in the source files. The generated main:

- 1 Initializes variables specified by `Variables` to initialize (`-main-generator-writes-variables`).
- 2 Before calling other functions, calls the functions specified by `Initialization functions` (`-functions-called-before-main`).
- 3 In all possible orders, calls the functions specified by `Functions to call` (`-main-generator-calls`).
- 4 (C++ only) Calls class methods specified by `Class` (`-class-analyzer`) and `Functions to call` within the specified classes (`-class-analyzer-calls`).

If you do not specify the function and variable options above, the generated main:

- Initializes all global variables except those declared with keywords `const` and `static`.
- In all possible orders, calls all functions that are not called anywhere in the source files. Polyspace considers that global variables can be written between two consecutive function calls. Therefore, in each called function, global variables initially have the full range of values allowed by their type.

Off

Polyspace stops if a main function is not present in the source files.

Tips

- If a `main` function is present in your source files, the verification uses that `main` function, irrespective of whether you enable or disable this option.

The option is relevant only if a `main` function is not present in your source files.

- If you specify multitasking options, the verification ignores your specifications for `main` generation. Instead, the verification introduces an empty `main` function.

For more information on the multitasking options, see “Verify Multitasking Applications”.

Command-Line Information

Parameter: -main-generator

Default: Off

Example: polyspace-code-prover-nodesktop -sources *file_name* -main-generator ...

See Also

Verify whole application | Variables to initialize (-main-generator-writes-variables) | Parameters (-variables-written-before-loop) | Inputs (-variables-written-in-loop) | Initialization functions (-functions-called-before-loop) | Step functions (-functions-called-in-loop) | Termination functions (-functions-called-after-loop)

Topics

“Specify Analysis Options”

“Verify C Application Without main Function”

“Configure Advanced Polyspace Analysis Options”

“Main Generation for Model Verification”

Main entry point (`-main`)

Specify a Microsoft Visual C++ extensions of `main`

Description

This option affects a Code Prover analysis only.

Specify the function that you want to use as `main`. If the function does not exist, the verification stops with an error message. Use this option to specify Microsoft Visual C++ extensions of `main`.

Set Option

User interface: In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-171 for other options that you must also enable.

Command line: Use the option `-main`. See “Command-Line Information” on page 1-171.

Settings

Default: `_tmain`

`_tmain`

Use `_tmain` as entry point to your code.

`wmain`

Use `wmain` as entry point to your code.

`_tWinMain`

Use `_tWinMain` as entry point to your code.

`wWinMain`

Use `wWinMain` as entry point to your code.

WinMain

Use WinMain as entry point to your code.

DllMain

Use DllMain as entry point to your code.

Dependencies

This option is enabled only if you:

- Set Source code language (-lang) to CPP.
- Set **Target operation system** (-target) to Visual.
- Select **Verify whole application**

Command-Line Information

Parameter: -main

Value: _tmain | wmain | _tWinMain | wWinMain | WinMain | DllMain

Example: polyspace-code-prover-nodesktop -sources *file_name* -OS-target visual -main _tmain

See Also

Verify module or library (-main-generator)

Topics

“Specify Analysis Options”

Variables to initialize (`-main-generator-writes-variables`)

Specify global variables that you want the generated `main` to initialize

Description

This option affects a Code Prover analysis only.

Specify global variables that you want the generated `main` to initialize. Polyspace considers these variables to have any value allowed by their type.

Set Option

User interface: In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-173 for other options that you must also enable.

Command line: Use the option `-main-generator-writes-variables`. See “Command-Line Information” on page 1-173.

Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option to specify which global variables the generated `main` must initialize.

Settings

Default:

- C code — `public`
- C++ Code — `uninit`

uninit

C++ Only

The generated main only initializes global variables that you have not initialized during declaration.

none

The generated main does not initialize global variables.


public

The generated main initializes all global variables except those declared with keywords `static` and `const`.

all

The generated main initializes all global variables except those declared with keyword `const`.

custom

The generated main only initializes global variables that you specify. Click  to add a field. Enter a global variable name.

Dependencies

You can use this option only if the following are true:

- Your code does not contain a main function.
- `Verify module or library (-main-generator)` is selected.

Command-Line Information

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

Value: `uninit` | `none` | `public` | `all` | `custom=variable1[,variable2[,...]]`

Default: (C) `public` | (C++) `uninit`

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -main-generator-writes-variables all`

See Also

Verify module or library (-main-generator) | Initialization functions (-functions-called-before-main) | Functions to call (-main-generator-calls) | Class (-class-analyzer) | Functions to call within the specified classes (-class-analyzer-calls) | Analyze class contents only (-class-only)

Topics

“Specify Analysis Options”

“Verify C Application Without main Function”

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

Specify functions that you want the generated `main` to call ahead of other functions

Description

This option affects a Code Prover analysis only.

Specify functions that you want the generated `main` to call ahead of other functions.

Set Option

User interface: In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-176 for other options that you must also enable.

Command line: Use the option `-functions-called-before-main`. See “Command-Line Information” on page 1-177.

Why Use This Option



If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option along with the option `Functions to call (-main-generator-calls)` to specify which functions the generated `main` must call. Unless a function is called directly or indirectly from `main`, the software does not analyze the function.

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

If the function or method is not overloaded, specify the function name. Otherwise, specify the function prototype with arguments. For instance, in the following code, you must specify the prototypes `func(int)` and `func(double)`.

```
int func(int x) {  
    return(x * 2);  
}  
double func(double x) {  
    return(x * 2);  
}
```

For C++, if the function is:

- A class method: The generated `main` calls the class constructor before calling this function.
- Not a class method: The generated `main` calls this function before calling class methods.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::init(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::init()`.

Dependencies

This option is enabled only if you select **Verify module or library** under **Code Prover Verification** and your code does not contain a `main` function.

Tips

Although these functions are called ahead of other functions, they can be called in arbitrary order. If you want to call your initialization functions in a specific order, manually write a `main` function to call them.

Command-Line Information

Parameter: `-functions-called-before-main`

Value: `function1[, function2[, ...]]`

No Default

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -functions-called-before-main myfunc`

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -functions-called-before-main myClass::init(int)`

See Also

Verify module or library (-main-generator) | Variables to initialize (-main-generator-writes-variables) | Functions to call (-main-generator-calls) | Class (-class-analyzer) | Functions to call within the specified classes (-class-analyzer-calls) | Analyze class contents only (-class-only)

Topics

“Specify Analysis Options”

“Verify C Application Without main Function”

Functions to call (`-main-generator-calls`)

Specify functions that you want the generated `main` to call after the initialization functions

Description

This option affects a Code Prover analysis only.

Specify functions that you want the generated `main` to call. The `main` calls these functions after the ones you specify through the option `Initialization functions (-functions-called-before-main)`.

Set Option

User interface: In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-179 for other options that you must also enable.

Command line: Use the option `-main-generator-calls`. See “Command-Line Information” on page 1-180.

Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option along with the option `Initialization functions (-functions-called-before-main)` to specify which functions the generated `main` must call. Unless a function is called directly or indirectly from `main`, the software does not analyze the function.

Settings

Default: `unused`

none

The generated main does not call any function.

unused

The generated main calls only those functions that are not called in the source code. It does not call inlined functions.



all

The generated main calls all functions except inlined ones.

custom

The generated main calls functions that you specify.

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

Dependencies

This option is available only if you select `Verify module or library` (-main-generator).

Tips

- Select `unused` when you use **Code Prover Verification > Verify files independently**.
- If you want the generated main to call an inlined function, select `custom` and specify the name of the function.
- To verify a multitasking application without a main, select `none`.

- The generated main can call the functions in arbitrary order. If you want to call your functions in a specific order, manually write a main function to call them.

Command-Line Information

Parameter: `-main-generator-calls`

Value: `none` | `unused` | `all` | `custom=function1[,function2[,...]]`

Default: `unused`

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -main-generator-calls all`

See Also

Verify module or library (`-main-generator`) | Variables to initialize (`-main-generator-writes-variables`) | Initialization functions (`-functions-called-before-main`) | Class (`-class-analyzer`) | Functions to call within the specified classes (`-class-analyzer-calls`) | Analyze class contents only (`-class-only`)

Topics

“Specify Analysis Options”

“Verify C Application Without main Function”

Verify files independently (`-unit-by-unit`)

Verify each source file independently of other source files

Description

This option affects a Code Prover analysis only.

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

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

Set Option

User interface: In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-182 for other options that you must also enable.

Command line: Use the option `-unit-by-unit`. See “Command-Line Information” on page 1-182.

Why Use This Option

There are many reasons you might want to verify each source file independently of other files.

For instance, if verification of a project takes very long, you can perform a file by file verification to identify which file is slowing the verification.

Settings

On

Polyspace creates a separate verification job for each source file.

Off (default)

Polyspace creates a single verification job for all source files in a module.

Dependencies

This option is enabled only if you select `Verify module or library` (`-main-generator`).

Tips

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

To generate a single report for all the files:

- 1 Open the results for one file.
- 2 Select **Reporting > Run Report**. Before generating the report, select the option **Generate a single report including all unit results**.

Command-Line Information

Parameter: `-unit-by-unit`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -unit-by-unit`

See Also

Common source files (-unit-by-unit-common-source)

Topics

“Specify Analysis Options”

“Run File-by-File Local Verification”

“Run File-by-File Remote Verification”

“Multiple File Error in File by File Verification”

Common source files (`-unit-by-unit-common-source`)

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

Description

This option affects a Code Prover analysis only.

For a file by file verification, specify files that you want to include with each source file verification. These files are compiled once, and then linked to each verification.

Set Option

User interface: In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-185 for other options that you must also enable.

Command line: Use the option `-unit-by-unit-common-source`. See “Command-Line Information” on page 1-185.



Why Use This Option

There are many reasons you might want to verify each source file independently of other files. For instance, if verification of a project takes very long, you can perform a file by file verification to identify which file is slowing the verification.

If you perform a file by file verification, some of your files might be missing information present in the other files. Place the missing information in a common file and use this option to specify the file for verification. For instance, if multiple source files call the same function, use this option to specify a file that contains the function definition or a function stub. Otherwise, Polyspace uses its own stubs for functions that are called but not defined in the source files. The assumptions behind the Polyspace stubs can be broader than what you want, leading to orange checks.

Settings

No Default

Click  to add a field. Enter the full path to a file. Otherwise, use the  button to navigate to the file location.

Dependencies

This option is enabled only if you select `Verify files independently (-unit-by-unit)`.

Command-Line Information

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

Value: `file1[,file2[,...]]`

No Default

Example: `polyspace-code-prover-nodesktop -sources file_name -unit-by-unit -unit-by-unit-common-source definitions.c`

See Also

`Verify files independently (-unit-by-unit)`

Topics

“Specify Analysis Options”

“Run File-by-File Local Verification”

“Run File-by-File Remote Verification”

Verify model generated code (`-main-generator`)

Specify that a `main` function must be generated if it is not present in source files

Description

Specify that Polyspace must generate a `main` function if it does not find one in the source files.

Set Option

User interface: In your project configuration, the option is on the **Code Prover Verification** node.

Command line: Use the option `-main-generator`. See “Command-Line Information” on page 1-187.

Settings

This option is always enabled for code generated from models.

Polyspace generates a `main` function for the analysis. The generated `main` contains cyclic code that executes in a loop. The loop can run an unspecified number of times.

The `main` performs the following functions before the loop begins:

- Initializes variables specified by `Parameters` (`-variables-written-before-loop`).
- Calls the functions specified by `Initialization functions` (`-functions-called-before-loop`).

The `main` then performs the following functions in the loop:

- Calls the functions specified by `Step functions` (`-functions-called-in-loop`).

- Writes to variables specified by `Inputs` (-variables-written-in-loop).

Finally, the main calls the functions specified by `Termination` functions (-functions-called-after-loop).

Command-Line Information

Parameter: -main-generator

Default: On

See Also

`Parameters` (-variables-written-before-loop) | `Inputs` (-variables-written-in-loop) | `Initialization` functions (-functions-called-before-loop) | `Step` functions (-functions-called-in-loop) | `Termination` functions (-functions-called-after-loop)

Topics

“Specify Analysis Options”

“Configure Advanced Polyspace Analysis Options”

“Recommended Polyspace options for Verifying Generated Code”

“Main Generation for Model Verification”

Parameters (`-variables-written-before-loop`)

Specify variables that the generated `main` must initialize before the cyclic code loop

Description

This option is available only for model-generated code.

Specify variables that the generated `main` must initialize before the cyclic code loop begins. Before the loop begins, Polyspace considers these variables to have any value allowed by their type.

Set Option

User interface: In your project configuration, the option is available on the **Code Prover Verification** node.

Command line: Use the option `-variables-written-before-loop`. See “Command-Line Information” on page 1-189.

Settings

Default: none


none

The generated `main` does not initialize variables.

all

The generated `main` initializes all variables except those declared with keyword `const`.

custom

The generated main only initializes variables that you specify. Click  to add a field. Enter variable name. For C++ class members, use the syntax `className::variableName`.

Command-Line Information

Parameter: `-variables-written-before-loop`

Value: `none | all | custom=variable1[,variable2[,...]]`

Default: `public`

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -variables-written-before-loop all`

See Also

Inputs (`-variables-written-in-loop`) | Initialization functions (`-functions-called-before-loop`) | Step functions (`-functions-called-in-loop`) | Termination functions (`-functions-called-after-loop`)

Topics

“Specify Analysis Options”

“Configure Advanced Polyspace Analysis Options”

“Recommended Polyspace options for Verifying Generated Code”

“Main Generation for Model Verification”

Inputs (`-variables-written-in-loop`)

Specify variables that the generated `main` must initialize in the cyclic code loop

Description

This option is available only for model-generated code.

Specify variables that the generated `main` must initialize at the beginning of every iteration of the cyclic code loop. At the beginning of every loop iteration, Polyspace considers these variables to have anyvalue allowed by their type.

Set Option

User interface: In your project configuration, the option is available on the **Code Prover Verification** node.

Command line: Use the option `-variables-written-in-loop`. See “Command-Line Information” on page 1-191.

Settings

Default: none


none

The generated `main` does not initialize variables.

all

The generated `main` initializes all variables except those declared with keyword `const`.

custom

The generated `main` only initializes variables that you specify. Click  to add a field. Enter variable name. For C++ class members, use the syntax `className::variableName`.

Command-Line Information

Parameter: -variables-written-in-loop

Value: none | all | custom=*variable1*[,*variable2*[,...]]

Default: none

Example: polyspace-code-prover-nodesktop -sources *file_name* -main-generator -variables-written-in-loop all

See Also

Parameters (-variables-written-before-loop) | Initialization functions (-functions-called-before-loop) | Step functions (-functions-called-in-loop) | Termination functions (-functions-called-after-loop)

Topics

“Specify Analysis Options”

“Configure Advanced Polyspace Analysis Options”

“Recommended Polyspace options for Verifying Generated Code”

“Main Generation for Model Verification”

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

Specify functions that the generated `main` must call before the cyclic code loop

Description

This option is available only for model-generated code.

Specify functions that the generated `main` must call before the cyclic code begins.

Set Option

User interface: In your project configuration, the option is available on the **Code Prover Verification** node.

Command line: Use the option `-functions-called-before-loop`. See “Command-Line Information” on page 1-192.

Settings

No Default

Click  to add a field. Enter function name.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::init(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::init()`.

Command-Line Information

Parameter: `-functions-called-before-loop`

No Default

Value: *function1[,function2[,...]]*

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -functions-called-before-loop myfunc`

See Also

Parameters (-variables-written-before-loop) | Inputs (-variables-written-in-loop) | Step functions (-functions-called-in-loop) | Termination functions (-functions-called-after-loop)

Topics

“Specify Analysis Options”

“Configure Advanced Polyspace Analysis Options”

“Recommended Polyspace options for Verifying Generated Code”

“Main Generation for Model Verification”

Step functions (**-functions-called-in-loop**)

Specify functions that the generated `main` must call in the cyclic code loop

Description

This option is available only for model-generated code.

Specify functions that the generated `main` must call in each cycle of the cyclic code.

Set Option

User interface: In your project configuration, the option is available on the **Code Prover Verification** node.

Command line: Use the option `-functions-called-in-loop`. See “Command-Line Information” on page 1-195.

Settings

Default: none


none

The generated `main` does not call functions in the cyclic code.

all

The generated `main` calls all functions except inlined ones. If you specify certain functions for the options **Initialization functions** or **Termination functions**, the generated `main` does not call those functions in the cyclic code.

custom

The generated `main` calls functions that you specify. Click  to add a field. Enter function name.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`.

If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

Tips

If you have specified a function for the option **Initialization functions** or **Termination functions**, to call it inside the cyclic code, use `custom` and specify the function name.

Command-Line Information

Parameter: `-functions-called-in-loop`

Value: `none` | `all` | `custom=function1[,function2[,...]]`

Default: `none`

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -functions-called-in-loop all`

See Also

`Parameters (-variables-written-before-loop)` | `Inputs (-variables-written-in-loop)` | `Initialization functions (-functions-called-before-loop)` | `Termination functions (-functions-called-after-loop)`

Topics

“Specify Analysis Options”

“Configure Advanced Polyspace Analysis Options”

“Recommended Polyspace options for Verifying Generated Code”

“Main Generation for Model Verification”

Termination functions (`-functions-called-after-loop`)

Specify functions that the generated `main` must call after the cyclic code loop

Description

This option is available only for model-generated code.

Specify functions that the generated `main` must call after the cyclic code ends.

Set Option

User interface: In your project configuration, the option is available on the **Code Prover Verification** node.

Command line: Use the option `-functions-called-after-loop`. See “Command-Line Information” on page 1-197.

Settings

No Default

Click  to add a field. Enter function name.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

Tips

- If you specify a function for the option **Initialization functions**, you cannot specify it for **Termination functions**.

Command-Line Information

Parameter: `-functions-called-after-loop`

No Default

Value: `function1[, function2[, ...]]`

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -functions-called-after-loop myfunc`

See Also

Parameters (-variables-written-before-loop) | Inputs (-variables-written-in-loop) | Initialization functions (-functions-called-before-loop) | Step functions (-functions-called-in-loop)

Topics

“Specify Analysis Options”

“Configure Advanced Polyspace Analysis Options”

“Recommended Polyspace options for Verifying Generated Code”

“Main Generation for Model Verification”

Class (`-class-analyzer`)

Specify classes that you want to verify

Description

This option affects a Code Prover analysis only.

Specify classes that Polyspace uses to generate a main.

Set Option

User interface: In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-199 for other options that you must also enable.

Command line: Use the option `-class-analyzer`. See “Command-Line Information” on page 1-199.

Why Use This Option

If you are verifying a module or library, Code Prover generates a main function if one does not exist. If a main exists, the analysis uses the existing main.

Use this option and the option `Functions to call` within the specified classes (`-class-analyzer-calls`) to specify the class methods that the generated main must call. Unless a class method is called directly or indirectly from main, the software does not analyze the method.

Settings

Default: `all`

all

Polyspace can use all classes to generate a main. The generated main calls methods that you specify using **Functions to call within the specified classes**.

none

The generated main cannot call any class method.

custom

Polyspace can use classes that you specify to generate a main. The generated main calls methods from classes that you specify using **Functions to call within the specified classes**.

Dependencies

You can use this option only if all of the following are true:

- Your code does not contain a main function.
- Source code language (-lang) is set to CPP.
- Verify module or library (-main-generator) is selected.

Tips

If you select none for this option, Polyspace will not verify class methods that you do not call explicitly in your code.

Command-Line Information

Parameter: -class-analyzer

Value: all | none | custom=*class1*[,*class2*,...]

Default: all

Example: polyspace-code-prover-nodesktop -sources *file_name* -main-generator -class-analyzer custom=myClass1,myClass2

See Also

Verify module or library (-main-generator) | Functions to call within the specified classes (-class-analyzer-calls) | Analyze class contents

only (-class-only) | Skip member initialization check (-no-constructors-init-check)

Topics

“Specify Analysis Options”

“Verify C++ Classes”

Functions to call within the specified classes (-class-analyzer-calls)

Specify class methods that you want to verify

Description

This option affects a Code Prover analysis only.

Specify class methods that Polyspace uses to generate a `main`. The generated `main` can call static, public and protected methods in classes that you specify using the **Class** option.

Set Option

User interface: In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-203 for other options that you must also enable.

Command line: Use the option `-class-analyzer-calls`. See “Command-Line Information” on page 1-203.

Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option and the option `Class` (`-class-analyzer`) to specify the class methods that the generated `main` must call. Unless a class method is called directly or indirectly from `main`, the software does not analyze the method.

Settings

Default: unused

`all`

The generated `main` calls all public and protected methods. It does not call methods inherited from a parent class.

`all-public`

The generated `main` calls all public methods. It does not call methods inherited from a parent class.

`inherited-all`

The generated `main` calls all public and protected methods including those inherited from a parent class.

`inherited-all-public`

The generated `main` calls all public methods including those inherited from a parent class.

`unused`

The generated `main` calls public and protected methods that are not called in the code.

`unused-public`

The generated `main` calls public methods that are not called in the code. It does not call methods inherited from a parent class.

`inherited-unused`

The generated `main` calls public and protected methods that are not called in the code including those inherited from a parent class.



`inherited-unused-public`

The generated `main` calls public methods that are not called in the code including those inherited from a parent class.

`custom`

The generated `main` calls the methods that you specify.

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`.

If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

Dependencies

You can use this option only if:

- Source code language (-lang) is set to CPP.
- Verify module or library (-main-generator) is selected.

Command-Line Information

Parameter: -class-analyzer-calls

Value: all | all-public | inherited-all | inherited-all-public | unused | unused-public | inherited-unused | inherited-unused-public | custom=*method1*[,*method2*,...]

Default: unused

Example: polyspace-code-prover-nodesktop -sources *file_name* -main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public

See Also

Verify module or library (-main-generator) | Class (-class-analyzer) | Analyze class contents only (-class-only) | Skip member initialization check (-no-constructors-init-check)

Topics

“Specify Analysis Options”

“Verify C++ Classes”

Analyze class contents only (`-class-only`)

Do not analyze code other than class methods

Description

This option affects a Code Prover analysis only.

Specify that Polyspace must verify only methods of classes that you specify using the option `Class (-class-analyzer)`.

Set Option

User interface: In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-205 for other options that you must also enable.

Command line: Use the option `-class-only`. See “Command-Line Information” on page 1-205.

Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use the following options to specify the class methods that the generated `main` must call:

- `Class (-class-analyzer)`
- Functions to call within the specified classes (`-class-analyzer-calls`)

Unless a class method is called directly or indirectly from `main`, the software does not analyze the method. Use this option to specify that only the class methods must be analyzed and not other functions.

Using these three options, you can check your classes for robustness against run-time errors.

Settings

On

Polyspace verifies the class methods only. It stubs functions out of class scope even if the functions are defined in your code.

Off (default)

Polyspace verifies functions out of class scope in addition to class methods.

Dependencies

You can use this option only if all of the following are true:

- Your code does not contain a main function.
- Source code language (`-lang`) is set to CPP.
- Verify module or library (`-main-generator`) is selected.

If you select this option, you must specify the classes using the `theClass (-class-analyzer)` option.

Tips

Use this option:

- For robustness verification of class methods. Unless you use this option, Polyspace verifies methods that you call in your code only for your input combinations.
- In case of scaling.

Command-Line Information

Parameter: `-class-only`

Default: Off

See Also

Verify module or library (`-main-generator`) | Class (`-class-analyzer`) | Functions to call within the specified classes (`-class-analyzer-`

calls) | Skip member initialization check (-no-constructors-init-check)

Topics

“Specify Analysis Options”

“Verify C++ Classes”

Skip member initialization check (`-no-constructors-init-check`)

Do not check if class constructor initializes class members

Description

This option affects a Code Prover analysis only.

Specify that Polyspace must not check whether each class constructor initializes all class members.

Set Option

User interface: In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-208 for other options that you must also enable.

Command line: Use the option `-no-constructors-init-check`. See “Command-Line Information” on page 1-208.

Why Use This Option

Use this option to disable checks for initialization of class members in constructors.

Settings

On

Polyspace does not check whether each class constructor initializes all class members.

Off (default)

Polyspace checks whether each class constructor initializes all class members. It uses the functions `check_NIV()` and `check_NIP()` in the generated `main` to perform these checks. It checks for initialization of:

- Integer types such as `int`, `char` and `enum`, both signed or unsigned.
- Floating-point types such as `float` and `double`.
- Pointers.

Dependencies

You can use this option only if all of the following are true:

- Your code does not contain a main function.
- Source code language (`-lang`) is set to `CPP`.
- `Verify module or library (-main-generator)` is selected.

If you select this option, you must specify the classes using the `theClass (-class-analyzer)` option.

Command-Line Information

Parameter: `-no-constructors-init-check`

Default: Off

See Also

`Verify module or library (-main-generator) | Class (-class-analyzer)`

Topics

“Specify Analysis Options”

Respect types in fields (`-respect-types-in-fields`)

Do not cast nonpointer fields of a structure to pointers

Description

This option affects a Code Prover analysis only.

Specify that structure fields not declared initially as pointers will not be cast to pointers later.

Set Option

User interface: In your project configuration, the option is available on the **Verification Assumptions** node.

Command line: Use the option `-respect-types-in-fields`. See “Command-Line Information” on page 1-210.

Why Use This Option

Use this option to identify and forbid casts from nonpointer structure fields to pointers.

Settings

On

The verification assumes that structure fields not declared initially as pointers will not be cast to pointers later.

Code with option off	Code with option on
<pre> struct { unsigned int x1; unsigned int x2; } S; void funct(void) { int var, *tmp; S.x1 = &var; tmp = (int*)S.x1; *tmp = 1; assert(var==1); } </pre> <p>In this example, the fields of <code>S</code> are declared as integers but <code>S.x1</code> is cast to a pointer. With the option turned off, Polyspace allows the cast.</p>	<pre> struct { unsigned int x1; unsigned int x2; } S; void funct(void) { int var, *tmp; S.x1 = &var; tmp = (int*)S.x1; *tmp = 1; assert(var==1); } </pre> <p>In this example, the fields of <code>S</code> are declared as integers but <code>S.x1</code> is cast to a pointer. With the option turned on, Polyspace ignores the cast. Therefore, it ignores the initialization of <code>var</code> through the pointer <code>(int*)S.x1</code> and produces a red Non-initialized local variable error when <code>var</code> is read.</p>

Off (default)

The verification assumes that structure fields can be cast to pointers even when they are not declared as pointers.

Command-Line Information

Parameter: `-respect-types-in-fields`

Default: Off

See Also

Respect types in global variables (`-respect-types-in-globals`) | Non-initialized local variable

Topics

“Specify Analysis Options”

Respect types in global variables (`-respect-types-in-globals`)

Do not cast nonpointer global variables to pointers

Description

This option affects a Code Prover analysis only.

Specify that global variables not declared initially as pointers will not be cast to pointers later.

Set Option

User interface: In your project configuration, the option is available on the **Verification Assumptions** node.

Command line: Use the option `-respect-types-in-globals`. See “Command-Line Information” on page 1-212.

Why Use This Option

Use this option to identify and forbid casts from nonpointer global variables to pointers.

Settings

On

The verification assumes that global variables not declared initially as pointers will not be cast to pointers later.

Off (default)

The verification assumes that global variables can be cast to pointers even when they are not declared as pointers.

Tips

If you select this option, the number of checks in your code can change. You can use this option and the change in results to identify cases where you cast nonpointer variables to pointers.

For instance, in the following example, when you select the option, the results have one less orange check and one more red check.

Code with option off	Code with option on
<pre data-bbox="238 565 783 765">int global; void main(void) { int local; global = (int)&local; *(int*)global = 5; assert(local==5); }</pre> <p data-bbox="238 795 783 894">In this example, <code>global</code> is declared as an <code>int</code> variable but cast to a pointer. With the option turned off, Polyspace allows the cast.</p>	<pre data-bbox="788 565 1332 765">int global; void main(void) { int local; global = (int)&local; *(int*)global = 5; assert(local==5); }</pre> <p data-bbox="788 795 1332 1060">In this example, <code>global</code> is declared as an <code>int</code> variable but cast to a pointer. With the option turned on, Polyspace ignores the cast. Therefore, it ignores the initialization of <code>local</code> through the pointer <code>(int*)global</code> and produces a red Non-initialized local variable error when <code>local</code> is read.</p>

Command-Line Information

Parameter: `-respect-types-in-globals`

Default: Off

See Also

Respect types in fields (`-respect-types-in-fields`) | Non-initialized local variable

Topics

“Specify Analysis Options”

Consider environment pointers as unsafe (`-stubbed-pointers-are-unsafe`)

Specify that environment pointers can be unsafe to dereference unless constrained otherwise

Description

This option affects a Code Prover analysis only.

Specify that the verification must consider environment pointers as unsafe unless otherwise constrained. Environment pointers are pointers that can be assigned values outside your code.

Environment pointers include:

- Global or `extern` pointers.
- Pointers returned from stubbed functions.

A function is stubbed if your code does not contain the function definition or you override a function definition by using the option `Functions to stub` (`-functions-to-stub`).

- Pointer parameters of functions whose calls are generated by the software.

A function call is generated if you verify a module or library and the module or library does not have an explicit call to the function. You can also force a function call to be generated with the option `Functions to call` (`-main-generator-calls`).

Set Option

User interface: In your project configuration, the option is available on the **Verification Assumptions** node.

Command line: Use the option `-stubbed-pointers-are-unsafe`. See “Command-Line Information” on page 1-216.

Why Use This Option

Use this option so that the verification makes more conservative assumptions about pointers from external sources.

If you specify this option, the verification considers that environment pointers can have a NULL value. If you read an environment pointer without checking for NULL, the **Illegally dereferenced pointer** check shows a potential error in orange. The message associated with the orange check shows the pointer can be NULL.

Settings

On

The verification considers that environment pointers can have a NULL value.

Off (default)

The verification considers that environment pointers:

- Cannot have a NULL value.
- Points within allowed bounds.

Tips

- Enable this option during the integration phase. In this phase, you provide complete code for verification. Even if an orange check originates from external sources, you are likely to place protections against unsafe pointers from such sources. For instance, if you obtain a pointer from an unknown source, you check the pointer for NULL value.

Disable this option during the unit testing phase. In this phase, you focus on errors originating from your unit.

- If you enable this option, the number of orange checks in your code might increase.

Environment Pointers Safe	Environment Pointers Unsafe
<p>The Illegally dereferenced pointer check is green. The verification assumes that <code>env_ptr</code> is not <code>NULL</code> and any dereference is within allowed bounds. The verification assumes that the result of the dereference is full range. For instance, in this case, the return value has the full range of type <code>int</code>.</p> <pre>int func (int *env_ptr) { return *env_ptr; }</pre>	<p>The Illegally dereferenced pointer check is orange. The verification assumes that <code>env_ptr</code> can be <code>NULL</code>.</p> <pre>int func (int *env_ptr) { return *env_ptr; }</pre>

If you enable this option, the number of gray checks might decrease.

Environment Pointers Safe	Environment Pointers Unsafe
<p>The verification assumes that <code>env_ptr</code> is not <code>NULL</code>. The <code>if</code> condition is always true and the <code>else</code> block is unreachable.</p> <pre>#include <stdlib.h> int func (int *env_ptr) { if(env_ptr!=NULL) return *env_ptr; else return 0; }</pre>	<p>The verification assumes that <code>env_ptr</code> can be <code>NULL</code>. The <code>if</code> condition is not always true and the <code>else</code> block can be reachable.</p> <pre>#include <stdlib.h> int func (int *env_ptr) { if(env_ptr!=NULL) return *env_ptr; else return 0; }</pre>

- Instead of considering all environment pointers as safe or unsafe, you can individually constrain some of the environment pointers. See the description of **Initialize Pointer** in “Constraints”.

When you individually constrain a pointer, you first specify an **Init Mode**, and then specify through the **Initialize Pointer** option whether the pointer is `Null`, `Not Null`, or `Maybe Null`. Depending on the **Init Mode**, you can either override the global specification for all environment pointers or not.

- If you set the **Init Mode** of the pointer to `INIT` or `PERMANENT`, your selection for **Initialize Pointer** overrides your specification for this option. For instance, if you specify `Not Null` for an environment pointer `ptr`, the verification assumes that

`ptr` is not `NULL` even if you specify that environment pointers must be considered unsafe.

- If you set the **Init Mode** to `MAIN GENERATOR`, the verification uses your specification for this option.

For pointers returned from stubbed functions, the option `MAIN GENERATOR` is not available. If you override the global specification for such a pointer through the **Initialize Pointer** option in constraints, you cannot toggle back to the global specification without changing the **Initialize Pointer** option too.

- If you disable this option, the verification considers that dereferences at all pointer depths are valid.

For instance, all the dereferences are considered valid in this code:

```
int*** stub(void);

void func2() {
    int ***ptr = stub();
    int **ptr2 = *ptr;
    int *ptr3 = *ptr2;
}
```

Command-Line Information

Parameter: `-stubbed-pointers-are-unsafe`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -stubbed-pointers-are-unsafe`

See Also

Constraint setup (`-data-range-specifications`)

Topics

“Specify Analysis Options”

“Specify External Constraints”

“Constraints”

Introduced in R2016b

Consider volatile qualifier on fields (`-consider-volatile-qualifier-on-fields`)

Assume that `volatile` qualified structure fields can have all possible values at any point in code

Description

This option affects a Code Prover analysis only.

Specify that the verification must take into account the `volatile` qualifier on fields of a structure.

Set Option

User interface: In your project configuration, the option is available on the **Verification Assumptions** node.

Command line: Use the option `-consider-volatile-qualifier-on-fields`. See “Command-Line Information” on page 1-221.

Why Use This Option

The `volatile` qualifier on a variable indicates that the variable value can change between successive operations even if you do not explicitly change it in your code. For instance, if `var` is a `volatile` variable, the consecutive operations `res = var;` `res =var;` can result in two different values of `var` being read into `res`.

Use this option so that the verification emulates the `volatile` qualifier for structure fields. If you select this option, the software assumes that a `volatile` structure field has a full range of values at any point in the code. The range is determined only by the data type of the structure field.

Settings

On

The verification considers the `volatile` qualifier on fields of a structure.

In the following example, the verification considers that the field `val1` can have all values allowed for the `int` type at any point in the code.

```
struct myStruct {
    volatile int val1;
    int val2;
};
```

Even if you write a specific value to `val1` and read the variable in the next operation, the variable read results in any possible value.

```
struct myStruct myStructInstance;
myStructInstance.val1 = 1;
assert (myStructInstance.val1 == 1); // Assertion can fail
```

Off (default)

The verification ignores the `volatile` qualifier on fields of a structure.

In the following example, the verification ignores the qualifier on field `val1`.

```
struct myStruct {
    volatile int val1;
    int val2;
};
```

If you write a specific value to `val1` and read the variable in the next operation, the variable read results in that specific value.

```
struct myStruct myStructInstance;
myStructInstance.val1 = 1;
assert (myStructInstance.val1 == 1); // Assertion passes
```

Tips

- If your volatile fields do not represent values read from hardware and you do not expect their values to change between successive operations, disable this option. You

are using the `volatile` qualifier for some other reason and the verification does not need to consider full range for the field values.

- If you enable this option, the number of red, gray, and green checks in your code can decrease. The number of orange checks can increase.

In the following example, a red or green check changes to orange or a gray check goes away when the option is used. Considering the `volatile` qualifier changes the check color. These examples use the following structure definition:

```
struct myStruct {
    volatile int field1;
    int field2;
};
```

Color Without Option	Result Without Option	Result With Option
Green	<pre>void main(){ struct myStruct structVal; structVal.field1 = 1; assert(structVal.field1 == 1); }</pre>	<pre>void main(){ struct myStruct structVal; structVal.field1 = 1; assert(structVal.field1 ==1); }</pre>
Red	<pre>void main(){ struct myStruct structVal; structVal.field1 = 1; assert(structVal.field1 != 1); }</pre>	<pre>void main(){ struct myStruct structVal; structVal.field1 = 1; assert(structVal.field1 !=1); }</pre>
Gray	<pre>void main(){ struct myStruct structVal; structVal.field1 = 1; if (structVal.field1 != 1) { /* Perform operation */ } }</pre>	<pre>void main(){ struct myStruct structVal; structVal.field1 = 1; if (structVal.field1 != 1) { /* Perform operation */ } }</pre>

- In C++ code, the option also applies to class members.

Command-Line Information

Parameter: -consider-volatile-qualifier-on-fields

Default: Off

Example: polyspace-code-prover-nodesktop -sources *file_name* -
consider-volatile-qualifier-on-fields

See Also

Topics

“Specify Analysis Options”

“Specify External Constraints”

Introduced in R2016b

Float rounding mode (`-float-rounding-mode`)

Specify rounding modes to consider when determining the results of floating point arithmetic

Description

This option affects a Code Prover analysis only.

Specify the rounding modes to consider when determining the results of floating-point arithmetic.

Set Option

User interface: In your project configuration, the option is available on the **Verification Assumptions** node.

Command line: Use the option `-float-rounding-mode`. See “Command-Line Information” on page 1-225.

Why Use This Option

The default verification uses the round-to-nearest mode.

Use the rounding mode `all` if your code contains routines such as `fesetround` to specify a rounding mode other than round-to-nearest. Although the verification ignores the `fesetround` specification, it considers all rounding modes including the rounding mode that you specified. Alternatively, for targets that can use extended precision (for instance, using the flag `-mfpmath=387`), use the rounding mode `all`. However, for your Polyspace analysis results to agree with run-time behavior, you must prevent use of extended precision through a flag such as `-ffloat-store`.

Otherwise, continue to use the default rounding mode `to-nearest`. Because all rounding modes are considered when you specify `all`, you can have many orange **Overflow** checks resulting from overapproximation.

Settings

Default: `to-nearest`

`to-nearest`

The verification assumes the round-to-nearest mode.

`all`

The verification assumes all rounding modes for each operation involving floating-point variables. The following rounding modes are considered: round-to-nearest, round-towards-zero, round-towards-positive-infinity, and round-towards-negative-infinity.

Tips

- The Polyspace analysis uses floating-point arithmetic that conforms to the IEEE® 754 standard. For instance, the arithmetic uses floating point instructions present in the SSE instruction set. The GNU C flag `-mfpmath=sse` enforces use of this instruction set. If you use the GNU C compiler with this flag to compile your code, your Polyspace analysis results agree with your run-time behavior.

However, if your code uses extended precision, for instance using the GNU C flag `-mfpmath=387`, your Polyspace analysis results might not agree with your run-time behavior in some corner cases. See some examples of these corner cases in `codeprover_limitations.pdf` in `matlabroot\polyspace\verifier\code_prover`. Here, `matlabroot` is the MATLAB installation folder, for instance, `C:\Program Files\MATLAB\R2017b`.

To prevent use of extended precision, on targets without SSE support, you can use a flag such as `-ffloat-store`. For your Polyspace analysis, use `all` for rounding mode to account for double rounding.

- The **Overflow** check uses the rounding modes that you specify. For instance, the following table shows the difference in the result of the check when you change your rounding modes.

Rounding mode: to-nearest	Rounding mode: a11
<p>If results of floating-point operations are rounded to nearest values:</p> <ul style="list-style-type: none"> • In the first addition operation, <code>eps1</code> is just large enough that the value nearest to <code>FLT_MAX + eps1</code> is greater than <code>FLT_MAX</code>. The Overflow check is red. • In the second addition operation, <code>eps2</code> is just small enough that the value nearest to <code>FLT_MAX + eps2</code> is <code>FLT_MAX</code>. The Overflow check is green. 	<p>Besides to-nearest mode, the Overflow check also considers other rounding modes.</p> <ul style="list-style-type: none"> • In the first addition operation, in to-nearest mode, the value nearest to <code>FLT_MAX + eps1</code> is greater than <code>FLT_MAX</code>, so the addition overflows. But if rounded towards negative infinity, the result is <code>FLT_MAX</code>, so the addition does not overflow. Combining these two rounding modes, the Overflow check is orange. • In the second addition operation, in to-nearest mode, the value nearest to <code>FLT_MAX + eps2</code> is <code>FLT_MAX</code>, so the addition does not overflow. But if rounded towards positive infinity, the result is greater than <code>FLT_MAX</code>, so the addition overflows. Combining these two rounding modes, the Overflow check is orange.
<pre>#include <float.h> #define eps1 0x1p103 #define eps2 0x0.FFFFFFFp103 float func(int ch) { float left_op = FLT_MAX; float right_op_1 = eps1, \ right_op_2 = eps2; switch(ch) { case 1: return (left_op + \ right_op_1); case 2: return (left_op + \ right_op_2); default: return 0; } }</pre>	<pre>#include <float.h> #define eps1 0x1p103 #define eps2 0x0.FFFFFFFp103 float func(int ch) { float left_op = FLT_MAX; float right_op_1 = eps1, \ right_op_2 = eps2; switch(ch) { case 1: return (left_op + \ right_op_1); case 2: return (left_op + \ right_op_2); default:</pre>

Rounding mode: to-nearest	Rounding mode: all
	<pre> return 0; } }</pre>

If you set the rounding mode to `all` and obtain an orange **Overflow** check, to determine how the overflow can occur, consider all rounding modes.

Command-Line Information

Parameter: `-float-rounding-mode`

Value: `to-nearest | all`

Default: `to-nearest`

Example: `polyspace-code-prover-nodesktop -sources file_name -float-rounding-mode all`

See Also

Overflow

Topics

“Specify Analysis Options”

Introduced in R2016a

Ignore overflowing computations on constants (`-ignore-constant-overflows`)

Allow overflow in computations involving constants

Description

This option affects a Code Prover analysis only.

Specify that the verification must allow overflow in computations involving constants.

Set Option

User interface: In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-ignore-constant-overflows`. See “Command-Line Information” on page 1-227.

Why Use This Option

Overflows in computations with compile-time constants can stop the analysis. Use this option to ignore these overflows and continue the analysis.

For instance, `char x = 0xff;` causes an overflow according to the ANSI C standard. However, if you use this option, Polyspace considers that this statement is equivalent to `char x = -1;`

Settings

On

The verification allows overflows in computations involving constants.

Off (default)

If an overflow occurs in computations involving constants, the verification can stop.

Tips

- This option applies to computations involving compile-time constants only. For instance, the statement `char x = (rand() ? 0xFF:0xFE);` causes an Overflow error irrespective of whether the option is used because the value of `x` is not known at compile-time.

Command-Line Information

Parameter: `-ignore-constant-overflows`

Default: Off

See Also

Overflow

Topics

“Specify Analysis Options”

Allow negative operand for left shifts (`-allow-negative-operand-in-shift`)

Allow left shift operations on a negative number

Description

This option affects a Code Prover analysis only.

Specify that the verification must allow left shift operations on a negative number.

Set Option

User interface: In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-allow-negative-operand-in-shift`. See “Command-Line Information” on page 1-229.

Why Use This Option

According to the C99 standard (sec 6.5.7), the result of a left shift operation on a negative number is undefined. Following the standard, the verification produces a red check on left shifts of negative numbers.

If your compiler has a well-defined behavior for left shifts of negative numbers, set this option. Note that allowing left shifts of negative numbers can reduce the cross-compiler portability of your code.

Settings

On

The verification allows shift operations on a negative number, for instance, `-2 << 2`.

Off (default)

If a shift operation is performed on a negative number, the verification generates an error.

Command-Line Information

Parameter: `-allow-negative-operand-in-shift`

Default: Off

See Also

`Invalid shift operations`

Topics

“Specify Analysis Options”

Detect overflows (`-scalar-overflows-checks`)

Specify whether to check for integer overflows on signed and unsigned variables

Description

This option affects a Code Prover analysis only.

Specify whether to check for integer overflows on signed and unsigned variables.

Set Option

User interface: In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-scalar-overflows-checks`. See “Command-Line Information” on page 1-231.

Why Use This Option

Use this option to specify the kinds of integer overflows that the verification must detect.

Settings

Default: `signed`

`signed`

The verification checks for overflows in computations involving signed integers alone. This behavior conforms to the ANSI C (ISO C++) standard.

`signed-and-unsigned`

The verification checks for overflows in all integer computations. This behavior is stricter than the ANSI C (ISO C++) standard.

none

The verification does not check for integer overflows. If a computed value exceeds the range of its type, the value is wrapped. For instance, in the following code, `x` is wrapped to 0 after the sum.

```
unsigned char x;
x = 255;
x = x+1;
```

Tips

- Following an overflow, unless you select `none`, Polyspace can either wrap the result or restrict it to its extremum value. Use **Overflow computation mode** to specify how the verification handles results of an overflow.
- Use the option `signed-and-unsigned` if you are computing the size of a buffer from unsigned integers. Using this option helps you detect an overflow at the buffer computation stage. Otherwise, you might see an error later due to insufficient buffer.
- If you use the option `signed-and-unsigned`, Polyspace does not produce an overflow error on bitwise NOT operations if you cast the result of the operation back to the operand type. For instance, Polyspace does not produce an overflow error on `(uint8_t) (~var)` where `var` is of type `uint8_t`.

Command-Line Information

Parameter: `-scalar-overflows-checks`

Value: `signed` | `signed-and-unsigned` | `none`

Default: `signed`

Example: `polyspace-code-prover-nodesktop -sources file_name -scalar-overflows-checks signed`

See Also

Overflow computation mode (-scalar-overflows-behavior) | Overflow

Topics

“Specify Analysis Options”

“Detect Overflows in Buffer Size Computation”

Overflow computation mode (`-scalar-overflows-behavior`)

Specify whether result of overflow must be wrapped around or truncated

Description

This option affects a Code Prover analysis only.

Specify whether Polyspace must wrap the result of an integer overflow or restrict it to its extremum value.

Set Option

User interface: In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-scalar-overflows-behavior`. See “Command-Line Information” on page 1-234.

Why Use This Option

Use this option to specify the assumptions to make following an integer overflow.

Settings

Default: `truncate-on-error`

`truncate-on-error`

If the **Overflow** check on an operation is:

- Red, Polyspace does not analyze the remaining code in the current scope.
- Orange, Polyspace analyzes the remaining code in the current scope. However, Polyspace considers that:

- After a positive **Overflow**, the result of the operation has an upper bound. This upper bound is the maximum value allowed by the type of the result.
- After a negative **Overflow**, the result of the operation has a lower bound. This lower bound is the minimum value allowed by the type of the result.

wrap-around

Polyspace analyzes the remaining code in the current scope even after a red integer **Overflow**. However, Polyspace wraps the result of the overflow. For instance, if you choose this option:

- In the following code, after the red **Overflow**, Polyspace considers that `i` has a value -2^{31} .

```
#include<stdio.h>

void main() {
    int i=1;
    i = i << 30;
    i = i *2;
    printf("%d",i);
}
```

- In the following code, before the orange **Overflow**, `i` has values in the range $[1..2^{31}-1]$. But, after the orange **Overflow**, Polyspace considers that `i` has even values in the range $[-2^{31}..2]$ or $[2..2^{31}-2]$.

```
#include<stdio.h>
int getVal();

void main() {
    int i=getVal();
    if(i>0) {
        i = i*2;
        printf("%d",i);
    }
}
```

Command-Line Information

Parameter: `-scalar-overflows-behavior`

Value: `wrap-around` | `truncate-on-error`

Default: `truncate-on-error`

Example: `polyspace-code-prover-nodesktop -sources file_name -scalar-overflows-behavior wrap-around`

See Also

Detect overflows (-scalar-overflows-checks) | Overflow

Topics

“Specify Analysis Options”

Disable checks for non-initialization (`-disable-initialization-checks`)

Disable checks for non-initialized variables and pointers

Description

This option affects a Code Prover analysis only.

Specify that Polyspace Code Prover must not check for non-initialization in your code.

Set Option

User interface: In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-disable-initialization-checks`. See “Command-Line Information” on page 1-238.

Why Use This Option

Use this option if you do not want to detect instances of non-initialized variables.

Settings

On

Polyspace Code Prover does not perform the following checks:

- Non-initialized local variable: Local variable is not initialized before being read.
- Non-initialized variable: Variable other than local variable is not initialized before being read.
- Non-initialized pointer: Pointer is not initialized before being read.

- Return value not initialized: C function does not return value when expected.

Polyspace assumes that, at declaration:

- Variables have full-range of values allowed by their type.
- Pointers can be NULL-valued or point to a memory block at an unknown offset.

Off (default)

Polyspace Code Prover checks for non-initialization in your code. The software displays red checks if, for instance, a variable is not initialized and orange checks if a variable is initialized only on some execution paths.

Tips

- If you select this option, the software does not report most violations of MISRA C: 2004, rule 9.1, and MISRA C:2012 Rule 9.1.
- If you select this option, the number and type of orange checks in your code can change.

For instance, the following table shows an additional orange check with the option enabled.

Checks for Non-initialization Enabled	Checks for Non-initialization Disabled
<pre>void func(int flag) { int var1,var2; if(flag==0) { var1=var2; } else { var1=0; } var2=var1 + 1; }</pre> <p>In this example, the software produces:</p> <ul style="list-style-type: none"> • A red Non-initialized local variable check on <code>var2</code> in the <code>if</code> branch. The verification continues as if only the <code>else</code> branch of the <code>if</code> statement exists. • A green Non-initialized local variable check on <code>var1</code> in the last statement. <code>var1</code> has the assigned value 0. • A green Overflow check on the <code>+</code> operation. 	<pre>void func(int flag) { int var1,var2; if(flag==0) { var1=var2; } else { var1=0; } var2=var1 + 1; }</pre> <p>In this example, the software:</p> <ul style="list-style-type: none"> • Does not produce Non-initialized local variable checks. At initialization, the software assumes that <code>var2</code> has full range of <code>int</code> values. Following the <code>if</code> statement, because the software considers both <code>if</code> branches, it assumes that <code>var1</code> also has full range of <code>int</code> values. • Produces an orange Overflow check on the <code>+</code> operation. For instance, if <code>var1</code> has the maximum <code>int</code> value, adding 1 to it can cause an overflow.

Command-Line Information

Parameter: `-disable-initialization-checks`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -disable-initialization-checks`

See Also

Topics

“Specify Analysis Options”

Detect stack pointer dereference outside scope (`-detect-pointer-escape`)

Find cases where a function returns a pointer to one of its local variables

Description

This option affects a Code Prover analysis only.

Specify that the verification must detect cases where you access a variable outside its scope via pointers. Such an access can happen, for example, when a function returns a pointer to a local variable and you dereference the pointer outside the function. The dereference causes undefined behavior because the local variable that the pointer points to does not live outside the function.

Set Option

User interface: In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-detect-pointer-escape`. See “Command-Line Information” on page 1-242.

Why Use This Option

Use this option to enable detection of pointer escape.

Settings

On

The **Illegally dereferenced pointer** check performs an additional task, besides its usual specifications. When you dereference a pointer, the check also determines if you are accessing a variable outside its scope through the pointer. The check is:

- Red, if all the variables that the pointer points to are accessed outside their scope.

For instance, you dereference a pointer `ptr` in a function `func` that is called twice in your code. In both calls, when you perform the dereference `*ptr`, `ptr` is pointing to variables outside their scope. Therefore, the **Illegally dereferenced pointer** check is red.

- Orange, if only some of the variables that the pointer points to are accessed outside their scope.
- Green, if none of the variables that the pointer points to are accessed outside their scope, and other requirements of the check are also satisfied.

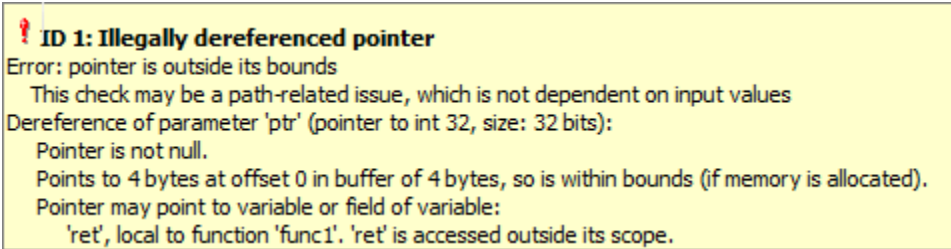
In the following code, if you enable this option, Polyspace Code Prover produces a red **Illegally dereferenced pointer** check on `*ptr`. Otherwise, the **Illegally dereferenced pointer** check on `*ptr` is green.

```
void func2(int *ptr) {
    *ptr = 0;
}

int* func1(void) {
    int ret = 0;
    return &ret ;
}

void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

The **Result Details** pane displays a message indicating that `ret` is accessed outside its scope.



! ID 1: Illegally dereferenced pointer
 Error: pointer is outside its bounds
 This check may be a path-related issue, which is not dependent on input values
 Dereference of parameter 'ptr' (pointer to int 32, size: 32 bits):
 Pointer is not null.
 Points to 4 bytes at offset 0 in buffer of 4 bytes, so is within bounds (if memory is allocated).
 Pointer may point to variable or field of variable:
 'ret', local to function 'func1'. 'ret' is accessed outside its scope.

- Off (default)

When you dereference a pointer, the **Illegally dereferenced pointer** check does not check for whether you are accessing a variable outside its scope. The check is green

even if the pointer dereference is outside the variable scope, as long as it satisfies requirements:

- The pointer is not NULL.
- The pointer points within the memory buffer.

Command-Line Information

Parameter: `-detect-pointer-escape`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -detect-pointer-escape`

See Also

Illegally dereferenced pointer

Introduced in R2015a

Enable pointer arithmetic across fields (`-allow-ptr-arith-on-struct`)

Allow arithmetic on pointer to a structure field so that it points to another field

Description

This option affects a Code Prover analysis only.

Specify that a pointer assigned to a structure field can point outside its bounds as long as it points within the structure.

Set Option

User interface: In your project configuration, the option is on the **Check Behavior** node. See “Dependency” on page 1-244 for other options you must also enable.

Command line: Use the option `-allow-ptr-arith-on-struct`. See “Command-Line Information” on page 1-244.

Why Use This Option

Use this option to relax the check for illegally dereferenced pointers. Once you assign a pointer to a structure field, you can use that pointer to access another structure field.

Settings

On

A pointer assigned to a structure field can point outside the bounds imposed by the field as long as it points within the structure. For instance, in the following code, unless you use this option, the verification will produce a red `Illegally dereferenced pointer` check:

```
void main(void) {  
  struct S {char a; char b; int c;} x;
```

```
char *ptr = &x.b;
ptr ++;
*ptr = 1; // Red on the dereference, because ptr points outside x.b
}
```

Off (default)

A pointer assigned to a structure field can point only within the bounds imposed by the field.

Tips

- The verification does not allow a pointer with negative offset values. This behavior occurs irrespective of whether you choose the option **Enable pointer arithmetic across fields**.

Dependency

This option is available only if you set `Source code language (-lang)` to C.

Command-Line Information

Parameter: `-allow-ptr-arith-on-struct`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -allow-ptr-arith-on-struct`

See Also

Allow incomplete or partial allocation of structures (`-size-in-bytes`) | Illegally dereferenced pointer

Topics

“Specify Analysis Options”

Allow incomplete or partial allocation of structures (`-size-in-bytes`)

Allow a pointer with insufficient memory buffer to point to a structure

Description

This option affects a Code Prover analysis only.

Specify that the verification must allow dereferencing a pointer that points to a structure but has a sufficient buffer for only some of the structure's fields.

This type of pointer results when a pointer to a smaller structure is cast to a pointer to a larger structure. The pointer resulting from the cast has sufficient buffer for only some fields of the larger structure.

Set Option

User interface: In your project configuration, the option is on the **Check Behavior** node. See “Dependency” on page 1-247 for other options that you must also enable.

Command line: Use the option `-size-in-bytes`. See “Command-Line Information” on page 1-247.

Why Use This Option

Use this option to relax the check for illegally dereferenced pointers. You can point to a structure even when the buffer allowed for the pointer is not sufficient for all the structure fields.

Settings

On

When a pointer with insufficient buffer is dereferenced, Polyspace does not produce an **Illegally dereferenced pointer** error, as long as the dereference occurs within allowed buffer.

For instance, in the following code, the pointer `p` has sufficient buffer for the first two fields of the structure `BIG`. Therefore, with the option on, Polyspace considers that the first two dereferences are valid. The third dereference takes `p` outside its allowed buffer. Therefore, Polyspace produces an **Illegally dereferenced pointer** error on the third dereference.

```
#include <stdlib.h>

typedef struct _little { int a; int b; } LITTLE;
typedef struct _big { int a; int b; int c; } BIG;

void main(void) {
    BIG *p = malloc(sizeof(LITTLE));

    if (p!= ((void *) 0) ) {
        p->a = 0 ;
        p->b = 0 ;
        p->c = 0 ;    // Red IDP check
    }
}
```

Off (default)

Polyspace does not allow dereferencing a pointer to a structure if the pointer does not have sufficient buffer for all fields of the structure. It produces an **Illegally dereferenced pointer** error the first time you dereference the pointer.

For instance, in the following code, even though the pointer `p` has sufficient buffer for the first two fields of the structure `BIG`, Polyspace considers that dereferencing `p` is invalid.

```
#include <stdlib.h>

typedef struct _little { int a; int b; } LITTLE;
typedef struct _big { int a; int b; int c; } BIG;
```

```
void main(void) {
    BIG *p = malloc(sizeof(LITTLE));

    if (p!= ((void *) 0) ) {
        p->a = 0 ;    // Red IDP check
        p->b = 0 ;
        p->c = 0 ;
    }
}
```

Tips

- The verification also allows partial allocation of structures when you select **Enable pointer arithmetic across fields**.
- If you do not turn on this option, you cannot point to the field of a partially allocated structure.

For instance, in the preceding example, if you do not turn on the option and perform the assignment

```
int *ptr = &(p->a);
```

Polyspace considers that the assignment is invalid. If you dereference `ptr`, it produces an **Illegally dereferenced pointer** error.

Dependency

This option is available only if you set `Source code language (-lang)` to C.

Command-Line Information

Parameter: `-size-in-bytes`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -size-in-bytes`

See Also

Enable pointer arithmetic across fields (`-allow-ptr-arith-on-struct`)
| Illegally dereferenced pointer

Topics

“Specify Analysis Options”

Permissive function pointer calls (`-permissive-function-pointer`)

Allow type mismatch between function pointers and the functions they point to

Description

This option affects a Code Prover analysis only.

Specify that the verification must allow function pointer calls where the type of the function pointer does not match the type of the function.

Set Option

User interface: In your project configuration, the option is on the **Check Behavior** node. See “Dependency” on page 1-250 for other options you must also enable.

Command line: Use the option `-permissive-function-pointer`. See “Command-Line Information” on page 1-250.

Settings

On

The verification must allow function pointer calls where the type of the function pointer does not match the type of the function. For instance, a function declared as `int f(int*)` can be called by a function pointer declared as `int fptr(void*)`.

Off (default)

The verification must require that the argument and return types of a function pointer and the function it calls are identical.

Tips

- With sources that use function pointers extensively, enabling this option can cause loss in performance. This loss occurs because the verification has to consider more execution paths.

Dependency

This option is available only if you set `Source code language (-lang)` to C.

Command-Line Information

Parameter: `-permissive-function-pointer`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -permissive-function-pointer`

See Also

Topics

“Specify Analysis Options”

Consider non finite floats (`-allow-non-finite-floats`)

Enable a verification mode that incorporates infinities and NaNs

Description

This option affects a Code Prover analysis only.

Enable a verification mode that incorporates infinities and NaNs for floating point operations.

Set Option

User interface: In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-allow-non-finite-floats`. See “Command-Line Information” on page 1-254.

Why Use This Option

By default, the verification does not incorporate infinities and NaNs. For instance, the verification terminates the execution thread where a division by zero occurs and does not consider that the result could be infinity.

If you use functions such as `isinf` or `isnan` and account for infinities and NaNs in your code, set this option. When you set this option and a division by zero occurs for instance, the execution thread continues with infinity as the result of the division.

Set this option alone if you are sure that you have accounted for infinities and NaNs in your code. Using the option alone effectively disables many numerical checks on floating point operations. If you have generally accounted for infinities and NaNs, but you are not sure that you have considered all situations, set these additional options:

- `Infinities (-check-infinite): Use warn-first.`

- `NaNs (-check-nan)`: Use `warn-first`.

Settings

On

The verification allows infinities and NaNs. For instance, in this mode:

- The verification assumes that floating-point operations can produce results such as infinities and NaNs.

By using options `Infinities (-check-infinite)` and `NaNs (-check-nan)`, you can choose to highlight operations that produce nonfinite results and stop the execution threads where the nonfinite results occur.

- The verification assumes that floating-point variables with unknown values can have any value allowed by their type, including infinite or NaN. Floating-point variables with unknown values include volatile variables and return values of stubbed functions.

Off (default)

The verification does not allow infinities and NaNs. For instance, in this mode:

- The verification produces a red check on a floating-point operation that produces an infinity or a NaN as the only possible result on all execution paths. The verification produces an orange check on a floating-point operation that can potentially produce an infinity or NaN.
- The verification assumes that floating-point variables with unknown values are full-range but finite.

Tips

- The IEEE 754 Standard allows special quantities such as infinities and NaN so that you can handle certain numerical exceptions without aborting the code. Some implementations of the C standard support infinities and NaN.
- If your compiler supports infinities and NaNs and you account for them explicitly in your code, use this option so that the verification also allows them.

For instance, if a division results in infinity, in your code, you specify an alternative action. Therefore, you do not want the verification to highlight division operations that result in infinity.

- If your compiler supports infinities and NaNs but you are not sure if you account for them explicitly in your code, use this option so that the verification incorporates infinities and NaNs. Use the options `-check-nan` and `-check-infinite` with argument `warn` so that the verification highlights operations that result in infinities and NaNs, but does not stop the execution thread.
- If you select this option, the number and type of checks in your code can change.

For instance, in the following example, when you select the option, the results have one less red check and three more green checks.

Infinities and NaNs Not Allowed	Infinities and NaNs Allowed
<p>Polyspace produces a Division by zero error and stops verification.</p> <pre data-bbox="284 800 593 968"> double func(void) { double x=1.0/0.0; double y=1.0/x; double z=x-x; return z; } </pre>	<p>If you select this option, Polyspace does not check for a Division by zero error.</p> <pre data-bbox="808 800 1113 968"> double func(void) { double x=1.0/0.0; double y=1.0/x; double z=x-x; return z; } </pre> <p>The verification assumes that dividing by zero results in:</p> <ul style="list-style-type: none"> • Value of <code>x</code> equal to <code>Inf</code> • Value of <code>y</code> equal to <code>0.0</code> • Value of <code>z</code> equal to <code>NaN</code> <p>In your verification results in the Polyspace user interface, if you place your cursor on <code>y</code> and <code>z</code>, you can see the nonfinite values <code>Inf</code> and <code>NaN</code> respectively in the tooltip.</p>

- You cannot run the Automatic Orange Tester if you incorporate non-finites in your verification.

Command-Line Information

Parameter: `-allow-non-finite-floats`

Default: Off

See Also

Infinities (`-check-infinite`) | NaNs (`-check-nan`) | Division by zero | Overflow | Invalid shift operations | Invalid use of standard library routine

Topics

“Specify Analysis Options”

Introduced in R2016a

Infinities (-check-infinite)

Specify how to handle floating-point operations that result in infinity

Description

This option affects a Code Prover analysis only.

Specify how the analysis must handle floating-point operations that result in infinities.

Set Option

User interface: In your project configuration, the option is on the **Check Behavior** node. See “Dependencies” on page 1-257 for other options you must also enable.

Command line: Use the option `-check-infinite`. See “Command-Line Information” on page 1-257.

Why Use This Option

Use this option to enable detection of floating-point operations that result in infinities.

If you specify that the analysis must consider nonfinite floats, by default, the analysis does not flag these operations. Use this option to detect these operations while still incorporating nonfinite floats.

Settings

Default: `allow`

`allow`

The verification does not produce a check on the operation.

For instance, in the following code, there is no **Overflow** check.

```
double func(void) {  
    double x=1.0/0.0;
```

```
        return x;
    }
```

warn-first

The verification produces a check on the operation. The check determines if the result of the operation is infinite when the operands themselves are not infinite. The verification does not terminate the execution thread that produces infinity.

If the verification detects an operation that produces infinity as the only possible result on all execution paths and the operands themselves are never infinite, the check is red. If the operation can potentially result in infinity, the check is orange.

For instance, in the following code, there is a nonblocking **Overflow** check for infinity.

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

Even though the **Overflow** check on the / operation is red, the verification continues. For instance, a green **Non-initialized local variable** check appears on x in the return statement.

forbid

The verification produces a check on the operation and terminates the execution thread that produces infinity.

If the check is red, the verification does not continue for the remaining code in the same scope as the check. If the check is orange, the verification continues but removes from consideration the variable values that produced infinity.

For instance, in the following code, there is a blocking **Overflow** check for infinity.

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

The verification stops because the **Overflow** check on the / operation is red. For instance, a **Non-initialized local variable** check does not appear on x in the return statement.

Dependencies

To use this option, you must enable the verification mode that incorporates infinities and NaNs. See `Consider non finite floats (-allow-non-finite-floats)`.

Command-Line Information

Parameter: `-check-infinite`

Value: `allow | warn-first | forbid`

Default: `allow`

Example: `polyspace-code-prover-nodesktop -sources file_name -check-infinite forbid`

See Also

Polyspace Analysis Options

`Consider non finite floats (-allow-non-finite-floats) | NaNs (-check-nan)`

Polyspace Results

`Overflow`

Topics

“Specify Analysis Options”

Introduced in R2016a

NaNs (`-check-nan`)

Specify how to handle floating-point operations that result in NaN

Description

This option affects a Code Prover analysis only.

Specify how the analysis must handle floating-point operations that result in NaN.

Set Option

User interface: In your project configuration, the option is on the **Check Behavior** node. See “Dependencies” on page 1-260 for other options you must also enable.

Command line: Use the option `-check-nan`. See “Command-Line Information” on page 1-260.

Why Use This Option

Use this option to enable detection of floating-point operations that result in NaN-s.

If you specify that the analysis must consider nonfinite floats, by default, the analysis does not flag these operations. Use this option to detect these operations while still incorporating nonfinite floats.

Settings

Default: `allow`

`allow`

The verification does not produce a check on the operation.

For instance, in the following code, there is no **Invalid operation on floats** check.

```
double func(void) {  
    double x=1.0/0.0;
```

```
    double y=x-x;
    return y;
}
```

warn-first

The verification produces a check on the operation. The check determines if the result of the operation is NaN when the operands themselves are not NaN. For instance, the check flags the operation `val1 + val2` only if the result can be NaN when *both* `val1` and `val2` are not NaN. The verification does not terminate the execution thread that produces NaN.

If the verification detects an operation that produces NaN as the only possible result on all execution paths and the operands themselves are never NaN, the check is red. If the operation can potentially result in NaN, the check is orange.

For instance, in the following code, there is a nonblocking **Invalid operation on floats** check for NaN.

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
    return y;
}
```

Even though the **Invalid operation on floats** check on the `-` operation is red, the verification continues. For instance, a green **Non-initialized local variable** check appears on `y` in the return statement.

forbid

The verification produces a check on the operation and terminates the execution thread that produces NaN.

If the check is red, the verification does not continue for the remaining code in the same scope as the check. If the check is orange, the verification continues but removes from consideration the variable values that produced a NaN.

For instance, in the following code, there is a blocking **Invalid operation on floats** check for NaN.

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
```

```
        return y;  
    }
```

The verification stops because the **Invalid operation on floats** check on the `-` operation is red. For instance, a **Non-initialized local variable** check does not appear on `y` in the `return` statement.

The **Invalid operation on floats** check for NaN also appears on the `/` operation and is green.

Dependencies

To use this option, you must enable the verification mode that incorporates infinities and NaNs. See `Consider non finite floats (-allow-non-finite-floats)`.

Command-Line Information

Parameter: `-check-nan`

Value: `allow | warn-first | forbid`

Default: `allow`

Example: `polyspace-code-prover-nodesktop -sources file_name -check-nan forbid`

See Also

Polyspace Analysis Options

`Consider non finite floats (-allow-non-finite-floats) | Infinities (-check-infinite)`

Polyspace Results

`Invalid operation on floats`

Topics

“Specify Analysis Options”

Introduced in R2016a

Subnormal detection mode (`-check-subnormal`)

Detect operations that result in subnormal floating-point values

Description

This option affects a Code Prover analysis only.

Specify that the verification must check floating-point operations for subnormal results.

Set Option

User interface: In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-check-subnormal`. See “Command-Line Information” on page 1-264.

Why Use This Option

Use this option to detect floating-point operations that result in subnormal values.

Subnormal numbers have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the significand. The presence of subnormal numbers indicates loss of significant digits. This loss can accumulate over subsequent operations and eventually result in unexpected values. Subnormal numbers can also slow down the execution on targets without hardware support.

Settings

Default: `allow`

`allow`

The verification does not check operations for subnormal results.

`forbid`

The verification checks for subnormal results.

The verification stops the execution path with the subnormal result and prevents subnormal values from propagating further. Therefore, in practice, you see only the first occurrence of the subnormal value.

`warn-all`

The verification checks for subnormal results and highlights all occurrences of subnormal values. Even if a subnormal result comes from previous subnormal values, the result is highlighted.

The verification continues even if the check is red.

`warn-first`

The verification checks for subnormal results but only highlights first occurrences of subnormal values. If a subnormal value propagates to further subnormal results, those subsequent results are not highlighted.

The verification continues even if the check is red.

For details of the result colors in each mode, see `Subnormal float`.

Tips

- If you want to see only those operations where a subnormal value originates from non-subnormal operands, use the `warn-first` mode.

For instance, in the following code, `arg1` and `arg2` are unknown. The verification assumes that they can take all values allowed for the type `double`. This assumption can lead to subnormal results from certain operations. If you use the `warn-first` mode, the first operation causing the subnormal result is highlighted.

warn-all	warn-first
<pre data-bbox="283 302 802 499">void func (double arg1, double arg2) { double difference1 = arg1 - arg2; double difference2 = arg1 - arg2; double val1 = difference1 * 2; double val2 = difference2 * 2; }</pre> <p data-bbox="283 562 783 652">In this example, all four operations can have subnormal results. The four checks for subnormal results are orange.</p>	<pre data-bbox="813 302 1337 499">void func (double arg1, double arg2) { double difference1 = arg1 - arg2; double difference2 = arg1 - arg2; double val1 = difference1 * 2; double val2 = difference2 * 2; }</pre> <p data-bbox="813 534 1317 826">In this example, difference1 and difference2 can be subnormal if arg1 and arg2 are sufficiently close. The first two checks for subnormal results are orange. val1 and val2 cannot be subnormal unless difference1 and difference2 are subnormal. The last two checks for subnormal results are green.</p> <p data-bbox="813 861 1322 1013">Through red/orange checks, you see only the first instance where a subnormal value appears. You do not see red/orange checks from those subnormal values propagating to subsequent operations.</p>

- If you want to see where a subnormal value originates and do not want to see subnormal results arising from the same cause more than once, use the `forbid` mode.

For instance, in the following code, `arg1` and `arg2` are unknown. The verification assumes that they can take all values allowed for the type `double`. This assumption can lead to subnormal results for `arg1-arg2`. If you use the `forbid` mode and perform the operation `arg1-arg2` twice in succession, only the first operation is highlighted. The second operation is not highlighted because the subnormal result for the second operation arises from the same cause as the first operation.

warn-all	forbid
<pre>void func (double arg1, double arg2) { double difference1 = arg1 - arg2; double difference2 = arg1 - arg2; double val1 = difference1 * 2; double val2 = difference2 * 2; }</pre> <p>In this example, all four operations can have subnormal results. The four checks for subnormal results are orange.</p>	<pre>void func (double arg1, double arg2) { double difference1 = arg1 - arg2; double difference2 = arg1 - arg2; double val1 = difference1 * 2; double val2 = difference2 * 2; }</pre> <p>In this example, difference1 can be subnormal if arg1 and arg2 are sufficiently close. The first check for subnormal results is orange. Following this check, the verification excludes from consideration:</p> <ul style="list-style-type: none"> The close values of arg1 and arg2 that led to the subnormal value of difference1. <p>In the subsequent operation arg1 - arg2, the check is green and difference2 is not subnormal. The result of the check on difference2 * 2 is green for the same reason.</p> <ul style="list-style-type: none"> The subnormal value of difference1. <p>In the subsequent operation difference1 * 2, the check is green.</p>

- You cannot run the Automatic Orange Tester if you check for subnormals in your verification.

Command-Line Information

Parameter: -check-subnormal

Value: allow | warn-first | warn-all | forbid

Default: allow

Example: polyspace-code-prover-nodesktop -sources *file_name* -check-subnormal forbid

See Also

Polyspace Results

Subnormal float

Topics

“Specify Analysis Options”

Introduced in R2016b

Detect uncalled functions (`-uncalled-function-checks`)

Detect functions that are not called directly or indirectly from `main` or another entry point function

Description

This option affects a Code Prover analysis only.

Detect functions that are not called directly or indirectly from `main` or another entry point function during run-time.

Set Option

User interface: In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-uncalled-function-checks`. See “Command-Line Information” on page 1-267.

Why Use This Option

Typically, after verification, the **Dashboard** pane shows functions that are not called during verification. However, you do not see them in your analysis results or reports. You cannot comment on them or justify them.

If you want to see these uncalled functions in your analysis results and reports, use this option.

Settings

Default: none

none

The verification does not generate checks for uncalled functions.

never-called

The verification generates checks for functions that are defined but not called.

called-from-unreachable

The verification generates checks for functions that are defined and called from an unreachable part of the code.

all

The verification generates checks for functions that are:

- Defined but not called
- Defined and called from an unreachable part of the code.

Command-Line Information

Parameter: -uncalled-function-checks

Value: none | never-called | called-from-unreachable | all

Default: none

Example: polyspace-code-prover-nodesktop -sources *file_name* -uncalled-function-checks all

See Also

Function not called | Function not reachable

Topics

“Specify Analysis Options”

“Review Gray Checks”

“Review and Fix Function Not Called Checks”

“Review and Fix Function Not Reachable Checks”

Precision level (-o)

Specify a precision level for the verification

Description

This option affects a Code Prover analysis only.

Specify the precision level that the verification must use.

Set Option

User interface: In your project configuration, the option is available on the **Precision** node.

Command line: Use the option `-o#`, for instance, `-o0` or `-o1`. See “Command-Line Information” on page 1-269.

Why Use This Option

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

In most cases, you see the optimal balance between precision and verification time at level 2.

Settings

Default: 2

0

This option corresponds to a static interval verification.

1

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

2

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

3

This option is only suitable for code having less than 1000 lines. Using this option, the percentage of proven results can be very high.

Tips

For best results in reasonable time, use the default level 2. If the verification takes a long time, reduce precision. However, the number of unproven checks can increase. Likewise, to reduce orange checks, you can improve your precision. But the verification can take significantly longer time.

Command-Line Information

Parameter: -O0 | -O1 | -O2 | -O3

Default: -O2

Example: `polyspace-code-prover-nodesktop -sources file_name -O1`

See Also

Verification level (-to)

Topics

“Specify Analysis Options”

“Improve Verification Precision”

Verification level (`-to`)

Specify number of times the verification process runs on your code

Description

This option affects a Code Prover analysis only.

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

Set Option

User interface: In your project configuration, the option is available on the **Precision** node.

Command line: Use the option `-to`. See “Command-Line Information” on page 1-273.

Why Use This Option

There are many reasons you might want to increase or decrease the verification level. For instance:

- Coding rules are checked early during the compilation phase, with some exceptions only. If you check for coding rules alone, you can lower the verification level.
- If you see many orange checks after verification, try increasing the verification level. However, increasing the verification level also increases verification time.

In most cases, you see the optimal balance between precision and verification time at level 2.

Settings

Default: `Software Safety Analysis level 2`

Source Compliance Checking

Polyspace completes coding rules checking at the end of the compilation phase.

Software Safety Analysis level 0

The verification process runs once on your source code.

Software Safety Analysis level 1

The verification process runs twice on your source code.

Software Safety Analysis level 2

The verification process runs three time on your source code. Use this option for most accurate results in reasonable time.

Software Safety Analysis level 3

The verification process runs four times on your source code.

Software Safety Analysis level 4

The verification process runs five times on your source code.

other

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

Tips

- Use a higher verification level for fewer orange checks.

Difference between Level 0 and 1

The following example illustrates the difference between Software Safety Analysis level 0 and Software Safety Analysis level 1:

Software Safety Analysis Level 0	Software Safety Analysis Level 1
<pre>#include <stdlib.h> void ratio (float x, float *y) { *y=(abs(x-*y))/(x+*y); } void levell1 (float x, float y, float *t) { float v; v = y; ratio (x, &y); *t = 1.0/(v - 2.0 * x); } float level2(float v) { float t; t = v; levell1(0.0, 1.0, &t); return t; } void main(void) { float r,d; d= level2(1.0); r = 1.0 / (2.0 - d); }</pre>	<pre>#include <stdlib.h> void ratio (float x, float *y) { *y=(abs(x-*y))/(x+*y); } void levell1 (float x, float y, float *t) { float v; v = y; ratio (x, &y); *t = 1.0/(v - 2.0 * x); } float level2(float v) { float t; t = v; levell1(0.0, 1.0, &t); return t; } void main(void) { float r,d; d= level2(1.0); r = 1.0 / (2.0 - d); }</pre>

In the table, verification produces an orange Division by Zero check during level 0 verification. The check turns green during level 1. The verification acquires more precise knowledge of x in the higher level.

If a higher verification level fails because the verification runs out of memory, but results are available at a lower level, Polyspace displays the results from the lower level.

- For best results, use the option `Software Safety Analysis level 2`. If the verification takes too long, use a lower **Verification level**. Fix red errors and gray code before rerunning the verification with higher verification levels.

- Use the option `Other` sparingly since it can increase verification time by an unreasonable amount. Using `Software Safety Analysis level 2` provides optimal verification of your code in most cases.
- If you want to check for coding rules only, you can run Polyspace on your source code up to the `Source Compliance Checking` phase.

With the exception of certain rules, Polyspace checks for coding rule violations during the compilation phase.

- If the **Verification Level** is set to `Source Compliance Checking`, do not run verification on a remote server. The source compliance checking, or compilation, phase takes place on your local computer anyway. Therefore, if you are running verification only to the end of compilation, run verification on your local computer.

Command-Line Information

Parameter: `-to`

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

Default: `pass2`

Example: `polyspace-code-prover-nodesktop -sources file_name -to pass2`

See Also

Topics

“Specify Analysis Options”

“Improve Verification Precision”

Verification time limit (`-timeout`)

Specify a time limit on your verification

Description

This option affects a Code Prover analysis only.

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

Set Option

User interface: In your project configuration, the option is available on the **Precision** node.

Command line: Use the option `-timeout`. See “Command-Line Information” on page 1-274.

Why Use This Option

Use this option to impose a time limit on the verification.

The option is useful only in very specific cases. Suppose your code has certain constructs that might slow down the verification. To check this, Technical Support can ask you to impose a time limit on the verification so that the verification stops if it takes too long.

Settings

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

Command-Line Information

Parameter: `-timeout`

Value: *time*

Example: polyspace-code-prover-nodesktop -sources *file_name* -timeout
5.75

See Also

Topics

“Specify Analysis Options”

“Improve Verification Precision”

Sensitivity context (`-context-sensitivity`)

Store call context information to identify function call that caused errors

Description

This option affects a Code Prover analysis only.

Specify the functions for which the verification must store call context information. If the function is called multiple times, using this option helps you to distinguish between the different calls.

Set Option

User interface: In your project configuration, the option is available on the **Precision** node.

Command line: Use the option `-context-sensitivity`. See “Command-Line Information” on page 1-277.

Why Use This Option

Suppose a function is called twice in your code. The check color on each operation in the function body is a combined result of both calls. If you want to distinguish between the colors in the two calls, use this option.

For instance, if a function contains a red or orange check and a green check on the same operation for two different calls, the software combines the contexts and displays an orange check on the operation. If you use this option, you can identify the color of the check for each call. For a tutorial on using this option, see “Identify Function Call with Run-Time Error”.

Settings

Default: none

none


The software does not store call context information for functions.

auto

The software stores call context information for checks in:

- Functions that form the leaves of the call tree. These functions are called by other functions, but do not call functions themselves.
- Small functions. The software uses an internal threshold to determine whether a function is small.

custom

The software stores call context information for functions that you specify. To enter the name of a function, click .

Command-Line Information

Parameter: `-context-sensitivity`

Value: `function1[,function2,...]`

Default: none

Example: `polyspace-code-prover-nodesktop -sources file_name -context-sensitivity myFunc1,myFunc2`

To allow the software to determine which functions receive call context storage, use the option `-context-sensitivity-auto`.

See Also

Topics

“Specify Analysis Options”

Improve precision of interprocedural analysis (`-path-sensitivity-delta`)

Avoid certain verification approximations for code with fewer lines

Description

This option affects a Code Prover analysis only.

For smaller code, use this option to improve the precision of cross-functional analysis.

Set Option

User interface: In your project configuration, the option is available on the **Precision** node.

Command line: Use the option `-path-sensitivity-delta`. See “Command-Line Information” on page 1-279.

Why Use This Option

Use this option to avoid certain software approximations on execution paths. Avoiding these approximations results in fewer orange checks but a much longer verification time.

For instance, for deep function call hierarchies or nested conditional statements, to complete verification in a reasonable amount of time, the software combines many execution paths and stores less information at each stage of verification. If you use this option, the software stores more information about the execution paths, resulting in a more precise verification.

Settings

Default: Off

Enter a positive integer to turn on this option.

Entering a higher value leads to a greater number of proven results, but also increases verification time exponentially. For instance, a value of 10 can result in very long verification times.

Tips

Use this option only when you have less than 1000 lines of code.

Command-Line Information

Parameter: `-path-sensitivity-delta`

Value: Positive integer

Example: `polyspace-code-prover-nodesktop -sources file_name -path-sensitivity-delta 1`

See Also

Topics

“Specify Analysis Options”

“Improve Verification Precision”

Specific precision (`-modules-precision`)

Specify source files you want to verify at higher precision than the remaining verification

Description

This option affects a Code Prover analysis only.

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

Set Option

User interface: In your project configuration, the option is available on the **Precision** node. See “Dependency” on page 1-281 for other options you must also enable.

Command line: Use the option `-modules-precision`. See “Command-Line Information” on page 1-281.


Why Use This Option

If a specific file is verified imprecisely leading to many orange checks in the file and elsewhere, you can improve the precision for that file.

Note that increasing precision also increases verification time.

Settings

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

Click  to enter the name of a file without the extension `.c` and the corresponding precision level.

Dependency

This option is available only if you set Source code language (-lang) to C.

Command-Line Information

Parameter: -modules-precision

Value: *file:00* | *file:01* | *file:02* | *file:03*

Example: polyspace-code-prover-nodesktop -sources *file_name* -O1 -modules-precision My_File:02

See Also

Precision level (-O)

Topics

“Specify Analysis Options”

“Improve Verification Precision”

Inline (`-inline`)

Specify functions that must be cloned internally for each function call

Description

This option affects a Code Prover analysis only.

Specify the functions that the verification must clone internally for every function call.

Set Option

User interface: In your project configuration, the option is available on the **Scaling** node.

Command line: Use the option `-inline`. See “Command-Line Information” on page 1-284.

Why Use This Option



Use this option sparingly. Sometimes, using the option helps to work around scaling issues during verification. If your verification takes too long, Technical Support can ask you to use this option for certain functions.

Do not use this option to understand results. For instance, suppose a function is called twice in your code. The check color on each operation in the function body is a combined result of both calls. If you want to distinguish between the colors in the two calls, use the option `Sensitivity context (-context-sensitivity)`.

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

The verification internally clones the function for each call. For instance, if you specify the function `func` for inlining and `func` is called twice, the software creates two copies of `func` for verification. The copies are named using the convention `func_pst_inlined_ver` where `ver` is the version number. You see both copies on the **Call Hierarchy** pane.

However, for each run-time check in the function body, you see only one color in your verification results. The semantics of the check color is different from the normal specification.

Red checks:

- Normally, if a function is called twice and an operation causes a definite error only in one of the calls, the check color is orange.
- If you use this option, the worst color is shown for the check. Therefore, the check is red.

Gray checks:

- Normally, if a function is called twice and an `if` statement branch is unreachable in only one of the calls, the branch is shown as reachable.
- If you use this option, the worst color is shown for the check. Therefore, the `if` branch appears gray.

Do not use this option to understand results. Use this option only if a certain function causes scaling issues.

Tips

- Use this option to identify the cause of a **Non-terminating call** error.
 - **Situation:** Sometimes, a red **Non-terminating call** check can appear on a function call though a red check does not appear in the function body. The function body represents all calls to the function. Therefore, if some calls to a function do not cause an error, an orange check appears in the function body.

- **Action:** If you use this option, for every function call, there is a corresponding function body. Therefore, you can trace a red check on a function call to a red check in the function body.
- Using this option can sometimes duplicate a lot of code and lead to scaling problems. Therefore choose functions to inline carefully.
- Choose functions to inline based on hints provided by the alias verification.
- Do not use this option for entry point functions, including `main`.
- Using this option can increase the number of gray **Unreachable code** checks.

For example, in the following code, if you enter `max` for **Inline**, you obtain two **Unreachable code** checks, one for each call to `max`.

```
int max(int a, int b) {
    return a > b ? a : b;
}

void main() {
    int i=3, j=1, k;
    k=max(i,j);
    i=0;
    k=max(i,j);
}
```

- If you use the keyword `inline` before a function definition, place the definition in a header file and call the function from multiple source files, you have the same result as using the option **Inline**.
- For C++ code, this option applies to all overloaded methods of a class.

Command-Line Information

Parameter: `-inline`

Value: `function1[,function2[,...]]`

No Default

Example: `polyspace-code-prover-nodesktop -sources file_name -inline func1,func2`

See Also

Topics

“Specify Analysis Options”

Depth of verification inside structures (`-k-limiting`)

Limit the depth of analysis for nested structures

Description

This option affects a Code Prover analysis only.

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

Set Option

User interface: In your project configuration, the option is available on the **Scaling** node.

Command line: Use the option `-k-limiting`. See “Command-Line Information” on page 1-287.

Why Use This Option

Use this option if the analysis is slow because your code has a structure that is many levels deep.

Typically, you see a warning message when a structure with a deep hierarchy is slowing down the verification.

Settings

Default: Full depth of nested structures is analyzed.

Enter a number to specify the depth of analysis for nested structures. For instance, if you specify 0, the analysis does not verify a structure inside a structure.

If you specify a number less than 2, the verification could be less precise.

Command-Line Information

Parameter: -k-limiting

Value: *positive integer*

Default: polyspace-code-prover-nodesktop -sources *file_name* -k-limiting 3

See Also

Topics

“Specify Analysis Options”

Generate report

Specify whether to generate a report after the analysis

Description

Specify whether to generate a report after the analysis.

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

Set Option

User interface: In your project configuration, the option is available on the **Reporting** node.

Command line: See “Command-Line Information” on page 1-289.

Why Use This Option

You can generate a report from your analysis results for archiving purposes. You can provide this report to your management or clients as proof of code quality.

Using other analysis options, you can tailor the report content and format for your specific needs. See Bug Finder and Code Prover report (`-report-template`) and Output format (`-report-output-format`).

Settings

On

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

Off (default)

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

Tips

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

Command-Line Information

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

See Also

Bug Finder and Code Prover report (`-report-template`) | Output format (`-report-output-format`)

Topics

“Specify Analysis Options”
“Generate Report”

Bug Finder and Code Prover report (`-report-template`)

Specify template for generating analysis report

Description

Specify template for generating analysis report.

.rpt files for the report templates are available in `matlabroot\toolbox\polyspace\psrptgen\templates\`. Here, `matlabroot` is the MATLAB installation folder.

Set Option

User interface: In your project configuration, the option is on the **Reporting** node. You have separate options for Bug Finder and Code Prover analysis. See “Dependencies” on page 1-296 for other options you must also enable.

Command line: Use the option `-report-template`. See “Command-Line Information” on page 1-297.

Why Use This Option

Depending on the template that you use, the report contains information about certain types of results from the **Results List** pane. The template also determines what information is presented in the report and how the information is organized. See the template descriptions below.

Settings – Bug Finder

Default: `BugFinderSummary`

`BugFinderSummary`

The report lists:

- **Polyspace Bug Finder Summary:** Number of results in the project. The results are summarized by file. The files that are partially analyzed because of compilation errors are listed in a separate table.
- **Code Metrics Summary:** Summary of the various code complexity metrics. For more information, see “Code Metrics” (Polyspace Bug Finder).
- **Defect Summary:** Defects that Polyspace Bug Finder looks for. For each defect, the report lists the:
 - Defect group.
 - Defect name.
 - Number of instances of the defect found in the source code.
- **Coding Rules Summary:** Coding rules along with number of violations.

BugFinder

The report lists:

- **Polyspace Bug Finder Summary:** Number of results in the project. The results are summarized by file. The files that are partially analyzed because of compilation errors are listed in a separate table.
- **Code Metrics Summary:** Summary of the various code complexity metrics. For more information, see “Code Metrics” (Polyspace Bug Finder).
- **Defects:** Defects found in the source code. For each defect, the report lists the:
 - Function containing the defect.
 - Defect information on the **Result Details** pane.
 - Review information, such as **Severity**, **Status** and comments.
- **Coding Rules:** Coding rule violations in the source code. For each rule violation, the report lists the:
 - Rule number and description.
 - Function containing the rule violation.
 - Review information, such as **Severity**, **Status** and comments.
- **Configuration Settings:** List of analysis options that Polyspace uses for analysis. For more information, see “Analysis Options” (Polyspace Bug Finder). If your project has source files with compilation errors, these files are also listed.

If you check for coding rules, an additional **Coding Rules Configuration** section states the rules along with the information whether they were enabled or disabled.

BugFinder_CWE

The report contains the same information as the BugFinder report. However, in the **Defects** chapter, an additional column lists the CWE™ identifiers for each defect.

CodeMetrics

The report lists the following:

- **Code Metrics Summary:** Various quantities related to the source code. For more information, see “Code Metrics” (Polyspace Bug Finder).
- **Code Metrics Details:** Various quantities related to the source code with the information broken down by file and function.

CodingRules

For C code, the report lists information about compliance with:

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

For C++ code, the report lists information about compliance with:

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

This report also contains the Polyspace configuration settings for the analysis. An additional section states the rules along with the information whether they were enabled or disabled.

Metrics

Only available for results downloaded from the Polyspace Metrics interface.

The report lists information useful to quality engineers and available on the Polyspace Metrics interface, including:

- Information about whether the project satisfies quality objectives

- Time taken in each phase of analysis
- Metrics about the whole project. For each metric, the report lists the quality threshold and whether the metric satisfies this threshold.
- Coding rule violations in the project. For each rule, the report lists the number of violations justified and whether the justifications satisfy quality objectives.
- Definite as well as possible run-time errors in the project. For each type of run-time error, the report lists the number of errors justified and whether the justifications satisfy quality objectives.

The appendices contain further details of Polyspace configuration settings, code metrics, coding rule violations, and run-time errors.

Settings – Code Prover

Default: Developer

CallHierarchy

The report displays the call hierarchy in your source code. For each function call in your source code, the report displays the following information:

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

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

- File containing the function call.

In addition, the line and column is also displayed.

- File containing the function definition.

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

In addition, the report also displays uncalled functions.

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

CodeMetrics

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

CodingRules

For C code, the report lists information about compliance with:

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

For C++ code, the report lists information about compliance with:

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

This report also contains the Polyspace configuration settings and modifiable assumptions used in the analysis. An additional section states the rules along with the information whether they were enabled or disabled.

Developer

The report lists information useful to developers, including:

- Summary of results
- Coding rule violations
- List of proven run-time errors or red checks
- List of unproven run-time errors or orange checks
- List of unreachable procedures or gray checks
- Global variable usage in code. See “Global Variables”.

The report also contains the Polyspace configuration settings and modifiable assumptions used in the analysis. If your project has source files with compilation errors, these files are also listed.

DeveloperReview

The report lists the same information as the Developer report. However, the reviewed results are sorted by severity and status, and unreviewed results are sorted by file location.

Developer_withGreenChecks

The report lists the same information as the `Developer` report. In addition, the report lists code proven to be error-free or green checks.

Quality

The report lists information useful to quality engineers, including:

- Summary of results
- Statistics about the code
- Graphs showing distributions of checks per file

The report also contains the Polyspace configuration settings and modifiable assumptions used in the analysis. If your project has source files with compilation errors, these files are also listed.

SoftwareQualityObjectives

The report lists information useful to quality engineers and available on the Polyspace Metrics interface, including:

- Information about whether the project satisfies quality objectives
- Time taken in each phase of verification
- Metrics about the whole project. For each metric, the report lists the quality threshold and whether the metric satisfies this threshold.
- Coding rule violations in the project. For each rule, the report lists the number of violations justified and whether the justifications satisfy quality objectives.
- Definite as well as possible run-time errors in the project. For each type of run-time error, the report lists the number of errors justified and whether the justifications satisfy quality objectives.

The appendices contain further details of Polyspace configuration settings, code metrics, coding rule violations, and run-time errors.

This template is available only if you generate a report from results downloaded from the Polyspace Metrics web dashboard.

SoftwareQualityObjectives_Summary

The report contains the same information as the `SoftwareQualityObjectives` report. However, it does not have the supporting appendices with details of code metrics, coding rule violations and run-time errors.

This template is available only if you generate a report from results downloaded from the Polyspace Metrics web dashboard.

VariableAccess

The report displays the global variable access in your source code. The report first displays the number of global variables of each type. For information on the types, see “Global Variables”. For each global variable, the report displays the following information:

- Variable name.

The entry for each variable is denoted by |.

- Type of the variable.
- Number of read and write operations on the variable.
- Details of read and write operations. For each read or write operation, the table displays the following information:

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

The entry for each read or write operation is denoted by ||. Write operations are denoted by < and read operations by >.

- Line and column number of the operation.

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

Dependencies

This option is available only if you select the `Generate report` check box.

Tips

The first chapter of the reports contain a summary of the relevant results. You can enter a Pass/Fail status in that chapter for your project based on the summary. If you use the template `SoftwareQualityObjectives` or `SoftwareQualityObjectives_Summary`, the status is automatically assigned based on your objectives and the verification results.

For more information on enforcing objectives using Polyspace Metrics, see “Compare Metrics Against Software Quality Objectives”.

Command-Line Information

Parameter: -report-template

Value: Full path to *template.rpt*

Example: polyspace-bug-finder-nodesktop -sources *file_name* -report-template *matlabroot\toolbox\polyspace\psrptgen\templates\bug_finder\BugFinder.rpt*

Example: polyspace-code-prover-nodesktop -sources *file_name* -report-template *matlabroot\toolbox\polyspace\psrptgen\templates\Developer.rpt*

See Also

Generate report | Output format (-report-output-format)

Topics

“Generate Reports” (Polyspace Bug Finder)

Output format (`-report-output-format`)

Specify output format of generated report

Description

Specify output format of analysis report.

Set Option

User interface: In your project configuration, the option is on the **Reporting** node. See “Dependencies” on page 1-299 for other options you must also enable.

Command line: Use the option `-report-output-format`. See “Command-Line Information” on page 1-299.

Why Use This Option

Use this option to specify whether you want a report in PDF, HTML or another format.

Settings

Default: Word

HTML

Generate report in `.html` format

PDF

Generate report in `.pdf` format

Word

Generate report in `.docx` format.

Tips

If the table of contents or graphics in a .docx report appear outdated, select the content of the report and refresh the document. Use keyboard shortcuts **Ctrl+A** to select the content and **F9** to refresh it.

Dependencies

This option is enabled only if you select the **Generate report** box.

Command-Line Information

Parameter: -report-output-format

Value: html | pdf | word

Default: word

Example: polyspace-code-prover-nodesktop -sources *file_name* -report-output-format pdf

See Also

Generate report | Bug Finder and Code Prover report (-report-template)

Topics

“Specify Analysis Options”

“Generate Report”

Run Bug Finder or Code Prover analysis on a remote cluster (`-batch`)

Enable batch remote analysis

Description

Enable batch remote analysis.

For batch remote analysis, you need:

- Polyspace and MATLAB Distributed Computing Server™ on the cluster
- MATLAB, Polyspace and Parallel Computing Toolbox™ on your local computer.

Set Option

User interface: In your project configuration, the option is on the **Run Settings** node. You have separate options for a Bug Finder and a Code Prover analysis.

Command line: Use the option `-batch`. See “Command-Line Information” on page 1-301.

Why Use This Option

Use this option if you want the analysis to run on a remote cluster instead of your local desktop.

For instance, you can run remote analysis when:

- You want to shut down your local machine but not interrupt the analysis.
- You want to free execution time on your local machine.
- You want to transfer the analysis to a more powerful computer.

Settings

On

Run batch analysis on a remote computer. In this remote analysis mode, the analysis is queued on a cluster after the compilation phase. Therefore, on your local computer, after the analysis is queued:

- If you are running the analysis from the Polyspace user interface, you can close the user interface.
- If you are running the analysis from the command line, you can close the command-line window.

You can manage the queue from the Polyspace Job Monitor. To use the Polyspace Job Monitor:

- In the Polyspace user interface, select **Tools > Open Job Monitor**.
- On the DOS or UNIX® command line, use the `polyspace-jobs-manager` command. For more information, see “Run Remote Analysis at the Command Line”.
- On the MATLAB command line, use the `polyspaceJobsManager` function.

After the analysis, you might have to manually download the results from the cluster.

Off (default)

Do not run batch analysis on a remote computer.

Dependency

Do not run analysis on a remote cluster if you run up to the **Verification Level of Source Compliance Checking**. For both local and remote analysis, the source compliance checking or compilation phase takes place on your local computer. Therefore, if you are running only up to this phase, run on your local computer.

Command-Line Information

To run a remote analysis from the command line, use with the `-scheduler` option.

Parameter: `-batch`

Value: `-scheduler host_name` if you have not set the **Job scheduler host name** in the Polyspace user interface

Default: Off

Example: `polyspace-code-prover-nodesktop -batch -scheduler NodeHost`
`polyspace-code-prover-nodesktop -batch -scheduler MJSName@NodeHost`

Example: `polyspace-bug-finder-nodesktop -batch -scheduler NodeHost`
`polyspace-bug-finder-nodesktop -batch -scheduler MJSName@NodeHost`

See Also

Upload results to Polyspace Metrics (`-add-to-results-repository`) | `-scheduler`

Topics

“Specify Analysis Options”

“Set Up Server for Metrics and Remote Analysis”

“Run Remote Verification”

“Run Remote Analysis at the Command Line”

Upload results to Polyspace Metrics (-add-to-results-repository)

Upload analysis results for viewing on Polyspace Metrics web dashboard

Description

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

Set Option

User interface: In your project configuration, the option is on the **Run Settings** node. You have separate options for a Bug Finder and a Code Prover analysis. See “Dependencies” on page 1-304 for other options that you must also enable.

Command line: Use the option `-add-to-results-repository`. See “Command-Line Information” on page 1-304.

Why Use This Option

Polyspace Metrics is a web dashboard that generates code quality metrics from your analysis results. Using this dashboard, you can:

- Provide your management a high-level overview of your code quality.
- Compare your code quality against predefined standards.
- Establish a process where you review in detail only those results that fail to meet standards.
- Track improvements or regression in code quality over time.

See “Generate Code Quality Metrics”.

Settings

On

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

Off (default)

Analysis results are stored locally.

Dependencies

The option to upload to Polyspace Metrics is available only if you select Run Bug Finder or Code Prover analysis on a remote cluster (`-batch`).

If you perform a local analysis on your desktop, you can later upload your results to Polyspace Metrics. Right-click your results file and select **Upload to Metrics**.

Command-Line Information

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

Default: Off

Example: `polyspace-code-prover-nodesktop -batch -scheduler NodeHost -add-to-results-repository -password passwordName`

Example: `polyspace-bug-finder-nodesktop -batch -scheduler NodeHost -add-to-results-repository -password passwordName`

The password is optional.

See Also

“Set Up Server for Metrics and Remote Analysis” | “Set Up Polyspace Metrics” |

“Generate Code Quality Metrics” | Run Bug Finder or Code Prover analysis on a remote cluster (`-batch`)

Topics

“Run Remote Verification”

“Generate Code Quality Metrics”

Command/script to apply after the end of the code verification (`-post-analysis-command`)

Specify command or script to be executed after analysis

Description

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

Set Option

User interface: In your project configuration, the option is on the **Advanced Settings** node.

Command line: Use the option `-post-analysis-command`. See “Command-Line Information” on page 1-307.


Why Use This Option

Create scripts for tasks that you want performed after the Polyspace analysis.

For instance, you want to be notified by email that the Polyspace analysis is over. Create a script that sends an email and use this option to execute the script after the Polyspace analysis.

Settings

No Default

Enter full path to the command or script, or click  to navigate to the location of the command or script. After the analysis, this script is executed.

For a Perl script, in Windows, specify the full path to the Perl executable followed by the full path to the script. For example, to specify a Perl script `send_email.pl` that sends

an email once the analysis is over, enter `matlabroot\sys\perl\win32\bin\perl.exe <absolute_path>\send_email.pl`. Here, `matlabroot` is the location of the current MATLAB installation, such as `C:\Program Files\MATLAB\R2015b\`, and `<absolute_path>` is the location of the Perl script.

Tips

If you perform verification on a remote server, after verification, the software executes your command on the server, not on the client desktop. If your command executes a script, the script must be present on the server.

For instance, if you specify the command, `/local/utils/send_mail.sh`, the Shell script `send_email.sh` must be present on the server in `/local/utils/`. The software does not copy the script `send_email.sh` from your desktop to the server before executing the command. If the script is not present on the server, you encounter an error. Sometimes, there are multiple servers that the MATLAB Job Scheduler can run the verification on. Place the script on each of the servers because you do not control which server eventually runs your verification.

Command-Line Information

Parameter: `-post-analysis-command`

Value: Path to executable file or command in quotes

No Default

Example in Linux: `polyspace-code-prover-nodesktop -sources file_name -post-analysis-command `pwd`/send_email.pl`

Example in Windows: `polyspace-code-prover-nodesktop -sources file_name -post-analysis-command "C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\send_email.pl"`

See Also

Command/script to apply to preprocessed files (`-post-preprocessing-command`)

Topics

“Specify Analysis Options”

Automatic Orange Tester (`-automatic-orange-tester`)

Specify that Automatic Orange Tester must be executed after verification

Description

This option affects a Code Prover analysis only.

Specify that the Automatic Orange Tester must be executed at the end of the verification.

Set Option

User interface: In your project configuration, the option is on the **Advanced Settings** node. See “Dependency” on page 1-310 for other options you must also enable.

Command line: Use the option `-automatic-orange-tester`. See “Command-Line Information” on page 1-310.

Why Use This Option

The Automatic Orange Tester runs dynamic tests on your code. The dynamic tests help you determine if an orange check represents a real run-time error or an imprecision of Polyspace analysis. For a tutorial, see “Test Orange Checks for Run-Time Errors”.

To run the Automatic Orange Tester after verification, you must select this option *before verification*. During verification, Polyspace generates additional source code to test each orange check for errors. When you run the Automatic Orange Tester later, the software uses this instrumented code for testing.

Settings

On

After verification, when you run the Automatic Orange Tester, Polyspace creates tests for unproven code and runs them.

Off (default)

You cannot launch the Automatic Orange Tester after verification.

Dependency

This option is available only if you set `Source code language (-lang)` to `C`.

Tips

- To launch the Automatic Orange Tester, after verification, open your results. Select **Tools > Automatic Orange Tester**.
- When using the automatic orange tester, you cannot:
 - Select **Division round down** under **Target & Compiler**.
 - Select the options `c18`, `tms320c3c`, `x86_64` or `sharc21x61` for **Target & Compiler > Target processor type**.
 - Specify the type `char` as 16-bit or `short` as 8-bit using the option `mcpu...` (Advanced) for **Target & Compiler > Target processor type**. For the same option, you must specify the type `pointer` as 32-bit.
 - Specify global asserts in the code, having the form `Pst_Global_Assert(A,B)`. In global assert mode, you cannot use **Constraint setup** under **Inputs & Stubbing**.
 - Select these options related to floating-point verification: **Subnormal detection mode** and **Consider non finite floats**.

Command-Line Information

Parameter: `-automatic-orange-tester`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -lang c -automatic-orange-tester`

See Also

Number of automatic tests (-automatic-orange-tester-tests-number) |
Maximum loop iterations (-automatic-orange-tester-loop-max-
iteration) | Maximum test time (-automatic-orange-tester-timeout)

Topics

“Specify Analysis Options”

“Test Orange Checks for Run-Time Errors”

“Limitations of Automatic Orange Tester”

Number of automatic tests (`-automatic-orange-tester-tests-number`)

Specify number of tests that Automatic Orange Tester must run

Description

This option affects a Code Prover analysis only.

Specify number of tests that you want the Automatic Orange Tester to run. The more the number of tests, the greater the possibility of finding a run-time error, but longer it takes to complete.

Set Option

User interface: In your project configuration, the option is on the **Advanced Settings** node. See “Dependencies” on page 1-312 for other options you must also enable.

Command line: Use the option `-automatic-orange-tester-tests-number`. See “Command-Line Information” on page 1-313.

Settings

Default: 500

Enter number of tests up to a maximum of 100,000.

Dependencies

This option is enabled only if you set the following options:

- Set Source code language (`-lang`) to C.
- Turn on Automatic Orange Tester (`-automatic-orange-tester`).

Command-Line Information

Parameter: -automatic-orange-tester-tests-number

Value: *positive integer*

Default: 500

Example: polyspace-code-prover-nodesktop -sources *file_name* -lang c -
automatic-orange-tester -automatic-orange-tester-tests-number 500

See Also

Automatic Orange Tester (-automatic-orange-tester) | Maximum loop iterations (-automatic-orange-tester-loop-max-iteration) | Maximum test time (-automatic-orange-tester-timeout)

Topics

“Specify Analysis Options”

“Test Orange Checks for Run-Time Errors”

Maximum loop iterations (`-automatic-orange-tester-loop-max-iteration`)

Specify number of loop iterations after which Automatic Orange Tester considers infinite loop

Description

This option affects a Code Prover analysis only.

Specify number of loop iterations after which the Automatic Orange Tester considers the loop to be infinite. Specifying a large number decreases the possibility of identifying an infinite loop incorrectly, but takes more time to complete.

Set Option

User interface: In your project configuration, the option is on the **Advanced Settings** node. See “Dependencies” on page 1-314 for other options you must also enable.

Command line: Use the option `-automatic-orange-tester-loop-max-iteration`. See “Command-Line Information” on page 1-315.

Settings

Default: 1000

Enter number of loop iterations. The maximum value that the software supports is 1000.

Dependencies

This option is enabled only if you set the following options:

- Set Source code language (`-lang`) to C.

- Turn on Automatic Orange Tester (-automatic-orange-tester).

Command-Line Information

Parameter: -automatic-orange-tester-loop-max-iteration

Value: *positive integer*

Default: 1000

Example: polyspace-code-prover-nodesktop -sources *file_name* -lang c -automatic-orange-tester -automatic-orange-tester-loop-max-iteration 500

See Also

Automatic Orange Tester (-automatic-orange-tester) | Number of automatic tests (-automatic-orange-tester-tests-number) | Maximum test time (-automatic-orange-tester-timeout)

Topics

“Specify Analysis Options”

“Test Orange Checks for Run-Time Errors”

Maximum test time (`-automatic-orange-tester-timeout`)

Specify time in seconds allowed for a single test in Automatic Orange Tester

Description

This option affects a Code Prover analysis only.

Specify time in seconds allowed for a single test. After this time is over, the Automatic Orange Tester proceeds to the next test. Increasing this time reduces number of tests that do not complete, but increases total verification time.

Set Option

User interface: In your project configuration, the option is on the **Advanced Settings** node. See “Dependencies” on page 1-316 for other options you must also enable.

Command line: Use the option `-automatic-orange-tester-timeout`. See “Command-Line Information” on page 1-317.

Settings

Default: 5

Enter time in seconds. The maximum value that the software supports is 60.

Dependencies

This option is enabled only if you set the following options:

- Set Source code language (`-lang`) to C.
- Turn on Automatic Orange Tester (`-automatic-orange-tester`).

Command-Line Information

Parameter: -automatic-orange-tester-timeout

Value: *time*

Default: 5

Example: polyspace-code-prover-nodesktop -sources *file_name* -lang c -automatic-orange-tester -automatic-orange-tester-test-timeout 10

See Also

Automatic Orange Tester (-automatic-orange-tester) | Number of automatic tests (-automatic-orange-tester-tests-number) | Maximum loop iterations (-automatic-orange-tester-loop-max-iteration)

Topics

“Specify Analysis Options”

“Test Orange Checks for Run-Time Errors”

Other

Specify additional flags for analysis

Description

Enter command-line-style flags such as `-max-processes`.

Set Option

In your project configuration, the option is on the **Advanced Settings** node. You can enter multiple options in this field. If you enter the same option multiple times with different arguments, the analysis uses your last argument.

Why Use This Option

Use this option to add nonofficial or command-line only options to the analyzer.

Tip

Nonofficial options: In rare circumstances, to work around very specific issues, MathWorks Technical Support might provide you some undocumented options. If you are running verification from the user interface, you use the **Other** field in the **Configuration** pane to enter the options. Sometimes, the options and their arguments have to be preceded by extra flags. When providing you the option, Technical Support will let you know if the extra flags are required.

Possible Flags: `-extra-flags` | `-c-extra-flags` | `-cpp-extra-flags` | `-cfe-extra-flags` | `-il-extra-flags`

Example: `polyspace-code-prover-nodesktop -extra-flags -option-name -extra-flags option_param`

Polyspace Analysis Options — Command Line Only

-asm-begin -asm-end

Exclude compiler-specific `asm` functions from analysis

Syntax

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

Description

`-asm-begin "mark1[,mark2,...]" -asm-end "mark1[,mark2,...]"` excludes compiler-specific assembly language source code functions from the analysis. You must use these two options together.

Polyspace recognizes most inline assemblers by default. Use the option only if compilation errors occur due to introduction of assembly code. For more information, see “Assembly Code” on page 4-29.

Mark the offending code block by two `#pragma` directives, one at the beginning of the assembly code and one at the end. In the command usage, give these marks in the same order for `-asm-begin` as they are for `-asm-end`.

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

Examples

A block of code is delimited by `#pragma start1` and `#pragma end1`. These names must be in the same order for their respective options. **Either:**

```
-asm-begin "start1" -asm-end "end1"
```

or

```
-asm-begin "mark1,...markN,start1" -asm-end "mark1,...markN,end1"
```


The following example marks two functions for exclusion, `foo_1` and `foo_2`.

Code:

```
#pragma asm_begin_foo
int foo(void) { /* asm code to be ignored by Polyspace */ }
#pragma asm_end_foo

#pragma asm_begin_bar
void bar(void) { /* asm code to be ignored by Polyspace */ }
#pragma asm_end_bar
```

Polyspace Command:

```
polyspace-code-prover-nodesktop -lang c -asm-begin "asm_begin_foo,asm_begin_bar"
    -asm-end "asm_end_foo,asm_end_bar"
```

`asm_begin_foo` and `asm_begin_bar` mark the beginning of the assembly source code sections to be ignored. `asm_end_foo` and `asm_end_bar` mark the end of those respective sections.

See Also

`polyspaceCodeProver`

Topics

“Run Local Verification at Command Line”

-author

Specify project author

Syntax

```
-author "value"
```

Description

`-author "value"` assigns an author to the Polyspace project. The name appears as the project owner in Polyspace Metrics and on generated reports.

The default value is the user name of the current user, given by the DOS or UNIX command `whoami`.

In the Polyspace user interface, select  to specify the Project name, Version, and Author parameters in the **Polyspace Project – Properties** dialog box.

Examples

Assign a project author to your Polyspace Project.

```
polyspace-code-prover-nodesktop -author "John Smith"
```

See Also

```
-date | -prog | polyspaceCodeProver
```

Topics

“Run Local Verification at Command Line”

-date

Specify date of analysis

Syntax

```
-date "date"
```

Description

`-date "date"` specifies the date stamp for the analysis in the format `dd/mm/yyyy`. By default the value is the date the analysis starts.

Examples

Assign a date to your Polyspace Project.

```
polyspace-code-prover-nodesktop -date "15/03/2012"
```

See Also

```
-author | -prog
```

Topics

“Run Local Verification at Command Line”

-function-behavior-specifications

Map imprecisely analyzed function to standard function for precise analysis

Syntax

```
-function-behavior-specifications file_path
```

Description

`-function-behavior-specifications file_path` specifies the path to an XML file. You can use this XML file to map some of your functions to corresponding standard functions that Polyspace recognizes. If you run verification from the command line, *file_path* is the absolute path or path relative to the folder from which you run the command. If you run verification from the user interface, *file_path* is the absolute path.

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

Using Option for Precision Improvement

This section applies only to a Code Prover analysis.

Use this option to reduce the number of orange checks from imprecise analysis of your function. Sometimes, the verification does not analyze certain kinds of functions precisely because of inherent limitations in static verification. In those cases, if you find a standard function that is a close analog of your function, use this mapping. Though your function itself is not analyzed, the analysis is more precise at the locations where you call the function. For instance, if the verification cannot analyze your function `cos32` precisely and considers full range for its return value, map it to the `cos` function for a return value in `[-1,1]`.

The verification ignores the body of your function. However, the verification emulates your function behavior in the following ways:

- The verification assumes the same return values for your function as the standard function.

For instance, if you map your function `cos32` to the standard function `cos`, the verification assumes that `cos32` returns values in `[-1,1]`.

- The verification checks for the same issues as it checks with the standard function.

For instance, if you map your function `acos32` to the standard function `acos`, the Invalid use of standard library routine check determines if the argument of `acos32` is in `[-1,1]`.

A sample file `function-behavior-specifications-sample.xml` shows the functions that you can map to. The file is in `matlabroot\polyspace\verifier\cxx\` where `matlabroot` is the MATLAB installation folder. The functions that you can map to include:

- Standard library functions from `math.h`.
- Memory management functions from `string.h`.
- `__ps_meminit`: A function specific to Polyspace that initializes a memory area.

Sometimes, the verification does not recognize your memory initialization function and produces an orange `Non-initialized local variable` check on a variable that you initialized through this function. If you know that your memory initialization function initializes the variable through its address, map your function to `__ps_meminit`. The check turns green.

- `__ps_lookup_table_clip`: A function specific to Polyspace that returns a value within the range of the input array.

Sometimes, the verification considers full range for the return values of functions that look up values in large arrays (look-up table functions). If you know that the return value of a look-up table function must be within the range of values in its input array, map the function to `__ps_lookup_table_clip`.

In code generated from models, the verification by default makes this assumption for look-up table functions. To identify if the look-up table uses linear interpolation and no extrapolation, the verification uses the function names. See “Stub lookup tables” on page 11-10. Use the mapping only for handwritten functions, for instance, functions in a C/C++ S-Function block. The names of those functions do not follow specific conventions. You must explicitly specify them.

Using Option for Concurrency Detection

This section applies both to a Bug Finder and a Code Prover analysis.

Use this option for automatic detection of thread-creation functions and functions that begin and end critical sections. Polyspace supports automatic detection for certain families of multitasking primitives only. Extend the support using this option.

If your thread-creation function, for instance, does not belong to one of the supported families, map your function to a supported concurrency primitive.

To find which multitasking primitives can be automatically detected, see “Verify Multitasking Applications”.

Examples

Specify Mapping to Standard Function

You can adapt the sample mapping XML file provided with your Polyspace installation and map your function to a standard function.

Suppose the default verification produces an orange `User assertion` check on this code:

```
double x = acos32(1.0) ;  
assert(x <= 2.0);
```

Suppose you know that the function `acos32` behaves like the function `acos` and the return value is 0. You expect the check on the `assert` statement to be green. However, the verification considers that `acos32` returns any value in the range of type `double` because `acos32` is not precisely analyzed. The check is orange. To map your function `acos32` to `acos`:

- 1 Copy the file `function-behavior-specifications-sample.xml` from `matlabroot\polyspace\verifier\cxx\` to another location, for instance, "`C:\Polyspace_projects\Common\Config_files`". Change the write permissions on the file.

- 2 To map your function to a standard function, modify the contents of the XML file. To map your function `acos32` to the standard library function `acos`, change the following code:

```
<function name="my_lib_cos" std="acos"> </function>
```

To:

```
<function name="acos32" std="acos"> </function>
```

- 3 Specify the location of the file for verification.

```
polyspace-code-prover-nodesktop -function-behavior-specifications
"C:\Polyspace_projects\Common\Config_files
\function-behavior-specifications-sample.xml"
```

Specify Mapping to Standard Function with Argument Remapping

Sometimes, the arguments of your function do not map one-to-one with arguments of the standard function. In those cases, remap your function argument to the standard function argument. For instance:

- `__ps_lookup_table_clip`:

This function specific to Polyspace takes only a look-up table array as argument and returns values within the range of the look-up table. Your look-up table function might have additional arguments besides the look-up table array itself. In this case, use argument remapping to specify which argument of your function is the look-up table array.

For instance, suppose a function `my_lookup_table` has the following declaration:

```
double my_lookup_table(double u0, const real_T *table,
                       const double *bp0);
```

The second argument of your function `my_lookup_table` is the look-up table array. In the file `function-behavior-specifications-sample.xml`, add this code:

```
<function name="my_lookup_table" std="__ps_lookup_table_clip">
  <mapping std_arg="1" arg="2"></mapping>
</function>
```

When you call the function:

```
res = my_lookup_table(u, table10, bp);
```

The verification interprets the call as:

```
res = __ps_lookup_table_clip(table10);
```

The verification assumes that the value of `res` lies within the range of values in `table10`.

- `__ps_meminit`:

This function specific to Polyspace takes a memory address as the first argument and a number of bytes as the second argument. The function assumes that the bytes in memory starting from the memory address are initialized with a valid value. Your memory initialization function might have additional arguments. In this case, use argument remapping to specify which argument of your function is the starting address and which argument is the number of bytes.

For instance, suppose a function `my_meminit` has the following declaration:

```
void my_meminit(enum InitKind k, void* dest, int is_aligned,
                unsigned int size);
```

The second argument of your function is the starting address and the fourth argument is the number of bytes. In the file `function-behavior-specifications-sample.xml`, add this code:

```
<function name="my_meminit" std="__ps_meminit">
  <mapping std_arg="1" arg="2"></mapping>
  <mapping std_arg="2" arg="4"></mapping>
</function>
```

When you call the function:

```
my_meminit(INIT_START_BY_END, &buffer, 0, sizeof(buffer));
```

The verification interprets the call as:

```
__ps_meminit(&buffer, sizeof(buffer));
```

The verification assumes that `sizeof(buffer)` number of bytes starting from `&buffer` are initialized.

- `memset`: Variable number of arguments.

If your function has variable number of arguments, you cannot map it directly to a standard function without explicit argument remapping. For instance, say your function is declared as:

```
void* my_memset(void*, int, size_t, ...)
```

To map the function to the `memset` function, use the following mapping:

```
<function name="my_memset" std="memset">
  <mapping std_arg="1" arg="1"></mapping>
  <mapping std_arg="2" arg="2"></mapping>
  <mapping std_arg="3" arg="3"></mapping>
</function>
```

Effect of Mapping on Precision

These examples show the result of mapping certain functions to standard functions:

- `my_acos` → `acos`:

If you use the mapping, the `User assertion` check turns green. The verification assumes that the return value of `my_acos` is 0.

- *Before mapping:*

```
double x = my_acos(1.0);
assert(x <= 2.0);
```

- *Mapping specification:*

```
<function name="my_acos" std="acos">
</function>
```

- *After mapping:*

```
double x = my_acos(1.0);
assert(x <= 2.0);
```

- `my_sqrt` → `sqrt`:

If you use the mapping, the `Invalid use of standard library routine` check turns red. Otherwise, the verification does not check whether the argument of `my_sqrt` is nonnegative.

- *Before mapping:*

```
res = my_sqrt(-1.0);
```

- *Mapping specification:*

```
<function name="my_sqrt" std="sqrt">
</function>
```

- *After mapping:*

```
res = my_sqrt(-1.0);
```

- `my_lookup_table` (argument 2) → `__ps_lookup_table_clip` (argument 1):

If you use the mapping, the User assertion check turns green. The verification assumes that the return value of `my_lookup_table` is within the range of the lookup table array `table`.

- *Before mapping:*

```
double table[3] = {1.1, 2.2, 3.3}
.
.
double res = my_lookup_table(u, table, bp);
assert(res >= 1.1 && res <= 3.3);
```

- *Mapping specification:*

```
<function name="my_lookup_table" std="__ps_lookup_table_clip">
  <mapping std_arg="1" arg="2"></mapping>
</function>
```

- *After mapping:*

```
double table[3] = {1.1, 2.2, 3.3}
.
.
res_real = my_lookup_table(u, table9, bp);
assert(res_real >= 1.1 && res_real <= 3.3);
```

- `my_meminit` → `__ps_meminit`:

If you use the mapping, the Non-initialized local variable check turns green. The verification assumes that all fields of the structure `x` are initialized with valid values.

- *Before mapping:*

```
struct X {
  int field1 ;
```

```

    int field2 ;
};
.
.
struct X x;
my_meminit(&x, sizeof(struct X));
return x.field1;

```

- *Mapping specification:*

```

<function name="my_meminit" std="__ps_meminit">
    <mapping std_arg="1" arg="1"></mapping>
    <mapping std_arg="2" arg="2"></mapping>
</function>

```

- *After mapping:*

```

struct X {
    int field1 ;
    int field2 ;
};
.
.
struct X x;
my_meminit(&x, sizeof(struct X));
return x.field1;

```

- `my_meminit` → `__ps_meminit`:

If you use the mapping, the Non-initialized local variable check turns red. The verification assumes that only the field `field1` of the structure `x` is initialized with valid values.

- *Before mapping:*

```

struct X {
    int field1 ;
    int field2 ;
};
.
.
struct X x;
my_meminit(&x, sizeof(int));
return x.field2;

```

- *Mapping specification:*

```
<function name="my_meminit" std="__ps_meminit">
</function>
```

- *After mapping:*

```
struct X {
    int field1 ;
    int field2 ;
};
.
.
struct X x;
my_meminit(&x, sizeof(int));
return x.field2;
```

Effect of Mapping on Concurrency Detection

In this example, the Polyspace support for automatic concurrency detection is extended by mapping unsupported functions to the supported Pthreads functions.

- Thread creation function: `createTask` → `pthread_create`
- Function that begins critical section: `takeLock` → `pthread_mutex_lock`
- Function that ends critical section: `releaseLock` → `pthread_mutex_unlock`

If you use the mapping, a Bug Finder analysis can determine the multitasking model used in your code and find possible race conditions.

- *Before mapping:*

The analysis does not detect the data race on `var2`.

```
typedef void* (*FUNT) (void*);

extern int takeLock(int* t);
extern int releaseLock(int* t);
// First argument is the function, second the id
extern int createTask(FUNT,int*,int*,void*);

int t_id1,t_id2;
int lock;

int var1;
int var2;
```

```

void* task1(void* a) {
    takeLock(&lock);
    var1++;
    var2++;
    releaseLock(&lock);
    return 0;
}

void* task2(void* a) {
    takeLock(&lock);
    var1++;
    releaseLock(&lock);
    var2++;
    return 0;
}

void main() {
    createTask(task1, &t_id1, 0, 0);
    createTask(task2, &t_id2, 0, 0);
}

```

- *Mapping specification:*

Based on the number and type of parameters of the function `createTask`, it is convenient to map `createTask` to the thread creation function `pthread_create`. The other available alternatives, `createThread` or `OSTaskCreate`, have different argument types.

Even when mapping to `pthread_create`, argument remapping is required, because the arguments do not correspond exactly. The thread start routine is the third argument of `pthread_create` but the first argument of `createTask`.

```

<function name="createTask" std="pthread_create" >
  <mapping std_arg="1" arg="2"></mapping>
  <mapping std_arg="3" arg="1"></mapping>
  <mapping std_arg="2" arg="3"></mapping>
  <mapping std_arg="4" arg="4"></mapping>
</function>
<function name="takeLock" std="pthread_mutex_lock" >
</function>
<function name="releaseLock" std="pthread_mutex_unlock" >
</function>

```

For the list of supported functions that you can map to, see the sample mapping file `function-behavior-specifications-sample.xml` in `matlabroot\polyspace\verifier\cxx\matlabroot` is the MATLAB installation folder, such as `C:\Program Files\MATLAB\R2017b`. See also “Verify Multitasking Applications”.

- *After mapping:*

The analysis detects the data race on `var2`.

```
typedef void* (*FUNT) (void*);

extern int takeLock(int* t);
extern int releaseLock(int* t);
// First argument is the function, second the id
extern int createTask(FUNT,int*,int*,void*);

int t_id1,t_id2;
int lock;

int var1;
int var2;

void* task1(void* a) {
    takeLock(&lock);
    var1++;
    var2++;
    releaseLock(&lock);
    return 0;
}

void* task2(void* a) {
    takeLock(&lock);
    var1++;
    releaseLock(&lock);
    var2++;
    return 0;
}

void main() {
    createTask(task1,&t_id1,0,0);
    createTask(task2,&t_id2,0,0);
}
```

See Also

“Stub lookup tables” on page 11-10

Topics

“Reduce Orange Checks”

Introduced in R2016b

-generate-launching-script-for

Extract information from project file

Syntax

```
-generate-launching-script-for PRJFILE
```

Description

`-generate-launching-script-for PRJFILE` extracts information from the project file *PRJFILE* so that you can run an analysis from the command line. For each module and each configuration in each module, a folder is created containing the following files:

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

Note The script that Polyspace generates runs the same analysis that Polyspace runs from the user interface. If your project runs in the Polyspace Code Prover interface, the script will run from the command line.

Examples

Extract information to run `myproject` from the command line. Use this option with the desktop binary `polyspace-code-prover`.

```
polyspace-code-prover -generate-launching-script-for myproject.psprj
```

See Also

Topics

“Create Command-Line Script from Project File”

“Run Local Verification at Command Line”

-h[elp]

Display list of possible options

Syntax

```
-h  
-help
```

Description

`-h` and `-help` display the list of possible options in the shell window and the argument `syntax`.

Examples

Display the command-line help.

```
polyspace-code-prover-nodesktop -h  
polyspace-code-prover-nodesktop -help
```

See Also

`polyspaceCodeProver`

Topics

“Run Local Verification at Command Line”

-I

Specify include folder for compilation

Syntax

```
-I folder
```

Description

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

The analysis looks for include files relative to the folder paths that you specify. For instance, if your code contains the preprocessor directive `#include<./mylib.h>` and you include the folder:

```
C:\My_Project\MySourceFiles\Includes
```

the folder `C:\My_Project\MySourceFiles` must contain a file `mylib.h`.

The analysis automatically includes the `./sources` folder (if it exists) after the include folders that you specify.

Examples

Include two folders with the analysis.

```
polyspace-code-prover-nodesktop -I /com1/inc -I /com1/sys/inc
```

Because `./sources` is included automatically, this Polyspace command is equivalent to:

```
polyspace-code-prover-nodesktop -I /com1/inc -I /com1/sys/inc  
                                -I ./sources
```

See Also

`polyspaceCodeProver`

Topics

“Run Local Verification at Command Line”

-import-comments

Import comments and justifications from previous analysis

Syntax

```
-import-comments resultsFolder
```

Description

`-import-comments resultsFolder` imports the comments and justifications from a previous analysis, as specified by the results folder. *resultsFolder* must be the same type of analysis you are running. For example, if you are running a file-by-file verification, you can only import comments from a file-by-file verification. Or, if you are running a Bug Finder analysis, you cannot import comments from a Code Prover verification.

If you are running an analysis from the user interface, on the **Configuration** pane, you can enter this option in the **Other** field. See [Other](#).

Examples

Increment your project's version number (`-version`) and import comments from the previous results.

```
polyspace-code-prover-nodesktop -version 1.3  
-import-comments C:\Results\myProj\1.2
```

See Also

```
-version | polyspaceCodeProver
```

Topics

“Run Local Verification at Command Line”

“Import Review Comments from Previous Verifications”

-lightweight-thread-model

Reduce task complexity

Syntax

```
-lightweight-thread-model
```

Description

`-lightweight-thread-model` specifies that the verification must use a slightly less precise model than default for interaction between tasks. Using this option can speed up verification, but you see a greater number of unproven results.

There is a loss of precision when variables shared between tasks are read through pointers.

If you are running an analysis from the user interface, on the **Configuration** pane, you can enter this option in the **Other** field. See [Other](#).

Examples

Define the language of your Polyspace project as C++.

```
polyspace-code-prover-nodesktop -lightweight-thread-model
```

See Also

`Entry points (-entry-points) | polyspaceCodeProver`

Topics

- “Run Local Verification at Command Line”
- “Specify Analysis Options”
- “Reduce Verification Time”

-max-processes

Specify maximum number of processors for analysis

Syntax

```
-max-processes num
```

Description

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

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

The option uses the physical processors available and not the logical processors. For instance, if you have 2 physical cores but 4 logical cores, the option `-max-processes 4` uses the 2 physical cores only. To determine number of physical processors available, check the system information in your operating system.

If you are running an analysis from the user interface, on the **Configuration** pane, you can enter this option in the **Other** field. See [Other](#).

Examples

Disable parallel processing during the analysis.

```
polyspace-code-prover-nodesktop -max-processes 1
```


Tips

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

See Also

`polyspaceCodeProver`

Topics

“Run Local Verification at Command Line”

-no-assumption-on-absolute-addresses

Remove assumption that absolute address usage is valid

Syntax

```
-no-assumption-on-absolute-addresses
```

Description

This option affects Code Prover analysis only.

`-no-assumption-on-absolute-addresses` removes the default assumption that absolute addresses used in your code are valid. If you use this option, the verification produces an orange `Absolute address usage` check when you assign an absolute address to a pointer. Otherwise, the check is green by default.

The type of the pointer to which you assign the address determines the initial value stored in the address. For instance, if you assign the address to an `int*` pointer, following this check, the verification assumes that the memory zone that the address points to is initialized with an `int` value. The value can be anything allowed for the data type `int`.

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

Examples

The use of option `-no-assumption-on-absolute-addresses` can increase the number of orange checks in your code. For instance, the following table shows an additional orange check with the option enabled.

Absolute Address Usage Green	Absolute Address Usage Orange
<pre data-bbox="244 309 734 442">void main() { int *p = (int *)0x32; int x; x=*p; }</pre> <p data-bbox="244 473 734 499">In this example, the software produces:</p> <ul data-bbox="244 531 734 734" style="list-style-type: none"> • A green Absolute address usage check when the address 0x32 is assigned to a pointer p. • A green Illegally dereferenced pointer check when the pointer p is read. <p data-bbox="244 765 734 826">x potentially has all values allowed for an int variable.</p>	<pre data-bbox="798 309 1288 442">void main() { int *p = (int *)0x32; int x; x=*p; }</pre> <p data-bbox="798 473 1288 499">In this example, the software produces:</p> <ul data-bbox="798 531 1288 734" style="list-style-type: none"> • An orange Absolute address usage check when the address 0x32 is assigned to a pointer p. • A green Illegally dereferenced pointer check when the pointer p is read. <p data-bbox="798 765 1288 826">x potentially has all values allowed for an int variable.</p>

For best use of the **Absolute address usage** check, leave this check green by default during initial stages of development. During integration stage, use the option `-no-assumption-on-absolute-addresses` and detect all uses of absolute memory addresses. Browse through them and make sure that the addresses are valid.

See Also

`polyspaceCodeProver`

Topics

“Run Local Verification at Command Line”

Introduced in R2016a

-non-preemptable-tasks

Specify functions that represent nonpreemptable tasks

Syntax

```
-non-preemptable-tasks function1 [, function2 [, ...]]
```

Description

This option affects a Bug Finder analysis only.

`-non-preemptable-tasks function1 [, function2 [, ...]]` specifies functions that represent nonpreemptable tasks.

The functions cannot be interrupted by other noncyclic entry points on page 1-116 and cyclic tasks on page 1-119 but can be interrupted by interrupts on page 1-122, preemptable or nonpreemptable.

To specify a function as a nonpreemptable cyclic task, you must first specify the following options:

- `Configure multitasking manually`
- `Entry points (-entry-points)` or `Cyclic tasks (-cyclic-tasks)`: Specify the function name.

The functions that you specify must have the prototype:

```
void function_name(void);
```

If you are running an analysis from the user interface, on the **Configuration** pane, you can enter this option in the **Other** field. See [Other](#).

See Also

`-preemptable-interrupts` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)` | `Entry points (-entry-points)` | `Critical section details`

`(-critical-section-begin -critical-section-end) | Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

“Specify Analysis Options”

“Verify Multitasking Applications”

“Manually Model Tasks if main Contains Infinite Loop”

“Manually Model Scheduling of Tasks”

Introduced in R2016b

-preemptable-interrupts

Specify functions that represent preemptable interrupts

Syntax

```
-preemptable-interrupts function1 [, function2 [, ...]]
```

Description

This option affects a Bug Finder analysis only.

`-preemptable-interrupts function1 [, function2 [, ...]]` specifies functions that represent preemptable interrupts.

The function acts as an interrupt in every way except that it can be interrupted by other interrupts on page 1-122, preemptable or nonpreemptable.

To specify a function as a preemptable interrupt, you must first specify the following options:

- Configure multitasking manually
- Interrupts (`-interrupts`): Specify the function name.

The functions that you specify must have the prototype:

```
void function_name(void);
```

If you are running an analysis from the user interface, on the **Configuration** pane, you can enter this option in the **Other** field. See [Other](#).

See Also

`-non-preemptable-tasks` | Cyclic tasks (`-cyclic-tasks`) | Interrupts (`-interrupts`) | Entry points (`-entry-points`) | Critical section details (`-critical-section-begin` `-critical-section-end`) | Temporally exclusive tasks (`-temporal-exclusions-file`)

Topics

“Specify Analysis Options”

“Verify Multitasking Applications”

“Manually Model Tasks if main Contains Infinite Loop”

“Manually Model Scheduling of Tasks”

Introduced in R2016b

-options-file

Run Polyspace using list of options

Syntax

```
-options-file file
```

Description

`-options-file file` specifies a file which lists your analysis options. The file must be a text file with each option on a separate line. Use # to add comments to this file.

Examples

- 1 Create an options file called `listofoptions.txt` with your options. For example:

```
#These are the options for MyCodeProverProject
-lang c
-prog MyCodeProverProject
-author jsmith
-sources "mymain.c,funAlgebra.c,funGeometry.c"
-OS-target no-predefined-OS
-target x86_64
-compiler generic
-dos
-misra2 required-rules
-do-not-generate-results-for all-headers
-main-generator
-results-dir C:\Polyspace\MyCodeProverProject
```

- 2 Run Polyspace using options in the file `listofoptions.txt`.

```
polyspace-code-prover-nodesktop -options-file listofoptions.txt
```


See Also

`polyspaceCodeProver` | `polyspaceConfigure`

Topics

“Run Local Verification at Command Line”

-prog

Specify name of project

Syntax

```
-prog projectName
```

Description

`-prog projectName` specifies the name of your Polyspace project. This name must use only letters, numbers, underscores (`_`), dashes (`-`), or periods (`.`).

Examples

Assign a session name to your Polyspace Project.

```
polyspace-code-prover-nodesktop -prog MyApp
```

See Also

```
-author | -date | polyspaceCodeProver
```

Topics

“Run Local Verification at Command Line”

-report-output-name

Specify name of report

Syntax

```
-report-output-name reportName
```

Description

`-report-output-name reportName` specifies the name of an analysis report.

The default name for a report is *Prog_Template.Format*:

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

If you are running an analysis from the user interface, on the **Configuration** pane, you can enter this option in the **Other** field. See [Other](#).

Examples

Specify the name of the analysis report.

```
polyspace-code-prover-nodesktop -report-template Developer  
-report-output-name Airbag_v3.doc
```

See Also

[Output format \(-report-output-format\) | Bug Finder and Code Prover report \(-report-template\) | polyspaceCodeProver](#)

Topics

“Run Local Verification at Command Line”

“Generate Report”

-results-dir

Specify the results folder

Syntax

```
-results-dir
```

Description

`-results-dir` specifies where to save the analysis results. The default location at the command line is the current folder.

If you are running analysis in the user interface, see “Specify Results Folder”.

Examples

Specify to store your results in the RESULTS folder.

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

See Also

`polyspaceCodeProver`

Topics

“Run Local Verification at Command Line”

-scheduler

Specify cluster or job scheduler

Syntax

```
-scheduler schedulingOption
```

Description

`-scheduler schedulingOption` specifies the head node of the cluster or MATLAB job scheduler on the node host. Use this command to manage the cluster, or to specify where to run batch analyses.

Examples

Run a batch analysis on a remote server.

```
polyspace-code-prover-nodesktop -batch -scheduler NodeHost  
polyspace-code-prover-nodesktop -batch -scheduler 192.168.1.124:12400  
polyspace-code-prover-nodesktop -batch -scheduler MJSName@NodeHost
```

```
polyspace-job-manager listjobs -scheduler NodeHost
```

See Also

`polyspaceCodeProver` | `polyspaceJobsManager` | `polyspaceJobsManager`

Topics

“Run Remote Analysis at the Command Line”

-sources

Specify source files

Syntax

```
-sources file1[,file2,...]
-sources file1 -sources file2
```

Description

`-sources file1[,file2,...]` or `-sources file1 -sources file2` specifies the list of source files that you want to analyze. You can use standard UNIX wildcards with this option to specify your sources.

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

Examples

Analyze the files `mymain.c`, `funAlgebra.c`, and `funGeometry.c`.

```
polyspace-code-prover-nodesktop -sources mymain.c
-sources funAlgebra.c -sources funGeometry.c
```

See Also

`polyspaceCodeProver`

Topics

“Run Local Verification at Command Line”

-sources-list-file

Specify file containing list of sources

Syntax

```
-sources-list-file file_path
```

Description

`-sources-list-file file_path` specifies the absolute path to a text file that lists each file name that you want to analyze.

To specify your sources in the text file, on each line, specify the absolute path to a source file. For example:

```
C:\Sources\myfile.c  
C:\Sources2\myfile2.c
```

This option is available only in batch analysis mode.

Examples

Run analysis on files listed in `files.txt`.

```
polyspace-code-prover-nodesktop -batch -scheduler NODEHOST  
    -sources-list-file "C:\Analysis\files.txt"  
polyspace-code-prover-nodesktop -batch -scheduler NODEHOST  
    -sources-list-file "/home/polyspace/files.txt"
```

See Also

`polyspaceCodeProver`

Topics

“Run Remote Analysis at the Command Line”

-submit-job-from-previous-compilation-results

Specify that the analysis job must be resubmitted without recompilation

Syntax

```
-submit-job-from-previous-compilation-results
```

Description

`-submit-job-from-previous-compilation-results` specifies that the Polyspace analysis must start after the compilation phase with compilation results from a previous analysis. If a remote analysis stops after compilation, for instance because of communication problems between the server and client computers, use this option.

When you perform a remote analysis:

- 1 On the local host computer, the Polyspace software performs code compilation and coding rule checking.
- 2 The Parallel Computing Toolbox™ software submits the analysis job to the MATLAB job scheduler (MJS) on the head node of the MATLAB Distributed Computing Server cluster.
- 3 The head node of the MATLAB Distributed Computing Server cluster assigns the verification job to a worker node, where the remaining phases of the Polyspace analysis occur.

If an analysis stops after completing the first step and you restart the analysis, use this option to reuse compilation results from the previous analysis. You thereby avoid restarting the analysis from the compilation phase.

If previous compilation results do not exist in the current folder, an error occurs. Remove the option and restart analysis from the compilation phase.

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

Examples

Specify remote analysis with compilation results from a previous analysis.

```
polyspace-code-prover-nodesktop -batch -scheduler localhost  
-submit-job-from-previous-compilation-results
```

See Also

`polyspaceCodeProver`

Topics

“Run Remote Verification”

“Run Remote Analysis at the Command Line”

-tmp-dir-in-results-dir

Keep temporary files in results folder

Syntax

```
-tmp-dir-in-results-dir
```

Description

`-tmp-dir-in-results-dir` specifies that temporary files must be stored in a subfolder of the results folder. Use this option only when the standard temporary folder does not have enough disk space. If the results folder is mounted on a network drive, this option can slow down your processor.

To learn how Polyspace determines the temporary folder location, see “Storage of Temporary Files”.

If you are running an analysis from the user interface, on the **Configuration** pane, you can enter this option in the **Other** field. See [Other](#).

Examples

Store temporary files in the results folder.

```
polyspace-code-prover-nodesktop -tmp-dir-in-results-dir
```

See Also

```
polyspaceCodeProver
```

Topics

“Run Local Verification at Command Line”

-v[ersion]

Display Polyspace version number

Syntax

`-v`
`-version`

Description

`-v` or `-version` displays the version number of your Polyspace product.

Examples

Display the version number and release of your Polyspace product.

```
polyspace-code-prover-nodesktop -v
```

See Also

`polyspaceCodeProver`

Topics

“Run Local Verification at Command Line”

-verif-version

Assign a version identifier

Syntax

```
-verif-version id
```

Description

`-verif-version id` assigns a verification identifier, *id*, to identify the verification. You can use this identifier to refer to different verifications at the command line. For example, you can import comments from a previous verification using the verification identifier.

Examples

Assign a verification identifier.

```
polyspace-code-prover-nodesktop -verif-version 1.3
```

See Also

`polyspaceCodeProver`

Topics

“Run Local Verification at Command Line”

-xml-annotations-description

Apply custom code annotations to Polyspace analysis results

Syntax

```
-xml-annotations-description file_path
```

Description

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

If you are running an analysis through the user interface, you can enter this option in the **Other** field, under the **Advanced Settings** node on the **Configuration** pane. See [Other](#).

Why Use This Option

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

Examples

Import Existing Annotations for Coding Rule Violations

Suppose that you have previously reviewed source file `zero_div.c` containing the following code, and justified certain MISRA C: 2012 violations by using custom annotations.

```
#include <stdio.h>

/* Violation of Misra C:2012
rules 8.4 and 8.7 on the next
line of code. */

int func(int p) //My_rule 50, 51
{
    int i;
    int j = 1;

    i = 1024 / (j - p);
    return i;
}

/* Violation of Misra C:2012
rule 8.4 on the next line of
code */

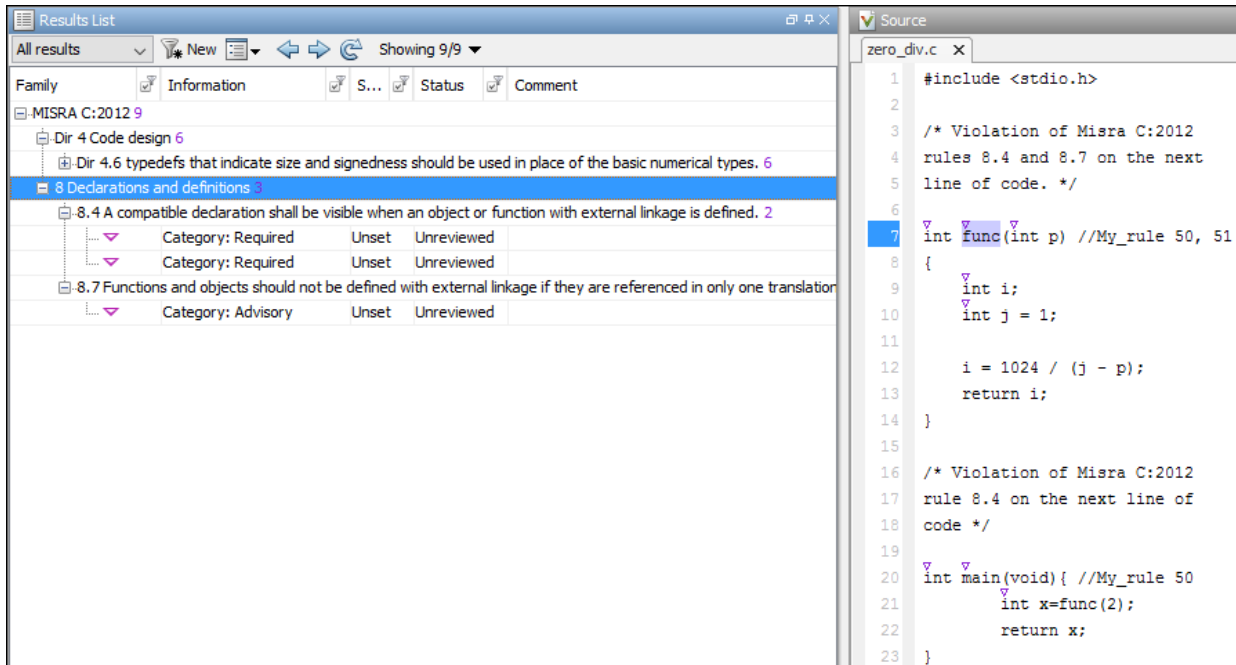
int main(void){ //My_rule 50
    int x=func(2);
    return x;
}
```

The code comments **My_rule 50, 51** and **My_rule 50** do not use the Polyspace annotation syntax. Instead, you use a convention where you place all MISRA rules in a single numbered list. In this list, rules 8.4 and 8.7 correspond to the numbers 50 and 51. You can check this code for MISRA C: 2012 violations by typing the command:

```
polyspace-code-prover-nodesktop -sources source_path -misra3 all
```

source_path is the path to `zero_div.c`.

The annotated violations appear in the **Results List** pane. You must review and justify them again.



This XML example defines the annotation format used in `zero_div.c` and maps it to the Polyspace annotation syntax:

- The format of the annotation is the keyword `My_rule`, followed by a space and one or more comma-separated alphanumeric rule identifiers.
- Rule identifiers 50 and 51 are mapped to MISRA C: 2012 rules 8.4 and 8.7 respectively. The mapping uses the Polyspace annotation syntax.

```
<?xml version="1.0" encoding="UTF-8"?>

<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
  Group="example annotation">

  <Expressions Search_For_Keywords="My_rule"
    Separator_Result_Name=", " >

    <!-- This section defines the annotation syntax format -->
    <Expression Mode="SAME_LINE"
      Regex="My_rule\s(\w+(\s*,\s*\w+)*)"
      Rule_Identifier_Position="1"
    />

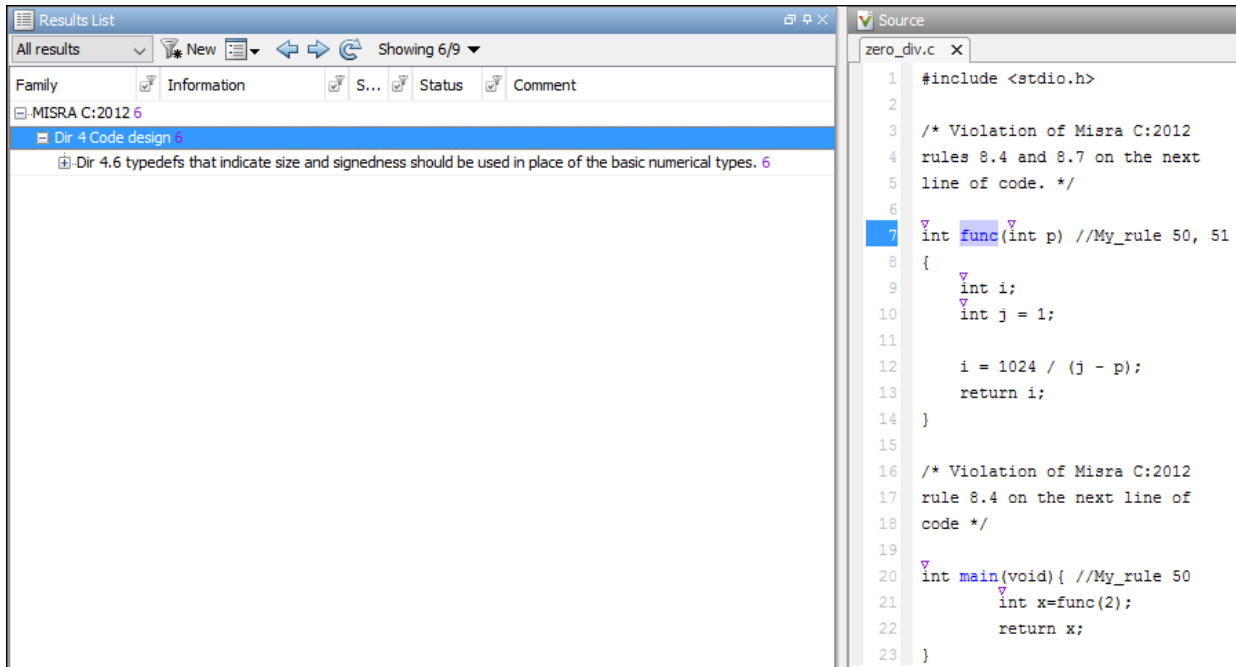
  </Expressions>
  <!-- This section maps the user annotation to the Polyspace
  annotation syntax -->
  <Mapping>
    <Result_Name_Mapping Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
    <Result_Name_Mapping Rule_Identifier="51" Family="MISRA-C3" Result_Name="8.7"/>
  </Mapping>
</Annotations>
```

To import the existing annotations and apply them to the corresponding Polyspace results:

- 1 Copy the preceding code example to a text editor and save it on your machine as `annotations_description.xml`, for instance in `C:\Polyspace_workspace\annotations\`.
- 2 Rerun the analysis on `zero_div.c` by using the command:

```
polyspace-code-prover-nodesktop -sources source_path -misra3 all ^
  -xml-annotations-descriptions ^
  C:\Polyspace_workspace\annotations\annotations_description.xml
```

Polyspace considers the annotated results justified and hides them in the **Results List** pane.



See Also

Topics

“Define Custom Annotation Format”

“Justify Results Through Code Annotations”

Introduced in R2017b

Functions — Alphabetical List

Absolute address usage

Absolute address is assigned to pointer

Description

This check appears when an absolute address is assigned to a pointer.

By default, this check is green. The software assumes the following about the absolute address:

- The address is valid.
- The type of the pointer to which you assign the address determines the initial value stored in the address.

If you assign the address to an `int*` pointer, the memory zone that the address points to is initialized with an `int` value. The value can be anything allowed for the data type `int`.

To turn this check orange by default for each absolute address usage, use the command-line option `-no-assumption-on-absolute-addresses`.

Diagnosing This Check

“Review and Fix Absolute Address Usage Checks”

Examples

Reading content of absolute address

```
enum typeList {CHAR,INT,LONG};
enum typeList showType(void);
long int returnLong(void);

void main() {
```

```

int *p = (int *)0x32; //Green absolute address usage
enum typeList myType = showType();

char x_char;
int x_int;
long int x_long;

if(myType == CHAR)
    x_char = *p;
else if(myType == INT)
    x_int = *p;
else {
    x_long = *p;
    long int x2_long = returnLong();
}
}

```

In this example, the option `-no-assumption-on-absolute-addresses` is not used. Therefore, the **Absolute address usage** check is green when the pointer `p` is assigned an absolute address.

Following this check, the verification assumes that the address is initialized with an `int` value. If you use `x86_64` for Target processor type (`-target`) (`sizeof(char) < sizeof(int) < sizeof(long int)`), the assumption results in the following:

- In the `if(myType == CHAR)` branch, an orange **Overflow** occurs because `x_char` cannot accommodate all values allowed for an `int` variable.
- In the `else if(myType == INT)` branch, if you place your cursor on `x_int` in your verification results, the tooltip shows that `x_int` potentially has all values allowed for an `int` variable.
- In the `else` branch, if you place your cursor on `x_long`, the tooltip shows that `x_long` potentially has all values allowed for an `int` variable. If you place your cursor on `x2_long`, the tooltip shows that `x2_long` potentially has all values allowed for a `long int` variable. The range of values that `x2_long` can take is wider than the values allowed for an `int` variable in the same target.

Arithmetic on pointers with absolute address

```

void main() {
    int *p = (int *)0x32;
    int x = *p;
}

```

```
    p++;  
    x = *p;  
}
```

In this example, the option `-no-assumption-on-absolute-addresses` is used. The **Absolute address usage** check is orange when the pointer `p` is assigned an absolute address.

Following this check:

- Polyspace considers that `p` points to a valid memory location. Therefore the **Illegally dereferenced pointer** check on the following line is green.
- In the next two lines, the pointer `p` is incremented and then dereferenced. In this case, an **Illegally dereferenced pointer** check appears on the dereference and not an **Absolute address usage** check even though `p` still points to an absolute address.

Check Information

Group: Static memory

Language: C | C++

Acronym: ABS_ADDR

Correctness condition

Mismatch occurs during pointer cast or function pointer use

Description

This check determines whether:

- An array is mapped to a larger array through a pointer cast
- A function pointer points to a function with a valid prototype
- A global variable falls outside the range specified through the **Global Assert** mode.

Diagnosing This Check

“Review and Fix Correctness Condition Checks”

Examples

Array is mapped to larger array

```
typedef int smallArray[10];
typedef int largeArray[100];

void main(void) {
    largeArray myLargeArray;
    smallArray *smallArrayPtr = (smallArray*) &myLargeArray;
    largeArray *largeArrayPtr = (largeArray*) smallArrayPtr;
}
```

In this example:

- In the first pointer cast, a pointer of type `largeArray` is cast to a pointer of type `smallArray`. Because the data type `smallArray` represents a smaller array, the **Correctness condition** check is green.

- In the second pointer cast, a pointer of type `smallArray` is cast to a pointer of type `largeArray`. Because the data type `largeArray` represents a larger array, the **Correctness condition** check is red.

Function pointer does not point to function

```
typedef void (*callBack) (float data);
typedef struct {
    char funcName[20];
    callBack func;
} funcStruct;

funcStruct myFuncStruct;

void main(void) {
    float val = 0.f;
    myFuncStruct.func(val);
}
```

In this example, the global variable `myFuncStruct` is not initialized, so the function pointer `myFuncStruct.func` contains `NULL`. When the pointer `myFuncStruct.func` is dereferenced, the **Correctness condition** check produces a red error.

Function pointer points to function through absolute address usage

```
#define MAX_MEMSEG 32764
typedef void (*ptrFunc)(int memseg);
ptrFunc operation = (ptrFunc)(0x003c);

void main(void) {
    for (int i=1; i <= MAX_MEMSEG; i++)
        operation(i);
}
```

In this example, the function pointer `operation` is cast to the contents of a memory location. Polyspace cannot determine whether the location contains a variable or a function code and whether the function is well-typed. Therefore, when the pointer `operation` is dereferenced and used in a function call, the **Correctness condition** check is orange.

After an orange **Correctness condition** check due to absolute address usage, the software assumes that the following variables have the full range of values allowed by their type:

- Variable storing the return value from the function call.

In the following example, the software assumes that the return value of `operation` is full-range.

```
typedef int (*ptrFunc)(int);
ptrFunc operation = (ptrFunc)(0x003c);

int main(void) {
    return operation(0);
}
```

- Variables that can be modified through the function arguments.

In the following example, the function pointer `operation` takes a pointer argument `ptr` that points to a variable `var`. After the call to `operation`, the software assumes that `var` has full-range value.

```
typedef void (*ptrFunc)(int*);
ptrFunc operation = (ptrFunc)(0x003c);

void main(void) {
    int var;
    int *ptr=&var;
    operation(ptr);
}
```

Function pointer points to function with wrong argument type

```
typedef struct {
    double real;
    double imag;
} complex;

typedef int (*typeFuncPtr) (complex*);

int func(int* x);

void main() {
    typeFuncPtr funcPtr = (typeFuncPtr)&func;
}
```

```
    int arg = 0, result = funcPtr((complex*)&arg);
}
```

In this example, the function pointer `funcPtr` points to a function with argument type `complex*`. However, the pointer is assigned the address of function `func` whose argument type is `int*`. Because of this type mismatch, the **Correctness condition** check is orange.

Function pointer points to function with wrong number of arguments

```
typedef int (*typeFuncPtr) (int, int);

int func(int);

void main() {
    typeFuncPtr funcPtr = (typeFuncPtr)&func;
    int arg1 = 0, arg2 = 0, result = funcPtr(arg1, arg2);
}
```

In this example, the function pointer `funcPtr` points to a function with two `int` arguments. However, it is assigned the function `func` which has one `int` argument only. Because of this mismatch in number of arguments, the **Correctness condition** check is orange.

Function pointer points to function with wrong return type

```
typedef double (*typeFuncPtr) (int);

int func(int);

void main() {
    typeFuncPtr funcPtr = (typeFuncPtr)&func;
    int arg = 0;
    double result = funcPtr(arg);
}
```

In this example, the function pointer `funcPtr` points to a function with return type `double`. However, it is assigned the function `func` whose return type is `int`. Because of this mismatch in return types, the **Correctness condition** check is orange.

Variable falls outside Global Assert range

```
int glob = 0;
int func();

void main() {
    glob = 5;
    glob = func();
    glob+= 20;
}
```

In this example, a range of 0..10 was specified for the global variable `glob`.

- In the statement `glob=5;`, a green **Correctness condition** check appears on `glob`.
- In the statement `glob=func();`, an orange **Correctness condition** check appears on `glob` because the return value of stubbed function `func` can be outside 0..10.

After this statement, Polyspace considers that `glob` has values in 0..10.

- In the statement `glob+=20;`, a red **Correctness condition** check appears on `glob` because after the addition, `glob` has values in 20..30.

Check Information

Group: Other

Language: C | C++

Acronym: COR

See Also

Constraint setup (-data-range-specifications) | Permissive function pointer calls (-permissive-function-pointer)

Topics

“Constrain Global Variable Range”

Division by zero

Division by zero occurs

Description

This check determines whether the right operand of a division or modulus operation is zero.

Diagnosing This Check

“Review and Fix Division by Zero Checks”

Examples

Red integer division by zero

```
#include <stdio.h>

void main() {
    int x=2;
    printf("Quotient=%d",100/(x-2));
}
```

In this example, the denominator $x-2$ is zero.

One possible correction is to check for a zero denominator before division.

In a complex code, it is difficult to keep track of values and avoid zero denominators. Therefore, it is good practice to check for zero denominator before every division.

```
#include <stdio.h>
int input();
void main() {
    int x=input();
```

```

    if(x>0) { //Avoid overflow
        if(x!=2 && x>0)
            printf("Quotient=%d",100/(x-2));
        else
            printf("Zero denominator.");
    }
}

```

Red integer division by zero after `for` loop

```

#include <stdio.h>
void main() {
    int x=-10;
    for (int i=0; i<10; i++)
        x+=3;
    printf("Quotient=%d",100/(x-20));
}

```

In this example, the denominator `x-20` is zero.

One possible correction is to check for a zero denominator before division.

After several iterations of a `for` loop, it is difficult to keep track of values and avoid zero denominators. Therefore, it is good practice to check for zero denominator before every division.

```

#include <stdio.h>
#define MAX 10000
int input();

void main() {
    int x=input();
    for (int i=0; i<10; i++) {
        if(x < MAX) //Avoid overflow
            x+=3;
    }

    if(x>0) { //Avoid overflow
        if(x!=20)
            printf("Quotient=%d",100/(x-20));
        else
            printf("Zero denominator.");
    }
}

```

```
    }  
}
```

Orange integer division by zero inside `for` loop

```
#include<stdio.h>  
  
void main() {  
    printf("Sequence of ratios: \n");  
    for(int count=-100; count<=100; count++)  
        printf(" %.2f ", 1/count);  
}
```

In this example, `count` runs from -100 to 100 through zero. When `count` is zero, the **Division by zero** check returns a red error. Because the check returns green in the other `for` loop runs, the `/` symbol is orange.

There is also a red **Non-terminating loop** error on the `for` loop. This red error indicates a definite error in one of the loop runs.

One possible correction is to check for a zero denominator before division.

```
#include<stdio.h>  
  
void main() {  
    printf("Sequence of ratios: \n");  
    for(int count=-100; count<=100; count++) {  
        if(count != 0)  
            printf(" %.2f ", 1/count);  
        else  
            printf(" Infinite ");  
    }  
}
```

Orange float division by zero inside `for` loop

```
#include <stdio.h>  
#include <math.h>  
  
#define stepSize 0.1  
  
void main() {
```



```

float divisor = -1.0;
int numberOfSteps = (int)((2.0*1.0)/stepSize);

printf("Divisor running from -1.0 to 1.0\n");
for(int count = 1; count <= numberOfSteps; count++) {
    divisor+= stepSize;
divisor = ceil(divisor * 10.) / 10.; // one digit of imprecision
    printf(" .2f ", 1.0/divisor);
}
}

```

In this example, divisor runs from -1.0 to 1.0 through 0.0 . When divisor is 0.0 , the **Division by zero** check returns a red error. Because the check returns green in the other for loop runs, the `/` symbol is orange.

There is no red **Non-terminating loop** error on the for loop. The red error does not appear because Polyspace approximates the values of divisor by a broader range. Therefore, Polyspace cannot determine if there is a definite error in one of the loop runs.

One possible correction is to check for a zero denominator before division. For float variables, do not check if the denominator is exactly zero. Instead, check whether the denominator is in a narrow range around zero.

```

#include <stdio.h>
#include <math.h>

#define stepSize 0.1

void main() {
    float divisor = -1.0;
    int numberOfSteps = (int)((2*1.0)/stepSize);

    printf("Divisor running from -1.0 to 1.0\n");
    for(int count = 1; count <= numberOfSteps; count++) {
        divisor += stepSize;
divisor = ceil(divisor * 10.) / 10.; // one digit of imprecision
        if(divisor < -0.00001 || divisor > 0.00001)
            printf(" .2f ", 1.0/divisor);
        else
            printf(" Infinite ");
    }
}

```

Check Information

Group: Numerical

Language: C | C++

Acronym: ZDV

See Also

Consider non finite floats (-allow-non-finite-floats)

Function not called

Function is defined but not called

Description

This check on a function definition determines if the function is called anywhere in the code. This check is disabled if your code does not contain a `main` function.

Use this check to satisfy ISO 26262 requirements about function coverage. For example, see table 15 of ISO 26262, part 6.

Note This check is not turned on by default. To turn on this check, you must specify the appropriate analysis option. For more information, see `Detect uncalled functions (-uncalled-function-checks)`.

Diagnosing This Check

“Review and Fix Function Not Called Checks”

Examples

Function not called

```
#define max 100
int var;
int getValue(void);
int getSaturation(void);

void reset() {
    var=0;
}
```

```
void main() {
    int saturation = getSaturation(),val;
    for(int index=1; index<=max; index++) {
        val = getValue();
        if(val>0 && val<10)
            var += val;
        if(var > saturation)
            var=0;
    }
}
```

In this example, the function `reset` is defined but not called. Therefore, a gray **Function not called** check appears on the definition of `reset`.

One possible correction is to call the function `reset`. In this example, the function call `reset` serves the same purpose as instruction `var=0;`. Therefore, replace the instruction with the function call.

```
#define max 100

int var;
int getValue(void);
int getSaturation(void);

void reset() {
    var=0;
}

void main() {
    int saturation = getSaturation(),val;
    for(int index=1; index<=max; index++) {
        val = getValue();
        if(val>0 && val<10)
            var += val;
        if(var > saturation)
            reset();
    }
}
```

Function Called from Another Uncalled Function

```
#define max 100

int var;
int numberOfResets;
int getValue();
int getSaturation();

void updateCounter() {
    numberOfResets++;
}

void reset() {
    updateCounter();
    var=0;
}

void main() {
    int saturation = getSaturation(),val;
    for(int index=1; index<=max; index++) {
        val = getValue();
        if(val>0 && val<10)
            var += val;
        if(var > saturation) {
            numberOfResets++;
            var=0;
        }
    }
}
```

In this example, the function `reset` is defined but not called. Since the function `updateCounter` is called only from `reset`, a gray **Function not called** error appears on the definition of `updateCounter`.

One possible correction is to call the function `reset`. In this example, the function call `reset` serves the same purpose as the instructions in the branch of `if (var > saturation)`. Therefore, replace the instructions with the function call.

```
#define max 100
```

```
int var;
int numberOfResets;
int getValue(void);
int getSaturation(void);

void updateCounter() {
    numberOfResets++;
}

void reset() {
    updateCounter();
    var=0;
}

void main() {
    int saturation = getSaturation(),val;
    for(int index=1; index<=max; index++) {
        val = getValue();
        if(val>0 && val<10)
            var += val;
        if(var > saturation)
            reset();
    }
}
```

Check Information

Group: Data flow

Language: C | C++

Acronym: FNC

See Also

Detect uncalled functions (-uncalled-function-checks) | Function not reachable

Function not reachable

Function is called from unreachable part of code

Description

This check appears on a function definition. The check appears gray if the function is called only from an unreachable part of the code. The unreachable code can occur in one of the following ways:

- The code is reached through a condition that is always false.
- The code follows a `break` or `return` statement.
- The code follows a red check.

If your code does not contain a `main` function, this check is disabled

Note This check is not turned on by default. To turn on this check, you must specify the appropriate analysis option. For more information, see `Detect uncalled functions (-uncalled-function-checks)`.

Diagnosing This Check

“Review and Fix Function Not Reachable Checks”

Examples

Function Call from Unreachable Branch of Condition

```
#include<stdio.h>
#define SIZE 100

void increase(int* arr, int index);
```

```
void printError()
{
    printf("Array index exceeds array size.");
}

void main() {
    int arr[SIZE],i;
    for(i=0; i<SIZE; i++)
        arr[i]=0;

    for(i=0; i<SIZE; i++) {
        if(i<SIZE)
            increase(arr,i);
        else
            printError();
    }
}
```

In this example, in the second `for` loop in `main`, `i` is always less than `SIZE`. Therefore, the `else` branch of the condition `if (i<SIZE)` is never reached. Because the function `printError` is called from the `else` branch alone, there is a gray **Function not reachable** check on the definition of `printError`.

Function Call Following Red Error

```
#include<stdio.h>

int getNum(void);

void printSuccess()
{
    printf("The operation is complete.");
}

void main() {
    int num=getNum(), den=0;
    printf("The ratio is %.2f", num/den);
    printSuccess();
}
```


In this example, the function `printSuccess` is called following a red **Division by Zero** error. Therefore, there is a gray **Function not reachable** check on the definition of `printSuccess`.

Function Call from Another Unreachable Function

```
#include<stdio.h>
#define MAX 1000
#define MIN 0

int getNum(void);

void checkUpperBound(double ratio)
{
    if(ratio < MAX)
        printf("The ratio is within bounds.");
}

void checkLowerBound(double ratio)
{
    if(ratio > MIN)
        printf("The ratio is within bounds.");
}

void checkRatio(double ratio)
{
    checkUpperBound(ratio);
    checkLowerBound(ratio);
}

void main() {
    int num=getNum(), den=0;
    double ratio;
    ratio=num/den;
    checkRatio(ratio);
}
```

In this example, the function `checkRatio` follows a red **Division by Zero** error. Therefore, there is a gray **Function not reachable** error on the definition of `checkRatio`. Because `checkUpperBound` and `checkLowerBound` are called only from `checkRatio`, there is also a gray **Function not reachable** check on their definitions.

Function Call from Unreachable Code Using Function Pointer

```
#include<stdio.h>

int  getNum(void);
int  getChoice(void);

int  num, den, choice;
double ratio;

void display(void)
{
    printf("Numerator = %d, Denominator = %d", num, den);
}

void display2(void)
{
    printf("Ratio = %.2f",ratio);
}

void main() {
    void (*fptr)(void);

    choice = getChoice();
    if(choice == 0)
        fptr = &display;
    else
        fptr = &display2;

    num = getNum();
    den = 0;
    ratio = num/den;

    (*fptr)();
}
```

In this example, depending on the value of `choice`, the function pointer `fptr` can point to either `display` or to `display2`. The call through `fptr` follows a red **Division by Zero** error. Because `display` and `display2` are called only through `fptr`, a gray **Function not reachable** check appears on their definitions.

Check Information

Group: Data flow

Language: C | C++

Acronym: FNR

See Also

Detect uncalled functions (-uncalled-function-checks) | Function not called | Unreachable code

Function not returning value

C++ function does not return value when expected

Description

This check determines whether a function with a return type other than `void` returns a value. This check appears on the function definition.

Diagnosing This Check

“Review and Fix Function Not Returning Value Checks”

Examples

Function does not return value for any input

```
#include <stdio.h>
int input();
int inputRep();

int reply(int msg) {
    int rep = inputRep();
    if (msg > 0) return rep;
}

void main(void) {
    int ch = input(), ans;
    if (ch<=0)
        ans = reply(ch);
    printf("The answer is %d.",ans);
}
```

In this example, for all values of `ch`, `reply(ch)` has no return value. Therefore the **Function not returning value** check returns a red error on the definition of `reply()`.

One possible correction is to return a value for all inputs to `reply()`.

```
#include <stdio.h>
int input();
int inputRep();

int reply(int msg) {
    int rep = inputRep();
    if (msg > 0) return rep;
    return 0;
}

void main(void) {
    int ch = input(), ans;
    if (ch<=0)
        ans = reply(ch);
    printf("The answer is %d.",ans);
}
```

Function does not return value for some inputs

```
#include <stdio.h>
int input();
int inputRep(int);

int reply(int msg) {
    int rep = inputRep(msg);
    if (msg > 0) return rep;
}

void main(void) {
    int ch = input(), ans;
    if (ch<10)
        ans = reply(ch);
    else
        ans = reply(10);
    printf("The answer is %d.",ans);
}
```

In this example, in the first branch of the `if` statement, the value of `ch` can be divided into two ranges:

- `ch <= 0`: For the function call `reply(ch)`, there is no return value.
- `ch > 0` and `ch < 10`: For the function call `reply(ch)`, there is a return value.

Therefore the **Function not returning value** check returns an orange error on the definition of `reply()`.

One possible correction is to return a value for all inputs to `reply()`.

```
#include <stdio.h>
int input();
int inputRep(int);

int reply(int msg) {
    int rep = inputRep(msg);
    if (msg > 0) return rep;
    return 0;
}

void main(void) {
    int ch = input(), ans;
    if (ch<10)
        ans = reply(ch);
    else
        ans = reply(10);
    printf("The answer is %d.",ans);
}
```

Check Information

Group: C++

Language: C++

Acronym: FRV

See Also

Return value not initialized

Illegally dereferenced pointer

Pointer is dereferenced outside bounds

Description

This check on a pointer dereference determines whether the pointer is NULL or points outside its bounds.

The check message shows you the pointer offset and buffer size in bytes. A pointer points outside its bounds when the sum of the offset and pointer size exceeds the buffer size.

- *Buffer*: When you assign an address to a pointer, a block of memory is allocated to the pointer. You cannot access memory beyond that block using the pointer. The size of this block is the buffer size.
- *Offset*: You can move a pointer within the allowed memory block by using pointer arithmetic. The difference between the initial location of the pointer and its current location is the offset.

For instance, if the pointer points to an array:

- The buffer size is the array size.
- The offset is the difference between the beginning of the array and the current location of the pointer.

Diagnosing This Check

“Review and Fix Illegally Dereferenced Pointer Checks”

Examples

Pointer points outside array bounds

```
#define Size 1024
```

```
int input(void);

void main() {
    int arr[Size];
    int *p = arr;

    for (int index = 0; index < Size ; index++, p++){
        *p = input();
    }
    *p = input();
}
```

In this example:

- Before the `for` loop, `p` points to the beginning of the array `arr`.
- After the `for` loop, `p` points outside the array.

The **Illegally dereferenced pointer** check on dereference of `p` after the `for` loop produces a red error.

One possible correction is to remove the illegal dereference of `p` after the `for` loop.

```
#define Size 1024

int input(void);

void main() {
    int arr[Size];
    int *p = arr;

    for (int index = 0; index < Size ; index++, p++) {
        *p = input();
    }
}
```

Pointer points outside structure field

```
typedef struct S {
    int f1;
    int f2;
    int f3;
} S;
```



```
void Initialize(int *ptr) {
    *ptr = 0;
    *(ptr+1) = 0;
    *(ptr+2) = 0;
}

void main(void) {
    S myStruct;
    Initialize(&myStruct.f1);
}
```

In this example, in the body of `Initialize`, `ptr` is an `int` pointer that points to the first field of the structure. When you attempt to access the second field through `ptr`, the **Illegally dereferenced pointer** check produces a red error.

One possible correction is to pass a pointer to the entire structure to `Initialize`.

```
typedef struct S {
    int f1;
    int f2;
    int f3;
} S;

void Initialize(S* ptr) {
    ptr->f1 = 0;
    ptr->f2 = 0;
    ptr->f3 = 0;
}

void main(void) {
    S myStruct;
    Initialize(&myStruct);
}
```

NULL pointer is dereferenced

```
#include<stdlib.h>

void main() {
    int *ptr=NULL;
    *ptr=0;
}
```

In this example, `ptr` is assigned the value `NULL`. Therefore when you dereference `ptr`, the **Illegally dereferenced pointer** check produces a red error.

One possible correction is to initialize `ptr` with the address of a variable instead of `NULL`.

```
void main() {
    int var;
    int *ptr=&var;
    *ptr=0;
}
```

Offset on NULL pointer

```
int getOffset(void);

void main() {
    int *ptr = (int*) 0 + getOffset();
    if(ptr != (int*)0)
        *ptr = 0;
}
```

In this example, although an offset is added to `(int*) 0`, Polyspace does not treat the result as a valid address. Therefore when you dereference `ptr`, the **Illegally dereferenced pointer** check produces a red error.

Bit field type is incorrect

```
struct flagCollection {
    unsigned int flag1: 1;
    unsigned int flag2: 1;
    unsigned int flag3: 1;
    unsigned int flag4: 1;
    unsigned int flag5: 1;
    unsigned int flag6: 1;
    unsigned int flag7: 1;
};

char getFlag(void);

int main()
{
```

```
    unsigned char myFlag = getFlag();
    struct flagCollection* myFlagCollection;
    myFlagCollection = (struct flagCollection *) &myFlag;
    if (myFlagCollection->flag1 == 1)
        return 1;
    return 0;
}
```

In this example:

- The fields of `flagCollection` have type `unsigned int`. Therefore, a `flagCollection` structure requires 32 bits of memory in a 32-bit architecture even though the fields themselves occupy 7 bits.
- When you cast a `char` address `&myFlag` to a `flagCollection` pointer `myFlagCollection`, you assign only 8 bits of memory to the pointer. Therefore, the **Illegally dereferenced pointer** check on dereference of `myFlagCollection` produces a red error.

One possible correction is to use `unsigned char` as field type of `flagCollection` instead of `unsigned int`. In this case:

- The structure `flagCollection` requires 8 bits of memory.
- When you cast the `char` address `&myFlag` to the `flagCollection` pointer `myFlagCollection`, you also assign 8 bits of memory to the pointer. Therefore, the **Illegally dereferenced pointer** check on dereference of `myFlagCollection` is green.

```
struct flagCollection {
    unsigned char flag1: 1;
    unsigned char flag2: 1;
    unsigned char flag3: 1;
    unsigned char flag4: 1;
    unsigned char flag5: 1;
    unsigned char flag6: 1;
    unsigned char flag7: 1;
};

char getFlag(void);

int main()
```

```
{
    unsigned char myFlag = getFlag();
    struct flagCollection* myFlagCollection;
    myFlagCollection = (struct flagCollection *) &myFlag;
    if (myFlagCollection->flag1 == 1)
        return 1;
    return 0;
}
```

Return value of `malloc` is not checked for `NULL`

```
#include <stdlib.h>

void main(void)
{
    char *p = (char*)malloc(1);
    char *q = p;
    *q = 'a';
}
```

In this example, `malloc` can return `NULL` to `p`. Therefore, when you assign `p` to `q` and dereference `q`, the **Illegally dereferenced pointer** check produces a red error.

One possible correction is to check `p` for `NULL` before dereferencing `q`.

```
#include <stdlib.h>
void main(void)
{
    char *p = (char*)malloc(1);
    char *q = p;
    if(p!=NULL) *q = 'a';
}
```

Pointer to union gets insufficient memory from `malloc`

```
#include <stdlib.h>

enum typeName {CHAR,INT};

typedef struct {
    enum typeName myTypeName;
```

```
    union {
        char myChar;
        int myInt;
    } myVar;
} myType;

void main() {
    myType* myTypePtr;
    myTypePtr = (myType*)malloc(sizeof(int) + sizeof(char));
    if(myTypePtr != NULL) {
        myTypePtr->myTypeName = INT;
    }
}
```

In this example:

- Because the union `myVar` has an `int` variable as a field, it must be assigned 4 bytes in a 32-bit architecture. Therefore, the structure `myType` must be assigned $4+4 = 8$ bytes.
- `malloc` returns `sizeof(int) + sizeof(char)=4+1=5` bytes of memory to `myTypePtr`, a pointer to a `myType` structure. Therefore, when you dereference `myTypePtr`, the **Illegally dereferenced pointer** check returns a red error.

One possible correction is to assign 8 bytes of memory to `myTypePtr` before dereference.

```
#include <stdlib.h>

enum typeName {CHAR,INT};

typedef struct {
    enum typeName myTypeName;
    union {
        char myChar;
        int myInt;
    } myVar;
} myType;

void main() {
    myType* myTypePtr;
    myTypePtr = (myType*)malloc(sizeof(int) + sizeof(int));
    if(myTypePtr != NULL) {
        myTypePtr->myTypeName = INT;
    }
}
```

```
    }  
}
```

Structure is allocated memory partially

```
#include <stdlib.h>  
typedef struct {  
    int length;  
    int breadth;  
} rectangle;  
  
typedef struct {  
    int length;  
    int breadth;  
    int height;  
} cuboid;  
  
void main() {  
    cuboid *cuboidPtr = (cuboid*)malloc(sizeof(rectangle));  
    if(cuboidPtr!=NULL) {  
        cuboidPtr->length = 10;  
        cuboidPtr->breadth = 10;  
    }  
}
```

In this example, `cuboidPtr` obtains sufficient memory to accommodate two of its fields. Because the ANSI C standards do not allow such partial memory allocations, the **Illegally dereferenced pointer** check on dereference of `cuboidPtr` produce a red error.

To observe ANSI C standards, `cuboidPtr` must be allocated full memory.

```
#include <stdlib.h>  
typedef struct {  
    int length;  
    int breadth;  
} rectangle;  
  
typedef struct {  
    int length;  
    int breadth;
```

```
    int height;
} cuboid;

void main() {
    cuboid *cuboidPtr = (cuboid*)malloc(sizeof(cuboid));
    if(cuboidPtr!=NULL) {
        cuboidPtr->length = 10;
        cuboidPtr->breadth = 10;
    }
}
```

You can allow partial memory allocation for structures, yet not have a red **Illegally dereferenced pointer** error. To allow partial memory allocation, on the **Configuration** pane, under **Check Behavior**, select **Allow incomplete or partial allocation of structures**.

This option is not available for C++ projects.

```
#include <stdlib.h>
typedef struct {
    int length;
    int breadth;
} rectangle;

typedef struct {
    int length;
    int breadth;
    int height;
} cuboid;

void main() {
    cuboid *cuboidPtr = (cuboid*)malloc(sizeof(rectangle));
    if(cuboidPtr!=NULL) {
        cuboidPtr->length = 10;
        cuboidPtr->breadth = 10;
    }
}
```

Pointer to one field of structure points to another field

```
#include <stdlib.h>
typedef struct {
    int length;
    int breadth;
} square;

void main() {
    square mySquare;
    char* squarePtr = (char*)&mySquare.length;
    //Assign zero to mySquare.length byte by byte
    for(int byteIndex=1; byteIndex<=4; byteIndex++) {
        *squarePtr=0;
        squarePtr++;
    }
    //Assign zero to first byte of mySquare.breadth
    *squarePtr=0;
}
```

In this example, although `squarePtr` is a `char` pointer, it is assigned the address of the integer `mySquare.length`. Because:

- `char` occupies 1 byte,
- `int` occupies 4 bytes in a 32-bit architecture,

`squarePtr` can access the four bytes of `mySquare.length` through pointer arithmetic. But when it accesses the first byte of another field `mySquare.breadth`, the **Illegally dereferenced pointer** check produces a red error.

One possible correction is to assign `squarePtr` the address of the full structure `mySquare` instead of `mySquare.length`. `squarePtr` can then access all the bytes of `mySquare` through pointer arithmetic.

```
#include <stdlib.h>
typedef struct {
    int length;
    int breadth;
} square;
```



```
void main() {
    square mySquare;
    char* squarePtr = (char*)&mySquare;
    //Assign zero to mySquare.length byte by byte
    for(int byteIndex=1; byteIndex<=4; byteIndex++) {
        *squarePtr=0;
        squarePtr++;
    }
    //Assign zero to first byte of mySquare.breadth
    *squarePtr=0;
}
```

You can use a pointer to navigate across the fields of a structure and not produce a red **Illegally dereferenced pointer** error. To allow such navigation, on the **Configuration** pane, under **Check Behavior**, select **Enable pointer arithmetic across fields**.

This option is not available for C++ projects.

```
#include <stdlib.h>
typedef struct {
    int length;
    int breadth;
} square;

void main() {
    square mySquare;
    char* squarePtr = (char*)&mySquare.length;
    //Assign zero to mySquare.length byte by byte
    for(int byteIndex=1; byteIndex<=4; byteIndex++) {
        *squarePtr=0;
        squarePtr++;
    }
    //Assign zero to first byte of mySquare.breadth
    *squarePtr=0;
}
```

Function returns pointer to local variable

```
void func2(int *ptr) {
```

```
    *ptr = 0;
}

int* func1(void) {
    int ret = 0;
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

In the following code, `ptr` points to `ret`. Because the scope of `ret` is limited to `func1`, when `ptr` is accessed in `func2`, the access is illegal. The verification produces a red **Illegally dereferenced pointer** check on `*ptr`.

By default, Polyspace Code Prover does not detect functions returning pointers to local variables. To detect such cases, use the option `Detect stack pointer dereference outside scope (-detect-pointer-escape)`.

Check Information

Group: Static memory

Language: C | C++

Acronym: IDP

See Also

Allow incomplete or partial allocation of structures (`-size-in-bytes`) | Enable pointer arithmetic across fields (`-allow-ptr-arith-on-struct`) | Detect stack pointer dereference outside scope (`-detect-pointer-escape`) | Non-initialized pointer

Incorrect object oriented programming

Dynamic type of `this` pointer is incorrect

Description

This check on a class member function call determines if the call is valid.

A member function call can be invalid for the following reasons:

- You call the member function through a function pointer that points to the function. However, the data types of the arguments or return values of the function and the function pointer do not match.
- You call a pure `virtual` member function from the class constructor or destructor.
- You call the member function through an incorrect `this` pointer. The `this` pointer stores the address of the object used to call the function. The `this` pointer can be incorrect because:
 - You obtain an object through a cast from another object. The objects are instances of two unrelated classes.
 - You perform pointer arithmetic on a pointer pointing to an array of objects. However, the pointer arithmetic causes the pointer to go outside the array bounds. When you dereference the pointer, it is not pointing to a valid object.

This class of **Incorrect object oriented programming** checks appear only if the member function belongs to a class that has or inherits at least one `virtual` function.

Diagnosing This Check

“Review and Fix Incorrect Object Oriented Programming Checks”

Examples

Pointer to method has incorrect type

```
#include <iostream>
class myClass {
public:
    void method() {}
};

void main() {
    myClass Obj;
    int (myClass::*methodPtr) (void) = (int (myClass::*) (void))
    &myClass::method;
    int res = (Obj.*methodPtr) ();
    std::cout << "Result = " << res;
}
```

In this example, the pointer `methodPtr` has return type `int` but points to `myClass::method` that has return type `void`. Therefore, when `methodPtr` is dereferenced, the **Incorrect object oriented programming** check produces a red error.

Pointer to method contains NULL when dereferenced

```
#include <iostream>
class myClass {
public:
    void method() {}
};

void main() {
    myClass Obj;
    void (myClass::*methodPtr) (void) = &myClass::method;
    methodPtr = 0;
    (Obj.*methodPtr) ();
}
```

In this example, `methodPtr` has value `NULL` when it is dereferenced.

Pure virtual function is called in base class constructor

```
class Shape {
public:
    Shape(Shape *myShape) {
        myShape->setShapeDimensions(0.0);
    }
    virtual void setShapeDimensions(double) = 0;
};

class Square: public Shape {
    double side;
public:
    Square():Shape(this) {
    }
    void setShapeDimensions(double);
};

void Square::setShapeDimensions(double val) {
    side=val;
}

void main() {
    Square sq;
    sq.setShapeDimensions(1.0);
}
```

In this example, the derived class constructor `Square::Square` calls the base class constructor `Shape::Shape()` with its `this` pointer. The base class constructor then calls the pure virtual function `Shape::setShapeDimensions` through the `this` pointer. Since the call to a pure virtual function from a constructor is undefined, the **Incorrect object oriented programming** check produces a red error.

Incorrect `this` Pointer: Cast Between Pointers to Unrelated Objects

```
#include <new>

class Foo {
public:
    void funcFoo() {}
};
```

```
class Bar {
public:
    void funcBar() {}
    virtual void anotherfuncBar() {}
};

void main() {
    Foo *FooPtr = new Foo;
    Bar *BarPtr = (Bar*)(void*)FooPtr;
    BarPtr->funcBar();
}
```

In this example, the classes `Foo` and `Bar` are not related. When a `Foo*` pointer is cast to a `Bar*` pointer and the `Bar*` pointer is used to call a member function of class `Bar`, the **Incorrect object oriented programming** check produces a red error.

Incorrect `this` Pointer: Pointer Out of Bounds

```
#include <new>
class Foo {
public:
    void func() {}
    virtual void anotherFunc() {}
};

void main() {
    Foo *FooPtr = new Foo[4];
    for(int i=0; i<=4; i++)
        FooPtr++;
    FooPtr->func();
    delete [] FooPtr;
}
```

In this example, the pointer `FooPtr` points outside the allocated bounds when it is used to call the member function `func()`. It does not point to a valid object. Therefore, the **Incorrect object oriented programming** check produces a red error.

Incorrect `this` Pointer: Non-initialized Object

```
class Foo {
public:
```

```
    virtual int func() {  
        return 1;  
    }  
};  
  
class Bar {  
private:  
    int m_int;  
    Foo m_Foo;  
public:  
    Bar() : m_int(m_Foo.func()) {}  
};
```

In this example, the constructor `Bar::Bar()` calls `m_Foo.func()` before `m_Foo` is initialized. Therefore, the **Incorrect object oriented programming** check produces a red error.

Check Information

Group: C++

Language: C++

Acronym: OOP

Invalid C++ specific operations

C++ specific invalid operations occur

Description

These checks on C++ code operations determine whether the operations are valid. The checks look for a range of invalid behaviors:

- Array size is not strictly positive.
- `typeid` operator dereferences a NULL pointer.
- `dynamic_cast` operator performs an invalid cast.

Diagnosing This Check

“Review and Fix Invalid C++ Specific Operations Checks”

Examples

Array size is not strictly positive

```
class License {
protected:
    int numberOfUsers;
    char (*userList)[20];
    int *licenseList;
public:
    License(int numberOfLicenses);
    void initializeList();
    char* getUser(int);
    int getLicense(int);
};

License::License(int numberOfLicenses) : numberOfUsers(numberOfLicenses) {
    userList = new char [numberOfUsers][20];
```



```
    licenseList = new int [numberOfUsers];
    initializeList();
}

int getNumberOfLicenses();
int getIndexForSearch();

void main() {
    int n = getNumberOfLicenses();
    if(n >= 0 && n <= 100) {
        License myFirm(n);
        int index = getIndexForSearch();
        myFirm.getUser(index);
        myFirm.getLicense(index);
    }
}
```

In this example, the argument `n` to the constructor `License::License` falls into two categories:

- `n = 0`: When the new operator uses this argument, the **Invalid C++ specific operations** produce an error.
- `n > 0`: When the new operator uses this argument, the **Invalid C++ specific operations** is green.

Combining the two categories of arguments, the **Invalid C++ specific operations** produce an orange error on the new operator.

typeid operator dereferences a NULL pointer

```
#include <iostream>
#include <typeinfo>
#define PI 3.142

class Shape {
public:
    Shape();
    virtual void setVal(double) = 0;
    virtual double area() = 0;
};

class Circle: public Shape {
    double radius;
```

```
public:
    Circle(double radiusVal):Shape() {
        setVal(radiusVal);
    }

    void setVal(double radiusVal) {
        if(radiusVal > 0)
            radius = radiusVal;
        else
            radius = 0;
    }

    double area() {
        return (PI * radius * radius);
    }
};

class Square: public Shape {
    double side;
public:
    Square(double sideVal):Shape() {
        setVal(sideVal);
    }

    void setVal(double sideVal) {
        if(sideVal > 0)
            side = sideVal;
        else
            side = 0;
    }

    double area() {
        return (side * side);
    }
};

Shape* getShapePtr();

void main() {
    Shape* shapePtr = getShapePtr();
    double val;

    if(typeid(*shapePtr)==typeid(Circle)) {
        std::cout<<"Enter radius:";
```

```

    std::cin>>val;
    shapePtr->setVal(val);
    std::cout<<"Area of circle = "<<shapePtr->area();
}
else if(typeid(*shapePtr) == typeid(Square)) {
    std::cout<<"Enter side:";
    std::cin>>val;
    shapePtr->setVal(val);
    std::cout<<"Area of square = "<<shapePtr->area();
}
else {
    std::cout<<"No valid shape.";
}
}

```

In this example, the `Shape*` pointer `shapePtr` returned by `getShapePtr()` function can be:

- **NULL:** When a NULL-valued `shapePtr` is used with the `typeid` operator, the **Invalid C++ specific operations** produce an error.
- **Not NULL:** When a non-NULL `shapePtr` is used with the `typeid` operator, the **Invalid C++ specific operations** is green.

Combining these two cases, the **Invalid C++ specific operations** produce an orange error on the `typeid` operator in the first `if` statement branch in `main`.

After this orange error, Polyspace considers that `shapePtr` is not NULL. Therefore, the **Invalid C++ specific operations** on the `typeid` operator in the second `if` statement branch are green.

Incorrect `dynamic_cast` on Pointers

```

class Base {
public :
    virtual void func() ;
};

class Derived : public Base {
};

Base* returnObj(int flag) {

```

```
    if(flag==0)
        return new Derived;
    else
        return new Base;
}

int main() {

    Base * ptrBase;
    Derived * ptrDerived;

    ptrBase = returnObj(0) ;
    ptrDerived = dynamic_cast<Derived*>(ptrBase); //Correct dynamic cast
    assert(ptrDerived != 0); //Returned pointer is not null

    ptrBase = returnObj(1);
    ptrDerived = dynamic_cast<Derived*>(ptrBase); //Incorrect dynamic cast
    // Verification continues despite red
    assert(ptrDerived == 0); //Returned pointer is null
}
```

In this example, the **Invalid C++ specific operations** on the `dynamic_cast` operator are:

- **Green**, when the pointer `ptrBase` that the operator casts to `Derived` is already pointing to a `Derived` object.
- **Red**, when the pointer `ptrBase` that the operator casts to `Derived` is pointing to a `Base` object.

Red checks typically stop the verification in the same scope as the check. However, after **red Invalid C++ specific operations** on `dynamic_cast` operation involving pointers, the verification continues. The software assumes that the `dynamic_cast` operator returns a `NULL` pointer.

Incorrect `dynamic_cast` on References

```
class Base {
public :
    virtual void func() ;
};
```

```
class Derived : public Base {
};

Base& returnObj(int flag) {
    if(flag==0)
        return *(new Derived);
    else
        return *(new Base);
}

int main() {
    Base & refBase1 = returnObj(0);
    Derived & refDerived1 = dynamic_cast<Derived&>(refBase1); //Correct dynamic cast;

    Base & refBase2 = returnObj(1);
    Derived & refDerived2 = dynamic_cast<Derived&>(refBase2); //Incorrect dynamic cast
    // Analysis stops
    assert(1);
}
```

In this example, the **Invalid C++ specific operations** on the `dynamic_cast` operator are:

- Green, when the reference `refBase1` that the operator casts to `Derived&` is already referring to a `Derived` object.
- Red, when the reference `refBase2` that the operator casts to `Derived&` is referring to a `Base` object.

After red **Invalid C++ specific operations** on `dynamic_cast` operation involving pointers, the software does not verify the code in the same scope as the check. For instance, the software does not perform the **User assertion** check on the `assert` statement.

Check Information

Group: C++

Language: C++

Acronym: CPP

See Also

External Websites

C++ Reference: `dynamic_cast` conversion

Invalid operation on floats

Result of floating-point operation is NaN for non-NaN operands

Description

This check determines if the result of a floating-point operation is NaN. The check is performed only if you enable a verification mode that incorporates NaNs and specify that the verification must highlight operations that result in NaN.

If you specify that the verification must produce a warning for NaN, the check is:

- Red, if the operation produces NaN on all execution paths that the software considers, and the operands are not NaN.
- Orange, if the operation produces NaN on some of the execution paths when the operands are not NaN.
- Green, if the operation does not produce NaN unless the operands are NaN.

If you specify that the verification must forbid NaN, the check color depends on the result of the operation only. The color does not depend on the operands.

To enable this verification mode, use these options:

- Consider non finite floats (`-allow-non-finite-floats`)
- NaNs (`-check-nan`): Use argument `warn-first` or `forbid`.

Examples

NaN Detected with Red Check

Results in forbid mode:

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
```

```
    return y;
}
```

In this example, both the operands of the `-` operation are not NaN but the result is NaN. The **Invalid operation on floats** check on the `-` operation is red. In the `forbid` mode, the verification stops after the red check. For instance, a **Non-initialized local variable** check does not appear on `y` in the `return` statement.

Results in `warn-first` mode:

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
    return y;
}
```

In this example, both the operands of the `-` operation are not NaN but the result is NaN. The **Invalid operation on floats** check on the `-` operation is red. The red checks in `warn-first` mode are different from red checks for other check types. The verification does not stop after the red check. For instance, a green **Non-initialized local variable** check appears on `y` in the `return` statement. If you place your cursor on `y` in the verification result, you see that it has the value NaN.

NaN Detected with Orange Check

Results in `forbid` mode:

```
double func(double arg1, double arg2) {
    double ret=arg1-arg2;
    return ret;
}
```

In this example, the values of `arg1` and `arg2` are unknown to the verification. The verification assumes that `arg1` and `arg2` can be both infinite, for instance, and the result of `arg1-arg2` can be NaN. In the `forbid` mode, following the check, the verification terminates the execution path that results in NaN. If you place your cursor on `ret` in the `return` statement, it does not have the value NaN.

Results in `warn-first` mode:

```
double func(double arg1, double arg2) {
    double ret=arg1-arg2;
```



```

    return ret;
}

```

In this example, the values of `arg1` and `arg2` are unknown to the verification. The verification assumes that `arg1` and `arg2` can be both infinite, for instance, and the result of `arg1-arg2` can be NaN. The orange checks in `warn-first` mode are different from orange checks for other check types. Following the check, the verification does not terminate the execution path that results in NaN. If you place your cursor on `ret` in the `return` statement, it continues to have the value NaN along with other possible values.

Orange Check Despite NaN Being the Only Result

```

double func(double arg1, double arg2) {
    double z=arg1-arg2;
    return z;
}

double caller() {
    double x=1.0/0.0;
    double y=x-x;
    func(x, x);
    func(y, y);
}

```

In this example, in `func`, the result of the `-` operation is always NaN but the **Invalid operation on floats** check is orange instead of red.

- In the first call to `func`, both the operands `arg1` and `arg2` are not NaN, but the result is NaN. So, the check is red.
- In the second call to `func`, both the operands `arg1` and `arg2` are NaN, and therefore the result is NaN. So, the check is green, indicating that the result is not NaN unless the operands are NaN.

Combining the colors for the two calls to `func`, the check is orange.

In the example, the option `-check-nan warn-first` was used.

Result Information

Group: Numerical

Language: C | C++

Acronym: INVALID_FLOAT_OP

See Also

Polyspace Analysis Options

NaNs (-check-nan)

Polyspace Results

Overflow | Subnormal float

Introduced in R2016a

Invalid shift operations

Shift operations are invalid

Description

This check on shift operations on a variable `var` determines:

- Whether the shift amount is larger than the range allowed by the type of `var`.
- If the shift is a left shift, whether `var` is negative.

Diagnosing This Check

“Review and Fix Invalid Shift Operations Checks”

Examples

Shift amount outside bounds

```
#include <stdlib.h>
#define shiftAmount 32
enum shiftType {
    SIGNED_LEFT,
    SIGNED_RIGHT,
    UNSIGNED_LEFT,
    UNSIGNED_RIGHT
};

enum shiftType getShiftType();

void main() {
    enum shiftType myShiftType = getShiftType();
    int signedInteger = 1;
    unsigned int unsignedInteger = 1;
    switch(myShiftType) {
        case SIGNED_LEFT:
```

```
        signedInteger = signedInteger << shiftAmount;
        break;
    case SIGNED_RIGHT:
        signedInteger = signedInteger >> shiftAmount;
        break;
    case UNSIGNED_LEFT:
        unsignedInteger = unsignedInteger << shiftAmount;
        break;
    case UNSIGNED_RIGHT:
        unsignedInteger = unsignedInteger >> shiftAmount;
        break;
    }
}
```

In this example, the shift amount `shiftAmount` is outside the allowed range for both signed and unsigned int. Therefore the **Invalid shift operations** check produces a red error.

One possible correction is to keep the shift amount in the range 0..31 for unsigned integers and 0...30 for signed integers. This correction works if the size of int is 32 on the target processor.

```
#include <stdlib.h>
#define shiftAmountSigned 30
#define shiftAmount 31
enum shiftType {
    SIGNED_LEFT,
    SIGNED_RIGHT,
    UNSIGNED_LEFT,
    UNSIGNED_RIGHT
};

enum shiftType getShiftType();

void main() {
    enum shiftType myShiftType = getShiftType();
    int signedInteger = 1;
    unsigned int unsignedInteger = 1;
    switch(myShiftType) {

    case SIGNED_LEFT:
        signedInteger = signedInteger << shiftAmountSigned;
        break;
```

```
case SIGNED_RIGHT:
    signedInteger = signedInteger >> shiftAmountSigned;
    break;

case UNSIGNED_LEFT:
    unsignedInteger = unsignedInteger << shiftAmount;
    break;

case UNSIGNED_RIGHT:
    unsignedInteger = unsignedInteger >> shiftAmount;
    break;
}
}
```

Left operand of left shift is negative

```
void main(void) {
    int x = -200;
    int y;
    y = x << 1;
}
```

In this example, the left operand of the left shift operation is negative.

You can use left shifts on negative numbers and not produce a red **Invalid shift operations** error. To allow such left shifts, on the **Configuration** pane, under **Check Behavior**, select **Allow negative operand for left shifts**.

```
void main(void) {
    int x = -200;
    int y;
    y = x << 1;
}
```

Left operand of left shift may be negative

```
short getVal();

int foo(void) {
    long lvar;
```

```
short svar1, svar2;

lvar = 0;
svar1 = getVal();
svar2 = getVal();

lvar = (svar1 - svar2) << 10;
if (svar1 < svar2) {
    return 1;
} else {
    return 0;
}
}

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

In this example, if `svar1 < svar2`, the left operand of `<<` can be negative. Therefore the **Shift operations** check on `<<` is orange. Following an orange check, execution paths containing the error get truncated. Therefore, following the orange **Invalid shift operations** check, Polyspace assumes that `svar1 >= svar2`. The branch of the statement, `if(svar1 < svar2)`, is unreachable.

Check Information

Group: Numerical

Language: C | C++

Acronym: SHF

See Also

Allow negative operand for left shifts (-allow-negative-operand-in-shift) | Consider non finite floats (-allow-non-finite-floats)

Invalid use of standard library routine

Standard library function is called with invalid arguments

Description

This check on a standard library function call determines whether the function is called with valid arguments.

The check works differently for memory routines, floating point routines or string routines because their arguments can be invalid in different ways. For more information on each type of routines, see the following examples.

Diagnosing This Check

“Review and Fix Invalid Use of Standard Library Routine Checks”

Examples

Invalid use of standard library float routine

```
#include <assert.h>
#include <math.h>

#define LARGE_EXP 710

enum operation {
    ASIN,
    ACOS,
    TAN,
    SQRT,
    LOG,
    EXP,
};

enum operation getOperation(void);
```

```
double getVal(void);

void main() {
    enum operation myOperation = getOperation();
    double myVal=getVal(), res;
    switch(myOperation) {
    case ASIN:
        assert( myVal <- 1.0 || myVal > 1.0);
        res = asin(myVal);
        break;
    case ACOS:
        assert( myVal < -1.0 || myVal > 1.0);
        res = acos(myVal);
        break;
    case SQRT:
        assert( myVal < 0.0);
        res = sqrt(myVal);
        break;
    case LOG:
        assert(myVal <= 0.0);
        res = log(myVal);
        break;
    case EXP:
        assert(myVal > LARGE_EXP);
        res = exp(myVal);
        break;
    }
}
```

In this example, following each `assert` statement, Polyspace considers that `myVal` contains only those values that make the `assert` condition true. For example, following `assert(myVal < 1.0);`, Polyspace considers that `myVal` contains values less than 1.0.

When `myVal` is used as argument in a standard library function, its values are invalid for the function. Therefore, the **Invalid use of standard library routine** check produces a red error.

To learn more about the specifications of this check for floating point routines, see “Invalid Use of Standard Library Floating Point Routines”.

Invalid use of standard library memory routine

```
#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);
    return str2;
}

int main(void) {
    (void*)Copy_First_Six_Letters();
    return 0;
}
```

In this example, the size of string `str2` is 5, but 6 characters of string `str1` are copied into `str2` using the `memcpy` function. Therefore, the **Invalid use of standard library routine** check on the call to `memcpy` produces a red error.

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) {
    char str1[10],str2[6];
    printf("Enter string:\n");
    scanf("%s",str1);
    memcpy(str2,str1,6);
    return str2;
}

int main(void) {
    (void*)Copy_First_Six_Letters();
    return 0;
}
```

Invalid use of standard library string routine

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

char* Copy_String(void)
{
    char *res;
    char gbuffer[5],text[20]="ABCDEFGHijkl";
    res=strcpy(gbuffer,text);
    return(res);
}

int main(void) {
    (void*)Copy_String();
}
```

In this example, the string `text` is larger in size than `gbuffer`. Therefore, when you copy `text` into `gbuffer`, the **Invalid use of standard library routine** check on the call to `strcpy` produces a red error.

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

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

char* Copy_String(void)
{
    char *res;
    char gbuffer[20],text[20]="ABCDEFGHijkl";
    res=strcpy(gbuffer,text);
    return(res);
}

int main(void) {
    (void*)Copy_String();
}
```

Check Information

Group: Other

Language: C | C++

Acronym: STD_LIB

See Also

Consider non finite floats (-allow-non-finite-floats) | Float rounding mode (-float-rounding-mode)

Non-initialized local variable

Local variable is not initialized before being read

Description

This check occurs for every local variable read. It determines whether the variable being read is initialized.

Diagnosing This Check

“Review and Fix Non-initialized Local Variable Checks”

Examples

Non-initialized variable used on right side of assignment operator

```
#include <stdio.h>

void main(void) {
    int sum;
    for(int i=1;i <= 10; i++)
        sum+=i;
    printf("The sum of the first 10 natural numbers is %d.", sum);
}
```

The statement `sum+=i;` is the shorthand for `sum=sum+i;`. Because `sum` is used on the right side of an expression before being initialized, the **Non-initialized local variable** check returns a red error.

One possible correction is to initialize `sum` before the `for` loop.

```
#include <stdio.h>

void main(void) {
```

```
int sum=0;
for(int i=1;i <= 10; i++)
    sum+=i;
printf("The sum of the first 10 natural numbers is %d.", sum);
}
```

Non-initialized variable used with relational operator

```
#include <stdio.h>

int getTerm();

void main(void) {
    int count,sum=0,term;

    while( count <= 10  && sum <1000) {
        count++;
        term = getTerm();
        if(term > 0 && term <= 1000) sum += term;
    }

    printf("The sum of 10 terms is %d.", sum);
}
```

In this example, the variable `count` is not initialized before the comparison `count <= 10`. Therefore, the **Non-initialized local variable** check returns a red error.

One possible correction is to initialize `count` before the comparison `count <= 10`.

```
#include <stdio.h>

int getTerm();

void main(void) {
    int count=1,sum=0,term;

    while( count <= 10  && sum <1000) {
        count++;
        term = getTerm();
        if(term > 0 && term <= 1000) sum+= term;
    }
}
```

```
    printf("The sum of 10 terms is %d.", sum);
}
```

Non-initialized variable passed to function

```
#include <stdio.h>

int getShift();
int shift(int var) {
    int shiftVal = getShift();
    if(shiftVal > 0 && shiftVal < 1000)
        return(var+shiftVal);
    return 1000;
}

void main(void) {
    int initVal;
    printf("The result of a shift is %d",shift(initVal));
}
```

In this example, `initVal` is not initialized when it is passed to `shift()`. Therefore, the **Non-initialized local variable** check returns a red error. Because of the red error, Polyspace does not verify the operations in `shift()`.

One possible correction is to initialize `initVal` before passing to `shift()`. `initVal` can be initialized through an input function. To avoid an overflow, the value returned from the input function must be within bounds.

```
#include <stdio.h>

int getShift();
int getInit();
int shift(int var) {
    int shiftVal = getShift();
    if(shiftVal > 0 && shiftVal < 1000)
        return(var+shiftVal);
    return 1000;
}

void main(void) {
    int initVal=getInit();
    if(initVal >0 && initVal < 1000)
```

```

    printf("The result of a shift is %d",shift(initVal));
else
    printf("Value must be between 0 and 1000.");
}

```

Non-initialized array element

```

#include <stdio.h>
#define arrSize 19

void main(void)
{
    int arr[arrSize],indexFront, indexBack;
    for(indexFront = 0,indexBack = arrSize - 1;
        indexFront < arrSize/2;
        indexFront++, indexBack--) {
        arr[indexFront] = indexFront;
        arr[indexBack] = arrSize - indexBack - 1;
    }
    printf("The array elements are: \n");
    for(indexFront = 0; indexFront < arrSize; indexFront++)
        printf("Element[%d]: %d", indexFront, arr[indexFront]);
}

```

In this example, in the first for loop:

- `indexFront` runs from 0 to 8.
- `indexBack` runs from 18 to 10.

Therefore, `arr[9]` is not initialized. In the second for loop, when `arr[9]` is passed to `printf`, the **Non-initialized local variable** check returns an error. The error is orange because the check returns an error only in one of the loop runs.

Due to the orange error in one of the loop runs, a red **Non-terminating loop** error appears on the second for loop.

One possible correction is to keep the first for loop intact and initialize `arr[9]` outside the for loop.

```

#include <stdio.h>
#define arrSize 19

```

```
void main(void)
{
    int arr[arrSize],indexFront, indexBack;
    for(indexFront = 0,indexBack = arrSize - 1;
        indexFront < arrSize/2;
        indexFront++, indexBack--) {
        arr[indexFront] = indexFront;
        arr[indexBack] = arrSize - indexBack - 1;
    }
    arr[indexFront] = indexFront;
    printf("The array elements are: \n");
    for(indexFront = 0; indexFront < arrSize; indexFront++)
        printf("Element[%d]: %d", indexFront, arr[indexFront]);
}
```

Non-initialized structure

```
typedef struct S {
    int integerField;
    char characterField;
}S;

void operateOnStructure(S);
void operateOnStructureField(int);

void main() {
    S myStruct;
    operateOnStructure(myStruct);
    operateOnStructureField(myStruct.integerField);
}
```

In this example, the structure `myStruct` is not initialized. Therefore, when the structure `myStruct` is passed to the function `operateOnStructure`, a **Non-initialized local variable** check on the structure appears red.

One possible correction is to initialize the structure `myStruct` before passing it to a function.

```
typedef struct S {
    int integerField;
    char characterField;
}S;
```



```
void operateOnStructure(S);
void operateOnStructureField(int);

void main() {
    S myStruct = {0, ' '};
    operateOnStructure(myStruct);
    operateOnStructureField(myStruct.integerField);
}
```

Partially initialized structure — All used fields initialized

```
typedef struct S {
    int integerField;
    char characterField;
    double doubleField;
}S;

int getIntegerField(void);
char getCharacterField(void);

void printIntegerField(int);
void printCharacterField(char);

void printFields(S s) {
    printIntegerField(s.integerField);
    printCharacterField(s.characterField);
}

void main() {
    S myStruct;

    myStruct.integerField = getIntegerField();
    myStruct.characterField = getCharacterField();
    printFields(myStruct);
}
```

In this example, the **Non-initialized local variable** check on `myStruct` is green because:

- The fields `integerField` and `characterField` that are used are both initialized.
- Although the field `doubleField` is not initialized, there is no read or write operation on the field `doubleField` in the code.

To determine which fields are checked for initialization:

- 1 Select the check on the **Results List** pane or **Source** pane.
- 2 View the message on the **Result Details** pane.

Partially initialized structure — Some used fields initialized

```
typedef struct S {
    int integerField;
    char characterField;
    double doubleField;
}S;

int getIntegerField(void);
char getCharacterField(void);

void printIntegerField(int);
void printCharacterField(char);
void printDoubleField(double);

void printFields(S s) {
    printIntegerField(s.integerField);
    printCharacterField(s.characterField);
    printDoubleField(s.doubleField);
}

void main() {
    S myStruct;

    myStruct.integerField = getIntegerField();
    myStruct.characterField = getCharacterField();
    printFields(myStruct);
}
```

In this example, the **Non-initialized local variable** check on `myStruct` is orange because:

- The fields `integerField` and `characterField` that are used are both initialized.
- The field `doubleField` is not initialized and there is a read operation on `doubleField` in the code.

To determine which fields are checked for initialization:

- 1 Select the check on the **Results List** pane or **Source** pane.
- 2 View the message on the **Result Details** pane.

Check Information

Group: Data flow

Language: C | C++

Acronym: NIVL

See Also

Disable checks for non-initialization (`-disable-initialization-checks`) | Non-initialized pointer | Non-initialized variable

Non-initialized pointer

Pointer is not initialized before being read

Description

This check occurs for every pointer read. It determines whether the pointer being read is initialized.

Diagnosing This Check

“Review and Fix Non-initialized Pointer Checks”

Examples

Non-initialized pointer passed to function

```
int assignValueToAddress(int *ptr) {
    *ptr = 0;
}

void main() {
    int* newPtr;
    assignValueToAddress(newPtr);
}
```

In this example, `newPtr` is not initialized before it is passed to `assignValueToAddress()`.

One possible correction is to assign `newPtr` an address before passing to `assignValueToAddress()`.

```
int assignValueToAddress(int *ptr) {
    *ptr = 0;
}
```

```
}

void main() {
    int val;
    int* newPtr = &val;
    assignValueToAddress(newPtr);
}
```

Non-initialized pointer to structure

```
#include <stdlib.h>
#define stackSize 25

typedef struct stackElement {
    int value;
    int *prev;
}stackElement;

int input();

void main() {
    stackElement *stackTop;

    for (int count = 0; count < stackSize; count++) {
        if(stackTop!=NULL) {
            stackTop -> value = input();
            stackTop -> prev = (int*)stackTop;
        }
        stackTop = (stackElement*)malloc(sizeof(stackElement));
    }
}
```

In this example, in the first run of the for loop, `stackTop` is not initialized and does not point to a valid address. Therefore, the **Non-initialized pointer** check on `stackTop!=NULL` returns a red error.

One possible correction is to initialize `stackTop` through `malloc()` before the check `stackTop!=NULL`.

```
#include <stdlib.h>
#define stackSize 25
```

```
typedef struct stackElement {
    int value;
    int *prev;
}stackElement;

int input();

void main() {
    stackElement *stackTop;

    for (int count = 0; count < stackSize; count++) {
        stackTop = (stackElement*)malloc(sizeof(stackElement));
        if(stackTop!=NULL) {
            stackTop->value = input();
            stackTop->prev = (int*)stackTop;
        }
    }
}
```

Non-initialized `char*` pointer used to store string

```
#include <stdio.h>

void main() {
    char *str;
    scanf("%s", str);
}
```

In this example, `str` does not point to a valid address. Therefore, when the `scanf` function reads a string from the standard input to `str`, the **Non-initialized pointer** check returns a red error.

One possible correction is to declare `str` as a `char` array. This declaration assigns an address to the `char*` pointer associated with the array name `str`. You can then use the pointer as input to `scanf`.

```
#include <stdio.h>

void main() {
    char str[10];
    scanf("%s", str);
}
```

Non-initialized array of `char*` pointers used to store variable-size strings

```
#include <stdio.h>

void assignDataBaseElement(char** str) {
    scanf("%s", *str);
}

void main() {
    char *dataBase[20];

    for(int count = 1; count < 20 ; count++) {
        assignDataBaseElement(&dataBase[count]);
        printf("Database element %d : %s",count,dataBase[count]);
    }
}
```

In this example, `dataBase` is an array of `char*` pointers. In each run of the `for` loop, an element of `dataBase` is passed via pointers to the function `assignDataBaseElement()`. The element passed is not initialized and does not contain a valid address. Therefore, when the element is used to store a string from standard input, the **Non-initialized pointer** check returns a red error.

One possible correction is to initialize each element of `dataBase` through the `calloc()` function before passing it to `assignDataBaseElement()`. The initialization through `calloc()` allows the `char` pointers in `dataBase` to point to strings of varying size.

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

void assignDataBaseElement(char** str) {
    scanf("%s", *str);
}

int inputSize();

void main() {
    char *dataBase[20];

    for(int count = 1; count < 20 ; count++) {
        dataBase[count] = (char*)calloc(inputSize(),sizeof(char));
        assignDataBaseElement(&dataBase[count]);
    }
}
```

```
        printf("Database element %d : %s",count,dataBase[count]);  
    }  
}
```

Check Information

Group: Data flow

Language: C | C++

Acronym: NIP

See Also

Disable checks for non-initialization (-disable-initialization-checks) | Non-initialized local variable | Non-initialized variable

Non-initialized variable

Variable other than local variable is not initialized before being read

Description

This check occurs when you read variables that are not local (global or static variables). It determines whether the variable being read is initialized.

Note By default, Polyspace considers that global variables are initialized. The verification checks global variables only if you prevent this default initialization. See “Initialization of Global Variables” on page 4-11.

Diagnosing This Check

“Review and Fix Non-initialized Variable Checks”

Examples

Non-initialized global variable

```
int globVar;
int getVal();

void main() {
    int val = getVal();
    if(val>=0 && val<= 100)
        globVar += val;
}
```

In this example, `globVar` does not have an initial value when incremented. Therefore, the **Non-initialized variable** check produces a red error.

The example uses the option to prevent default initialization of global variables.

One possible correction is to initialize the global variable `globVar` before use.

```
int globVar;
int getVal();

void main() {
    int val = getVal();
    globVar = 0;
    if(val>=0 && val<= 100)
        globVar += val;
}
```

Check Information

Group: Data flow

Language: C | C++

Acronym: NIV

See Also

Ignore default initialization of global variables (`-no-def-init-glob`) | Disable checks for non-initialization (`-disable-initialization-checks`) | Non-initialized local variable | Non-initialized pointer

Non-terminating call

Called function does not return to calling context

Description

This check on a function call appears when the following conditions hold:

- The called function does not return to its calling context. The call leads to a definite run-time error or a process termination function like `exit()` in the function body.
- There are other calls to the same function that do not lead to a definite error or process termination function in the function body.

When only a fraction of calls to a function lead to a definite error, this check helps identify those function calls. In the function body, even though a definite error occurs, the error appears in orange instead of red because the verification results in a function body are aggregated over all function calls. To indicate that a definite error has occurred, a red **Non-terminating call** check is shown *on the function call* instead.

Otherwise, if all the calls to a function lead to a definite error or process termination function in the function body, the **Non-terminating call** error is not displayed. The error appears in red in the function body and a dashed red underline appears on the function calls. However, following the function call, like other red errors, Polyspace does not analyze the remaining code in the same scope as the function call.

You can navigate directly from the function call to the operation causing the run-time error in the function body.

- To find the source of error, on the **Source** pane, place your cursor on the loop keyword and view the tooltip.
- Navigate to the source of error in the function body. Right-click the function call and select **Go to Cause** if the option exists.

If the error is the result of multiple causes, the option takes you to the first cause in the function body. Multiple causes can occur, for instance, when some values of a function argument trigger one specific error and other values trigger other errors.

Diagnosing This Check

“Review and Fix Non-Terminating Call Checks”

Examples

Dashed red underline on function call

```
#include<stdio.h>
double ratio(int num, int den) {
    return(num/den);
}

void main() {
    int i,j;
    i=2;
    j=0;
    printf("%.2f",ratio(i,j));
}
```

In this example, a red **Division by zero** error appears in the body of `ratio`. This **Division by zero** error in the body of `ratio` causes a dashed red underline on the call to `ratio`.

Red underline on function call

```
#include<stdio.h>
double ratio(int num, int den) {
    return(num/den);
}

int inputCh();

void main() {
    int i,j,ch=inputCh();
    i=2;

    if(ch==1) {
        j=0;
        printf("%.2f",ratio(i,j));
    }
}
```

```

    }
    else {
        j=2;
        printf("%.2f",ratio(i,j));
    }
}

```

In this example, there are two calls to `ratio`. In the first call, a **Division by zero** error occurs in the body of `ratio`. In the second call, Polyspace does not find errors. Therefore, combining the two calls, an orange **Division by zero** check appears in the body of `ratio`. A red **Non-terminating call** check on the first call indicates the error.

Red underline on call through function pointer

```

typedef void (*f)(void);
// function pointer type

void f1(void) {
    int x;
    x++;
}

void f2(void) { }
void f3(void) { }

f fptr_array[3] = {f1,f2,f3};
unsigned char getIndex(void);

void main(void) {
    unsigned char index = getIndex() % 3;
    // Index is between 0 and 2

    fptr_array[index]();
    fptr_array[index]();
}

```

In this example, because `index` can lie between 0 and 2, the first `fptr_array[index]()` can call `f1`, `f2` or `f3`. If `index` is zero, the statement calls `f1`. `f1` contains a red **Non-initialized local variable** error, therefore, a dashed red error appears on the function call. Unlike other red errors, the verification continues.

After this statement, the software considers that `index` is either 1 or 2. An error does not occur on the second `fptr_array[index]()`.

Check Information

Group: Control flow

Language: C | C++

Acronym: NTC

See Also

Non-terminating loop

Topics

“Identify Function Call with Run-Time Error”

Non-terminating loop

Loop does not terminate or contains an error

Description

This check on a loop determines if the loop has one of the following issues:

- The loop definitely does not terminate.

The check appears only if Polyspace cannot detect an exit path from the loop. For example, if the loop appears in a function and the loop termination condition is met for some function inputs, the check does not appear, even though the condition might not be met for some other inputs.

- The loop contains a definite error in one its iterations.

Even though a definite error occurs in one loop iteration, because the verification results in a loop body are aggregated over all loop iterations, the error shows as an orange check in the loop body. To indicate that a definite failure has occurred, a red **Non-terminating loop** check is shown on the loop command.

Unlike other checks, this check appears only when a definite error occurs. In your verification results, the check is always red.

The check does not appear if both conditions are true:

- The loop has a trivial predicate such as `for(;;)` or `while(1)`.
- The loop has an empty body, or a body without an exit statement such as `break`, `goto`, `return` or an exception.

Instead, the loop statement is underlined with red dashes. If you place your cursor on the loop statement, you see that the verification considers the loop as intentional. If you deliberately introduce infinite loops, for instance, to emulate cyclic tasks, you do not have to justify red checks.

Using this check, you can identify the operation in the loop that causes the run-time error.

- To find the source of error, on the **Source** pane, place your cursor on the function call and view the tooltip.
- For loops with fewer iterations, you can navigate to the source of error in the loop body. Select the loop to see the full history of the result. Alternatively, right-click the loop keyword and select **Go to Cause** if the option exists.

Diagnosing This Check

“Review and Fix Non-Terminating Loop Checks”

Examples

Loop does not terminate

```
#include<stdio.h>

void main() {
    int i=0;
    while(i<10) {
        printf("%d",i);
    }
}
```

In this example, in the `while` loop, `i` does not increase. Therefore, the test `i<10` never fails.

One possible correction is to update `i` such that the test `i<10` fails after some loop iterations and the loop terminates.

```
#include<stdio.h>

void main() {
    int i=0;
    while(i < 10) {
        printf("%d",i);
        i++;
    }
}
```


Loop contains an out of bounds array index error

```
void main() {
    int arr[20];
    for(int i=0; i<=20; i++) {
        arr[i]=0;
    }
}
```

In this example, the last run of the `for` loop contains an **Out of bounds array index** error. Therefore, the **Non-terminating loop** check on the `for` loop is red. A tooltip appears on the `for` loop stating the maximum number of iterations including the one containing the run-time error.

One possible correction is to reduce the number of loop iterations so that the **Out of bounds array index** error does not occur.

```
void main() {
    int arr[20];
    for(int i=0; i<20; i++) {
        arr[i]=0;
    }
}
```

Loop contains an error in function call

```
int arr[4];

void assignValue(int index) {
    arr[index] = 0;
}

void main() {
    for(int i=0;i<=4;i++)
        assignValue(i);
}
```

In this example, the call to function `assignValue` in the last `for` loop iteration contains an error. Therefore, although an error does not show in the `for` loop body, a red **Non-terminating loop** appears on the loop itself.

One possible correction is to reduce the number of loop iterations so the error in the call to `assignValue` does not occur.

```
int arr[4];

void assignValue(int index) {
    arr[index] = 0;
}

void main() {
    for(int i=0;i<4;i++)
        assignValue(i);
}
```

Loop contains an overflow error

```
#define MAX 1024
void main() {
    int i=0,val=1;
    while(i<MAX) {
        val*=2;
        i++;
    }
}
```

In this example, an **Overflow** error occurs in iteration number 31. Therefore, the **Non-terminating loop** check on the `while` loop is red. A tooltip appears on the `while` loop stating the maximum number of iterations including the one containing the run-time error.

One possible correction is to reduce the number of loop iterations so that the overflow does not occur.

```
#define MAX 30
void main() {
    int i=0,val=1;
    while(i<MAX) {
        val*=2;
        i++;
    }
}
```

Check Information

Group: Control flow

Language: C | C++

Acronym: NTL

See Also

Non-terminating call

Topics

“Identify Loop Operation with Run-Time Error”

Null this-pointer calling method

this pointer is null during member function call

Description

This check on a `this` pointer dereference determines whether the pointer is `NULL`.

Diagnosing This Check

“Review and Fix Null This-pointer Calling Method Checks”

Examples

Pointer to object is `NULL` during member function call

```
#include <stdlib.h>
class Company {
public:
    Company(int initialNumber):numberOfClients(initialNumber) {}
    void addNewClient() {
        numberOfClients++;
    }
protected:
    int numberOfClients;
};

void main() {
    Company* myCompany = NULL;
    myCompany->addNewClient();
}
```

In this example, the pointer `myCompany` is initialized to `NULL`. Therefore when the pointer is used to call the member function `addNewClient`, the **Null this-pointer calling method** produces a red error.

One possible correction is to initialize `myCompany` with a valid memory address using the `new` operator.

```
#include <stdlib.h>
class Company {
public:
    Company(int initialNumber):numberOfClients(initialNumber) {}
    void addNewClient() {
        numberOfClients++;
    }
protected:
    int numberOfClients;
};

void main() {
    Company* myCompany = new Company(0);
    myCompany->addNewClient();
}
```

Check Information

Group: C++

Language: C++

Acronym: NNT

Out of bounds array index

Array is accessed outside range

Description

This check on an array element access determines whether the element is outside the array range.

Diagnosing This Check

“Review and Fix Out of Bounds Array Index Checks”

Examples

Array index is equal to array size

```
#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]);
}

int main(void) {
```

```
    fibonacci();  
}
```

In this example, 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, when the `printf` statement attempts to access `fib[10]` through `i`, the **Out of bounds array index** check produces a red error.

The check also produces a red error if `printf` uses `*(fib+i)` instead of `fib[i]`.

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];  
    }  
  
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);  
}  
  
int main(void) {  
    fibonacci();  
}
```

Check Information

Group: Static memory

Language: C | C++

Acronym: OBAI

See Also

Illegally dereferenced pointer

Overflow

Arithmetic operation causes overflow

Description

This check on an arithmetic operation determines whether the result overflows. The result of this check depends on whether you allow nonfinite float results such as infinity and NaN.

The result of the check also depends on the float rounding mode you specify. By default, the rounding mode is `to-nearest`. See `Float rounding mode (-float-rounding-mode)`.

Nonfinite Floats Not Allowed

By default, nonfinite floats are not allowed. When the result of an operation falls outside the allowed range, an overflow occurs. The check is:

- Red, if the result of the operation falls outside the allowed range.
- Orange, if the result of the operation falls outside the allowed range on some of the execution paths.
- Green, if the result of the operation does not fall outside the allowed range.

The operand data types determine the allowed range for the arithmetic operation. If the operation involves two operands, the verification uses the ANSI C conversion rules to determine a common data type. This common data type determines the allowed range. For some examples of conversion rules, see “Implicit Data Type Conversions” on page 4-16.

Nonfinite Floats Allowed

If you enable a verification mode that incorporates infinities and specify that the verification must warn about operations that produce infinities, the check is:

- Red, if the operation produces infinity on all execution paths that the software considers, and the operands themselves are not infinite.

- Orange, if the operation produces infinity on some of the execution paths when the operands themselves are not infinite.
- Green, if the operation does not produce infinity unless the operands themselves are infinite.

If you specify that the verification must forbid operations that produce infinities, the check color depends on the result of the operation only. The color does not depend on the operands.

To enable this verification mode, use these options:

- `Consider non finite floats (-allow-non-finite-floats)`
- `Infinities (-check-infinite): Use argument warn or forbid.`

Diagnosing This Check

“Review and Fix Overflow Checks”

Examples

Integer Overflow

```
void main() {
    int i=1;
    i = i << 30; //i = 2^30
    i = 2*i-2;
}
```

In this example, the operation $2 * i$ results in a value 2^{31} . The **Overflow** check on the multiplication produces a red error because the maximum value that the type `int` can hold on a 32-bit target is $2^{31}-1$.

Overflow Due to Left Shift on Signed Integers

```
void main(void)
{
    unsigned int i;
```

```
    i = 1090654225 << 1;
}
```

In this example, an **Overflow** error occurs due to integer promotion.

Float Overflow

```
#include <float.h>

void main() {
    float val = FLT_MAX;
    val = val * 2 + 1.0;
}
```

In this example, `FLT_MAX` is the maximum value that `float` can represent on a 32-bit target. Therefore, the operation `val * 2` results in an **Overflow** error.

Overflow on Casts from Negative Floats to Unsigned Integers

```
void func(void) {
    float fVal = -2.0f;
    unsigned int iVal = (unsigned int)fVal;
}
```

In this example, a red **Overflow** check appears on the cast from `float` to `unsigned int`. According to the C99 Standard (footnote to paragraph 6.3.1.4), the range of values that can be converted from floating-point values to unsigned integers while keeping the code portable is $(-1, \text{MAX} + 1)$. For floating-point values outside this range, the conversion to unsigned integers is not well-defined. Here, `MAX` is the maximum number that can be stored by the unsigned integer type.

Even if a run-time error does not occur when you execute the code on your target, the cast might fail on another target.

One possible solution is to cast the floating-point value to a signed integer first. The signed integer can then be cast to an unsigned integer type. For these casts, the conversion rules are well-defined.

```
void func(void) {
    float fVal = -2.0f;
    int iValTemp = (int)fVal;
```

```
    unsigned int iVal = (unsigned int)iValTemp;
}
```

Negative Overflow

```
#define FLT_MAX 3.40282347e+38F

int input();

void float_negative_overflow() {
    float min_float = -FLT_MAX;
    min_float = -min_float * min_float;
}

void int_negative_overflow() {
    int min_int = -2147483648;
}

void main() {
    int choice=input();

    if(choice==0)
        float_negative_overflow();
    else
        int_negative_overflow();
}
```

In this example, a negative overflow is illustrated with two data types.

- In `float_negative_overflow`, `min_float` contains the most negative number that the type `float` can represent. Because the operation `-min_float * min_float` produces a number that is more negative than this number, the type `float` cannot represent it. The **Overflow** check produces a red error.
- In `int_negative_overflow`, the variable `min_int` is assigned the value `-2147483648`. This assignment occurs in three steps:
 - 1 The value `2147483648` is assigned to an unsigned 32-bit integer.
 - 2 The unsigned integer is cast to a signed integer.
 - 3 The unary minus is performed on the signed integer.

An overflow occurs in the second step because the maximum value that a signed integer can have is `2147483647`. Even though the minimum value a signed integer

can have is -2147483648, a red **Overflow** error appears on the operation `int min_int = -2147483648; .`

Overflows on Constants

```
void main() {
    char x = 0xFFFF;
    x=x+1;
}
```

In this example, the constant `0xFFFF` is greater than the maximum value that the type `char` can represent. The **Overflow** check produces a red error.

The following table lists three kinds of constants with the corresponding data types. For each kind, the data type assigned to a constant is the first data type in the corresponding column that can hold the constant.

Decimal	int, long, unsigned long
Hexadecimal	int, unsigned int, long, unsigned long
Float	float, double

For example, (assuming a 16-bit target) the data types for the following values are listed in this table.

5.8	double
6	int
65536	long
0x6	int
0xFFFF	unsigned int
5.8F	float
65536U	unsigned int

To avoid red **Overflow** errors on constants, on the **Configuration** pane, use the analysis option **Check Behavior > Ignore overflowing computations on constants**.

Overflows on Unsigned Bit Fields

```
#include <stdio.h>

struct
{
    unsigned int dayOfWeek : 2;
} Week;

void main()
{
    Week.dayOfWeek = 2;
    Week.dayOfWeek = 3;
    Week.dayOfWeek = 4;
}
```

In this example, `dayOfWeek` occupies 2 bits. It can take values in `[0, 3]` because it is an unsigned integer. When you assign 4 to `dayOfWeek`, the **Overflow** check is red.

To detect overflows on signed and unsigned integers, on the **Configuration** pane, under **Check Behavior**, select `signed-and-unsigned` for **Detect overflows**.

Overflows on Signed and enum Bit Fields

```
enum tBit {
    ZERO = 0x00,
    ONE = 0x01 ,
    TWO = 0x02
};

struct twoBit
{
    enum tBit myBit:2;
} myBitField;

void main()
{
    myBitField.myBit = ZERO;
    myBitField.myBit = ONE;
    myBitField.myBit = TWO;
}
```

In this example, being an enum variable, `myBit` is implemented through a signed integer according to the ANSI C90 standard. `myBit` occupies 2 bits. It can take values in $[-2, 1]$ because it is a signed integer. When you assign 2 to `myBit`, the **Overflow** check is red.

To detect overflows on signed integers alone, on the **Configuration** pane, under **Check Behavior**, select `signed` for **Detect overflows**.

Nonfinite Floats: Infinity Detected with Red Check

Results in `forbid` mode:

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

In this example, both the operands of the `/` operation is not infinite but the result is infinity. The **Overflow** check on the `-` operation is red. In the `forbid` mode, the verification stops after the red check. For instance, a **Non-initialized local variable** check does not appear on `x` in the `return` statement. If you do not turn on the option **Allow non finite floats**, a **Division by zero** check appears because infinities are not allowed.

Results in `warn-first` mode:

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

In this example, both the operands of the `/` operation are not infinite but the result is infinity. The **Overflow** check on the `-` operation is red. The red checks in `warn-first` mode are different from red checks for other check types. The verification does not stop after the red check. For instance, a green **Non-initialized local variable** check appears on `x` in the `return` statement. In the verification result, if you place your cursor on `x`, you see that it has the value `Inf`.

Nonfinite Floats: Infinity Detected with Orange Check

Results in forbid mode:

```
void func(double arg1, double arg2) {
    double ratio1=arg1/arg2;
    double ratio2=arg1/arg2;
}
```

In this example, the values of `arg1` and `arg2` are unknown to the verification. The verification assumes that `arg1` and `arg2` can have all possible double values. For instance, `arg1` can be nonzero and `arg2` can be zero and the result of `ratio1=arg1/arg2` can be infinity. Therefore, an orange **Overflow** check appears on the division operation. Following the check, the verification terminates the execution thread that results in infinity. The verification assumes that `arg2` cannot be zero following the orange check. The **Overflow** check on the second division operation `ratio2=arg1/arg2` is green.

Results in warn-first mode:

```
void func(double arg1, double arg2) {
    double ratio1=arg1/arg2;
    double ratio2=arg1/arg2;
}
```

In this example, the values of `arg1` and `arg2` are unknown to the verification. The verification assumes that `arg1` and `arg2` can have all possible double values. For instance, `arg1` can be non-zero and `arg2` can be zero and the result of `ratio1=arg1/arg2` can be infinity. An orange **Overflow** check appears on the division operation. The orange checks in warn-first mode are different from orange checks for other check types. Following the check, the verification does not terminate the execution thread that results in infinity. The verification retains the zero value of `arg2` following the orange check. Therefore, the **Overflow** check on the second division operation `ratio2=arg1/arg2` is also orange.

Check Information

Group: Numerical

Language: C | C++

Acronym: OVFL

See Also

Polyspace Analysis Options

Detect overflows (-scalar-overflows-checks) | Ignore overflowing computations on constants (-ignore-constant-overflows) | Overflow computation mode (-scalar-overflows-behavior) | Consider non finite floats (-allow-non-finite-floats) | Infinities (-check-infinite)

Polyspace Results

Invalid operation on floats | Subnormal float

Return value not initialized

C function does not return value when expected

Description

This check determines whether a function with a return type other than `void` returns a value. This check appears on every function call.

Diagnosing This Check

“Review and Fix Return Value Not Initialized Checks”

Examples

Function does not return value for given input

```
#include <stdio.h>
int input(void);
int inputRep(void);

int reply(int msg) {
    int rep = inputRep();
    if (msg > 0) return rep;
}

void main(void) {
    int ch = input(), ans;
    if (ch <= 0)
        ans = reply(0);
    else
        ans = reply(ch);
    printf("The answer is %d.",ans);
}
```

In this example, for the function call `reply(0)`, there is no return value. Therefore the **Return value not initialized** check returns a red error. The second call `reply(ch)` always returns a value. Therefore, the check on this call is green.

One possible correction is to return a value for all inputs to `reply()`.

```
#include <stdio.h>
int input();
int inputRep();

int reply(int msg) {
    int rep = inputRep();
    if (msg > 0) return rep;
    return 0;
}

void main(void) {
    int ch = input(), ans;
    if (ch <= 0)
        ans = reply(0);
    else
        ans = reply(ch);
    printf("The answer is %d.",ans);
}
```

Function does not return value for some inputs

```
#include <stdio.h>
int input();
int inputRep(int);

int reply(int msg) {
    int rep = inputRep(msg);
    if (msg > 0) return rep;
}

void main(void) {
    int ch = input(), ans;
    if (ch < 10)
        ans = reply(ch);
    else
        ans = reply(10);
}
```

```
    printf("The answer is %d.",ans);  
}
```

In this example, in the first branch of the `if` statement, the value of `ch` can be divided into two ranges:

- `ch <= 0`: For the function call `reply(ch)`, there is no return value.
- `ch > 0` and `ch < 10`: For the function call `reply(ch)`, there is a return value.

Therefore the **Return value not initialized** check returns an orange error on `reply(ch)`.

One possible correction is to return a value for all inputs to `reply()`.

```
#include <stdio.h>  
int input();  
int inputRep(int);  
  
int reply(int msg) {  
    int rep = inputRep(msg);  
    if (msg > 0) return rep;  
    return 0;  
}  
  
void main(void) {  
    int ch = input(), ans;  
    if (ch < 10)  
        ans = reply(ch);  
    else  
        ans = reply(10);  
    printf("The answer is %d.",ans);  
}
```

Check Information

Group: Data flow

Language: C

Acronym: IRV

See Also

Disable checks for non-initialization (`-disable-initialization-checks`) | Function not returning value

Subnormal float

Floating-point operation has subnormal results

Description

This check determines if a floating-point operation produces a subnormal result.

Subnormal numbers have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the significand. The presence of subnormal numbers indicates loss of significant digits. This loss can accumulate over subsequent operations and eventually result in unexpected values. Subnormal numbers can also slow down the execution on targets without hardware support.

By default, the results of the check do not appear in your verification results. To see the results of the check, change the default value of the option `Subnormal detection mode` (`-check-subnormal`). The results of the check vary based on the detection mode that you specify. In all modes other than `allow`, to identify the subnormal results, look for red or orange **Subnormal float** checks on operations.

Mode	Check Colors	Behavior Following Check
<p><code>forbid:</code></p> <p>This mode detects the occurrence of a subnormal value. This mode stops the execution path with the subnormal result and prevents subnormal values from propagating further. Therefore, in practice, you see only the first occurrence of the subnormal value.</p>	<p>The color of the check depends only on the result of the operation. The check flags an operation that has subnormal results even if those results come only from subnormal operands.</p> <p>For instance, if x is unknown, $x * 2$ can be subnormal because x can be subnormal. The result of the check is orange.</p>	<p>Blocking check.</p> <p>If the check is red, the verification stops. If the check is orange, the verification removes the execution paths containing the subnormal result from consideration. For instance, the tooltip on the result does not show the subnormal values.</p>
<p><code>warn-all:</code></p> <p>This mode highlights all occurrences of subnormal values. Even if a subnormal result comes from previous subnormal values, the result is highlighted.</p>	<p>The color of the check depends only on the result of the operation. The check flags an operation that has subnormal results even if those results come only from subnormal operands.</p> <p>For instance, if x is unknown, $x * 2$ can be subnormal because x can be subnormal. The result of the check is orange.</p>	<p>Non-blocking check.</p> <p>The verification continues even if the check is red. If the check is orange, the verification does not remove the execution paths containing the subnormal result from consideration.</p>

Mode	Check Colors	Behavior Following Check
<p>warn-first:</p> <p>This mode highlights the first occurrence of a subnormal value. If a subnormal value propagates to further subnormal results, those subsequent results are not highlighted.</p>	<p>The check color depends on the result of the operation and the operand values. The check does not flag a subnormal result if it comes only from subnormal operands.</p> <p>In this mode, the check is:</p> <ul style="list-style-type: none"> • Red, if the operation produces subnormal results on all execution paths that the software considers, and the operands are not subnormal. • Orange, if the operation produces subnormal results on some of the execution paths when the operands are not subnormal. <p>For instance, if x is unknown, $x * 0.5$ can be subnormal even if x is not subnormal.</p> • Green, if the operation does not produce subnormal results unless the operands are subnormal. <p>For instance, even if x is unknown, $x * 2$ cannot be subnormal unless x is subnormal.</p>	<p>Non-blocking check.</p> <p>The verification continues even if the check is red. If the check is orange, the verification does not remove the execution paths containing the subnormal result from consideration.</p>

If you choose to check for subnormals, you can also identify from the tooltips whether a variable range excludes subnormal values. For instance, if the tooltips show `[-1.0 ..`

-1.1754E-38] or [-0.0..0.0] or [1.1754E-38..1.0], you can interpret that the variable does not have subnormal values.

Examples

Subnormal Results Detected with Red Checks

In the following examples, `DBL_MIN` is the minimum normal value that can be represented using the type `double`.

Results in `forbid` mode:

```
#include <float.h>

void func(){
    double val = DBL_MIN/4.0;
    double val2 = val * 2.0;
}
```

In this example, the first **Subnormal float** check is red because the result of `DBL_MIN/4.0` is subnormal. The red check stops the verification. The following operation, `val * 2.0`, is not verified for run-time errors.

Results in `warn-all` mode:

```
#include <float.h>

void func(){
    double val = DBL_MIN/4.0;
    double val2 = val * 2.0;
}
```

In this example, both **Subnormal float** checks are red because both operations have subnormal results.

Results in `warn-first` mode:

```
#include <float.h>

void func(){
    double val = DBL_MIN/4.0;
```

```
double val2 = val * 2.0;
}
```

In this example, `DBL_MIN` is not subnormal but the result of `DBL_MIN/4.0` is subnormal. The first **Subnormal float** check is red. The second **Subnormal float** check is green. The reason is that `val * 2.0` is subnormal only because `val` is subnormal. Through red/orange checks, you see only the first instance where a subnormal value appears. You do not see red/orange checks from those subnormal values propagating to subsequent operations.

Subnormal Results Detected with Orange Checks

In the following examples, `arg1` and `arg2` are unknown. The verification assumes that they can take all values allowed for the type `double`.

Results in `forbid` mode:

```
void func (double arg1, double arg2) {
    double difference1 = arg1 - arg2;
    double difference2 = arg1 - arg2;
    double val1 = difference1 * 2;
    double val2 = difference2 * 2;
}
```

In this example, `difference1` can be subnormal if `arg1` and `arg2` are sufficiently close. The first **Subnormal float** check is orange. Following this check, the verification excludes from consideration the following:

- The close values of `arg1` and `arg2` that led to the subnormal value of `difference1`.

In the subsequent operation `arg1 - arg2`, the **Subnormal float** check is green and `difference2` is not subnormal. The result of the check on `difference2 * 2` is green for the same reason.

- The subnormal value of `difference1`.

In the subsequent operation `difference1 * 2`, the **Subnormal float** check is green.

Results in `warn-all` mode:

```
void func (double arg1, double arg2) {
    double difference1 = arg1 - arg2;
```

```
double difference2 = arg1 - arg2;
double val1 = difference1 * 2;
double val2 = difference2 * 2;
}
```

In this example, the four operations can have subnormal results. The four **Subnormal float** checks are orange.

Results in warn-first mode:

```
void func (double arg1, double arg2) {
    double difference1 = arg1 - arg2;
    double difference2 = arg1 - arg2;
    double val1 = difference1 * 2;
    double val2 = difference2 * 2;
}
```

In this example, if `arg1` and `arg2` are sufficiently close, `difference1` and `difference2` can be subnormal. The first two **Subnormal float** checks are orange. `val1` and `val2` cannot be subnormal unless `difference1` and `difference2` are also subnormal. The last two **Subnormal float** checks are green. Through red/orange checks, you see only the first instance where a subnormal value appears. You do not see red/orange checks from those subnormal values propagating to subsequent operations.

Result Information

Group: Numerical

Language: C | C++

Acronym: SUBNORMAL

See Also

Polyspace Analysis Options

Subnormal detection mode (`-check-subnormal`)

Polyspace Results

Invalid operation on floats | Overflow

Introduced in R2016b

Uncaught exception

Exception handling

Description

This check determines whether a function call throws an uncaught exception.

This check appears on both a function call as well as the function body. Use this check to follow the propagation of an exception from an entry-point function down the branches of the call tree.

Note The **Uncaught exception** check on functions from the Standard Template Library such as `operator new` is green, even though Polyspace stubs these functions and does not check if a function throws an exception. To prevent the stubbing, use the option `No STL stubs (-no-stl-stubs)`.

Diagnosing This Check

“Review and Fix Uncaught Exception Checks”

Examples

Exception in call to function

```
#include <vector>

class error {};

class initialVector {
private:
    int sizeVector;
    vector<int> table;
public:
```

```
    initialVector(int size) {
        sizeVector = size;
        table.resize(sizeVector);
        Initialize();
    }
    void Initialize();
    int getValue(int number) throw(error);
};

void initialVector::Initialize() {
    for(int i=0; i<table.size(); i++)
        table[i]=0;
}

int initialVector::getValue(int index) throw(error) {
    if(index >= 0 && index < sizeVector)
        return table[index];
    else throw error();
}

void main() {
    initialVector *vectorPtr = new initialVector(5);
    vectorPtr->getValue(5);
}
```

In this example, the call to method `initialVector::getValue` throws an exception. This exception appears as a red **Uncaught exception** error on both the function call and the return type in the function body. A red **Uncaught exception** error also appears on main because a function call inside main throws an exception.

Exception handled through `try/catch` construct

```
class error {
public:
    error() { }
    error(const error&) { }
};

void funcNegative() {
    try {
        throw error() ;
    } catch (error NegativeError) {
```

```
    }
}

void funcPositive() {
    try {
    }
    catch (error PositiveError) {
        /* Gray code */
    }
}

int input();
void main()
{
    int val=input();
    if(val < 0)
        /* Green check */
        funcNegative();
    else
        /* Green check */
        funcPositive();
}
```

In this example:

- The call to `funcNegative` throws an exception. However, the exception is placed inside a `try` block and is caught by the corresponding handler (`catch` clause). The **Uncaught exception** check on the function body, function call, and the `main` function appears green.
- The call to `funcPositive` does not throw an exception in the `try` block. Therefore, the `catch` block following the `try` block appears gray.

Exception in call to constructor

```
class error {
};

class X
{
public:
    X() {
        throw error();
    }
}
```

```
    ~X() {
        ;
    }
};

void main() {
    try {
        X * px = new X ;
        delete px;
    } catch (error) {
        assert(1) ;
    }
}
```

In this example, the `new` operator calls the constructor `X::X()`. The constructor throws an exception. The exception appears as a red **Uncaught exception** error on the constructor body and the `new` operator. The exception then propagates to the `catch` block and does not continue farther. Therefore the **Uncaught exception** check on the `main` function appears green.

The green `assert` statement shows that the exception has propagated to the `catch` block.

Exception in call to destructor

```
class error {
};

class X
{
public:
    X() {
        ;
    }
    ~X() {
        throw error();
    }
};

int main() {
    try {
        X * px = new X ;
        delete px;
    }
}
```



```

    } catch (error) {
        assert(1) ;
    }
}

```

In this example, the `delete` operator calls the destructor `X::~X()`. The destructor throws an exception that appears as a red error on the destructor body and dashed red on the `delete` operator. The exception does not propagate to the `catch` block. The code following the exception is not verified. This behavior enforces the requirement that a destructor must not throw an exception.

The black `assert` statement suggests that the exception has not propagated to the `catch` block.

Exception in infinite loop

```

#include<stdio.h>
#define SIZE 100

int arr[SIZE];
int getIndex();

int runningSum() {
    int index, sum=0;
    while(1) {
        index=getIndex();
        if(index < 0 || index >= SIZE)
            throw int(1);
        sum+=arr[index];
    }
}

void main() {
    printf("The sum of elements is: %d",runningSum());
}

```

In this example, the `runningSum` function throws an exception only if `index` is outside the range `[0, SIZE]`. Typically, an error that occurs due to instructions in an `if` statement is orange, not red. The error is orange because an alternate execution path that does not involve the `if` statement does not produce an error. Here, because the loop is infinite, there is no alternate execution path that goes outside the loop. The only way to go outside the loop is through the exception in the `if` statement. Therefore, the **Uncaught exception** error is red.

Type mismatch between `throw` declaration and usage

```
#include <string>

class negativeBalance {
public:
    negativeBalance(const string & s): errorMessage(s) {}
    ~negativeBalance() {}
private:
    string errorMessage;
};

class Account {
public:
    Account(long initVal):balance(initVal) {}
    ~Account() {}
    void debitAccount(long debitAmount) throw (int, char);
private:
    long balance;
};

void Account::debitAccount(long debitAmount) throw (int, char) {
    if((balance - debitAmount) < 0 )
        throw negativeBalance("Negative balance");
    else
        balance -= debitAmount;
}

void main() {
    Account *myAccount = new Account(1000);
    try {
        myAccount->debitAccount(2000);
    }
    catch(negativeBalance &) {
    }
    delete myAccount;
}
```

In this example, the types associated with the `throw` statement in the `Account::debitAccount` method are `int` and `char`. However, the method throws an exception with type `negativeBalance`. Therefore, the **Uncaught exception** check produces a red error on `throw`.

Rethrow outside catch block

```
#include <string>

void f() { throw; }           //rethrow not allowed - an error is raised here
void main() {
    try {
        throw std::string("hello");
    }
    catch (std::string& exc) {
        f();
    }
}
```

In this example, an exception is rethrown in the function `f()` outside a `catch` block. A rethrow occurs when you call `throw` by itself without an exception argument. A rethrow is typically used *inside* a `catch` block to propagate an exception to an outer `try-catch` sequence. Polyspace Code Prover does not support a rethrow *outside* a `catch` block and produces a red **Uncaught exception** error.

Check Information

Group: C++

Language: C++

Acronym: EXC

Unreachable code

Code cannot be reached during execution

Description

This check determines whether a section of code can be reached during execution.

Examples of unreachable code include the following:

- If a test condition always evaluates to false, the corresponding code branch cannot be reached. On the **Source** pane, the opening brace of the branch is gray.
- If a test condition always evaluates to true, the condition is redundant. On the **Source** pane, the condition keyword such as `if` appears gray.
- The code follows a `break` or `return` statement.

If an opening brace of a code block appears gray on the **Source** pane, to highlight the entire block, double-click the brace.

The check operates on code inside a function. The checks **Function not called** and **Function not reachable** determine if the function itself is not called or called from unreachable code.

Diagnosing This Check

“Review and Fix Unreachable Code Checks”

Examples

Test in `if` Statement Always False

```
#define True 1
#define False 0
```

```

typedef enum {
    Intermediate, End, Wait, Init
} enumState;

enumState input();
enumState inputRef();
void operation(enumState, int);

int checkInit (enumState stateval) {
    if (stateval == Init)
        return True;
    return False;
}

int checkWait (enumState stateval) {
    if (stateval == Wait)
        return True;
    return False;
}

void main() {
    enumState myState = input(), refState = inputRef() ;
    if (checkInit (myState)) {
        if (checkWait (myState)) {
            operation (myState, checkInit (refState));
        } else {
            operation (myState, checkWait (refState));
        }
    }
}

```

In this example, the main enters the branch of `if (checkInit (myState))` only if `myState = Init`. Therefore, inside that branch, Polyspace considers that `myState` has value `Init`. `checkWait (myState)` always returns `False` and the first branch of `if (checkWait (myState))` is unreachable.

One possible correction is to remove the redundant test `if (checkWait (myState))`.

```

#define True 1
#define False 0

typedef enum {
    Intermediate, End, Wait, Init

```

```
    } enumState;

enumState input();
enumState inputRef();
void operation(enumState, int);

int checkInit (enumState stateval) {
    if (stateval == Init)
        return True;
    return False;
}

int checkWait (enumState stateval) {
    if (stateval == Wait) return True;
    return False;
}

void main() {
    enumState myState = input(), refState = inputRef() ;
    if (checkInit (myState))
        operation (myState, checkWait (refState));
}
```

Test in `if` Statement Always True

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

int roll() {
    return(rand()%6+1);
}

void operation(int);

void main() {
    srand(time(NULL));
    int die = roll();
    if(die >= 1 && die <= 6)
        /*Unreachable code*/
        operation(die);
}
```

In this example, `roll()` returns a value between 1 and 6. Therefore the `if` test in `main` always evaluates to true and is redundant. If there is a corresponding `else` branch, the

gray error appears on the `else` statement. Without an `else` branch, the gray error appears on the `if` keyword to indicate the redundant condition.

One possible correction is to remove the condition `if(die >= 1 && die <=6)`.

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

int roll() {
    return(rand()%6+1);
}

void operation(int);

void main() {
    srand(time(NULL));
    int die = roll();
    operation(die);
}
```

Test in `if` Statement Unreachable

```
#include <stdlib.h>
#include <time.h>
#define True 1
#define False 0

int roll1() {
    return(rand()%6+1);
}

int roll2();
void operation(int,int);

void main() {
    srand(time(NULL));
    int die1 = roll1(),die2=roll2();
    if((die1>=1 && die1<=6) ||
        (die2>=1 && die2 <=6))
        /*Unreachable code*/
        operation(die1,die2);
}
```

In this example, `roll1()` returns a value between 1 and 6. Therefore, the first part of the `if` test, `if((die1>=1) && (die1<=6))` is always true. Because the two parts of the `if` test are combined with `||`, the `if` test is always true irrespective of the second part. Therefore, the second part of the `if` test is unreachable.

One possible correction is to combine the two parts of the `if` test with `&&` instead of `||`.

```
#include <stdlib.h>
#include <time.h>
#define True 1
#define False 0

int roll1() {
    return(rand()%6+1);
}

int roll2();
void operation(int,int);

void main()    {
    srand(time(NULL));
    int die1 = roll1(),die2=roll2();
    if((die1>=1 && die1<=6) &&
        (die2>=1 && die2<=6))
        operation(die1,die2);
}
```

Check Information

Group: Data flow

Language: C | C++

Acronym: UNR

See Also

Function not called | Function not reachable

User assertion

`assert` statement fails

Description

This check determines whether the argument to an `assert` macro is true.

The argument to the `assert` macro must be true when the macro executes. Otherwise the program aborts and prints an error message. Polyspace models this behavior by treating a failed `assert` statement as a run-time error. This check allows you to detect failed `assert` statements before program execution.

Diagnosing This Check

“Review and Fix User Assertion Checks”

Examples

Red `assert` on array index

```
#include<stdio.h>
#define size 20

int getElement();

void initialize(int* array) {
    for(int i=0;i<size;i++)
        array[i] = getElement();
}

void printElement(int* array,int index) {
    assert(index < size);
    printf("%d", array[index]);
}
```

```
int getIndex() {
    int i = size;
    return i;
}

void main() {
    int array[size];
    int index;

    initialize(array);
    index = getIndex();
    printElement(array, index);
}
```

In this example, the `assert` statement in `printElement` causes program abort if `index >= size`. The `assert` statement makes sure that the array index is not outside array bounds. If the code does not contain exceptional situations, the `assert` statement must be green. In this example, `getIndex` returns an index equal to `size`. Therefore the `assert` statement appears red.

When an `assert` statement is red, investigate the cause of the exceptional situation. In this example, one possible correction is to force `getIndex` to return an index equal to `size-1`.

```
#include<stdio.h>
#define size 20

int getArrayElement();

void initialize(int* array) {
    for(int i=0;i<size;i++)
        array[i] = getArrayElement();
}

void printElement(int* array,int index) {
    assert(index < size);
    printf("%d", array[index]);
}

int getIndex() {
    int i = size;
```

```
    return (i-1);
}

void main() {
    int array[size];
    int index;

    initialize(array);
    index = getIndex();
    printElement(array, index);
}
```

Orange assert on malloc return value

```
#include <stdlib.h>

void initialize(int*);
int getNumberOfElements();

void main() {
    int numberOfElements, *myArray;

    numberOfElements = getNumberOfElements();

    myArray = (int*)malloc(numberOfElements);
    assert(myArray!=NULL);

    initialize(myArray);
}
```

In this example, malloc can return NULL to myArray. Therefore, myArray can have two possible values:

- myArray == NULL: The assert condition is false.
- myArray != NULL: The assert condition is true.

Combining these two cases, the **User assertion** check on the assert statement is orange. After the orange assert, Polyspace considers that myArray is not equal to NULL.

One possible correction is to write a customized function myMalloc where you always check the return value of malloc for NULL.

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

void initialize(int*);
int getNumberOfElements();

void myMalloc(int **ptr, int num) {
    *ptr = (int*)malloc(num);
    if(*ptr==NULL) {
        printf("Memory allocation error");
        exit(1);
    }
}

void main() {
    int numberOfElements, *myArray=NULL;

    numberOfElements = getNumberOfElements();

    myMalloc(&myArray,numberOfElements);
    assert(myArray!=NULL);

    initialize(myArray);
}
```

Imposing constraint through orange assert

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

float getNumber();
void squareRootOfDifference(float firstNumber, float secondNumber) {
    assert(firstNumber > secondNumber);
    if(firstNumber > 0 && secondNumber > 0)
        printf("Square root = %.2f",sqrt(firstNumber-secondNumber));
}

void main() {
    double firstNumber = getNumber(), secondNumber = getNumber();
    squareRootOfDifference(firstNumber,secondNumber);
}
```

In this example, the `assert` statement in `squareRootOfDifference()` causes program abort if `firstNumber` is less than `secondNumber`. Because Polyspace does not

have enough information about `firstNumber` and `secondNumber`, the `assert` is orange.

Following the `assert`, all execution paths that cause assertion failure terminate. Therefore, following the `assert`, Polyspace considers that `firstNumber >= secondNumber`. The **Invalid use of standard library routine** check on `sqrt` is green.

Use `assert` statements to help Polyspace determine:

- Relationships between variables
- Constraints on variable ranges

Check Information

Group: Other

Language: C | C++

Acronym: ASRT

Approximations Used During Verification

- “Why Polyspace Verification Uses Approximations” on page 4-2
- “Variable Ranges” on page 4-3
- “Stubbed Functions” on page 4-4
- “Initialization of Global Variables” on page 4-11
- “Volatile Variables” on page 4-13
- “Definitions and Declarations” on page 4-15
- “Implicit Data Type Conversions” on page 4-16
- “Using memset and memcpy” on page 4-19
- “#pragma Directives” on page 4-23
- “Standard Library Float Routines” on page 4-25
- “Unions” on page 4-26
- “Variable Cast as Void Pointer” on page 4-28
- “Assembly Code” on page 4-29
- “Determination of Program Stack Usage” on page 4-35
- “Limitations of Polyspace Verification” on page 4-40

Why Polyspace Verification Uses Approximations

Polyspace Code Prover uses *static verification* to prove the absence of run-time errors. Static verification derives the dynamic properties of a program without actually executing it. Static verification differs significantly from other techniques such as run-time debugging because the verification does not rely on a specific test case or set of test cases. The properties obtained from static verification are true for *all* executions of your program¹.

Static verification uses representative approximations of software operations and data. For instance, consider the following code:

```
for (i=0 ; i<1000 ; ++i) {  
    tab[i] = foo(i);  
}
```

To check that the variable `i` never overflows the range of `tab`, one approach can be to consider each possible value of `i`. This approach requires a thousand checks.

In static verification, the software models a variable by its domain. In this case, the software models that `i` belongs to the static interval, `[0..999]`. Depending on the complexity of the data, the software uses more elaborate models such as convex polyhedrons or integer lattices for this purpose.

An approximation, by definition, leads to information loss. For instance, the verification loses the information that `i` is incremented by one every cycle in the loop. However, even without this information, it is possible to ensure that the variation domain of `i` is smaller than the range of `tab`. Only one check is required to establish this property. Therefore, static verification is more efficient compared to traditional approaches.

When performing approximations, the verification does not compromise with exhaustiveness. The reason is that the approximations performed are upper approximations or over-approximations. In other words, the computed domain of a variable is a superset of its actual domain.

-
1. The properties obtained from static verification hold true only if you execute your program under the same conditions that you specified through the analysis options. For instance, the default verification assumes that pointers obtained from external sources are non-null. Unless you specify the option `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`, the verification results are obtained under this assumption. They might not hold true during program execution if the assumption is invalidated and a null pointer is obtained from an external source.

Variable Ranges

If Polyspace cannot determine a variable value from the code, it assumes that the variable has a full range of values allowed by its type.

For instance, for a variable of integer type, to determine the minimum and maximum value allowed, Polyspace uses the following criteria:

- The C standard specifies that the range of a signed n -bit integer-type variable must be at least $[-(2^{n-1}-1), 2^{n-1}-1]$.

The **Target processor type** that you specify determines the number of bits allocated for a certain type. For more information, see `Target processor type (-target)`.

- Polyspace assumes that your target uses the two's complement representation for signed integers. The software uses this representation to determine the exact range of a variable. In this representation, the range of a signed n -bit integer-type variable is $[-2^{n-1}, 2^{n-1}-1]$.

For example, for an `i386` processor:

- A `char` variable has 8 bits. The C standard specifies that the range of the `char` variable must be at least $[-127,127]$.
- Using the two's complement representation, Polyspace assumes that the exact range of the `char` variable is $[-128,127]$.

To determine the range that Polyspace assumes for a certain type:

- 1 Run verification on this code. Replace `type` with the type name such as `int`.

```
type getVal(void);
void main() {
    type val = getVal();
}
```

- 2 Open your verification results. On the **Source** pane, place your cursor on `val`.

The tooltip provides the range that Polyspace assumes for `type`. Since `getVal` is not defined, Polyspace assumes that the return value of `getVal` has full range of values allowed by `type`.

Stubbed Functions

The verification stubs functions that are not defined in your source code or that you choose to stub. For a stubbed function:

- The verification makes certain assumptions about the function return value and other side effects of the function.

You can fine-tune the assumptions by specifying constraints.

- The verification ignores the function body if it exists. Operations in the function body are not checked for run-time errors.

If the verification of a function body is imprecise and causes many orange checks when you call the function, you can choose to stub the function. To reduce the number of orange checks, you stub the function, and then constrain the return value of the function and specify other side effects.

To stub functions, you can use these options:

- `Functions to stub (-functions-to-stub)`: Specify functions that you want stubbed.
- `Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions)`: Stub functions that contain lookup tables in code generated from models using Embedded Coder®.
- `-function-behavior-specifications`: Stub functions that correspond to a standard function that Polyspace recognizes.

If you use the first option to stub a function, you constrain the function return value and model other side effects by specifying constraints. If you want to specify constraints more fine-grained than the ones available through the Polyspace constraint specification interface, define your own stubs. If you use the other options to stub functions, the software itself constrains the function return value and models its side effects appropriately.

The verification makes the following assumptions about the arguments of stubbed functions.

Function Return Value

Assumptions

The verification assumes that:

- The variable returned by the function takes the full range of values allowed by its data type.

If the function returns an enum variable, the variable value is in the range of the enum. For instance, if an enum type takes values {0,5,-1,32} and a stubbed function has that return type, the verification assumes that the function returns values in the range -1..32.

- If the function returns a pointer, the pointer is not NULL and safe to dereference. The pointer does not point to dynamically allocated memory or another variable in your code.
- C++ specific assumptions: The operator new returns allocated memory. Operators such as operator=, operator+=, operator--(prefixed version) or operator<< returns:

- A reference to *this, if the operator is part of a class definition.

For instance, if an operator is defined as:

```
class X {
    X& operator=(const X& arg) ;
};
```

It returns a reference to *this (the object that calls the operator). The object that calls the operator or its data members have the full range of values allowed by their type.

- The first argument, if the operator is not part of a class definition.

For instance, if an operator is defined as:

```
X& operator+=(X& arg1, const X& arg2) ;
```

It returns arg1. The object that arg1 refers to or its data members have the full range of values allowed by their type.

Functions declared with `__declspec(no_return)` (Visual Studio) or `__attribute__((noreturn))` (GCC) do not return.

How to Change Assumptions

You can change the default assumptions about the function return value.

- If the function returns a non-pointer variable, you can constrain its range. See “Constraints”.

Through the constraint specification interface, you can specify an absolute range $[min..max]$. To specify more complicated constraints, write a function stub.

For instance, an undefined function has the prototype:

```
int func(int ll, int ul);
```

Suppose you know that the function return value lies between the first and the second arguments. However, the software assumes full range for the return value because the function is not defined. To model the behavior that you want and reduce orange checks from the imprecision, write a function stub as follows:

```
int func(int ll, int ul) {
    int ret;
    assert(ret>=ll && ret <=ul);
    return ret;
}
```

Provide the function stub in a separate file for verification. The verification uses your stub as the function definition.

If the definition of `func` exists in your code and you want to override the definition because the verification of the function body is imprecise, embed the actual definition and the stub in a `#ifdef` statement:

```
#ifdef POLYSPACE
int func(int ll, int ul) {
    int ret;
    assert(ret>=ll && ret <=ul);
    return ret;
}
#else
int func(int ll, int ul) {
    /*Your function body */
}
#endif
```

Define the macro `POLYSPACE` by using the option `Preprocessor definitions (-D)`. The verification uses your stub instead of the actual function definition.

- If the function returns a pointer variable, you can specify that the pointer might be `NULL`.
 - To specify this assumption for all stubbed functions, use the option `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`.
 - To specify this assumption for specific stubbed functions, use the constraint specification interface. See “Constraints”.

Function Arguments That are Pointers

Assumptions

The verification assumes that:

- If the argument is a pointer, the function can write any value to the object that the pointer points to. The range of values is constrained by the argument data type alone.

For instance, in this example, the verification assumes that the stubbed function `stubbedFunc` writes any possible value to `val`. Therefore, the assertion is orange.

```
void stubbedFunc(int*);

void main() {
    int val=0, *ptr=&val;
    stubbedFunc(ptr);
    assert(val==0);
}
```

- If the argument is a pointer to a structure, the function can write any value to the structure fields. The range of values is constrained only by the data type of the fields.
- If the argument is a pointer to another pointer, the function can write any value to the object that the second pointer points to (C code only). This assumption continues to arbitrary depths of a pointer hierarchy.

For instance, suppose that a pointer `**pp` points to another pointer `*p`, which points to an `int` variable `var`. If a stubbed function takes `**p` as argument, the verification assumes that following the function call, `var` has any `int` value. `*p` can point to anywhere in allocated memory or can point to `var` but does not point to another variable in the code.

- If the argument is a function pointer, the function that it points to gets called (C code only).

For instance, in this example, the stubbed function `stubbedFunc` takes a function pointer `funcPtr` as argument. `funcPtr` points to `func`, which gets called when you call `stubbedFunc`.

```
typedef int (*typeFuncPtr) (int);

int func(int x){
    return x;
}

int stubbedFunc(typeFuncPtr);

void main() {
    typeFuncPtr funcPtr = (typeFuncPtr)(&func);
    int result = stubbedFunc(funcPtr);
}
```

If the function pointer takes another function pointer as argument, the function that the second function pointer points to gets stubbed.

How to Change Assumptions

You can constrain the range of the argument that is passed by reference. See “Constraints”.

Through the constraint specification interface, you can specify an absolute range [*min*..*max*]. To specify more complicated constraints, write a function stub.

For instance, an undefined function has the prototype:

```
void func(int *x, int ll, int ul);
```

Suppose you know that the value written to `x` lies between the second and the third arguments. However, the software assumes full range for the value of `*x` because the function is not defined. To model the behavior that you want and reduce orange checks from the imprecision, write a function stub as follows:

```
void func(int *x, int ll, int ul) {
    assert(*x>=ll && *x <=ul);
}
```

Provide the function stub in a separate file for verification. The verification uses your stub as the function definition.

If the definition of `func` exists in your code and you want to override the definition because the verification of the function body is imprecise, embed the actual definition and the stub in a `#ifdef` statement:

```
#ifdef POLYSPACE
void func(int *x, int ll, int ul) {
    assert(*x>=ll && *x <=ul);
}
#else
void func(int *x, int ll, int ul) {
    /* Your function body */
}
#endif
```

Define the macro `POLYSPACE` by using the option `Preprocessor definitions (-D)`. The verification uses your stub instead of the actual function definition.

Global Variables

Assumptions

The verification assumes that the function stub does not modify global variables.

How to Change Assumptions

To model write operations on a global variable, write a function stub.

For instance, an undefined function has the prototype:

```
void func(void);
```

Suppose you know that the function writes the value 0 or 1 to a global variable `glob`. To model the behavior that you want, write a function stub as follows:

```
void func(void) {
    volatile int randomVal;
    if(randomVal)
        glob = 0;
    else
        glob = 1;
}
```

Provide the function stub in a separate file for verification. The verification uses your stub as the function definition.

If the definition of `func` exists in your code and you want to override the definition because the verification of the function body is imprecise, embed the actual definition and the stub in a `#ifdef` statement as follows:

```
#ifdef POLYSPACE
void func(void) {
    volatile int randomVal;
    if(randomVal)
        glob = 0;
    else
        glob = 1;
}
#else
void func(void) {
    /* Your function body */
}
#endif
```

Define the macro `POLYSPACE` using the option `Preprocessor definitions (-D)`. The verification uses your stub instead of the actual function definition.

Initialization of Global Variables

Unless you use the option `Ignore default initialization of global variables (-no-def-init-glob)`, Polyspace considers that global variables are initialized according to ANSI C standards. The default values are:

- 0 for `int`
- 0 for `char`
- 0.0 for `float`

If you define global variables in your code, the software uses the dummy function `_init_globals()` to initialize the global variables. The `_init_globals()` function is the first function called in the main function.

Consider the following code in the application `gv_example.c`.

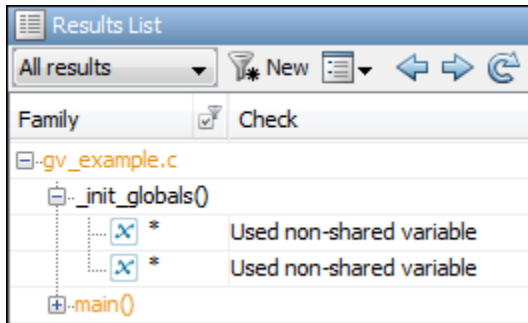
```
extern int func(int); /* External function */

/* Global variables initialized in _init_globals() */
/* before the execution of main() procedure */
int garray[3] = {1, 2, 3};
/* Initialized: written in __init_globals() */
int gvar = 12;
/* Initialized: written in __init_globals() */

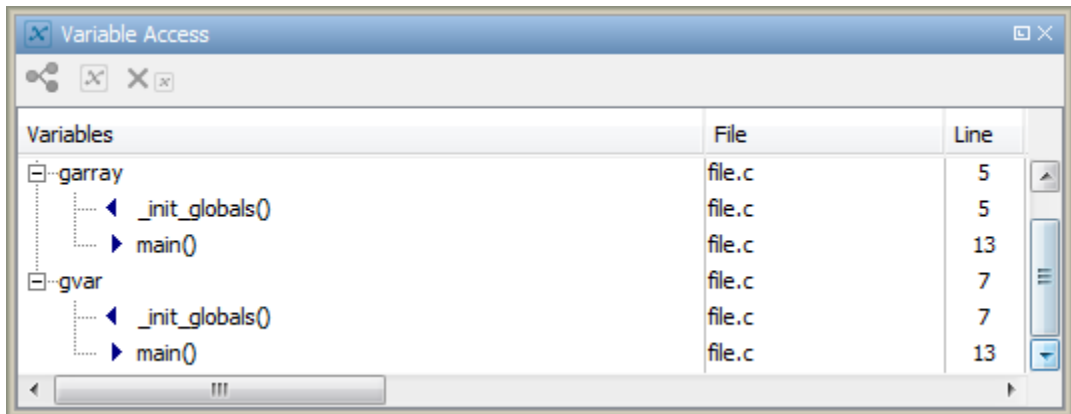
int main(void) {
    int i, lvar = 0;
    for (i = 0; i < 3; i++)
        lvar += func(garray[i] + gvar);
    return lvar;
}
```

After verification:

- On the **Results List** pane, if you select **File** from the  list, under the node `gv_example.c`, you see `_init_globals`.



- On the **Variable Access** pane, `gv_example._init_globals` represents the first write operation on a global variable, for example, `garray`. In the **Values** column, the corresponding value represents the value of the global variable immediately after initialization.



Volatile Variables

The values of volatile variables can change without explicit write operations.

For local volatile variables:

- Polyspace assumes that the variable has a full range of values allowed by its type.
- Unless you explicitly initialize the variable, when you read the variable, Polyspace produces an orange **Non-initialized local variable** check.

In this example, Polyspace assumes that `val1` is potentially noninitialized but `val2` is initialized. Polyspace considers that the `+` operation can cause an overflow because it assumes both variables to have full range of values.

```
int func (void)
{
    volatile int val1, val2=0;
    return( val1 + val2);
}
```

For global volatile variables:

- Polyspace assumes that the variable has a full range of values allowed by its type.

You can constrain the range externally. See “Constrain Global Variable Range”.

- Even if you do not explicitly initialize the variable, when you read the variable, Polyspace produces a green **Non-initialized variable** check.

If the root cause of an orange check is a local volatile variable, you cannot override the default assumptions and constrain the values of the volatile variables. Instead try one of the following:

- If the volatile variable represents hardware-supplied data, see if you can use a function call to model this data retrieval. For example, replace `volatile int port_A` with `int port_A = read_location()`. You do not have to define the function. Polyspace stubs the undefined functions. You can then specify constraints on the function return values. See “Constrain Stubbed Functions”.
- See if you can copy the contents of the volatile variable to a global nonvolatile variable. You can then constrain the global variable values throughout your code. See “Constrain Global Variable Range”.

- Replace the volatile variable with a stubbed function, but only for verification. Before verification, specify constraints on the stubbed functions.

- 1 Write a Perl script that replaces each volatile variable declaration with a nonvolatile declaration where you obtain the variable value from a function call.

For example, if your code contains the line `volatile s8 PORT_A`, your Perl script can contain this substitution:

```
$line=~ s/^\s*volatile\s*s8\s*PORT_A;/s8 PORT_A = random_s8();/g;
```

- 2 Specify the location of this Perl script for the analysis option `Command/script` to apply to preprocessed files (`-post-preprocessing-command`).
- 3 In an include file, provide the function declaration. For example, for a function `random_s8`, the include file can contain the following declaration:

```
#ifndef POLYSPACE_H
#define POLYSPACE_H
signed char random_s8(void);
#endif
```

- 4 Insert a `#include` directive for your include file in the relevant source files

Instead of a manual insertion, specify the location of your include file for the analysis option `Include (-include)`.

Definitions and Declarations

The definition and declaration of a variable are two different but related operations.

Definition

- If you define a function it means that the body of the function is written: `int f(void) { return 0; }`
- If you define a variable, it means that a part of memory is reserved for the variable: `int x;` or `extern int x=0;`

When a variable is not defined, the software considers the variable to be initialized, and to have potentially any value in its full range.

When a function is not defined, the software stubs the function.

Declaration

- Function declaration: `int f(void);`
- Variable declaration: `extern int x;`

A declaration provides information about the type of the function or variable. If you use the function or variable in a file where it has not been declared, a compilation error results.

Implicit Data Type Conversions

If an operation involves two operands, the verification assumes that before the operation takes place, the operands can undergo implicit data type conversion. Whether this conversion happens depends on the original data types of the operands.

Following are the conversion rules that apply if the operands in a binary operation have the same data type. Both operands can be converted to `int` or `unsigned int` type before the operation is performed. This conversion is called integer promotion. The conversion rules are based on the ANSI C99 Standard.

- `char` and `signed short` variables are converted to `int` variables.
- `unsigned short` variables are converted to `int` variables only if an `int` variable can represent all possible values of an `unsigned short` variable.

For targets where the size of `int` is the same as size of `short`, `unsigned short` variables are converted to `unsigned int` variables. For more information on data type sizes, see `Target processor type (-target)`.

- Types such as `int`, `long` and `long long` remain unchanged.

Following are some of the conversion rules that apply when the operands have different data types. The rules are based on the ANSI C99 Standard.

- If both operands are `signed` or `unsigned`, the operand with a lower-ranked data type is converted to the data type of the operand with the higher-ranked type. The rank increases in the order `char`, `short`, `int`, `long`, and `long long`.
- If one operand is `unsigned` and the other `signed`, and the `unsigned` operand data type has a rank higher or the same as the `signed` operand data type, the `signed` operand is converted to the `unsigned` operand type.

For instance, if one operand has data type `int` and the other has type `unsigned int`, the `int` operand is converted to `unsigned int`.

Implicit Conversion When Operands Have Same Data Type

This example shows implicit conversions when the operands in a binary operation have the same data type. If you run verification on the examples, you can use tooltips on the **Source** pane to see the conversions.

In the first addition, `i1` and `i2` are not converted before the addition. Their sum can have values outside the range of an `int` type because `i1` and `i2` have full-range values. Therefore, the Overflow on page 3-93 check on the first addition is orange.

In the second addition, `c1` and `c2` are promoted to `int` before the addition. The addition does not overflow because an `int` variable can represent all values that result from the sum of two `char` variables. The **Overflow** check on the second addition is green. However, when the sum is assigned to a `char` variable, an overflow occurs during the conversion from `int` back to `char`. An orange **Overflow** check appears on the `=` operation.

```
extern char input_char(void);
extern int input_int(void);

void main(void) {
    char c1 = input_char();
    char c2 = input_char();
    int i1 = input_int();
    int i2 = input_int();

    i1 = i1 + i2;
    c1 = c1 + c2;
}
```

Implicit Conversion When Operands Have Different Data Types

The following examples show implicit conversions that happen when the operands in a binary operation have different data types. If you run verification on the examples, you can use tooltips on the **Source** pane to see the conversions.

In this example, before the `<=` operation, `x` is implicitly converted to unsigned `int`. Therefore, the User assertion on page 3-125 check is red.

```
#include <assert.h>
int func(void) {
    int x = -2;
    unsigned int y = 5;
    assert(x <= y);
}
```

In this example, in the first `assert` statement, `x` is implicitly converted to unsigned `int` before the operation `x <= y`. Because of this conversion, in the second `assert`

statement, x is greater than or equal to zero. The User assertion on page 3-125 check on the second `assert` statement is green.

```
int input(void);

void func(void) {
    unsigned int y = 7;
    int x = input();
    assert ( x >= -7 && x <= y );
    assert ( x >=0 && x <= 7);
}
```


Using memset and memcpy

In this section...

“Polyspace Specifications for memcpy” on page 4-19

“Polyspace Specifications for memset” on page 4-20

Polyspace Specifications for memcpy

Syntax:

```
#include <string.h>
void * memcpy ( void * destinationPtr, const void * sourcePtr, size_t num );
```

If your code uses the memcpy function, see the information in this table.

Specification	Example
Polyspace runs a Invalid use of standard library routine check on the function. The check determines if the memory block that sourcePtr or destinationPtr points to is greater than or equal in size to the memory assigned to them through num.	<pre>#include <string.h> typedef struct { char a; int b; } S; void func(int); void main() { S s; int d; memcpy(&d, &s, sizeof(S)); }</pre> <p>In this code, Polyspace produces a red Invalid use of standard library routine error because:</p> <ul style="list-style-type: none"> • d is an int variable. • sizeof(S) is greater than sizeof(int). • A memory block of size sizeof(S) is assigned to &d.

Specification	Example
<p>Polyspace does not check if the memory that sourcePtr points to is itself initialized.</p> <p>Following the use of memcpy, Polyspace considers that the variables that destinationPtr points to can have any value allowed by their type.</p>	<pre>#include <string.h> typedef struct { char a; int b; } S; void func(int); void main() { S s, d={'a',1}; int val; val = d.b; // val=1 memcpy(&d, &s, sizeof(S)); val = d.b; // val can have any int value }</pre> <p>In this code, when the memcpy function copies s to d, Polyspace does not produce a red Non-initialized local variable error. Following the copy, the verification considers that the fields of d can have any value allowed by their type. For instance, d.b can have any value in the range allowed for an int variable.</p>

Polyspace Specifications for memset

Syntax:

```
#include <string.h>
void * memset ( void * ptr, int value, size_t num );
```

If your code uses the memset function, see the information in this table.

Specification	Example
<p>Polyspace runs a Invalid use of standard library routine check on the function. The check determines if the memory block that <code>ptr</code> points to is greater than or equal in size to the memory assigned to them through <code>num</code>.</p>	<pre data-bbox="795 305 1169 586">#include <string.h> typedef struct { char a; int b; } S; void main() { int val; memset(&val,0,sizeof(S)); }</pre> <p data-bbox="795 618 1332 708">In this code, Polyspace produces a red Invalid use of standard library routine error because:</p> <ul data-bbox="795 739 1299 927" style="list-style-type: none"> • <code>val</code> is an <code>int</code> variable. • <code>sizeof(S)</code> is greater than <code>sizeof(int)</code>. • A memory block of size <code>sizeof(S)</code> is assigned to <code>&val</code>.
<p>If <code>value</code> is 0, following the use of <code>memset</code>, Polyspace considers that the variables that <code>ptr</code> points to have the value 0.</p>	<pre data-bbox="795 947 1139 1281">#include <string.h> typedef struct { char a; int b; } S; void main() { S s; int val; memset(&s,0,sizeof(S)); val=s.b; //val=0 }</pre> <p data-bbox="795 1312 1323 1409">In this code, Polyspace considers that following the use of <code>memset</code>, each field of <code>s</code> has value 0.</p>

Specification	Example
<p>Following the use of <code>memset</code>, if value is anything other than 0, Polyspace considers that:</p> <ul style="list-style-type: none"> • The variables that <code>ptr</code> points to can be noninitialized. • If initialized, the variables can have any value that their type allows. 	<pre data-bbox="793 302 1231 677"> #include <string.h> typedef struct { char a; int b; } S; void main() { S s; int val; memset(&s,1,sizeof(S)); val=s.b; // val can have any int value } </pre> <p data-bbox="793 703 1320 868">In this code, Polyspace considers that following the use of <code>memset</code>, each field of <code>s</code> has any value that its type allows. For instance, <code>s.b</code> can have any value in the range allowed for an <code>int</code> variable.</p>

#pragma Directives

The verification ignores most `#pragma` directives, because they do not carry information relevant to the verification.

However, the verification takes into account the behavior of these pragmas.

Pragma	Effect on Verification
<code>#pragma asm</code> and <code>#pragma endasm</code> , or <code>#asm</code> and <code>#endasm</code>	The verification ignores the content between the pragmas.
<code>#pragma hdrstop</code>	For Visual C++ compilers, the verification stops processing precompiled headers at the point where it encounters the pragma.
<code>#pragma once</code>	The verification allows the current source file to be included only once in a compilation.
<code>#pragma pack(n)</code> , <code>#pragma pack(push[,n])</code> , <code>#pragma pack(pop)</code>	The verification takes into account the boundary alignment specified in the pragmas. For more information, see the following example.
<code>#error message</code>	The verification stops if it encounters the directive. For more information, see “Error Related to <code>#error</code> Directive”.

For more information on the pragmas, see your compiler documentation. If the verification does not take into account a certain pragma from the preceding list, see if you specified the right compiler for your verification. For more information, see `Compiler (-compiler)`.

For instance, in this code, the directives `#pragma pack(n)` force a new alignment boundary in the structure. The User assertion on page 3-125 checks in the `main` function are green because the verification takes into account the behavior of the directives. The verification uses these options:

- Target processor type (`-target`): i386 (char: 1 byte, int: 4 bytes)

```
• Compiler (-compiler):gnu4.9

#include <assert.h>

#pragma pack(2)

struct _s6 {
    char c;
    int i;
} s6;

#pragma pack() /* Restores default packing: pack(4) */

struct _sb {
    char c;
    int i;
} sb;

#pragma pack(1)

struct _s5 {
    char c;
    int i;
} s5;

int main(void) {
    assert(sizeof(s6) == 6);
    assert(sizeof(sb) == 8);
    assert(sizeof(s5) == 5);
    return 0;
}
```

Standard Library Float Routines

For some two-argument standard library float routines, the verification can ignore the function arguments and assume that the function returns all possible values in its range.

In this code, the first `assert` statement is true and the second `assert` statement is false. However, because the verification assumes that `fmodf` and `nextafterf` return full-range values, it considers that the `assert` statements are false but only on a fraction of possible execution paths. Therefore, the User assertion checks on the `assert` statements are orange.

```
#include <math.h>
int main() {
    float val1=10.0, val2=3.0, res;
    res = fmodf(val1/val2);
    assert(res==1.0);

    res = nextafterf(val2, val1);
    assert(res<3.0);
}
```

Unions

In some situations, unions can help you construct efficient code. However, if you write a union member and read back a different union member, the behavior depends on the member sizes and can be implementation-dependent. You have to determine the following for your implementation:

- **Padding** – Padding can be inserted at the end of an union.
- **Alignment** – Members of structures within a union can have different alignments.
- **Endianness** – Whether the most significant byte of a word is stored at the lowest or highest memory address.
- **Bit-order** – Bits within bytes can have both different numbering and allocation to bit fields.

When you use unions in your code, because of these issues, Polyspace verification can lose precision.

If you write a union member, but read back another union member, Polyspace considers that the latter member can have any value that its type allows. In this code, the member `b` of `X` is written, but `a` is read. Polyspace considers that `a` can have any `int` value and both branches of the `if-else` statement are reachable.

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

void main() {
    my_union X;

    X.b[0] = 1;
    X.b[1] = 1;
    X.b[2] = 1;
    X.b[1] = 1;
    if (X.a == 0x1111) {
    }
    else {
    }
}
```

To avoid using unions in your code, check for violations of MISRA C:2012 Rule 19.2.

Note If you initialize a union using a static initializer, following ANSI C standard, Polyspace considers that the union member appearing first in the declaration list gets initialized.

Variable Cast as Void Pointer

The C language allows the use of statements that cast a variable as a void pointer. However, Polyspace verification of these statements entails a loss of precision.

Consider:

```
1  typedef struct {
2  int x1;
3  } s1;
4
5  s1 object;
6
7  void g(void *t) {
8  int x;
9  s1 *p;
10
11 p = (s1 *)t;
12 x = p->x1; // x should be assigned value 5 but p->x1 is full-range
13 }
14
15 void main(void) {
16 s1 * p;
17
18 object.x1 = 5;
19 p = &object;
20 g((void *)p); // p cast as void pointer
21 }
```

On line 12, the variable `x` must be assigned the value 5. However, the software assumes that `p->x1` has full range of values allowed by its type.

Assembly Code

Polyspace recognizes most inline assemblers as introduction of assembly code. The verification ignores the assembly code but accounts for the fact that the assembly code can modify variables in the C code.

If introduction of assembly code causes compilation errors:

- 1 Embed the assembly code between a `#pragma my_asm_begin` and a `#pragma my_asm_end` statement.
- 2 Specify the analysis option `-asm-begin my_asm_begin -asm-end my_asm_end`.

For more information, see `-asm-begin -asm-end`.

In this section...

“Recognized Inline Assemblers” on page 4-29

“Single Function Containing Assembly Code” on page 4-32

“Multiple Functions Containing Assembly Code” on page 4-32

“Local Variables in Functions with Assembly Code” on page 4-33

Recognized Inline Assemblers

Polyspace recognizes these inline assemblers as introduction of assembly code.

- `asm`

Examples:

- ```
int f(void)
{
 asm ("% reg val; mtmsr val;");
 asm("\tmovw.w #2700,sr");
 asm("\ttrap #7");
 asm(" stw r11,0(r3) ");
 assert (1); // is green
 return 1;
}
```
- ```
int other_ignored2(void)
{
```

```
asm "% reg val; mtmsr val;";
asm mtmsr val;
assert (1); // is green
asm ("px = pm(0,%2); \
     %0 = px1; \
     %1 = px2;"
     : "=d" (data_16), "=d" (data_32)
     : "y" ((UI_32 pm *)ram_address):
"px");
    assert (1); // is green
}
• int other_ignored4(void)
{
    asm {
        port_in: /* byte = port_in(port); */
        mov EAX, 0
        mov EDX, 4[ESP]
        in AL, DX
        ret
        port_out: /* port_out(byte,port); */
        mov EDX, 8[ESP]
        mov EAX, 4[ESP]
        out DX, AL
        ret }
    assert (1); // is green
}
• __asm__
```

Examples:

```
• int other_ignored6(void)
{
#define A_MACRO(bus_controller_mode) \
__asm__ volatile("nop"); \
__asm__ volatile("nop"); \
__asm__ volatile("nop"); \
__asm__ volatile("nop"); \
__asm__ volatile("nop"); \
__asm__ volatile("nop")
    assert (1); // is green
    A_MACRO(x);
    assert (1); // is green
    return 1;
}
```

- ```
int other_ignored1(void)
{
 __asm
 {MOV R8,R8
 MOV R8,R8
 MOV R8,R8
 MOV R8,R8
 MOV R8,R8}
 assert (1); // is green
}

int GNUC_include (void)
{
 extern int __P (char *__pattern, int __flags,
int (*__errfunc) (char *, int),
unsigned *__pglob) __asm__ ("glob64");
__asm__ ("rorw $8, %w0" \
: "=r" (__v) \
: "0" (((uint16) (val)));
__asm__ ("st g14,%0" : "=m" (*(AP)));
__asm__("" \
: "=r" (__t.c) \
: "0" (((union { int i, j; } *) (AP))++->i));
assert (1); // is green
return (int) 3 __asm__ ("% reg val");
}

int other_ignored3(void)
{
 __asm {ldab 0xffff,0;trapdis;};
__asm {ldab 0xffff,1;trapdis;};
assert (1); // is green
__asm__ ("% reg val");
__asm__ ("mtmsr val");
assert (1); // is green
return 2;
}

#pragma asm #pragma endasm
```

**Examples:**

- ```
int pragma_ignored(void)
{
    #pragma asm
    SRST
    #pragma endasm
```

```
        assert (1); // is green
    }
    • void test(void)
      {
        #asm
          mov _as:pe, reg
          jre _nop
        #endasm
        int r;
        r=0;
        r++;
      }
```

Single Function Containing Assembly Code

The software stubs a function that is preceded by `asm`, even if a body is defined.

```
asm int h(int tt)           // function h is stubbed even if body is defined
{
    % reg val;              // ignored
    mtmsr val;              // ignored
    return 3;                // ignored
};

void f(void) {
    int x;
    x = h(3);                // x is full-range
}
```

Multiple Functions Containing Assembly Code

The functions that you specify through the following pragma are stubbed automatically, even if function bodies are defined.

```
#pragma inline_asm(list of functions)
```

Code examples:

```
#pragma inline_asm(ex1, ex2)
    // The functions ex1 and ex2 are
    // stubbed, even if their bodies are defined

int ex1(void)
{
    % reg val;
```

```

    mtmsr val;
    return 3;                // ignored
};

int ex2(void)
{
    % reg val;
    mtmsr val;
    assert (1);            // ignored
    return 3;
};

#pragma inline_asm(ex3) // the definition of ex3 is ignored

int ex3(void)
{
    % reg val;
    mtmsr val;            // ignored
    return 3;
};

void f(void) {
    int x;

    x = ex1();            // ex1 is stubbed : x is full-range
    x = ex2();            // ex2 is stubbed : x is full-range
    x = ex3();            // ex3 is stubbed : x is full-range
}

```

Local Variables in Functions with Assembly Code

The verification ignores the content of assembly language instructions, but following the instructions, it makes some assumptions about:

- *Uninitialized local variables*: The assembly instructions can initialize these variables.
- *Initialized local variables*: The assembly instructions can write any possible value to the variables allowed by the variable data types.

For instance, the function `f` has assembly code introduced through the `asm` statement.

```

int f(void) {
    int val1, val2 = 0;
    asm("mov 4%0,%%eax"::"m"(val1));
}

```

```
    return (val1 + val2);  
}
```

On the `return` statement, the **Non-initialized local variable** check has the following results:

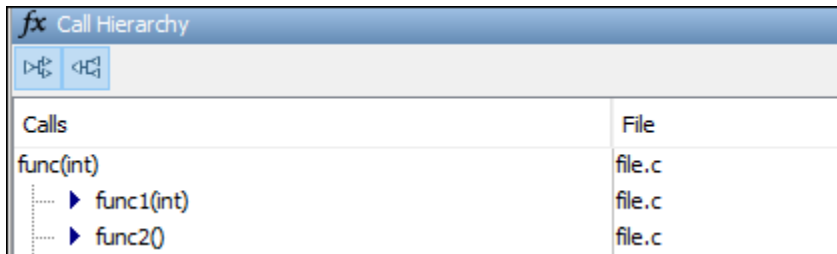
- `val1`: The check is orange because the assembly instruction can initialize `val1`.
- `val2`: The check is green. However, `val2` can have any `int` value.

If the variable is static, the assumptions are true anywhere in the function body, even before the assembly instructions.

Determination of Program Stack Usage

The Polyspace Code Prover analysis can estimate stack usage of each function in your program and compute the entire program stack usage. The analysis uses the function call hierarchy of your program to estimate stack usage. The stack usage of a function is the sum of local variable sizes in the function plus the maximum stack usage from function callees. The stack usage of the function at the top of the call hierarchy is the program stack usage.

For instance, for this call hierarchy, the stack usage of `func` is the size of local variables in `func` plus the maximum stack usage from `func1` and `func2` (unless they are called in mutually exclusive branches of a conditional statement).



The screenshot shows the 'Call Hierarchy' pane in Polyspace. It displays a tree structure of function calls. The root node is 'func(int)' from 'file.c'. It has two children: 'func1(int)' and 'func2()', both also from 'file.c'.

Calls	File
func(int)	file.c
▶ func1(int)	file.c
▶ func2()	file.c

For details, see:

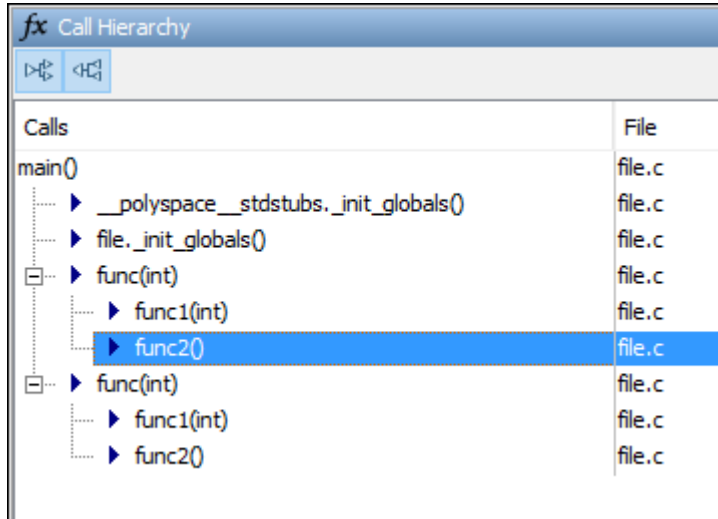
- **Function metrics:** Maximum Stack Usage and Minimum Stack Usage
- **Project metrics:** Program Maximum Stack Usage and Program Minimum Stack Usage

Investigate Possible Stack Overflow

If your stack usage exceeds available stack space, you can identify which function is responsible. Begin at the `main` function and navigate your program call tree. During navigation, look for the function that has an unreasonable size of local variables. If you cannot identify such a function, look for a call sequence that is unreasonably long. The detailed steps for navigation are:

- 1 On the **Source** pane, select the `main` function. On the **Call Hierarchy** pane, you see the functions called from `main` (callees). To see the full hierarchy, right-click a function and expand all nodes.

If the **Call Hierarchy** pane is not open by default, select **Window > Show/Hide View > Call Hierarchy**.



- 2 To navigate to the callee definition in your source, on the **Call Hierarchy** pane, double-click each callee name. Then, click the callee name on the **Source** pane. The **Result Details** pane shows the higher estimate of local variable size and stack usage by the callee.

Result Review

Status: Unreviewed

Severity: Unset

Select one or more results to review:

- ★* Maximum Stack Usage (Value: 8)
- ★* Number of Function Parameters (Value: 0)
- ★* Number of Goto Statements (Value: 0)
- ★* Higher Estimate of Size of Local Variables (Value: 8)
- ★* Number of Instructions (Value: 0)

★ Maximum Stack Usage (Value: 8) ?

This metric shows the total size of all local variables in a function plus the maximum stack usage from its callees (called functions).

Source

file.c x

```

15 void func1(int status) {
16     if(status == 0)
17         int temp;
18     else
19         double temp2;
20 }
21
22 void func2(void) {
23     double temp;
24 }

```

Stage Usage Not Computed

The analysis does not compute stack usage if your code has:

- Red checks. If a definite run-time error occurs in a function or one of its callees, the analysis does not compute its stack usage or the program stack usage. The reason is that code following a red check is not analyzed. If the unanalyzed code contains function calls, any stack usage estimate for the caller function is inaccurate.

In this example, the stack usage of `func` is not computed because following the red overflow, the remainder of the function is not analyzed. If the stack usage was computed, function calls in the unanalyzed code, such as the call to `func2`, would not be part of the computation.

```
#include <limits.h>
void func(void) {
    int val=INT_MAX;
    val++;
    func2();
}
```

- Recursive functions and C++ function templates (or methods in class templates).

Stack Usage Assumptions

If a function is called but not defined in the code that you provide to Polyspace, the stack usage determination does take the function call into account.

This assumption applies to:

- Implicit C++ constructors.

For instance, in this example, `func` calls the constructor of class `myClass` when `myObj` is defined. Stack usage determination does not consider the constructor as a callee of `func`.

```
class myClass {std::string str;};

void func() {
    myClass myObj;
}
```

- Standard library functions or other functions whose definitions are missing from the code in your Polyspace project.

For instance, in this example, `func` calls the standard library function `cos`. Unless you provide the definition of `cos`, stack usage determination does not consider it as a callee of `func`.

```
#include <math.h>

double func(double arg) {
```

```
    return cos(arg);  
}
```

Limitations of Polyspace Verification

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

This document is stored as `codeprover_limitations.pdf` in the following folder:

```
matlabroot\polyspace\verifier\code_prover
```

Here, *matlabroot* is the MATLAB installation folder, for instance, `C:\Program Files\MATLAB\R2017a`.

Functions, Classes, Methods, Properties, and Apps

pslinkfun

Manage model analysis at the command line

Syntax

```
pslinkfun('annotations','type',typeValue,'kind',kindValue,  
Name,Value)
```

```
pslinkfun('openresults',systemName)
```

```
pslinkfun('settemplate',psprjFile)  
prjTemplate = pslinkfun('gettemplate')
```

```
pslinkfun('advancedoptions')  
pslinkfun('enablebacktomodel')  
pslinkfun('help')  
pslinkfun('metrics')  
pslinkfun('jobmonitor')  
pslinkfun('stop')
```

Description

`pslinkfun('annotations','type',typeValue,'kind',kindValue,Name,Value)` adds an annotation of type `typeValue` and kind `kindValue` to the selected block in the model. You can specify a different block using a `Name,Value` pair argument. You can also add notes about a severity classification, an action status, or other comments using `Name,Value` pairs.

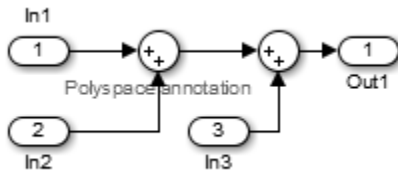
In the generated code associated with the annotated block, Polyspace adds code comments before and after the lines of code. Polyspace reads these comments and marks Polyspace results of the specified `kind` with the annotated information.

Syntax limitations:

- You can have only one annotation per block. If a block produces both a rule violation and an error, you can annotate only one type.

- Even though you apply annotations to individual blocks, the scope of the annotation can 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.

```

/* polyspace:begin<RTE:OVFL:Medium:To Fix>*/
annotate_y.Out1=(annotate_u.In1+annotate_U.In2)+annotate_U.In3;
/* polyspace:end<RTE:OVFL:Medium:To Fix> */
  
```

Therefore, the annotation justifies both summations.

`pslinkfun('openresults',systemName)` opens the Polyspace results associated with the model or subsystem `systemName` in the Polyspace environment.

`pslinkfun('settemplate',psprjFile)` sets the configuration file for new verifications.

`prjTemplate = pslinkfun('gettemplate')` returns the template configuration file used for new analyses.

`pslinkfun('advancedoptions')` opens the advanced verification options window to configure additional options for the current model.

`pslinkfun('enablebacktomodel')` enables the back-to-model feature of the Simulink plug-in. If your Polyspace results do not properly link to back to the model blocks, run this command.

`pslinkfun('help')` opens the Polyspace documentation in a separate window. Use this option for only pre-R2013b versions of MATLAB.

`pslinkfun('metrics')` opens the Polyspace Metrics interface.

`pslinkfun('jobmonitor')` opens the Polyspace Job Monitor to display remote verifications in the queue.

`pslinkfun('stop')` kills the code analysis that is currently running. Use this option for local analyses only.

Examples

Annotate a Block and Run a Polyspace Code Prover Verification

Use the Polyspace annotation function to annotate a block and see the annotation in the verification results.

In the example model `WhereAreTheErrors`, set the current block to the division block of the `10* x // (x-y)` subsystem. Then, add an annotation to the current block to mark division by zero (DIV) errors as justified with the annotation.

```
model = 'WhereAreTheErrors';
open(model)
gcb = 'WhereAreTheErrors/10* x // (x-y)/Divide';
pslinkfun('annotations','type','RTE','kind','ZDV','status',...
         'justified','comment','verified not an error')
```

In Simulink, the division block of the `10* x // (x-y)` subsystem now has a Polyspace annotation.

At the command line, generate code for the model and run a verification. After the analysis is finished, open the result in the Polyspace environment:

```
slbuild(model)
pslinkrun(model)
pslinkfun('openresults',model)
```

If you look at the orange division by zero error, the check is justified and includes the status and comments from your annotation.

Add Batch Options to Default Configuration Template

Change advanced Polyspace options and set the new configuration as a template.

Load the model `WhereAreTheErrors` and open the advanced options window.

```
model = 'WhereAreTheErrors';
load_system(model)
pslinkfun('advancedoptions')
```

In the **Run Settings** pane, select the options **Run Code Prover analysis on a remote cluster** and **Upload results to Polyspace Metrics**.

Set the configuration template for new Polyspace analyses to have these options.

```
pslinkfun('settemplate',fullfile(cd,'pslink_config',...
    'WhereAreTheErrors_config.psprj'))
```

View the current Polyspace template.

```
template = pslinkfun('gettemplate')

template =
C:\ModelLinkDemo\pslink_config\WhereAreTheErrors_config.psprj
```

View Polyspace Queue and Metrics

Run a remote analysis, view the analysis in the queue, and review the metrics.

Before performing this example, check that your Polyspace configuration is set up for remote analysis and Polyspace Metrics.

Build the model `WhereAreTheErrors`, create a Polyspace options object, set the verification mode, and open the advanced options window.

```
model = 'WhereAreTheErrors';
load_system(model)
slbuild(model)
opts = pslinkoptions(model);
opts.VerificationMode = 'CodeProver';
pslinkfun('advancedoptions')
```

In the **Run Settings** pane, select the options **Run Code Prover analysis on a remote cluster** and **Upload results to Polyspace Metrics**.

Run Polyspace, then open the Job Monitor to monitor your remote job.

```
pslinkrun(model,opts)
pslinkfun('jobmonitor')
```

After your job is finished, open the metrics server to see your job in the repository.

```
pslinkfun('metrics')
```

Input Arguments

typeValue — type of result

'RTE' | 'MISRA-C' | 'MISRA-AC-AGC' | 'MISRA-CPP' | 'JSF'

The type of result with which to annotate the block, specified as:

- 'RTE' for run-time errors.
- 'MISRA-C' for MISRA C coding rule violations (C code only).
- 'MISRA-AC-AGC' 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
'RTE'	Use the abbreviation associated with the type of check that you want to annotate. For example, 'UNR' – Unreachable Code. For the list of possible checks, see: “Run-Time Checks”.
'MISRA-C'	Use the rule number that you want to annotate. For example, '2.2'. For the list of supported MISRA C rules and their numbers, see “Supported MISRA C:2004 and MISRA AC AGC Rules”.

type Value	kind Values
'MISRA-AC-AGC'	Use the rule number that you want to annotate. For example, '2.2'. For the list of supported MISRA AC AGC rules and their numbers, see “Supported MISRA C:2004 and MISRA AC AGC Rules”.
'MISRA-CPP'	Use the rule number that you want to annotate. For example, '0-1-1'. For the list of supported MISRA C++ rules and their numbers, see “Supported MISRA C++ Coding Rules”.
'JSF'	Use the rule number that you want to annotate. For example, '3'. For the list of supported JSF C++ rules and their numbers, see “Supported JSF C++ Coding Rules”.

Example: `pslinkfun('annotations','type','MISRA-CPP','kind','1-2-3')`

Data Types: char

systemName — Simulink model

system | subsystem

Simulink model specified by the system or subsystem name.

Example: `pslinkfun('openresults','WhereAreTheErrors')`

psprjFile — Polyspace project file

standard Polyspace template (default) | absolute path to .psprj file

Polyspace project file specified as the absolute path to the .psprj project file. If psprjFile is empty, Polyspace uses the standard Polyspace template file. New Polyspace projects start with this project configuration.

Example: `pslinkfun('settemplate', fullfile(matlabroot, 'polyspace', 'examples', 'cxx', 'Bug_Finder_Example', 'Bug_Finder_Example.bf.psprj'));`

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', 'to fix'`

block — block to be annotated

`gcb` (default) | block name

The block you want to annotate 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 — severity of the check

`'high'` | `'medium'` | `'low'` | `'not a defect'` | `'unset'`

Severity of the check specified as `high`, `medium`, `low`, `not a defect`, or `unset`.

Example: `'class', 'high'`

status — action status

`'undecided'` | `'to investigate'` | `'to fix'` | `'justified'` | `'no action planned'` | `'other'`

Action status of the check specified as `to investigate`, `to fix`, `justified`, `no action planned`, or `other`.

The statuses, `justified` and `no action planned`, also mark the result as justified.

Example: `'status', 'no action planned'`

comment — additional comments

character vector

Additional comments specified as a character vector. The comments provide more information about why the results are justified.

Example: `'comment', 'defensive code'`

See Also

`pslinkrun` | `pslinkoptions` | `gcb`

Introduced in R2014a

pslinkoptions

Create options object to customize Polyspace runs from MATLAB command line

Syntax

```
opts = pslinkoptions(codegen)
opts = pslinkoptions(model)
opts = pslinkoptions(sfunc)
```

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.

`opts = pslinkoptions(sfunc)` returns an options object with the configuration options for the S-Function.

Examples

Use a Simulink model to create and edit an options objects

Load `psdemo_model_link_sl` and create a Polyspace® options object from the model:

```
load_system('psdemo_model_link_sl');
model_opt = pslinkoptions('psdemo_model_link_sl')

model_opt =

    ResultDir: 'results_$(ModelName$'
VerificationSettings: 'PrjConfig'
    OpenProjectManager: 1
```



```

    AddSuffixToResultDir: 0
  EnableAdditionalFileList: 0
    AdditionalFileList: {}
    VerificationMode: 'CodeProver'
  EnablePrjConfigFile: 0
    PrjConfigFile: ''
  AddToSimulinkProject: 0
    InputRangeMode: 'DesignMinMax'
    ParamRangeMode: 'None'
    OutputRangeMode: 'None'
    ModelRefVerifDepth: 'All'
  ModelRefByModelRefVerif: 0
    AutoStubLUT: 0
  CxxVerificationSettings: 'PrjConfig'
  CheckConfigBeforeAnalysis: 'OnWarn'

```

The model is already configured for Embedded Coder®, so only the Embedded Coder configuration options appear.

Change the results folder name option and set `OpenProjectManager` to true.

```

model_opt.ResultDir = 'results_v1_$ModelName$';
model_opt.OpenProjectManager = true

model_opt =

    ResultDir: 'results_v1_$ModelName$'
  VerificationSettings: 'PrjConfig'
    OpenProjectManager: 1
  AddSuffixToResultDir: 0
  EnableAdditionalFileList: 0
    AdditionalFileList: {}
    VerificationMode: 'CodeProver'
  EnablePrjConfigFile: 0
    PrjConfigFile: ''
  AddToSimulinkProject: 0
    InputRangeMode: 'DesignMinMax'
    ParamRangeMode: 'None'
    OutputRangeMode: 'None'
    ModelRefVerifDepth: 'All'
  ModelRefByModelRefVerif: 0
    AutoStubLUT: 0

```

```
CxxVerificationSettings: 'PrjConfig'  
CheckConfigBeforeAnalysis: 'OnWarn'
```

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: {}  
  VerificationMode: 'CodeProver'  
  EnablePrjConfigFile: 0  
    PrjConfigFile: ''  
AddToSimulinkProject: 0  
  InputRangeMode: 'DesignMinMax'  
  ParamRangeMode: 'None'  
  OutputRangeMode: 'None'  
  ModelRefVerifDepth: 'Current model only'  
ModelRefByModelRefVerif: 0  
  AutoStubLUT: 1  
  CxxVerificationSettings: 'PrjConfig'  
CheckConfigBeforeAnalysis: 'OnWarn'
```

To Follow the progress in the Polyspace interface, set the `OpenProjectManager` option to true. Change the configuration to check for both checks and MISRA C® 2012 coding rule violations:

```
new_opt.OpenProjectManager = true;  
new_opt.VerificationSettings = 'PrjConfigAndMisraC2012'  
  
new_opt =  
  
           ResultDir: 'results_ $ModelName$'
```

```

VerificationSettings: 'PrjConfigAndMisraC2012'
  OpenProjectManager: 1
  AddSuffixToResultDir: 0
EnableAdditionalFileList: 0
  AdditionalFileList: {}
  VerificationMode: 'CodeProver'
  EnablePrjConfigFile: 0
  PrjConfigFile: ''
AddToSimulinkProject: 0
  InputRangeMode: 'DesignMinMax'
  ParamRangeMode: 'None'
  OutputRangeMode: 'None'
  ModelRefVerifDepth: 'Current model only'
ModelRefByModelRefVerif: 0
  AutoStubLUT: 1
  CxxVerificationSettings: 'PrjConfig'
CheckConfigBeforeAnalysis: 'OnWarn'

```

Create and edit an options object for TargetLink at the command line

Create a Polyspace® options object called `new_opt` with TargetLink® parameters:

```
new_opt = pslinkoptions('tl')
```

```
new_opt =
```

```

          ResultDir: 'results_$(ModelName$)'
VerificationSettings: 'PrjConfig'
  OpenProjectManager: 0
  AddSuffixToResultDir: 0
EnableAdditionalFileList: 0
  AdditionalFileList: {}
  VerificationMode: 'CodeProver'
  EnablePrjConfigFile: 0
  PrjConfigFile: ''
AddToSimulinkProject: 0
  InputRangeMode: 'DesignMinMax'
  ParamRangeMode: 'None'
  OutputRangeMode: 'None'
  AutoStubLUT: 1

```

Set the `OpenProjectManager` option to `true` to follow the progress in the Polyspace interface. Also change the configuration to check for both run-time errors and MISRA C® coding rule violations:

```
new_opt.OpenProjectManager = true;
new_opt.VerificationSettings = 'PrjConfigAndMisra'

new_opt =

    ResultDir: 'results_$ModelName$'
    VerificationSettings: 'PrjConfigAndMisra'
    OpenProjectManager: 1
    AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
    AdditionalFileList: {}
    VerificationMode: 'CodeProver'
    EnablePrjConfigFile: 0
    PrjConfigFile: ''
    AddToSimulinkProject: 0
    InputRangeMode: 'DesignMinMax'
    ParamRangeMode: 'None'
    OutputRangeMode: 'None'
    AutoStubLUT: 1
```

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 properties specific to that code generator.

For a description of all configuration options and their values, see `pslinkoptions`.

Example: `ec_opt = pslinkoptions('ec')`

Example: `tl_opt = pslinkoptions('tl')`

Data Types: `char`

model — Simulink model name

model name

Simulink model, specified by the model name. Creates a Polyspace options object with the configuration options of that model. If you have not set any options, the object has the default configuration options. If you have set a code generator, the object has the default options for that code generator.

For a description of all configuration options and their values, see `pslinkoptions`.

```
Example: model_opt = pslinkoptions('my_model')
```

Data Types: char

sfunc — path to S-Function

character vector

Path to S-Function, specified as a character vector. Creates a Polyspace options object with the configuration options for the S-function. If you have not set any options, the object has the default configuration options.

For a description of all configuration options and their values, see `pslinkoptions`.

```
Example: sfunc_opt = pslinkoptions('path/to/sfunction')
```

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 Polyspace from the MATLAB command line.

For the list of object properties, see `pslinkoptions`.

```
Example: opts= pslinkoptions('ec')  
opts.VerificationSettings = 'Misra'
```

See Also

`pslinkfun` | `pslinkrun`

Topics

`pslinkoptions`

Introduced in R2012a

pslinkrun

Run Polyspace analysis on model, system, or S-Function

Syntax

```
[polyspaceFolder, resultsFolder] = pslinkrun  
[polyspaceFolder, resultsFolder]= pslinkrun(target)  
[polyspaceFolder, resultsFolder] = pslinkrun(target,opts)  
[polyspaceFolder, resultsFolder] = pslinkrun(target,opts,asModelRef)
```

Description

`[polyspaceFolder, resultsFolder] = pslinkrun` analyzes code generated from the current system using the configuration options associated with the current system. It returns the location of the results folder. The current system is the system returned by the command `bdroot`.

`[polyspaceFolder, resultsFolder]= pslinkrun(target)` analyzes `target` with the configuration options associated with the model containing `target`. Before you run an analysis, you must:

- Generate code for models and subsystems.
- Compile S-Functions.

`[polyspaceFolder, resultsFolder] = pslinkrun(target,opts)` analyzes `target` with the configuration options from the options object `opts`. It returns the location of the results folder.

`[polyspaceFolder, resultsFolder] = pslinkrun(target,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.

Examples

Analyze Generated Code

Use a Simulink model to generate code, set configuration options, and then run an analysis from the command line.

```
% Generate code from the model WhereAreTheErrors.
model = 'WhereAreTheErrors';
load_system(model);
slbuild(model);

% Create a Polyspace options object from the model.
opts = pslinkoptions(model);

% Set properties that define the Polyspace analysis.
opts.VerificationMode = 'CodeProver';
opts.VerificationSettings = 'PrjConfigAndMisraC2012';

% Run Polyspace using the options object.
[polyspaceFolder, resultsFolder] = pslinkrun(model,opts);
bdclose(model);
```

The results and the corresponding Polyspace project are saved to the `results_WhereAreTheErrors` folder, listed in the `polyspaceFolder` variable. The full path to the results folder is in the `resultsFolder` variable.

Analyze Referenced Model Code

Use a Simulink model to generate model reference code, set configuration options, and then run an analysis from the command line.

```
% Generate code from the model WhereAreTheErrors.
% Treat WhereAreTheErrors as if referenced by another model.
model = 'WhereAreTheErrors';
load_system(model);
slbuild(model, 'ModelReferenceRTWTargetOnly');

% Create a Polyspace options object from the model.
opts = pslinkoptions(model);
```



```

% Set properties that define the Polyspace analysis.
opts.VerificationMode = 'CodeProver';
opts.VerificationSettings = 'PrjConfigAndMisraC2012';

% Run Polyspace with the options object.
[polyspaceFolder, resultsFolder] = pslinkrun(model,opts,true);
bdclose(model);

```

The results and corresponding Polyspace project are saved to the `results_mr_WhereAreTheErrors` folder, listed in the `polyspaceFolder` variable. The full path to the results folder is in the `resultsFolder` variable.

Reuse Analysis Options for Multiple Models

This example shows how to reuse a subset of options for Polyspace analysis of multiple models. Create a generic options object and specify properties that describe the common options. Associate the generic options object with a model-specific options object. Optionally, set some model-specific options and run the Polyspace analysis.

```

% Generate code from the model WhereAreTheErrors.
model = 'psdemo_model_link_sl';
load_system(model);
slbuild(model);

% Create a generic options object to use for multiple model analyses.
opts = polyspace.ModelLinkOptions();
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
opts.CodingRulesCodeMetrics.MisraC3Subset = 'all';
opts.MergedReporting.ReportOutputFormat = 'PDF';
opts.MergedReporting.EnableReportGeneration = true;

% Create a model-specific options object.
mlopts = pslinkoptions(model);

% Create a project from the generic options object.
% Associate the project with the model-specific options object.
prjfile = opts.generateProject('model_link_opts');
mlopts.EnablePrjConfigFile = true;
mlopts.PrjConfigFile = prjfile;
mlopts.VerificationMode = 'BugFinder';

```

```
% Run Polyspace with the model-specific options object.
[polyspaceFolder, resultsFolder] = pslinkrun(model,mlopts);
bdclose(model);
```

After the analysis completes, results open automatically in the Polyspace interface.

Input Arguments

target — Target of the analysis

bdroot (default) | model or system name | path to S-Function block

Target of the analysis specified as a character vector, with the model, system, or S-function in single quotes. The default value is the system returned by bdroot.

Example: [polyspaceFolder, resultsFolder] = pslinkrun('demo') where demo is the name of a model.

Example: [polyspaceFolder, resultsFolder] = pslinkrun('path/to/sfunction')

Data Types: char

opts — Configuration options

options associated with target (default) | options object

Configuration options for the analysis, specified as a Polyspace options object. The function pslinkoptions creates an options object. You can customize the options object by changing the pslinkoption properties.

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 that is 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 that is generated as model referenced code. This option is equivalent to choosing **Verify Code Generated For >**

Referenced Model in the Simulink Polyspace options. Specifying model reference code indicates that Polyspace must look for the generated code in a different location from the location for standalone code.

Data Types: `logical`

Output Arguments

polyspaceFolder — Folder containing Polyspace project and results

character vector

Name of the folder containing Polyspace project and results, specified as a character vector. The default value of this variable is `results_$(modelName)`.

To change this value, see “Output folder” on page 11-18.

resultsFolder — Full path to subfolder containing Polyspace results

character vector

Full path to subfolder containing Polyspace results, specified as a character vector.

The folder `results_$(modelName)` contains your Polyspace project and a subfolder `$(modelName)` with the analysis results. This variable gives you the full path to the subfolder. You can use this path with the `polyspace.CodeProverResults` class.

To change the parent folder `results_$(modelName)`, see “Output folder” on page 11-18.

See Also

`pslinkfun` | `pslinkoptions` | `pslinkoptions`

Topics

“Verify S-Function Code”

“Recommended Model Settings for Code Analysis”

Introduced in R2012a

polyspaceCodeProver

Run Polyspace Code Prover verification from MATLAB

Note For easier scripting, run Polyspace® analysis using a `polyspace.Project` object.

Syntax

```
polyspaceCodeProver
polyspaceCodeProver(projectFile)

polyspaceCodeProver(optsObject)
polyspaceCodeProver(projectFile, '-nodesktop')

polyspaceCodeProver(resultsFile)
polyspaceCodeProver('-results-dir', resultsFolder)

polyspaceCodeProver('-help')

polyspaceCodeProver('-sources', sourceFiles)
polyspaceCodeProver('-sources', sourceFiles, Name, Value)
```

Description

`polyspaceCodeProver` opens Polyspace Code Prover.

`polyspaceCodeProver(projectFile)` opens a Polyspace project file in Polyspace Code Prover.

`polyspaceCodeProver(optsObject)` runs a verification on the Polyspace options object in MATLAB.

`polyspaceCodeProver(projectFile, '-nodesktop')` runs a verification on the Polyspace project file in MATLAB. If you have multiple modules or configurations, Polyspace runs the active configuration and active module. To see which module and configuration are active, open the project in the Polyspace interface and look for the bold,

selected module and configuration. To change which module or configuration is active, before closing the Polyspace interface, select the module and configuration you want to verify.

Alternatively, you can use the function `polyspaceCodeProverNoDesktop` with the syntax `polyspaceCodeProverNoDesktop(projectfile)`.

`polyspaceCodeProver(resultsFile)` opens a Polyspace results file in Polyspace Code Prover.

`polyspaceCodeProver('-results-dir', resultsFolder)` opens a Polyspace results file from `resultsFolder` in Polyspace Code Prover.

`polyspaceCodeProver('-help')` displays all options that can be supplied to the `polyspaceCodeProver` command to run a Polyspace Code Prover verification.

`polyspaceCodeProver('-sources', sourceFiles)` runs a Polyspace Code Prover verification on the source files specified in `sourceFiles`.

`polyspaceCodeProver('-sources', sourceFiles, Name, Value)` runs a Polyspace Code Prover verification on the source files with additional options specified by one or more `Name, Value` pair arguments.

Examples

Open Polyspace Projects from MATLAB

This example shows how to open a Polyspace project file with extension `.psprj` from MATLAB. In this example, open the project file `Code_Prover_Example.psprj`.

Assign the full project file path to a MATLAB variable `prjFile`.

```
prjFile = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', ...  
                  'Code_Prover_Example', 'Code_Prover_Example.psprj');
```

Open the project.

```
polyspaceCodeProver(prjFile)
```

Open Polyspace Results from MATLAB

This example shows how to open a Polyspace results file from MATLAB. In this example, you open the results file from the folder `matlabroot\polyspace\examples\cxx\Code_Prover_Example\Module_1\CP_Result`.

Assign the full folder path to a MATLAB variable `resFolder`.

```
resFolder = fullfile(matlabroot, 'polyspace', 'examples', ...  
    'cxx', 'Code_Prover_Example', 'Module_1', 'CP_Result');
```

Open the results.

```
polyspaceCodeProver('-results-dir', resFolder)
```

Run Polyspace Verification with Options Object

This example shows how to run a Polyspace verification in MATLAB using objects. For this example:

- Save a C source file, `source.c`, in the folder `C:\Polyspace_Sources`.
- Save an include file in the folder `C:\Polyspace_Includes`.

Create an options object and add the source file and include folder to the properties.

```
opts = polyspace.CodeProverOptions;  
opts.Sources = {'C:\Polyspace_Sources\source.c'};  
opts.EnvironmentSettings.IncludeFolders = {'C:\Polyspace_Includes'};  
opts.ResultsDir = 'C:\Polyspace_Results';
```

Run the verification and view the results.

```
polyspaceCodeProver(opts);  
polyspaceCodeProver('-results-dir', opts.ResultsDir)
```

Polyspace runs on the file `C:\Polyspace_Sources\source.c` and stores the result in `C:\Polyspace_Results`.

Run Polyspace Verification from MATLAB with DOS/UNIX Options

This example shows how to run a Polyspace verification on a single source file. For this example:

- Save a C source file, `source.c`, in the folder `C:\Polyspace_Sources`.
- Save an include file in the folder `C:\Polyspace_Includes`.

Run the analysis and open the results.

```
polyspaceCodeProver('-sources','C:\Polyspace_Sources\source.c', ...
                  '-I','C:\Polyspace_Includes', ...
                  '-results-dir','C:\Polyspace_Results')
polyspaceCodeProver('-results-dir','C:\')
```

Run Polyspace Verification with Coding Rules Checking

This example shows two different ways to customize a verification in MATLAB. You can customize as many additional options as you want by changing properties in an options object or by using Name-Value pairs. You specify checking of MISRA C 2012 coding rules, exclude headers from coding rule checking, and generate a main.

To create variables for source file path, include folder path, and results folder path that you can use for either analysis method.

```
sourceFileName = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', ...
                          'Code_Prover_Example', 'sources', 'example.c');
includeFileName = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', ...
                           'Code_Prover_Example', 'sources', 'include.h');
resFolder1 = fullfile('Polyspace_Results_1');
resFolder2 = fullfile('Polyspace_Results_2');
```

Verify coding rules with an options object.

```
opts = polyspace.CodeProverOptions('C');
opts.Sources = {sourceFileName};
opts.EnvironmentSettings.IncludeFolders = {includeFileName};
opts.ResultsDir = resFolder1;
opts.CodingRulesCodeMetrics.MisraC3Subset = 'mandatory';
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
opts.CodeProverVerification.EnableMain = true;
opts.InputsStubbing.DoNotGenerateResultsFor = 'all-headers';
```

```
polyspaceCodeProver(opts);  
polyspaceCodeProver('-results-dir', resFolder1);
```

Verify coding rules with DOS/UNIX options.

```
polyspaceCodeProver('-sources', sourceFileName, ...  
    '-I', includeFileName, ...  
    '-results-dir', resFolder2, ...  
    '-misra3', 'mandatory', ...  
    '-do-not-generate-results-for', 'all-headers', ...  
    '-main-generator');  
polyspaceCodeProver('-results-dir', resFolder2);
```

Input Arguments

optsObject — Polyspace options object name

object handle

Polyspace options object name, specified as the object handle.

To create an options object, use one of the Polyspace options classes:

`polyspace.Options` or `polyspace.ModelLinkOptions`.

Example: `opts`

projectFile — Name of `.psprj` file

character vector

Name of project file with extension `.psprj`, specified as a character vector.

If the file is not in the current folder, `projectFile` must include a full or relative path.

To identify the current folder, use `pwd`. To change the current folder, use `cd`.

Example: `'C:\Polyspace_Projects\myProject.psprj'`

resultsFile — Name of `.pscp` file

character vector

Name of results file with extension `.pscp`, specified as a character vector.

If the file is not in the current folder, `resultsFile` must include a full or relative path.

Example: `'myResults.pscp'`

resultsFolder — Name of result folder

character vector

Name of result folder, specified as a character vector. The folder must contain the results file with extension `.pscp`. If the results file resides in a subfolder of the specified folder, this command does not open the results file.

If the folder is not in the current folder, `resultsFolder` must include a full or relative path.

Example: `'C:\Polyspace\Results\'`

sourceFiles — Comma-separated names of `.c` or `.cpp` files

character vector

Comma-separated source file names with extension `.c` or `.cpp`, specified as a single character vector.

If the files are not in the current folder, `sourceFiles` must include a full or relative path.

Example: `'myFile.c', 'C:\mySources\myFile1.c,C:\mySources\myFile2.c'`

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: `'-OS-target', 'Linux', '-compiler', 'gnu4.6'` specifies that the source code is intended for the Linux operating system and contains non-ANSI C syntax for the GCC 4.6 compiler.

For the full list of analysis options, see “Analysis Options”.

See Also

`polyspace.CodeProverOptions` | `polyspace.ModelLinkCodeProverOptions`

Introduced in R2013b

polyspaceConfigure

Create Polyspace project from your build system at the MATLAB command line

Syntax

```
polyspaceConfigure buildCommand
```

```
polyspaceConfigure -option value buildCommand
```

Description

`polyspaceConfigure buildCommand` traces your build system and creates a Polyspace project with information gathered from your build system.

`polyspaceConfigure -option value buildCommand` traces your build system and uses `-option value` to modify the default operation of `polyspaceConfigure`. Specify the modifiers before `buildCommand`, otherwise they are considered as options in the build command itself.

Examples

Create Polyspace Project from Makefile

This example shows how to create a Polyspace project if you use the command `make targetName buildOptions` to build your source code.

Create a Polyspace project specifying a unique project name. Use the `-B` or `-W` `makefileName` option with `make` so that the all prerequisite targets in the makefile are remade.

```
polyspaceConfigure -prog myProject ...  
                    make -B targetName buildOptions
```

Open the Polyspace project in the **Project Browser**.

```
polyspaceCodeProver('myProject.psprj')
```

Run Command-Line Polyspace Analysis from Makefile

This example shows how to run Polyspace analysis if you use the command `make targetName buildOptions` to build your source code. In this example, you use `polyspaceConfigure` to trace your build system but do not create a Polyspace project. Instead you create an options file that you can use to run Polyspace analysis from command-line.

Create a Polyspace options file specifying the `-output-options-file` command. Use the `-B` or `-W makefileName` option with `make` so that all prerequisite targets in the makefile are remade.

```
polyspaceConfigure -no-project -output-options-file ...  
                  myOptions make -B targetName buildOptions
```

Use the options file that you created to run a Polyspace analysis at the command line:

```
polyspaceCodeProver -options-file myOptions
```

- “Create Project Automatically”

Input Arguments

buildCommand — Command for building source code

build command

Build command specified exactly as you use to build your source code.

Example: `make -B, make -W makefileName`

-option value — Options for changing default operation of `polyspaceConfigure`
single option starting with `-`, followed by argument | multiple space-separated option-argument pairs

Basic Options

Option	Argument	Description
-allow-build-error	None	<p>Option to create a Polyspace project even if an error occurs in the build process.</p> <p>If an error occurs, the build trace log shows the following message:</p> <pre>polyspace-configure ERROR: build command command_name fail [status=status_value]</pre> <p><i>command_name</i> is the build command name that you use and <i>status_value</i> is the non-zero exit status or error level that indicates which error occurred in your build process.</p>
-allow-overwrite	None	<p>Option to overwrite a project with the same name, if it exists.</p> <p>By default, polyspaceConfigure throws an error if a project with the same name already exists in the output folder. Use this option to overwrite the project.</p>
-author	Author name	<p>Name of project author.</p> <p>Example: -author jsmith</p>
-debug	None	Option used by MathWorks technical support
-help	None	Option to display the full list of polyspaceConfigure commands

Option	Argument	Description
-lang	auto (default) c cpp cpp11	<p>Option to specify source code language. The languages are:</p> <ul style="list-style-type: none"> • C99: Use argument <code>c</code>. • C++03: Use argument <code>cpp</code>. • C++11: Use argument <code>cpp11</code>. <p>By default, <code>polyspaceConfigure</code> detects the language. If it detects the same language for all files, it assigns the corresponding argument to the <code>-lang</code> option. If it detects a mixture of languages in the compilation units, it uses the argument <code>cpp11</code>.</p> <p>The argument to this option maps to the Source code language option in your Polyspace project. See <code>Source code language (-lang)</code>.</p>
-output-options-file	None	<p>Option to create a Polyspace analysis options file. Use this file for command-line analysis using <code>polyspaceCodeProver</code>.</p>
-output-project	Path	<p>Project file name and location for saving project. The default is the file <code>polyspace.psprj</code> in the current folder.</p> <p>Example: <code>-output-project ../myProjects/project1</code> creates a project <code>project1.psprj</code> in the folder with the relative path <code>../myProjects/</code>.</p>

Option	Argument	Description
-prog	Project name	<p>Project name that appears in the Polyspace user interface. The default is <code>polyspace</code>.</p> <p>If you do not use the option <code>-output-project</code>, the <code>-prog</code> argument also sets the project name.</p> <p>Example: <code>-prog myProject</code> creates a project that has the name <code>myProject</code> in the user interface. If you do not use the option <code>-output-project</code>, the project name is also <code>myProject.psrprj</code>.</p>
-silent (default) -verbose	None	Option to suppress or display additional messages from running <code>polyspaceConfigure</code> .

Advanced Options

Option	Argument	Description
-compiler-config	Path and file name	<p>Location and name of compiler configuration file.</p> <p>The file must be in a specific format. For guidance, see the existing configuration files in <code>matlabroot\polyspace\configure\compiler_configuration\</code>. For information on the contents of the file, see “Compiler Not Supported for Project Creation from Build Systems”.</p> <p>Example: <code>-compiler-configuration myCompiler.xml</code></p>

Option	Argument	Description
-no-build	None	Option to create a Polyspace project using previously saved build trace information. To use this option, you must have the build trace information saved from an earlier run of <code>polyspaceConfigure</code> with the <code>-no-project</code> option. If you use this option, you do not need to specify the <code>buildCommand</code> argument.
-no-project	None	Option to trace your build system without creating a Polyspace project and save the build trace information. Use this option to save your build trace information for a later run of <code>polyspaceConfigure</code> with the <code>-no-build</code> option.
-tmp-path	Path	Location of folder where temporary files are stored.

Cache Control Options

Option	Argument	Description
-build-trace	Path and file name	Location and name of file where build information is stored. The default is <code>./polyspace_configure_build_trace.log</code> . Example: <code>-build-trace ../build_info/trace.log</code>
-no-cache -cache-sources (default) -cache-all-files	None	Option to perform one of the following: <ul style="list-style-type: none"> • Not create a cache • Cache only source and header files. • Cache all files including binaries.

Option	Argument	Description
<code>-keep-cache</code> <code>-no-keep-cache</code> (default)	None	Option to preserve or clean up cache information after <code>polyspaceConfigure</code> completes execution. If <code>polyspaceConfigure</code> fails, you can provide this cache information to technical support for debugging purposes.
<code>-cache-path</code>	Path	Location of folder where cache information is stored. Example: <code>-cache-path ../cache</code>

See Also

Topics

“Create Project Automatically”

“Requirements for Project Creation from Build Systems”

“Compiler Not Supported for Project Creation from Build Systems”

Introduced in R2013b

polyspaceJobsManager

Manage Polyspace jobs on a MATLAB Distributed Computing Server cluster

Syntax

```
polyspaceJobsManager('listjobs')
polyspaceJobsManager('cancel','-job',jobNumber)
polyspaceJobsManager('remove','-job',jobNumber)
polyspaceJobsManager('getlog','-job',jobNumber)
polyspaceJobsManager('wait','-job',jobNumber)
polyspaceJobsManager('promote','-job',jobNumber)
polyspaceJobsManager('demote','-job',jobNumber)

polyspaceJobsManager('download','-job',jobNumber)
polyspaceJobsManager('download','-job',jobNumber,'-results-folder',
resultsFolder)

polyspaceJobsManager( ____, '-scheduler', scheduler)
```

Description

`polyspaceJobsManager('listjobs')` lists all Polyspace jobs in your cluster.

`polyspaceJobsManager('cancel','-job',jobNumber)` cancels the specified job. The job appears in your queue as cancelled.

`polyspaceJobsManager('remove','-job',jobNumber)` removes the specified job from your cluster.

`polyspaceJobsManager('getlog','-job',jobNumber)` displays the log for the specified job.

`polyspaceJobsManager('wait','-job',jobNumber)` pauses until the specified job is done.

`polyspaceJobsManager('promote', '-job', jobNumber)` moves the specified job up in the MATLAB job scheduler queue.

`polyspaceJobsManager('demote', '-job', jobNumber)` moves the specified job down in the MATLAB job scheduler queue.

`polyspaceJobsManager('download', '-job', jobNumber)` downloads the results from the specified job. The results are downloaded to the folder you specified when starting analysis, using the `-results-dir` on page 2-39 option.

`polyspaceJobsManager('download', '-job', jobNumber, '-results-folder', resultsFolder)` downloads the results from the specified job to `resultsFolder`.

`polyspaceJobsManager(____, '-scheduler', scheduler)` performs the specified action on the job scheduler specified. If you do not specify a server with any of the previous syntaxes, Polyspace uses the server stored in your Polyspace preferences.

Examples

Manipulate Two Jobs in the Cluster

In this example, use a MJS scheduler to run Polyspace remotely and monitor your jobs through the queue.

Before performing this example, set up an MJS and Polyspace Metrics. This example uses the `myMJS@myCompany.com` scheduler. When you perform this example, replace this scheduler with your own cluster name.

Set up your source files.

```
mkdir 'C:\psdemo\src'
demo = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', ...
'Code_Prover_Example', 'sources');
copyfile(demo, 'C:\psdemo\src\')
```

Submit two jobs to your scheduler.

```
polyspaceCodeProver -batch -scheduler myMJS@myCompany.com
-sources C:\psdemo\src\*.c'
-results-dir 'C:\psdemo\res1'
```

```
polyspaceCodeProver -batch -scheduler myMJS@myCompany.com
  -sources 'C:\psdemo\src\main.c'
  -results-dir 'C:\psdemo\res2'
  -add-to-results-repository
polyspaceJobsManager('listjobs', '-scheduler', 'myMJS@myCompany.com')
```

If your jobs have not started running, promote the second job to run before the first job.

```
polyspaceJobsManager('promote', '-job', '20', '-scheduler', ...
  'myMJS@myCompany.com')
```

Job 20 starts running before job 19.

Cancel job 19.

```
polyspaceJobsManager('cancel', '-job', '19', '-scheduler', ...
  'myMJS@myCompany.com')
polyspaceJobsManager('listjobs', '-scheduler', 'myMJS@myCompany.com')
```

Remove job 19.

```
polyspaceJobsManager('remove', '-job', '19', '-scheduler', ...
  'myMJS@myCompany.com')
polyspaceJobsManager('listjobs', '-scheduler', 'myMJS@myCompany.com')
```

Get the log for job 20.

```
polyspaceJobsManager('getlog', '-job', '20', '-scheduler', ...
  'myMJS@myCompany.com')
```

Download the information from job 20.

```
polyspaceJobsManager('download', '-job', '20', '-results-folder', ...
  'C:\psdemo\res3', '-scheduler', 'myCluster')
```

Input Arguments

jobNumber — Queued job number

character vector of job number

Number of the queued job that you want to manage, specified as a character vector in single quotes.

Example: '-job', '10'

resultsFolder — Path to results folder

character vector

Path to results folder specified as a character vector in single quotes. This folder stores the downloaded results files.

Example: `'-results-folder', 'C:\psdemo\myresults'`

scheduler — job scheduler

head node of your cluster | job scheduler name | cluster profile

Job scheduler for remote verifications specified as one of the following:

- Name of the computer that hosts the head node of your MATLAB Distributed Computing Server cluster (*NodeHost*).
- Name of the MJS on the head node host (*MJSName@NodeHost*).
- Name of a MATLAB cluster profile (*ClusterProfile*).

Example: `'-scheduler', 'myscheduler@mycompany.com'`

See Also

polyspaceCodeProver

Topics

“Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox)

“Run Remote Analysis at the Command Line”

Introduced in R2013b

polyspace-code-prover-nodesktop

Run a Code Prover verification from the DOS or UNIX command line

Syntax

```
polyspace-code-prover-nodesktop -sources sourceFiles  
polyspace-code-prover-nodesktop -sources sourceFiles -option value
```

```
polyspace-code-prover-nodesktop -sources-list-file listOfSources  
polyspace-code-prover-nodesktop -sources-list-file listOfSources -  
option value
```

```
polyspace-code-prover-nodesktop -options-file optFile
```

```
polyspace-code-prover-nodesktop -h[elp]
```

Description

`polyspace-code-prover-nodesktop -sources sourceFiles` runs a Code Prover verification on the source file or files `sourceFiles`. The verification uses the default analysis options.

`polyspace-code-prover-nodesktop -sources sourceFiles -option value` customizes the verification of `sourceFiles` with the `-option value` pairs specified.

`polyspace-code-prover-nodesktop -sources-list-file listOfSources` runs a Code Prover verification on the source files listed in the text file `listOfSources`. The verification uses the default analysis options.

`polyspace-code-prover-nodesktop -sources-list-file listOfSources -option value` customizes the verification of `listOfSources` with the `-option value` pairs specified.

`polyspace-code-prover-nodesktop -options-file optFile` runs a Code Prover verification with the options specified in the option file.

`polyspace-code-prover-nodesktop -h[elp]` lists a summary of possible analysis options.

Examples

Run Verification by Directly Specifying Options

Run a local Code Prover verification by specifying analysis options in the command itself. This example uses source files from the Polyspace Code Prover example. To run this example, replace `matlabroot` with the path to your MATLAB installation, for example `C:\Program Files\MATLAB\R2017a`.

Run a verification on `numerical.c` and `programming.c`, checking for MISRA C:2012 mandatory rules and using GNU 4.7 compiler settings. This example command is split by `^` characters for readability. In practice, you can put all commands on one line.

```
matlabroot\polyspace\bin\polyspace-code-prover-nodesktop -lang C^
-sources matlabroot\polyspace\examples\cxx\Code_Prover_Example\sources\*.c,^
-I matlabroot\polyspace\examples\cxx\Code_Prover_Example\sources\^
-compiler generic -misra3 mandatory^
-author jlittle -prog myProject -results-dir C:\Polyspace_Workspace\Results\
```

Open the results.

```
matlabroot\polyspace\bin\polyspace-code-prover C:\Polyspace_Workspace\Results\^
ps_results.psbf
```

To rerun the verification, you must rerun it from the command line.

Run Verification with Options File

Run a verification by using an options file to specify your source files and analysis options.

Save this text to a text file called `myOptsFile.txt`.

```
# Options for
-I matlabroot\polyspace\examples\cxx\Code_Prover_Example\sources
-verif-version 1.0
```

```
-sources-list-file matlabroot\polyspace\examples\cxx\Code_Prover_Example\sources\*.c
-lang C
-target i386
-compiler generic
-dos
-do-not-generate-results-for all-headers
-misra3 mandatory-required
-custom-rules matlabroot\polyspace\examples\cxx\Code_Prover_Example\sources\^
CustomRulesDefinition.txt
-entry-points procl,proc2,server1,server2,tregulate
-critical-section-begin Begin_CS:Cs10
-critical-section-end End_CS:Cs10
-temporal-exclusions-file matlabroot\polyspace\examples\cxx\Code_Prover_Example\^
sources\temporal_exclusions.txt
-float-rounding-mode to-nearest
-scalar-overflows-checks signed
-scalar-overflows-behavior truncate-on-error
-uncalled-function-checks none
-check-subnormal allow
-O2
-to Software Safety Analysis level 2
-context-sensitivity-auto
-path-sensitivity-delta 0
-author jlittle
-prog myProject
-results-dir C:\Polyspace_Workspace\Results\
```

Run the verification with the options specified in the text file.

```
matlabroot\polyspace\bin\polyspace-code-prover-nodesktop -options-file myOptsFile.txt
```

Open the results.

```
matlabroot\polyspace\bin\polyspace-code-prover C:\Polyspace_Workspace\Results\^
ps_results.pscp
```

To rerun the verification, you must rerun it from the command line.

- “Run Local Verification at Command Line”
- “Run Remote Analysis at the Command Line”

Input Arguments

sourceFiles — Comma-separated names of C or C++ files to analyze

`-sources string`

Comma-separated C or C++ source file names, specified as `-sources` followed by a string. If the files are not in the current folder (pwd), `sourceFiles` must include a full or relative path. For more information, see `-sources`.

Example: `-sources myFile.c,-sources C:\mySources\myFile1.c,C:\mySources\myFile2.c`

listOfSources — Text file listing names of C or C++ files to analyze

`-sources-list-file file`

Text file which lists the name of C or C++ files, specified as `-sources-list-file` followed by the file. If the files are not in the current folder (pwd), `listOfSources` must include a full or relative path. For more information, see `-sources-list-file`.

Example: `-sources-list-file filename.txt,-sources-list-file "C:\ps_analysis\source_files.txt"`

-option value — Analysis option and corresponding value

`option name`

Analysis options and their corresponding values, specified by the option name and if applicable value. For syntax specifications, see the individual analysis option reference pages.

Example: `-lang C-CPP,-OS-target no-predefined-OS`

optFile — Text file listing analysis options and values

`-options-file filepath`

Text file listing analysis options and values, specified as `-options-file` followed by the file. For more information, see `-options-file`.

Example: `-options-file opts.txt,-options-file "C:\ps_analysis\options.txt"`

See Also

`polyspaceCodeProver`

Topics

“Run Local Verification at Command Line”
“Run Remote Analysis at the Command Line”
“Analysis Options”

Introduced in R2013b

Polyspace Code Prover

Prove the absence of run-time errors in software

Description

The **Polyspace Code Prover** app uses static analysis and abstract interpretation to prove the absence of run-time errors in C and C++ code.

You can use it on handwritten code, generated code, or a combination of the two. Each operation is color-coded to indicate whether it is free of run-time errors, proven to fail, unreachable, or unproven. Polyspace Code Prover also displays range information for variables and function return values, and can prove which variables exceed specified range limits.

Open the Polyspace Code Prover App

- MATLAB Toolstrip: On the **Apps** tab, under **Code Verification**, click the app icon.
- MATLAB command prompt: Enter `polyspaceCodeProver`.

Examples

- “Run Verification”

Programmatic Use

```
polyspaceCodeProver
```

See Also

Apps
Polyspace Bug Finder

Functions

`polyspaceConfigure`

Topics

“Run Verification”

“Polyspace Code Prover”

Introduced in R2013b

polyspace.Project class

Package: polyspace

Run Polyspace analysis on C and C++ code and read results

Description

Run a Polyspace analysis on C and C++ source files by using this MATLAB object.

- To specify source files and customize analysis options, use the `Configuration` property.
- To run the analysis, use the `run` method.
- To read results after analysis, use the `Results` property.

Construction

`proj = polyspace.Project` creates an object that you can use to configure and run a Polyspace analysis, and then read the analysis results.

Properties

Configuration — Analysis options

`polyspace.Options` object

Options for running Polyspace analysis, implemented as a `polyspace.Options` object. The object has properties corresponding to the analysis options. For more information on those properties, see `polyspace.Options`.

You can retain the default options or change them in one of these ways:

- Modify the properties directly.

```
proj = polyspace.Project;  
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
```

- Obtain the options from another `polyspace.Project` object.

```
proj1 = polyspace.Project;
proj1.Configuration.TargetCompiler.Compiler = 'gnu4.9';

proj2 = proj1;
```

To use common analysis options across multiple projects, follow this approach. For instance, you want to reuse all options and change only the source files.

- Obtain the options from a project created in the user interface (`.psprj` file).

```
proj = polyspace.Project;
projectLocation = fullfile(matlabroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'Bug_Finder_Example.psprj')
proj.Configuration = polyspace.loadProject(projectLocation);
```

To determine the optimal set of options, set your options in the user interface and then import them to a `polyspace.Project` object. In the user interface, you can access help from features such as the Compilation Assistant and get tooltip help on options.

- Obtain the options from a Simulink model. Before obtaining the options, generate code from the model.

```
modelName = 'sldemo_bounce';
load_system(modelName);

% Set parameters for Embedded Coder target
set_param(modelName, 'SystemTargetFile', 'ert.tlc');
set_param('sldemo_bounce', 'Solver', 'FixedStepAuto');
set_param('sldemo_bounce', 'SupportContinuousTime', 'on')

% Generate code
rtwbuild(modelName);

% Obtain configuration from model
proj = polyspace.Project;
proj.Configuration = polyspace.ModelLinkOptions(modelName);
```

Use the options to analyze the code generated from the model.

Results — Analysis results

`polyspace.BugFinderResults` or `polyspace.CodeProverResults` object

Results of Polyspace analysis. When you create a `polyspace.Project` object, this property is initially empty. The property is populated only after you execute the `run` method of the object. Depending on the argument to the `run` method, `'bugFinder'` or `'codeProver'`, the property is implemented as a `polyspace.BugFinderResults` or `polyspace.CodeProverResults` object.

To read the results, use these methods of the `polyspace.BugFinderResults` or `polyspace.CodeProverResults` object:

- `getSummary`: Obtain a summarized format of the results into a MATLAB table.

```
proj = polyspace.Project;
proj.Configuration.Sources = {fullfile(matlabroot, 'polyspace', 'examples', ...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.ResultsDir = fullfile(pwd, 'results');

proj.run('bugFinder');

resTable = proj.Results.getSummary('defects');
```

For more information, see `polyspace.BugFinderResults.getSummary` or `polyspace.CodeProverResults.getSummary`.

- `getResults`: Obtain the full results or a more readable format into a MATLAB table.

```
proj = polyspace.Project;
proj.Configuration.Sources = {fullfile(matlabroot, 'polyspace', 'examples', ...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.ResultsDir = fullfile(pwd, 'results');

proj.run('bugFinder');

resTable = proj.Results.getResults('readable');
```

For more information, see `polyspace.BugFinderResults.getResults` or `polyspace.CodeProverResults.getResults`.

Methods

`run` Run a Polyspace analysis

Examples

Check for Bugs

Run a Polyspace Bug Finder analysis on the example file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(matlabroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd, 'results');

% Run analysis
bfStatus = proj.run('bugFinder');

% Read results
bfSummary = proj.Results.getSummary('defects');
```

Prove Absence of Run-Time Errors

Run a Polyspace Code Prover analysis on the example file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a main function must be generated, if the function does not exist in the source code.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(matlabroot, 'polyspace', 'examples', ...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd, 'results');
proj.Configuration.CodeProverVerification.MainGenerator = true;
```



```
% Run analysis
cpStatus = proj.run('codeProver');

% Read results
cpSummary = proj.Results.getSummary('runtime');
```

Check for Bugs and MISRA C:2012 Violations

Run a Polyspace Bug Finder analysis on the example file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Enable checking of MISRA C:2012 rules. Check for the mandatory rules only.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(matlabroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd, 'results');
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = 'mandatory';

% Run analysis
bfStatus = proj.run('bugFinder');

% Read results
defectsSummary = proj.Results.getSummary('defects');
misraSummary = proj.Results.getSummary('misraC2012');
```

See Also

Topics

- “Run Polyspace Analysis by Using MATLAB Scripts”
- “Generate MATLAB Scripts from Polyspace User Interface”
- “Troubleshoot Polyspace Analysis from MATLAB”

Introduced in R2017b

polyspace.Options class

Package: polyspace

Create object for running Polyspace analysis on handwritten code

Note For easier scripting, specify the Polyspace® analysis options using the Configuration property of a polyspace.Project object. Do not create a polyspace.Options object directly.

Description

Run a Polyspace analysis from MATLAB by using an options object. To specify source files and customize analysis options, change the object properties.

To analyze model-generated code, use polyspace.ModelLinkOptions instead.

Construction

`opts = polyspace.Options` creates an object whose properties correspond to options for running a Polyspace analysis.

`proj = polyspace.Project` creates a polyspace.Project object. The object has a property Configuration, which is a polyspace.Options object.

`opts = polyspace.Options(lang)` creates a Polyspace options object with options that are applicable to the language lang.

`opts = polyspace.loadProject(projectFile)` creates a Polyspace options object from an existing Polyspace project projectFile. You set the options in your project in the Polyspace user interface and create the options object from that project for programmatically running the analysis.

Input Arguments

lang — Language of analysis

'C-CPP' (default) | 'C' | 'CPP'

The language of the analysis specified as 'C-CPP', 'C', or 'CPP'. This argument determines the object properties.

Data Types: `char`

projectFile — Name of .psprj file

character vector

Name of Polyspace project file with extension `.psprj`, specified as a character vector.

If the file is not in the current folder, `projectFile` must include a full or relative path. To identify the current folder, use `pwd`. To change the current folder, use `cd`.

Example: `'C:\projects\myProject.psprj'`

Properties

The object properties correspond to the analysis options for Polyspace projects. The properties are organized in the same categories as the Polyspace interface. The property names are a shortened version of the DOS/UNIX command-line name. For syntax details, see `polyspace.Options`.

Methods

<code>copyTo</code>	Copy common settings between Polyspace options objects
<code>generateProject</code>	Generate <code>psprj</code> project from options object
<code>toScript</code>	Add Polyspace options object definition to a script

Examples

Customize and Run Analysis

Create a Polyspace analysis options object and customize the properties. Then, run an analysis.

Create object and customize properties. In case you do not have write access to your current folder, a temporary folder is being used for storing analysis results.

```
sources = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', 'Bug_Finder_Example', ...
    'sources', 'numerical.c');
opts = polyspace.Options();
opts.Prog = 'MyProject';
opts.Sources = {sources};
opts.TargetCompiler.Compiler = 'gnu4.7';
opts.ResultsDir = tempname;
```

Run a Bug Finder analysis. To run a Code Prover analysis, use `polyspaceCodeProver` instead of `polyspaceBugFinder`.

```
results = polyspaceBugFinder(opts);
```

Open the results in the Polyspace user interface.

```
polyspaceBugFinder('-results-dir', opts.ResultsDir);
```

Run Polyspace by Generating a Project File

Create a Polyspace analysis options object and customize the properties. Then, run a Bug Finder analysis.

Create object and customize properties.

```
sources = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', 'Bug_Finder_Example', ...
    'sources', 'numerical.c');
opts = polyspace.Options();
opts.Prog = 'MyProject';
opts.Sources = {sources};
opts.TargetCompiler.Compiler = 'gnu4.7';
opts.ResultsDir = tempname;
```

Generate a Polyspace project, name it using the `Prog` property, and open the project in the Polyspace interface.

```
psprj = opts.generateProject(opts.Prog);  
polyspaceBugFinder(psprj);
```

You can also analyze the project from the command line. Run the analysis and open the results in the Polyspace interface.

```
results = polyspaceBugFinder(psprj, '-nodesktop');  
polyspaceBugFinder('-results-dir',opts.ResultsDir);
```

- “Run Polyspace Analysis by Using MATLAB Scripts”
- “Generate MATLAB Scripts from Polyspace User Interface”

Alternatives

If you are analyzing code generated from a model, use `polyspace.ModelLinkOptions` instead.

See Also

`polyspace.ModelLinkOptions` | `polyspace.Project` | `polyspaceCodeProver`

Topics

- “Run Polyspace Analysis by Using MATLAB Scripts”
- “Generate MATLAB Scripts from Polyspace User Interface”

Introduced in R2017a

polyspace.ModelLinkOptions class

Package: polyspace

Create object for running Polyspace analysis on generated code

Description

Run a Polyspace analysis from MATLAB by using an options object. To specify source files and customize analysis options, change the object properties.

This class is intended for model-generated code. If you are analyzing handwritten code, use `polyspace.Options` instead.

Construction

`opts = polyspace.ModelLinkOptions` creates an object whose properties correspond to options for running a Polyspace analysis on generated code.

`opts = polyspace.ModelLinkOptions(lang)` creates a Polyspace options object with options that are applicable to the language `lang`.

`opts = polyspace.ModelLinkOptions(model)` creates a Polyspace options object with options that are applicable to `model`. Prior to extracting options from the model, you must load the model and generate code.

Input Arguments

lang — Language of analysis

'C-CPP' (default) | 'C' | 'CPP'

The language of the analysis specified as 'C-CPP', 'C', or 'CPP'. This argument determines the object properties.

model — Model or subsystem name

character vector

Name or path to model or subsystem, specified as a character vector.

Prior to extracting options from the model, you must:

- 1 Load the model. Use `load_system` or `open_system`.
- 2 Generate code from the model. Use `rtwbuild`.

Example: `'psdemo_model_link_sl'`

Properties

The object properties correspond to the analysis options for Polyspace projects. The properties are organized in the same categories as the Polyspace interface. The property names are a shortened version of the DOS command-line name. For syntax details, see `polyspace.ModelLinkOptions`.

Methods

<code>copyTo</code>	Copy common settings between Polyspace options objects
<code>generateProject</code>	Generate <code>psprj</code> project from options object
<code>toScript</code>	Add Polyspace options object definition to a script

Examples

Script Analysis of Model Generated Code

This example shows how to customize and run an analysis on code generated from a model.

Generate code from the model `sldemo_bounce`. Before code generation, set a system target file appropriate for code analysis. See also “Recommended Model Settings for Code Analysis”.

```
modelName = 'sldemo_bounce';  
load_system(modelName);
```



```

% Set parameters for Embedded Coder target
set_param(modelname, 'SystemTargetFile', 'ert.tlc');
set_param('sldemo_bounce', 'Solver', 'FixedStepAuto');
set_param('sldemo_bounce', 'SupportContinuousTime', 'on')

if exist(fullfile(pwd, 'sldemo_bounce_ert_rtw'), 'dir') == 0
    rtwbuild(modelName);
end

```

Associate a `polyspace.ModelLinkOptions` object with the model. A subset of the object properties are set from the configuration parameters associated with the model. The other properties take their default values. For details on the configuration parameters, see “Configure Code Verification”.

```
opts = polyspace.ModelLinkOptions(modelName);
```

Change the property values if needed. For instance, you can specify that the analysis must check for all MISRA C: 2012 violations and generate a PDF report of the results. You can also specify a folder for the analysis results.

```

opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
opts.CodingRulesCodeMetrics.MisraC3Subset = 'all';

opts.MergedReporting.EnableReportGeneration = true;
opts.MergedReporting.ReportOutputFormat = 'PDF';

opts.ResultsDir = 'newResfolder';

```

Create a `polyspace.Project` object. Associate the Configuration property of this object to the options that you previously specified.

```
proj = polyspace.Project;
proj.Configuration = opts;
```

Run analysis and open results.

```
cpStatus = proj.run('codeProver');
proj.Results.getResults('readable');
```

- “Run Polyspace Analysis by Using MATLAB Scripts”

Alternatives

If you are analyzing handwritten code, use a `polyspace.Project` object directly. Alternatively, use a `polyspace.Options` object.

See Also

`polyspace.Options` | `polyspace.Project` | `polyspaceCodeProver` | `pslinkrun`

Topics

“Run Polyspace Analysis by Using MATLAB Scripts”

Introduced in R2017a

polyspace.CodeProverOptions class

Package: polyspace

Create Polyspace Code Prover object for handwritten code

Note This class is deprecated and will be removed in a future release. Use `polyspace.Options` instead.

Description

Customize a Polyspace Code Prover verification from MATLAB by creating a Code Prover options object. To specify source files and customize analysis options, change the object properties.

If you are verifying model-generated code, use `polyspace.ModelLinkCodeProverOptions` instead.

Construction

`opts = polyspace.CodeProverOptions` creates a Code Prover options object with options for C code verification.

`opts = polyspace.CodeProverOptions(lang)` creates a Code Prover options object with options applicable for the language `lang`.

Input Arguments

lang — Language of analysis

'C' (default) | 'CPP'

Language of verification specified as either 'C' or 'CPP'. This argument determines which properties the object has.

Properties

The object properties match the analysis options found in the Polyspace interface. For syntax details, see `polyspace.Options`.

Methods

<code>copyTo</code>	Copy common settings between Polyspace options objects
<code>generateProject</code>	Generate psprj project from options object
<code>toScript</code>	Add Polyspace options object definition to a script

Examples

Use Code Prover Object to Customize and Run Verification

Create a Code Prover options object and customize the properties. Then, run a verification.

Create object and customize properties.

```
sources = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', 'Code_Prover_Example', ...  
'sources', 'single_file_analysis.c');  
includes = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', 'Code_Prover_Example', ...  
'sources');  
optsCP = polyspace.CodeProverOptions();  
optsCP.Prog = 'MyProject';  
optsCP.Sources = {sources};  
optsCP.EnvironmentSettings.IncludeFolders = {includes};  
optsCP.TargetCompiler.Compiler = 'gnu4.7';  
optsCP.ResultsDir = tempname;
```

Run the analysis and open the results in the Polyspace interface.

```
results = polyspaceCodeProver(optsCP);  
polyspaceCodeProver('-results-dir', optsCP.ResultsDir);
```

Run Polyspace by Generating a Project File

Create a Code Prover analysis options object and customize the properties. Then, run an analysis.

Create object and customize properties.

```
sources = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', 'Code_Prover_Example', ...  
'sources', 'single_file_analysis.c');  
includes = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', 'Code_Prover_Example', ...  
'sources');  
optsCP = polyspace.CodeProverOptions();  
optsCP.Prog = 'MyProject';  
optsCP.Sources = {sources};  
optsCP.EnvironmentSettings.IncludeFolders = {includes};  
optsCP.TargetCompiler.Compiler = 'gnu4.7';  
optsCP.ResultsDir = tempname;
```

Generate a Polyspace project, name it using the `Prog` property, and open the project in the Polyspace interface.

```
psprj = generateProject(optsCP, optsCP.Prog);  
polyspaceCodeProver(psprj);
```

You can also analyze the project from the command line. Run the analysis and open the results in the Polyspace interface.

```
results = polyspaceCodeProver(psprj, '-nodesktop');  
polyspaceCodeProver('-results-dir', optsCP.ResultsDir);
```

Alternatives

If you are verifying model-generated code, use `polyspace.ModelLinkCodeProverOptions` instead.

See Also

`polyspace.ModelLinkCodeProverOptions` | `polyspace.Options` | `polyspaceCodeProver`

Topics

“Run Polyspace Analysis by Using MATLAB Scripts”

Introduced in R2016b

polyspace.ModelLinkCodeProverOptions class

Package: polyspace

Create Polyspace Code Prover object for generated code

Note This class is deprecated and will be removed in a future release. Use `polyspace.ModelLinkOptions` instead.

Description

Customize a Polyspace Code Prover verification from MATLAB by creating a Code Prover options object. To specify source files and customize analysis options, change the object properties.

If you are verifying handwritten code, use `polyspace.CodeProverOptions` instead.

Construction

`opts = polyspace.ModelLinkCodeProverOptions` creates a Code Prover options object with options for C code verification.

`opts = polyspace.ModelLinkCodeProverOptions(lang)` creates a Code Prover options object with options applicable for the language `lang`.

Input Arguments

lang — Language of analysis

C (default) | CPP

Language of verification specified as either 'C' or 'CPP'. This argument determines which properties the object has.

Example: `opts = polyspace.ModelLinkCodeProverOptions('CPP')`

Properties

The object properties are the analysis options for Polyspace Code Prover model link projects. The properties are organized in the same categories as in the Polyspace interface. The property names are a shortened version of the DOS command-line name. For syntax details, see `polyspace.ModelLinkOptions`.

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

To copy properties between Polyspace objects, use `.` You can copy corresponding properties between `BugFinderOptions` objects and `CodeProverOptions` objects.

Examples

Script Verification of Model Generated Code

This example shows how to customize and run a verification on model-generated code with MATLAB functions and objects.

Create a custom configuration that checks MISRA C 2012 rules and generates a PDF report.

```
opts = polyspace.ModelLinkCodeProverOptions('C');
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
opts.CodingRulesCodeMetrics.MisraC3Subset = 'all';
opts.MergedReporting.ReportOutputFormat = 'PDF';
opts.MergedReporting.EnableReportGeneration = true;
```

Generate code from `psdemo_model_link_sl`.

```
model = 'psdemo_model_link_sl';
load_system(model);
slbuild(model);
```


Add the configuration to `pslinkoptions` object.

```
prjfile = opts.generateProject('model_link_opts');  
mlopts = pslinkoptions(model);  
mlopts.EnablePrjConfigFile = true;  
mlopts.PrjConfigFile = prjfile;  
mlopts.VerificationMode = 'CodeProver';
```

Run the verification.

```
[polyspaceFolder, resultsFolder] = pslinkrun(model);
```

- “Run Polyspace Analysis by Using MATLAB Scripts” (Polyspace Bug Finder)

Alternatives

If you are verifying handwritten code, use `polyspace.CodeProverOptions` instead.

See Also

[polyspace.CodeProverOptions](#) | [polyspace.ModelLinkOptions](#) |
[polyspaceCodeProver](#) | [pslinkrun](#)

Topics

“Run Polyspace Analysis by Using MATLAB Scripts” (Polyspace Bug Finder)

Introduced in R2016b

polyspace.GenericTargetOptions class

Package: polyspace

Create a generic target configuration

Description

If your target processor does not match one of the preset targets, use this object to create a custom generic target. To use your custom target in an analysis, you must add it to a `polyspace.CodeProverOptions` or `polyspace.ModelLinkCodeProverOptions` object. In your options object, add your generic target options object to the `TargetCompiler.Target` property.

Construction

`genericTarget = polyspace.GenericTargetOptions` creates a generic target that you can customize. To specify the size and alignment of types, change the properties of the `genericTarget` object.

Properties

For more details about any of these properties, see `Generic target options`.

Alignment — Largest alignment of struct or array objects

32 (default) | 16 | 8

Largest alignment of struct or array objects, specified as 32, 16, or 8. Comparable with the DOS/UNIX command-line option `-align`.

Example: `target.Alignment = 8`

CharNumBits — Define the number of bits for a char

8 (default) | 16

Define the number of bits for a char, specified as 8 or 16. Comparable with the DOS/UNIX command-line option `-char-is-16bits`.

Example: `target.CharNumBits = 16`

DoubleNumBits — Define the number of bits for a double

32 (default) | 64

Define the number of bits for a double, specified as 32 or 64. Comparable with the DOS/UNIX command-line option `-double-is-64bits`.

Example: `target.DoubleNumBits = 64`

Endianness — Endianness of target architecture

little (default) | big

Endianness of target architecture, specified as `little` or `big`. Comparable with the DOS/UNIX command-line options `-little-endian` or `-big-endian`.

Example: `target.Endianness = 'big'`

IntNumBits — Define the number of bits for an int

16 (default) | 32

Define the number of bits for an int, specified as 16 or 32. Comparable with the DOS/UNIX command-line option `-int-is-32bits`.

Example: `target.IntNumBits = 32`

LongLongNumBits — Define the number of bits for a long long

32 (default) | 64

Define the number of bits for a long long, specified as 32 or 64. Comparable with the DOS/UNIX command-line option `-long-long-is-64bits`.

Example: `target.LongNumBits = 64`

LongNumBits — Define the number of bits for a long

32 (default)

Define the number of bits for a long, specified as 32. Comparable with the DOS/UNIX command-line option `-long-is-32bits`.

Example: `target.LongNumBits = 32`

PointerNumBits — Define the number of bits for a pointer

16 (default) | 24 | 32

Define the number of bits for a pointer, specified as 16, 24, or 32. Comparable with the DOS/UNIX command-line options `-pointer-is-24bits` and `-pointer-is-32bits`.

Example: `target.PointerNumBits = 32`

ShortNumBits — Define the number of bits for a short

16 (default) | 8

Define the number of bits for an `int`, specified as 16 or 8. Comparable with the DOS/UNIX command-line option `-short-is-8bits`.

Example: `target.ShortNumBits = 8`

SignOfChar — Default sign of plain char

signed (default) | unsigned

Default sign of plain `char`, specified as `signed` or `unsigned`. Comparable with the DOS/UNIX command-line option `-default-sign-of-char`.

Example: `target.SignOfChar = 'unsigned'`

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Examples

Customize Generic Target Settings

Use a `polyspace.GenericTargetOptions` object to customize a generic target for your analysis.

Create options objects.

```
target = polyspace.GenericTargetOptions;  
opts = polyspace.Options;
```

Add the custom target to the Bug Finder options object.

```
opts.TargetCompiler.Target = target;
```

Customize the generic target.

```
target.Endianness = 'big';  
target.LongLongNumBits = 64;  
target.ShortNumBits = 8;
```

See Also

Target processor type (-target) | polyspace.CodeProverOptions |
polyspace.ModelLinkCodeProverOptions

Introduced in R2016b

polyspace.CodingRulesOptions class

Package: polyspace

Create custom list of coding rules to check

Description

Create a custom list of coding rules to check for one of the supported standard coding rule sets. To use your custom target in an analysis, you must add it to a `polyspace.Options` or `polyspace.ModelLinkOptions` object. In your options object:

- Add your coding rules options object to one of the `CodingRulesCodeMetrics.RULESETSubset` properties.
- Activate your coding rule set with one of the `CodingRulesCodeMetrics.EnableRULESET` properties.

Construction

`ruleList = polyspace.CodingRulesOptions(RuleSet)` creates the coding rules object `ruleList` for the `RuleSet` coding rule set. Set the active rules in the coding rules object.

Input Arguments

RuleSet — Standard coding rule set

`misraC` (default) | `misraC2012` | `misraAcAgc` | `misraCpp` | `jsf`

Standard coding rule set specified as one of the coding rule acronyms.

Example: `'misraCpp'`

Data Types: `char`

Properties

For each coding rule set, an object is created with all supported rules for that rule set. By default, all rules are on. To turn off a rule, set the rule to false. For example:

```
ruleList.rule_20_1 = false
```

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Customize List of Coding Rules to Check

Customize the coding rules that are checked during your analysis by using a coding rules options object.

Create options objects.

```
misraRules = polyspace.CodingRulesOptions('misraC2012');  
opts = polyspace.Options;
```

Add the customized list of coding rules to the Bug Finder options object and activate them.

```
opts.CodingRulesCodeMetrics.MisraC3Subset = misraRules;  
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
```

Customize the coding rule list by turning off rules 2.1-2.7.

```
misraRules.rule_2_1 = false;  
misraRules.rule_2_2 = false;  
misraRules.rule_2_3 = false;  
misraRules.rule_2_4 = false;  
misraRules.rule_2_5 = false;
```

```
misraRules.rule_2_6 = false;  
misraRules.rule_2_7 = false;
```

- “Select Specific MISRA or JSF Coding Rules” (Polyspace Bug Finder)

See Also

`polyspace.CodeProverOptions` | `polyspace.ModelLinkCodeProverOptions`

Topics

“Select Specific MISRA or JSF Coding Rules” (Polyspace Bug Finder)

Introduced in R2016b

polyspace.CodeProverResults class

Package: polyspace

Read Polyspace Code Prover results from MATLAB

Description

Read Polyspace Code Prover analysis results to MATLAB tables by using this object.

You can obtain a high-level overview or read each individual result, for example, each instance of a run-time check.

Construction

`resObj = polyspace.CodeProverResults(resultsFolder)` creates an object for reading a specific set of Code Prover results into MATLAB tables. Use the object methods to read the results.

`proj = polyspace.Project` creates a `polyspace.Project` object. The object has a property `Results`. If you run a Code Prover analysis, this property is a `polyspace.CodeProverResults` object.

Input Arguments

resultsFolder — Name of result folder

character vector

Name of result folder, specified as a character vector. The folder must contain the results file with extension `.pscp`. Even if the results file resides in a *subfolder* of the specified folder, it cannot be accessed.

If the folder is not in the current folder, `resultsFolder` must include a full or relative path.

Example: `'C:\Polyspace\Results\'`

Methods

`getSummary` View number of run-time checks organized by color and file
`getResults` Read Code Prover results into MATLAB table

Examples

Copy Existing Results to MATLAB Tables

This example shows how to read Code Prover analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', 'Code_Prover_Example', ...  
'Module_1', 'CP_Result');  
userResPath = tempname;  
copyfile(resPath, userResPath);
```

Create the results object.

```
resObj = polyspace.CodeProverResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = getSummary (resObj);  
resTable = getResults (resObj);
```

Run Analysis and Read Results to MATLAB Tables

Run a Polyspace Code Prover analysis on the demo file `single_file_analysis.c`.
Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a main function must be generated, if it does not exist in the source code.

```
proj = polyspace.Project  
  
% Configure analysis
```

```
proj.Configuration.Sources = {fullfile(matlabroot, 'polyspace', 'examples', ...  
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};  
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';  
proj.Configuration.ResultsDir = fullfile(pwd, 'results');  
proj.Configuration.CodeProverVerification.MainGenerator = true;  
  
% Run analysis  
cpStatus = proj.run('codeProver');  
  
% Read results  
cpSummary = proj.Results.getResults('readable');
```

Alternatives

To read Bug Finder results from MATLAB, use the class `polyspace.BugFinderResults`.

Introduced in R2017a

copyTo

Class: polyspace.Options

Package: polyspace

Copy common settings between Polyspace options objects

Syntax

```
optsFrom.copyTo(optsTo)
```

Description

`optsFrom.copyTo(optsTo)` copies the common options from `optsFrom` to `optsTo`. The options objects do not need to be the same type of options object. This method copies only properties that are common between the two objects.

Input Arguments

optsFrom — Options object you want to copy properties from

`polyspace.Options` or `polyspace.ModelLinkOptions` object

Option object that you want to copy properties from, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

optsTo — Options object you want to copy properties to

`polyspace.Options` object

Option object that you want to copy properties to, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

Examples

Copy Polyspace Options Object

This example shows how to set the properties of one options object and then copy that object to another one.

Create a Polyspace options object and set properties.

```
opts1 = polyspace.Options();  
opts1.Prog = 'DataRaceProject';  
opts1.Sources = {'datarace.c'};  
opts1.TargetCompiler.Compiler = 'diab';
```

Create another object and use `copyTo` to copy over options from the previous object.

```
opts2 = polyspace.Options();  
copyTo(opts1, opts2);
```

See Also

`polyspace.CodeProverOptions` | `polyspace.ModelLinkCodeProverOptions` |
`polyspace.Options` | `polyspace.Options.generateProject`

Introduced in R2016b

generateProject

Class: polyspace.Options

Package: polyspace

Generate psprj project from options object

Syntax

```
opts.generateProject(projectName)
```

Description

`opts.generateProject(projectName)` creates a `.psprj` project called `projectName` from the options specified in the `polyspace.Options` object `opts`.

Input Arguments

opts — Options object to convert into a `psprj` file

`polyspace.Options` or `polyspace.ModelLinkOptions` object

Option object to convert into a `psprj` file specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

projectName — Project file name

character vector

Project file name specified as a character vector. This argument is used as the name of the `psprj` file.

Example: `'myProject'`

Examples

Generate Project from a Bug Finder Options Object

This example shows how to create and use a Polyspace project that was generated from an options object.

Create a Bug Finder object and set properties.

```
sources = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', 'Bug_Finder_Example', ...  
    'sources', 'numerical.c');  
opts = polyspace.Options();  
opts.Prog = 'MyProject';  
opts.Sources = {sources};  
opts.TargetCompiler.Compiler = 'gnu4.7';
```

Generate a Polyspace project. Name the project using the Prog property.

```
psprj = opts.generateProject(opts.Prog);
```

Run a Bug Finder analysis using one of these commands. Both commands produce identical analysis results. The only difference is that the `psprj` project can be rerun in the Polyspace interface.

```
polyspaceBugFinder(psprj, '-nodesktop');  
polyspaceBugFinder(opts);
```

To run a Code Prover analysis, use `polyspaceCodeProver` instead of `polyspaceBugFinder`.

Tips

If you want to include an options object in a `pslinkoptions` object:

- 1 Use this method to convert your object to a project.
- 2 Add the project to the `pslinkoptions` property `PrjConfig`.
- 3 Turn on the property `EnablePrjConfig`.

See Also

`polyspace.CodeProverOptions` | `polyspace.ModelLinkCodeProverOptions` |
`polyspace.Options` | `polyspace.Options.copyTo`

Introduced in R2016b

toScript

Class: polyspace.Options

Package: polyspace

Add Polyspace options object definition to a script

Syntax

```
filePath = opts.toScript(fileName,positionInScript)
```

Description

`filePath = opts.toScript(fileName,positionInScript)` adds the properties of a `polyspace.Options` object to a MATLAB script. The script shows the values assigned to all the properties of the object. You can run the script later to define the object in the MATLAB workspace and use it.

Input Arguments

opts — Options object with Polyspace analysis options

`polyspace.Options` or `polyspace.ModelLinkOptions` object

Option object to store in MATLAB script, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

fileName — Script name

Character vector

Name or path to script, specified as a character vector. If you specify a relative path, the script is created in subfolder of the current working folder.

Example: `'runPolyspace.m'`

positionInScript — Where to add object definition

'create' (default) | 'append'

Position in script where the object properties are added, specified as 'create' or 'append'. If you specify 'append', the object properties are added to the end of an existing script. Otherwise, a new script is created.

Output Arguments

filePath — Full path to script

Character vector

Full path to script, specified as a character vector.

Example: 'C:\myScripts\runPolyspace.m'

See Also

[polyspace.CodeProverOptions](#) | [polyspace.ModelLinkCodeProverOptions](#) | [polyspace.Options](#) | [polyspace.Options.generateProject](#)

Topics

“Generate MATLAB Scripts from Polyspace User Interface”

Introduced in R2017b

run

Class: polyspace.Project

Package: polyspace

Run a Polyspace analysis

Syntax

```
proj.run(product)
```

Description

`status = proj.run(product)` runs a Polyspace Bug Finder or Polyspace Code Prover analysis using the configuration specified in the `polyspace.Project` object `proj`. The analysis results are also stored in `proj`.

Input Arguments

proj — Polyspace project

`polyspace.Project` object

Polyspace project with configuration and results, specified as a `polyspace.Project` object.

product — Type of analysis

'bugFinder' | 'codeProver'

Type of analysis to run.

Output Arguments

status — Results of a Code Prover analysis

true | false

Status of analysis. If the analysis fails, the status is `false`. Otherwise, it is `true`.

The analysis can fail for multiple reasons:

- You provide source files that do not exist.
- None of your files compile. Even if one file compiles, unless you set the property `StopWithCompileError` to `true`, the analysis succeeds and returns a `true` status.

There can be many other reasons why the analysis fails. If the analysis fails, in your results folder, check the log file. You can see the results folder using the `Configuration` property of the `polyspace.Project` object:

```
proj = polyspace.Project;  
proj.Configuration.ResultsDir
```

The log file is named `Polyspace_R20##n_ProjectName_date-time.log`.

Examples

Read Results to MATLAB Tables

Run a Polyspace Bug Finder analysis on the demo file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project  
  
% Configure analysis  
proj.Configuration.Sources = {fullfile(matlabroot, 'polyspace', ...  
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};  
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';  
proj.Configuration.ResultsDir = fullfile(pwd, 'results');  
  
% Run analysis  
bfStatus = proj.run('bugFinder');
```

```
% Read results  
bfSummary = proj.Results.getSummary('defects');
```

Introduced in R2017b

getSummary

Class: polyspace.CodeProverResults

Package: polyspace

View number of run-time checks organized by color and file

Syntax

```
resObj.getSummary(resultsType)
```

Description

`resSummary = resObj.getSummary(resultsType)` returns the distribution of results of type `resultsType` in a Code Prover result set denoted by the `polyspace.CodeProverResults` object `resObj`. For instance, if you choose to see run-time checks, you see how many red, orange, gray and green checks are present in each file.

Input Arguments

resultsType — Type of Code Prover analysis result

```
'runtime' (default) | 'misraC' | 'misraCAGC' | 'misraCPP' | 'misraC2012' |  
'jsf' | 'metrics' | 'customRules'
```

Type of result, specified as a character vector.

Entry	Meaning
'runtime'	Checks for run-time errors. See “Run-Time Checks”.
'misraC'	MISRA C:2004 rules. See “MISRA C:2004 Rules”.
'misraCAGC'	MISRA C:2004 rules for generated code. See “MISRA C:2004 Rules”.

Entry	Meaning
'misraCPP'	MISRA C++ rules. See “MISRA C++:2008 Rules”.
'misraC2012'	MISRA C:2012 rules. See “MISRA C:2012 Directives and Rules”.
'jsf'	JSF C++ rules. See “JSF C++ Rules”.
'metrics'	Code complexity metrics. See “Code Metrics”.
'customRules'	Custom rules enforcing naming conventions for identifiers. See “Custom Coding Rules”.

Output Arguments

resSummary — Distribution of run-time checks by check color and file table

Distribution of run-time checks by check color and file, specified as a table. For instance, an extract of the table looks like this:

File	Proven	Green	Red	Gray	Orange
file1.c	92.0%	87	3	2	8
file2.c	97.7%	41	0	1	1

The table above shows that `file1.c` has:

- 3 red, 2 gray and 8 orange checks.
- 92% of operations proven.

In other words, of every 100 operations that the verification checked, 92 operations were proven green, red or gray. See “Result and Source Code Colors”.

For more information on MATLAB tables, see “Tables” (MATLAB).

Examples

Copy Existing Results to MATLAB Tables

This example shows how to read Code Prover analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', 'Code_Prover_Example', ...  
'Module_1', 'CP_Result');  
userResPath = tempname;  
copyfile(resPath, userResPath);
```

Create the results object.

```
resObj = polyspace.CodeProverResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = resObj.getSummary('runtime');  
resTable = resObj.getResults();
```

Run Analysis and Read Results to MATLAB Tables

Run a Polyspace Code Prover analysis on the demo file `single_file_analysis.c`.

Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a main function must be generated, if it does not exist in the source code.

```
proj = polyspace.Project  
  
% Configure analysis  
proj.Configuration.Sources = {fullfile(matlabroot, 'polyspace', 'examples', ...  
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};  
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';  
proj.Configuration.ResultsDir = fullfile(pwd, 'results');  
proj.Configuration.CodeProverVerification.MainGenerator = true;  
  
% Run analysis  
cpStatus = proj.run('codeProver');
```



```
% Read results  
cpSummary = proj.Results.getResults('readable');
```

See Also

`polyspace.CodeProverResults`

Topics

“Result and Source Code Colors”

Introduced in R2017a

getResults

Class: polyspace.CodeProverResults

Package: polyspace

Read Code Prover results into MATLAB table

Syntax

```
resObj.getResults(content)
```

Description

`resTable = resObj.getResults(content)` returns a table showing all results in a Code Prover result set denoted by the `polyspace.CodeProverResults` object `resObj`. You can manipulate the table to produce graphs and statistics about your results that you cannot obtain readily from the user interface.

Input Arguments

content — Result information to include

'full' (default) | 'readable'

Amount of information to be included for each result. If you specify 'full', all information is included. See “Export Polyspace Analysis Results”. If you specify 'readable', the following information is not included:

- ID: Unique number for a result for the current analysis.
- Group: Check groups, MISRA C:2012 groups, etc.
- Status, Severity, Comment: Information that *you* enter about a result.

If you do not specify this argument, the full table is included.

Output Arguments

resTable — Results of a Code Prover analysis

table

Table showing all results from a single Code Prover analysis. For each result, the table has information such as file, family, and so on. If a particular information is not available for a result, the entry in the table states <undefined>.

For more information on:

- The columns of the table, see “Export Polyspace Analysis Results”.
- MATLAB tables, see “Tables” (MATLAB).

Examples

Copy Existing Results to MATLAB Tables

This example shows how to read Code Prover analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', 'Code_Prover_Example', ...
    'Module_1', 'CP_Result');
userResPath = tempname;
copyfile(resPath, userResPath);
```

Create the results object.

```
resObj = polyspace.CodeProverResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = getSummary (resObj);
resTable = getResults (resObj);
```

Run Analysis and Read Results to MATLAB Tables

Run a Polyspace Code Prover analysis on the demo file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a main function must be generated, if it does not exist in the source code.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(matlabroot, 'polyspace', 'examples', ...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd, 'results');
proj.Configuration.CodeProverVerification.MainGenerator = true;

% Run analysis
cpStatus = proj.run('codeProver');

% Read results
cpSummary = proj.Results.getResults('readable');
```

See Also

`polyspace.CodeProverResults`

Introduced in R2017a

pslinkoptions Properties

Properties for the `pslinkoptions` object

Description

You can create a `pslinkoptions` object to customize your analysis at the command-line. Use these properties to specify configuration options, where and how to store results, additional files to include, and data range modes.

Properties

Configuration Options

VerificationSettings — Coding rule and configuration settings for C code

```
'PrjConfig' (default) | 'PrjConfigAndMisraAGC' | 'PrjConfigAndMisra' |  
'PrjConfigAndMisraC2012' | 'MisraAGC' | 'Misra' | 'MisraC2012'
```

Coding rule and configuration settings for C code specified as:

- `'PrjConfig'` – Inherit options from the project configuration.
- `'PrjConfigAndMisraAGC'` – Inherit options from the project configuration and enable MISRA AC AGC rule checking.
- `'PrjConfigAndMisra'` – Inherit options from the project configuration and enable MISRA C:2004 rule checking.
- `'PrjConfigAndMisraC2012'` – Inherit options from the project configuration and enable MISRA C:2012 guideline checking.
- `'MisraAGC'` – Enable MISRA AC AGC rule checking. This option runs only compilation and rule checking.
- `'Misra'` – Enable MISRA C:2004 rule checking. This option runs only compilation and rule checking.
- `'MisraC2012'` – Enable MISRA C:2012 rule checking. This option runs only compilation and guideline checking.

Example: `opt.VerificationSettings = 'PrjConfigAndMisraC2012'`

VerificationMode — Polyspace mode

'CodeProver' (default) | 'BugFinder'

Polyspace mode specified as 'BugFinder', for a Bug Finder analysis, or 'CodeProver', for a Code Prover verification.

Example: `opt.VerificationMode = 'BugFinder';`

EnablePrjConfigFile — Allow a custom configuration file

false (default) | true

Allows a custom configuration file instead of the default configuration specified as true or false. Use the `PrjConfigFile` option to specify the configuration file.

Example: `opt.EnablePrjConfigFile = true;`

PrjConfigFile — Custom configuration file

' ' (default) | full path to a .prprj file

Custom configuration file to use instead of the default configuration specified by the full path to a .psprj file. Use the `EnablePrjConfigFile` option to use this configuration file during your analysis.

Example: `opt.PrjConfigFile = 'C:\Polyspace\config.psprj';`

CheckConfigBeforeAnalysis — Configuration check before analysis

'OnWarn' (default) | 'OnHalt' | 'Off'

This property sets the level of configuration checking done before the analysis starts. The configuration check before analysis is specified as:

- **'Off'** — Checks only for errors. Stops if errors are found.
- **'OnWarn'** — Stops for errors. Displays a message for warnings.
- **'OnHalt'** — Stops for errors and warnings.

Example: `opt.CheckConfigBeforeAnalysis = 'OnHalt';`

Results

ResultDir — Results folder name and location

'{C:\Polyspace_Results\results_ \$ModelName\$' (default) | folder name | folder path

Results folder name and location specified as the local folder name or the folder path. This folder is where Polyspace writes the analysis results. This folder name can be either an absolute path or a path relative to the current folder. The text `$ModelName$` is replaced with the name of the original model.

Example: `opt.ResultDir = '\results_v1_ $ModelName$';`

AddSuffixToResultDir — Add unique number to the results folder name

`false` (default) | `true`

Add unique number to the results folder name specified as `true` or `false`. If `true`, a unique number is added to the end of every new result. Using this option helps you avoid overwriting the previous results folders.

Example: `opt.AddSuffixToResultDir = true;`

OpenProjectManager — Open the Polyspace environment

`false` (default) | `true`

Open the Polyspace environment to monitor the progress of the analysis, specified as `true` or `false`. Afterward, you can review the results.

Example: `opt.OpenProjectManager = true;`

AddToSimulinkProject — Add results to the open Simulink project

`false` (default) | `true`

Add your results to the currently open Simulink project, if any, specified as `true` or `false`. This option allows you to keep your Polyspace results organized with the rest of your project files. If a Simulink project is not open, the results are not added to a Simulink project.

Example: `opt.AddToSimulinkProject = true;`

Additional Files

EnableAdditionalFileList — Allow an additional file list

`false` (default) | `true`

Allow an additional file list to be analyzed, specified as `true` or `false`. Use with the `AdditionalFileList` option.

Example: `opt.EnableAdditionalFileList = true;`

AdditionalFileList — List of additional files to be analyzed

{0x1 cell} (default) | cell array of files

List of additional files to be analyzed specified as a cell array of files. Use with the `EnableAdditionalFileList` option to add these files to the analysis.

Example: `opt.AdditionalFileList = {'sources\file1.c', 'sources\file2.c'};`

Data Types: `cell`

Data Ranges

InputRangeMode — Enable design range information

'DesignMinMax' (default) | 'FullRange'

Enable design range information specified as 'DesignMinMax', to use data ranges defined in blocks and workspaces, or 'FullRange', to treat inputs as full-range values.

Example: `opt.InputRangeMode = 'FullRange';`

ParamRangeMode — Enable constant parameter values

'None' (default) | 'DesignMinMax'

Enable constant parameter values, specified as 'None', to use constant parameters values specified in the code, or 'DesignMinMax' to use a range defined in blocks and workspaces.

Example: `opt.ParamRangeMode = 'DesignMinMax';`

OutputRangeMode — Enable output assertions

'None' (default) | 'DesignMinMax'

Enable output assertions specified by 'None', to not apply assertions, or 'DesignMinMax' to apply assertions to outputs using a range defined in blocks and workspace.

Example: `opt.ParamRangeMode = 'DesignMinMax';`

Embedded Coder Only

ModelRefVerifDepth — Depth of verification

'Current model only' (default) | '1' | '2' | '3' | 'All'

Depth of verification specified by the model reference level to which you want to analyze.

Only for Embedded Coder

Example: `opt.ModelRefVerifDepth = '3';`

ModelRefByModelRefVerif — Model reference analysis mode

false (default) | true

Model reference analysis mode specified as false to verify reference models within the model hierarchy, or true to verify referenced models individually.

Only for Embedded Coder

Example: `opt.ModelRefByModelRefVerif = true;`

CxxVerificationSettings — Coding rule and configuration settings for C++ code

'PrjConfig' (default) | 'PrjConfigAndMisraCxx' | 'PrjConfigAndJSF' |
'MisraCxx' | 'JSF'

Coding rule and configuration settings for C++ code specified as:

- 'PrjConfig' – Inherit options from project configuration and run complete analysis.
- 'PrjConfigAndMisraCxx' – Inherit options from project configuration, enable MISRA C++ rule checking, and run complete analysis.
- 'PrjConfigAndJSF' – Inherit options from project configuration, enable JSF rule checking, and run complete analysis.
- 'MisraCxx' – Enable MISRA C++ rule checking, and run compilation phase only.
- 'JSF' – Enable JSF rule checking, and run compilation phase only.

Only for Embedded Coder

Example: `opt.CxxVerificationSettings = 'MisraCxx';`

TargetLink Only

AutoStubLUT — Lookup Table code usage

false (default) | true

Lookup Table code usage, specified as true or false.

- `true` — use Lookup Table code during the analysis.
- `false` — stub Lookup Table code.

Only for `TargetLink`

Example: `opts.AutoStubLUT = true;`

See Also

`pslinkoptions` | `pslinkrun`

polyspace.Project.Configuration Properties

Customize Polyspace analysis of handwritten code with options object properties

Description

To customize your Polyspace analysis, use these `polyspace.Options` or `polyspace.Project.Configuration` properties. Each property corresponds to an analysis option on the **Configuration** pane in the Polyspace user interface.

The properties are grouped using the same categories as the **Configuration** pane. This page only shows what values each property can take. For details about:

- The different options, see the analysis option reference pages.
- How to create and use the object, see `polyspace.Options` or `polyspace.Project`.

The same properties are also available with the deprecated classes `polyspace.BugFinderOptions` and `polyspace.CodeProverOptions`.

Each property description below also highlights if the option affects only one of Bug Finder or Code Prover.

Note Some options might not be available depending on the language setting of the object. You can set the source code language (`Language`) to 'C', 'CPP' or 'C-CPP' during object creation, but cannot change it later.

Properties

Advanced

Additional — Additional flags for analysis

character vector

Additional flags for analysis specified as a character vector.

For more information, see `Other`.

Example: `opts.Advanced.Additional = '-extra-flags -option -extra-flags value'`

PostAnalysisCommand — Command or script software should execute after analysis finishes

character vector

Command or script software should execute after analysis finishes, specified as a character vector.

For more information, see `Command/script` to apply after the end of the code verification (`-post-analysis-command`).

Example: `opts.Advanced.PostAnalysisCommand = '"C:\Program Files\perl\win32\bin\perl.exe" "C:\My_Scripts\send_email"'`

AutomaticOrangeTester — Run the Automatic Orange Tester

false (default) | true

This property affects Code Prover analysis only.

Run the Automatic Orange Tester after verification, specified as true or false.

For more information, see `Automatic Orange Tester` (`-automatic-orange-tester`).

Example: `opts.Advanced.AutomaticOrangeTester = true`

AutomaticOrangeTesterLoopMaxIteration — Number of loop iterations after which Automatic Orange Tester considers infinite loop

1000 (default) | positive integer

This property affects Code Prover analysis only.

Number of loop iterations after which Automatic Orange Tester considers the test an infinite loop, specified as a positive integer, maximum of 1000.

For more information, see `Maximum loop iterations` (`-automatic-orange-tester-loop-max-iteration`).

Example: `opts.Advanced.AutomaticOrangeTesterLoopMaxIteration = 500`

AutomaticOrangeTesterTestsNumber — Number of tests that Automatic Orange Tester must run

500 (default) | positive integer

This property affects Code Prover analysis only.

Number of tests that Automatic Orange Tester must run, specified as a positive integer, maximum of 100,000.

For more information, see `Number of automatic tests (-automatic-orange-tester-tests-number)`.

```
Example: opts.Advanced.AutomaticOrangeTesterTestsNumber = 1000
```

AutomaticOrangeTesterTimeout — Time in seconds allowed for a single test in Automatic Orange Tester

5 (default) | positive integer

This property affects Code Prover analysis only.

Time in seconds allowed for a single test in Automatic Orange Tester, specified as a positive integer, maximum of 60.

For more information, see `Maximum test time (-automatic-orange-tester-timeout)`.

```
Example: opts.Advanced.AutomaticOrangeTesterTimeout = 10
```

BugFinderAnalysis (Affects Bug Finder Only)**CheckersList** — List of custom checkers to activate

name of defects options object | cell array of defect acronyms

This property affects Bug Finder analysis only.

List of custom checkers to activate specified by using the name of a `polyspace.DefectsOptions` object or a cell array of defect acronyms. To use this custom list in your analysis, set `CheckersPreset` to `custom`.

For more information, see `polyspace.DefectsOptions`.

```
Example: defects = polyspace.DefectsOptions;  
opts.BugFinderAnalysis.CheckersList = defects
```

```
Example: opts.BugFinderAnalysis.CheckersList =
{'INT_ZERO_DIV', 'FLOAT_ZERO_DIV'}
```

CheckersPreset — Subset of Bug Finder defects

default (default) | all | custom

This property affects Bug Finder analysis only.

Preset checker list, specified as a character vector of one the preset options: default, all, or custom. To use custom, specify a CheckersList.

For more information, see `Find defects (-checkers)`.

```
Example: opts.BugFinderAnalysis.CheckersPreset = 'all'
```

EnableCheckers — Activate defect checking

true (default) | false

This property affects Bug Finder analysis only.

Activate defect checking, specified as true or false. Setting this property to false disables all defects. If you want to disable defect checking but still get results, turn on coding rules checking or code metric checking.

This property is equivalent to the **Find defects** check box in the Polyspace interface.

```
Example: opts.BugFinderAnalysis.EnableCheckers = false
```

ChecksAssumption (Affects Code Prover Only)

AllowNegativeOperandInShift — Allow left shift operations on a negative number

false (default) | true

This property affects Code Prover analysis only.

Allow left shift operations on a negative number, specified as true or false.

For more information, see `Allow negative operand for left shifts (-allow-negative-operand-in-shift)`.

```
Example: opts.ChecksAssumption.AllowNegativeOperandInShift = true
```

AllowNonFiniteFloats — Incorporate infinities and/or NaNs

false (default) | true

This property affects Code Prover analysis only.

Incorporate infinities and/or NaNs, specified as true or false.

For more information, see Consider non finite floats (`-allow-non-finite-floats`).

Example: `opts.ChecksAssumption.AllowNonFiniteFloats = true`

AllowPtrArithOnStruct — Allow arithmetic on pointer to a structure field so that it points to another field

false (default) | true

This property affects Code Prover analysis only.

Allow arithmetic on pointer to a structure field so that it points to another field, specified as true or false.

For more information, see Enable pointer arithmetic across fields (`-allow-ptr-arith-on-struct`).

Example: `opts.ChecksAssumption.AllowPtrArithOnStruct = true`

CheckSubnormal — Detect operations that result in subnormal floating point values

allow (default) | warn-first | warn-all | forbid

This property affects Code Prover analysis only.

Detect operations that result in subnormal floating point values.

For more information, see Subnormal detection mode (`-check-subnormal`).

Example: `opts.ChecksAssumption.CheckSubnormal = 'forbid'`

DetectPointerEscape — Find cases where a function returns a pointer to one of its local variables

false (default) | true

This property affects Code Prover analysis only.

Find cases where a function returns a pointer to one of its local variables, specified as true or false.

For more information, see `Detect stack pointer dereference outside scope (-detect-pointer-escape)`.

Example: `opts.ChecksAssumption.DetectPointerEscape = true`

DisableInitializationChecks — Disable checks for noninitialized variables and pointers

false (default) | true

This property affects Code Prover analysis only.

Disable checks for noninitialized variables and pointers, specified as true or false.

For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`.

Example: `opts.ChecksAssumption.DisableInitializationChecks = true`

IgnoreConstantOverflows — Allow overflow in computations involving constants

false (default) | true

This property affects Code Prover analysis only.

Allow overflow in computations involving constants, specified as true or false.

For more information, see `Ignore overflowing computations on constants (-ignore-constant-overflows)`.

Example: `opts.ChecksAssumption.IgnoreConstantOverflows = true`

PermissiveFunctionPointer — Allow type mismatch between function pointers and the functions they point to

false (default) | true

This property affects Code Prover analysis only.

Allow type mismatch between function pointers and the functions they point to, specified as true or false.

For more information, see `Permissive function pointer calls (-permissive-function-pointer)`.

Example: `opts.ChecksAssumption.PermissiveFunctionPointer = true`

ScalarOverflowsBehavior — Behavior of scalar overflows

wrap-around (default) | truncate-on-error

This property affects Code Prover analysis only.

Behavior of scalar overflows, specified as wrap-around or truncate-on-error.

For more information, see Overflow computation mode (-scalar-overflows-behavior).

Example: `opts.ChecksAssumption.ScalarOverflowsBehavior = 'truncate-on-error'`**ScalarOverflowsChecks — Check for integer overflows on signed and unsigned variables**

signed (default) | signed-and-unsigned | none

This property affects Code Prover analysis only.

Check for integer overflows on signed and unsigned variables, specified as signed, signed-and-unsigned, or none.

For more information, see Detect overflows (-scalar-overflows-checks).

Example: `opts.ChecksAssumption.ScalarOverflowsChecks = 'signed-and-unsigned'`**SizeInBytes — Allow a pointer with insufficient memory buffer to point to a structure**

false (default) | true

This property affects Code Prover analysis only.

Allow a pointer with insufficient memory buffer to point to a structure, specified as true or false.

For more information, see Allow incomplete or partial allocation of structures (-size-in-bytes).

Example: `opts.ChecksAssumption.SizeInBytes = true`**UncalledFunctionCheck — Detect functions that are not called directly or indirectly from main or another entry-point function**

none (default) | never-called | called-from-unreachable | all

This property affects Code Prover analysis only.

Detect functions that are not called directly or indirectly from main or another entry-point function, specified as `none`, `never-called`, `called-from-unreachable`, or `all`.

For more information, see `Detect uncalled functions` (`-uncalled-function-checks`).

Example: `opts.ChecksAssumption.UncalledFunctionCheck = 'all'`

CodeProverVerification (Affects Code Prover only)

ClassAnalyzer — Classes that you want to verify

`all` (default) | `none` | cell array of class names

This property affects Code Prover analysis only.

Classes that you want to verify, specified as `all`, `none`, or a cell array of class names.

For more information, see `Class` (`-class-analyzer`).

Example: `opts.CodeProverVerification.ClassAnalyzer = 'none'`

ClassAnalyzerCalls — Class methods that you want to verify

`unused` (default) | `all` | `all-public` | `inherited-all` | `inherited-all-public` | `unused-public` | `inherited-unused` | `inherited-unused-public` | cell array of class methods

This property affects Code Prover analysis only.

Class methods that you want to verify, specified as one of the predefined sets or a cell array of class methods that you want to verify.

For more information, see `Functions to call within the specified classes` (`-class-analyzer-calls`).

Example: `opts.CodeProverVerification.ClassAnalyzerCalls = 'unused-public'`

ClassOnly — Analyze only class methods

`false` (default) | `true`

This property affects Code Prover analysis only.

Analyze only class methods, specified as true or false.

For more information, see `Analyze class contents only (-class-only)`.

Example: `opts.CodeProverVerification.ClassOnly = true`

EnableMain — Use main function provided in application

false (default) | true

This property affects Code Prover analysis only.

Use main function provided in application, specified as true or false. If you set this property to false, the analysis generates a main function, if it is not present in the source files.

For more information, see `Verify whole application`.

Example: `opts.CodeProverVerification.EnableMain = true`

FunctionsCalledBeforeMain — Functions that you want the generated main to call ahead of other functions

cell array of function names

This property affects Code Prover analysis only.

Functions that you want the generated main to call ahead of other functions, specified as a cell array of function names.

For more information, see `Initialization functions (-functions-called-before-main)`.

Example: `opts.CodeProverVerification.FunctionsCalledBeforeMain = {'func1', 'func2'}`

Main — Use a Microsoft Visual C++ extensions of main

`_tmain` (default) | `wmain` | `_tWinMain` | `wWinMain` | `WinMain` | `DllMain`

This property applies to a Code Prover analysis only.

Use a Microsoft Visual C++ extension of main, specified as one of the predefined main extensions.

For more information, see `Main entry point (-main)`.

Example: `opts.CodeProverVerification.Main = 'wmain'`

MainGenerator — Generate a main function if it is not present in source files

true (default) | false

This property applies to a Code Prover analysis only .

Generate a main function if it is not present in source files, specified as true or false.

For more information, see `Verify` module or library (`-main-generator`).

Example: `opts.CodeProverVerification.MainGenerator = false`

MainGeneratorCalls — Functions that you want the generated main to call after the initialization functions

unused (default) | none | all | cell array of function names

This property applies to a Code Prover analysis only .

Functions that you want the generated main to call after the initialization functions, specified as unused, all, none, or a cell array of function names.

For more information, see `Functions to call` (`-main-generator-calls`).

Example: `opts.CodeProverVerification.MainGeneratorCalls = 'all'`

MainGeneratorWriteVariables — Global variables that you want the generated main to initialize

uninit (C++ default) | public (C default) | none | all | cell array of global variable names

This property applies to a Code Prover analysis only .

Global variables that you want the generated main to initialize, specified as one of the predefined sets or a cell array of global variable names.

For more information, see `Variables to initialize` (`-main-generator-writes-variables`).

Example: `opts.CodeProverVerification.MainGeneratorWriteVariables = 'all'`

NoConstructorsInitCheck — Do not check if class constructor initializes class members

false (default) | true

This property applies to a Code Prover analysis only .

Do not check if class constructor initializes class members, specified as true or false.

For more information, see Skip member initialization check (-no-constructors-init-check).

Example: `opts.CodeProverVerification.NoConstructorsInitCheck = true`

UnitByUnit — Verify each source file independently of other source files

false (default) | true

This property affects Code Prover analysis only.

Verify each source file independently of other source files, specified as true or false.

For more information, see Verify files independently (-unit-by-unit).

Example: `opts.CodeProverVerification.UnitByUnit = true`

UnitByUnitCommonSource — Files that you want to include with each source file during a file-by-file verification

cell array of file paths

This property affects Code Prover analysis only.

Files that you want to include with each source file during a file-by-file verification, specified as a cell array of file paths.

For more information, see Common source files (-unit-by-unit-common-source).

Example: `opts.CodeProverVerification.UnitByUnitCommonSource = {'/inc/file1.h', '/inc/file2.h'}`

CodingRulesCodeMetrics

AcAgcSubset — Subset of MISRA AC AGC rules to check

OBL-rules (default) | OBL-REC-rules | all-rules | SQO-subset1 | SQO-subset2
| coding rules object | file

Subset of MISRA AC AGC rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA AC AGC (-misra-ac-agc)`.
- MISRA AC AGC coding rules object created with `polyspace.CodingRulesOptions('misraAcAgc')`.
- Full path to a file containing your MISRA AC AGC subset. You can create this file manually or in the Polyspace interface. See “Select Specific MISRA or JSF Coding Rules” (Polyspace Bug Finder).

To check MISRA AC AGC rules, also set `EnableAcAgc` to `true`.

```
Example: opts.CodingRulesCodeMetrics.AcAgcSubset = 'all-rules'
```

Data Types: `char`

AllowedPragmas — Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied

cell array of character vectors

Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied, specified as a cell array of character vectors. This property affects only MISRA C:2004 or MISRA AC AGC rule checking.

For more information, see `Allowed pragmas (-allowed-pragmas)`.

```
Example: opts.CodingRulesCodeMetrics.AllowedPragmas =  
{ 'pragma_01', 'pragma_02' }
```

Data Types: `cell`

BooleanTypes — Data types the coding rule checker must treat as effectively Boolean

cell array of character vectors

Data types that the coding rule checker must treat as effectively Boolean, specified as a cell array of character vectors.

For more information, see `Effective boolean types (-boolean-types)`.

```
Example: opts.CodingRulesCodeMetrics.BooleanTypes =  
{ 'boolean1_t', 'boolean2_t' }
```

Data Types: `cell`

CodeMetrics — Activate code metric calculations

false (default) | true

Activate code metric calculations, specified as true or false. If this property is turned off, Polyspace does not calculate code metrics even if you upload your results to Polyspace Metrics.

For more information about the code metrics, see `Calculate code metrics (-code-metrics)`.

Example: `opts.CodingRulesCodeMetrics.CodeMetrics = true`

CustomRulesSubset — Custom naming conventions to check against
custom coding rules file

Custom naming conventions to check against, specified as a custom coding rules file. You can create the custom coding rules file manually or in the Polyspace interface.

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.CustomRulesSubset = 'C:\ps_settings\coding_rules\custom_rules.txt'`

Data Types: char

EnableAcAgc — Check MISRA AC AGC rules
false (default) | true

Check MISRA AC AGC rules, specified as true or false. To customize which rules are checked, use `AcAgcSubset`.

For more information about the MISRA AC AGC checker, see `Check MISRA AC AGC (-misra-ac-agc)`.

Example: `opts.CodingRulesCodeMetrics.EnableAcAgc = true;`

EnableCustomRules — Check custom coding rules
false (default) | true

Check custom coding rules, specified as true or false. Use with `CustomRulesSubset`.

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCustomRules = true;`

EnableJsfc — Check JSF C++ rules
false (default) | true

Check JSF C++ rules, specified as true or false. To customize which rules are checked, use `JsfSubset`.

For more information, see `Check JSF C++ rules (-jsf-coding-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableJsf = true;`

EnableMisraC — Check MISRA C:2004 rules

false (default) | true

Check MISRA C:2004 rules, specified as true or false. To customize which rules are checked, use `MisraCSubset`.

For more information, see `Check MISRA C:2004 (-misra2)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC = true;`

EnableMisraC3 — Check MISRA C:2012 rules

false (default) | true

Check MISRA C:2012 rules, specified as true or false. To customize which rules are checked, use `MisraC3Subset`.

For more information about the MISRA C:2012 checker, see `Check MISRA C:2012 (-misra3)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC3 = true;`

EnableMisraCpp — Check MISRA C++:2008 rules

false (default) | true

Check MISRA C++:2008 rules, specified as true or false. To customize which rules are checked, use `MisraCppSubset`.

For more information about the MISRA C++:2008 checker, see `Check MISRA C++ rules (-misra-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraCpp = true;`

JsfSubset — Subset of JSF C++ rules to check

shall-rules (default) | shall-will-rules | all-rules | coding rules object | file

Subset of JSF C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check JSF C++ rules (-jsf-coding-rules)`.
- JSF C++ coding rules object created with `polyspace.CodingRulesOptions('jsf')`.
- Full path to a file containing your JSF C++ subset. You can create this file manually or from the Polyspace interface. See “Select Specific MISRA or JSF Coding Rules” (Polyspace Bug Finder).

To check JSF C++ rules, set `EnableJsf` to true.

Example: `opts.CodingRulesCodeMetrics.JsfSubset = 'all-rules'`

Data Types: char

Misra3AgcMode — Use the MISRA C:2012 categories for automatically generated code

false (default) | true

Use the MISRA C:2012 categories for automatically generated code, specified as true or false.

For more information, see `Use generated code requirements (-misra3-agc-mode)`.

Example: `opts.CodingRulesCodeMetrics.Misra3AgcMode = true;`

MisraC3Subset — Subset of MISRA C:2012 rules to check

mandatory-required (default) | mandatory | all | SQO-subset1 | SQO-subset2 | coding rules object | file

Subset of MISRA C:2012 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2012 (-misra3)`.
- MISRA C:2012 coding rules object created with `polyspace.CodingRulesOptions('misraC2012')`.
- Full path to a file containing your MISRA C:2012 subset. You can create the custom coding rules file manually or in the Polyspace interface. See “Select Specific MISRA or JSF Coding Rules” (Polyspace Bug Finder).

To check MISRA C:2012 rules, also set `EnableMisraC3` to true.

Example: `opts.CodingRulesCodeMetrics.MisraC3Subset = 'all'`

Data Types: char

MisraCSubset — Subset of MISRA C:2004 rules to check

`required-rules` (default) | `all-rules` | `SQO-subset1` | `SQO-subset2` | coding rules object | file

Subset of MISRA C:2004 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2004 (-misra2)`.
- MISRA C:2004 coding rules object created with `polyspace.CodingRulesOptions('misraC')`.
- Full path to a file containing your MISRA C:2004 subset. You can create the custom coding rules file manually or in the Polyspace interface. See “Select Specific MISRA or JSF Coding Rules” (Polyspace Bug Finder).

To check MISRA C:2004 rules, also set `EnableMisraC` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCSubset = 'all-rules'`

Data Types: char

MisraCppSubset — Subset of MISRA C++ rules

`required-rules` (default) | `all-rules` | `SQO-subset1` | `SQO-subset2` | coding rules object | file

Subset of MISRA C++:2008 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C++ rules (-misra-cpp)`.
- MISRA C++ coding rules object created with `polyspace.CodingRulesOptions('misraCpp')`.
- Full path to a file containing your MISRA C++ subset. You can create this file manually or from the Polyspace interface. See “Select Specific MISRA or JSF Coding Rules” (Polyspace Bug Finder).

To check MISRA C++ rules, set `EnableMisraCpp` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCppSubset = 'all-rules'`

Data Types: char

EnvironmentSettings**Dos** — Consider that file paths are in MS-DOS style

true (default) | false

Consider that file paths are in MS-DOS style, specified as true or false.

For more information, see `Code from DOS or Windows file system (-dos)`.

Example: `opts.EnvironmentSettings.Dos = true;`

IncludeFolders — Include folders needed for compilation

cell array of include folder paths

Include folders needed for compilation, specified as a cell array of the include folder paths.

To specify all subfolders of a folder, use folder path followed by `**`, for instance, `'C:\includes**'`. The notation follows the syntax of the `dir` function. See also “Specify Multiple Source Files”.

For more information, see `-I`.

Example: `opts.EnvironmentSettings.IncludeFolders = {'/includes','/com1/inc'};`

Example: `opts.EnvironmentSettings.IncludeFolders = {'C:\project1\common\includes'};`

Data Types: `cell`

Includes — Files to be #include-ed by each C file

cell array of files

Files to be #include-ed by each C source file in the analysis, specified by a cell array of files.

For more information, see `Include (-include)`.

Example: `opts.EnvironmentSettings.Includes = {'/inc/inc_file.h','/inc/inc_math.h'};`

NoExternC — Ignore linking errors inside extern blocks

false (default) | true

Ignore linking errors inside extern blocks, specified as true or false.

For more information, see `Ignore link errors (-no-extern-c)`.

Example: `opts.EnvironmentSettings.NoExternC = false;`

PostPreProcessingCommand — Command or script to run on source files after preprocessing

character vector

Command or script to run on source files after preprocessing, specified as a character vector of the command to run.

For more information, see `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

Example: Linux — `opts.EnvironmentSettings.PostPreProcessingCommand = 'pwd'/replace_keyword.pl'`

Example: Windows — `opts.EnvironmentSettings.PostPreProcessingCommand = '"C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\replace_keyword.pl"'`

StopWithCompileError — Stop analysis if a file does not compile

false (default) | true

Stop analysis if a file does not compile, specified as true or false.

For more information, see `Stop analysis if a file does not compile (-stop-if-compile-error)`.

Example: `opts.EnvironmentSettings.StopWithCompileError = true;`

InputsStubbing

DataRangeSpecifications — Constrain global variables, function inputs, and return values of stubbed functions

file path

Constrain global variables, function inputs, and return values of stubbed functions specified by the path to an XML constraint file. For more information about the constraint file, see “Specify External Constraints”.

For more information about this option, see `Constraint setup (-data-range-specifications)`.

```
Example: opts.InputsStubbing.DataRangeSpecifications = 'C:\project
\constraint_file.xml'
```

DoNotGenerateResultsFor — Files on which you do not want analysis results

`include-folders` (default) | `all-headers` | cell array of files or folders

Files on which you do not want analysis results, specified by `include-folders`, `all-headers`, or a character array beginning with `custom=` and containing a comma-separated file or folder names.

Use this option with `InputsStubbing.GenerateResultsFor`. For more information, see `Do not generate results for (-do-not-generate-results-for)`.

```
Example: opts.InputsStubbing.DoNotGenerateResultsFor = 'custom=C:
\project\file1.c,C:\project\file2.c'
```

GenerateResultsFor — Files on which you want analysis results

`source-headers` (default) | `all-headers` | character array

Files on which you do not want analysis results, specified by `source-headers`, `all-headers`, or a character array beginning with `custom=` and containing a comma-separated file or folder names.

Use this option with `InputsStubbing.DoNotGenerateResultsFor`. For more information, see `Generate results for sources and (-generate-results-for)`.

```
Example: opts.InputsStubbing.GenerateResultsFor = 'custom=C:\project
\includes_common_1,C:\project\includes_common_2'
```

FunctionsToStub — Functions to stub during analysis

cell array of function names

This property affects Code Prover analysis only.

Functions to stub during analysis, specified as a cell array of function names.

For more information, see `Functions to stub (-functions-to-stub)`.

```
Example: opts.InputsStubbing.FunctionsToStub = {'func1', 'func2'}
```

NoDefInitGlob — Consider global variables as uninitialized

false (default) | true

This property affects Code Prover analysis only.

Consider global variables as uninitialized, specified as true or false.

For more information, see `Ignore default initialization of global variables (-no-def-init-glob)`.

Example: `opts.InputsStubbing.NoDefInitGlob = true`

NoStlStubs — Do not use Polyspace implementations of functions in the Standard Template Library

false (default) | true

This property applies only to a Code Prover analysis of C++ code.

Do not use Polyspace implementations of functions in the Standard Template Library, specified as true or false.

For more information, see `No STL stubs (-no-stl-stubs)`.

Example: `opts.InputsStubbing.NoStlStubs = true`

StubECoderLookupTables — Specify that the analysis must stub functions in the generated code that use lookup tables

true (default) | false

This property applies only to a Code Prover analysis of code generated from models.

Specify that the analysis must stub functions in the generated code that use lookup tables. By replacing the functions with stubs, the analysis assumes more precise return values for the functions.

For more information, see `Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions)`.

Example: `opts.InputsStubbing.StubECoderLookupTables = true`

Macros

DefinedMacros — Macros to be replaced

cell array of macros

In preprocessed code, macros are replaced by the definition, specified in a cell array of macros and definitions. Specify the macro as `Macro=Value`. If you want Polyspace to ignore the macro, leave the `Value` blank. A macro with no equal sign replaces all instances of that macro by 1.

For more information, see `Preprocessor definitions (-D)`.

Example: `opts.Macros.DefinedMacros = {'uint32=int', 'name3=', 'var'}`

UndefinedMacros — Macros to undefine

cell array of macros

In preprocessed code, macros are undefined, specified by a cell array of macros to undefine.

For more information, see `Disabled preprocessor definitions (-U)`.

Example: `opts.Macros.DefinedMacros = {'name1', 'name2'}`

MergedComputingSettings

AddToResultsRepositoryBugFinder — Upload Bug Finder results to Polyspace Metrics web dashboard

false (default) | true

This property affects Bug Finder analysis only.

Upload Bug Finder analysis results to Polyspace Metrics web dashboard, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Upload results to Polyspace Metrics (-add-to-results-repository)`.

Example: `opts.MergedComputingSettings.AddToResultsRepositoryBugFinder = true;`

AddToResultsRepositoryCodeProver — Upload Code Prover results to Polyspace Metrics web dashboard

false (default) | true

This property affects Code Prover analysis only.

Upload Code Prover analysis results to Polyspace Metrics web dashboard, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Upload results to Polyspace Metrics (-add-to-results-repository)`.

Example: `opts.MergedComputingSettings.AddToResultsRepositoryCodeProver = true;`

BatchBugFinder — Send Bug Finder analysis to remote server

false (default) | true

This property affects Bug Finder analysis only.

Send Bug Finder analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`.

Example: `opts.MergedComputingSettings.BatchBugFinder = true;`

BatchCodeProver — Send Code Prover analysis to remote server

false (default) | true

This property affects Code Prover analysis only.

Send Code Prover analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`.

Example: `opts.MergedComputingSettings.BatchCodeProver = true;`

FastAnalysis — Run Bug Finder analysis using faster local mode

false (default) | true

This property affects Bug Finder analysis only.

Use fast analysis mode for Bug Finder analysis, specified as true or false.

For more information, see `Use fast analysis mode for Bug Finder (-fast-analysis)`.

Example: `opts.MergedComputingSettings.FastAnalysis = true;`

MergedReporting

EnableReportGeneration — Generate a report after the analysis

false (default) | true

After the analysis, generate a report, specified as true or false.

For more information, see `Generate report`.

Example: `opts.MergedReporting.EnableReportGeneration = true`

ReportOutputFormat — Output format of generated report

Word (default) | HTML | PDF

Output format of generated report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Output format (-report-output-format)`.

Example: `opts.MergedReporting.ReportOutputFormat = 'PDF'`

BugFinderReportTemplate — Template for generating Bug Finder analysis report

BugFinderSummary (default) | BugFinder | BugFinder_CWE | CodeMetrics | Metrics

This property affects a Bug Finder analysis only.

Template for generating analysis report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder` and `Code Prover report (-report-template)`.

Example: `opts.MergedReporting.BugFinderReportTemplate = 'CodeMetrics'`

CodeProverReportTemplate — Template for generating Code Prover analysis report

Developer (default) | CallHierarchy | CodeMetrics | CodingRules | Developer | DeveloperReview | Developer_withGreenChecks | Quality |

```
SoftwareQualityObjectives | SoftwareQualityObjectives_Summary |  
VariableAccess
```

This property affects a Code Prover analysis only.

Template for generating analysis report, specified as one of the predefined report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder` and `Code Prover report (-report-template)`.

```
Example: opts.MergedReporting.CodeProverReportTemplate = 'CodeMetrics'
```

Multitasking

CriticalSectionBegin — Functions that begin critical sections

cell array of critical section function names

Functions that begin critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionEnd`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

```
Example: opts.Multitasking.CriticalSectionBegin =  
{ 'function1:cs1', 'function2:cs2' }
```

CriticalSectionEnd — Functions that end critical sections

cell array of critical section function names

Functions that end critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionBegin`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

```
Example: opts.Multitasking.CriticalSectionEnd =  
{ 'function1:cs1', 'function2:cs2' }
```

CyclicTasks — Specify functions that represent cyclic tasks

cell array of function names

Specify functions that represent cyclic tasks.

To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Cyclic tasks (-cyclic-tasks)`.

Example: `opts.Multitasking.CyclicTasks = {'function1','function2'}`

EnableConcurrencyDetection — Enable automatic detection of certain families of threading functions

false (default) | true

This property affects Code Prover analysis only.

Enable automatic detection of certain families of threading functions, specified as true or false.

For more information, see `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

Example: `opts.Multitasking.EnableConcurrencyDetection = true`

EnableMultitasking — Configure multitasking manually

false (default) | true

Configure multitasking manually by specifying true. This property activates the other manual, multitasking properties.

For more information, see `Configure multitasking manually`.

Example: `opts.Multitasking.EnableMultitasking = 1`

EnableOsekMultitasking — Enable automatic multitasking configuration for OSEK project

false (default) | true

Enable multitasking configuration of your OSEK project from the OIL files you provide. Activate this option to enable `Multitasking.OsekMultitasking`.

For more information, see `OSEK multitasking configuration (-osek-multitasking)`

Example: `opts.Multitasking.EnableOsekMultitasking = 1`

EntryPoints — Functions that serve as entry-points to your multitasking application

cell array of entry-point function names

Functions that serve as entry-points to your multitasking application specified as a cell array of entry-point function names. To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Entry points (-entry-points)`.

Example: `opts.Multitasking.EntryPoints = {'function1','function2'}`

Interrupts — Specify functions that represent nonpreemptable interrupts

cell array of function names

Specify functions that represent nonpreemptable interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Interrupts (-interrupts)`.

Example: `opts.Multitasking.Interrupts = {'function1','function2'}`

InterruptsDisableAll — Specify routine that disable interrupts

cell array with one function name

This property affects Bug Finder analysis only.

Specify function that disables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsDisableAll = {'function'}`

InterruptsEnableAll — Specify routine that reenables interrupts

cell array with one function name

This property affects Bug Finder analysis only.

Specify function that reenables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsEnableAll = {'function'}`

OsekMultitasking — Specify path of OIL files to parse for multitasking configuration

`auto` | cell array of files

Specify the path to the OIL files the software parses to set up your multitasking configuration:

- In `auto` mode, the analysis uses OIL files in your project source and include folders, but not their subfolders.
- In `custom` mode, the analysis uses the OIL files at the specified path, and the path subfolders.

To activate this option, specify `MultitaskingEnableOsekMultitasking`.

For more information, see `OSEK multitasking configuration (-osek-multitasking)`

Example: `opts.Multitasking.OsekMultitasking = 'custom='file_path, dir_path'`

TemporalExclusion — Entry-point functions that cannot execute concurrently

cell array of entry-point function names

Entry-point functions that cannot execute concurrently specified as a cell array of entry-point function names. Each set of exclusive tasks is one cell array entry with functions separated by spaces. To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Temporally exclusive tasks (-temporal-exclusions-file)`.

Example: `opts.Multitasking.TemporalExclusion = {'function1 function2', 'function3 function4 function5'}` where `function1` and `function2` are temporally exclusive, and `function3`, `function4`, and `function 5` are temporally exclusive.

Precision (Affects Code Prover Only)

ContextSensitivity — Store call context information to identify function call that caused errors

none (default) | auto | cell array of function names

This property affects Code Prover analysis only.

Store call context information to identify a function call that caused errors, specified as none, auto, or a cell array of function names.

For more information, see `Sensitivity context` (-context-sensitivity).

Example: `opts.Precision.ContextSensitivity = 'auto'`

Example: `opts.Precision.ContextSensitivity = 'custom=func1'`

ModulesPrecision — Source files you want to verify at higher precision

cell array of file names and precision levels

This property affects Code Prover analysis only.

Source files that you want to verify at higher precision, specified as a cell array of file names without the extension and precision levels using this syntax: `filename:Olevel`

For more information, see `Specific precision` (-modules-precision).

Example: `opts.Precision.ModulesPrecision = {'file1:00', 'file2:03'}`

OLevel — Precision level for the verification

2 (default) | 0 | 1 | 3

This property affects Code Prover analysis only.

Precision level for the verification, specified as 0, 1, 2, or 3.

For more information, see `Precision level` (-O).

Example: `opts.Precision.OLevel = 3`

PathSensitivityDelta — Avoid certain verification approximations for code with fewer lines

positive integer

This property affects Code Prover analysis only.

Avoid certain verification approximations for code with fewer lines, specified as a positive integer representing how sensitive the analysis is. Higher values can increase verification time exponentially.

For more information, see `Improve precision of interprocedural analysis (-path-sensitivity-delta)`.

Example: `opts.Precision.PathSensitivityDelta = 2`

Timeout — Time limit on your verification

character vector

This property affects Code Prover analysis only.

Time limit on your verification, specified as a character vector of time in hours.

For more information, see `Verification time limit (-timeout)`.

Example: `opts.Precision.Timeout = '5.75'`

To — Number of times the verification process runs

Software Safety Analysis level 2 (default) | Software Safety Analysis level 0 | Software Safety Analysis level 1 | Software Safety Analysis level 3 | Software Safety Analysis level 4 | Source Compliance Checking | other

This property affects Code Prover analysis only.

Number of times the verification process runs, specified as one of the preset analysis levels.

For more information, see `Verification level (-to)`.

Example: `opts.Precision.To = 'Software Safety Analysis level 3'`

Scaling (Affects Code Prover Only)

Inline — Functions on which separate results must be generated for each function call

cell array of function names

This property affects Code Prover analysis only.

Functions on which separate results must be generated for each function call, specified as a cell array of function names.

For more information, see `Inline` (`-inline`).

Example: `opts.Scaling.Inline = {'func1','func2'}`

KLimiting — Limit depth of analysis for nested structures

positive integer

This property affects Code Prover analysis only.

Limit depth of analysis for nested structures, specified as a positive integer indicating how many levels into a nested structure to verify.

For more information, see `Depth of verification inside structures` (`-k-limiting`).

Example: `opts.Scaling.KLimiting = 3`

TargetCompiler

Compiler — Compiler that builds your source code

generic (default) | clang3.5 | gnu3.4 | gnu4.6 | gnu4.7 | gnu4.8 | gnu4.9 | iar
| iso | keil | visual9.0 | visual10 | visual11.0 | visual12.0 | visual14.0

Compiler that builds your source code.

For more information, see `Compiler` (`-compiler`).

Example: `opts.TargetCompiler.Compiler = 'Visual11.0'`

Cpp11Extension — Allow C++11 language extensions

false (default) | true

Allow C++11 language extensions, specified as true or false.

For more information, see `C++11 extensions` (`-cpp11-extension`).

Example: `opts.TargetCompiler.Cpp11Extension = true`

DivRoundDown — Round down quotients from division or modulus of negative numbers

false (default) | true

Round down quotients from division or modulus of negative numbers, specified as true or false.

For more information, see `Division round down (-div-round-down)`.

Example: `opts.TargetCompiler.DivRoundDown = true`

EnumTypeDefinition — Base type representation of enum

`defined-by-compiler (default) | auto-signed-first | auto-unsigned-first`

Base type representation of enum, specified by an allowed base-type set. For more information about the different values, see `Enum type definition (-enum-type-definition)`.

Example: `opts.TargetCompiler.EnumTypeDefinition = 'auto-unsigned-first'`

IgnorePragmaPack — Ignore #pragma pack directives

`false (default) | true`

Ignore `#pragma pack` directives, specified as true or false.

For more information, see `Ignore pragma pack directives (-ignore-pragma-pack)`.

Example: `opts.TargetCompiler.IgnorePragmaPack = true`

Language — Language of analysis

`C-CPP (default) | C | CPP`

This property is read-only.

Language of the analysis, specified during the object construction. This value changes which properties appear.

For more information, see `Source code language (-lang)`.

LogicalSignedRightShift — Treatment of signed bit on signed variables

`Arithmetical (default) | Logical`

Treatment of signed bit on signed variables, specified as `Arithmetical` or `Logical`. For more information, see `Signed right shift (-logical-signed-right-shift)`.

Example: `opts.TargetCompiler.LogicalSignedRightShift = 'Logical'`

NoLanguageExtensions — Restrict analysis to C language specified in ANSI C standard

false (default) | true

Restrict analysis to the C language that is specified in the ANSI C standard, specified as true or false. For more information, see `Respect C90 standard (-no-language-extensions)`.

Example: `opts.TargetCompiler.NoLanguageExtensions = true`

NoUliterals — Do not use predefined typedefs for `char16_t` or `char32_t`

false (default) | true

Do not use predefined typedefs for `char16_t` or `char32_t`, specified as true or false. For more information, see `Block char16/32_t types (-no-uliterals)`.

Example: `opts.TargetCompiler.NoUliterals = true`

PackAlignmentValue — Default structure packing alignment

8 (default) | 1 | 2 | 4 | 16

Default structure packing alignment, specified as 1, 2, 4, 8, or 16. This property is available only for Visual C++ code.

For more information, see `Pack alignment value (-pack-alignment-value)`.

Example: `opts.TargetCompiler.PackAlignmentValue = '4'`

SizeTTypeIs — Underlying type of `size_t`

defined-by-compiler (default) | unsigned-int | unsigned-long | unsigned-long-long

Underlying type of `size_t`, specified as `unsigned-int`, `unsigned-long` or `unsigned-long-long`. See `Management of size_t (-size-t-type-is)`.

Example: `opts.TargetCompiler.SizeTTypeIs = 'unsigned-long'`

wcharTTypeIs — Underlying type of `wchar_t`

defined-by-compiler (default) | signed-short | unsigned-short | signed-int | unsigned-int | signed-long | unsigned-long

Underlying type of `wchar_t`, specified as signed-short, unsigned-short, signed-int, unsigned-int, signed-long or unsigned-long. See Management of `wchar_t` (`-wchar-t-type-is`).

Example: `opts.TargetCompiler.WcharTTypeIs = 'unsigned-int'`

SfrTypes — sfr types

cell array of `sfr` keywords

`sfr` types, specified as a cell array of `sfr` keywords using the syntax `sfr_name=size_in_bits`. For more information, see Sfr type support (`-sfr-types`).

Example: `opts.TargetCompiler.SfrTypes = {'sfr32=32'}`

Target — Target processor

`i386` (default) | `sparc` | `m68k` | `powerpc` | `powerpc64` | `c-167` | `tms320c3x` | `sharc21x61` | `necv580` | `hc08` | `hc12` | `mpc5xx` | `c18` | `x86_64` | generic target object

Set size of data types and endianness of processor, specified as one of the predefined target processors or a generic target object.

For more information about the predefined processors, see Target processor type (`-target`).

For more information about creating a generic target, see `polyspace.GenericTargetOptions`.

Example: `opts.TargetCompiler.Target = 'hc12'`

VerificationAssumption (Affects Code Prover Only)

ConsiderVolatileQualifierOnFields — Assume that volatile qualified structure fields can have all possible values at any point in code

false (default) | true

This property affects Code Prover analysis only.

Assume that volatile qualified structure fields can have all possible values at any point in code.

For more information, see `Consider volatile qualifier on fields` (`-consider-volatile-qualifier-on-fields`).

Example: `opts.VerificationAssumption.ConsiderVolatileQualifierOnFields = true`

ConstraintPointersMaybeNull — Specify that environment pointers can be NULL unless constrained otherwise

false (default) | true

This property affects Code Prover analysis only.

Specify that environment pointers can be NULL unless constrained otherwise.

For more information, see `Consider environment pointers as unsafe` (`-stubbed-pointers-are-unsafe`).

Example: `opts.VerificationAssumption.ConstraintPointersMaybeNull = true`

FloatRoundingMode — Rounding modes to consider when determining the results of floating-point arithmetic

to-nearest (default) | all

This property affects Code Prover analysis only.

Rounding modes to consider when determining the results of floating-point arithmetic, specified as `to-nearest` or `all`.

For more information, see `Float rounding mode` (`-float-rounding-mode`).

Example: `opts.VerificationAssumption.FloatRoundingMode = 'all'`

RespectTypesInFields — Do not cast nonpointer fields of a structure to pointers

false (default) | true

This property affects Code Prover analysis only.

Do not cast nonpointer fields of a structure to pointers, specified as `true` or `false`.

For more information, see `Respect types in fields` (`-respect-types-in-fields`).

Example: `opts.VerificationAssumption.RespectTypesInFields = true`

RespectTypesInGlobals — Do not cast nonpointer global variables to pointers

false (default) | true

This property affects Code Prover analysis only.

Do not cast nonpointer global variables to pointers, specified as true or false.

For more information, see `Respect types in global variables (-respect-types-in-globals)`.

```
Example: opts.VerificationAssumption.RespectTypesInGlobals = true
```

Other Properties**Prog — Project name**

PolyspaceProject (default) | character vector

Project name, specified as a character vector.

For more information, see `-prog`.

```
Example: opts.Prog = 'myProject'
```

ResultsDir — Location to store results

folder path

Location to store results, specified as a folder path. By default, the results are stored in the current folder.

For more information, see `-results-dir`.

```
Example: opts.ResultsDir = 'C:\project\myproject\results\'
```

Sources — Source files

cell array of files

Source files to analyze, specified as a cell array of files.

To specify all files in a folder, use folder path followed by `*`, for instance, `'C:\src*'`. To specify all files in a folder and its subfolders, use folder path followed by `**`, for instance, `'C:\src**'`. The notation follows the syntax of the `dir` function. See also “Specify Multiple Source Files”.

For more information, see `-sources`.

```
Example: opts.Sources = {'file1.c', 'file2.c', 'file3.c'}
```

```
Example: opts.Sources = {'project/src1/file1.c', 'project/src2/  
file2.c', 'project/src3/file3.c'}
```

version — Project version number

1.0 (default) | character array of a number

Version number of project, specified as a character array of a number. This option is useful if you upload your results to Polyspace Metrics. If you increment version numbers each time that you reanalyze your object, you can compare the results from two versions in Polyspace Metrics.

For more information, see `-v[ersion]`.

```
Example: opts.Version = '2.3'
```

See Also

Topics

“Analysis Options”

Introduced in R2017a

polyspace.ModelLinkOptions Properties

Customize Polyspace analysis of generated code with options object properties

Description

To customize your Polyspace analysis of generated code, modify the `polyspace.ModelLinkOptions` object properties. Each property corresponds to an analysis option on the **Configuration** pane in the Polyspace user interface.

The properties are grouped using the same categories as the **Configuration** pane. This page only shows what values each property can take. For details about:

- The different options, see the analysis options reference pages.
- How to create and use the object, see `polyspace.ModelLinkOptions`.

The same properties are also available with the deprecated classes `polyspace.ModelLinkBugFinderOptions` and `polyspace.ModelLinkCodeProverOptions`.

Each property description below also highlights if the option affects only one of Bug Finder or Code Prover.

Note Some options might not be available depending on the language setting of the object. You can set the source code language (`Language`) to 'C', 'CPP' or 'C-CPP' during object creation, but cannot change it later.

Properties

Advanced

Additional — Additional flags for analysis

character vector

Additional flags for analysis specified as a character vector.

For more information, see `Other`.

Example: `opts.Advanced.Additional = '-extra-flags -option -extra-flags value'`

PostAnalysisCommand — Command or script software should execute after analysis finishes

character vector

Command or script software should execute after analysis finishes, specified as a character vector.

For more information, see `Command/script` to apply after the end of the code verification (`-post-analysis-command`).

Example: `opts.Advanced.PostAnalysisCommand = '"C:\Program Files\perl\win32\bin\perl.exe" "C:\My_Scripts\send_email"'`

AutomaticOrangeTester — Run the Automatic Orange Tester

false (default) | true

This property affects Code Prover analysis only.

Run the Automatic Orange Tester after verification, specified as true or false.

For more information, see `Automatic Orange Tester` (`-automatic-orange-tester`).

Example: `opts.Advanced.AutomaticOrangeTester = true`

AutomaticOrangeTesterLoopMaxIteration — Number of loop iterations after which Automatic Orange Tester considers infinite loop

1000 (default) | positive integer

This property affects Code Prover analysis only.

Number of loop iterations after which Automatic Orange Tester considers the test an infinite loop, specified as a positive integer, maximum of 1000.

For more information, see `Maximum loop iterations` (`-automatic-orange-tester-loop-max-iteration`).

Example: `opts.Advanced.AutomaticOrangeTesterLoopMaxIteration = 500`

AutomaticOrangeTesterTestsNumber — Number of tests that Automatic Orange Tester must run

500 (default) | positive integer

This property affects Code Prover analysis only.

Number of tests that Automatic Orange Tester must run, specified as a positive integer, maximum of 100,000.

For more information, see `Number of automatic tests (-automatic-orange-tester-tests-number)`.

Example: `opts.Advanced.AutomaticOrangeTesterTestsNumber = 1000`

AutomaticOrangeTesterTimeout — Time in seconds allowed for a single test in Automatic Orange Tester

5 (default) | positive integer

This property affects Code Prover analysis only.

Time in seconds allowed for a single test in Automatic Orange Tester, specified as a positive integer, maximum of 60.

For more information, see `Maximum test time (-automatic-orange-tester-timeout)`.

Example: `opts.Advanced.AutomaticOrangeTesterTimeout = 10`

BugFinderAnalysis (Affects Bug Finder Only)**CheckersList** — List of custom checkers to activate

name of defects options object | cell array of defect acronyms

This property affects Bug Finder analysis only.

List of custom checkers to activate specified by using the name of a `polyspace.DefectsOptions` object or a cell array of defect acronyms. To use this custom list in your analysis, set `CheckersPreset` to `custom`.

For more information, see `polyspace.DefectsOptions`.

Example: `defects = polyspace.DefectsOptions;`
`opts.BugFinderAnalysis.CheckersList = defects`

```
Example: opts.BugFinderAnalysis.CheckersList =
{'INT_ZERO_DIV', 'FLOAT_ZERO_DIV'}
```

CheckersPreset — Subset of Bug Finder defects

default (default) | all | custom

This property affects Bug Finder analysis only.

Preset checker list, specified as a character vector of one the preset options: default, all, or custom. To use custom, specify a CheckersList.

For more information, see `Find defects (-checkers)`.

```
Example: opts.BugFinderAnalysis.CheckersPreset = 'all'
```

EnableCheckers — Activate defect checking

true (default) | false

This property affects Bug Finder analysis only.

Activate defect checking, specified as true or false. Setting this property to false disables all defects. If you want to disable defect checking but still get results, turn on coding rules checking or code metric checking.

This property is equivalent to the **Find defects** check box in the Polyspace interface.

```
Example: opts.BugFinderAnalysis.EnableCheckers = false
```

ChecksAssumption (Affects Code Prover Only)

AllowNegativeOperandInShift — Allow left shift operations on a negative number

false (default) | true

This property affects Code Prover analysis only.

Allow left shift operations on a negative number, specified as true or false.

For more information, see `Allow negative operand for left shifts (-allow-negative-operand-in-shift)`.

```
Example: opts.ChecksAssumption.AllowNegativeOperandInShift = true
```

AllowNonFiniteFloats — Incorporate infinities and/or NaNs

false (default) | true

This property affects Code Prover analysis only.

Incorporate infinities and/or NaNs, specified as true or false.

For more information, see Consider non finite floats (`-allow-non-finite-floats`).

Example: `opts.ChecksAssumption.AllowNonFiniteFloats = true`

AllowPtrArithOnStruct — Allow arithmetic on pointer to a structure field so that it points to another field

false (default) | true

This property affects Code Prover analysis only.

Allow arithmetic on pointer to a structure field so that it points to another field, specified as true or false.

For more information, see Enable pointer arithmetic across fields (`-allow-ptr-arith-on-struct`).

Example: `opts.ChecksAssumption.AllowPtrArithOnStruct = true`

CheckSubnormal — Detect operations that result in subnormal floating point values

allow (default) | warn-first | warn-all | forbid

This property affects Code Prover analysis only.

Detect operations that result in subnormal floating point values.

For more information, see Subnormal detection mode (`-check-subnormal`).

Example: `opts.ChecksAssumption.CheckSubnormal = 'forbid'`

DetectPointerEscape — Find cases where a function returns a pointer to one of its local variables

false (default) | true

This property affects Code Prover analysis only.

Find cases where a function returns a pointer to one of its local variables, specified as true or false.

For more information, see `Detect stack pointer dereference outside scope` (`-detect-pointer-escape`).

Example: `opts.ChecksAssumption.DetectPointerEscape = true`

DisableInitializationChecks — Disable checks for noninitialized variables and pointers

false (default) | true

This property affects Code Prover analysis only.

Disable checks for noninitialized variables and pointers, specified as true or false.

For more information, see `Disable checks for non-initialization` (`-disable-initialization-checks`).

Example: `opts.ChecksAssumption.DisableInitializationChecks = true`

IgnoreConstantOverflows — Allow overflow in computations involving constants

false (default) | true

This property affects Code Prover analysis only.

Allow overflow in computations involving constants, specified as true or false.

For more information, see `Ignore overflowing computations on constants` (`-ignore-constant-overflows`).

Example: `opts.ChecksAssumption.IgnoreConstantOverflows = true`

PermissiveFunctionPointer — Allow type mismatch between function pointers and the functions they point to

false (default) | true

This property affects Code Prover analysis only.

Allow type mismatch between function pointers and the functions they point to, specified as true or false.

For more information, see `Permissive function pointer calls` (`-permissive-function-pointer`).

Example: `opts.ChecksAssumption.PermissiveFunctionPointer = true`

ScalarOverflowsBehavior — Behavior of scalar overflows

wrap-around (default) | truncate-on-error

This property affects Code Prover analysis only.

Behavior of scalar overflows, specified as wrap-around or truncate-on-error.

For more information, see `Overflow computation mode (-scalar-overflows-behavior)`.

Example: `opts.ChecksAssumption.ScalarOverflowsBehavior = 'truncate-on-error'`

ScalarOverflowsChecks — Check for integer overflows on signed and unsigned variables

signed (default) | signed-and-unsigned | none

This property affects Code Prover analysis only.

Check for integer overflows on signed and unsigned variables, specified as signed, signed-and-unsigned, or none.

For more information, see `Detect overflows (-scalar-overflows-checks)`.

Example: `opts.ChecksAssumption.ScalarOverflowsChecks = 'signed-and-unsigned'`

SizeInBytes — Allow a pointer with insufficient memory buffer to point to a structure

false (default) | true

This property affects Code Prover analysis only.

Allow a pointer with insufficient memory buffer to point to a structure, specified as true or false.

For more information, see `Allow incomplete or partial allocation of structures (-size-in-bytes)`.

Example: `opts.ChecksAssumption.SizeInBytes = true`

UncalledFunctionCheck — Detect functions that are not called directly or indirectly from main or another entry-point function

none (default) | never-called | called-from-unreachable | all

This property affects Code Prover analysis only.

Detect functions that are not called directly or indirectly from main or another entry-point function, specified as `none`, `never-called`, `called-from-unreachable`, or `all`.

For more information, see `Detect uncalled functions` (`-uncalled-function-checks`).

Example: `opts.ChecksAssumption.UncalledFunctionCheck = 'all'`

CodeProverVerification (Affects Code Prover only)

ClassAnalyzer — Classes that you want to verify

`all` (default) | `none` | cell array of class names

This property affects Code Prover analysis only.

Classes that you want to verify, specified as `all`, `none`, or a cell array of class names.

For more information, see `Class` (`-class-analyzer`).

Example: `opts.CodeProverVerification.ClassAnalyzer = 'none'`

FunctionsCalledAfterLoop — Functions that the generated main must call after the cyclic code loop

cell array of function names

This property affects Code Prover analysis only.

Functions that the generated main must call after the cyclic code loop, specified as a cell array of function names.

For more information, see `Termination functions` (`-functions-called-after-loop`).

Example: `opts.CodeProverVerification.FunctionsCalledAfterLoop = {'func1', 'func2'}`

FunctionsCalledBeforeLoop — Functions that the generated main must call before the cyclic code loop

cell array of function names

This property affects Code Prover analysis only.

Model Link only. Functions that the generated main must call before the cyclic code loop, specified as a cell array of function names.

For more information, see `Initialization functions` (`-functions-called-before-loop`).

Example: `opts.CodeProverVerification.FunctionsCalledBeforeLoop = {'func1','func2'}`

FunctionsCalledInLoop — Functions that the generated main must call in the cyclic code loop

`none` (default) | `all` | cell array of function names

This property affects Code Prover analysis only.

Functions that the generated main must call in the cyclic code loop, specified as `none`, `all`, or a cell array of function names.

For more information, see `Step functions` (`-functions-called-in-loop`).

Example: `opts.CodeProverVerification.FunctionsCalledInLoop = 'all'`

MainGenerator — Generate a main function if it is not present in source files

`true` (default) | `false`

This property affects Code Prover analysis only.

Generate a main function if it is not present in source files, specified as `true` or `false`.

For more information, see `Verify module or library` (`-main-generator`).

Example: `opts.CodeProverVerification.MainGenerator = false`

VariablesWrittenBeforeLoop — Variables that the generated main must initialize before the cyclic code loop

`none` (default) | `all` | cell array of variable names

This property affects Code Prover analysis only.

Variables that the generated main must initialize before the cyclic code loop, specified as `none`, `all`, or a cell array of variable names.

For more information, see `Parameters` (`-variables-written-before-loop`).

```
Example: opts.CodeProverVerification.VariablesWrittenBeforeLoop = 'all'
```

VariablesWrittenInLoop — Variables that the generated main must initialize in the cyclic code loop

none (default) | all | cell array of variable names

This property affects Code Prover analysis only.

Variables that the generated main must initialize in the cyclic code loop, specified as none, all, or a cell array of variable names.

For more information, see `Inputs` (-variables-written-in-loop).

```
Example: opts.CodeProverVerification.VariablesWrittenInLoop = 'all'
```

CodingRulesCodeMetrics

AcAgcSubset — Subset of MISRA AC AGC rules to check

OBL-rules (default) | OBL-REC-rules | all-rules | SQO-subset1 | SQO-subset2
| coding rules object | file

Subset of MISRA AC AGC rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA AC AGC` (-misra-ac-agc).
- MISRA AC AGC coding rules object created with `polyspace.CodingRulesOptions('misraAcAgc')`.
- Full path to a file containing your MISRA AC AGC subset. You can create this file manually or in the Polyspace interface. See “Select Specific MISRA or JSF Coding Rules” (Polyspace Bug Finder).

To check MISRA AC AGC rules, also set `EnableAcAgc` to true.

```
Example: opts.CodingRulesCodeMetrics.AcAgcSubset = 'all-rules'
```

Data Types: char

AllowedPragmas — Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied

cell array of character vectors

Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied, specified as a cell array of character vectors. This property affects only MISRA C:2004 or MISRA AC AGC rule checking.

For more information, see `Allowed pragmas` (`-allowed-pragmas`).

Example: `opts.CodingRulesCodeMetrics.AllowedPragmas = {'pragma_01', 'pragma_02'}`

Data Types: `cell`

BooleanTypes — Data types the coding rule checker must treat as effectively Boolean

cell array of character vectors

Data types that the coding rule checker must treat as effectively Boolean, specified as a cell array of character vectors.

For more information, see `Effective boolean types` (`-boolean-types`).

Example: `opts.CodingRulesCodeMetrics.BooleanTypes = {'boolean1_t', 'boolean2_t'}`

Data Types: `cell`

CodeMetrics — Activate code metric calculations

false (default) | true

Activate code metric calculations, specified as true or false. If this property is turned off, Polyspace does not calculate code metrics even if you upload your results to Polyspace Metrics.

For more information about the code metrics, see `Calculate code metrics` (`-code-metrics`).

Example: `opts.CodingRulesCodeMetrics.CodeMetrics = true`

CustomRulesSubset — Custom naming conventions to check against

custom coding rules file

Custom naming conventions to check against, specified as a custom coding rules file. You can create the custom coding rules file manually or in the Polyspace interface.

For more information, see `Check custom rules` (`-custom-rules`).

```
Example: opts.CodingRulesCodeMetrics.CustomRulesSubset = 'C:\ps_settings\coding_rules\custom_rules.txt'
```

Data Types: char

EnableAcAgc — Check MISRA AC AGC rules

false (default) | true

Check MISRA AC AGC rules, specified as true or false. To customize which rules are checked, use `AcAgcSubset`.

For more information about the MISRA AC AGC checker, see `Check MISRA AC AGC (-misra-ac-agc)`.

```
Example: opts.CodingRulesCodeMetrics.EnableAcAgc = true;
```

EnableCustomRules — Check custom coding rules

false (default) | true

Check custom coding rules, specified as true or false. Use with `CustomRulesSubset`.

For more information, see `Check custom rules (-custom-rules)`.

```
Example: opts.CodingRulesCodeMetrics.EnableCustomRules = true;
```

EnableJsf — Check JSF C++ rules

false (default) | true

Check JSF C++ rules, specified as true or false. To customize which rules are checked, use `JsfSubset`.

For more information, see `Check JSF C++ rules (-jsf-coding-rules)`.

```
Example: opts.CodingRulesCodeMetrics.EnableJsf = true;
```

EnableMisraC — Check MISRA C:2004 rules

false (default) | true

Check MISRA C:2004 rules, specified as true or false. To customize which rules are checked, use `MisraCSubset`.

For more information, see `Check MISRA C:2004 (-misra2)`.

```
Example: opts.CodingRulesCodeMetrics.EnableMisraC = true;
```

EnableMisraC3 — Check MISRA C:2012 rules

false (default) | true

Check MISRA C:2012 rules, specified as true or false. To customize which rules are checked, use `MisraC3Subset`.

For more information about the MISRA C:2012 checker, see `Check MISRA C:2012 (-misra3)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC3 = true;`

EnableMisraCpp — Check MISRA C++:2008 rules

false (default) | true

Check MISRA C++:2008 rules, specified as true or false. To customize which rules are checked, use `MisraCppSubset`.

For more information about the MISRA C++:2008 checker, see `Check MISRA C++ rules (-misra-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraCpp = true;`

JsfSubset — Subset of JSF C++ rules to check

shall-rules (default) | shall-will-rules | all-rules | coding rules object | file

Subset of JSF C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check JSF C++ rules (-jsf-coding-rules)`.
- JSF C++ coding rules object created with `polyspace.CodingRulesOptions('jsf')`.
- Full path to a file containing your JSF C++ subset. You can create this file manually or from the Polyspace interface. See “Select Specific MISRA or JSF Coding Rules” (Polyspace Bug Finder).

To check JSF C++ rules, set `EnableJsf` to true.

Example: `opts.CodingRulesCodeMetrics.JsfSubset = 'all-rules'`

Data Types: char

Misra3AgcMode — Use the MISRA C:2012 categories for automatically generated code

false (default) | true

Use the MISRA C:2012 categories for automatically generated code, specified as true or false.

For more information, see Use generated code requirements (`-misra3-agc-mode`).

Example: `opts.CodingRulesCodeMetrics.Misra3AgcMode = true;`

MisraC3Subset — Subset of MISRA C:2012 rules to check

`mandatory-required` (default) | `mandatory` | `all` | `SQO-subset1` | `SQO-subset2` | coding rules object | file

Subset of MISRA C:2012 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2012 (-misra3)`.
- MISRA C:2012 coding rules object created with `polyspace.CodingRulesOptions('misraC2012')`.
- Full path to a file containing your MISRA C:2012 subset. You can create the custom coding rules file manually or in the Polyspace interface. See “Select Specific MISRA or JSF Coding Rules” (Polyspace Bug Finder).

To check MISRA C:2012 rules, also set `EnableMisraC3` to true.

Example: `opts.CodingRulesCodeMetrics.MisraC3Subset = 'all'`

Data Types: `char`

MisraCSubset — Subset of MISRA C:2004 rules to check

`required-rules` (default) | `all-rules` | `SQO-subset1` | `SQO-subset2` | coding rules object | file

Subset of MISRA C:2004 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2004 (-misra2)`.
- MISRA C:2004 coding rules object created with `polyspace.CodingRulesOptions('misraC')`.
- Full path to a file containing your MISRA C:2004 subset. You can create the custom coding rules file manually or in the Polyspace interface. See “Select Specific MISRA or JSF Coding Rules” (Polyspace Bug Finder).

To check MISRA C:2004 rules, also set `EnableMisraC` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCSubset = 'all-rules'`

Data Types: char

MisraCppSubset — Subset of MISRA C++ rules

`required-rules` (default) | `all-rules` | `SQO-subset1` | `SQO-subset2` | coding rules object | file

Subset of MISRA C++:2008 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C++ rules (-misra-cpp)`.
- MISRA C++ coding rules object created with `polyspace.CodingRulesOptions('misraCpp')`.
- Full path to a file containing your MISRA C++ subset. You can create this file manually or from the Polyspace interface. See “Select Specific MISRA or JSF Coding Rules” (Polyspace Bug Finder).

To check MISRA C++ rules, set `EnableMisraCpp` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCppSubset = 'all-rules'`

Data Types: char

EnvironmentSettings

Dos — Consider that file paths are in MS-DOS style

`true` (default) | `false`

Consider that file paths are in MS-DOS style, specified as true or false.

For more information, see `Code from DOS or Windows file system (-dos)`.

Example: `opts.EnvironmentSettings.Dos = true;`

IncludeFolders — Include folders needed for compilation

cell array of include folder paths

Include folders needed for compilation, specified as a cell array of the include folder paths.

To specify all subfolders of a folder, use folder path followed by `**`, for instance, `'C:\includes**'`. The notation follows the syntax of the `dir` function. See also “Specify Multiple Source Files”.

For more information, see `-I`.

```
Example: opts.EnvironmentSettings.IncludeFolders = {'/includes','/com1/inc'};
```

```
Example: opts.EnvironmentSettings.IncludeFolders = {'C:\project1\common\includes'};
```

Data Types: `cell`

Includes — Files to be #include-ed by each C file

cell array of files

Files to be #include-ed by each C source file in the analysis, specified by a cell array of files.

For more information, see `Include (-include)`.

```
Example: opts.EnvironmentSettings.Includes = {'/inc/inc_file.h','/inc/inc_math.h'}
```

NoExternC — Ignore linking errors inside extern blocks

false (default) | true

Ignore linking errors inside extern blocks, specified as true or false.

For more information, see `Ignore link errors (-no-extern-c)`.

```
Example: opts.EnvironmentSettings.NoExternC = false;
```

PostPreProcessingCommand — Command or script to run on source files after preprocessing

character vector

Command or script to run on source files after preprocessing, specified as a character vector of the command to run.

For more information, see `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

Example: Linux — `opts.EnvironmentSettings.PostPreProcessingCommand = 'pwd'/replace_keyword.pl'`

Example: Windows — `opts.EnvironmentSettings.PostPreProcessingCommand = '"C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\replace_keyword.pl"'`

StopWithCompileError — Stop analysis if a file does not compile

false (default) | true

Stop analysis if a file does not compile, specified as true or false.

For more information, see Stop analysis if a file does not compile (`-stop-if-compile-error`).

Example: `opts.EnvironmentSettings.StopWithCompileError = true;`

InputsStubbing

DataRangeSpecifications — Constrain global variables, function inputs, and return values of stubbed functions

file path

Constrain global variables, function inputs, and return values of stubbed functions specified by the path to an XML constraint file. For more information about the constraint file, see “Specify External Constraints”.

For more information about this option, see Constraint setup (`-data-range-specifications`).

Example: `opts.InputsStubbing.DataRangeSpecifications = 'C:\project\constraint_file.xml'`

DoNotGenerateResultsFor — Files on which you do not want analysis results

include-folders (default) | all-headers | cell array of files or folders

Files on which you do not want analysis results, specified by `include-folders`, `all-headers`, or a character array beginning with `custom=` and containing a comma-separated file or folder names.

Use this option with `InputsStubbing.GenerateResultsFor`. For more information, see Do not generate results for (`-do-not-generate-results-for`).

```
Example: opts.InputsStubbing.DoNotGenerateResultsFor = 'custom=C:\project\file1.c,C:\project\file2.c'
```

GenerateResultsFor — Files on which you want analysis results

source-headers (default) | all-headers | character array

Files on which you do not want analysis results, specified by source-headers, all-headers, or a character array beginning with custom= and containing a comma-separated file or folder names.

Use this option with `InputsStubbing.DoNotGenerateResultsFor`. For more information, see `Generate results for sources and (-generate-results-for)`.

```
Example: opts.InputsStubbing.GenerateResultsFor = 'custom=C:\project\includes_common_1,C:\project\includes_common_2'
```

FunctionsToStub — Functions to stub during analysis

cell array of function names

This property affects Code Prover analysis only.

Functions to stub during analysis, specified as a cell array of function names.

For more information, see `Functions to stub (-functions-to-stub)`.

```
Example: opts.InputsStubbing.FunctionsToStub = {'func1', 'func2'}
```

NoDefInitGlob — Consider global variables as uninitialized

false (default) | true

This property affects Code Prover analysis only.

Consider global variables as uninitialized, specified as true or false.

For more information, see `Ignore default initialization of global variables (-no-def-init-glob)`.

```
Example: opts.InputsStubbing.NoDefInitGlob = true
```

NoStlStubs — Do not use Polyspace implementations of functions in the Standard Template Library

false (default) | true

This property applies only to a Code Prover analysis of C++ code.

Do not use Polyspace implementations of functions in the Standard Template Library, specified as true or false.

For more information, see `No STL stubs (-no-stl-stubs)`.

Example: `opts.InputsStubbing.NoStlStubs = true`

StubECoderLookupTables — Specify that the analysis must stub functions in the generated code that use lookup tables

true (default) | false

This property applies only to a Code Prover analysis of code generated from models.

Specify that the analysis must stub functions in the generated code that use lookup tables. By replacing the functions with stubs, the analysis assumes more precise return values for the functions.

For more information, see `Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions)`.

Example: `opts.InputsStubbing.StubECoderLookupTables = true`

Macros

DefinedMacros — Macros to be replaced

cell array of macros

In preprocessed code, macros are replaced by the definition, specified in a cell array of macros and definitions. Specify the macro as `Macro=Value`. If you want Polyspace to ignore the macro, leave the `Value` blank. A macro with no equal sign replaces all instances of that macro by 1.

For more information, see `Preprocessor definitions (-D)`.

Example: `opts.Macros.DefinedMacros = {'uint32=int', 'name3=', 'var'}`

UndefinedMacros — Macros to undefine

cell array of macros

In preprocessed code, macros are undefined, specified by a cell array of macros to undefine.

For more information, see `Disabled preprocessor definitions (-U)`.

Example: `opts.Macros.DefinedMacros = {'name1', 'name2'}`

MergedComputingSettings

AddToResultsRepositoryBugFinder — Upload Bug Finder results to Polyspace Metrics web dashboard

false (default) | true

This property affects Bug Finder analysis only.

Upload Bug Finder analysis results to Polyspace Metrics web dashboard, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Upload results to Polyspace Metrics (-add-to-results-repository)`.

Example: `opts.MergedComputingSettings.AddToResultsRepositoryBugFinder = true;`

AddToResultsRepositoryCodeProver — Upload Code Prover results to Polyspace Metrics web dashboard

false (default) | true

This property affects Code Prover analysis only.

Upload Code Prover analysis results to Polyspace Metrics web dashboard, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Upload results to Polyspace Metrics (-add-to-results-repository)`.

Example: `opts.MergedComputingSettings.AddToResultsRepositoryCodeProver = true;`

BatchBugFinder — Send Bug Finder analysis to remote server

false (default) | true

This property affects Bug Finder analysis only.

Send Bug Finder analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Run Bug Finder` or `Code Prover` analysis on a remote cluster (`-batch`).

Example: `opts.MergedComputingSettings.BatchBugFinder = true;`

BatchCodeProver — Send Code Prover analysis to remote server

false (default) | true

This property affects Code Prover analysis only.

Send Code Prover analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Run Bug Finder` or `Code Prover` analysis on a remote cluster (`-batch`).

Example: `opts.MergedComputingSettings.BatchCodeProver = true;`

FastAnalysis — Run Bug Finder analysis using faster local mode

false (default) | true

This property affects Bug Finder analysis only.

Use fast analysis mode for Bug Finder analysis, specified as true or false.

For more information, see `Use fast analysis mode for Bug Finder` (`-fast-analysis`).

Example: `opts.MergedComputingSettings.FastAnalysis = true;`

MergedReporting

EnableReportGeneration — Generate a report after the analysis

false (default) | true

After the analysis, generate a report, specified as true or false.

For more information, see `Generate report`.

Example: `opts.MergedReporting.EnableReportGeneration = true`

ReportOutputFormat — Output format of generated report

Word (default) | HTML | PDF

Output format of generated report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Output format (-report-output-format)`.

Example: `opts.MergedReporting.ReportOutputFormat = 'PDF'`

BugFinderReportTemplate — Template for generating Bug Finder analysis report

BugFinderSummary (default) | BugFinder | BugFinder_CWE | CodeMetrics | Metrics

This property affects a Bug Finder analysis only.

Template for generating analysis report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder` and `Code Prover report (-report-template)`.

Example: `opts.MergedReporting.BugFinderReportTemplate = 'CodeMetrics'`

CodeProverReportTemplate — Template for generating Code Prover analysis report

Developer (default) | CallHierarchy | CodeMetrics | CodingRules | Developer | DeveloperReview | Developer_withGreenChecks | Quality | SoftwareQualityObjectives | SoftwareQualityObjectives_Summary | VariableAccess

This property affects a Code Prover analysis only.

Template for generating analysis report, specified as one of the predefined report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder` and `Code Prover report (-report-template)`.

Example: `opts.MergedReporting.CodeProverReportTemplate = 'CodeMetrics'`

Multitasking

CriticalSectionBegin — Functions that begin critical sections

cell array of critical section function names

Functions that begin critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionEnd`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionBegin = {'function1:cs1','function2:cs2'}`

CriticalSectionEnd — Functions that end critical sections

cell array of critical section function names

Functions that end critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionBegin`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionEnd = {'function1:cs1','function2:cs2'}`

CyclicTasks — Specify functions that represent cyclic tasks

cell array of function names

Specify functions that represent cyclic tasks.

To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Cyclic tasks (-cyclic-tasks)`.

Example: `opts.Multitasking.CyclicTasks = {'function1','function2'}`

EnableConcurrencyDetection — Enable automatic detection of certain families of threading functions

false (default) | true

This property affects Code Prover analysis only.

Enable automatic detection of certain families of threading functions, specified as true or false.

For more information, see `Enable automatic concurrency detection for Code Prover` (`-enable-concurrency-detection`).

Example: `opts.Multitasking.EnableConcurrencyDetection = true`

EnableMultitasking — Configure multitasking manually

false (default) | true

Configure multitasking manually by specifying true. This property activates the other manual, multitasking properties.

For more information, see `Configure multitasking manually`.

Example: `opts.Multitasking.EnableMultitasking = 1`

EnableOsekMultitasking — Enable automatic multitasking configuration for OSEK project

false (default) | true

Enable multitasking configuration of your OSEK project from the OIL files you provide. Activate this option to enable `Multitasking.OsekMultitasking`.

For more information, see `OSEK multitasking configuration` (`-osek-multitasking`)

Example: `opts.Multitasking.EnableOsekMultitasking = 1`

EntryPoints — Functions that serve as entry-points to your multitasking application

cell array of entry-point function names

Functions that serve as entry-points to your multitasking application specified as a cell array of entry-point function names. To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Entry points` (`-entry-points`).

Example: `opts.Multitasking.EntryPoints = {'function1','function2'}`

Interrupts — Specify functions that represent nonpreemptable interrupts

cell array of function names

Specify functions that represent nonpreemptable interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Interrupts` (`-interrupts`).

Example: `opts.Multitasking.Interrupts = {'function1','function2'}`

InterruptsDisableAll — Specify routine that disable interrupts

cell array with one function name

This property affects Bug Finder analysis only.

Specify function that disables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts` (`-routine-disable-interrupts -routine-enable-interrupts`).

Example: `opts.Multitasking.InterruptsDisableAll = {'function'}`

InterruptsEnableAll — Specify routine that reenables interrupts

cell array with one function name

This property affects Bug Finder analysis only.

Specify function that reenables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts` (`-routine-disable-interrupts -routine-enable-interrupts`).

Example: `opts.Multitasking.InterruptsEnableAll = {'function'}`

osekMultitasking — Specify path of OIL files to parse for multitasking configuration

auto | cell array of files

Specify the path to the OIL files the software parses to set up your multitasking configuration:

- In `auto` mode, the analysis uses OIL files in your project source and include folders, but not their subfolders.
- In `custom` mode, the analysis uses the OIL files at the specified path, and the path subfolders.

To activate this option, specify `MultitaskingEnableOsekMultitasking`.

For more information, see OSEK multitasking configuration (`-osek-multitasking`)

```
Example: opts.Multitasking.OsekMultitasking = 'custom='file_path,
dir_path'
```

TemporalExclusion — Entry-point functions that cannot execute concurrently
cell array of entry-point function names

Entry-point functions that cannot execute concurrently specified as a cell array of entry-point function names. Each set of exclusive tasks is one cell array entry with functions separated by spaces. To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see Temporally exclusive tasks (`-temporal-exclusions-file`).

```
Example: opts.Multitasking.TemporalExclusion = {'function1 function2',
'function3 function4 function5'} where function1 and function2 are temporally
exclusive, and function3, function4, and function 5 are temporally exclusive.
```

Precision (Affects Code Prover Only)

ContextSensitivity — Store call context information to identify function call that caused errors

`none` (default) | `auto` | cell array of function names

This property affects Code Prover analysis only.

Store call context information to identify a function call that caused errors, specified as `none`, `auto`, or a cell array of function names.

For more information, see `Sensitivity context` (`-context-sensitivity`).

```
Example: opts.Precision.ContextSensitivity = 'auto'
```


Example: `opts.Precision.ContextSensitivity = 'custom=func1'`

ModulesPrecision — Source files you want to verify at higher precision

cell array of file names and precision levels

This property affects Code Prover analysis only.

Source files that you want to verify at higher precision, specified as a cell array of file names without the extension and precision levels using this syntax: `filename:Olevel`

For more information, see `Specific precision (-modules-precision)`.

Example: `opts.Precision.ModulesPrecision = {'file1:00', 'file2:03'}`

OLevel — Precision level for the verification

2 (default) | 0 | 1 | 3

This property affects Code Prover analysis only.

Precision level for the verification, specified as 0, 1, 2, or 3.

For more information, see `Precision level (-O)`.

Example: `opts.Precision.OLevel = 3`

PathSensitivityDelta — Avoid certain verification approximations for code with fewer lines

positive integer

This property affects Code Prover analysis only.

Avoid certain verification approximations for code with fewer lines, specified as a positive integer representing how sensitive the analysis is. Higher values can increase verification time exponentially.

For more information, see `Improve precision of interprocedural analysis (-path-sensitivity-delta)`.

Example: `opts.Precision.PathSensitivityDelta = 2`

Timeout — Time limit on your verification

character vector

This property affects Code Prover analysis only.

Time limit on your verification, specified as a character vector of time in hours.

For more information, see `Verification time limit (-timeout)`.

Example: `opts.Precision.Timeout = '5.75'`

to — Number of times the verification process runs

Software Safety Analysis level 2 (default) | Software Safety Analysis level 0 | Software Safety Analysis level 1 | Software Safety Analysis level 3 | Software Safety Analysis level 4 | Source Compliance Checking | other

This property affects Code Prover analysis only.

Number of times the verification process runs, specified as one of the preset analysis levels.

For more information, see `Verification level (-to)`.

Example: `opts.Precision.To = 'Software Safety Analysis level 3'`

Scaling (Affects Code Prover Only)

inline — Functions on which separate results must be generated for each function call
cell array of function names

This property affects Code Prover analysis only.

Functions on which separate results must be generated for each function call, specified as a cell array of function names.

For more information, see `Inline (-inline)`.

Example: `opts.Scaling.Inline = {'func1', 'func2'}`

κLimiting — Limit depth of analysis for nested structures
positive integer

This property affects Code Prover analysis only.

Limit depth of analysis for nested structures, specified as a positive integer indicating how many levels into a nested structure to verify.

For more information, see `Depth of verification inside structures (-k-limiting)`.

Example: `opts.Scaling.KLimiting = 3`

TargetCompiler

Compiler — Compiler that builds your source code

`generic (default) | clang3.5 | gnu3.4 | gnu4.6 | gnu4.7 | gnu4.8 | gnu4.9 | iar | iso | keil | visual9.0 | visual10 | visual11.0 | visual12.0 | visual14.0`

Compiler that builds your source code.

For more information, see `Compiler (-compiler)`.

Example: `opts.TargetCompiler.Compiler = 'Visual11.0'`

Cplusplus11Extension — Allow C++11 language extensions

`false (default) | true`

Allow C++11 language extensions, specified as true or false.

For more information, see `C++11 extensions (-cplusplus11-extension)`.

Example: `opts.TargetCompiler.Cplusplus11Extension = true`

DivRoundDown — Round down quotients from division or modulus of negative numbers

`false (default) | true`

Round down quotients from division or modulus of negative numbers, specified as true or false.

For more information, see `Division round down (-div-round-down)`.

Example: `opts.TargetCompiler.DivRoundDown = true`

EnumTypeDefinition — Base type representation of enum

`defined-by-compiler (default) | auto-signed-first | auto-unsigned-first`

Base type representation of enum, specified by an allowed base-type set. For more information about the different values, see `Enum type definition (-enum-type-definition)`.

Example: `opts.TargetCompiler.EnumTypeDefinition = 'auto-unsigned-first'`

IgnorePragmaPack — Ignore #pragma pack directives

false (default) | true

Ignore #pragma pack directives, specified as true or false.

For more information, see Ignore pragma pack directives (`-ignore-pragma-pack`).

Example: `opts.TargetCompiler.IgnorePragmaPack = true`

Language — Language of analysis

C-CPP (default) | C | CPP

This property is read-only.

Language of the analysis, specified during the object construction. This value changes which properties appear.

For more information, see Source code language (`-lang`).

LogicalSignedRightShift — Treatment of signed bit on signed variables

Arithmetical (default) | Logical

Treatment of signed bit on signed variables, specified as Arithmetical or Logical. For more information, see Signed right shift (`-logical-signed-right-shift`).

Example: `opts.TargetCompiler.LogicalSignedRightShift = 'Logical'`

NoLanguageExtensions — Restrict analysis to C language specified in ANSI C standard

false (default) | true

Restrict analysis to the C language that is specified in the ANSI C standard, specified as true or false. For more information, see Respect C90 standard (`-no-language-extensions`).

Example: `opts.TargetCompiler.NoLanguageExtensions = true`

NoUliterals — Do not use predefined typedefs for `char16_t` or `char32_t`

false (default) | true

Do not use predefined typedefs for `char16_t` or `char32_t`, specified as `true` or `false`. For more information, see `Block char16/32_t types (-no-uliterals)`.

Example: `opts.TargetCompiler.NoUliterals = true`

PackAlignmentValue — Default structure packing alignment

8 (default) | 1 | 2 | 4 | 16

Default structure packing alignment, specified as 1, 2, 4, 8, or 16. This property is available only for Visual C++ code.

For more information, see `Pack alignment value (-pack-alignment-value)`.

Example: `opts.TargetCompiler.PackAlignmentValue = '4'`

SizeTTypeIs — Underlying type of `size_t`

defined-by-compiler (default) | unsigned-int | unsigned-long | unsigned-long-long

Underlying type of `size_t`, specified as `unsigned-int`, `unsigned-long` or `unsigned-long-long`. See `Management of size_t (-size-t-type-is)`.

Example: `opts.TargetCompiler.SizeTTypeIs = 'unsigned-long'`

WcharTTypeIs — Underlying type of `wchar_t`

defined-by-compiler (default) | signed-short | unsigned-short | signed-int | unsigned-int | signed-long | unsigned-long

Underlying type of `wchar_t`, specified as `signed-short`, `unsigned-short`, `signed-int`, `unsigned-int`, `signed-long` or `unsigned-long`. See `Management of wchar_t (-wchar-t-type-is)`.

Example: `opts.TargetCompiler.WcharTTypeIs = 'unsigned-int'`

SfrTypes — sfr types

cell array of `sfr` keywords

`sfr` types, specified as a cell array of `sfr` keywords using the syntax `sfr_name=size_in_bits`. For more information, see `Sfr type support (-sfr-types)`.

Example: `opts.TargetCompiler.SfrTypes = {'sfr32=32'}`

Target — Target processor

i386 (default) | sparc | m68k | powerpc | powerpc64 | c-167 | tms320c3x | sharc21x61 | necv580 | hc08 | hc12 | mpc5xx | c18 | x86_64 | generic target object

Set size of data types and endianness of processor, specified as one of the predefined target processors or a generic target object.

For more information about the predefined processors, see `Target processor type` (`-target`).

For more information about creating a generic target, see `polyspace.GenericTargetOptions`.

Example: `opts.TargetCompiler.Target = 'hc12'`

VerificationAssumption (Affects Code Prover Only)

ConsiderVolatileQualifierOnFields — Assume that volatile qualified structure fields can have all possible values at any point in code

false (default) | true

This property affects Code Prover analysis only.

Assume that volatile qualified structure fields can have all possible values at any point in code.

For more information, see `Consider volatile qualifier on fields` (`-consider-volatile-qualifier-on-fields`).

Example: `opts.VerificationAssumption.ConsiderVolatileQualifierOnFields = true`

ConstraintPointersMayBeNull — Specify that environment pointers can be NULL unless constrained otherwise

false (default) | true

This property affects Code Prover analysis only.

Specify that environment pointers can be NULL unless constrained otherwise.

For more information, see `Consider environment pointers as unsafe` (`-stubbed-pointers-are-unsafe`).

Example: `opts.VerificationAssumption.ConstraintPointersMayBeNull = true`

FloatRoundingMode — Rounding modes to consider when determining the results of floating-point arithmetic

`to-nearest` (default) | `all`

This property affects Code Prover analysis only.

Rounding modes to consider when determining the results of floating-point arithmetic, specified as `to-nearest` or `all`.

For more information, see `Float` rounding mode (`-float-rounding-mode`).

Example: `opts.VerificationAssumption.FloatRoundingMode = 'all'`

RespectTypesInFields — Do not cast nonpointer fields of a structure to pointers

`false` (default) | `true`

This property affects Code Prover analysis only.

Do not cast nonpointer fields of a structure to pointers, specified as `true` or `false`.

For more information, see `Respect types in fields` (`-respect-types-in-fields`).

Example: `opts.VerificationAssumption.RespectTypesInFields = true`

RespectTypesInGlobals — Do not cast nonpointer global variables to pointers

`false` (default) | `true`

This property affects Code Prover analysis only.

Do not cast nonpointer global variables to pointers, specified as `true` or `false`.

For more information, see `Respect types in global variables` (`-respect-types-in-globals`).

Example: `opts.VerificationAssumption.RespectTypesInGlobals = true`

Other Properties

Prog — Project name

`PolyspaceProject` (default) | character vector

Project name, specified as a character vector.

For more information, see `-prog`.

Example: `opts.Prog = 'myProject'`

ResultsDir — Location to store results

folder path

Location to store results, specified as a folder path. By default, the results are stored in the current folder.

For more information, see `-results-dir`.

Example: `opts.ResultsDir = 'C:\project\myproject\results\'`

Sources — Source files

cell array of files

Source files to analyze, specified as a cell array of files.

To specify all files in a folder, use folder path followed by `*`, for instance, `'C:\src*'`. To specify all files in a folder and its subfolders, use folder path followed by `**`, for instance, `'C:\src**'`. The notation follows the syntax of the `dir` function. See also “Specify Multiple Source Files”.

For more information, see `-sources`.

Example: `opts.Sources = {'file1.c', 'file2.c', 'file3.c'}`

Example: `opts.Sources = {'project/src1/file1.c', 'project/src2/file2.c', 'project/src3/file3.c'}`

Version — Project version number

1.0 (default) | character array of a number

Version number of project, specified as a character array of a number. This option is useful if you upload your results to Polyspace Metrics. If you increment version numbers each time that you reanalyze your object, you can compare the results from two versions in Polyspace Metrics.

For more information, see `-v[ersion]`.

Example: `opts.Version = '2.3'`

See Also

Topics

“Analysis Options”

Introduced in R2017a

MISRA C 2012

MISRA C:2012 Dir 1.1

Any implementation-defined behavior on which the output of the program depends shall be documented and understood

Description

Directive Definition

Any implementation-defined behavior on which the output of the program depends shall be documented and understood.

Rationale

A code construct has implementation-defined behavior if the C standard allows compilers to choose their own specifications for the construct. The full list of implementation-defined behavior is available in Annex J.3 of the standard ISO/IEC 9899:1999 (C99) and in Annex G.3 of the standard ISO/IEC 9899:1990 (C90).

If you understand and document all implementation-defined behavior, you can be assured that all output of your program is intentional and not produced by chance.

Polyspace Specification

The analysis detects the following possibilities of implementation-defined behavior in C99 and their counterparts in C90. If you know the behavior of your compiler implementation, justify the analysis result with appropriate comments. To justify a result, assign one of these statuses: `Justified`, `No action planned`, or `Not a defect`.

Tip To mass-justify all results that indicate the same implementation-defined behavior, use the **Detail** column on the **Results List** pane. Click the column header so that all results with the same entry are grouped together. Select the first result and then select the last result while holding the `Shift` key. Assign a status to one of the results. If you do not see the **Detail** column, right-click any other column header and enable this column.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps
J.3.2: Environment	An alternative manner in which main function may be defined.	<p>The analysis flags main with arguments and return types other than:</p> <pre>int main(void) { ... }</pre> <p>or</p> <pre>int main(int argc, char *argv[]) { ... }</pre> <p>See section 5.1.2.2.1 of the C99 Standard.</p>
J.3.2: Environment	The set of environment names and the method for altering the environment list used by the getenv function.	<p>The analysis flags uses of the getenv function. For this function, you need to know the list of environment variables and how the list is modified.</p> <p>See section 7.20.4.5 of the C99 Standard.</p>
J.3.6: Floating Point	The rounding behaviors characterized by non-standard values of FLT_ROUNDS.	<p>The analysis flags the include of float.h if values of FLT_ROUNDS are outside the set, {-1, 0, 1, 2, 3}. Only the values in this set lead to well-defined rounding behavior.</p> <p>See section 5.2.4.2.2 of the C99 Standard.</p>
J.3.6: Floating Point	The evaluation methods characterized by non-standard negative values of FLT_EVAL_METHOD.	<p>The analysis flags the include of float.h if values of FLT_EVAL_METHOD are outside the set, {-1, 0, 1, 2}. Only the values in this set lead to well-defined behavior for floating-point operations.</p> <p>See section 5.2.4.2.2 of the C99 Standard.</p>

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps
J.3.6: Floating Point	The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value.	The analysis flags conversions from integer to floating-point data types of smaller size (for example, 64-bit <code>int</code> to 32-bit <code>float</code>). See section 6.3.1.4 of the C99 Standard.
J.3.6: Floating Point	The direction of rounding when a floating-point number is converted to a narrower floating-point number.	The analysis flags these conversions: <ul style="list-style-type: none"> • <code>double</code> to <code>float</code> • <code>long double</code> to <code>double</code> or <code>float</code> See section 6.3.1.5 of the C99 Standard.
J.3.6: Floating Point	The default state for the <code>FENV_ACCESS</code> pragma.	The analysis flags use of the pragma other than: <pre>#pragma STDC FENV_ACCESS ON</pre> or <pre>#pragma STDC FENV_ACCESS OFF</pre> See section 7.6.1 of the C99 Standard.
J.3.6: Floating Point	The default state for the <code>FP_CONTRACT</code> pragma.	The analysis flags use of the pragma other than: <pre>#pragma STDC FP_CONTRACT ON</pre> or <pre>#pragma STDC FP_CONTRACT OFF</pre> See section 7.12.2 of the C99 Standard.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps
J.3.11: Preprocessing Directives	The behavior on each recognized non-STDC <code>#pragma</code> directive.	The analysis flags the pragma usage: <code>#pragma pp-tokens</code> where the processing token STDC does not immediately follow <code>pragma</code> . For instance: <code>#pragma FENV_ACCESS ON</code> See section 6.10.6 of the C99 Standard.
J.3.12: Library Functions	Whether the <code>feraiseexcept</code> function raises the “inexact” floating-point exception in addition to the “overflow” or “underflow” floating-point exception.	The analysis flags calls to the <code>feraiseexcept</code> function. See section 7.6.2.3 of the C99 Standard.
J.3.12: Library Functions	Strings other than "C" and "" that may be passed as the second argument to the <code>setlocale</code> function.	The analysis flags calls to the <code>setlocale</code> function when its second argument is not "C" or "". See section 7.11.1.1 of the C99 Standard.
J.3.12: Library Functions	The types defined for <code>float_t</code> and <code>double_t</code> when the value of the <code>FLT_EVAL_METHOD</code> macro is less than 0 or greater than 2.	The analysis flags the include of <code>math.h</code> if <code>FLT_EVAL_METHOD</code> has values outside the set {0,1,2}. See section 7.12 of the C99 Standard.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps
J.3.12: Library Functions	The base-2 logarithm of the modulus used by the <code>remquo</code> functions in reducing the quotient.	The analysis flags calls to the <code>remquo</code> , <code>remquof</code> and <code>remquol</code> function. See section 7.12.10.3 of the C99 Standard.
J.3.12: Library Functions	The termination status returned to the host environment by the <code>abort</code> , <code>exit</code> , or <code>_Exit</code> function.	The analysis flags calls to the <code>abort</code> , <code>exit</code> , or <code>_Exit</code> function. See sections 7.20.4.1, 7.20.4.3 or 7.20.4.4 of the C99 Standard.

Check Information

Group: The implementation

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2017b

MISRA C:2012 Dir 2.1

All source files shall compile without any compilation errors

Description

Directive Definition

All source files shall compile without any compilation errors.

Rationale

A conforming compiler is permitted to produce an object module despite the presence of compilation errors. However, execution of the resulting program can produce unexpected behavior.

Polyspace Specification

The software raises a violation of this directive if it finds a compilation error. Because Code Prover is more strict about compilation errors compared to Bug Finder, the coding rules checking in the two products can produce different results for this directive.

Message in Report

All source files shall compile without any compilation errors.

Check Information

Group: Compilation and build

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 1.1

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Dir 4.1

Run-time failures shall be minimized

Description

Directive Definition

Run-time failures shall be minimized.

Rationale

Some areas to concentrate on are:

- Arithmetic errors
- Pointer arithmetic
- Array bound errors
- Function parameters
- Pointer dereferencing
- Dynamic memory

Polyspace Specification

This directive is checked through the Polyspace analysis. For more information, see:

- “Defects” (Polyspace Bug Finder).
- “Run-Time Checks”.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

Run-time failures shall be minimized.

Check Information

Group: Code design

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Dir 4.11 | MISRA C:2012 Rule 1.3 | MISRA C:2012 Rule 18.1
| MISRA C:2012 Rule 18.2 | MISRA C:2012 Rule 18.3

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Dir 4.3

Assembly language shall be encapsulated and isolated

Description

Directive Definition

Assembly language shall be encapsulated and isolated.

Rationale

Encapsulating assembly language is beneficial because:

- It improves readability.
- The name, and documentation, of the encapsulating macro or function makes the intent of the assembly language clear.
- All uses of assembly language for a given purpose can share encapsulation, which improves maintainability.
- You can easily substitute the assembly language for a different target or for purposes of static analysis.

Polyspace Specification

Polyspace does not raise a warning on assembly language code encapsulated in the following:

- `asm` functions or `asm` pragmas
- Macros

Message in Report

Assembly language shall be encapsulated and isolated

Examples

Assembly Language Code in C Function

```
enum boolVal {TRUE, FALSE};
enum boolVal isTaskActive;
void taskHandler(void);

void taskHandler(void) {
    isTaskActive = FALSE;
    // Software interrupt for task switching
    asm volatile
    (
        "SWI &02"      /* Service #1: calculate run-time */
    );
    return;
}
```

In this example, the rule violation occurs because the assembly language code is embedded directly in a C function `taskHandler` that contains other C language statements.

One possible correction is to encapsulate the assembly language code in a macro and invoke the macro in the function `taskHandler`.

```
#define RUN_TIME_CALC \
asm volatile \
( \
    "SWI &02"      /* Service #1: calculate run-Time */ \
)\

enum boolVal {TRUE, FALSE};
enum boolVal isTaskActive;
void taskHandler(void);

void taskHandler(void) {
    isTaskActive = FALSE;
    RUN_TIME_CALC;
    return;
}
```

Check Information

Group: Code design

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 1.2

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Dir 4.5

Identifiers in the same name space with overlapping visibility should be typographically unambiguous

Description

Directive Definition

Identifiers in the same name space with overlapping visibility should be typographically unambiguous.

Rationale

What “unambiguous” means depends on the alphabet and language in which source code is written. When you use identifiers that are typographically close, you can confuse between them.

For the Latin alphabet as used in English words, at a minimum, the identifiers should not differ by:

- The interchange of a lowercase letter with its uppercase equivalent.
- The presence or absence of the underscore character.
- The interchange of the letter O and the digit 0.
- The interchange of the letter I and the digit 1.
- The interchange of the letter l and the letter 1.
- The interchange of the letter S and the digit 5.
- The interchange of the letter Z and the digit 2.
- The interchange of the letter n and the letter h.
- The interchange of the letter B and the digit 8.
- The interchange of the letters rn and the letter m.

Message in Report

Identifiers in the same name space with overlapping visibility should be typographically unambiguous.

Examples

Typographically Ambiguous Identifiers

```
void func(void) {
    int id1_numval;
    int id1_num_val; /* Non-compliant */

    int id2_numval;
    int id2_numVal; /* Non-compliant */

    int id3_lvalue;
    int id3_lvalue; /* Non-compliant */

    int id4_xyz;
    int id4_xy2; /* Non-compliant */

    int id5_zer0;
    int id5_zer0; /* Non-compliant */

    int id6_rn;
    int id6_m; /* Non-compliant */
}
```

In this example, the rule is violated when identifiers that can be confused for each other are used.

Check Information

Group: Code design

Category: Advisory

AGC Category: Readability

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Dir 4.6

`typedefs` that indicate size and signedness should be used in place of the basic numerical types

Description

Directive Definition

typedefs that indicate size and signedness should be used in place of the basic numerical types.

Rationale

When the amount of memory being allocated is important, using specific-length types makes it clear how much storage is being reserved for each object.

Polyspace Specification

Polyspace does consider the use of basic types in a `typedef` statement as a violation of this directive.

Message in Report

`typedefs` that indicate size and signedness should be used in place of the basic numerical types

Check Information

Group: Code design

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Dir 4.9

A function should be used in preference to a function-like macro where they are interchangeable

Description

Directive Definition

A function should be used in preference to a function-like macro where they are interchangeable.

Rationale

In most circumstances, use functions instead of macros. Functions perform argument type-checking and evaluate their arguments once, avoiding problems with potential multiple side effects.

Polyspace Specification

Polyspace considers all function-like macro definitions.

Message in Report

A function should be used in preference to a function-like macro where they are interchangeable

Check Information

Group: Code design

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 13.2 | MISRA C:2012 Rule 20.7

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Dir 4.10

Precautions shall be taken in order to prevent the contents of a header file being included more than once

Description

Directive Definition

Precautions shall be taken in order to prevent the contents of a header file being included more than once.

Rationale

When a translation unit contains a complex hierarchy of nested header files, it is possible for a particular header file to be included more than once, leading to confusion. If this multiple inclusion produces multiple or conflicting definitions, then your program can have undefined or erroneous behavior.

For instance, suppose that a header file contains:

```
#ifndef _WIN64
    int env_var;
#elseif
    long int env_var;
#endif
```

If the header file is contained in two inclusion paths, one that defines the macro `_WIN64` and another that undefines it, you can have conflicting definitions of `env_var`.

Polyspace Specification

If you include a header file whose contents are not guarded from multiple inclusion, the analysis raises a violation of this directive. The violation is shown at the beginning of the header file.

You can guard the contents of a header file from multiple inclusion by using one of the following methods:

```
<start-of-file>
#ifndef <control macro>
#define <control macro>
    /* Contents of file */
#endif
<end-of-file>
```

or

```
<start-of-file>
#ifdef <control macro>
#error ...
#else
#define <control macro>
    /* Contents of file */
#endif
<end-of-file>
```

Unless you use one of these methods, Polyspace flags the header file inclusion as noncompliant.

Message in Report

Precautions shall be taken in order to prevent the contents of a header file being included more than once.

Examples

Code After Macro Guard

```
#ifndef __MY_MACRO__
#define __MY_MACRO__
    void func(void);
#endif
void func2(void);
```

If a header file contains this code, it is noncompliant because the macro guard does not cover the entire content of the header file. The line `void func2(void)` is outside the guard.

Note You can have comments outside the macro guard.

Code Before Macro Guard

```
void func(void);
#ifndef __MY_MACRO__
#define __MY_MACRO__
    void func2(void);
#endif
```

If a header file contains this code, it is noncompliant because the macro guard does not cover the entire content of the header file. The line `void func(void)` is outside the guard.

Note You can have comments outside the macro guard.

Mismatch in Macro Guard

```
#ifndef __MY_MACRO__
#define __MY_MARCO__
    void func(void);
    void func2(void);
#endif
```

If a header file contains this code, it is noncompliant because the macro name in the `#ifndef` statement is different from the name in the following `#define` statement.

Check Information

Group: Code Design

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Dir 4.11

The validity of values passed to library functions shall be checked

Description

Directive Definition

The validity of values passed to library functions shall be checked.

Rationale

Many Standard C functions do not check the validity of parameters passed to them. Even if checks are performed by a compiler, there is no guarantee that the checks are adequate. For example, you should not pass negative numbers to `sqrt` or `log`.

Polyspace Specification

Polyspace raises a violation result for library function arguments if the following are all true:

- Argument is a local variable.
- Local variable is not tested between last assignment and call to the library function.
- Corresponding parameter of the library function has a restricted input domain.
- Library function is one of the following common mathematical functions:
 - `sqrt`
 - `tan`
 - `pow`
 - `log`
 - `log10`
 - `fmod`
 - `acos`

- `asin`
- `acosh`
- `atanh`
- `or atan2`

Tip To mass-justify all results related to the same library function, use the **Detail** column on the **Results List** pane. Click the column header so that all results with the same entry are grouped together. Select the first result and then select the last result while holding the `Shift` key. Assign a status to one of the results. If you do not see the **Detail** column, right-click any other column header and enable this column.

Message in Report

The validity of values passed to library functions shall be checked

Check Information

Group: Code design

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Dir 4.1

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 1.1

The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits

Description

Rule Definition

The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.

Polyspace Specification

The rule violation can come from multiple causes. Standard compilation error messages do not lead to a violation of this MISRA rule.

Tip To mass-justify all results that come from the same cause, use the **Detail** column on the **Results List** pane. Click the column header so that all results with the same entry are grouped together. Select the first result and then select the last result while holding the **Shift** key. Assign a status to one of the results. If you do not see the **Detail** column, right-click any other column header and enable this column.

Message in Report

- Too many nesting levels of #includes: N1. The limit is N0.
- Integer constant is too large.
- ANSI C does not allow '#XX'.
- Text following preprocessing directive violates ANSI standard.
- Too many macro definitions: N1. The limit is N0.
- Array of zero size should not be used.
- Integer constant does not fit within long int.

- Integer constant does not fit within unsigned long int.
- Too many nesting levels for control flow: N1. The limit is N0.
- Assembly language should not be used.
- Too many enumeration constants: N1. The limit is N0.

Check Information

Group: Standard C Environment

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 1.2

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 1.2

Language extensions should not be used

Description

Rule Definition

Language extensions should not be used.

Rationale

If a program uses language extensions, its portability is reduced. Even if you document the language extensions, the documentation might not describe the behavior in all circumstances.

Polyspace Specification

All the supported extensions lead to a violation of this MISRA rule.

Message in Report

- ANSI C90 forbids hexadecimal floating-point constants.
- ANSI C90 forbids universal character names.
- ANSI C90 forbids mixed declarations and code.
- ANSI C90/C99 forbids case ranges.
- ANSI C90/C99 forbids local label declaration.
- ANSI C90 forbids mixed declarations and code.
- ANSI C90/C99 forbids typeof operator.
- ANSI C90/C99 forbids casts to union.
- ANSI C90 forbids compound literals.
- ANSI C90/C99 forbids statements and declarations in expressions.

- ANSI C90 forbids `__func__` predefined identifier.
- ANSI C90 forbids keyword `'_Bool'`.
- ANSI C90 forbids 'long long int' type.
- ANSI C90 forbids long long integer constants.
- ANSI C90 forbids 'long double' type.
- ANSI C90/C99 forbids 'short long int' type.
- ANSI C90 forbids `_Pragma` preprocessing operator.
- ANSI C90 does not allow macros with variable arguments list.
- ANSI C90 forbids designated initializer.

Keyword 'inline' should not be used.

Check Information

Group: Standard C Environment

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 1.1

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 1.3

There shall be no occurrence of undefined or critical unspecified behaviour

Description

Rule Definition

There shall be no occurrence of undefined or critical unspecified behaviour.

Message in Report

There shall be no occurrence of undefined or critical unspecified behavior

- 'defined' without an identifier.
- macro 'XX' used with too few arguments.
- macro 'XX' used with too many arguments.

Check Information

Group: Standard C Environment

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Dir 4.1

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 2.1

A project shall not contain unreachable code

Description

Rule Definition

A project shall not contain unreachable code.

Rationale

Unless a program exhibits any undefined behavior, unreachable code cannot execute. The unreachable code cannot affect the program output. The presence of unreachable code can indicate an error in the program logic. Unreachable code that the compiler does not remove wastes resources, for example:

- It occupies space in the target machine memory.
- Its presence can cause a compiler to select longer, slower jump instructions when transferring control around the unreachable code.
- Within a loop, it can prevent the entire loop from residing in an instruction cache.

Polyspace Specification

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

The Code Prover run-time check for unreachable code shows more cases than the MISRA checker for rule 2.1. See also `Unreachable code`. The run-time check performs a more exhaustive analysis. In the process, the check can show some instances that are not strictly unreachable code but unreachable only in the context of the analysis. For instance, in the following code, the run-time check shows a potential division by zero in the first line and then removes the zero value of `flag` for the rest of the analysis. Therefore, it considers the `if` block unreachable.

```
val=1.0/flag;  
if(!flag) {}
```

The MISRA checker is designed to prevent these kinds of results.

Message in Report

A project shall not contain unreachable code.

Examples

Code Following `return` Statement

```
enum light { red, amber, red_amber, green };  
  
enum light next_light ( enum light color )  
{  
    enum light res;  
  
    switch ( color )  
    {  
    case red:  
        res = red_amber;  
        break;  
    case red_amber:  
        res = green;  
        break;  
    case green:  
        res = amber;  
        break;  
    case amber:  
        res = red;  
        break;  
    default:  
    {  
        error_handler ();  
        break;  
    }  
    }  
  
    res = color;
```

```
    return res;
    res = color;    /* Non-compliant */
}
```

In this example, the rule is violated because there is an unreachable operation following the return statement.

Check Information

Group: Unused Code

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 14.3 | MISRA C:2012 Rule 16.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 2.2

There shall be no dead code

Description

Rule Definition

There shall be no dead code.

Rationale

If an operation is reachable but removing the operation does not affect program behavior, the operation constitutes dead code.

The presence of dead code can indicate an error in the program logic. Because a compiler can remove dead code, its presence can cause confusion for code reviewers.

Operations involving language extensions such as `__asm ("NOP");` are not considered dead code.

Polyspace Specification

Polyspace Code Prover does not detect useless write operations. For instance, if you assign a value to a local variable but do not read it later, Polyspace Code Prover does not detect this useless assignment. Use Polyspace Bug Finder to detect such useless write operations. For more information, see MISRA C:2012 in Polyspace Bug Finder.

Message in Report

There shall be no dead code.

Examples

Redundant Operations

```
extern volatile unsigned int v;
extern char *p;

void f ( void ) {
    unsigned int x;

    ( void ) v;      /* Compliant - Exception*/
    ( int ) v;       /* Non-compliant */
    v >> 3;          /* Non-compliant */

    x = 3;           /* Non-compliant - Detected in Bug Finder only */

    *p++;            /* Non-compliant */
    ( *p )++;        /* Compliant */
}
```

In this example, the rule is violated when an operation is performed on a variable, but the result of that operation is not used. For instance,

- The operations (int) and >> on the variable v are redundant because the results are not used.
- The operation = is redundant because the local variable x is not read after the operation.
- The operation * on p++ is redundant because the result is not used.

The rule is not violated when:

- A variable is cast to void. The cast indicates that you are intentionally not using the value.
- The result of an operation is used. For instance, the operation * on p is not redundant, because *p is incremented.

Redundant Function Call

```
void g ( void ) {
    /* Compliant */
}
```

```
}  
  
void h ( void) {  
    g( );      /* Non-compliant */  
}
```

In this example, `g` is an empty function. Though the function itself does not violate the rule, a call to the function violates the rule.

Check Information

Group: Unused Code

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 17.7

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 2.3

A project should not contain unused type declarations

Description

Rule Definition

A project should not contain unused type declarations.

Rationale

If a type is declared but not used, a reviewer does not know if the type is redundant or if it is unused by mistake.

Message in Report

A project should not contain unused type declarations: type XX is not used.

Examples

Unused Local Type

```
signed short unusedType (void){  
  
    typedef signed short myType;    /* Non-compliant */  
    return 67;  
  
}  
  
signed short usedType (void){  
  
    typedef signed short myType;    /* Compliant */  
    myType tempVar = 67;  
    return tempVar;
```

```
}
```

In this example, in function `unusedType`, the `typedef` statement defines a new local type `myType`. However, this type is never used in the function. Therefore, the rule is violated.

The rule is not violated in the function `usedType` because the new type `myType` is used.

Check Information

Group: Unused Code

Category: Advisory

AGC Category: Readability

Language: C90, C99

See Also

MISRA C:2012 Rule 2.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 2.4

A project should not contain unused tag declarations

Description

Rule Definition

A project should not contain unused tag declarations.

Rationale

If a tag is declared but not used, a reviewer does not know if the tag is redundant or if it is unused by mistake.

Message in Report

A project should not contain unused tag declarations: tag *tag_name* is not used.

Examples

Tag Defined in Function but Not Used

```
void unusedTag ( void )
{
    enum state1 { S_init, S_run, S_sleep };    /* Non-compliant */
}

void usedTag ( void )
{
    enum state2 { S_init, S_run, S_sleep };    /* Compliant */
    enum state2 my_State = S_init;
}
```

In this example, in the function `unusedTag`, the tag `state1` is defined but not used. Therefore, the rule is violated.

Tag Used in typedef Only

```
typedef struct record_t          /* Non-compliant */
{
    unsigned short key;
    unsigned short val;
} record1_t;

typedef struct                  /* Compliant */
{
    unsigned short key;
    unsigned short val;
} record2_t;

record1_t myRecord1_t;
record2_t myRecord2_t;
```

In this example, the tag `record_t` appears only in the typedef of `record1_t`. In the rest of the translation unit, the type `record1_t` is used. Therefore, the rule is violated.

Check Information

Group: Unused Code

Category: Advisory

AGC Category: Readability

Language: C90, C99

See Also

MISRA C:2012 Rule 2.3

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 2.5

A project should not contain unused macro declarations

Description

Rule Definition

A project should not contain unused macro declarations.

Rationale

If a macro is declared but not used, a reviewer does not know if the macro is redundant or if it is unused by mistake.

Message in Report

A project should not contain unused macro declarations: macro *macro_name* is not used.

Examples

Unused Macro Definition

```
void use_macro (void)
{
    #define SIZE 4
    #define DATA 3

    use_int16(SIZE);
}
```

In this example, the macro `DATA` is never used in the `use_macro` function.

Check Information

Group: Unused Code

Category: Advisory

AGC Category: Readability

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 2.6

A function should not contain unused label declarations

Description

Rule Definition

A function should not contain unused label declarations.

Rationale

If you declare a label but do not use it, it is not clear to a reviewer of your code if the label is redundant or unused by mistake.

Message in Report

A function should not contain unused label declarations.

Label *label_name* is not used.

Examples

Unused Label Declarations

```
void use_var(signed short);

void unused_label ( void )
{
    signed short x = 6;

label1:                                /* Non-compliant - label1 not used */
    use_var ( x );
}
```

```
void unused_label ( void )
{
    signed short x = 6;

    for (int i=0; i < 5; i++) {
        if ( i==2 ) goto labell1;
    }

labell1:                                /* Compliant - labell1 used */
    use_var ( x );
}
```

In this example, the rule is violated when the label `labell1` in function `unused_label` is not used.

Check Information

Group: Unused code

Category: Advisory

AGC Category: Readability

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 2.7

There should be no unused parameters in functions

Description

Rule Definition

There should be no unused parameters in functions.

Rationale

If a parameter is unused, it is possible that the implementation of the function does not match its specifications. This rule can highlight such mismatches.

Message in Report

There should be no unused parameters in functions.

Parameter `parameter_name` is not used.

Examples

Unused Function Parameters

```
double func(int param1, int* param2) {  
    return (param1/2.0);  
}
```

In this example, the rule is violated because the parameter `param2` is not used.

Check Information

Group: Unused code

Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”
“Review Coding Rule Violations”
“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 3.1

The character sequences `/*` and `//` shall not be used within a comment

Description

Rule Definition

The character sequences `/` and `//` shall not be used within a comment.*

Rationale

These character sequences are not allowed in code comments because:

- If your code contains a `/*` or a `//` in a `/* */` comment, it typically means that you have inadvertently commented out code.
- If your code contains a `/*` in a `//` comment, it typically means that you have inadvertently uncommented a `/* */` comment.

Polyspace Specification

You cannot annotate this rule in the source code.

For information on annotations, see “Justify Results Through Code Annotations”.

Message in Report

The character sequence `/*` shall not appear within a comment.

Examples

`/*` Used in `//` Comments

```
int x;  
int y;
```

```
int z;

void non_compliant_comments ( void )
{
    x = y //      /* Non-compliant
        + z
        // */
    ;
    z++; //      Compliant with exception: // permitted within a // comment
}

void compliant_comments ( void )
{
    x = y /*      Compliant
        + z
        */
    ;
    z++; //      Compliant with exception: // is permitted within a // comment
}
```

In this example, in the `non_compliant_comments` function, the `/*` character occurs in what appears to be a `//` comment, violating the rule. Because of the comment structure, the operation that takes place is `x = y + z`; . However, without the two `//`-s, an entirely different operation `x=y`; takes place. It is not clear which operation is intended.

Use a comment format that makes your intention clear. For instance, in the `compliant_comments` function, it is clear that the operation `x=y`; is intended.

Check Information

Group: Comments

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”
“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 3.2

Line-splicing shall not be used in `//` comments

Description

Rule Definition

Line-splicing shall not be used in `//` comments.

Rationale

Line-splicing occurs when the `\` character is immediately followed by a new-line character. Line splicing is used for statements that span multiple lines.

If you use line-splicing in a `//` comment, the following line can become part of the comment. In most cases, the `\` is spurious and can cause unintentional commenting out of code.

Message in Report

Line-splicing shall not be used in `//` comments.

Examples

Line Splicing in `//` Comment

```
#include <stdbool.h>

extern _Bool b;

void func ( void )
{
    unsigned short x = 0;    // Non-compliant - Line-splicing \
    if ( b )
```

```
    {  
        ++b;  
    }  
}
```

Because of line-splicing, the statement `if (b)` is a part of the previous `// comment`. Therefore, the statement `b++` always executes, making the `if` block redundant.

Check Information

Group: Comments

Category: Required

AGC Category: Required

Language: C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Software Quality Objective Subsets (C:2012)” (Polyspace Bug Finder)

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 4.1

Octal and hexadecimal escape sequences shall be terminated

Description

Rule Definition

Octal and hexadecimal escape sequences shall be terminated.

Rationale

There is potential for confusion if an octal or hexadecimal escape sequence is followed by other characters. For example, the character constant '\x1f' consists of a single character, whereas the character constant '\x1g' consists of the two characters '\x1' and 'g'. The manner in which multi-character constants are represented as integers is implementation-defined.

If every octal or hexadecimal escape sequence in a character constant or string literal is terminated, you reduce potential confusion.

Message in Report

Octal and hexadecimal escape sequences shall be terminated.

Examples

Compliant and Noncompliant Escape Sequences

```
const char *s1 = "\x41g";      /* Non-compliant */
const char *s2 = "\x41" "g";  /* Compliant - Terminated by end of literal */
const char *s3 = "\x41\x67";  /* Compliant - Terminated by another escape sequence*/

int c1 = '\141t';             /* Non-compliant */
int c2 = '\141\t';           /* Compliant - Terminated by another escape sequence*/
```


In this example, the rule is violated when an escape sequence is not terminated with the end of string literal or another escape sequence.

Check Information

Group: Character Sets and Lexical Conventions

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 4.2

Trigraphs should not be used

Description

Rule Definition

Trigraphs should not be used.

Rationale

You denote trigraphs with two question marks followed by a specific third character (for instance, '??-' represents a '~' (tilde) character and '??)' represents a ']'). These trigraphs can cause accidental confusion with other uses of two question marks.

Note Digraphs (<: :>, <% %>, %:, %:%:) are permitted because they are tokens.

Polyspace Specification

The Polyspace analysis converts trigraphs to the equivalent character for the run-time verification. However, Polyspace also raises a MISRA violation.

The standard requires that trigraphs must be transformed *before* comments are removed during preprocessing. Therefore, Polyspace raises a violation of this rule even if a trigraph appears in code comments.

Message in Report

Trigraphs should not be used.

Check Information

Group: Character Sets and Lexical Conventions

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.1

External identifiers shall be distinct

Description

Rule Definition

External identifiers shall be distinct.

Rationale

External identifiers are ones declared with global scope or storage class `extern`.

Polyspace considers two names as distinct if there is a difference between their first 31 characters. If the difference between two names occurs only beyond the first 31 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 6 characters. To use the C90 rules checking, use the option, Respect C90 Standard.

Message in Report

External %s %s conflicts with the external identifier XX in file YY.

Examples

C90: First Six Characters of Identifiers Not Unique

```
int engine_temperature_raw;  
int engine_temperature_scaled; /* Non-compliant */  
int engin2_temperature; /* Compliant */
```

In this example, the identifier `engine_temperature_scaled` has the same first six characters as a previous identifier, `engine_temperature_raw`.

C99: First 31 Characters of Identifiers Not Unique

```
int engine_exhaust_gas_temperature_raw;
int engine_exhaust_gas_temperature_scaled; /* Non-compliant */

int eng_exhaust_gas_temp_raw;
int eng_exhaust_gas_temp_scaled;          /* Compliant */
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same first 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

C90: First Six Characters Identifiers in Different Translation Units Differ in Case Alone

```
/* file1.c */
int abc = 0;

/* file2.c */
int ABC = 0; /* Non-compliant */
```

In this example, the implementation supports 6 significant case-insensitive characters in *external identifiers*. The identifiers in the two translation are different but are not distinct in their significant characters.

Check Information

Group: Identifiers

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 5.2 | MISRA C:2012 Rule 5.4 | MISRA C:2012 Rule 5.5

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.2

Identifiers declared in the same scope and name space shall be distinct

Description

Rule Definition

Identifiers declared in the same scope and name space shall be distinct.

Rationale

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the option, Respect C90 Standard.

Message in Report

Identifier XX has same significant characters as identifier YY.

Examples

C90: First 31 Characters of Identifiers Not Unique

```
extern int engine_exhaust_gas_temperature_raw;
static int engine_exhaust_gas_temperature_scaled;      /* Non-compliant */

extern double engine_exhaust_gas_temperature_raw;
static double engine_exhaust_gas_temperature2_scaled; /* Compliant */

void func ( void )
{
    /* Not in the same scope */
```

```
int engine_exhaust_gas_temperature_local;          /* Compliant */  
}
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

The rule does not apply if the two identifiers have the same 31 characters but have different scopes. For instance, `engine_exhaust_gas_temperature_local` has the same 31 characters as `engine_exhaust_gas_temperature_raw` but different scope.

C99: First 63 Characters of Identifiers Not Unique

```
extern int engine_xxx_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx_x_raw;  
static int engine_xxx_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx_x_scale;  
/* Non-compliant */  
  
extern int engine_gas_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx_raw;  
static int engine_gas_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx_scale;  
/* Compliant */  
  
void func ( void )  
{  
/* Not in the same scope */  
int engine_xxx_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx_x_local;  
/* Compliant */  
}
```

In this example, the identifier

`engine_xxx_xxx_x_scale`
has the same 63 characters as a previous identifier,

`engine_xxx_xxx_x_raw`.

Check Information

Group: Identifiers

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 5.1 | MISRA C:2012 Rule 5.3 | MISRA C:2012 Rule 5.4 |
MISRA C:2012 Rule 5.5

Topics

“Set Up Coding Rules Checking”
“Review Coding Rule Violations”
“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.3

An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

Description

Rule Definition

An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.

Rationale

If two identifiers have the same name but different scope, the identifier in the inner scope hides the identifier in the outer scope. All uses of the identifier name refers to the identifier in the inner scope. This behavior forces the developer to keep track of the scope and reduces code readability.

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the option, Respect C90 Standard.

Message in Report

Variable XX hides variable XX (FILE line LINE column COLUMN).

Examples

Local Variable Hidden by Another Local Variable in Inner Block

```
typedef signed short int16_t;
```

```

void func( void )
{
    int16_t i;
    {
        int16_t i;           /* Non-compliant */
        i = 3;
    }
}

```

In this example, the identifier `i` defined in the inner block in `func` hides the identifier `i` with function scope.

It is not immediately clear to a reader which `i` is referred to in the statement `i=3`.

Global Variable Hidden by Function Parameter

```

typedef signed short int16_t;

struct astruct
{
    int16_t m;
};

extern void g ( struct astruct *p );
int16_t xyz = 0;

void func ( struct astruct xyz ) /* Non-compliant */
{
    g ( &xyz );
}

```

In this example, the parameter `xyz` of function `func` hides the global variable `xyz`.

It is not immediately clear to a reader which `xyz` is referred to in the statement `g (&xyz)`.

Check Information

Group: Identifiers

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 5.2 | MISRA C:2012 Rule 5.8

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.4

Macro identifiers shall be distinct

Description

Rule Definition

Macro identifiers shall be distinct.

Rationale

The names of macro identifiers must be distinct from both other macro identifiers and their parameters.

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the option, Respect C90 Standard.

Message in Report

- Macro identifiers shall be distinct. Macro XX has same significant characters as macro YY.
- Macro identifiers shall be distinct. Macro parameter XX has same significant characters as macro parameter YY in macro ZZ.

Examples

C90: First 31 Characters of Macro Names Not Unique

```
#define engine_exhaust_gas_temperature_raw egt_r
#define engine_exhaust_gas_temperature_scaled egt_s /* Non-compliant */
```

```
#define engine_exhaust_gas_temp_raw egt_r
#define engine_exhaust_gas_temp_scaled egt_s          /* Compliant */
```

In this example, the macro `engine_exhaust_gas_temperature_scaled egt_s` has the same first 31 characters as a previous macro `engine_exhaust_gas_temperature_scaled`.

C99: First 63 Characters of Macro Names Not Unique

```
#define engine_xxx_XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX_raw egt_r
#define engine_xxx_XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX_raw_scaled egt_s
    /* Non-compliant */

/* 63 significant case-sensitive characters in macro identifiers */
#define new_engine_XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX_raw egt_r
#define new_engine_XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX_scaled egt_s
    /* Compliant */
```

In this example, the macro `engine_xxx_XX_gaz_s` caled has the same first 63 characters as a previous macro `engine_xxx_XX_raw`.

Check Information

Group: Identifiers

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 5.1 | MISRA C:2012 Rule 5.2 | MISRA C:2012 Rule 5.5

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.5

Identifiers shall be distinct from macro names

Description

Rule Definition

Identifiers shall be distinct from macro names.

Rationale

The rule requires that macro names that exist only prior to processing must be different from identifier names that also exist after preprocessing. Keeping macro names and identifiers distinct help avoid confusion.

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the option, Respect C90 Standard.

Message in Report

Identifier XX has same significant characters as macro YY.

Examples

Macro Names Same as Identifier Names

```
#define Sum_1(x, y) ( ( x ) + ( y ) )
short Sum_1;                               /* Non-compliant */

#define Sum_2(x, y) ( ( x ) + ( y ) )
short x = Sum_2 ( 1, 2 );                   /* Compliant */
```


In this example, `Sum_1` is both the name of an identifier and a macro. `Sum_2` is used only as a macro.

C90: First 31 Characters of Macro Name Same as Identifier Name

```
#define      low_pressure_turbine_temperature_1 lp_tb_temp_1
static int low_pressure_turbine_temperature_2;      /* Non-compliant */
```

In this example, the identifier `low_pressure_turbine_temperature_2` has the same first 31 characters as a previous macro `low_pressure_turbine_temperature_1`.

Check Information

Group: Identifiers

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 5.1 | MISRA C:2012 Rule 5.2 | MISRA C:2012 Rule 5.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.6

A typedef name shall be a unique identifier

Description

Rule Definition

A typedef name shall be a unique identifier.

Rationale

Reusing a typedef name as another typedef or as the name of a function, object or enum constant can cause developer confusion.

Message in Report

XX conflicts with the typedef name YY.

Examples

`typedef` Names Reused

```
void func ( void ){
    {
        typedef unsigned char u8_t;
    }
    {
        typedef unsigned char u8_t; /* Non-compliant */
    }
}

typedef float mass;
void func1 ( void ){
    float mass = 0.0f;           /* Non-compliant */
}
```

In this example, the typedef name `u8_t` is used twice. The typedef name `mass` is also used as an identifier name.

typedef Name Same as Structure Name

```
typedef struct list{           /* Compliant - exception */
    struct list *next;
    unsigned short element;
} list;

typedef struct{
    struct chain{             /* Non-compliant */
        struct chain *list2;
        unsigned short element;
    } s1;
    unsigned short length;
} chain;
```

In this example, the typedef name `list` is the same as the original name of the struct type. The rule allows this exceptional case.

However, the typedef name `chain` is not the same as the original name of the struct type. The name `chain` is associated with a different struct type. Therefore, it clashes with the typedef name.

Check Information

Group: Identifiers

Category:

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 5.7

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.7

A tag name shall be a unique identifier

Description

Rule Definition

A tag name shall be a unique identifier.

Rationale

Reusing a tag name can cause developer confusion.

Message in Report

XX conflicts with the tag name YY.

Check Information

Group: Identifiers

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 5.6

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.8

Identifiers that define objects or functions with external linkage shall be unique

Description

Rule Definition

Identifiers that define objects or functions with external linkage shall be unique.

Rationale

External identifiers are those declared with global scope or with storage class `extern`. Reusing an external identifier name can cause developer confusion.

Identifiers defined within a function have smaller scope. Even if names of such identifiers are not unique, they are not likely to cause confusion.

Message in Report

- Object XX conflicts with the object name YY.
- Function XX conflicts with the function name YY.

Check Information

Group: Identifiers

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 5.3

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.9

Identifiers that define objects or functions with internal linkage should be unique

Description

Rule Definition

Identifiers that define objects or functions with internal linkage should be unique.

Polyspace Specification

This rule checker assumes that rule 5.8 is not violated.

Message in Report

- Object XX conflicts with the object name YY.
- Function XX conflicts with the function name YY.

Check Information

Group: Identifiers

Category: Advisory

AGC Category: Readability

Language: C90, C99

See Also

MISRA C:2012 Rule 8.10

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 6.1

Bit-fields shall only be declared with an appropriate type

Description

Rule Definition

Bit-fields shall only be declared with an appropriate type.

Rationale

Using `int` is implementation-defined because bit-fields of type `int` can be either signed or unsigned.

The use of `enum`, `short char`, or any other type of bit-field is not permitted in C90 because the behavior is undefined.

In C99, the implementation can potentially define other integer types that are permitted in bit-field declarations.

Message in Report

Bit-fields shall only be declared with an appropriate type.

Check Information

Group: Types

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 6.2

Single-bit named bit fields shall not be of a signed type

Description

Rule Definition

Single-bit named bit fields shall not be of a signed type.

Rationale

According to the C99 Standard Section 6.2.6.2, a single-bit signed bit-field has one sign bit and no value bits. In any representation of integers, zero value bits cannot specify a meaningful value.

A single-bit signed bit-field is therefore unlikely to behave in a useful way. Its presence is likely to indicate programmer confusion.

Although the C90 Standard does not provide much detail regarding the representation of types, the same single-bit bit-field considerations apply.

Polyspace Specification

This rule does not apply to unnamed bit fields because their values cannot be accessed.

Message in Report

Single-bit named bit fields shall not be of a signed type.

Check Information

Group: Types

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 7.1

Octal constants shall not be used

Description

Rule Definition

Octal constants shall not be used.

Rationale

Octal constants are denoted by a leading zero. Developers can mistake an octal constant as a decimal constant with a redundant leading zero.

Message in Report

Octal constants shall not be used.

Examples

Use of octal constants

```
#define CST      021
#define VALUE    010          /* Compliant - constant not used */
#if 010 == 01              /* Non-Compliant - constant used */
#define CST 021          /* Compliant - constant not used */
#endif

extern short code[5];
static char* str2 = "abcd\0efg"; /* Compliant */

void main(void) {
    int value1 = 0;          /* Compliant */
    int value2 = 01;        /* Non-Compliant - decimal 01 */
}
```

```
int value3 = 1;           /* Compliant */
int value4 = '\109';     /* Compliant */

code[1] = 109;          /* Compliant - decimal 109 */
code[2] = 100;          /* Compliant - decimal 100 */
code[3] = 052;          /* Non-Compliant - decimal 42 */
code[4] = 071;          /* Non-Compliant - decimal 57 */

if (value1 != CST) {    /* Non-Compliant - decimal 17 */
    value1 = !(value1 != 0); /* Compliant */
}
}
```

In this example, the rule is not violated when octal constants are used to define macros CST and VALUE. The rule is violated only when the macros are used.

Check Information

Group: Literals and Constants

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 7.2

A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type

Description

Rule Definition

A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type.

Rationale

The signedness of a constant is determined from:

- Value of the constant.
- Base of the constant: octal, decimal or hexadecimal.
- Size of the various types.
- Any suffixes used.

Unless you use a suffix u or U, another developer looking at your code cannot determine easily whether a constant is signed or unsigned.

Message in Report

A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type.

Check Information

Group: Literals and Constants

Category: Required

AGC Category: Readability

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 7.3

The lowercase character “l” shall not be used in a literal suffix

Description

Rule Definition

The lowercase character “l” shall not be used in a literal suffix.

Rationale

The lowercase character “l” can be confused with the digit “1”. Use the uppercase “L” instead.

Message in Report

The lowercase character “l” shall not be used in a literal suffix.

Check Information

Group: Literals and Constants

Category: Required

AGC Category: Readability

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 7.4

A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char"

Description

Rule Definition

A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Rationale

This rule prevents assignments that allow modification of a string literal.

An attempt to modify a string literal can result in undefined behavior. For example, some implementations can store string literals in read-only memory. An attempt to modify the string literal can result in an exception or crash.

Message in Report

A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Examples

Incorrect Assignment of String Literal

```
char *str1 = "AccountHolderName";  
const char *str2 = "AccountHolderName";  
  
void checkAccount1(char*);           /* Non-Compliant */  
void checkAccount2(const char*);     /* Compliant */
```

```
void main() {  
    checkAccount1("AccountHolderName");    /* Non-Compliant */  
    checkAccount2("AccountHolderName");    /* Compliant */  
}
```

In this example, the rule is not violated when string literals are assigned to `const char*` pointers, either directly or through copy of function arguments. The rule is violated only when the `const` qualifier is not used.

Check Information

Group: Literals and Constants

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 11.4 | MISRA C:2012 Rule 11.8

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.1

Types shall be explicitly specified

Description

Rule Definition

Types shall be explicitly specified.

Rationale

The C90 standard permits types to be omitted in some circumstances, in which case the `int` type is implicitly specified. Examples of potential circumstances in which you can use an implicit `int` are:

- Object declarations
- Parameter declarations
- Member declarations
- `typedef` declarations
- Function return types

The omission of an explicit type can lead to confusion. For example, in the declaration `extern void foo (char c, const k);`, the type of `k` is `const int`, but `const char` might have been expected.

Message in Report

Types shall be explicitly specified.

Check Information

Group: Declarations and Definitions

Category: Required

AGC Category: Required
Language: C90

See Also

MISRA C:2012 Rule 8.2

Topics

“Set Up Coding Rules Checking”
“Review Coding Rule Violations”
“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.2

Function types shall be in prototype form with named parameters

Description

Rule Definition

Function types shall be in prototype form with named parameters.

Rationale

The mismatch between the number of arguments and parameters, their types, and the expected and actual return type of a function provides potential for undefined behavior. This rule also requires that you specify names for all the parameters in a declaration. The parameter names provide useful information regarding the function interface. A mismatch between a declaration and definition can indicate a programming error.

Polyspace Specification

Polyspace also checks the function definition.

Message in Report

- Too many arguments to *function_name*.
- Too few arguments to *function_name*.
- Function types shall be in prototype form with named parameters.

Check Information

Group: Declarations and Definitions

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 8.1 | MISRA C:2012 Rule 8.4 | MISRA C:2012 Rule 17.3

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.3

All declarations of an object or function shall use the same names and type qualifiers

Description

Rule Definition

All declarations of an object or function shall use the same names and type qualifiers.

Rationale

Consistently using types and qualifiers across declarations of the same object or function encourages stronger typing. By specifying parameter names in function prototypes, Polyspace can check for interface consistency between the function definition and declarations.

Polyspace Specification

Polyspace generates some violations of this rule during the link phase.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

- Definition of function *function_name* incompatible with its declaration.
- Global declaration of *function_name* function has incompatible type with its definition.
- Global declaration of *variable_name* variable has incompatible type with its definition.
- All declarations of an object or function shall use the same names and type qualifiers.

Check Information

Group: Declarations and Definitions

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 8.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.4

A compatible declaration shall be visible when an object or function with external linkage is defined

Description

Rule Definition

A compatible declaration shall be visible when an object or function with external linkage is defined.

Rationale

If a declaration for an object or function is visible when the object or function is defined, a compiler must check that the declaration and definition are compatible. In the presence of function prototypes, as required by rule 8.2, checking extends to the number and type of function parameters. A better way of implementing declarations of objects and functions with external linkage is to declare them in a header file. Then include the header file in all those code files that require them, including the one that defines them.

Message in Report

- Global definition of *variable_name* variable has no previous declaration.
- Function *function_name* has no visible compatible prototype at definition.

Check Information

Group: Declarations and Definitions

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 8.2 | MISRA C:2012 Rule 8.3 | MISRA C:2012 Rule 8.5 |
MISRA C:2012 Rule 17.3

Topics

“Set Up Coding Rules Checking”
“Review Coding Rule Violations”
“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.5

An external object or function shall be declared once in one and only one file

Description

Rule Definition

An external object or function shall be declared once in one and only one file.

Rationale

Typically, a single declaration is made in a header file that you include in any translation unit in which the identifier is defined or used. This inclusion ensures consistency between:

- The declaration and the definition
- The declarations in different translation units

Note It is possible to have many header files in a project, but each external object or function is declared in only one header file.

Polyspace Specification

Polyspace checks only explicit `extern` declarations (tentative definitions are ignored). The rule checker considers that variables or functions declared `extern` in a non-header file violates this rule.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

- Object `object_name` has external declarations in multiples files.

- Function *function_name* has external declarations in multiples files.

Check Information

Group: Declarations and Definitions

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 8.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.6

An identifier with external linkage shall have exactly one external definition

Description

Rule Definition

An identifier with external linkage shall have exactly one external definition.

Rationale

The behavior is undefined if you use an identifier for which multiple definitions exist (in different files) or no definition exists. Multiple definitions in different files are not permitted by this rule even if the definitions are the same. If the declarations are different, or initialize the identifier to different values, it is undefined behavior.

Polyspace Specification

The checker flags multiple definitions only if the definitions occur in different files. The checker does not:

- Consider tentative definitions as definitions.

For instance, the following code does not violate the rule:

```
int val;  
int val=1;
```

- Does not show a violation for functions that are called in the source code with external linkage but not defined.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

- Forbidden multiple definitions for function *function_name*.
- Forbidden multiple tentative definitions for object *object_name*.
- Global variable *variable_name* multiply defined.
- Function *function_name* multiply defined.
- Global variable has multiple tentative definitions.
- Undefined global variable *variable_name*.

Check Information

Group: Declarations and Definitions

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.7

Functions and objects should not be defined with external linkage if they are referenced in only one translation unit

Description

Rule Definition

Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.

Rationale

Restricting or reducing the visibility of an object by giving it internal linkage or no linkage reduces the chance that it is accessed inadvertently. Compliance with this rule also avoids any possibility of confusion between your identifier and an identical identifier in another translation unit or a library.

Polyspace Specification

If your program does not use the externally defined function or object, Polyspace does not raise a warning.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

- Variable `variable_name` should have internal linkage.
- Function `function_name` should have internal linkage.

Check Information

Group: Declarations and Definitions

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.8

The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage

Description

Rule Definition

The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.

Rationale

If you have an object or function declared with `extern`, and another declaration of the object or function is already visible, the linkage can be confusing. You expect that the `extern` storage class specifier creates external linkage. Apply the static storage class specifier to objects and functions with internal linking.

Message in Report

The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.

Examples

Internal and External Linkage Conflicts

```
static int foo = 0;
extern int foo;          /* Non-compliant */

extern int hhh;
static int hhh;         /* Non-compliant */
```

In this example, the first line defines `foo` with internal linkage. Because the example uses the `static` keyword, the first line is compliant. However, the second line does not use `static` in the declaration, so the declaration is noncompliant. By comparison, the third line declares `hhh` with an `extern` keyword creating external linkage. The fourth line declares `hhh` with internal linkage, but this declaration conflicts with the first declaration of `hhh`.

One possible correction is to use `static` and `extern` consistently:

```
static int foo = 0;
static int foo;

extern int hhh;
extern int hhh;
```

Internal linkage

```
static int fee(void); /* Compliant - declaration: internal linkage */
int fee(void){ /* Non-compliant */
    return 1;
}

static int ggg(void); /* Compliant - declaration: internal linkage */
extern int ggg(void){ /* Non-compliant */
    return 1 + x;
}
```

This example shows two internal linkage violations. Because `fee` and `ggg` have internal linkage, you must use a `static` class specifier to be compliant with MISRA

Check Information

Group: Declarations and Definitions

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.9

An object should be defined at block scope if its identifier only appears in a single function

Description

Rule Definition

An object should be defined at block scope if its identifier only appears in a single function.

Rationale

Defining an object at block scope reduces the possibility that you inadvertently access the object . It ensures your program does not access the object elsewhere.

Polyspace Specification

Polyspace raises a warning only for static objects.

Message in Report

An object should be defined at block scope if its identifier only appears in a single function.

Check Information

Group: Declarations and Definitions

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.10

An inline function shall be declared with the static storage class

Description

Rule Definition

An inline function shall be declared with the static storage class.

Rationale

If you call an inline function with external linkage, you can call the external definition of the function or the inline definition. This behavior can affect the execution time and therefore impact your program.

Tip To make an inline function available to several translation units, place its definition in a header file.

Message in Report

An inline function shall be declared with the static storage class.

Check Information

Group: Declarations and Definitions

Category: Required

AGC Category: Required

Language: C99

See Also

MISRA C:2012 Rule 5.9

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.11

When an array with external linkage is declared, its size should be explicitly specified

Description

Rule Definition

When an array with external linkage is declared, its size should be explicitly specified.

Rationale

Although it is possible to declare an array with incomplete type and access its elements, it is safer to state the size of the array explicitly. Providing size information for each declaration allows the software to check the declarations for consistency. It also allows a static checker to perform array bounds analysis without analyzing more than one unit.

Message in Report

Size of array `array_name` should be explicitly stated. When an array with external linkage is declared, its size should be explicitly specified.

Examples

Array Declarations

```
extern int32_t array1[10];    /* Compliant */
extern int32_t array2[];    /* Non-compliant */
```

In this example, two arrays are declared `array1` and `array2`. `array1` has external linkage (the `extern` keyword) and a size of 10. `array2` also has external linkage, but no specified size. `array2` is noncompliant because for arrays with external linkage, you must explicitly specify a size.

Check Information

Group: Declarations and Definitions

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.12

Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique

Description

Rule Definition

Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.

Rationale

An implicitly specified enumeration constant has a value 1 greater than its predecessor. If the first enumeration constant is implicitly specified, then its value is 0. An explicitly specified enumeration constant has the value of the associate constant expression.

If implicitly and explicitly specified constants are mixed within an enumeration list, it is possible for your program to replicate values. Such replications can be unintentional and can cause unexpected behavior.

Message in Report

The constant *constant1* has same value as the constant *constant2*.

Check Information

Group: Declarations and Definitions

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.13

A pointer should point to a const-qualified type whenever possible

Description

Rule Definition

A pointer should point to a const-qualified type whenever possible.

Rationale

This rule ensures that you do not inadvertently use pointers to modify objects.

Polyspace Specification

Polyspace issues a warning if a non-const pointer parameter either:

- Does not modify the addressed object.
- Is passed to a call of a function that is declared with a const pointer parameter.

Message in Report

A pointer should point to a const-qualified type whenever possible.

Examples

Pointer Parameters

```
#include <string.h>

typedef unsigned short uint16_t;

uint16_t ptr_ex(uint16_t *p) {      /* Non-compliant */
```



```

    return *p;
}

char last_char(char * const s){    /* Non-compliant */
    return s[strlen(s) - 1u];
}

uint16_t first(uint16_t a[5]){    /* Non-compliant */
    return a[0];
}

```

This example shows three different noncompliant pointer parameters. In the `ptr_ex` function, `p` does not modify an object. However, the type to which `p` points is not `const`-qualified, so it is noncompliant. In `last_char`, the pointer `s` is `const`-qualified but the type it points to is not. Because `s` does not modify an object, this parameter is noncompliant. The function `first` does not modify the elements of the array `a`. However, the element type is not `const`-qualified, so `a` is also noncompliant.

One possible correction is to add `const` qualifiers to the definitions.

```

#include <string.h>

typedef unsigned short uint16_t;

uint16_t ptr_ex(const uint16_t *p){    /* Compliant */
    return *p;
}

char last_char(const char * const s){    /* Compliant */
    return s[strlen( s ) - 1u];
}

uint16_t first(const uint16_t a[5]) {    /* Compliant */
    return a[0];
}

```

Check Information

Group: Declarations and Definitions

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.14

The restrict type qualifier shall not be used

Description

Rule Definition

The restrict type qualifier shall not be used.

Rationale

When you use a `restrict` qualifier carefully, it improves the efficiency of code generated by a compiler. It can also improve static analysis. However, when using the `restrict` qualifier, make sure that the memory areas operated on by two or more pointers do not overlap.

Message in Report

The restrict type qualifier shall not be used.

Check Information

Group: Declarations and Definitions

Category: Required

AGC Category: Advisory

Language: C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 9.1

The value of an object with automatic storage duration shall not be read before it has been set

Description

Message in Report:

Rule Definition

The value of an object with automatic storage duration shall not be read before it has been set.

Rationale

A variable with an automatic storage duration is allocated memory at the beginning of an enclosing code block and deallocated at the end. All non-global variables have this storage duration, except those declared `static` or `extern`.

Variables with automatic storage duration are not automatically initialized and have indeterminate values. Therefore, you must not read such a variable before you have set its value through a write operation.

Polyspace Specification

The Polyspace analysis checks some of the violations as non-initialized variables. For more information, see `Non-initialized local variable`.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

The value of an object with automatic storage duration shall not be read before it has been set.

Check Information

Group: Initialization

Category: Mandatory

AGC Category: Mandatory

Language: C90, C99

See Also

MISRA C:2012 Rule 15.1 | MISRA C:2012 Rule 15.3

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 9.2

The initializer for an aggregate or union shall be enclosed in braces

Description

Rule Definition

The initializer for an aggregate or union shall be enclosed in braces.

Rationale

The rule applies to both objects and subobjects. For example, when initializing a structure that contains an array, the values assigned to the structure must be enclosed in braces. Within these braces, the values assigned to the array must be enclosed in another pair of braces.

Enclosing initializers in braces improves clarity of code that contains complex data structures such as multidimensional arrays and arrays of structures.

Tip To avoid nested braces for subobjects, use the syntax `{0}`, which sets all values to zero.

Message in Report

The initializer for an aggregate or union shall be enclosed in braces.

Examples

Initialization of Two-dimensional Arrays

```
void initialize(void) {  
    int x[4][2] = {{0,0},{1,0},{0,1},{1,1}}; /* Compliant */  
}
```

```
int y[4][2] = {{0},{1,0},{0,1},{1,1}}; /* Compliant */
int z[4][2] = {0}; /* Compliant */
int w[4][2] = {0,0,1,0,0,1,1,1}; /* Non-compliant */
}
```

In this example, the rule is not violated when:

- Initializers for each row of the array are enclosed in braces.
- The syntax {0} initializes all elements to zero.

The rule is violated when a separate pair of braces is not used to enclose the initializers for each row.

Check Information

Group: Initialization

Category: Required

AGC Category: Readability

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 9.3

Arrays shall not be partially initialized

Description

Rule Definition

Arrays shall not be partially initialized.

Rationale

Providing an explicit initialization for each array element makes it clear that every element has been considered.

Message in Report

Arrays shall not be partially initialized.

Examples

Partial and Complete Initializations

```
void func(void) {  
    int x[3] = {0,1,2};           /* Compliant */  
    int y[3] = {0,1};           /* Non-compliant */  
    int z[3] = {0};             /* Compliant - exception */  
    int a[30] = {[1] = 1, [15]=1}; /* Compliant - exception */  
    int b[30] = {{1} = 1, 1};   /* Non-compliant */  
    char c[20] = "Hello World"; /* Compliant - exception */  
}
```

In this example, the rule is not violated when each array element is explicitly initialized.

The rule is violated when some elements of the array are implicitly initialized. Exceptions include the following:

- The initializer has the form `{ 0 }`, which initializes all elements to zero.
- The array initializer consists *only* of designated initializers. Typically, you use this approach for sparse initialization.
- The array is initialized using a string literal.

Check Information

Group: Initialization

Category: Required

AGC Category: Readability

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 9.4

An element of an object shall not be initialized more than once

Description

Rule Definition

An element of an object shall not be initialized more than once.

Rationale

Designated initializers allow explicitly initializing elements of objects such as arrays in any order. However, using designated initializers, one can inadvertently initialize the same element twice and therefore overwrite the first initialization.

Message in Report

An element of an object shall not be initialized more than once.

Examples

Array Initialization Using Designated Initializers

```
void func(void) {
    int a[5] = {-2,-1,0,1,2};           /* Compliant */
    int b[5] = {[0]=-2, [1]=-1, [2]=0, [3]=1, [4]=2};
                                        /* Compliant */
    int c[5] = {[0]=-2, [1]=-1, [1]=0, [3]=1, [4]=2};
                                        /* Non-compliant */
}
```

In this example, the rule is violated when the array element `c[1]` is initialized twice using a designated initializer.

Structure Initialization Using Designated Initializers

```
struct myStruct {
    int a;
    int b;
    int c;
    int d;
};

void func(void) {
    struct myStruct struct1 = {-4,-2,2,4}; /* Compliant */
    struct myStruct struct2 = {.a=-4, .b=-2, .c=2, .d=4};
                                        /* Compliant */
    struct myStruct struct3 = {.a=-4, .b=-2, .b=2, .d=4};
                                        /* Non-compliant */
}
```

In this example, the rule is violated when `struct3.b` is initialized twice using a designated initializer.

Check Information

Group: Initialization

Category: Required

AGC Category: Required

Language: C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 9.5

Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly

Description

Rule Definition

Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

Rationale

If the size of an array is not specified explicitly, it is determined by the highest index of the elements that are initialized. When using long designated initializers, it might not be immediately apparent which element has the highest index.

Message in Report

Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

Examples

Using Designated Initializers Without Specifying Array Size

```
int a[5] = {[0]= 1, [2] = 1, [4]= 1, [1] = 1};           /* Compliant */
int b[] = {[0]= 1, [2] = 1, [4]= 1, [1] = 1};           /* Non-compliant */
int c[] = {[0]= 1, [1] = 1, [2]= 1, [3]=0, [4] = 1};    /* Non-compliant */

void display(int);

void main() {
    func(a,5);
}
```

```
    func(b, 5);  
    func(c, 5);  
}  
  
void func(int* arr, int size) {  
    for(int i=0; i<size; i++)  
        display(arr[i]);  
}
```

In this example, the rule is violated when the arrays `b` and `c` are initialized using designated initializers but the array size is not specified.

Check Information

Group: Initialization

Category: Required

AGC Category: Readability

Language: C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 10.1

Operands shall not be of an inappropriate essential type

Description

Rule Definition

Operands shall not be of an inappropriate essential type.

Rationale

An essential type category defines the essential type of an object or expression.

Essential type category	Standard types
Essentially Boolean	bool or _Bool (defined in <code>stdbool.h</code>) If you define a boolean type through a <code>typedef</code> , you must specify this type name before coding rules checking. For more information, see “Specify Effective Boolean Types”.
Essentially character	char
Essentially enum	named enum
Essentially signed	signed char, signed short, signed int, signed long, signed long long
Essentially unsigned	unsigned char, unsigned short, unsigned int, unsigned long, unsigned long long
Essentially floating	float, double, long double

For operands of some operators, you cannot use certain essential types. In the table below, each row represents an operator/operand combination. If the essential type column is not empty for that row, there is a MISRA restriction when using that type as the operand. The number in the table corresponds to the rationale list after the table.

Operation		Essential type category of arithmetic operand					
Operator	Operand	Boolean	character	enum	signed	unsigned	floating
[]	integer	3	4				1
+ (unary)		3	4	5			
- (unary)		3	4	5		8	
+ -	either	3		5			
* /	either	3	4	5			
%	either	3	4	5			1
< > <= >=	either	3					
== !=	either						
! &&	any		2	2	2	2	2
<< >>	left	3	4	5,6	6		1
<< >>	right	3	4	7	7		1
~ & ^	any	3	4	5,6	6		1
?:	1st		2	2	2	2	2
?:	2nd and 3rd						

- 1 An expression of essentially floating type for these operands is a constraint violation.
- 2 When an operand is interpreted as a Boolean value, use an expression of essentially Boolean type.
- 3 When an operand is interpreted as a numeric value, do not use an operand of essentially Boolean type.
- 4 When an operand is interpreted as a numeric value, do not use an operand of essentially character type. The numeric values of character data are implementation-defined.
- 5 In an arithmetic operation, do not use an operand of essentially enum type. An enum object uses an implementation-defined integer type. An operation involving an enum object can therefore yield a result with an unexpected type.
- 6 Perform only shift and bitwise operations on operands of essentially unsigned type. When you use shift and bitwise operations on essentially signed types, the resulting numeric value is implementation-defined.

- 7 To avoid undefined behavior on negative shifts, use an essentially unsigned right-hand operand.
- 8 For the unary minus operator, do not use an operand of essentially unsigned type. The implemented size of int determines the signedness of the result.

Message in Report

The *operand_name* operand of the *operator_name* operator is of an inappropriate essential type category *category_name*.

Examples

Violation of Rule 10.1, Rationale 2: Inappropriate Operand Types for Operators That Take Essentially Boolean Operands

```
typedef unsigned char boolean;

extern float f32a;
extern char cha;
extern signed char s8a;
extern unsigned char u8a;
enum enuma { a1, a2, a3 } ena;

extern boolean bla, blb, rbla;

void foo(void) {

    rbla = cha && bla;           /* Non-compliant: cha is essentially char */
    enb = ena ? a1 : a2;        /* Non-compliant: ena is essentially enum */
    rbla = s8a && bla;           /* Non-compliant: s8a is essentially signed char */
    ena = u8a ? a1 : a2;        /* Non-compliant: u8a is essentially unsigned char */
    rbla = f32a && bla;         /* Non-compliant: f32a is essentially float */

    rbla = bla && blb;           /* Compliant */
    ru8a = bla ? u8a : u8b;     /* Compliant */

}
```

In the noncompliant examples, rule 10.1 is violated because:

- The operator `&&` expects only essentially Boolean operands. However, at least one of the operands used has a different type.
- The first operand of `?:` is expected to be essentially Boolean. However, a different operand type is used.

Note For Polyspace to detect the rule violation, you must define the type name `boolean` as an effective Boolean type. For more information, see “Specify Effective Boolean Types”.

Violation of Rule 10.1, Rationale 3: Inappropriate Boolean Operands

```
typedef unsigned char boolean;

enum enuma { a1, a2, a3 } ena;
enum { K1 = 1, K2 = 2 }; /* Essentially signed */
extern char cha, chb;
extern boolean bla, blb, rbla;
extern signed char rs8a, s8a;

void foo(void) {

    rbla = bla * blb; /* Non-compliant - Boolean used as a numeric value */
    rbla = bla > blb; /* Non-compliant - Boolean used as a numeric value */

    rbla = bla && blb; /* Compliant */
    rbla = cha > chb; /* Compliant */
    rbla = ena > a1; /* Compliant */
    rbla = u8a > 0U; /* Compliant */
    rs8a = K1 * s8a; /* Compliant - K1 obtained from anonymous enum */

}
```

In the noncompliant examples, rule 10.1 is violated because the operators `*` and `>` do not expect essentially Boolean operands. However, the operands used here are essentially Boolean.

Note For Polyspace to detect the rule violation, you must define the type name `boolean` as an effective Boolean type. For more information, see “Specify Effective Boolean Types”.

Violation of Rule 10.1, Rationale 4: Inappropriate Character Operands

```
extern char rcha, cha, chb;
extern unsigned char ru8a, u8a;

void foo(void) {

    rcha = cha & chb;          /* Non-compliant - char type used as a numeric value */
    rcha = cha << 1;          /* Non-compliant - char type used as a numeric value */

    ru8a = u8a & 2U;         /* Compliant */
    ru8a = u8a << 2U;       /* Compliant */

}
```

In the noncompliant examples, rule 10.1 is violated because the operators `&` and `<<` do not expect essentially character operands. However, at least one of the operands used here has essentially character type.

Violation of Rule 10.1, Rationale 5: Inappropriate Enum Operands

```
typedef unsigned char boolean;

enum enuma { a1, a2, a3 } rena, ena, enb;

void foo(void) {

    ena--;                    /* Non-Compliant - arithmetic operation with enum type*/
    rena = ena * a1;         /* Non-Compliant - arithmetic operation with enum type*/
    ena += a1;               /* Non-Compliant - arithmetic operation with enum type*/

}
```

In the noncompliant examples, rule 10.1 is violated because the arithmetic operators `--`, `*` and `+=` do not expect essentially enum operands. However, at least one of the operands used here has essentially enum type.

Violation of Rule 10.1, Rationale 6: Inappropriate Signed Operand for Bitwise Operations

```
extern signed char s8a;
extern unsigned char ru8a, u8a;

void foo(void) {

    ru8a = s8a & 2;          /* Non-compliant - bitwise operation on signed type */
    ru8a = 2 << 3U;        /* Non-compliant - shift operation on signed type */

    ru8a = u8a << 2U;      /* Compliant */

}
```

In the noncompliant examples, rule 10.1 is violated because the `&` and `<<` operations must not be performed on essentially signed operands. However, the operands used here are signed.

Violation of Rule 10.1, Rationale 7: Inappropriate Signed Right Operand for Shift Operations

```
extern signed char s8a;
extern unsigned char ru8a, u8a;

void foo(void) {

    ru8a = u8a << s8a;      /* Non-compliant - shift magnitude uses signed type */
    ru8a = u8a << -1;      /* Non-compliant - shift magnitude uses signed type */

    ru8a = u8a << 2U;      /* Compliant */
    ru8a = u8a << 1;      /* Compliant - exception */

}
```

In the noncompliant examples, rule 10.1 is violated because the operation `<<` does not expect an essentially signed right operand. However, the right operands used here are signed.

Check Information

Group: The Essential Type Model

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 10.2

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 10.2

Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations

Description

Rule Definition

Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.

Rationale

Essentially character type expressions are `char` variables. Do not use character data arithmetically because the data does not represent numeric values.

For information on essential types, see MISRA C:2012 Rule 10.1.

Message in Report

- The *operand_name* operand of the + operator applied to an expression of essentially character type shall have essentially signed or unsigned type.
- The right operand of the - operator applied to an expression of essentially character type shall have essentially signed or unsigned or character type.
- The left operand of the - operator shall have essentially character type if the right operand has essentially character type.

Check Information

Group: The Essential Type Model

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 10.1

Topics

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 10.3

The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category

Description

Rule Definition

The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.

Rationale

The use of implicit conversions between types can lead to unintended results, including possible loss of value, sign, or precision.

For information on essential types, see MISRA C:2012 Rule 10.1.

Message in Report

- The expression is assigned to an object with a different essential type category.
- The expression is assigned to an object with a narrower essential type.

Check Information

Group: The Essential Type Model

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 10.4 | MISRA C:2012 Rule 10.5 | MISRA C:2012 Rule 10.6

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 10.4

Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category

Description

Rule Definition

Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.

Rationale

The use of implicit conversions between types can lead to unintended results, including possible loss of value, sign, or precision.

For information on essential types, see MISRA C:2012 Rule 10.1.

Polyspace Specification

Polyspace does not produce a violation of this rule:

- If one of the operands is the constant zero.
- If one of the operands is a signed constant and the other operand is unsigned, and the signed constant has the same representation as its unsigned equivalent.

For instance, the statement `u8b = u8a + 3;`, where `u8a` and `u8b` are unsigned `char` variables, does not violate the rule because the constants `3` and `3U` have the same representation.

Message in Report

Operands of `operator_name` operator shall have the same essential type category.

Check Information

Group: The Essential Type Model

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 10.3 | MISRA C:2012 Rule 10.7

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 10.5

The value of an expression should not be cast to an inappropriate essential type

Description

Rule Definition

The value of an expression should not be cast to an inappropriate essential type.

Rationale

Converting Between Variable Types

		From					
		Boolean	character	enum	signed	unsigned	floating
To	Boolean		Avoid	Avoid	Avoid	Avoid	Avoid
	character	Avoid					Avoid
	enum	Avoid	Avoid	Avoid	Avoid	Avoid	Avoid
	signed	Avoid					
	unsigned	Avoid					
	floating	Avoid	Avoid				

Some inappropriate explicit casts are:

- In C99, the result of a cast of assignment to `_Bool` is always 0 or 1. This result is not necessarily the case when casting to another type which is defined as essentially Boolean.
- A cast to an essential enum type may result in a value that does not lie within the set of enumeration constants for that type.
- A cast from essential Boolean to any other type is unlikely to be meaningful.
- Converting between floating and character types is not meaningful as there is no precise mapping between the two representations.

Some acceptable explicit casts are:

- To change the type in which a subsequent arithmetic operation is performed.
- To truncate a value deliberately.
- To make a type conversion explicit in the interests of clarity.

For more information on essential types, see MISRA C:2012 Rule 10.1.

Message in Report

The value of an expression should not be cast to an inappropriate essential type.

Check Information

Group: The Essential Type Model

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 10.3 | MISRA C:2012 Rule 10.8

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 10.6

The value of a composite expression shall not be assigned to an object with wider essential type

Description

Rule Definition

The value of a composite expression shall not be assigned to an object with wider essential type.

Rationale

A *composite expression* is a nonconstant expression using a composite operator. In the Essential Type Model, composite operators are:

- Multiplicative (*, /, %)
- Additive (binary +, binary -)
- Bitwise (&, |, ^)
- Shift (<<, >>)
- Conditional (?, :)

If you assign the result of a composite expression to a larger type, the implicit conversion can result in loss of value, sign, precision, or layout.

For information on essential types, see MISRA C:2012 Rule 10.1.

Message in Report

The composite expression is assigned to an object with a wider essential type.

Check Information

Group: The Essential Type Model

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 10.3 | MISRA C:2012 Rule 10.7

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 10.7

If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type

Description

Rule Definition

If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed, then the other operand shall not have wider essential type.

Rationale

A *composite expression* is a nonconstant expression using a composite operator. In the Essential Type Model, composite operators are:

- Multiplicative (*, /, %)
- Additive (binary +, binary -)
- Bitwise (&, |, ^)
- Shift (<<, >>)
- Conditional (?, :)

Restricting implicit conversion on composite expressions mean that sequences of arithmetic operations within expressions must use the same essential type. This restriction reduces confusion and avoids loss of value, sign, precision, or layout. However, this rule does not imply that all operands in an expression are of the same essential type.

For information on essential types, see MISRA C:2012 Rule 10.1.

Message in Report

- The right operand shall not have wider essential type than the left operand which is a composite expression.
- The left operand shall not have wider essential type than the right operand which is a composite expression.

Check Information

Group: The Essential Type Model

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 10.8

The value of a composite expression shall not be cast to a different essential type category or a wider essential type

Description

Rule Definition

The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Rationale

A *composite expression* is a non-constant expression using a composite operator. In the Essential Type Model, composite operators are:

- Multiplicative (*, /, %)
- Additive (binary +, binary -)
- Bitwise (&, |, ^)
- Shift (<<, >>)
- Conditional (?, :)

Casting to a wider type is not permitted because the result may vary between implementations. Consider this expression:

```
(uint32_t) (u16a +u16b);
```

On a 16-bit machine the addition is performed in 16 bits. The result is wrapped before it is cast to 32 bits. On a 32-bit machine, the addition takes place in 32 bits and preserves high-order bits that are lost on a 16-bit machine. Casting to a narrower type with the same essential type category is acceptable as the explicit truncation of the results always leads to the same loss of information.

For information on essential types, see MISRA C:2012 Rule 10.1.

Message in Report

- The value of a composite expression shall not be cast to a different essential type category.
- The value of a composite expression shall not be cast to a wider essential type.

Examples

Casting to Different or Wider Essential Type

```
extern unsigned short ru16a, u16a, u16b;
extern unsigned int  u32a, ru32a;
extern signed int    s32a, s32b;

void foo(void)
{
    ru16a = (unsigned short) (u32a + u32a); /* Compliant */
    ru16a += (unsigned short) s32a + s32b;
                                   /* Noncompliant - different essential type */
    ru16a += (unsigned short) s32a; /* Compliant - s32a is not composite */
    ru32a = (unsigned int) (u16a + u16b); /* Noncompliant - wider essential type */
}
```

In this example, rule 10.8 is violated in the following cases:

- s32a and s32b are essentially signed variables. However, the result (s32a + s32b) is cast to an essentially unsigned type.
- u16a and u16b are essentially unsigned short variables. However, the result (s32a + s32b) is cast to a wider essential type, unsigned int.

Check Information

Group: The Essential Type Model

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 10.5

Topics

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 11.1

Conversions shall not be performed between a pointer to a function and any other type

Description

Rule Definition

Conversions shall not be performed between a pointer to a function and any other type.

Rationale

The rule forbids the following two conversions:

- Conversion from a function pointer to any other type. This conversion causes undefined behavior.
- Conversion from a function pointer to another function pointer, if the function pointers have different argument and return types.

The conversion is forbidden because calling a function through a pointer with incompatible type results in undefined behavior.

Polyspace Specification

Polyspace considers both explicit and implicit casts when checking this rule. However, casts from `NULL` or `(void*)0` do not violate this rule.

Message in Report

Conversions shall not be performed between a pointer to a function and any other type.

Examples

Cast between two function pointers

```
typedef void (*fp16) (short n);
typedef void (*fp32) (int n);

#include <stdlib.h>                                /* To obtain macro NULL */

void func(void) { /* Exception 1 - Can convert a null pointer
                  * constant into a pointer to a function */
    fp16 fp1 = NULL;                               /* Compliant - exception */
    fp16 fp2 = (fp16) fp1;                         /* Compliant */
    fp32 fp3 = (fp32) fp1;                         /* Non-compliant */
    if (fp2 != NULL) {}                           /* Compliant - exception */
    fp16 fp4 = (fp16) 0x8000;                      /* Non-compliant - integer to
                                                    * function pointer */
}
```

In this example, the rule is violated when:

- The pointer `fp1` of type `fp16` is cast to type `fp32`. The function pointer types `fp16` and `fp32` have different argument types.
- An integer is cast to type `fp16`.

The rule is not violated when function pointers `fp1` and `fp2` are cast to `NULL`.

Check Information

Group: Pointer Type Conversions

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.2

Conversions shall not be performed between a pointer to an incomplete type and any other type

Description

Rule Definition

Conversions shall not be performed between a pointer to an incomplete type and any other type.

Rationale

An incomplete type is a type that does not contain sufficient information to determine its size. For example, the statement `struct s;` describes an incomplete type because the fields of `s` are not defined. The size of a variable of type `s` cannot be determined.

Conversions to or from a pointer to an incomplete type result in undefined behavior. Typically, a pointer to an incomplete type is used to hide the full representation of an object. This encapsulation is broken if another pointer is implicitly or explicitly cast to such a pointer.

Message in Report

Conversions shall not be performed between a pointer to an incomplete type and any other type.

Examples

Casts from incomplete type

```
struct s *sp;  
struct t *tp;
```



```
short *ip;
struct ct *ctp1;
struct ct *ctp2;

void foo(void) {

    ip = (short *) sp;           /* Non-compliant */
    sp = (struct s *) 1234;     /* Non-compliant */
    tp = (struct t *) sp;      /* Non-compliant */
    ctp1 = (struct ct *) ctp2; /* Compliant */

    /* You can convert a null pointer constant to
     * a pointer to an incomplete type */
    sp = NULL;                 /* Compliant - exception */

    /* A pointer to an incomplete type may be converted into void */
    struct s *f(void);
    (void) f();                /* Compliant - exception */

}
```

In this example, types `s`, `t` and `ct` are incomplete. The rule is violated when:

- The variable `sp` with an incomplete type is cast to a basic type.
- The variable `sp` with an incomplete type is cast to a different incomplete type `t`.

The rule is not violated when:

- The variable `ctp2` with an incomplete type is cast to the same incomplete type.
- The `NULL` pointer is cast to the variable `sp` with an incomplete type.
- The return value of `f` with incomplete type is cast to `void`.

Check Information

Group: Pointer Type Conversions

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 11.5

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.3

A cast shall not be performed between a pointer to object type and a pointer to a different object type

Description

Rule Definition

A cast shall not be performed between a pointer to object type and a pointer to a different object type.

Rationale

If a pointer to an object is cast into a pointer to a different object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.

Even if the conversion produces a pointer that is correctly aligned, the behavior can be undefined if the pointer is used to access an object.

Exception: You can convert a pointer to object type into a pointer to one of the following types:

- char
- signed char
- unsigned char

Message in Report

A cast shall not be performed between a pointer to object type and a pointer to a different object type.

Examples

Noncompliant: Cast to Pointer Pointing to Object of Wider Type

```
signed char *p1;
unsigned int *p2;

void foo(void){
    p2 = ( unsigned int * ) p1;    /* Non-compliant */
}
```

In this example, `p1` can point to a `signed char` object. However, `p1` is cast to a pointer that points to an object of wider type, `unsigned int`.

Noncompliant: Cast to Pointer Pointing to Object of Narrower Type

```
extern unsigned int read_value ( void );
extern void display ( unsigned int n );

void foo ( void ){
    unsigned int u = read_value ( );
    unsigned short *hi_p = ( unsigned short * ) &u;    /* Non-compliant */
    *hi_p = 0;
    display ( u );
}
```

In this example, `u` is an `unsigned int` variable. `&u` is cast to a pointer that points to an object of narrower type, `unsigned short`.

On a big-endian machine, the statement `*hi_p = 0` attempts to clear the high bits of the memory location that `&u` points to. But, from the result of `display(u)`, you might find that the high bits have not been cleared.

Compliant: Cast Adding a Type Qualifier

```
const short *p;
const volatile short *q;
void foo (void){
    q = ( const volatile short * ) p;    /* Compliant */
}
```

In this example, both `p` and `q` can point to `short` objects. The cast between them adds a `volatile` qualifier only and is therefore compliant.

Check Information

Group: Pointer Type Conversions

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 11.4 | MISRA C:2012 Rule 11.5 | MISRA C:2012 Rule 11.8

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.4

A conversion should not be performed between a pointer to object and an integer type

Description

Rule Definition

A conversion should not be performed between a pointer to object and an integer type.

Rationale

Conversion between integers and pointers can cause errors or undefined behavior.

- If an integer is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.
- If a pointer is cast to an integer, the resulting value can be outside the allowed range for the integer type.

Polyspace Specification

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Message in Report

A conversion should not be performed between a pointer to object and an integer type.

Examples

Casts between pointer and integer

```
#include <stdbool.h>

typedef unsigned char      uint8_t;
```

```

typedef      char      char_t;
typedef unsigned short uint16_t;
typedef signed  int    int32_t;

typedef _Bool bool_t;
uint8_t *PORTA = (uint8_t *) 0x0002;          /* Non-compliant */

void foo(void) {

    char_t c = 1;
    char_t *pc = &c;                          /* Compliant */

    uint16_t ui16 = 7U;
    uint16_t *pui16 = &ui16;                  /* Compliant */
    pui16 = (uint16_t *) ui16;                /* Non-compliant */

    uint16_t *p;
    int32_t addr = (int32_t) p;                /* Non-compliant */
    bool_t b = (bool_t) p;                    /* Non-compliant */
    enum etag { A, B } e = ( enum etag ) p;   /* Non-compliant */
}

```

In this example, the rule is violated when:

- The integer 0x0002 is cast to a pointer.

If the integer defines an absolute address, it is more common to assign the address to a pointer in a header file. To avoid the assignment being flagged, you can then exclude headers files from coding rules checking. For more information, see `Do not generate results for (-do-not-generate-results-for)`.

- The pointer `p` is cast to integer types such as `int32_t`, `bool_t` or `enum etag`.

The rule is not violated when the address `&ui16` is assigned to a pointer.

Check Information

Group: Pointer Type Conversions

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 11.3 | MISRA C:2012 Rule 11.7 | MISRA C:2012 Rule 11.9

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.5

A conversion should not be performed from pointer to void into pointer to object

Description

Rule Definition

A conversion should not be performed from pointer to void into pointer to object.

Rationale

If a pointer to `void` is cast into a pointer to an object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior. However, such a cast can sometimes be necessary, for example, when using memory allocation functions.

Polyspace Specification

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Message in Report

A conversion should not be performed from pointer to void into pointer to object.

Examples

Cast from Pointer to `void`

```
void foo(void) {  
    unsigned int  u32a = 0;  
    unsigned int  *p32 = &u32a;  
    void          *p;  
    unsigned int  *p16;
```

```
p    = p32;                /* Compliant - pointer to uint32_t
                          *                into pointer to void */
p16 = p;                  /* Non-compliant */

p    = (void *) p16;      /* Compliant */
p32 = (unsigned int *) p; /* Non-compliant */
}
```

In this example, the rule is violated when the pointer `p` of type `void*` is cast to pointers to other types.

The rule is not violated when `p16` and `p32`, which are pointers to non-void types, are cast to `void*`.

Check Information

Group: Pointer Type Conversions

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 11.2 | MISRA C:2012 Rule 11.3

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.6

A cast shall not be performed between pointer to void and an arithmetic type

Description

Rule Definition

A cast shall not be performed between pointer to void and an arithmetic type.

Rationale

Conversion between integer types and pointers to `void` can cause errors or undefined behavior.

- If an integer type is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.
- If a pointer is cast to an arithmetic type, the resulting value can be outside the allowed range for the type.

Conversion between non-integer arithmetic types and pointers to `void` is undefined.

Polyspace Specification

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Message in Report

A cast shall not be performed between pointer to void and an arithmetic type.

Examples

Casts Between Pointer to `void` and Arithmetic Types

```
void foo(void) {  
  
    void          *p;  
    unsigned int  u;  
    unsigned short r;  
  
    p = (void *) 0x1234u;           /* Non-compliant - undefined */  
    u = (unsigned int) p;          /* Non-compliant - undefined */  
  
    p = (void *) 0;                /* Compliant - Exception */  
  
}
```

In this example, `p` is a pointer to `void`. The rule is violated when:

- An integer value is cast to `p`.
- `p` is cast to an `unsigned int` type.

The rule is not violated if an integer constant with value 0 is cast to a pointer to `void`.

Check Information

Group: Pointer Type Conversions

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.7

A cast shall not be performed between pointer to object and a non-integer arithmetic type

Description

Rule Definition

A cast shall not be performed between pointer to object and a non-integer arithmetic type.

Rationale

This rule covers types that are essentially Boolean, character, enum or floating.

- If an essentially Boolean, character or enum variable is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior. If a pointer is cast to one of those types, the resulting value can be outside the allowed range for the type.
- Casts to or from a pointer to a floating type results in undefined behavior.

Message in Report

A cast shall not be performed between pointer to object and a non-integer arithmetic type.

Examples

Casts from Pointer to Non-Integer Arithmetic Types

```
int foo(void) {  
    short *p;  
    float  f;  
    long  *l;
```

```
f = (float) p;          /* Non-compliant */
p = (short *) f;       /* Non-compliant */

l = (long *) p;        /* Compliant */
}
```

In this example, the rule is violated when:

- The pointer `p` is cast to `float`.
- A `float` variable is cast to a pointer to `short`.

The rule is not violated when the pointer `p` is cast to `long*`.

Check Information

Group: Pointer Type Conversions

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 11.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.8

A cast shall not remove any `const` or `volatile` qualification from the type pointed to by a pointer

Description

Rule Definition

A cast shall not remove any `const` or `volatile` qualification from the type pointed to by a pointer.

Rationale

This rule forbids:

- Casts from a pointer to a `const` object to a pointer that does not point to a `const` object.
- Casts from a pointer to a `volatile` object to a pointer that does not point to a `volatile` object.

Such casts violate type qualification. For example, the `const` qualifier indicates the read-only status of an object. If a cast removes the qualifier, the object is no longer read-only.

Polyspace Specification

Polyspace flags both implicit and explicit conversions that violate this rule.

Message in Report

A cast shall not remove any `const` or `volatile` qualification from the type pointed to by a pointer.

Examples

Casts That Remove Qualifiers

```
void foo(void) {

    /* Cast on simple type */
    unsigned short      x;
    unsigned short * const  cpi = &x; /* const pointer */
    unsigned short * const  *pcpi; /* pointer to const pointer */
    unsigned short **ppi;

    const unsigned short  *pci; /* pointer to const */
    volatile unsigned short *pvi; /* pointer to volatile */
    unsigned short      *pi;

    pi = cpi; /* Compliant - no cast required */
    pi = (unsigned short *) pci; /* Non-compliant */
    pi = (unsigned short *) pvi; /* Non-compliant */
    ppi = (unsigned short **)pcpi; /* Non-compliant */
}
```

In this example:

- The variables `pci` and `pcpi` have the `const` qualifier in their type. The rule is violated when the variables are cast to types that do not have the `const` qualifier.
- The variable `pvi` has a `volatile` qualifier in its type. The rule is violated when the variable is cast to a type that does not have the `volatile` qualifier.

Even though `cpi` has a `const` qualifier in its type, the rule is not violated in the statement `p=cpi;`. The assignment does not cause a type conversion because both `p` and `cpi` have type `unsigned short`.

Check Information

Group: Pointer Type Conversions

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 11.3

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.9

The macro `NULL` shall be the only permitted form of integer null pointer constant

Description

Rule Definition

The macro `NULL` shall be the only permitted form of integer null pointer constant.

Rationale

The following expressions require the use of a null pointer constant:

- Assignment to a pointer
- The `==` or `!=` operation, where one operand is a pointer
- The `?:` operation, where one of the operands on either side of `:` is a pointer

Using `NULL` rather than `0` makes it clear that a null pointer constant was intended.

Message in Report

The macro `NULL` shall be the only permitted form of integer null pointer constant.

Examples

Using `0` for Pointer Assignments and Comparisons

```
void main(void) {  
  
    int *p1 = 0;           /* Non-compliant */  
    int *p2 = ( void * ) 0; /* Compliant      */  
  
#define MY_NULL_1 0
```

```
#define MY_NULL_2 ( void * ) 0

    if ( p1 == MY_NULL_1 )    /* Non-compliant */
    { }
    if ( p2 == MY_NULL_2 )    /* Compliant    */
    { }

}
```

In this example, the rule is violated when the constant 0 is used instead of (void*) 0 for pointer assignments and comparisons.

Check Information

Group: Pointer Type Conversions

Category: Required

AGC Category: Readability

Language: C90, C99

See Also

MISRA C:2012 Rule 11.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 12.1

The precedence of operators within expressions should be made explicit

Description

Rule Definition

The precedence of operators within expressions should be made explicit.

Rationale

The C language has a large number of operators and their precedence is not intuitive. Inexperienced programmers can easily make mistakes. Remove any ambiguity by using parentheses to explicitly define operator precedence.

The following table list the MISRA C definition of operator precedence for this rule.

Description	Operator and Operand	Precedence
Primary	identifier, constant, string literal, (expression)	16
Postfix	[] () (function call) . -> ++(post-increment) --(post-decrement) () { } (C99: compound literals)	15
Unary	++(post-increment) --(post-decrement) & * + - ~ ! sizeof defined (preprocessor)	14
Cast	()	13
Multiplicative	* / %	12
Additive	+ -	11
Bitwise shift	<< >>	10
Relational	<> <= >=	9
Equality	== !=	8
Bitwise AND	&	7

Description	Operator and Operand	Precedence
Bitwise XOR	^	6
Bitwise OR		5
Logical AND	&&	4
Logical OR		3
Conditional	?:	2
Assignment	= *= /= += -= <<= >>= &= ^= =	1
Comma	,	0

Message in Report

Operand of logical %s is not a primary expression. The precedence of operators within expressions should be made explicit.

Examples

Ambiguous Precedence in Multi-Operation Expressions

```
int a, b, c, d, x;

void foo(void) {
    x = sizeof a + b;           /* Non-compliant - MISRA-12.1 */

    x = a == b ? a : a - b;     /* Non-compliant - MISRA-12.1 */

    x = a << b + c ;           /* Non-compliant - MISRA-12.1 */

    if (a || b && c) { }        /* Non-compliant - MISRA-12.1 */

    if ( (a>x) && (b>x) || (c>x) ) { } /* Non-compliant - MISRA-12.1 */
}
```

This example shows various violations of MISRA rule 12.1. In each violation, if you do not know the order of operations, the code could execute unexpectedly.

To comply with this MISRA rule, add parentheses around individual operations in the expressions. One possible solution is shown here.

```
int a, b, c, d, x;

void foo(void) {
    x = sizeof(a) + b;

    x = ( a == b ) ? a : ( a - b );

    x = a << ( b + c );

    if ( ( a || b ) && c ) { }

    if ( ((a>x) && (b>x)) || (c>x) ) { }
}
```

Ambiguous Precedence In Preprocessing Expressions

```
# if defined X && X + Y > Z    /* Non-compliant - MISRA-12.1 */
# endif

# if ! defined X && defined Y /* Non-compliant - MISRA-12.1 */
# endif
```

In this example, two violations of MISRA rule 12.1 are shown in preprocessing code. In each violation, if you do not know the correct order of operations, the results can be unexpected and cause problems.

To comply with this MISRA rule, add parentheses around individual operations in the expressions. One possible solution is shown here.

```
# if defined (X) && ( (X + Y) > Z )
# endif

# if ! defined (X) && defined (Y)
# endif
```

Compliant Expressions Without Parentheses

```
int a, b, c, x;
struct {int a; } s, *ps, *pp[2];
```

```
void foo(void) {
    ps = &s

    pp[i]-> a;          /* Compliant - no need to write (pp[i])->a */
    *ps++;             /* Compliant - no need to write *( p++ ) */

    x = f ( a + b, c ); /* Compliant - no need to write f ( (a+b),c) */

    x = a, b;          /* Compliant - parsed as ( x = a ), b */

    if ( a && b && c ){ /* Compliant - all operators have
                        * the same precedence */
    }
}
```

In this example, the expressions shown have multiple operations. However, these expressions are compliant because operator precedence is already clear.

Check Information

Group: Expressions

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 12.2 | MISRA C:2012 Rule 12.3 | MISRA C:2012 Rule 12.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 12.2

The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand

Description

Rule Definition

The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.

Rationale

Consider the following statement:

```
var = abc << num;
```

If `abc` is a 16-bit integer, then `num` must be in the range 0–15, (nonnegative and less than 16). If `num` is negative or greater than 16, then the shift behavior is undefined.

Polyspace Specification

In Polyspace, the numbers that are manipulated in preprocessing directives are 64 bits wide. The valid shift range is between 0 and 63. When bitfields are within a complex expression, Polyspace extends this check onto the bitfield field width or the width of the base type.

Message in Report

- Shift amount is bigger than *size*.
- Shift amount is negative.
- The right operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left operand.

Check Information

Group: Expressions

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 12.1

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 12.3

The comma operator should not be used

Description

Rule Definition

The comma operator should not be used.

Rationale

The comma operator can be detrimental to readability. You can often write the same code in another form.

Message in Report

The comma operator should not be used.

Examples

Comma Usage in C Code

```
typedef signed int abc, xyz, jkl;

static void func1 ( abc, xyz, jkl );           /* Compliant - case 1 */

int foo(void)
{
    volatile int rd = 1;                       /* Compliant - case 2*/
    int var=0, foo=0, k=0, n=2, p, t[10];      /* Compliant - case 3*/

    int abc = 0, xyz = abc + 1;                /* Compliant - case 4*/
    int jkl = ( abc + xyz, abc + xyz );        /* Not compliant - case 1*/
}
```

```

var = 1, foo += var, kkk = 3;          /* Not compliant - case 2*/
var = (kkk = 1, foo = 2);           /* Not compliant - case 3*/

for ( var = 0, ptr = &t[ 0 ]; var < num; ++var, ++ptr){}
                                     /* Not compliant - case 4*/

if ((abc,xyz)<0) { return 1; }       /* Not compliant - case 5*/
}

```

In this example, the code shows various uses of commas in C code.

Case	Reason for noncompliance
1	When reading the code, it is not immediately obvious what <code>ijkl</code> is initialized to. For example, you could infer that <code>ijkl</code> has a value <code>abc+xyz</code> , <code>(abc+xyz) * (abc+xyz)</code> , <code>f((abc+xyz) , (abc+xyz))</code> , and so on.
2	When reading the code, it is not immediately obvious whether <code>foo</code> has a value 0 or 1 after the statement.
3	When reading the code, it is not immediately obvious what value is assigned to <code>var</code> .
4	When reading the code, it is not immediately obvious which values control the <code>for</code> loop.
5	When reading the code, it is not immediately obvious whether the <code>if</code> statement depends on <code>abc</code> , <code>xyz</code> , or both.

Case	Reason for compliance
1	Using commas to call functions with variables is allowed.
2	Comma operator is not used.
3 & 4	When using the comma for initialization, the variables and their values are immediately obvious.

Check Information

Group: Expressions

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 12.1

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 12.4

Evaluation of constant expressions should not lead to unsigned integer wrap-around

Description

Rule Definition

Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Rationale

Unsigned integer expressions do not strictly overflow, but instead wraparound. Although there may be good reasons to use modulo arithmetic at run time, intentional use at compile time is less likely.

Message in Report

Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Check Information

Group: Expressions

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 12.1

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 13.1

Initializer lists shall not contain persistent side effects

Description

Rule Definition

Initializer lists shall not contain persistent side effects.

Rationale

C99 permits initializer lists with expressions that can be evaluated only at run-time. However, the order in which elements of the list are evaluated is not defined. If one element of the list modifies the value of a variable which is used in another element, the ambiguity in order of evaluation causes undefined values. Therefore, this rule requires that expressions occurring in an initializer list cannot modify the variables used in them.

Message in Report

Initializer lists shall not contain persistent side effects.

Examples

Initializers with Persistent Side Effect

```
volatile int v;  
int x;  
int y;  
  
void f(void) {  
    int arr[2] = {x+y,x-y}; /* Compliant */  
    int arr2[2] = {v,0}; /* Non-compliant */  
    int arr3[2] = {x++,y}; /* Non-compliant */  
}
```


In this example, the rule is not violated in the first initialization because the initializer does not modify either `x` or `y`. The rule is violated in the other initializations.

- In the second initialization, because `v` is volatile, the initializer can modify `v`.
- In the third initialization, the initializer modifies the variable `x`.

Check Information

Group: Side Effects

Category: Required

AGC Category: Required

Language: C99

See Also

MISRA C:2012 Rule 13.2

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 13.2

The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders

Description

Rule Definition

The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.

Rationale

An expression can have different values under the following conditions:

- The same variable is modified more than once in the expression, or is both read and written.
- The expression allows more than one order of evaluation.

Therefore, this rule forbids expressions where a variable is modified more than once and can cause different results under different orders of evaluation.

Message in Report

The value of 'XX' depends on the order of evaluation. The value of volatile 'XX' depends on the order of evaluation because of multiple accesses.

Examples

Variable Modified More Than Once in Expression

```
int a[10], b[10];  
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])
```

```

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);          /* Compliant */
    COPY_ELEMENT (i++);       /* Non-compliant */
}

```

In this example, the rule is violated by the statement `COPY_ELEMENT(i++)` because `i++` occurs twice and the order of evaluation of the two expressions is unspecified.

Variable Modified and Used in Multiple Function Arguments

```

void f (unsigned int param1, unsigned int param2) {}

void main () {
    unsigned int i=0;
    f ( i++, i );          /* Non-compliant */
}

```

In this example, the rule is violated because it is unspecified whether the operation `i++` occurs before or after the second argument is passed to `f`. The call `f(i++, i)` can translate to either `f(0, 0)` or `f(0, 1)`.

Check Information

Group: Side Effects

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Dir 4.9 | MISRA C:2012 Rule 13.1 | MISRA C:2012 Rule 13.3
| MISRA C:2012 Rule 13.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 13.3

A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator

Description

Rule Definition

A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.

Rationale

The rule is violated if the following happens in the same line of code:

- The increment or decrement operator acts on a variable.
- Another read or write operation is performed on the variable.

For example, the line `y=x++` violates this rule. The ++ and = operator both act on `x`.

Although the operator precedence rules determine the order of evaluation, placing the ++ and another operator in the same line can reduce the readability of the code.

Message in Report

A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.

Examples

Increment Operator Used in Expression with Other Side Effects

```
int input(void);
int choice(void);
int operation(int, int);

int func() {
    int x = input(), y = input(), res;
    int ch = choice();
    if (choice == -1)
        return(x++);
    if (choice == 0) {
        res = x++ + y++;
        return(res);          /* Non-compliant */
    }
    else if (choice == 1) {
        x++;                  /* Compliant */
        y++;                  /* Compliant */
        return (x+y);
    }
    else {
        res = operation(x++,y);
        return(res);        /* Non-compliant */
    }
}
```

In this example, the rule is violated when the expressions containing the ++ operator have side effects other than that caused by the operator. For example, in the expression `return(x++)`, the other side-effect is the `return` operation.

Check Information

Group: Side Effects

Category: Advisory

AGC Category: Readability

Language: C90, C99

See Also

MISRA C:2012 Rule 13.2

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 13.4

The result of an assignment operator should not be used

Description

Rule Definition

The result of an assignment operator should not be used.

Rationale

The rule is violated if the following happens in the same line of code:

- The assignment operator acts on a variable.
- Another read or operation is performed on the result of the assignment.

For example, the line `a[x]=a[x=y];` violates this rule. The `[]` operator acts on the result of the assignment `x=y`.

Message in Report

The result of an assignment operator should not be used.

Examples

Result of Assignment Used

```
int x, y, b, c, d;
int a[10];
unsigned int bool_var, false=0, true=1;

int foo(void) {
    x = y;           /* Compliant - x is not used */
}
```



```

a[x] = a[x = y]; /* Non-compliant - Value of x=y is used */

if ( bool_var = false ) {}
    /* Non-compliant - bool_var=false is used */

if ( bool_var == false ) {} /* Compliant */

if ( ( 0u == 0u ) || ( bool_var = true ) ) {}
/* Non-compliant - even though (bool_var=true) is not evaluated */

if ( ( x = f () ) != 0 ) {}
    /* Non-compliant - value of x=f() is used */

a[b += c] = a[b];
    /* Non-compliant - value of b += c is used */

b = c = d = 0; /* Non-compliant - value of d=0 and c=d=0 are used */
}

```

In this example, the rule is violated when the result of an assignment is used.

Check Information

Group: Side Effects

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 13.2

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 13.5

The right hand operand of a logical `&&` or `||` operator shall not contain persistent side effects

Description

Rule Definition

The right hand operand of a logical `&&` or `||` operator shall not contain persistent side effects.

Rationale

The right operand of an `||` operator is not evaluated if the left operand is true. The right operand of an `&&` operator is not evaluated if the left operand is false. In these cases, if the right operand modifies the value of a variable, the modification does not take place. Following the operation, if you expect a modified value of the variable, the modification might not always happen.

Polyspace Specification

- For this rule, Polyspace considers that all function calls have a persistent side effect.
- If the right operand is a volatile variable, Polyspace does not flag this as a rule violation.

Message in Report

The right hand operand of a `&&` operator shall not contain side effects. The right hand operand of a `||` operator shall not contain side effects.

Examples

Right Operand of Logical Operator with Persistent Side Effects

```
int check (int arg) {
    static int count;
    if(arg > 0) {
        count++;
        return 1;
    }
    else
        return 0;
}

int getSwitch(void);
int getVal(void);

void main(void) {
    int val = getVal();
    int mySwitch = getSwitch();
    int checkResult;

    if(mySwitch && check(val)) { /* Non-compliant */
    }

    checkResult = check(val);
    if(checkResult && mySwitch) { /* Compliant */
    }

    if(check(val) && mySwitch) { /* Compliant */
    }
}
```

In this example, the rule is violated when the right operand of the `&&` operation contains a function call. The function call has a persistent side effect because the static variable `count` is modified in the function body. Depending on `mySwitch`, this modification might or might not happen.

The rule is not violated when the left operand contains a function call. Alternatively, to avoid the rule violation, assign the result of the function call to a variable. Use this variable in the logical operation in place of the function call.

In this example, the function call has the side effect of modifying a `static` variable. Polyspace flags all function calls when used on the right-hand side of a logical `&&` or `||` operator, even when the function does not have a side effect. Manually inspect your function body to see if it has side effects. If the function does not have side effects, add a comment and justification in your Polyspace result explaining why you retained your code.

Check Information

Group: Side Effects

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 13.6

The operand of the `sizeof` operator shall not contain any expression which has potential side effects

Description

Rule Definition

The operand of the `sizeof` operator shall not contain any expression which has potential side effects.

Rationale

The argument of a `sizeof` operator is usually not evaluated at run time. If the argument is an expression, you might wrongly expect that the expression is evaluated.

Polyspace Specification

The rule is not violated if the argument is a `volatile` variable.

Message in Report

The operand of the `sizeof` operator shall not contain any expression which has potential side effects.

Examples

Expressions in `sizeof` Operator

```
#include <stddef.h>
int x;
int y[40];
struct S {
```

```
    int a;
    int b;
};
struct S myStruct;

void main() {
    size_t sizeOfType;
    sizeOfType = sizeof(x);           /* Compliant */
    sizeOfType = sizeof(y);           /* Compliant */
    sizeOfType = sizeof(myStruct);    /* Compliant */
    sizeOfType = sizeof(x++);         /* Non-compliant */
}
```

In this example, the rule is violated when the expression `x++` is used as argument of `sizeof` operator.

Check Information

Group: Side Effects

Category: Mandatory

AGC Category: Mandatory

Language: C90, C99

See Also

MISRA C:2012 Rule 18.8

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 14.1

A loop counter shall not have essentially floating type

Description

Rule Definition

A loop counter shall not have essentially floating type.

Rationale

When using a floating-point loop counter, accumulation of rounding errors can result in a mismatch between the expected and actual number of iterations. This rounding error can happen when a loop step that is not a power of the floating point radix is rounded to a value that can be represented by a float.

Even if a loop with a floating-point loop counter appears to behave correctly on one implementation, it can give a different number of iteration on another implementation.

Polyspace Specification

If the `for` index is a variable symbol, Polyspace checks that it is not a float.

Message in Report

A loop counter shall not have essentially floating type.

Examples

`for` Loop Counters

```
int main(void){
    unsigned int counter = 0u;
```



```

int result = 0;
float foo;

// Float loop counters
for(float foo = 0.0f; foo < 1.0f; foo +=0.001f){
    /* Non-compliant - counter = 1000 at the end of the loop */
    ++counter;
}

float fff = 0.0f;
for(fff = 0.0f; fff <12.0f; fff += 1.0f){    /* Non-compliant*/
    result++;
}

// Integer loop count
for(unsigned int count = 0u; count < 1000u; ++count){ /* Compliant */
    foo = (float) count * 0.001f;
}
}

```

In this example, the three `for` loops show three different loop counters. The first and second `for` loops use float variables as loop counters, and therefore are not compliant. The third loop uses the integer `count` as the loop counter. Even though `count` is used as a float inside the loop, the variable remains an integer when acting as the loop index. Therefore, this `for` loop is compliant.

while Loop Counters

```

int main(void){
    unsigned int u32a;
    float foo;

    foo = 0.0f;
    while (foo < 1.0f){
        foo += 0.001f; /* Non-compliant - foo used as a loop counter */
    }

    foo = read_float32();
    do{
        u32a = read_u32();
    }while( ((float)u32a - foo) > 10.0f );
        /* Compliant - foo doesn't change in the loop */
        /* so cannot be a counter */
}

```

```
    return 1;
}
```

This example shows two `while` loops both of which use `foo` in the `while`-loop conditions.

The first `while` loop uses `foo` in the condition and inside the loop. Because `foo` changes, floating-point rounding errors can cause unexpected behavior.

The second `while` loop does not use `foo` inside the loop, but does use `foo` inside the `while`-condition. So `foo` is not the loop counter. The integer `u32a` is the loop counter because it changes inside the loop and is part of the `while` condition. Because `u32a` is an integer, the rounding error issue is not a concern, making this `while` loop compliant.

Check Information

Group: Control Statement Expressions

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 14.2

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 14.2

A for loop shall be well-formed

Description

Rule Definition

A for loop shall be well-formed.

Rationale

The `for` statement provides a general-purpose looping facility. Using a restricted form of loop makes code easier to review and to analyze.

Polyspace Specification

Polyspace checks that:

- The `for` loop index (`v`) is a variable symbol.
- `v` is the last assigned variable in the first expression (if present).
- If the first expression exists, it contains an assignment of `v`.
- If the second expression exists, it is a comparison of `v`.
- If the third expression exists, it is an assignment of `v`.
- There are no direct assignments of the `for` loop index.

Message in Report

- 1st expression should be an assignment. The following kinds of `for` loops are allowed:
 - all three expressions shall be present;
 - the 2nd and 3rd expressions shall be present with prior initialization of the loop counter;

- all three expressions shall be empty for a deliberate infinite loop.
- 3rd expression should be an assignment of a loop counter.
- 3rd expression : assigned variable should be the loop counter (*counter*).
- 3rd expression should be an assignment of loop counter (*counter*) only.
- 2nd expression should contain a comparison with loop counter (*counter*).
- Loop counter (*counter*) should not be modified in the body of the loop.
- Bad type for loop counter (*counter*).

Examples

Altering the Loop Counter Inside the Loop

```
void foo(void){  
  
    for(short index=0; index < 5; index++){ /* Non-compliant */  
        index = index + 3;          /* Altering the loop counter */  
    }  
}
```

In this example, the loop counter `index` changes inside the `for` loop. It is hard to determine when the loop terminates.

One possible correction is to use an extra flag to terminate the loop early.

In this correction, the second clause of the `for` loop depends on the counter value, `index < 5`, and upon an additional flag, `!flag`. With the additional flag, the `for` loop definition and counter remain readable, and you can escape the loop early.

```
#define FALSE 0  
#define TRUE 1  
  
void foo(void){  
  
    int flag = FALSE;  
  
    for(short index=0; (index < 5) && !flag; index++){ /* Compliant */  
        if((index % 4) == 0){
```

```

        flag = TRUE;          /* allows early termination of loop */
    }
}

```

for Loops With Empty Clauses

```

void foo(void)
    for(short index = 0; ; index++) {} /* Non-compliant */

    for(short index = 0; index < 10;) {} /* Non-compliant */

    short index;
    for(; index < 10;) {} /* Non-compliant */

    for(; index < 10; i++) {} /* Compliant */

    for(;;){}
        /* Compliant - Exception all three clauses can be empty */
}

```

This example shows `for` loops definitions with a variety of missing clauses. To be compliant, initialize the first clause variable before the `for` loop (line 9). However, you cannot have a `for` loop without the second or third clause.

The one exception is a `for` loop with all three clauses empty, so as to allow for infinite loops.

Check Information

Group: Control Statement Expressions

Category: Required

AGC Category: Readability

Language: C90, C99

See Also

MISRA C:2012 Rule 14.1 | MISRA C:2012 Rule 14.3 | MISRA C:2012 Rule 14.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 14.3

Controlling expressions shall not be invariant

Description

Rule Definition

Controlling expressions shall not be invariant.

Rationale

If the controlling expression, for example an `if` condition, has a constant value, the non-changing value can point to a programming error.

Polyspace Specification

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Polyspace Bug Finder flags some violations of MISRA C 14.3 through the `Dead code` and `Useless if` checkers.

Polyspace Code Prover does not use gray code to flag MISRA C 14.3 violations.

Message in Report

- Boolean operations whose results are invariant shall not be permitted.
- Expression is always true.
- Boolean operations whose results are invariant shall not be permitted.
- Expression is always false.
- Controlling expressions shall not be invariant.

Check Information

Group: Control Statement Expressions

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 2.1 | MISRA C:2012 Rule 14.2

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 14.4

The controlling expression of an `if` statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Description

Rule Definition

The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Rationale

Strong typing requires the controlling expression on an `if` statement or iteration statement to have *essentially Boolean* type.

Polyspace Specification

Polyspace does not flag integer constants, for example `if (2)`.

If your configuration includes the option `-boolean-types`, the number of warnings can increase or decrease.

Message in Report

The controlling expression of an `if` statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Examples

Controlling Expression in `if`, `while`, and `for`

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

```
#define TRUE = 1

typedef _Bool bool_t;
extern bool_t flag;

void foo(void){
    int *p = 1;
    int *q = 0;
    int i = 0;
    while(p){}          /* Non-compliant - p is a pointer */

    while(q != NULL){} /* Compliant */

    while(TRUE){}      /* Compliant */

    while(flag){}      /* Compliant */

    if(i){}            /* Non-compliant - int32_t is not boolean */

    if(i != 0){}       /* Compliant */

    for(int i=-10; i;i++){} /* Non-compliant - int32_t is not boolean */

    for(int i=0; i<10;i++){} /* Compliant */
}
```

This example shows various controlling expressions in while, if, and for statements.

The noncompliant statements (the first while, if, and for examples), use a single non-Boolean variable. If you use a single variable as the controlling statement, it must be essentially Boolean (lines 17 and 19). Boolean expressions are also compliant with MISRA.

Check Information

Group: Control Statement Expressions

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 14.2 | MISRA C:2012 Rule 20.8

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 15.1

The `goto` statement should not be used

Description

Rule Definition

The goto statement should not be used.

Rationale

Unrestricted use of `goto` statements makes the program unstructured and difficult to understand.

Message in Report

The `goto` statement should not be used.

Examples

Use of `goto` Statements

```
void foo(void) {
    int i = 0, result = 0;

label1:
    for ( i; i < 5; i++ ) {
        if (i > 2) goto label2;    /* Non-compliant */
    }

label2: {
    result++;
    goto label1;                 /* Non-compliant */
}
}
```

In this example, the rule is violated when `goto` statements are used.

Check Information

Group: Control Flow

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 15.2 | MISRA C:2012 Rule 15.3 | MISRA C:2012 Rule 15.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 15.2

The goto statement shall jump to a label declared later in the same function

Description

Rule Definition

The goto statement shall jump to a label declared later in the same function.

Rationale

Unrestricted use of goto statements makes the program unstructured and difficult to understand. You can use a forward goto statement together with a backward one to implement iterations. Restricting backward goto statements ensures that you use only iteration statements provided by the language such as for or while to implement iterations. This restriction reduces visual complexity of the code.

Message in Report

The goto statement shall jump to a label declared later in the same function.

Examples

Use of Backward goto Statements

```
void foo(void) {
    int i = 0, result = 0;

label1:
    for ( i; i < 5; i++ ) {
        if (i > 2) goto label2;    /* Compliant */
    }

label2: {
```

```
        result++;
        goto label1;           /* Non-compliant */
    }
}
```

In this example, the rule is violated when a `goto` statement causes a backward jump to `label1`.

The rule is not violated when a `goto` statement causes a forward jump to `label2`.

Check Information

Group: Control Flow

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 15.1 | MISRA C:2012 Rule 15.3 | MISRA C:2012 Rule 15.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 15.3

Any label referenced by a `goto` statement shall be declared in the same block, or in any block enclosing the `goto` statement

Description

Rule Definition

Any label referenced by a `goto` statement shall be declared in the same block, or in any block enclosing the `goto` statement.

Rationale

Unrestricted use of `goto` statements makes the program unstructured and difficult to understand. Restricting use of `goto` statements to jump between blocks or into nested blocks reduces visual code complexity.

Message in Report

Any label referenced by a `goto` statement shall be declared in the same block, or in any block enclosing the `goto` statement.

Examples

`goto` Statements Jump Inside Block

```
void f1(int a) {
    if(a <= 0) {
        goto L2;          /* Non-compliant - L2 in different block*/
    }

    goto L1;             /* Compliant - L1 in same block*/

    if(a == 0) {
```



```

        goto L1;          /* Compliant - L1 in outer block*/
    }

    goto L2;             /* Non-compliant - L2 in inner block*/

L1: if(a > 0) {
        L2;;
    }
}

```

In this example, `goto` statements cause jumps to different labels. The rule is violated when:

- The label occurs in a block different from the block containing the `goto` statement.
The block containing the label neither encloses nor is enclosed by the current block.
- The label occurs in a block enclosed by the block containing the `goto` statement.

The rule is not violated when:

- The label occurs in the same block as the block containing the `goto` statement..
- The label occurs in a block that encloses the block containing the `goto` statement..

goto Statements in switch Block

```

void f2 ( int x, int z ) {
    int y = 0;

    switch(x) {
    case 0:
        if(x == y) {
            goto L1; /* Non-compliant - switch-clauses are treated as blocks */
        }
        break;
    case 1:
        y = x;
        L1: ++x;
        break;
    default:
        break;
    }
}

```

In this example, the label for the `goto` statement appears to occur in a block that encloses the block containing the `goto` statement. However, for the purposes of this rule, the software considers that each `case` statement begins a new block. Therefore, the `goto` statement violates the rule.

Check Information

Group: Control Flow

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 15.1 | MISRA C:2012 Rule 15.2 | MISRA C:2012 Rule 15.4 | MISRA C:2012 Rule 16.1

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 15.4

There should be no more than one break or goto statement used to terminate any iteration statement

Description

Rule Definition

There should be no more than one break or goto statement used to terminate any iteration statement.

Rationale

If you use one `break` or `goto` statement in your loop, you have one secondary exit point from the loop. Restricting number of exits from a loop in this way reduces visual complexity of your code.

Message in Report

There should be no more than one break or goto statement used to terminate any iteration statement.

Examples

`break` Statements in Inner and Outer Loops

```
volatile int stop;

int func(int *arr, int size, int sat) {
    int i,j;
    int sum = 0;
    for (i=0; i< size; i++) { /* Compliant */
        if(sum >= sat)
            break;
    }
}
```

```
        for (j=0; j< i; j++) { /* Compliant */
            if(stop)
                break;
            sum += arr[j];
        }
    }
}
```

In this example, the rule is not violated in both the inner and outer loop because both loops have one `break` statement each.

break and goto Statements in Loop

```
volatile int stop;

void displayStopMessage();

int func(int *arr, int size, int sat) {
    int i;
    int sum = 0;
    for (i=0; i< size; i++) { /* Non-compliant */
        if(sum >= sat)
            break;
        if(stop)
            goto L1;
        sum += arr[i];
    }

    L1: displayStopMessage();
}
```

In this example, the rule is violated because the `for` loop has one `break` statement and one `goto` statement.

goto Statement in Inner Loop and break Statement in Outer Loop

```
volatile int stop;

void displayMessage();

int func(int *arr, int size, int sat) {
    int i,j;
```

```
int sum = 0;
for (i=0; i< size; i++) { /* Non-compliant */
    if(sum >= sat)
        break;
    for (j=0; j< i; j++) { /* Compliant */
        if(stop)
            goto L1;
        sum += arr[i];
    }
}

L1: displayMessage();
}
```

In this example, the rule is not violated in the inner loop because you can exit the loop only through the one `goto` statement. However, the rule is violated in the outer loop because you can exit the loop through either the `break` statement or the `goto` statement in the inner loop.

Check Information

Group: Control Flow

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 15.1 | MISRA C:2012 Rule 15.2 | MISRA C:2012 Rule 15.3

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 15.5

A function should have a single point of exit at the end

Description

Rule Definition

A function should have a single point of exit at the end.

Rationale

This rule requires that a `return` statement must occur as the last statement in the function body. Otherwise, the following issues can occur:

- Code following a `return` statement can be unintentionally omitted.
- If a function that modifies some of its arguments has early `return` statements, when reading the code, it is not immediately clear which modifications actually occur.

Message in Report

A function should have a single point of exit at the end.

Examples

More Than One `return` Statement in Function

```
#define MAX ((unsigned int)2147483647)
#define NULL (void*)0

typedef unsigned int bool_t;
bool_t false = 0;
bool_t true = 1;
```

```
bool_t f1(unsigned short n, char *p) {           /* Non-compliant */
    if(n > MAX) {
        return false;
    }

    if(p == NULL) {
        return false;
    }

    return true;
}
```

In this example, the rule is violated because there are three return statements.

One possible correction is to store the return value in a variable and return this variable just before the function ends.

```
#define MAX ((unsigned int)2147483647)
#define NULL (void*)0

typedef unsigned int bool_t;
bool_t false = 0;
bool_t true = 1;
bool_t return_value;

bool_t f2 (unsigned short n, char *p) {         /* Compliant */
    return_value = true;
    if(n > MAX) {
        return_value = false;
    }

    if(p == NULL) {
        return_value = false;
    }

    return return_value;
}
```

Check Information

Group: Control Flow

Category: Advisory

AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 17.4

Topics

“Set Up Coding Rules Checking”
“Review Coding Rule Violations”
“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 15.6

The body of an iteration-statement or a selection-statement shall be a compound statement

Description

Rule Definition

The body of an iteration-statement or a selection-statement shall be a compound-statement.

Rationale

The rule applies to:

- Iteration statements such as `while`, `do ... while` or `for`.
- Selection statements such as `if ... else` or `switch`.

If the block of code associated with an iteration or selection statement is not contained in braces, you can make mistakes about the association. For example:

- You can wrongly associate a line of code with an iteration or selection statement because of its indentation.
- You can accidentally place a semicolon following the iteration or selection statement. Because of the semicolon, the line following the statement is no longer associated with the statement even though you intended otherwise.

Message in Report

- The `else` keyword shall be followed by either a compound statement, or another `if` statement.
- An `if (expression)` construct shall be followed by a compound statement.
- The statement forming the body of a `while` statement shall be a compound statement.

- The statement forming the body of a do ... while statement shall be a compound statement.
- The statement forming the body of a for statement shall be a compound statement.
- The statement forming the body of a switch statement shall be a compound statement.

Examples

Iteration Block

```
int data_available = 1;
void f1(void) {
    while(data_available)                /* Non-compliant */
        process_data();

    while(data_available) {              /* Compliant */
        process_data();
    }
}
```

In this example, the second while block is enclosed in braces and does not violate the rule.

Nested Selection Statements

```
void f1(void) {
    if(flag_1)                            /* Non-compliant */
        if(flag_2)                        /* Non-compliant */
            action_1();
    else                                    /* Non-compliant */
        action_2();
}
```

In this example, the rule is violated because the if or else blocks are not enclosed in braces. Unless indented as above, it is easy to associate the else statement with the inner if.

One possible correction is to enclose each block associated with an if or else statement in braces.

```

void f1(void) {
    if(flag_1) {
        if(flag_2) {
            action_1();
        }
    }
    else {
        action_2();
    }
}

```

Spurious Semicolon After Iteration Statement

```

void f1(void) {
    while(flag_1);
    {
        flag_1 = action_1();
    }
}

```

In this example, the rule is violated even though the `while` statement is followed by a block in braces. The semicolon following the `while` statement causes the block to dissociate from the `while` statement.

The rule helps detect such spurious semicolons.

Check Information

Group: Control Flow

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 15.7

All if ... else if constructs shall be terminated with an else statement

Description

Rule Definition

All if ... else if constructs shall be terminated with an else statement.

Rationale

Unless there is a terminating else statement in an if...elseif...else construct, during code review, it is difficult to tell if you considered all possible results for the if condition.

Message in Report

All if ... else if constructs shall be terminated with an else statement.

Examples

Missing else Block

```
int get_flag_1(void);
int get_flag_2(void);
void action_1(void);
void action_2(void);

void f1(void) {
    int flag_1 = get_flag_1(), flag_2 = get_flag_2();
    if(flag_1) {
        action_1();
    }
    else if(flag_2) {
```

```
        /* Non-compliant */  
        action_2();  
    }  
}
```

In this example, the rule is violated because the `if ... else if` construct does not have a terminating `else` block.

To avoid the rule violation, add a terminating `else` block. The block can be empty.

```
int get_flag_1(void);  
int get_flag_2(void);  
void action_1(void);  
void action_2(void);  
  
void f1(void) {  
    int flag_1 = get_flag_1(), flag_2 = get_flag_2();  
    if(flag_1) {  
        action_1();  
    }  
    else if(flag_2) {  
        /* Non-compliant */  
        action_2();  
    }  
    else {  
        /* No statement required */  
        /* ; is optional */  
    }  
}
```

Check Information

Group: Control Flow

Category: Required

AGC Category: Readability

Language: C90, C99

See Also

MISRA C:2012 Rule 16.5

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 16.1

All switch statements shall be well-formed

Description

Rule Definition

All switch statements shall be well-formed

Rationale

The syntax for switch statements in C is not particularly rigorous and can allow complex, unstructured behavior. This rule and other rules impose a simple consistent structure on the `switch` statement.

Polyspace Specification

Following the MISRA specifications, the coding rules checker also raises a violation of rule 16.1 if a `switch` statement violates one of these rules: 16.2, 16.3, 16.4, 16.5 or 16.6.

Message in Report

All messages in report file begin with "MISRA-C switch statements syntax normative restriction."

- Initializers shall not be used in switch clauses.
- The child statement of a switch shall be a compound statement.
- All switch clauses shall appear at the same level.
- A switch clause shall only contain switch labels and switch clauses, and no other code.
- A switch statement shall only contain switch labels and switch clauses, and no other code.

Check Information

Group: Switch Statements

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 15.3 | MISRA C:2012 Rule 16.2 | MISRA C:2012 Rule 16.3 | MISRA C:2012 Rule 16.4 | MISRA C:2012 Rule 16.5 | MISRA C:2012 Rule 16.6

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 16.2

A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

Description

Rule Definition

A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

Rationale

The C Standard permits placing a switch label (for instance, `case` or `default`) before any statement contained in the body of a switch statement. This flexibility can lead to unstructured code. To prevent unstructured code, make sure a switch label appears only at the outermost level of the body of a switch statement.

Message in Report

All messages in report file begin with "MISRA-C switch statements syntax normative restriction."

- Initializers shall not be used in switch clauses.
- The child statement of a switch shall be a compound statement.
- All switch clauses shall appear at the same level.
- A switch clause shall only contain switch labels and switch clauses, and no other code.
- A switch statement shall only contain switch labels and switch clauses, and no other code.

Check Information

Group: Switch Statements

Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 16.1

Topics

“Set Up Coding Rules Checking”
“Review Coding Rule Violations”
“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 16.3

An unconditional break statement shall terminate every switch-clause

Description

Rule Definition

An unconditional break statement shall terminate every switch-clause

Rationale

A *switch-clause* is a case containing at least one statement. Two consecutive labels without an intervening statement is compliant with MISRA.

If you fail to end your switch-clauses with a break statement, then control flow “falls” into the next statement. This next statement can be another switch-clause, or the end of the switch. This behavior is sometimes intentional, but more often it is an error. If you add additional cases later, an unterminated switch-clause can cause problems.

Polyspace Specification

Polyspace raises a warning for each noncompliant `case` clause.

Message in Report

An unconditional break statement shall terminate every switch-clause.

Check Information

Group: Switch Statements

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 16.1

Topics

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 16.4

Every switch statement shall have a default label

Description

Rule Definition

Every switch statement shall have a default label

Rationale

The requirement for a `default` label is defensive programming. Even if your switch covers all possible values, there is no guarantee that the input takes one of these values. Statements following the `default` label take some appropriate action. If the `default` label requires no action, use comments to describe why there are no specific actions.

Message in Report

Every switch statement shall have a default label.

Examples

Switch Statement Without `default`

```
short func1(short xyz){  
  
    switch(xyz){          /* Non-compliant - default label is required */  
        case 0:  
            ++xyz;  
            break;  
        case 1:  
        case 2:  
            break;  
    }  
}
```

```

    return xyz;
}

```

In this example, the switch statement does not include a default label, and is therefore noncompliant.

One possible correction is to use the default label to flag input errors. If your switch-clauses cover all expected input, then the default cases flags any input errors.

```

short func1(short xyz){

    switch(xyz){          /* Compliant */
        case 0:
            ++xyz;
            break;
        case 1:
        case 2:
            break;
        default:
            errorflag = 1;
            break;
    }
    if (errorflag == 1)
        return errorflag;
    else
        return xyz;
}

```

Switch Statement for Enumerated Inputs

```

enum Colors{
    RED, GREEN, BLUE
};

enum Colors func2(enum Colors color){
    enum Colors next;

    switch(color){      /* Non-compliant - default label is required */
        case RED:
            next = GREEN;
            break;
        case GREEN:
            next = BLUE;

```

```
        break;
    case BLUE:
        next = RED;
        break;
    }
    return next;
}
```

In this example, the switch statement does not include a `default` label, and is therefore noncompliant. Even though this switch statement handles all values of the enumeration, there is no guarantee that `color` takes one of the those values.

To be compliant, add the `default` label to the end of your switch. You can use this case to flag unexpected inputs.

```
enum Colors{
    RED, GREEN, BLUE, ERROR
};

enum Colors func2(enum Colors color){
    enum Colors next;

    switch(color){          /* Compliant */
        case RED:
            next = GREEN;
            break;
        case GREEN:
            next = BLUE;
            break;
        case BLUE:
            next = RED;
            break;
        default:
            next = ERROR;
            break;
    }

    return next;
}
```


Check Information

Group: Switch Statements

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 2.1 | MISRA C:2012 Rule 16.1

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 16.5

A default label shall appear as either the first or the last switch label of a switch statement

Description

Rule Definition

A default label shall appear as either the first or the last switch label of a switch statement.

Rationale

Using this rule, you can easily locate the `default` label within a `switch` statement.

Message in Report

A default label shall appear as either the first or the last switch label of a switch statement.

Examples

Default Case in `switch` Statements

```
void foo(int var){  
  
    switch(var){  
        default: /* Compliant - default is the first label */  
        case 0:  
            ++var;  
            break;  
        case 1:  
        case 2:  
            break;  
    }  
}
```

```
    }

    switch(var) {
        case 0:
            ++var;
            break;
        default: /* Non-compliant - default is mixed with the case labels */
        case 1:
        case 2:
            break;
    }

    switch(var) {
        case 0:
            ++var;
            break;
        case 1:
        case 2:
        default: /* Compliant - default is the last label */
            break;
    }

    switch(var) {
        case 0:
            ++var;
            break;
        case 1:
        case 2:
            break;
        default: /* Compliant - default is the last label */
            var = 0;
            break;
    }
}
```

This example shows the same switch statement several times, each with `default` in a different place. As the first, third, and fourth switch statements show, `default` must be the first or last label. `default` can be part of a compound switch-clause (for instance, the third switch example), but it must be the last listed.

Check Information

Group: Switch Statements

Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 15.7 | MISRA C:2012 Rule 16.1

Topics

“Set Up Coding Rules Checking”
“Review Coding Rule Violations”
“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 16.6

Every switch statement shall have at least two switch-clauses

Description

Rule Definition

Every switch statement shall have at least two switch-clauses.

Rationale

A switch statement with a single path is redundant and can indicate a programming error.

Message in Report

Every switch statement shall have at least two switch-clauses.

Check Information

Group: Switch Statements

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 16.1

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 16.7

A switch-expression shall not have essentially Boolean type

Description

Rule Definition

A switch-expression shall not have essentially Boolean type

Rationale

The C Standard requires the controlling expression to a `switch` statement to have an integer type. Because C implements Boolean values with integer types, it is possible to have a Boolean expression control a `switch` statement. For controlling flow with Boolean types, an `if-else` construction is more appropriate.

Polyspace Specification

If your configuration uses the `-boolean-types` option, the number of reported violations can increase.

Message in Report

A switch-expression shall not have essentially Boolean type.

Check Information

Group: Switch Statements

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

Topics

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 17.1

The features of <starg.h> shall not be used

Description

Rule Definition

The features of <stdarg.h> shall not be used..

Rationale

The rule forbids use of `va_list`, `va_arg`, `va_start`, `va_end`, and `va_copy`.

You can use these features in ways where the behavior is not defined in the Standard. For instance:

- You invoke `va_start` in a function but do not invoke the corresponding `va_end` before the function block ends.
- You invoke `va_arg` in different functions on the same variable of type `va_list`.
- `va_arg` has the syntax `type va_arg (va_list ap, type)`.

You invoke `va_arg` with a `type` that is incompatible with the actual type of the argument retrieved from `ap`.

Message in Report

The features of <stdarg.h> shall not be used.

Examples

Use of `va_start`, `va_list`, `va_arg`, and `va_end`

```
#include<stdarg.h>
void f2(int n, ...) {
```

```
int i;
double val;
va_list vl;                                /* Non-compliant */

va_start(vl, n);                            /* Non-compliant */

for(i = 0; i < n; i++)
{
    val = va_arg(vl, double);              /* Non-compliant */
}

va_end(vl);                                 /* Non-compliant */
}
```

In this example, the rule is violated because `va_start`, `va_list`, `va_arg` and `va_end` are used.

Undefined Behavior of `va_arg`

```
#include <stdarg.h>
void h(va_list ap) {                        /* Non-compliant */
    double y;

    y = va_arg(ap, double );              /* Non-compliant */
}

void g(unsigned short n, ...) {
    unsigned int x;
    va_list ap;                            /* Non-compliant */

    va_start(ap, n);                      /* Non-compliant */
    x = va_arg(ap, unsigned int);         /* Non-compliant */

    h(ap);

    /* Undefined - ap is indeterminate because va_arg used in h () */
    x = va_arg(ap, unsigned int);        /* Non-compliant */
}

void f(void) {
    /* undefined - uint32_t:double type mismatch when g uses va_arg () */
}
```

```
    g(1, 2.0, 3.0);  
}
```

In this example, `va_arg` is used on the same variable `ap` of type `va_list` in both functions `g` and `h`. In `g`, the second argument is `unsigned int` and in `h`, the second argument is `double`. This type mismatch causes undefined behavior.

Check Information

Group: Function

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 17.2

Functions shall not call themselves, either directly or indirectly

Description

Rule Definition

Functions shall not call themselves, either directly or indirectly.

Rationale

Variables local to a function are stored in the call stack. If a function calls itself directly or indirectly several times, the available stack space can be exceeded, causing serious failure. Unless the recursion is tightly controlled, it is difficult to determine the maximum stack space required.

Message in Report

Message in Report: Function XX shall not call itself either directly or indirectly. Function XX is called indirectly by YY.

Examples

Direct and Indirect Recursion

```
void foo1( void ) {      /* Non-compliant - Indirect recursion foo1->foo2->foo1... */
    foo2();
    foo1();              /* Non-compliant - Direct recursion */
}

void foo2( void ) {
    foo1();
}
```

In this example, the rule is violated because of:

- Direct recursion `foo1 → foo1`.
- Indirect recursion `foo1 → foo2 → foo1`.

Check Information

Group: Function

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Polyspace Results

Number of Recursions | Number of Direct Recursions

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 17.3

A function shall not be declared implicitly

Description

Rule Definition

A function shall not be declared implicitly.

Rationale

An implicit declaration occurs when you call a function before declaring or defining it. When you declare a function explicitly before calling it, the compiler can match the argument and return types with the parameter types in the declaration. If an implicit declaration occurs, the compiler makes assumptions about the argument and return types. For instance, it assumes a return type of `int`. The assumptions might not agree with what you expect and cause undesired type conversions.

Message in Report

Function 'XX' has no complete visible prototype at call.

Examples

Function Not Declared Before Call

```
#include <math.h>

extern double power3 (double val, int exponent);
int getChoice(void);

double func() {
    double res;
    int ch = getChoice();
```

```
    if(ch == 0) {
        res = power(2.0, 10);    /* Non-compliant */
    }
    else if( ch==1) {
        res = power2(2.0, 10);  /* Non-compliant */
    }
    else {
        res = power3(2.0, 10);  /* Compliant */
        return res;
    }
}

double power2 (double val, int exponent) {
    return (pow(val, exponent));
}
```

In this example, the rule is violated when a function that is not declared is called in the code. Even if a function definition exists later in the code, the rule violation occurs.

The rule is not violated when the function is declared before it is called in the code. If the function definition exists in another file and is available only during the link phase, you can declare the function in one of the following ways:

- Declare the function with the `extern` keyword in the current file.
- Declare the function in a header file and include the header file in the current file.

Check Information

Group: Function

Category: Mandatory

AGC Category: Mandatory

Language: C90

See Also

MISRA C:2012 Rule 8.2 | MISRA C:2012 Rule 8.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 17.4

All exit paths from a function with non-void return type shall have an explicit return statement with an expression

Description

Rule Definition

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Rationale

If a non-void function does not explicitly return a value but the calling function uses the return value, the behavior is undefined. To prevent this behavior:

- 1 You must provide `return` statements with an explicit expression.
- 2 You must ensure that during run time, at least one `return` statement executes.

Message in Report

Missing return value for non-void function 'XX'.

Examples

Missing Return Statement Along Certain Execution Paths

```
int absolute(int v) {
    if(v < 0) {
        return v;
    }
}
```

In this example, the rule is violated because a `return` statement does not exist on all execution paths. If `v >= 0`, then the control returns to the calling function without an explicit return value.

Return Statement Without Explicit Expression

```
#define SIZE 10
int table[SIZE];

unsigned short lookup(unsigned short v) {
    if((v < 0) || (v > SIZE)) {
        return;
    }
    return table[v];
}
```

In this example, the rule is violated because the `return` statement in the `if` block does not have an explicit expression.

Check Information

Group: Function

Category: Mandatory

AGC Category: Mandatory

Language: C90, C99

See Also

MISRA C:2012 Rule 15.5

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 17.5

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements

Description

Rule Definition

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.

Rationale

If you use an array declarator for a function parameter instead of a pointer, the function interface is clearer because you can state the minimum expected array size. If you do not state a size, the expectation is that the function can handle an array of any size. In such cases, the size value is typically another parameter of the function, or the array is terminated with a sentinel value.

However, it is legal in C to specify an array size but pass an array of smaller size. This rule prevents you from passing an array of size smaller than the size you declared.

Message in Report

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.

The argument type has *actual_size* elements whereas the parameter type expects *expected_size* elements.

Examples

Incorrect Array Size Passed to Function

```
void func(int arr[4]);

int main() {
    int arrSmall[3] = {1,2,3};
    int arr[4] = {1,2,3,4};
    int arrLarge[5] = {1,2,3,4,5};

    func(arrSmall);      /* Non-compliant */
    func(arr);           /* Compliant */
    func(arrLarge);     /* Compliant */

    return 0;
}
```

In this example, the rule is violated when `arrSmall`, which has size 3, is passed to `func`, which expects at least 4 elements.

Check Information

Group: Functions

Category: Advisory

AGC Category: Readability

Language: C90. C99

See Also

MISRA C:2012 Rule 17.6

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 17.6

The declaration of an array parameter shall not contain the `static` keyword between the `[]`.

Description

Rule Definition

The declaration of an array parameter shall not contain the `static` keyword between the `[]`.

Rationale

If you use the `static` keyword within `[]` for an array parameter of a function, you can inform a C99 compiler that the array contains a minimum number of elements. The compiler can use this information to generate efficient code for certain processors. However, in your function call, if you provide less than the specified minimum number, the behavior is not defined.

Message in Report

The declaration of an array parameter shall not contain the `static` keyword between the `[]`.

Examples

Use of `static` Keyword Within `[]` in Array Parameter

```
extern int arr1[20];
extern int arr2[10];

/* Non-compliant: static keyword used in array declarator */
unsigned int total (unsigned int n, unsigned int arr[static 20]) {
    unsigned int i;
    unsigned int sum = 0;
```

```
    for (i=0U; i < n; i++) {
        sum+= arr[i];
    }

    return sum;
}

void func (void) {
    int res, res2;
    res = total (10U, arr1); /* Non-compliant - behavior not defined */
    res2 = total (20U, arr2); /* Non-compliant, even if behavior is defined */
}
```

In this example, the rule is violated when the `static` keyword is used within `[]` in the array parameter of function `total`. Even if you call `total` with array arguments where the behavior is well-defined, the rule violation occurs.

Check Information

Group: Function

Category: Mandatory

AGC Category: Mandatory

Language: C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 17.7

The value returned by a function having non-void return type shall be used

Description

Rule Definition

The value returned by a function having non-void return type shall be used.

Rationale

You can unintentionally call a function with a non-void return type but not use the return value. Because the compiler allows the call, you might not catch the omission. This rule forbids calls to a non-void function where the return value is not used. If you do not intend to use the return value of a function, explicitly cast the return value to void.

Message in Report

The value returned by a function having non-void return type shall be used.

Examples

Used and Unused Return Values

```
unsigned int cutOff(unsigned int val) {
    if (val > 10 && val < 100) {
        return val;
    }
    else {
        return 0;
    }
}
```



```
unsigned int getVal(void);

void func2(void) {
    unsigned int val = getVal(), res;
    cutOff(val);          /* Non-compliant */
    res = cutOff(val);    /* Compliant */
    (void)cutOff(val);    /* Compliant */
}
```

In this example, the rule is violated when the return value of `cutOff` is not used subsequently.

The rule is not violated when the return value is:

- Assigned to another variable.
- Explicitly cast to `void`.

Check Information

Group: Function

Category: Required

AGC Category: Readability

Language: C90, C99

See Also

MISRA C:2012 Rule 2.2

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 17.8

A function parameter should not be modified

Description

Rule Definition

A function parameter should not be modified.

Rationale

When you modify a parameter, the function argument corresponding to the parameter is not modified. However, you or another programmer unfamiliar with C can expect by mistake that the argument is also modified when you modify the parameter.

Message in Report

A function parameter should not be modified.

Examples

Function Parameter Modified

```
int input(void);

void func(int param1, int* param2) {
    param1 = input(); /* Non-compliant */
    *param2 = input(); /* Compliant */
}
```

In this example, the rule is violated when the parameter `param1` is modified.

The rule is not violated when the parameter is a pointer param2 and *param2 is modified.

Check Information

Group: Functions

Category: Advisory

AGC Category: Readability

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 18.1

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

Description

Rule Definition

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

Rationale

Using an invalid array subscript can lead to erroneous behavior of the program. Runtime derived array subscripts are especially troublesome because they cannot be easily checked by manual review or static analysis.

The C Standard defines the creation of a pointer to one beyond the end of the array. The rule permits the C Standard. Dereferencing a pointer to one beyond the end of an array causes undefined behavior and is noncompliant.

Polyspace Specification

Polyspace flags this rule during the analysis as:

- Bug Finder — `Array access out-of-bounds` and `Pointer access out-of-bounds`
- Code Prover — `Illegally dereferenced pointer` and `Out of bounds array index`

Message in Report

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

Check Information

Group: Pointers and Arrays

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Dir 4.1 | MISRA C:2012 Rule 18.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 18.2

Subtraction between pointers shall only be applied to pointers that address elements of the same array

Description

Rule Definition

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Rationale

This rule applies to expressions of the form `pointer_expression1 - pointer_expression2`. The behavior is undefined if `pointer_expression1` and `pointer_expression2`:

- Do not point to elements of the same array,
- Or do not point to the element one beyond the end of the array.

Message in Report

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Examples

Subtracting Pointers

```
#include <stddef.h>

void f1 (int32_t *ptr)
{
```

```
int32_t a1[10];
int32_t a2[10];
int32_t *p1 = &a1[ 1];
int32_t *p2 = &a2[10];
ptrdiff_t diff1, diff2, diff3;

diff1 = p1 - a1;    // Compliant
diff2 = p2 - a2;    // Compliant
diff3 = p1 - p2;    // Non-compliant
}
```

In this example, the three subtraction expressions show the difference between compliant and noncompliant pointer subtractions. The `diff1` and `diff2` subtractions are compliant because the pointers point to the same array. The `diff3` subtraction is not compliant because `p1` and `p2` point to different arrays.

Check Information

Group: Pointers and Arrays

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Dir 4.1 | MISRA C:2012 Rule 18.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 18.3

The relational operators `>`, `>=`, `<` and `<=` shall not be applied to objects of pointer type except where they point into the same object

Description

Rule Definition

The relational operators `>`, `>=`, `<`, and `<=` shall not be applied to objects of pointer type except where they point into the same object.

Rationale

If two pointers do not point to the same object, comparisons between the pointers produces undefined behavior.

You can address the element beyond the end of an array, but you cannot access this element.

Message in Report

The relational operators `>`, `>=`, `<` and `<=` shall not be applied to objects of pointer type except where they point into the same object.

Examples

Pointer and Array Comparisons

```
void f1(void){
    int arr1[10];
    int arr2[10];
    int *ptr1 = arr1;

    if(ptr1 < arr2){}    /* Non-compliant */
```



```

    if(ptr1 < arr1){}    /* Compliant */
}

```

In this example, `ptr1` is a pointer to `arr1`. To be compliant with rule 18.3, you can compare only `ptr1` with `arr1`. Therefore, the comparison between `ptr1` and `arr2` is noncompliant.

Structure Comparisons

```

struct limits{
    int lower_bound;
    int upper_bound;
};

void func2(void){
    struct limits lim_1 = { 2, 5 };
    struct limits lim_2 = { 10, 5 };

    if(&lim_1.lower_bound <= &lim_2.upper_bound){} /* Non-compliant */
    if(&lim_1.lower_bound <= &lim_1.upper_bound){} /* Compliant */
}

```

This example defines two `limits` structures, `lim1` and `lim2`, and compares the elements. To be compliant with rule 18.3, you can compare only the structure elements within a structure. The first comparison compares the `lower_bound` of `lim1` and the `upper_bound` of `lim2`. This comparison is noncompliant because the `lim_1.lower_bound` and `lim_2.upper_bound` are elements of two different structures.

Check Information

Group: Pointers and Arrays

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Dir 4.1

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 18.4

The `+`, `-`, `+=` and `-=` operators should not be applied to an expression of pointer type

Description

Rule Definition

The `+`, `-`, `+=` and `-=` operators should not be applied to an expression of pointer type.

Rationale

The preferred form of pointer arithmetic is using the array subscript syntax `ptr[expr]`. This syntax is clear and less prone to error than pointer manipulation. With pointer manipulation, any explicitly calculated pointer value has the potential to access unintended or invalid memory addresses. Array indexing can also access unintended or invalid memory, but it is easier to review.

To a new C programmer, the expression `ptr+1` can be mistakenly interpreted as one plus the address of `ptr`. However, the new memory address depends on the size, in bytes, of the pointer's target. This confusion can lead to unexpected behavior.

When used with caution, pointer manipulation using `++` can be more natural (for instance, sequentially accessing locations during a memory test).

Polyspace Specification

Polyspace flags operations on pointers, for example, `Pointer + Integer`, `Integer + Pointer`, `Pointer - Integer`.

Message in Report

The `+`, `-`, `+=` and `-=` operators should not be applied to an expression of pointer type.

Examples

Pointers and Array Expressions

```
void fun1(void){
    unsigned char arr[10];
    unsigned char *ptr;
    unsigned char index = 0U;

    index = index + 1U;    /* Compliant - rule only applies to pointers */

    arr[index] = 0U;      /* Compliant */
    ptr = &arr[5];       /* Compliant */
    ptr = arr;
    ptr++;                /* Compliant - increment operator not + */
    *(ptr + 5) = 0U;     /* Non-compliant */
    ptr[5] = 0U;         /* Compliant */
}
```

This example shows various operations with pointers and arrays. The only operation in this example that is noncompliant is using the + operator directly with a pointer (line 12).

Adding Array Elements Inside a `for` Loop

```
void fun2(void){
    unsigned char array_2_2[2][2] = {{1U, 2U}, {4U, 5U}};
    unsigned char i = 0U;
    unsigned char j = 0U;
    unsigned char sum = 0U;

    for(i = 0u; i < 2U; i++){
        unsigned char *row = array_2_2[ i ];

        for(j = 0u; j < 2U; j++){
            sum += row[ j ];                /* Compliant */
        }
    }
}
```

In this example, the second `for` loop uses the array pointer `row` in an arithmetic expression. However, this usage is compliant because it uses the array index form.

Pointers and Array Expressions

```
void fun3(unsigned char *ptr1, unsigned char ptr2[ ]){
    ptr1++;           /* Compliant */
    ptr1 = ptr1 - 5;  /* Non-compliant */
    ptr1 -= 5;        /* Non-compliant */
    ptr1[2] = 0U;     /* Compliant */

    ptr2++;           /* Compliant */
    ptr2 = ptr2 + 3;  /* Non-compliant */
    ptr2 += 3;        /* Non-compliant */
    ptr2[3] = 0U;     /* Compliant */
}
```

This example shows the offending operators used on pointers and arrays. Notice that the same types of expressions are compliant and noncompliant for both pointers and arrays.

If `ptr1` does not point to an array with at least six elements, and `ptr2` does not point to an array with at least 4 elements, this example violates rule 18.1.

Check Information

Group: Pointers and Arrays

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 18.1 | MISRA C:2012 Rule 18.2

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 18.5

Declarations should contain no more than two levels of pointer nesting

Description

Rule Definition

Declarations should contain no more than two levels of pointer nesting.

Rationale

The use of more than two levels of pointer nesting can seriously impair the ability to understand the behavior of the code. Avoid this usage.

Message in Report

Declarations should contain no more than two levels of pointer nesting.

Examples

Pointer Nesting

```
typedef char *INTPTR;

void function(char ** arrPar[ ])    /* Non-compliant - 3 levels */
{
    char    **  obj2;                /* Compliant */
    char    *** obj3;                /* Non-compliant */
    INTPTR *  obj4;                /* Compliant */
    INTPTR * const * const obj5;    /* Non-compliant */
    char    ** arr[10];             /* Compliant */
    char    ** (*parr)[10];         /* Compliant */
    char    * (**pparr)[10];        /* Compliant */
}
```

```
struct s{
    char *   s1;           /* Compliant */
    char **  s2;           /* Compliant */
    char *** s3;           /* Non-compliant */
};

struct s *   ps1;         /* Compliant */
struct s **  ps2;         /* Compliant */
struct s *** ps3;         /* Non-compliant */

char ** ( *pfunc1)(void); /* Compliant */
char ** ( **pfunc2)(void); /* Compliant */
char ** (***pfunc3)(void); /* Non-compliant */
char *** ( **pfunc4)(void); /* Non-compliant */
```

This example shows various pointer declarations and nesting levels. Any pointer with more than two levels of nesting is considered noncompliant.

Check Information

Group: Pointers and Arrays

Category: Advisory

AGC Category: Readability

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 18.6

The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist

Description

Rule Definition

The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.

Rationale

The address of an object becomes indeterminate when the lifetime of that object expires. Any use of an indeterminate address results in undefined behavior.

Polyspace Specification

Polyspace flags a violation when assigning an address to a global variable, returning a local variable address, or returning a parameter address.

Message in Report

The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.

Examples

Address of Local Variables

```
char *func(void){
    char local_auto;
    return &local_auto /* Non-compliant
```



```

        * &local_auto is indeterminate */
    }

```

In this example, because `local_auto` is a local variable, after the function returns, the address of `local_auto` is indeterminate.

Copying Pointer Addresses to Local Variables

```

char *sp;

void f(unsigned short u){
    g(&u);
}

void g(unsigned short *p){
    sp = p; /* Non-compliant
            * the parameter u from f is copied to static sp */
}

void h(void){
    static unsigned short *q;

    unsigned short x =0u;
    q = &x; /* Non-compliant -
            * &x stored in object with greater lifetime */
}

```

In this example, the function `g` stores a copy of its pointer parameter `p`. If `p` always points to an object with static storage duration, then the code is compliant with this rule. However, in this example, `p` points to an object with automatic storage duration. In such a case, copying the parameter `p` is noncompliant.

Check Information

Group: Pointers and Arrays

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 18.7

Flexible array members shall not be declared

Description

Rule Definition

Flexible array members shall not be declared.

Rationale

Flexible array members are usually used with dynamic memory allocation. Dynamic memory allocation is banned by Directive 4.12 and Rule 21.3 on page 6-329.

Message in Report

Flexible array members shall not be declared.

Check Information

Group: Pointers and Arrays

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 21.3

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 18.8

Variable-length array types shall not be used

Description

Rule Definition

Variable-length array types shall not be used.

Rationale

When the size of an array declared in a block or function prototype is not an integer constant expression, you specify variable array types. Variable array types are typically implemented as a variable size object stored on the stack. Using variable type arrays can make it impossible to determine statistically the amount of memory for the stack requires.

If the size of a variable-length array is negative or zero, the behavior is undefined.

If a variable-length array must be compatible with another array type, then the size of the array types must be identical and positive integers. If your array does not meet these requirements, the behavior is undefined.

If you use a variable-length array type in a `sizeof`, it is uncertain if the array size is evaluated or not.

Message in Report

Variable-length array types shall not be used.

Check Information

Group: Pointers and Arrays

Category: Required

AGC Category: Required
Language: C99

See Also

MISRA C:2012 Rule 13.6

Topics

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 19.1

An object shall not be assigned or copied to an overlapping object

Description

Rule Definition

An object shall not be assigned or copied to an overlapping object.

Rationale

When you assign an object to another object with overlapping memory, the behavior is undefined. The exceptions are:

- You assign an object to another object with exactly overlapping memory and compatible type.
- You copy one object to another using `memmove`.

Message in Report

- An object shall not be assigned or copied to an overlapping object.
- Destination and source of XX overlap, the behavior is undefined.

Examples

Assignment of Unions

```
void func (void) {  
    union {  
        short i;  
        int j;  
    } a = {0}, b = {1};  
}
```

```
    a.j = a.i;    /* Non-compliant */
    a = b;       /* Compliant */
}
```

In this example, the rule is violated when `a.i` is assigned to `a.j` because the two variables have overlapping regions of memory.

Assignment of Array Segments

```
#include <string.h>

int arr[10];

void func(void) {
    memcpy (&arr[5], &arr[4], 2u * sizeof(arr[0]));    /* Non-compliant */
    memcpy (&arr[5], &arr[4], sizeof(arr[0]));        /* Compliant */
    memcpy (&arr[1], &arr[4], 2u * sizeof(arr[0]));    /* Compliant */
}
```

In this example, memory equal to twice `sizeof(arr[0])` is the memory space taken up by two array elements. If that memory space begins from `&a[4]` and `&a[5]`, the two memory regions overlap. The rule is violated when the `memcpy` function is used to copy the contents of these two overlapping memory regions.

Check Information

Group: Overlapping Storage

Category: Mandatory

AGC Category: Mandatory

Language: C90, C99

See Also

MISRA C:2012 Rule 19.2

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 19.2

The union keyword should not be used

Description

Rule Definition

The union keyword should not be used.

Rationale

If you write to a union member and read the same union member, the behavior is well-defined. But if you read a different member, the behavior depends on the relative sizes of the members. For instance:

- If you read a union member with wider memory size, the value you read is unspecified.
- Otherwise, the value is implementation-dependant.

Message in Report

The union keyword should not be used.

Examples

Possible Problems with `union` Keyword

```
unsigned int zext(unsigned int s)
{
    union                /* Non-compliant */
    {
        unsigned int ul;
        unsigned short us;
    }
}
```

```
    } tmp;

    tmp.us = s;
    return tmp.ul;          /* Unspecified value */
}
```

In this example, the 16-bit short field `tmp.us` is written but the wider 32-bit int field `tmp.ul` is read. Using the `union` keyword can cause such unspecified behavior. Therefore, the rule forbids using the `union` keyword.

Check Information

Group: Overlapping Storage

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 19.1

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 20.1

`#include` directives should only be preceded by preprocessor directives or comments

Description

Rule Definition

#include directives should only be preceded by preprocessor directives or comments.

Rationale

For better code readability, group all `#include` directives in a file at the top of the file. Undefined behavior can occur if you use `#include` to include a standard header file within a declaration or definition, or if you use part of the Standard Library before including the related standard header files.

Polyspace Specification

Polyspace flags text that precedes a `#include` directive. Polyspace ignores preprocessor directives, comments, spaces, or "new lines".

Message in Report

`#include` directives should only be preceded by preprocessor directives or comments.

Check Information

Group: Preprocessing Directives

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

Topics

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.2

The ', " or \ characters and the /* or // character sequences shall not occur in a header file name

Description

Rule Definition

The ', " or \ characters and the / or // character sequences shall not occur in a header file name.*

Rationale

The program's behavior is undefined if:

- You use ', ", \, /* or // between < > delimiters in a header name preprocessing token.
- You use ', \, /* or // between " delimiters in a header name preprocessing token.

Although \ results in undefined behavior, many implementations accept / in its place.

Polyspace Specification

Polyspace flags the characters ', ", \, /* or // between < and > in #include <filename>.

Polyspace flags the characters ', \, /* or // between " and " in #include "filename".

Message in Report

The ', " or \ characters and the /* or // character sequences shall not occur in a header file name.

Check Information

Group: Preprocessing Directives

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.3

The `#include` directive shall be followed by either a `<filename>` or `\ "filename\"` sequence

Description

Rule Definition

The `#include` directive shall be followed by either a `<filename>` or `"filename"` sequence.

Rationale

This rule applies only after macro replacement.

The behavior is undefined if an `#include` directive does not use one of the following forms:

- `#include <filename>`
- `#include "filename"`

Message in Report

- `'#include'` expects `\ "FILENAME\"` or `<FILENAME>`
- `'#include_next'` expects `\ "FILENAME\"` or `<FILENAME>`
- `'#include'` does not expect string concatenation.
- `'#include_next'` does not expect string concatenation.

Check Information

Group: Preprocessing Directives

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.4

A macro shall not be defined with the same name as a keyword

Description

Rule Definition

A macro shall not be defined with the same name as a keyword.

Rationale

Using macros to change the meaning of keywords can be confusing. The behavior is undefined if you include a standard header while a macro is defined with the same name as a keyword.

Message in Report

- The macro *macro_name* shall not be redefined.
- The macro *macro_name* shall not be undefined.

Examples

Redefining `int` keyword

```
#define int some_other_type
        /* Non-compliant - int keyword behavior altered */
#include <stdlib.h>
...
```

In this example, the `#define` violates Rule 20.4 because it alters the behavior of the `int` keyword. The inclusion of the standard header results in undefined behavior.

One possible correction is to use a different keyword:

```
#define int_mine some_other_type
#include <stdlib.h>
...
```

Redefining keywords versus statements

```
#define while(E) for ( ; (E) ; ) /* Non-compliant - while redefined*/
#define unless(E) if ( !(E) ) /* Compliant*/

#define seq(S1, S2) do{ S1; S2;} while(false) /* Compliant*/
#define compound(S) {S;} /* Compliant*/
...
```

In this example, it is noncompliant to redefine the keyword `while`, but it is compliant to define a macro that expands to statements.

Redefining keywords in different standards

```
#define inline
```

In this example, redefining `inline` is compliant in C90, but not in C99 because `inline` is not a keyword in C90.

Check Information

Group: Preprocessing Directives

Category: Required

AGC Category: Required

Languages: C90, C99

See Also

MISRA C:2012 Rule 21.1

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.5

#undef should not be used

Description

Rule Definition

#undef should not be used.

Rationale

#undef can make the software unclear which macros exist at a particular point within a translation unit.

Message in Report

#undef shall not be used.

Check Information

Group: Preprocessing Directives

Category: Advisory

AGC Category: Readability

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.6

Tokens that look like a preprocessing directive shall not occur within a macro argument

Description

Rule Definition

Tokens that look like a preprocessing directive shall not occur within a macro argument.

Rationale

An argument containing sequences of tokens that otherwise act as preprocessing directives leads to undefined behavior.

Polyspace Specification

Polyspace looks for the # character in a macro arguments (outside a string or character constant).

Message in Report

Macro argument shall not look like a preprocessing directive.

Examples

Macro Expansion Causing Non-Compliance

```
#define M( A ) printf ( #A )

#include <stdio.h>

void foo(void){
    M(
```

```
#ifndef SW          /* Non-compliant */
    "Message 1"
#else
    "Message 2"    /* Compliant - SW not defined */
#endif            /* Non-compliant */
    );
}
```

This example shows a macro definition and the macro usage. `#ifndef SW` and `#endif` are noncompliant because they look like a preprocessing directive. Polyspace does not flag `#else "Message 2"` because after macro expansion, Polyspace knows `SW` is not defined. The expanded macro is `printf ("\nMessage 2\n");`

Check Information

Group: Preprocessing Directives

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.7

Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses

Description

Rule Definition

Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.

Rationale

If you do not use parentheses, then it is possible that operator precedence does not give the results that you want when macro substitution occurs.

If you are not using a macro parameter as an expression, then the parentheses are not necessary because no operators are involved in the macro.

Message in Report

Expanded macro parameter *param* shall be enclosed in parentheses.

Examples

Macro Expressions

```
#define mac1(x, y) (x * y)
#define mac2(x, y) ((x) * (y))

void foo(void){
    int r;

    r = mac1(1 + 2, 3 + 4);          /* Non-compliant */
```

```
    r = mac1((1 + 2), (3 + 4)); /* Compliant */  
  
    r = mac2(1 + 2, 3 + 4);      /* Compliant */  
}
```

In this example, `mac1` and `mac2` are two defined macro expressions. The definition of `mac1` does not enclose the arguments in parentheses. In line 7, the macro expands to `r = (1 + 2 * 3 + 4);` This expression can be `(1 + (2 * 3) + 4)` or `(1 + 2) * (3 + 4)`. However, without parentheses, the program does not know the intended expression. Line 8 uses parentheses, so the line expands to `(1 + 2) * (3 + 4)`. This macro expression is compliant.

The definition of `mac2` does enclose the argument in parentheses. Line 10 (the same macro arguments in line 7) expands to `(1 + 2) * (3 + 4)`. This macro and macro expression are compliant.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Dir 4.9

Topics

“Set Up Coding Rules Checking”
“Review Coding Rule Violations”
“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.8

The controlling expression of a `#if` or `#elif` preprocessing directive shall evaluate to 0 or 1

Description

Rule Definition

The controlling expression of a `#if` or `#elif` preprocessing directive shall evaluate to 0 or 1.

Rationale

Strong typing requires that conditional inclusion preprocessing directives, `#if` or `#elif`, have a controlling expression that evaluates to a Boolean value.

Message in Report

The controlling expression of a `#if` or `#elif` preprocessing directive shall evaluate to 0 or 1.

Check Information

Group: Preprocessing Directives

Category: Required

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 14.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.9

All identifiers used in the controlling expression of `#if` or `#elif` preprocessing directives shall be `#define'd` before evaluation

Description

Rule Definition

All identifiers used in the controlling expression of `#if` or `#elif` preprocessing directives shall be `#define'd` before evaluation.

Rationale

If attempt to use a macro identifier in a preprocessing directive, and you have not defined that identifier, then the preprocessor assumes that it has a value of zero. This value might not meet developer expectations.

Message in Report

Identifier is not defined.

Examples

Macro Identifiers

```
#if M == 0                                /* Non-compliant - Not defined */
#endif

#if defined (M)                            /* Compliant - M is not evaluate */
#if M == 0                                  /* Compliant - M is known to be defined */
#endif
#endif

#if defined (M) && (M == 0)                /* Compliant
```

```
                                * if M defined, M evaluated in ( M == 0 ) */  
#endif
```

This example shows various uses of `M` in preprocessing directives. The second and third `#if` clauses check to see if the software defines `M` before evaluating `M`. The first `#if` clause does not check to see if `M` is defined, and because `M` is not defined, the statement is noncompliant.

Check Information

Group: Preprocessing Directives

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Dir 4.9

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.10

The # and ## preprocessor operators should not be used

Description

Rule Definition

The # and ## preprocessor operators should not be used.

Rationale

The order of evaluation associated with multiple #, multiple ##, or a mix of # and ## preprocessor operators is unspecified. In some cases, it is therefore not possible to predict the result of macro expansion.

The use of ## can result in obscured code.

Message in Report

The # and ## preprocessor operators should not be used.

Check Information

Group: Preprocessing Directives

Category: Advisory

AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 1.3 | MISRA C:2012 Rule 20.11

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”
“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.11

A macro parameter immediately following a # operator shall not immediately be followed by a ## operator

Description

Rule Definition

A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.

Rationale

The order of evaluation associated with multiple #, multiple ##, or a mix of # and ## preprocessor operators, is unspecified. Rule 20.10 discourages the use of # and ##. The result of a # operator is a string literal. It is extremely unlikely that pasting this result to any other preprocessing token results in a valid token.

Message in Report

The ## preprocessor operator shall not follow a macro parameter following a # preprocessor operator.

Examples

Use of # and

```
#define A( x )    #x                /* Compliant */
#define B( x, y ) x ## y           /* Compliant */
#define C( x, y ) #x ## y         /* Non-compliant */
```

In this example, you can see three uses of the # and ## operators. You can use these preprocessing operators alone (line 1 and line 2), but using # then ## is noncompliant (line 3).

Check Information

Group: Preprocessing Directives

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 20.10

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.12

A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators

Description

Rule Definition

A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.

Rationale

The parameter to # or ## is not expanded prior to being used. The same parameter appearing elsewhere in the replacement text is expanded. If the macro parameter is itself subject to macro replacement, its use in mixed contexts within a macro replacement might not meet developer expectations.

Message in Report

Expanded macro parameter *param1* is also an operand of *op* operator.

Check Information

Group: Preprocessing Directives

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”
“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.13

A line whose first token is # shall be a valid preprocessing directive

Description

Rule Definition

A line whose first token is # shall be a valid preprocessing directive

Rationale

You typically use a preprocessing directive to conditionally exclude source code until a corresponding #else, #elif, or #endif directive is encountered. If your compiler does not detect a preprocessing directive because it is malformed or invalid, you can end up excluding more code than you intended.

If all preprocessing directives are syntactically valid, even in excluded code, this unintended code exclusion cannot happen.

Message in Report

Directive is not syntactically meaningful.

Check Information

Group: Preprocessing Directives

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.14

All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if`, `#ifdef` or `#ifndef` directive to which they are related

Description

Rule Definition

All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if`, `#ifdef` or `#ifndef` directive to which they are related.

Rationale

When conditional compilation directives include or exclude blocks of code and are spread over multiple files, confusion arises. If you terminate an `#if` directive within the same file, you reduce the visual complexity of the code and the chances of an error.

If you terminate `#if` directives within the same file, you can use `#if` directives in included files

Message in Report

- `'#else'` not within a conditional.
- `'#elsif'` not within a conditional.
- `'#endif'` not within a conditional. unterminated conditional directive.

Check Information

Group: Preprocessing Directives

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 21.1

#define and #undef shall not be used on a reserved identifier or reserved macro name

Description

Rule Definition

#define and #undef shall not be used on a reserved identifier or reserved macro name.

Rationale

Reserved identifiers and reserved macro names are intended for use by the implementation. Removing or changing the meaning of a reserved macro can result in undefined behavior. This rule applies to the following:

- Identifiers or macro names beginning with an underscore
- Identifiers in file scope described in the C Standard Library (ISO/IEC 9899:1999, Section 7, "Library")
- Macro names described in the C Standard Library as being defined in a standard header (ISO/IEC 9899:1999, Section 7, "Library").

Message in Report

- The macro *macro_name* shall not be redefined.
- The macro *macro_name* shall not be undefined.
- The macro *macro_name* shall not be defined.

Examples

Defining or undefining Reserved Identifiers

```
#undef __LINE__          /* Non-compliant - begins with _ */
#define _Guard_H 1      /* Non-compliant - begins with _ */
```

```
#undef _ BUILTIN_sqrt          /* Non-compliant - implementation may
                               * use _BUILTIN_sqrt for other purposes,
                               * e.g. generating a sqrt instruction */
#define defined                /* Non-compliant - reserved identifier */
#define errno my_errno        /* Non-compliant - library identifier */
#define isneg(x) ( (x) < 0 ) /* Compliant - rule doesn't include
                               * future library directions */
```

Check Information

Group: Standard Libraries

Category: Required

AGC Category: Required

Languages: C90, C99

See Also

MISRA C:2012 Rule 20.4

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.2

A reserved identifier or macro name shall not be declared

Description

Rule Definition

A reserved identifier or macro name shall not be declared.

Rationale

The Standard allows implementations to treat reserved identifiers specially. If you reuse reserved identifiers, you can cause undefined behavior.

Polyspace Specification

- If you define a macro name that corresponds to a standard library macro, object, or function, rule 21.1 is violated.
- The rule considers tentative definitions as definitions.

Message in Report

Identifier 'XX' shall not be reused.

Check Information

Group: Standard Libraries

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.3

The memory allocation and deallocation functions of `<stdlib.h>` shall not be used

Description

Rule Definition

The memory allocation and deallocation functions of `<stdlib.h>` shall not be used.

Rationale

Using memory allocation and deallocation routines can cause undefined behavior. For instance:

- You free memory that you had not allocated dynamically.
- You use a pointer that points to a freed memory location.

Polyspace Specification

If you use names of dynamic heap memory allocation functions for macros, and you expand the macros in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro `<name>` shall not be used.
- Identifier `XX` should not be used.

Examples

Use of `malloc`, `calloc`, `realloc` and `free`

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

```
static int foo(void);

typedef struct struct_1 {
    int a;
    char c;
} S_1;

static int foo(void) {

    _S_1 * ad_1;
    int * ad_2;
    int * ad_3;

    ad_1 = (S_1*)calloc(100U, sizeof(S_1));           /* Non-compliant */
    ad_2 = malloc(100U * sizeof(int));                /* Non-compliant */
    ad_3 = realloc(ad_3, 60U * sizeof(long));        /* Non-compliant */

    free(ad_1);                                       /* Non-compliant */
    free(ad_2);                                       /* Non-compliant */
    free(ad_3);                                       /* Non-compliant */

    return 1;
}
```

In this example, the rule is violated when the functions `malloc`, `calloc`, `realloc` and `free` are used.

Check Information

Group: Standard Libraries

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 18.7

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.4

The standard header file `<setjmp.h>` shall not be used

Description

Rule Definition

The standard header file `<setjmp.h>` shall not be used.

Rationale

Using `setjmp` and `longjmp`, you can bypass normal function call mechanisms and cause undefined behavior.

Polyspace Specification

If the `longjmp` function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '`<name>`' shall not be used.
- Identifier `XX` should not be used.

Check Information

Group: Standard Libraries

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.5

The standard header file <signal.h> shall not be used

Description

Rule Definition

The standard header file <signal.h> shall not be used.

Rationale

Using signal handling functions can cause implementation-defined and undefined behavior.

Polyspace Specification

If the signal function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '<name>' shall not be used.
- Identifier XX should not be used.

Check Information

Group: Standard Libraries

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.6

The Standard Library input/output functions shall not be used

Description

Rule Definition

The Standard Library input/output functions shall not be used.

Rationale

This rule applies to the functions that are provided by `<stdio.h>` and in C99, their character-wide equivalents provided by `<wchar.h>`. Using these functions can cause unspecified, undefined and implementation-defined behavior.

Polyspace Specification

If the Standard Library function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '`<name>`' shall not be used.
- Identifier `XX` should not be used.

Check Information

Group: Standard Libraries

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.7

The `atof`, `atoi`, `atol`, and `atoll` functions of `<stdlib.h>` shall not be used

Description

Rule Definition

The `atof`, `atoi`, `atol`, and `atoll` functions of `<stdlib.h>` shall not be used.

Rationale

When a string cannot be converted, the behavior of these functions can be undefined.

Polyspace Specification

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '`<name>`' shall not be used.
- Identifier `XX` should not be used.

Check Information

Group: Standard Libraries

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.8

The library functions `abort`, `exit`, `getenv` and `system` of `<stdlib.h>` shall not be used

Description

Rule Definition

The library functions `abort`, `exit`, `getenv` and `system` of `<stdlib.h>` shall not be used.

Rationale

Using these functions can cause undefined and implementation-defined behaviors.

Polyspace Specification

In case the `abort`, `exit`, `getenv`, and `system` functions are actually macros, and the macros are expanded in the code, this rule is detected as violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '`<name>`' shall not be used.
- Identifier `XX` should not be used.

Check Information

Group: Standard Libraries

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.9

The library functions `bsearch` and `qsort` of `<stdlib.h>` shall not be used

Description

Rule Definition

The library functions `bsearch` and `qsort` of `<stdlib.h>` shall not be used.

Rationale

The comparison function in these library functions can behave inconsistently when the elements being compared are equal. Also, the implementation of `qsort` can be recursive and place unknown demands on the call stack.

Polyspace Specification

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '`<name>`' shall not be used.
- Identifier `XX` should not be used.

Check Information

Group: Standard Libraries

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.10

The Standard Library time and date functions shall not be used

Description

Rule Definition

The Standard Library time and date functions shall not be used.

Rationale

Using these functions can cause unspecified, undefined and implementation-defined behavior.

Polyspace Specification

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '<name>' shall not be used.
- Identifier XX should not be used.

Check Information

Group: Standard Libraries

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.11

The standard header file `<tgmath.h>` shall not be used

Description

Rule Definition

The standard header file `<tgmath.h>` shall not be used.

Rationale

Using the facilities of this header file can cause undefined behavior.

Polyspace Specification

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '`<name>`' shall not be used.
- Identifier `XX` should not be used.

Examples

Use of Function in `tgmath.h`

```
#include <tgmath.h>

float f1, res;

void func(void) {
```

```
    res = sqrt(f1); /* Non-compliant */  
}
```

In this example, the rule is violated when the `sqrt` macro defined in `tgmath.h` is used.

For this example, one possible correction is to use the function `sqrtf` defined in `math.h` for float arguments.

```
#include <math.h>  
  
float f1, res;  
  
void func(void) {  
    res = sqrtf(f1);  
}
```

Check Information

Group: Standard Libraries

Category: Required

AGC Category: Required

Language: C90, C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.12

The exception handling features of `<fenv.h>` should not be used

Description

Rule Definition

The exception handling features of `<fenv.h>` should not be used.

Rationale

In some cases, the values of the floating-point status flags are unspecified. Attempts to access them can cause undefined behavior.

Message in Report

The exception handling features of `<fenv.h>` should not be used

Examples

Use of Features in `<fenv.h>`

```
#include <fenv.h>

void func(float x, float y) {
    float z;

    feclearexcept(FE_DIVBYZERO);           /* Non-compliant */
    z = x/y;

    if(fetestexcept (FE_DIVBYZERO)) {     /* Non-compliant */
    }
    else {
#pragma STDC FENV_ACCESS ON
```

```
        z=x*y;
        if(z>x) {
#pragma STDC FENV_ACCESS OFF
            if(fetestexcept (FE_OVERFLOW)) { /* Non-compliant */
                }
            }
        }
    }
```

In this example, the rule is violated when the identifiers `feclearexcept` and `fetestexcept`, and the macros `FE_DIVBYZERO` and `FE_OVERFLOW` are used.

Check Information

Group: Standard libraries

Category: Advisory

AGC Category: Advisory

Language: C99

See Also

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 21.15

The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types

Description

Rule Definition

The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types.

Rationale

The functions

```
memcpy( arg1, arg2, num_bytes );  
memmove( arg1, arg2, num_bytes );  
memcmp( arg1, arg2, num_bytes );
```

perform a byte-by-byte copy, move or comparison between the memory locations that `arg1` and `arg2` point to. A byte-by-byte copy, move or comparison is meaningful only if `arg1` and `arg2` have compatible types.

Using pointers to different data types for `arg1` and `arg2` typically indicates a coding error.

Message in Report

The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types.

Examples

Incompatible Argument Types for `memcpy`

```
void f ( uint8_t s1[ 8 ], uint16_t s2[ 8 ] )
{
    ( void ) memcpy ( s1, s2, 8 ); /* Non-compliant */
}
```

In this example, `s1` and `s2` are pointers to different data types. The `memcpy` statement copies eight bytes from one buffer to another.

Eight bytes represent the entire span of the buffer that `s1` points to, but only part of the buffer that `s2` points to. Therefore, the `memcpy` statement copies only part of `s2` to `s1`, which might be unintended.

Check Information

Group: Standard libraries

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 21.16

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Rule 21.16

The pointer arguments to the Standard Library function `memcmp` shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type

Description

Rule Definition

The pointer arguments to the Standard Library function `memcmp` shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type.

Rationale

The Standard Library function

```
memcmp ( lhs, rhs, num );
```

performs a byte-by-byte comparison of the first `num` bytes of the two objects that `lhs` and `rhs` point to.

Do not use `memcmp` for a byte-by-byte comparison of the following.

Type	Rationale
Structures	If members of a structure have different data types, your compiler introduces additional padding for data alignment in memory. The content of these extra padding bytes is meaningless. If you perform a byte-by-byte comparison of structures with <code>memcmp</code> , you compare even the meaningless data stored in the padding. You might reach the false conclusion that two data structures are not equal, even if their corresponding members have the same value.

Type	Rationale
Objects with essentially floating type	The same floating point value can be stored using different representations. If you perform a byte-by-byte comparison of two variables with <code>memcmp</code> , you can reach the false conclusion that the variables are unequal even when they have the same value. The reason is that the values are stored using two different representations.
Essentially char arrays	Essentially char arrays are typically used to store strings. In strings, the content in bytes after the null terminator is meaningless. If you perform a byte-by-byte comparison of two strings with <code>memcmp</code> , you might reach the false conclusion that two strings are not equal, even if the bytes before the null terminator store the same value.

Message in Report

The pointer arguments to the Standard Library function `memcmp` shall point to either a pointer type, an *essentially signed* type, an *essentially unsigned* type, an *essentially Boolean* type or an *essentially enum* type.

Examples

Using `memcmp` for Comparison of Structures, Unions, and *essentially char* Arrays

```

struct S;
bool_t f1 ( struct S *s1, struct S *s2 )
{
    return ( memcmp ( s1, s2, sizeof ( struct S ) ) != 0 ); /* Non-compliant */
}

union U
{
    uint32_t range;
    uint32_t height;
};
bool_t f2 ( union U *u1, union U *u2 )
{
    return ( memcmp ( u1, u2, sizeof ( union U ) ) != 0 ); /* Non-compliant */
}

```

```
const char a[ 6 ] = "task";
bool_t f3 ( const char b[ 6 ] )
{
    return ( memcmp ( a, b, 6 ) != 0 ); /* Non-compliant */
}
```

In this example:

- Structures `s1` and `s2` are compared in the `bool_t f1` function. The return value of this function might indicate that `s1` and `s2` are different due to padding. This comparison is noncompliant.
- Unions `u1` and `u2` are compared in the `bool_t f2` function. The return value of this function might indicate that `u1` and `u2` are the same due to unintentional comparison of `u1.range` and `u2.height`, or `u1.height` and `u2.range`. This comparison is noncompliant.
- Essentially char arrays `a` and `b` are compared in the `bool_t f3` function. The return value of this function might incorrectly indicate that the strings are different because the length of `a` (four) is less than the number of bytes compared (six). This comparison is noncompliant.

Check Information

Group: Standard libraries

Category: Required

AGC Category: Required

Language: C90, C99

See Also

MISRA C:2012 Rule 21.15

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Rule 22.5

A pointer to a `FILE` object shall not be dereferenced

Description

Rule Definition

A pointer to a `FILE` object shall not be dereferenced.

Rationale

The Standard states that the address of a `FILE` object used to control a stream can be significant. Copying that object might not give the same behavior. This rule ensures that you cannot perform such a copy.

Directly manipulating a `FILE` object might be incompatible with its use as a stream designator.

Message in Report

A pointer to a `FILE` object shall not be dereferenced

Examples

FILE* Pointer Dereferenced

```
#include <stdio.h>

void func(void) {
    FILE *pf1;
    FILE *pf2;
    FILE f3;

    pf2 = pf1;          /* Compliant */
}
```

```
f3 = *pf2;          /* Non-compliant */  
pf2->_flags=0;     /* Non-compliant */  
}
```

In this example, the rule is violated when the FILE* pointer pf2 is dereferenced.

Check Information

Group: Resources

Category: Mandatory

AGC Category: Mandatory

Language: C90, C99

See Also

MISRA C:2012 Rule 21.6

Topics

“Set Up Coding Rules Checking”

“Review Coding Rule Violations”

“Polyspace MISRA C:2012 Checker”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

Code Metrics

Comment Density

Ratio of number of comments to number of statements

Description

The metric specifies the ratio of comments to statements expressed as a percentage.

Multi-line comments are counted as one comment. A statement typically ends with a semi-colon with some exceptions. Exceptions include semi-colons in `for` loops or structure field declarations.

The recommended lower limit for this metric is 20. For better readability of your code, try to place at least one comment for every five statements.

To enforce limits on metrics:

- In the Polyspace user interface, see “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

Examples

Comment Density Calculation

```
struct record {
    char name[40];
    long double salary;
    int isEmployed;
};

struct record dataBase[100];

struct record fetch(void);
void remove(int);

void maintenanceRoutines() {
```

```
// This function implements
// regular maintenance on an internal database
int i;
struct record tempRecord;

for(i=0; i <100; i++) {
    tempRecord = fetch(); // This function fetches a record
    // from the database
    if(tempRecord.isEmployed == 0)
        remove(i); // Remove employee record
    //from the database
}
}
```

In this example, the comment density is 38. The calculation is done as follows:

Code	Running Total of Comments	Running Total of Statements
struct record { char name[40]; long double salary; int isEmployed; };	0	1
struct record dataBase[100]; struct record fetch(void); void remove(int);	0	4
void maintenanceRoutines() {	0	4
// This function implements // regular maintenance on an internal database	1	4
int i; struct record tempRecord;	1	6
for(i=0; i <100; i++) {	1	6
tempRecord = fetch(); // This function fetches a record // from the database	2	7

Code	Running Total of Comments	Running Total of Statements
<pre>if(tempRecord.isEmployed == 0) remove(i); // Remove employee record //from the database } }</pre>	3	8

There are 3 comments and 8 statements. The comment density is $3/8 * 100 = 38$.

Metric Information

Group: File

Acronym: COMF

HIS Metric: Yes

See Also

Calculate code metrics (-code-metrics)

Cyclomatic Complexity

Number of linearly independent paths in function body

Description

This metric calculates the number of decision points in a function and adds one to the total. A decision point is a statement that causes your program to branch into two paths.

The recommended upper limit for this metric is 10. If the cyclomatic complexity is high, the code is both difficult to read and can cause more orange checks. Therefore, try to limit the value of this metric.

To enforce limits on metrics:

- In the Polyspace user interface, see “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

Computation Details

The metric calculation uses the following rules to identify decision points:

- An `if` statement is one decision point.
- The statements `for` and `while` count as one decision point, even when no condition is evaluated, for example, in infinite loops.
- Boolean combinations (`&&`, `||`) do not count as decision points.
- `case` statements do not count as decision points unless they are followed by a `break` statement. For instance, this code has a cyclomatic complexity of two:

```
switch(num) {
    case 0:
    case 1:
    case 2:
        break;
    case 3:
    case 4:
```

```
}
```

- The calculation is done after preprocessing:
 - Macros are expanded.
 - Conditional compilation is applied. The blocks hidden by preprocessing directives are ignored.

Examples

Function with Nested `if` Statements

```
int foo(int x,int y)
{
    int flag;
    if (x <= 0)
        /* Decision point 1*/
        flag = 1;
    else
    {
        if (x < y )
            /* Decision point 2*/
            flag = 1;
        else if (x==y)
            /* Decision point 3*/
            flag = 0;
        else
            flag = -1;
    }
    return flag;
}
```

In this example, the cyclomatic complexity of `foo` is 4.

Function with `?` Operator

```
int foo (int x, int y) {
    if((x <0) || (y < 0))
        /* Decision point 1*/
        return 0;
```

```

    else
        return (x > y ? x : y);
        /* Decision point 2*/
}

```

In this example, the cyclomatic complexity of `foo` is 3. The `?` operator is the second decision point.

Function with `switch` Statement

```

#include <stdio.h>

int foo(int x,int y, int ch)
{
    int val = 0;
    switch(ch) {
    case 1:
        /* Decision point 1*/
        val = x + y;
        break;
    case 2:
        /* Decision point 2*/
        val = x - y;
        break;
    default:
        printf("Invalid choice.");
    }
    return val;
}

```

In this example, the cyclomatic complexity of `foo` is 3.

Function with Nesting of Different Control-Flow Statements

```

int foo(int x,int y, int bound)
{
    int count = 0;
    if (x <= y)
        /* Decision point 1*/
        count = 1;
    else
        while(x>y) {
            /* Decision point 2*/

```

```
        x--;
        if(count < bound) {
            /* Decision point 3*/
            count++;
        }
    }
    return count;
}
```

In this example, the cyclomatic complexity of `foo` is 4.

Metric Information

Group: Function

Acronym: VG

HIS Metric: Yes

See Also

Calculate code metrics (`-code-metrics`)

Estimated Function Coupling

Measure of complexity between levels of call tree

Description

This metric provides an approximate measure of complexity between different levels of the call tree. The metric is defined as:

number of call occurrences – number of function definitions + 1

If there are more function definitions than function calls, the estimated function coupling result is negative.

This metric:

- Counts function calls and function definitions in the current file only.
 - It does not count function definitions in a header file included in the current file.
- Treats `static` and `inline` functions like any other function.

Examples

Same Function Called Multiple Times

```
void checkBounds(int *);
int getUnboundedValue();

int getBoundedValue(void) {
    int num = getUnboundedValue();
    checkBounds(&num);
    return num;
}

void main() {
    int input1=getBoundedValue(), input2= getBoundedValue(), prod;
    prod = input1 * input2;
```

```
        checkBounds (&prod);
    }
```

In this example, there are:

- 5 call occurrences. Both `getBoundedValue` and `checkBounds` are called twice and `getUnboundedValue` is called once.
- 2 function definitions. `main` and `getBoundedValue` are defined.

Therefore, the Estimated function coupling is $5 - 2 + 1 = 4$.

Negative Estimated Function Coupling

```
int foobar(int a, int b){
    return a+b;
}

int bar(int b){
    return b+2;
}

int foo(int a){
    return a<<2;
}

int main(int x){
    foobar(x,x+2);
    return 0;
}
```

This example shows how you can get a negative estimated function coupling result. In this example, you see:

- 1 function call in `main`.
- 4 defined functions: `foobar`, `bar`, `foo`, and `main`.

Therefore, the estimated function coupling is $1 - 4 + 1 = -2$.

Metric Information

Group: File

Acronym: FCO

HIS Metric: No

See Also

Number of Call Occurrences | Calculate code metrics (-code-metrics)

Higher Estimate of Local Variable Size

Total size of all local variables in function

Description

This metric provides a conservative estimate of the total size of local variables in a function. The metric is the sum of the following sizes in bytes:

- Size of function return value
- Sizes of function parameters
- Sizes of local variables
- Additional padding introduced for memory alignment

Your actual stack usage due to local variables can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. For instance, compilers store the address to which the execution returns following the function call. When computing this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. When computing this metric, Polyspace does not consider this hidden memory usage.
- (C++ only) Destructors and `try-catch` statements can introduce hidden contributions to the metric value.

However, the metric provides a reasonable estimate of the stack usage due to local variables.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric also takes into account `#pragma pack` directives in your code.

Examples

All Variables of Same Type

```
int flag();

int func(int param) {
    int var_1;
    int var_2;
    if (flag()) {
        int var_3;
        int var_4;
    } else {
        int var_5;
    }
}
```

In this example, assuming 4 bytes for `int`, the higher estimate of local variable size is 28. The breakup of the size is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	4	4
Parameter <code>param</code>	4	8
Local variables <code>var_1</code> and <code>var_2</code>	$4+4=8$	16
Local variables defined in the <code>if</code> condition	$(4+4)+4=12$ The size of variables in the first branch is eight bytes. The size in the second branch is four bytes. The sum of the two branches is 12 bytes.	28

No padding is introduced for memory alignment because all the variables involved have the same type.

Variables of Different Types

```
char func(char param) {
    int var_1;
```

```

char var_2;
double var_3;
}

```

In this example, assuming one byte for char, four bytes for int and eight bytes for double and four bytes for alignment, the higher estimate of local variable size is 20. The alignment is usually the word size on your platform. In your Polyspace project, you specify the alignment through your target processor. For more information, see the Alignment column in Target processor type (-target). The breakup of the size is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	1	1
Additional padding introduced before param is stored	0 No memory alignment is required because the next variable param has the same size.	1
Parameter param	1	2
Additional padding introduced before var_1 is stored	2 Memory must be aligned using padding because the next variable var_1 requires four bytes. The storage must start from a memory address at a multiple of four.	4
var_1	4	8
Additional padding introduced before var_2 is stored	0 No memory alignment is required because the next variable var_2 has smaller size.	8
var_2	1	9

Variable	Size (in Bytes)	Running Total
Additional padding introduced before <code>var_3</code> is stored	3 Memory must be aligned using padding because the next variable <code>var_3</code> has eight bytes. The storage must start from a memory address at a multiple of the alignment, four bytes.	12
<code>var_3</code>	8	20

The rules for the amount of padding are:

- If the next variable stored has the same or smaller size, no padding is required.
- If the next variable has a greater size:
 - If the variable size is the same as or less than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of its size.
 - If the variable size is greater than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of the alignment.

C++ Methods and Objects

```
class MySimpleClass {
public:
    MySimpleClass() {};
    MySimpleClass(int) {};
    ~MySimpleClass() {};
};

int main() {
    MySimpleClass c;
    return 0;
}
```

In this example, the estimated local variable sizes are:

- Constructor `MySimpleClass::MySimpleClass()`: Four bytes.

The size comes from the `this` pointer, which is an implicit argument to the constructor. You specify the pointer size using the option `Target processor type (-target)`.

- Constructor `MySimpleClass::MySimpleClass(int)`: Eight bytes.

The size comes from the `this` pointer and the `int` argument.

- Destructor `MySimpleClass::~~MySimpleClass()`: Eight bytes.

The size comes from the `this` pointer and a hidden contribution from an internal variable.

- `main()`: Five bytes.

The size comes from the `int` return value and the size of object `c`. The minimum size of an object is the alignment that you specify using the option `Target processor type (-target)`.

C++ Functions with Object Arguments

```
class MyClass {
public:
    MyClass() {};
    MyClass(int) {};
    ~MyClass() {};
private:
    int i[10];
};

void func1(const MyClass& c) {
}

void func2() {
    func1(4);
}
```

In this example, the estimated local variable size for `func2()` is 40 bytes. When `func2()` calls `func1()`, a temporary object of the class `MyClass` is created. The object has ten `int` variables, each with a size of four bytes.

Metric Information

Group: Function

Acronym: LOCAL_VARS_MAX

HIS Metric: No

See Also

Lower Estimate of Local Variable Size | Calculate code metrics (-code-metrics)

Introduced in R2016b

Language Scope

Language scope

Description

This metric measures the cost of maintaining or changing a function. It is calculated as:

$$(N1 + N2) / (n1 + n2)$$

Here:

- N1 is the number of occurrences of operators.
- N2 is the number of occurrences of operands.
- n1 is the number of distinct operators.
- n2 is the number of distinct operands.

The recommended upper limit for this metric is 4. For lower maintenance cost for a function, try to enforce an upper limit on this metric. For instance, if the same operand occurs many times, to change the operand name, you have to make many substitutions.

To enforce limits on metrics:

- In the Polyspace user interface, see “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

Examples

Language Scope Calculation

```
int f(int i)
{
    if (i == 1)
        return i;
    else
```

```
        return i * g(i-1);  
    }
```

In this example:

- $N1 = 17$.
- $N2 = 9$.
- $n1 = 12$.

The distinct operators are `int, (,), {, if, ==, return, else, *, -, ;, }`.

- $n2 = 4$.

The distinct operands are `f, i, 1` and `g`.

The language scope of `f` is $(17 + 9) / (12 + 4) = 1.8$.

Metric Information

Group: Function

Acronym: VOCE

HIS Metric: Yes

See Also

Calculate code metrics (`-code-metrics`)

Lower Estimate of Local Variable Size

Total size of local variables in function taking nested scopes into account

Description

This metric provides an optimistic estimate of the total size of local variables in a function. The metric is the sum of the following sizes in bytes:

- Size of function return value
- Sizes of function parameters
- Sizes of local variables

Suppose that the function has variable definitions in nested scopes as follows:

```
type func (type param_1, ...) {  
  
    {  
        /* Scope 1 */  
        type var_1, ...;  
    }  
    {  
        /* Scope 2 */  
        type var_2, ...;  
    }  
}
```

The software computes the total variable size in each scope and uses whichever total is greatest. For instance, if a conditional statement has variable definitions, the software computes the total variable size in each branch, and then uses whichever total is greatest. If a nested scope itself has further nested scopes, the same process is repeated for the inner scopes.

A variable defined in a nested scope is not visible outside the scope. Therefore, some compilers reuse stack space for variables defined in separate scopes. This metric provides a more accurate estimate of stack usage for such compilers. Otherwise, use the metric `Higher Estimate of Local Variable Size`. This metric adds the size of all local variables, whether or not they are defined in nested scopes.

- Additional padding introduced for memory alignment

Your actual stack usage due to local variables can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. When computing this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. For instance, compilers store the address to which the execution returns following the function call. When computing this metric, Polyspace does not consider this hidden memory usage.
- (C++ only) Destructors and `try-catch` statements can introduce hidden contributions to the metric value.

However, the metric provides a reasonable estimate of the stack usage due to local variables.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric also takes into account `#pragma pack` directives in your code.

Examples

All Variables of Same Type

```
int flag();

int func(int param) {
    int var_1;
    int var_2;
    if (flag()) {
        int var_3;
        int var_4;
    } else {
        int var_5;
    }
}
```

In this example, assuming four bytes for `int`, the lower estimate of local variable size is 24. The breakup of the metric is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	4	4
Parameter param	4	8
Local variables var_1 and var_2	4+4=8	16
Local variables defined in the if condition	$\max(4+4, 4) = 8$ The size of variables in the first branch is eight bytes. The size in the second branch is four bytes. The maximum of the two branches is eight bytes.	24

No padding is introduced for memory alignment because all the variables involved have the same type.

Variables of Different Types

```
char func(char param) {
    int var_1;
    char var_2;
    double var_3;
}
```

In this example, assuming one byte for char, four bytes for int, eight bytes for double and four bytes for alignment, the lower estimate of local variable size is 20. The alignment is usually the word size on your platform. In your Polyspace project, you specify the alignment through your target processor. For more information, see the Alignment column in Target processor type (-target). The breakup of the size is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	1	1

Variable	Size (in Bytes)	Running Total
Additional padding introduced before <code>param</code> is stored	0 No memory alignment is required because the next variable <code>param</code> has the same size.	1
Parameter <code>param</code>	1	2
Additional padding introduced before <code>var_1</code> is stored	2 Memory must be aligned using padding because the next variable <code>var_1</code> requires four bytes. The storage must start from a memory address at a multiple of four.	4
<code>var_1</code>	4	8
Additional padding introduced before <code>var_2</code> is stored	0 No memory alignment is required because the next variable <code>var_2</code> has smaller size.	8
<code>var_2</code>	1	9
Additional padding introduced before <code>var_3</code> is stored	3 Memory must be aligned using padding because the next variable <code>var_3</code> requires eight bytes. The storage must start from a memory address at a multiple of the alignment, four bytes.	12
<code>var_3</code>	8	20

The rules for the amount of padding are:

- If the next variable stored has the same or smaller size, no padding is required.
- If the next variable has a greater size:
 - If the variable size is the same as or less than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of its size.
 - If the variable size is greater than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of the alignment.

C++ Methods and Objects

```
class MySimpleClass {
public:
    MySimpleClass() {};
    MySimpleClass(int) {};
    ~MySimpleClass() {};
};

int main() {
    MySimpleClass c;
    return 0;
}
```

In this example, the estimated local variable sizes are:

- Constructor `MySimpleClass::MySimpleClass()`: Four bytes.

The size comes from the `this` pointer, which is an implicit argument to the constructor. You specify the pointer size using the option `Target processor type (-target)`.

- Constructor `MySimpleClass::MySimpleClass(int)`: Eight bytes.

The size comes from the `this` pointer and the `int` argument.

- Destructor `MySimpleClass::~MySimpleClass()`: Eight bytes.

The size comes from the `this` pointer and a hidden contribution from an internal variable.

- `main()`: Five bytes.

The size comes from the `int` return value and the size of object `c`. The minimum size of an object is the alignment that you specify using the option `Target processor type (-target)`.

C++ Functions with Object Arguments

```
class MyClass {
public:
    MyClass() {};
    MyClass(int) {};
    ~MyClass() {};
private:
    int i[10];
};

void func1(const MyClass& c) {
}

void func2() {
    func1(4);
}
```

In this example, the estimated local variable size for `func2()` is 40 bytes. When `func2()` calls `func1()`, a temporary object of the class `MyClass` is created. The object has ten `int` variables, each with a size of four bytes.

Metric Information

Group: Function

Acronym: LOCAL_VARS_MIN

HIS Metric: No

See Also

Higher Estimate of Local Variable Size | Calculate code metrics (`-code-metrics`)

Introduced in R2016b

Maximum Stack Usage

Total size of local variables in function plus maximum stack usage from callees

Description

This metric provides a conservative estimate of the stack usage by a function. The metric is the sum of these sizes in bytes:

- Higher Estimate of Local Variable Size
- Maximum value from the stack usages of the function callees. The computation uses the maximum stack usage of each callee.

For instance, in this example, the maximum stack usage of `func` is the same as the maximum stack usage of `func1` or `func2`, *whichever is greater*.

```
void func(void) {  
    func1();  
    func2();  
}
```

If the function calls are in different branches of a conditional statement, this metric considers the branch with the greatest stack usage.

The analysis does the stack size estimation later on when it has resolved which function calls actually occur. For instance, if a function call occurs in unreachable code, the stack size does not take the call into account. The analysis can also take into account calls through function pointers.

Your actual stack usage can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. When estimating this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. For instance, compilers store the address to which the execution returns following the function call. When estimating this metric, Polyspace does not consider this hidden memory usage.

However, the metric provides a reasonable estimate of the stack usage.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric takes into account `#pragma pack` directives in your code.

Examples

Function with One Callee

```
double func(int);
double func2(int);

double func(int status) {
    double res = func2(status);
    return res;
}

double func2(int status) {
    double res;
    if(status == 0) {
        int temp;
        res = 0.0;
    }
    else {
        double temp;
        res = 1.0;
    }
    return res;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func2`: 32 bytes

This value includes the sizes of its parameter (4 bytes), local variable `res` (8 bytes), local variable `temp` counted twice (4+8=12 bytes), and return value (8 bytes).

The metric does not take into account that the first `temp` is no longer live when the second `temp` is defined.

- `func`: 52 bytes

This value includes the sizes of its parameter, local variable `res`, and return value, a total of 20 bytes. This value includes the 32 bytes of maximum stack usage by its callee, `func2`.

Function with Multiple Callees

```
void func1(int);
void func2(void);

void func(int status) {
    func1(status);
    func2();
}

void func1(int status) {
    if(status == 0) {
        int val;
    }
    else {
        double val2;
    }
}

void func2(void) {
    double val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func1`: 16 bytes

This value includes the sizes of its parameter (4 bytes) and local variable `temp` counted twice (4+8=12 bytes).

- `func2`: 8 bytes
- `func`: 20 bytes

This value includes the sizes of its parameter (4 bytes) and the maximum of stack usages of `func1` and `func2` (16 bytes).

Function with Multiple Callees in Different Branches

```
void func1(void);
void func2(void);

void func(int status) {
    if(status==0)
        func1();
    else
        func2();
}

void func1(void) {
    double val;
}

void func2(void) {
    int val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func1`: 8 bytes
- `func2`: 4 bytes
- `func`: 12 bytes

This value includes the sizes of its parameter (4 bytes) and the maximum stack usage from the two branches (8 bytes).

Metric Information

Group: Function

Acronym: `MAX_STACK`

HIS Metric: No

See Also

Minimum Stack Usage | Program Maximum Stack Usage | Higher Estimate of Local Variable Size | Calculate code metrics (`-code-metrics`)

Topics

“Determination of Program Stack Usage” on page 4-35

Introduced in R2017b

Minimum Stack Usage

Total size of local variables in function taking nested scopes into account plus maximum stack usage from callees

Description

This metric provides an optimistic estimate of the stack usage by a function. Unlike the metric `Maximum Stack Usage`, this metric takes nested scopes into account. For instance, if variables are defined in two mutually exclusive branches of a conditional statement, the metric considers that the stack space allocated to the variables in one branch can be reused in the other branch.

The metric is the sum of these sizes in bytes:

- Lower Estimate of Local Variable Size.
- Maximum value from the stack usages of the function callees. The computation uses the minimum stack usage of each callee.

For instance, in this example, the minimum stack usage of `func` is the same as the minimum stack usage of `func1` or `func2`, *whichever is greater*.

```
void func(void) {  
    func1();  
    func2();  
}
```

If the function calls are in different branches of a conditional statement, this metric considers the branch with the least stack usage.

The analysis does the stack size estimation later on when it has resolved which function calls actually occur. For instance, if a function call occurs in unreachable code, the stack size does not take the call into account. The analysis can also take into account calls through function pointers.

Your actual stack usage can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.

- Your compiler performs variable liveness analysis to enable certain memory optimizations. When estimating this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. For instance, compilers store the address to which the execution returns following the function call. When estimating this metric, Polyspace does not consider this hidden memory usage.

However, the metric provides a reasonable estimate of the stack usage.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric takes into account `#pragma pack` directives in your code.

Examples

Function with One Callee

```
double func2(int);

double func(int status) {
    double res = func2(status);
    return res;
}

double func2(int status) {
    double res;
    if(status == 0) {
        int temp;
        res = 0.0;
    }
    else {
        double temp;
        res = 1.0;
    }
    return res;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func2`: 28 bytes

This value includes the sizes of its parameter (4 bytes), local variable `res` (8 bytes), one of the two local variables `temp` (8 bytes), and return value (8 bytes).

The metric takes into account that the first `temp` is no longer live when the second `temp` is defined. It uses the variable `temp` with data type `double` because its size is greater.

- `func`: 48 bytes

This value includes the sizes of its parameter, local variable `res`, and return value, a total of 20 bytes. This value includes the 28 bytes of minimum stack usage by its callee, `func2`.

Function with Multiple Callees

```
void func1(int);
void func2(void);

void func(int status) {
    func1(status);
    func2();
}

void func1(int status) {
    if(status == 0) {
        int val;
    }
    else {
        double val2;
    }
}

void func2(void) {
    double val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- func1: 12 bytes

This value includes the sizes of its parameter (4 bytes) and one of the two local variables `temp` (8 bytes). The metric takes into account that the first `temp` is no longer live when the second `temp` is defined.

- func2: 8 bytes
- func: 16 bytes

This value includes the sizes of its parameter (4 bytes) and the maximum of stack usages of `func1` and `func2` (12 bytes).

Function with Multiple Callees in Different Branches

```
void func1(void);
void func2(void);
```

```
void func(int status) {
    if(status==0)
        func1();
    else
        func2();
}
```

```
void func1(void) {
    double val;
}
```

```
void func2(void) {
    int val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- func1: 8 bytes
- func2: 4 bytes
- func: 8 bytes

This value includes the sizes of its parameter (4 bytes) and the minimum stack usage from the two branches (4 bytes).

Metric Information

Group: Function

Acronym: MIN_STACK

HIS Metric: No

See Also

Program Minimum Stack Usage | Lower Estimate of Local Variable Size |
Maximum Stack Usage | Calculate code metrics (-code-metrics)

Topics

“Determination of Program Stack Usage” on page 4-35

Introduced in R2017b

Number of Call Levels

Maximum depth of nesting of control flow structures

Description

This metric specifies the maximum nesting depth of control flow statements such as `if`, `switch`, `for`, or `while` in a function. A function without control-flow statements has a call level 1.

The recommended upper limit for this metric is 4. For better readability of your code, try to enforce an upper limit for this metric.

To enforce limits on metrics:

- In the Polyspace user interface, see “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

Examples

Function with Nested `if` Statements

```
int foo(int x,int y)
{
    int flag = 0;
    if (x <= 0)
        /* Call level 1*/
        flag = 1;
    else
    {
        if (x <= y )
            /* Call level 2*/
            flag = 1;
        else
            flag = -1;
    }
}
```

```
    return flag;
}
```

In this example, the number of call levels of `foo` is 2.

Function with Nesting of Different Control-Flow Statements

```
int foo(int x,int y, int bound)
{
    int count = 0;
    if (x <= y)
        /* Call level 1*/
        count = 1;
    else
        while(x>y) {
            /* Call level 2*/
            x--;
            if(count< bound) {
                /* Call level 3*/
                count++;
            }
        }
    return count;
}
```

In this example, the number of call levels of `foo` is 3.

Metric Information

Group: Function

Acronym: LEVEL

HIS Metric: Yes

See Also

Calculate code metrics (`-code-metrics`)

Number of Call Occurrences

Number of calls in function body

Description

This metric specifies the number of function calls in the body of a function.

Calls through a function pointer are not counted. Calls in unreachable code and calls to standard library functions are counted. `assert` is considered as a macro and not a function, so it is not counted.

Examples

Same Function Called Multiple Times

```
int func1(void);
int func2(void);

int foo() {
    return (func1() + func1()*func1() + 2*func2());
}
```

In this example, the number of call occurrences in `foo` is 4.

Function Called in a Loop

```
#include<stdio.h>

void fillArraySize10(int *arr) {
    for(int i=0; i<10; i++)
        arr[i]=getVal();
}

int getVal(void) {
    int val;
    printf("Enter a value:");
```

```
    scanf("%d", &val);
    return val;
}
```

In this example, the number of call occurrences in `fillArraySize10` is 1.

Recursive Function

```
#include <stdio.h>

void main() {
    int count;
    printf("How many numbers ?");
    scanf("%d",&count);
    fibonacci(count);
}

int fibonacci(int num)
{
    if ( num == 0 )
        return 0;
    else if ( num == 1 )
        return 1;
    else
        return ( fibonacci(num-1) + fibonacci(num-2) );
}
```

In this example, the number of call occurrences in `fibonacci` is 2.

Metric Information

Group: Function

Acronym: NCALLS

HIS Metric: No

See Also

Number of Called Functions | Calculate code metrics (-code-metrics)

Number of Called Functions

Number of callees of a function

Description

This metric specifies the number of callees of a function.

Calls through a function pointer are not counted. Calls in unreachable code and calls to standard library functions are counted. `assert` is considered as a macro and not a function, so it is not counted. For C++ templates, the first instantiation of the template is used to calculate this metric.

The recommended upper limit for this metric is 7. For more self-contained code, try to enforce an upper limit on this metric.

To enforce limits on metrics:

- In the Polyspace user interface, see “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

Examples

Same Function Called Multiple Times

```
int func1(void);
int func2(void);

int foo() {
    return (func1() + func1()*func1() + 2*func2());
}
```

In this example, the number of called functions in `foo` is 2. The called functions are `func1` and `func2`.

Recursive Function

```
#include <stdio.h>

void main() {
    int count;
    printf("How many numbers ?");
    scanf("%d",&count);
    fibonacci(count);
}

int fibonacci(int num)
{
    if ( num == 0 )
        return 0;
    else if ( num == 1 )
        return 1;
    else
        return ( fibonacci(num-1) + fibonacci(num-2) );
}
```

In this example, the number of called functions in `fibonacci` is 1. The called function is `fibonacci` itself.

Metric Information

Group: Function

Acronym: CALLS

HIS Metric: Yes

See Also

Number of Call Occurrences | Number of Calling Functions | Calculate code metrics (-code-metrics)

Number of Calling Functions

Number of distinct callers of a function

Description

This metric measures the number of distinct callers of a function.

Calls through a function pointer are not counted. Calls in unreachable code are counted. Even if a caller calls a function more than once, it is counted only once when this metric is calculated. For C++ templates, the first instantiation of the template is used to calculate this metric.

The recommended upper limit for this metric is 5. For more self-contained code, try to enforce an upper limit on this metric.

To enforce limits on metrics:

- In the Polyspace user interface, see “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

Examples

Same Function Calling a Function Multiple Times

```
#include <stdio.h>

int getVal() {
    int myVal;
    printf("Enter a value:");
    scanf("%d", &myVal);
    return myVal;
}

int func() {
    int val=getVal();
```

```
    if(val<0)
        return 0;
    else
        return val;
}

int func2() {
    int val=getVal();
    while(val<0)
        val=getVal();
    return val;
}
```

In this example, the number of calling functions for `getVal` is 2. The calling functions are `func` and `func2`.

Recursive Function

```
#include <stdio.h>

void main() {
    int count;
    printf("How many numbers ?");
    scanf("%d",&count);
    fibonacci(count);
}

int fibonacci(int num)
{
    if ( num == 0 )
        return 0;
    else if ( num == 1 )
        return 1;
    else
        return ( fibonacci(num-1) + fibonacci(num-2) );
}
```

In this example, the number of calling functions for `fibonacci` is 2. The calling functions are `main` and `fibonacci` itself.

Metric Information

Group: Function

Acronym: CALLING

HIS Metric: Yes

See Also

Number of Called Functions | Calculate code metrics (-code-metrics)

Number of Direct Recursions

Number of instances of a function calling itself directly

Description

This metric specifies the number of direct recursions in your project.

A direct recursion is a recursion where a function calls itself in its own body. If indirect recursions do not occur, the number of direct recursions is equal to the number of recursive functions.

The recommended upper limit for this metric is 0. To avoid the possibility of exceeding available stack space, do not use recursions in your code. To detect use of recursions, check for violations of MISRA C:2012 Rule 17.2.

To enforce limits on metrics, see “Compare Metrics Against Software Quality Objectives”.

Examples

Direct Recursion

```
int getVal(void);

void main() {
    int count = getVal(), total;
    assert(count > 0 && count <100);
    total = sum(count);
}

int sum(int val) {
    if(val<0)
        return 0;
    else
        return (val + sum(val-1));
}
```

In this example, the number of direct recursions is 1.

Metric Information

Group: Project

Acronym: AP_CG_DIRECT_CYCLE

HIS Metric: Yes

See Also

MISRA C:2012 Rule 17.2 | Calculate code metrics (-code-metrics)

Number of Executable Lines

Number of executable lines in function body

Description

This metric measures the number of executable lines in a function body. When calculating the value of this metric, Polyspace excludes declarations without static initializers, comments, blank lines, braces or preprocessing directives.

If the function body contains a `#include` directive, the included file source code is also calculated as part of this metric.

This metric is not calculated for C++ templates.

Examples

Function with Declarations, Braces and Comments

```
void func(int);

int getSign(int arg) {
    int sign;
    if(arg<0) {
        sign=-1;
        func(-arg);
        /* func takes positive arguments */
    }
    else if(arg==0)
        sign=0;
    else {
        sign=1;
        func(arg);
    }
    return sign;
}
```

In this example, the number of executable lines of `getSign` is 9. The calculation excludes:

- The declaration `int sign;`
- The comment `/* ... */`.
- The two lines with braces only.

Metric Information

Group: Function

Acronym: FXLN

HIS Metric: No

See Also

Number of Lines Within Body | Number of Instructions | Calculate code metrics (-code-metrics)

Number of Files

Number of source files

Description

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

Metric Information

Group: Project

Acronym: FILES

HIS Metric: No

See Also

Number of Header Files | Calculate code metrics (-code-metrics)

Number of Function Parameters

Number of function arguments

Description

This metric measures the number of function arguments.

If ellipsis is used to denote variable number of arguments, when calculating this metric, the ellipsis is not counted.

The recommended upper limit for this metric is 5. For less dependency between functions and fewer side effects, try to enforce an upper limit on this metric.

To enforce limits on metrics:

- In the Polyspace user interface, see “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

Examples

Function with Fixed Arguments

```
int initializeArray(int* arr, int size) {  
}
```

In this example, `initializeArray` has two parameters.

Function with Type Definition in Arguments

```
int getValueInLoc(struct {int* arr; int size;}myArray, int loc) {  
}
```

In this example, `getValueInLoc` has two parameters.

Function with Variable Arguments

```
double average ( int num, ... )
{
    va_list arg;
    double sum = 0;

    va_start ( arg, num );

    for ( int x = 0; x < num; x++ )
    {
        sum += va_arg ( arg, double );
    }
    va_end ( arg);

    return sum / num;
}
```

In this example, `average` has one parameter. The ellipsis denoting variable number of arguments is not counted.

Metric Information

Group: Function

Acronym: PARAM

HIS Metric: Yes

See Also

Calculate code metrics (`-code-metrics`)

Number of Goto Statements

Number of `goto` statements

Description

This metric measures the number of `goto` statements in a function.

`break` and `continue` statements are not counted.

The recommended upper limit on this metric is 0. For better readability of your code, avoid `goto` statements in your code. To detect use of `goto` statements, check for violations of MISRA C:2012 Rule 15.1.

To enforce limits on metrics:

- In the Polyspace user interface, see “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

Examples

Function with `goto` Statements

```
#define SIZE 10
int initialize(int **arr, int loc);
void printString(char *);
void printErrorMessage(void);
void printExecutionMessage(void);

int main()
{
    int *arrayOfStrings[SIZE], len[SIZE], i;
    for ( i = 0; i < SIZE; i++ )
    {
        len[i] = initialize(arrayOfStrings, i);
    }
}
```

```
for ( i = 0; i < SIZE; i++ )
{
    if(len[i] == 0)
        goto emptyString;
    else
        goto nonEmptyString;
loop: printExecutionMessage();
}

emptyString:
    printErrorMessage();
    goto loop;
nonEmptyString:
    printString(arrayOfStrings[i]);
    goto loop;
}
```

In this example, the function main has 4 goto statements.

Metric Information

Group: Function

Acronym: GOTO

HIS Metric: Yes

See Also

Calculate code metrics (-code-metrics)

Number of Header Files

Number of included header files

Description

This metric measures the number of header files in the project. Both directly and indirectly included header files are counted.

The metric gives a slightly higher number than the actual number of header files that you use because Polyspace® internal header files and header files included by those files are also counted. For the same reason, the metric can vary slightly even if you do not explicitly include new header files or remove inclusion of header files from your code. For instance, the number of Polyspace® internal header files can vary if you change your analysis options.

Metric Information

Group: Project

Acronym: INCLUDES

HIS Metric: No

See Also

Number of Files | Calculate code metrics (-code-metrics)

Number of Instructions

Number of instructions per function

Description

This metric measures the number of instructions in a function body.

The recommended upper limit for this metric is 50. For more modular code, try to enforce an upper limit for this metric.

To enforce limits on metrics:

- In the Polyspace user interface, see “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

Computation Details

The metric is calculated using the following rules:

- A simple statement ending with a `;` is one instruction.

If the statement is empty, it does not count as an instruction.

- A variable declaration counts as one instruction only if the variable is also initialized.
- Control flow statements such as `if`, `for`, `break`, `goto`, `return`, `switch`, `while`, `do-while` count as one instruction.
- The following do not count as instructions by themselves:
 - Beginning of a block of code

For instance, the following counts as one instruction:

```
{
    var = 1;
}
```

- Labels

For instance, the following counts as two instructions. The case labels do not count as instructions.

```
switch (1) { // Instruction 1: switch
    case 0:
    case 1:
    case 2:
    default:
        break; // Instruction 2: break
}
```

Examples

Calculation of Number of Instructions

```
int func(int* arr, int size) {
    int i, countPos=0, countNeg=0, countZero = 0;
    for(i=0; i<size; i++) {
        if(arr[i] >0)
            countPos++;
        else if(arr[i] ==0)
            countZero++;
        else
            countNeg++;
    }
}
```

In this example, the number of instructions in `func` is 9. The instructions are:

- 1 `countPos=0`
- 2 `countNeg=0`
- 3 `countZero=0`
- 4 `for(i=0;i<size;i++) { ... }`
- 5 `if(arr[i] >=0)`
- 6 `countPos++`
- 7 `else if(arr[i]==0)`

The ending `else` is counted as part of the `if-else` instruction.

8 `countZero++`

9 `countNeg++`

Note This metric is different from the number of executable lines. For instance:

- `for (i=0;i<size;i++)` has 1 instruction and 1 executable line.
- The following code has 1 instruction but 3 executable lines.

```
for (i=0;
     i<size;
     i++)
```

Metric Information

Group: Function

Acronym: *STMT*

HIS Metric: Yes

See Also

`Calculate code metrics (-code-metrics)`

Number of Lines

Total number of lines in a file

Description

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

This metric is calculated for source files and header files in the same folders as source files. If you want:

- The metric reported for other header files, change the default value of the option `Generate results for sources` and `(-generate-results-for)`.
- The metric not reported for header files at all, change the value of the option `Do not generate results for` `(-do-not-generate-results-for)` to `all-headers`.

Metric Information

Group: File

Acronym: `TOTAL_LINES`

HIS Metric: No

See Also

Number of Lines Without Comment | Calculate code metrics `(-code-metrics)`

Number of Lines Within Body

Number of lines in function body

Description

This metric calculates the number of lines in function body. When calculating the value of this metric, Polyspace includes declarations, comments, blank lines, braces and preprocessing directives.

If the function body contains a `#include` directive, the included file source code is also calculated as part of this metric.

This metric is not calculated for C++ templates.

Examples

Function with Declarations, Braces and Comments

```
void func(int);

int getSign(int arg) {
    int sign;
    if(arg<0) {
        sign=-1;
        func(-arg);
        /* func takes positive arguments */
    }
    else if(arg==0)
        sign=0;
    else {
        sign=1;
        func(arg);
    }
    return sign;
}
```

In this example, the number of executable lines of `getSign` is 13. The calculation includes:

- The declaration `int sign;`
- The comment `/* ... */`.
- The two lines with braces only.

Metric Information

Group: Function

Acronym: FLIN

HIS Metric: No

See Also

Number of Executable Lines | Calculate code metrics (`-code-metrics`)

Number of Lines Without Comment

Number of lines of code excluding comments

Description

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

This metric is calculated for source files and header files in the same folders as source files. If you want:

- The metric reported for other header files, change the default value of the option `Generate results for sources` and `(-generate-results-for)`.
- The metric not reported for header files at all, change the value of the option `Do not generate results for` `(-do-not-generate-results-for)` to `all-headers`.

Metric Information

Group: File

Acronym: `LINES_WITHOUT_CMT`

HIS Metric: No

See Also

Number of Lines | Calculate code metrics `(-code-metrics)`

Number of Local Non-Static Variables

Total number of local variables in function

Description

This metric provides the number of local variables in a function.

The metric excludes static variables. To find number of static variables, use the metric Number of Local Static Variables.

Examples

Non-Structured Variables

```
int flag();

int func(int param) {
    int var_1;
    int var_2;
    if (flag()) {
        int var_3;
        int var_4;
    } else {
        int var_5;
    }
}
```

In this example, the number of local non-static variables in `func` is 5. The number does not include the function arguments and return value.

Arrays and Structured Variables

```
typedef struct myStruct{
    char  arr1[50];
    char  arr2[50];
    int   val;
```

```
} myStruct;

void func(void) {
    myStruct var;
    char localArr[50];
}
```

In this example, the number of local non-static variables in `func` is 2: the structured variable `var` and the array `localArr`.

Variables in Class Methods

```
class Rectangle {
    int width, height;
public:
    void set (int,int);
    int area (void);
} rect;

int Rectangle::area (void) {
    int temp;
    temp = width * height;
    return(temp);
}
```

In this example, the number of local non-static variables in `Rectangle::area` is 1: the variable `temp`.

Metric Information

Group: Function

Acronym: LOCAL_VARS

HIS Metric: No

See Also

Number of Local Static Variables | Higher Estimate of Local Variable Size | Lower Estimate of Local Variable Size | Calculate code metrics (-code-metrics)

Introduced in R2017a

Number of Local Static Variables

Total number of local static variables in function

Description

This metric provides the number of local static variables in a function.

Examples

Number of Static Variables

```
void func(void) {  
    static int var_1 = 0;  
    int var_2;  
}
```

In this example, the number of static variables in `func` is 1. For examples of different types of variables, see [Number of Local Non-Static Variables](#).

Metric Information

Group: Function

Acronym: LOCAL_STATIC_VARS

HIS Metric: No

See Also

[Higher Estimate of Local Variable Size | Number of Local Non-Static Variables | Lower Estimate of Local Variable Size | Calculate code metrics \(-code-metrics\)](#)

Introduced in R2017a

Number of Paths

Estimated static path count

Description

This metric measures the number of paths in a function.

If `goto` statements are present in your code, Polyspace cannot calculate the number of paths. The software displays a metric value of -1.

The recommended upper limit for this metric is 80. If the number of paths is high, the code is difficult to read and can cause more orange checks. Try to limit the value of this metric.

To enforce limits on metrics:

- In the Polyspace user interface, see “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

Computation Details

The number of paths is calculated according to these rules:

- If the statements in a function do not break the control flow, the number of paths is one.

Even an empty statement such as `;` or empty block such as `{ }` counts as one path.

- The number of paths for a control flow statement is calculated as follows:
 - `if-else if-else`: The number of paths is the sum of paths calculated in the `if` block, each `else if` block, and the concluding `else` block. When the concluding `else` block is omitted, the path count is increased by 1.

For instance, the statement `if(..) {} else if(..) {} else {}` counts as three paths. The statement `if() {}` counts as two paths, one for the `if` block and one for the omitted `else` block.

- **switch-case:** Every case with break statement adds one to the path count. The default statement counts as one path, even if it is omitted.

For instance, the statement `switch (var) { case 1: .. break; case 2: .. break; default: .. }` counts as three paths.

- **for, while, and do-while:** The number of paths is equal to the number of paths in the loop body + 1.

For instance, the statement `while(0) {;}` counts as two paths.

- If more than one control flow statement are present in a sequence, the number of paths is the product of the path count for each control flow statement.

For instance, if a function has three `for` loops and two `if-else` statements, the number of paths is $2 \times 2 \times 2 \times 2 \times 2 = 32$.

If many control flow statements are present in a function, the number of paths can be large. Nested control flow statements reduce the number of paths at the cost of increasing the depth of nesting. For an example, see “Function with Nested Control Flow Statements” on page 7-68.

Examples

Function with One Path

```
void func(int ch) {
    switch (ch)
    {
        case 1:
        case 2:
        case 3:
        case 4:
        default:
    }
}
```

In this example, `func` has one path.

Function with Control Flow Statement Causing Multiple Paths

```
void func(int ch) {
    switch (ch)
    {
    case 1:
        break;
    case 2:
        break;
    case 3:
        break;
    case 4:
        break;
    default:
    }
}
```

In this example, `func` has five paths. Apart from the path that goes through the cases and default, each `break` causes the creation of a new path.

Function with Nested Control Flow Statements

```
void func()
{
    int i = 0, j = 0, k = 0;
    for (i=0; i<10; i++)
    {
        for (j=0; j<10; j++)
        {
            for (k=0; k<10; k++)
            {
                if (i < 2)
                    ;
                else
                {
                    if (i > 5)
                        ;
                    else
                        ;
                }
            }
        }
    }
}
```

In this example, `func` has six paths. The number is calculated as follows:

- The innermost `if-else` block counts as two paths.
- The outer `if-else` block counts as three paths, one path for the `if` block and the previous two paths for the `else` block.
- The innermost `for` loop counts as four paths, one path for the loop and the previous three paths for the `if-else` blocks.
- The next two outer loops add one path each.

Therefore, the number of paths in `func` is six.

Metric Information

Group: Function

Acronym: PATH

HIS Metric: Yes

See Also

`Calculate code metrics (-code-metrics)`

Number of Protected Shared Variables

Number of protected shared variables

Description

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

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

Note This metric is available only in the Polyspace Metrics web interface. In the Polyspace user interface, each protected shared variable is reported separately. For more information, see [Shared protected global variable](#).

Examples

Shared Variables Protected Through Temporal Exclusion

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        reset();
        inc();
    }
}
```

```

        inc();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}

void main() {
}

```

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

Option	Value
Entry points	task interrupt_handler
Temporally exclusive tasks	task interrupt_handler

The variable is shared between `task` and `interrupt_handler`. However, because `task` and `interrupt_handler` are temporally exclusive, operations on the variable cannot interrupt each other.

Shared Variables Protected Through Critical Sections

```

#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

```

```

}

void take_semaphore(void);
void give_semaphore(void);

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        take_semaphore();
        reset();
        inc();
        inc();
        give_semaphore();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        take_semaphore();
        interrupt();
        give_semaphore();
    }
}

void main() {
}

```

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

Option	Value	
Entry points	task	
	interrupt_handler	
Critical section details	Starting routine	Ending routine
	take_semaphore	give_semaphore

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

Metric Information

Group: Project

Acronym: PSHV

HIS Metric: No

See Also

Entry points (`-entry-points`) | Critical section details (`-critical-section-begin -critical-section-end`) | Temporally exclusive tasks (`-temporal-exclusions-file`) | Calculate code metrics (`-code-metrics`)

Number of Recursions

Number of call graph cycles over one or more functions

Description

This metric specifies the number of recursions in your project. Even if more than one function is involved in one recursive cycle, the number of recursions is counted as one.

Calls through a function pointer are not considered.

The recommended upper limit for this metric is 0. To avoid the possibility of exceeding available stack space, do not use recursions in your code. To detect use of recursions, check for violations of MISRA C:2012 Rule 17.2.

To enforce limits on metrics:

- In the Polyspace user interface, see “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

Examples

Direct Recursion

```
int getVal(void);

void main() {
    int count = getVal(), total;
    assert(count > 0 && count <100);
    total = sum(count);
}

int sum(int val) {
    if(val<0)
        return 0;
    else
```

```
        return (val + sum(val-1));  
    }  
}
```

In this example, the number of recursions is 1.

A direct recursion is a recursion where a function calls itself in its own body. For direct recursions, the number of recursions is equal to the number of recursive functions.

Indirect Recursion with One Call Graph Cycle

```
volatile int signal;  
  
void operation1() {  
    int stop = signal%2;  
    if(!stop)  
        operation2();  
}  
  
void operation2() {  
    operation1();  
}  
  
void main() {  
    operation1();  
}
```

In this example, the number of recursions is 1. Although two functions `operation1` and `operation2` indirectly call themselves, they are involved in the same call graph cycle `operation1 → operation2 → operation1`.

An indirect function is a recursion where a function calls itself through other functions. For indirect recursions, the number of recursions can be different from the number of recursive functions.

Indirect Recursion with Two Call Graph Cycles

```
volatile int signal;  
  
void operation1() {  
    int stop = signal%3;  
    if(stop==1)  
        operation2();  
}
```

```
        else if(stop==2)
            operation3();
    }

void operation2() {
    operation1();
}

void operation3() {
    operation3();
}

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

In this example, the number of recursions is 2.

There are two call graph cycles:

- operation1 → operation2 → operation1
- operation1 → operation3 → operation1

Same Function Called in Direct and Indirect Recursion

```
volatile int signal;

void operation1() {
    int stop = signal%3;
    if(stop==1)
        operation1();
    else if(stop==2)
        operation2();
}

void operation2() {
    operation1();
}

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

In this example, the number of call graph cycles is 1.

If the same function calls itself both directly and indirectly, the two cycles are counted as 1.

Metric Information

Group: Project

Acronym: AP_CG_CYCLE

HIS Metric: Yes

See Also

MISRA C:2012 Rule 17.2 | Calculate code metrics (-code-metrics)

Number of Return Statements

Number of `return` statements in a function

Description

This metric measures the number of `return` statements in a function.

The recommended upper limit for this metric is 1. If one return statement is present, when reading the code, you can easily identify what the function returns.

To enforce limits on metrics:

- In the Polyspace user interface, see “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

Examples

Function with Return Points

```
int getSign (int arg) {  
    if (arg < 0)  
        return -1;  
    else if (arg > 0)  
        return 1;  
    return 0;  
}
```

In this example, `getSign` has 3 `return` statements.

Metric Information

Group: Function

Acronym: RETURN

HIS Metric: Yes

See Also

Calculate code metrics (-code-metrics)

Topics

“Review Code Metrics”

“Compare Metrics Against Software Quality Objectives”

Number of Unprotected Shared Variables

Number of unprotected shared variables

Description

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

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

Note This metric is available only in the Polyspace Metrics web interface. In the Polyspace user interface, each unprotected shared variable is reported separately. For more information, see [Shared unprotected global variable](#).

Examples

Unprotected Shared Variables

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        reset();
        inc();
    }
}
```



```
        inc();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}

void main() {
}
```

In this example, `shared_var` is an unprotected shared variable if you specify `task` and `interrupt_handler` as entry points and do not specify protection mechanisms.

The operation `shared_var = INT_MAX` can interrupt the other operations on `shared_var` and cause unpredictable behavior.

Metric Information

Group: Project

Acronym: UNPSHV

HIS Metric: No

See Also

Calculate code metrics (`-code-metrics`)

Program Maximum Stack Usage

Maximum stack usage in the analyzed program

Description

This metric shows the maximum stack usage from your program.

The metric shows the maximum stack usage for the function with the highest stack usage. If you provide a complete application, the function with the highest stack usage is typically the `main` function because the `main` function is at the top of the call hierarchy. For a description of maximum stack usage for a function, see the metric `Maximum Stack Usage`.

Metric Information

Group: Project

Acronym: `PROG_MAX_STACK`

HIS Metric: No

See Also

[Higher Estimate of Local Variable Size](#) | [Maximum Stack Usage](#) | [Program Minimum Stack Usage](#) | `Calculate code metrics (-code-metrics)`

Topics

“Determination of Program Stack Usage” on page 4-35

Introduced in R2017b

Program Minimum Stack Usage

Maximum stack usage in the analyzed program taking nested scopes into account

Description

This metric shows the maximum stack usage from your program, taking nested scopes into account.

The metric shows the minimum stack usage for the function with the highest stack usage. If you provide a complete application, the function with the highest stack usage is typically the `main` function because the `main` function is at the top of the call hierarchy. For a description of minimum stack usage for a function, see the metric `Minimum Stack Usage`.

Considering nested scopes is useful for compilers that reuse stack space for variables defined in nested scopes. For instance, in this code, the space for `var_1` is reused for `var_2`.

```
type func (type param_1, ...) {  
  
    {  
        /* Scope 1 */  
        type var_1, ...;  
    }  
    {  
        /* Scope 2 */  
        type var_2, ...;  
    }  
}
```

Metric Information

Group: Project

Acronym: `PROG_MIN_STACK`

HIS Metric: No

See Also

Lower Estimate of Local Variable Size | Minimum Stack Usage | Program
Maximum Stack Usage | Calculate code metrics (-code-metrics)

Topics

“Determination of Program Stack Usage” on page 4-35

Introduced in R2017b

Custom Coding Rules

Group 1: Files

Number	Rule Applied	Message generated if rule is violated	Other details
1.1	All source file names must follow the specified pattern.	The source file name “file_name” does not match the specified pattern.	Only the base name is checked. A source file is a file that is not included.
1.2	All source folder names must follow the specified pattern.	The source dir name “dir_name” does not match the specified pattern.	Only the folder name is checked. A source file is a file that is not included.
1.3	All include file names must follow the specified pattern.	The include file name “file_name” does not match the specified pattern.	Only the base name is checked. An include file is a file that is included.
1.4	All include folder names must follow the specified pattern.	The include dir name “dir_name” does not match the specified pattern.	Only the folder name is checked. An include file is a file that is included.

Group 2: Preprocessing

Number	Rule Applied	Message generated if rule is violated	Other details
2.1	All macros must follow the specified pattern.	The macro "macro_name" does not match the specified pattern.	Macro names are checked before preprocessing.
2.2	All macro parameters must follow the specified pattern.	The macro parameter "param_name" does not match the specified pattern.	Macro parameters are checked before preprocessing.

Group 3: Type definitions

Number	Rule Applied	Message generated if rule is violated	Other details
3.1	All integer types must follow the specified pattern.	The integer type “type_name” does not match the specified pattern.	Applies to integer types specified by typedef statements. Does not apply to enumeration types. For example: <pre>typedef signed int int32_t;</pre>
3.2	All float types must follow the specified pattern.	The float type “type_name” does not match the specified pattern.	Applies to float types specified by typedef statements. For example: <pre>typedef float f32_t;</pre>
3.3	All pointer types must follow the specified pattern.	The pointer type “type_name” does not match the specified pattern.	Applies to pointer types specified by typedef statements. For example: <pre>typedef int* p_int;</pre>
3.4	All array types must follow the specified pattern.	The array type “type_name” does not match the specified pattern.	Applies to array types specified by typedef statements. For example: <pre>typedef int[3] a_int_3;</pre>
3.5	All function pointer types must follow the specified pattern.	The function pointer type “type_name” does not match the specified pattern.	Applies to function pointer types specified by typedef statements. For example: <pre>typedef void (*pf_callback)(int);</pre>

Group 4: Structures

Number	Rule Applied	Message generated if rule is violated	Other details
4.1	All <code>struct</code> tags must follow the specified pattern.	The <code>struct</code> tag “ <code>tag_name</code> ” does not match the specified pattern.	
4.2	All <code>struct</code> types must follow the specified pattern.	The <code>struct</code> type “ <code>type_name</code> ” does not match the specified pattern.	This is the <code>typedef</code> name.
4.3	All <code>struct</code> fields must follow the specified pattern.	The <code>struct</code> field “ <code>field_name</code> ” does not match the specified pattern.	
4.4	All <code>struct</code> bit fields must follow the specified pattern.	The <code>struct</code> bit field “ <code>field_name</code> ” does not match the specified pattern.	

Group 5: Classes (C++)

Number	Rule Applied	Message generated if rule is violated	Other details
5.1	All class names must follow the specified pattern.	The class tag “tag_name” does not match the specified pattern.	
5.2	All class types must follow the specified pattern.	The class type “type_name” does not match the specified pattern.	This is the typedef name.
5.3	All data members must follow the specified pattern.	The data member “member_name” does not match the specified pattern.	
5.4	All function members must follow the specified pattern.	The function member “member_name” does not match the specified pattern.	
5.5	All static data members must follow the specified pattern.	The static data member “member_name” does not match the specified pattern.	
5.6	All static function members must follow the specified pattern.	The static function member “member_name” does not match the specified pattern.	
5.7	All bitfield members must follow the specified pattern.	The bitfield “member_name” does not match the specified pattern.	

Group 6: Enumerations

Number	Rule Applied	Message generated if rule is violated	Other details
6.1	All enumeration tags must follow the specified pattern.	The enumeration tag “tag_name” does not match the specified pattern.	
6.2	All enumeration types must follow the specified pattern.	The enumeration type “type_name” does not match the specified pattern.	This is the typedef name.
6.3	All enumeration constants must follow the specified pattern.	The enumeration constant “constant_name” does not match the specified pattern.	

Group 7: Functions

Number	Rule Applied	Message generated if rule is violated	Other details
7.1	All global functions must follow the specified pattern.	The global function “function_name” does not match the specified pattern.	A global function is a function with external linkage.
7.2	All static functions must follow the specified pattern.	The static function “function_name” does not match the specified pattern.	A static function is a function with internal linkage.
7.3	All function parameters must follow the specified pattern.	The function parameter “param_name” does not match the specified pattern.	In C++, applies to non-member functions.

Group 8: Constants

Number	Rule Applied	Message generated if rule is violated	Other details
8.1	All global constants must follow the specified pattern.	The global constant “constant_name” does not match the specified pattern.	A global constant is a constant with external linkage.
8.2	All static constants must follow the specified pattern.	The static constant “constant_name” does not match the specified pattern.	A static constant is a constant with internal linkage.
8.3	All local constants must follow the specified pattern.	The local constant “constant_name” does not match the specified pattern.	A local constant is a constant without linkage.
8.4	All static local constants must follow the specified pattern.	The static local constant “constant_name” does not match the specified pattern.	A static local constant is a constant declared static in a function.

Group 9: Variables

Number	Rule Applied	Message generated if rule is violated	Other details
9.1	All global variables must follow the specified pattern.	The global variable “var_name” does not match the specified pattern.	A global variable is a variable with external linkage.
9.2	All static variables must follow the specified pattern.	The static variable “var_name” does not match the specified pattern.	A static variable is a variable with internal linkage.
9.3	All local variables must follow the specified pattern.	The local variable “var_name” does not match the specified pattern.	A local variable is a variable without linkage.
9.4	All static local variables must follow the specified pattern.	The static local variable “var_name” does not match the specified pattern.	A static local variable is a variable declared static in a function.

Group 10: Name spaces (C++)

Number	Rule Applied	Message generated if rule is violated	Other details
10.1	All names paces must follow the specified pattern.	The name space "name space_name" does not match the specified pattern.	

Group 11: Class templates (C++)

Number	Rule Applied	Message generated if rule is violated	Other details
11.1	All class templates must follow the specified pattern.	The class template "template_name" does not match the specified pattern.	
11.2	All class template parameters must follow the specified pattern.	The class template parameter "param_name" does not match the specified pattern.	

Group 12: Function templates (C++)

Number	Rule Applied	Message generated if rule is violated	Other details
12.1	All function templates must follow the specified pattern.	The function template “template_name” does not match the specified pattern.	Applies to non-member functions.
12.2	All function template parameters must follow the specified pattern.	The function template parameter “param_name” does not match the specified pattern.	Applies to non-member functions.
12.3	All function template members must follow the specified pattern.	The function template member “member_name” does not match the specified pattern.	

Global Variables

Shared protected global variable

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

Description

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

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

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

In your verification results, these variables are colored green on the **Source**, **Results List** and **Variable Access** panes. On the **Source** pane, the coloring is applied to the variable only during declaration.

Examples

Shared Variables Protected Through Temporal Exclusion

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
```

```

        while(randomValue) {
            reset();
            inc();
            inc();
        }
    }

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}

void main() {
}

```

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

Option	Value
Configure multitasking manually on page 1-113	<input checked="" type="checkbox"/>
Entry points on page 1-116	task interrupt_handler
Temporally exclusive tasks on page 1-129	task interrupt_handler

On the command-line, you can use the following:

```

polyspace-code-prover-nodesktop
-entry-points task,interrupt_handler
-temporal-exclusions-file "C:\exclusions_file.txt"

```

where the file `C:\exclusions_file.txt` has the following line:

```
task interrupt_handler
```

The variable is shared between `task` and `interrupt_handler`. However, because `task` and `interrupt_handler` are temporally exclusive, operations on the variable cannot interrupt each other.

Shared Variables Protected Through Critical Sections

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void take_semaphore(void);
void give_semaphore(void);

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        take_semaphore();
        reset();
        inc();
        inc();
        give_semaphore();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        take_semaphore();
        interrupt();
        give_semaphore();
    }
}
```

```
void main() {
}
```

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

Option	Value	
Configure multitasking manually on page 1-113	<input checked="" type="checkbox"/>	
Entry points on page 1-116	task interrupt_handler	
Critical section details on page 1-125	Starting routine	Ending routine
	take_semaphore	give_semaphore

On the command-line, you can use the following:

```
polyspace-code-prover-nodesktop
  -entry-points task,interrupt_handler
  -critical-section-begin take_semaphore:cs1
  -critical-section-end give_semaphore:cs1
```

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

Shared Structure Variables Protected Through Access Pattern

```
struct S {
    unsigned int var_1;
    unsigned int var_2;
};

volatile int randomVal;

struct S sharedStruct;

void task1(void) {
    while(randomVal)
        operation1();
}

void task2(void) {
```

```

        while(randomVal)
            operation2();
    }

void operation1(void) {
    sharedStruct.var_1++;
}

void operation2(void) {
    sharedStruct.var_2++;
}

int main(void) {
    return 0;
}

```

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

Option	Value
Configure multitasking manually on page 1-113	<input checked="" type="checkbox"/>
Entry points on page 1-116	task1 task2

On the command-line, you can use the following:

```

polyspace-code-prover-nodesktop
    -entry-points task1,task2

```

The software determines that `sharedStruct` is protected because:

- `task1` operates only on `sharedStruct.var_1`.
- `task2` operates only on `sharedStruct.var_2`.

If you select the result, the **Result Details** pane indicates that the access pattern protects all operations on the variable. On the **Variable Access** pane, the row for variable `sharedStruct` lists `Access pattern` as the protection type.

Shared Variables Protected Through Design Pattern and Mutex

```

#include <pthread.h>
#include <stdlib.h>

```



```

pthread_mutex_t lock;
pthread_t id1, id2;

int var;

void * t1(void* b) {
    pthread_mutex_lock(&lock);
    var++;
    pthread_mutex_unlock(&lock);
}

void * t2(void* a) {
    pthread_mutex_lock(&lock);
    var = 1;
    pthread_mutex_unlock(&lock);
}

int main(void) {
    pthread_create(&id1, NULL, t1, NULL);
    pthread_create(&id2, NULL, t2, NULL);

    return 0;
}

```

`var` is a shared, protected variable if you specify the following options:

Option Name	Value
Enable automatic concurrency detection on page 1-102	<input checked="" type="checkbox"/>

On the command-line, you can use the following:

```

polyspace-code-prover-nodesktop
    -enable-concurrency-detection

```

In this example, if you specify the concurrency detection option, Polyspace Code Prover detects that your program uses multitasking. Two task, `lock` and `var`, share two variables. `lock` is a pthread mutex variable, which `pthread_mutex_lock` and `pthread_mutex_unlock` use to lock and unlock their mutexes. The inherent pthread design patterns protect `lock`. The **Results Details** pane and **Variable Access** pane list Design Pattern as the protection type.

The mutex locking and unlocking mechanisms protect `var`, the other shared variable. The **Results Details** pane and **Variable Access** pane list `Mutex` as the protection type.

Check Information

Language: C | C++

See Also

Entry points (`-entry-points`) | Critical section details (`-critical-section-begin -critical-section-end`) | Temporally exclusive tasks (`-temporal-exclusions-file`) | Shared unprotected global variable | Non-shared used global variable | Non-shared unused global variable

Topics

“Multitasking”

Shared unprotected global variable

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

Description

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

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

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

In your verification results, these variables are colored orange on the **Source**, **Results List** and **Variable Access** panes. On the **Source** pane, the coloring is applied to the variable only during declaration.

Examples

Unprotected Shared Variables

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
```

```

        reset();
        inc();
        inc();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}

void main() {
}

```

In this example, `shared_var` is an unprotected shared variable if you specify the following multitasking options:

Option	Value
Configure multitasking manually on page 1-113	<input checked="" type="checkbox"/>
Entry points on page 1-116	task interrupt_handler

You do not specify protection mechanisms such as critical sections.

The operation `shared_var = INT_MAX` can interrupt the other operations on `shared_var` and cause unpredictable behavior.

Check Information

Language: C | C++

See Also

Entry points (-entry-points) | Critical section details (-critical-section-begin -critical-section-end) | Temporally exclusive tasks (-

temporal-exclusions-file) | Shared protected global variable | Non-shared used global variable | Non-shared unused global variable

Topics

“Review Global Variable Usage”

“Multitasking”

Non-shared used global variable

Global variables used in a single task

Description

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

- The variable is used only in a single task.
- Polyspace detects at least one read or write operation on the variable.

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

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

Examples

Used and Unused Global Variables

```
int var1;
int var2;
int var3;
int var4;

int input(void);

void main() {
    int loc_var = input(), flag=0;

    var1 = loc_var;
    if(0) {
        var3 = loc_var;
    }
    if(flag!=0) {
        var4 =loc_var;
    }
}
```

```
}
```

If you verify the above code in a C project, the software lists `var2`, `var3` and `var4` as non-shared unused variables, and `var1` as a non-shared used variable.

`var3` and `var4` are used in unreachable code and are therefore marked as unused.

Note In a C++ project, the software does not list the unused variable `var2`.

Non-shared variables in multitasking code

```
unsigned int var_1;
unsigned int var_2;
volatile int randomVal;

void task1(void) {
    while(randomVal)
        operation(1);
}

void task2(void) {
    while(randomVal)
        operation(2);
}

void operation(int i) {
    if(i==1) {
        var_1++;
    }
    else {
        var_2++;
    }
}

int main(void) {
    return 0;
}
```

In this example, even when you specify `task1` and `task2` for the option `Entry points (-entry-points)`, the software determines that `var_1` and `var_2` are non-shared.

Even though both `task1` and `task2` call the function operation, because of the `if` statement in operation, `task1` can operate only on `var_1` and `task2` only on `var_2`.

Check Information

Language: C | C++

See Also

Shared protected global variable | Shared unprotected global variable
| Non-shared unused global variable

Non-shared unused global variable

Global variables declared but not used

Description

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

- The variable is declared in the code.
- Polyspace cannot detect a read or write operation on the variable.

In your verification results, these variables are colored gray on the **Source**, **Results List** and **Variable Access** panes. On the **Source** pane, the coloring is applied to the variable only during declaration.

Note The software does not display a complete list of unused global variables. Especially, in C++ projects, unused global variables can be suppressed from display.

Examples

Used and Unused Global Variables

```
int var1;
int var2;
int var3;
int var4;

int input(void);

void main() {
    int loc_var = input(), flag=0;

    var1 = loc_var;
    if(0) {
        var3 = loc_var;
    }
}
```

```
    if(flag!=0) {  
        var4 =loc_var;  
    }  
  
}
```

If you verify the above code in a C project, the software lists `var2`, `var3` and `var4` as non-shared unused variables, and `var1` as a non-shared used variable.

`var3` and `var4` are used in unreachable code and are therefore marked as unused.

Note In a C++ project, the software does not list the unused variable `var2`.

Check Information

Language: C | C++

See Also

Shared protected global variable | Shared unprotected global variable
| Non-shared used global variable

Topics

“Review Global Variable Usage”

Polyspace Report Components — Alphabetical List

Acronym Definitions

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

Description

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

See Also

Topics

“Customize Existing Report Template”

Call Hierarchy

Create table showing call graph in source code

Description

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

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

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

- File containing the function call.

In addition, the line and column is also displayed.

- File containing the function definition.

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

In addition, the table also displays uncalled functions.

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

See Also

Topics

“Customize Existing Report Template”

Code and Verification Information

Create table of verification times and code characteristics

Description

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

Properties

Include Verification Time Information

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

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

Include Code Details

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

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

See Also

Topics

“Customize Existing Report Template”

Code Metrics Details

Create table of Polyspace metrics broken down by file and function

Description

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

Properties

Project Metrics

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

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

File Metrics

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

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

Function Metrics

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

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

See Also

Topics

“Customize Existing Report Template”

Code Metrics Summary

Create table of Polyspace metrics

Description

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

See Also

Topics

“Customize Existing Report Template”

Code Verification Summary

Create table of Polyspace analysis results

Description

This component creates tables containing the following results:

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

Properties

Include Checks from Polyspace Standard Library Stub Functions

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

See Also

Topics

“Customize Existing Report Template”

Coding Rules Details

Create table of coding rule violations broken down by file

Description

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

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

Properties

Select Coding Rules Type

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

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

Display by

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

See Also

Topics

“Customize Existing Report Template”

Coding Rules Summary

Create table with number of coding rule violations

Description

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

Properties

Select Coding Rules Type

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

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

Include Files/Rules with No Problems Detected

If you select this option, the table displays:

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

Display by

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

- Rule number
- File

See Also

Topics

“Customize Existing Report Template”

Configuration Parameters

Create table of analysis options, assumptions and coding rules configuration

Description

This component creates the following tables:

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

See Also

Topics

“Customize Existing Report Template”

Defects Summary

Create table of Polyspace Bug Finder defects

Description

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

Properties

Include Checkers with No Defects Detected

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

See Also

Topics

“Customize Existing Report Template”

Global Variable Checks

Create table of Polyspace Code Prover global variables

Description

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

See Also

Topics

“Customize Existing Report Template”

Recursive Functions

Create table of recursive functions

Description

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

See Also

Topics

“Customize Existing Report Template”

Report Customization (Filtering)

Create filters that apply to your Polyspace reports

Description

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

Properties

Code Metrics Filters

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

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

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

Coding Rules Filters

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

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

Run-time Check Filters

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

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

Advanced Filters

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

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

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

See Also

Topics

“Customize Existing Report Template”

“Regular Expressions” (MATLAB)

Run-time Checks Details Ordered by Color/File

Create overrides for global filters in Polyspace reports

Description

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

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

Properties

Categories To Include

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

Property	Purpose
Red Checks	Specify whether to include red checks in your report. Red checks indicate proven run-time errors.
Gray Checks	Specify whether to include gray checks in your report. Gray checks indicate unreachable code.
Orange Checks	Specify whether to include orange checks in your report. Orange checks indicate possible run-time errors.

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

Advanced Filters

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

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

See Also

Topics

“Customize Existing Report Template”

Run-time Checks Details Ordered by Review Information

Create table with Polyspace Code Prover checks ordered by review information

Description

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

See Also

Topics

“Customize Existing Report Template”

Run-time Checks Summary Ordered by File

Create table with Polyspace Code Prover checks ordered by file

Description

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

Properties

Sort the data

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

Display as

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

Display ratio of checks in a file

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

Include checks from Polyspace standard library stub functions

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

See Also

Topics

“Customize Existing Report Template”

Software Quality Objectives - Coding Rules Summary

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

Description

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

See Also

Topics

“Customize Existing Report Template”

Software Quality Objectives - Run-time Checks Details

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

Description

This component creates tables showing run-time checks in results downloaded from Polyspace Metrics.

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

See Also

Topics

“Customize Existing Report Template”

Software Quality Objectives - Run-time Checks Summary

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

Description

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

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

See Also

Topics

“Customize Existing Report Template”

Summary By File

Create table showing summary of Polyspace results by file

Description

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

See Also

Topics

“Customize Existing Report Template”

Variable Access

Create table showing global variable access in source code

Description

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

- Variable name.

The entry for each variable is denoted by |.

- Type of the variable.
- Number of read and write operations on the variable.
- Details of read and write operations. For each read or write operation, the table displays the following information:
 - File and function containing the operation in the form *file_name.function_name*.

The entry for each read or write operation is denoted by ||. Write operations are denoted by < and read operations by >.

- Line and column number of the operation.

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

See Also

Topics

“Customize Existing Report Template”

Variable Checks Details Ordered By Review Information

Create table with Polyspace Code Prover global variable results ordered by review information

Description

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

See Also

Topics

“Customize Existing Report Template”

Configuration Parameters

- “Product mode” on page 11-2
- “Settings from (C)” on page 11-3
- “Settings from (C++)” on page 11-5
- “Use custom project file” on page 11-7
- “Project configuration” on page 11-8
- “Enable additional file list” on page 11-9
- “Stub lookup tables” on page 11-10
- “Input” on page 11-12
- “Tunable parameters” on page 11-13
- “Output” on page 11-14
- “Model reference verification depth” on page 11-15
- “Model by model verification” on page 11-17
- “Output folder” on page 11-18
- “Make output folder name unique by adding a suffix” on page 11-19
- “Add results to current Simulink project” on page 11-20
- “Open results automatically after verification” on page 11-21
- “Check configuration before verification” on page 11-22
- “Verify all occurrences” on page 11-23

Product mode

Select type of Polyspace code analysis to run.

Model Configuration Parameters Category: Polyspace

Settings

Default: Code Prover

Code Prover

Run a Polyspace Code Prover verification.

Bug Finder

Run a Polyspace Bug Finder analysis.

Dependency

You see only the products for which you have a license. If you do not have a Polyspace Code Prover license, the default product mode is Bug Finder.

Command-Line Information

Use the `pslinkoptions` property `VerificationMode`.

See Also

`pslinkoptions` | `pslinkoptions`

Related Examples

- “Run Analysis for Embedded Coder”

Settings from (C)

Select settings for the analysis configuration. You can quickly activate coding rules checking for generated C code

Model Configuration Parameters Category: Polyspace

Settings

Default: Project configuration

Project configuration

Run Polyspace with the options specified in the “Project configuration” on page 11-8 or “Use custom project file” on page 11-7.

You do not check coding rules unless you select a rule set in the configuration.

Project configuration and MISRA AC AGC checking

Run Polyspace with the options specified in the **Project configuration** plus MISRA AC-AGC obligatory and recommended rules.

Project configuration and MISRA C 2004 checking

Run Polyspace with the options specified in the **Project configuration** plus all MISRA C 2004 rules.

Project configuration and MISRA C 2012 checking

Run Polyspace with the options specified in the **Project configuration** plus all MISRA C 2012 rules. This option automatically applies the rule categories for generated code. See Use generated code requirements (-misra3-agc-mode).

MISRA AC AGC checking

Check compliance with the MISRA AC-AGC obligatory and recommended rules. After rules checking, Polyspace stops.

MISRA C 2004 checking

Check compliance with all MISRA C 2004 rules. After rules checking, Polyspace stops.

MISRA C 2012 checking

Check compliance with all MISRA C 2012 rules. This option automatically applies the rule categories for generated code. See `Use generated code requirements (-misra3-agc-mode)`. After rules checking, Polyspace stops.

Dependency

This setting overrides custom configuration settings in “Project configuration” on page 11-8 and “Use custom project file” on page 11-7. If you want to use your custom coding rule settings, select the `Project configuration` option.

Command-Line Information

Use the `pslinkoptions` property `VerificationSettings`.

See Also

`pslinkoptions` | `pslinkoptions`

Related Examples

- “Specify Type of Analysis to Perform”

Settings from (C++)

Select settings for the analysis configuration. This option allows you to quickly activate coding rules checking for generated C++ code.

Model Configuration Parameters Category: Polyspace

Settings

Default: Project configuration

Project configuration

Run Polyspace with the options specified in the “Project configuration” on page 11-8 or “Use custom project file” on page 11-7.

You do not check coding rules unless you select a rule set in the configuration.

Project configuration and MISRA C++ checking

Run Polyspace with the options specified in the **Project configuration** plus MISRA C++ required rules.

Project configuration and JSF C++ checking

Run Polyspace with the options specified in the **Project configuration** plus JSF C++ shall rules.

MISRA C++ checking

Check compliance with the MISRA C++: 2008 required rules. After rules checking, Polyspace stops.

JSF C++ checking

Check compliance with the JSF C++ shall rules. After rules checking, Polyspace stops.

Dependency

This setting overrides custom configuration settings in “Project configuration” on page 11-8 and “Use custom project file” on page 11-7. If you want to use your custom coding rule settings, select the Project configuration option.

Command-Line Information

Use the `pslinkoptions` property `CxxVerificationSettings`.

See Also

`pslinkoptions` | `pslinkoptions`

Related Examples

- “Specify Type of Analysis to Perform”

Use custom project file

Set Polyspace configuration options with a custom `.psprj` file

Model Configuration Parameters Category: Polyspace

Settings

Default: Off

Off

Analysis uses configuration options from **Project configuration** on page 11-8 parameters.

On

Analysis uses configuration options from the specified `.psprj` project file.

Dependency

The **Settings from** parameter overrides custom configuration settings for coding rules. If you want to use your custom coding rule settings, set **Settings from > Project configuration**.

Command-Line Information

Use the `pslinkoptions` properties `EnablePrjConfigFile` and `PrjConfigFile`. For details, see `pslinkoptionspslinkoptions`.

See Also

`pslinkoptions` | `pslinkoptions`

Related Examples

- “Configure Advanced Polyspace Analysis Options”

Project configuration

Set advanced configuration options to customize the analysis.

Settings

Open the Polyspace Configuration window by using the **Configure** button. Customize additional settings in this window and save your project configuration. If you added a custom project file in the parameter “Use custom project file” on page 11-7, that project file configuration is shown. Otherwise, the default project template is used.

For details about the advanced options, see “Analysis Options”.

Dependency

The **Settings from** parameter overrides custom configuration settings for coding rules. If you want to use your custom coding rule settings, set **Settings from > Project configuration**.

Command-Line Information

Use `polyspace.ModelLinkCodeProverOptions` with the `pslinkoptions` properties `EnablePrjConfigFile` and `PrjConfigFile`.

See Also

`polyspace.ModelLinkCodeProverOptions` | `pslinkoptions` | `pslinkoptions`

Related Examples

- “Configure Advanced Polyspace Analysis Options”

More About

- “Analysis Options”

Enable additional file list

Add additional supporting code files to the analysis.

Model Configuration Parameters Category: Polyspace

Settings

Default: Off

Off

The analysis includes no additional files.

On

Polyspace analyzes the specified C/C++ files with the generated code. Use the **Select files** button to specify these additional files.

Command-Line Information

Use the `pslinkoptions` properties `EnableAdditionalFileList` and `AdditionalFileList`.

See Also

`pslinkoptions` | `pslinkoptions`

Related Examples

- “Include Handwritten Code”

Stub lookup tables

Specify that the verification must stub auto-generated functions that use certain kinds of lookup tables in their body. The lookup tables in these functions use linear interpolation and do not allow extrapolation. That is, the result of using the lookup table always lies between the lower and upper bounds of the table.

If you use this option, the verification is more precise and has fewer orange checks. The verification of lookup table functions is usually imprecise. The software has to make certain assumptions about these functions. To avoid missing a run-time error, the verification assumes that the result of using the lookup table is within the full range allowed by the result data type. This assumption can cause many unproven results (orange checks) when a lookup table function is called. By using this option, you narrow down the assumption. For functions using lookup tables with linear interpolation and no extrapolation, the result is at least within the bounds of the table.

The option is relevant only if your model uses Lookup Table blocks.

Model Configuration Parameters Category: Polyspace

Settings

Default: On

On

For autogenerated functions that use lookup tables with linear interpolation and no extrapolation, the verification:

- Does not check for run-time errors in the function body.
- Calls a function stub instead of the actual function at the function call sites. The stub ensures that the result of using the lookup table is within the bounds of the table.

To identify if the lookup table in the function uses linear interpolation and no extrapolation, the verification uses information provided by the code generation product. For instance, if you use Embedded Coder to generate code, the lookup table functions with linear interpolation and no extrapolation follow specific naming conventions.

Off

The verification does not stub autogenerated functions that use lookup tables.

Tips

- The option applies only to autogenerated functions. If you integrate your own C/C++ S-Function using lookup tables with the model, the option does not cause them to be stubbed.
- The option is on by default. For certification purposes, if you want your verification tool to be independent of the code generation tool, turn off the option.

Command-Line Information

Use the `pslinkoptions` property `AutoStubLUT`.

See Also

`pslinkoptions` | `pslinkoptions`

Input

Choose whether to constrain input block variables.

Model Configuration Parameters Category: Polyspace

Settings

Default: Use specified minimum and maximum values

Use specified minimum and maximum values

Analysis assumes minimum and maximum values for input variables. These values are specified in the input block dialog box. Use this value to reduce the number of orange results.

Unbounded inputs

Analysis assumes full range for input variables. Use this value to run a robust analysis that includes values outside the expected range.

Command-Line Information

Use the `pslinkoptions` property `InputRangeMode`.

See Also

`pslinkoptions` | `pslinkoptions`

Related Examples

- “Specify Signal Ranges”

Tunable parameters

Choose how to treat tunable parameter values during the analysis. Treat values as either constants or a range of values.

Model Configuration Parameters Category: Polyspace

Settings

Default: Use calibration data

Use calibration data

Analysis assumes constant values for tunable parameters. Use this value to run a contextual analysis. This option can reduce the number of orange results.

Use specified minimum and maximum values

Analysis assumes a range of values for the tunable parameter variables. Specify maximum and minimum values in the model. Use this option to run a robust analysis that includes values outside the expected parameter value.

Command-Line Information

Use the `pslinkoptions` property `ParamRangeMode`.

See Also

`pslinkoptions` | `pslinkoptions`

Related Examples

- “Specify Signal Ranges”

Output

Choose whether to verify output values.

Code Prover option only. Bug Finder cannot check output values.

Model Configuration Parameters Category: Polyspace

Settings

Default: No verification

No verification

Polyspace does not verify output values.

Verify outputs are within minimum and maximum values

Polyspace checks to see if the output variable values are within the expected minimum and maximum values. Specify the minimum and maximum values in the output block dialog boxes.

Command-Line Information

Use the `pslinkoptions` property `OutputRangeMode`.

See Also

`pslinkoptions` | `pslinkoptions`

Related Examples

- “Specify Signal Ranges”

Model reference verification depth

Only for models that use Embedded Coder generated code. Indicate how deep into the model hierarchy to analyze.

Model Configuration Parameters Category: Polyspace

Settings

Default: `Current model only`

`Current model only`

Polyspace analyzes only the current model

1

Polyspace analyzes the current model and the referenced models that are one level below the current model.

2

Polyspace analyzes the current model and the referenced models that are up to two levels below the current model.

3

Polyspace analyzes the current model and the referenced models that are up to three levels below the current model.

All

Polyspace analyzes the current model and all referenced models.

Command-Line Information

Use the `pslinkoptions` property `ModelRefVerifDepth`.

See Also

`pslinkoptions` | `pslinkoptions`

Related Examples

- “Configure Analysis Depth for Referenced Models”

Model by model verification

Only for models that use Embedded Coder generated code. Analyze each model or referenced model individually. If you have a large project, this option can help modularize your analysis .

Model Configuration Parameters Category: Polyspace

Settings

Default: Off

Off

Polyspace analyzes your models together. Model interactions are analyzed.

On

Polyspace analyzes your model and each of its referenced models in isolation. This option does not analyze model interactions.

Command-Line Information

Use the `pslinkoptions` property `ModelRefByModelRefVerif`.

See Also

[pslinkoptions](#) | [pslinkoptions](#)

Related Examples

- “Configure Analysis Depth for Referenced Models”

Output folder

Specify the location and folder name for your analysis results.

Model Configuration Parameters Category: Polyspace

Settings

Default: `results_$(ModelName)`

Enter a path for your results folder. If you do not use a full path, the results folder is relative to your current MATLAB folder.

If you select “Add results to current Simulink project” on page 11-20, the results folder is relative to the Simulink project folder.

By default, the software stores your results in *Current Folder* `\results_model_name`.

Command-Line Information

Use the `pslinkoptions` property `ResultDir`.

See Also

`pslinkoptions` | `pslinkoptions`

Related Examples

- “Manage Results”

Make output folder name unique by adding a suffix

Add a unique suffix to the results folder for every run to avoid overwriting previous results.

Model Configuration Parameters Category: Polyspace

Settings

Default: Off

Off

Every time you rerun your analysis, your results are overwritten.

On

For each run of the analysis, Polyspace specifies a new location for the results folder by appending a unique number to the folder name.

Command-Line Information

Use the `pslinkoptions` property `AddSuffixToResultDir`.

See Also

`pslinkoptions` | `pslinkoptions`

Related Examples

- “Manage Results”

Add results to current Simulink project

Add your Polyspace results to the current Simulink project. To use this option, you must have a Simulink project open.

Model Configuration Parameters Category: Polyspace

Settings

Default: Off

Off

Results are saved to the current folder.

On

Results are saved to the currently open Simulink project.

Dependencies

You must have a Simulink project open to use this option.

Command-Line Information

Use the `pslinkoptions` property `AddToSimulinkProject`.

See Also

`pslinkoptions` | `pslinkoptions`

Related Examples

- “Manage Results”

Open results automatically after verification

Decide whether to open your results in the Polyspace interface after running analysis from Simulink.

Model Configuration Parameters Category: Polyspace

Settings

Default: On

On

After you run an analysis, your results open automatically in the Polyspace interface.

Off

You must manually open your results after running an analysis.

Command-Line Information

Use the `pslinkoptions` property `OpenProjectManager`.

See Also

[pslinkoptions](#) | [pslinkoptions](#)

Related Examples

- “Manage Results”

Check configuration before verification

Check whether model and code configurations are optimal for code analysis.

Model Configuration Parameters Category: Polyspace

Settings

Default: On (proceed with warnings)

On (proceed with warnings)

The process stops for errors, but continues the code analysis if the configuration has only warnings.

On (stop for warnings)

If the configuration has errors or warnings, the process stops.

Off

The software does not check the configuration.

Command-Line Information

Use the `pslinkoptions` property `CheckConfigBeforeAnalysis`. For details, see `pslinkoptions`.

See Also

`pslinkoptions` | `pslinkoptions`

Related Examples

- “Check Simulink Model Settings”

Verify all occurrences

For S-Function analyses only. Run an analysis on all instances of the selected S-Function.

Model Configuration Parameters Category: Polyspace

Settings

Default: Off

Off

Analyze only the selected S-Function block. The analysis includes only information from the selected S-Function block.

On

Analyze all occurrences of the S-function in the model. If the S-Function is included in the model multiple times, information from all occurrences is included in the analysis.

Command-Line Information

Use the `pslinkoptions` property `VerifALLSFcnInstances`.

See Also

`pslinkoptions` | `pslinkoptions`

Related Examples

- “Verify S-Function Code”

