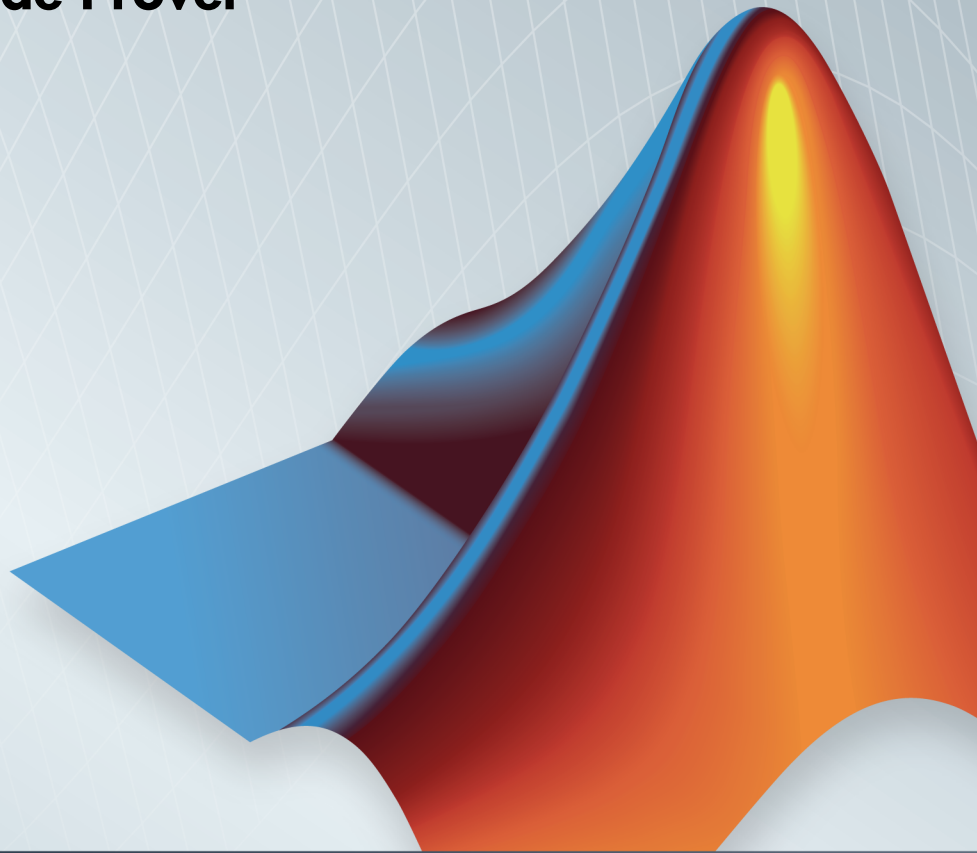


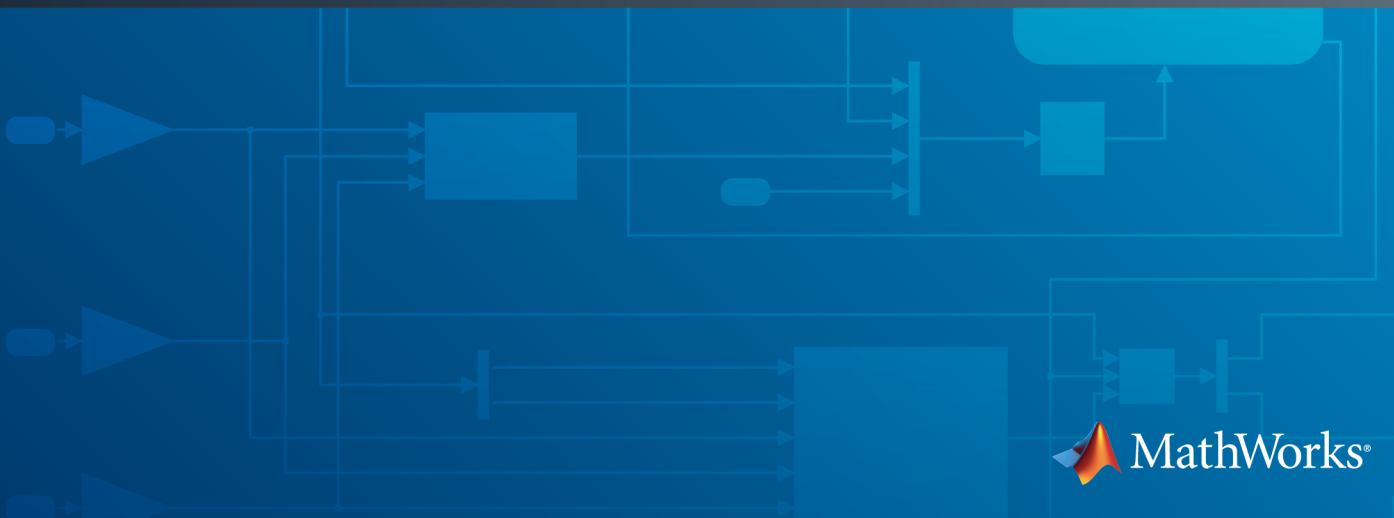
Polyspace[®] Code Prover[™]

Reference

R2014b



MATLAB[®] & SIMULINK[®]



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 2013–2014 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)

Option Descriptions

1

Target operating system (C/C++)	1-4
Settings	1-4
Dependencies	1-5
Command-Line Information	1-5
Target processor type (C)	1-6
Settings:	1-6
Tips	1-7
Command-Line Information	1-7
Generic target options (C/C++)	1-9
Command-Line Options	1-9
Dialect (C)	1-13
Settings	1-13
Dependency	1-13
Limitations	1-14
Command-Line Information	1-15
Sfr type support (C)	1-16
Settings	1-16
Dependency	1-16
Command-Line Information	1-16
Division round down (C)	1-17
Settings	1-17
Command-Line Information	1-17
Enum type definition (C)	1-18
Settings	1-18
Command-Line Information	1-18

Signed right shift (C)	1-19
Settings	1-19
Command-Line Information	1-19
Preprocessor definitions (C/C++)	1-20
Settings	1-20
Command-Line Information	1-20
Disabled preprocessor definitions (C/C++)	1-21
Settings	1-21
Command-Line Information	1-21
Code from DOS or Windows file system (C/C++)	1-22
Settings	1-22
Command-Line Information	1-22
Command/script to apply to preprocessed files (C/C++) ...	1-23
Example Script	1-23
Command-Line Information	1-24
Continue with compile error (C/C++)	1-25
Settings	1-25
Command-Line Information	1-25
Include (C/C++)	1-26
Settings	1-26
Command-Line Information	1-26
Include folders (C/C++)	1-27
Settings	1-27
Command-Line Information	1-27
Multitasking (C/C++)	1-28
Settings	1-28
Dependencies	1-28
Command-Line Information	1-28
Entry points (C/C++)	1-30
Settings	1-30
Dependencies	1-30
Tips	1-30
Command-Line Information	1-31

Critical section details (C/C++)	1-32
Settings	1-32
Dependencies	1-32
Tips	1-32
Command-Line Information	1-32
Temporally exclusive tasks (C/C++)	1-34
Settings	1-34
Dependencies	1-34
Command-Line Information	1-34
Check MISRA C:2004	1-36
Settings	1-36
Tips	1-37
Command-Line Information	1-37
Check MISRA AC AGC	1-38
Settings	1-38
Tips	1-39
Command-Line Information	1-39
Check MISRA C:2012	1-40
Settings	1-40
Tips	1-41
Command-Line Information	1-41
Use generated code requirements (C)	1-42
Settings	1-42
Dependency	1-43
Command-Line Information	1-43
Check custom rules (C/C++)	1-44
Settings	1-44
Command-Line Information	1-45
Files and folders to ignore (C)	1-47
Settings	1-47
Dependencies	1-47
Command-Line Information	1-47
Effective boolean types (C)	1-49
Settings	1-49
Dependencies	1-49

Command-Line Information	1-49
Allowed pragmas (C)	1-51
Settings	1-51
Dependencies	1-51
Command-Line Information	1-51
Verify whole application (C/C++)	1-52
Settings	1-52
Command-Line Information	1-52
Verify module (C)	1-53
Settings	1-53
Command-Line Information	1-53
Variables to initialize (C)	1-55
Settings	1-55
Dependencies	1-55
Command-Line Information	1-55
Initialization functions (C)	1-57
Settings	1-57
Tips	1-57
Command-Line Information	1-57
Dependencies	1-57
Functions to call (C)	1-59
Settings	1-59
Dependencies	1-59
Tips	1-59
Command-Line Information	1-60
Verify files independently (C/C++)	1-61
Settings	1-61
Dependencies	1-61
Tips	1-61
Command-Line Information	1-61
Common source files (C/C++)	1-63
Settings	1-63
Dependencies	1-63
Command-Line Information	1-63

Parameters (C)	1-64
Settings	1-64
Command-Line Information	1-64
Inputs (C)	1-66
Settings	1-66
Command-Line Information	1-66
Initialization functions (C)	1-68
Settings	1-68
Command-Line Information	1-68
Step functions (C)	1-69
Settings	1-69
Tips	1-69
Command-Line Information	1-69
Termination functions (C)	1-71
Settings	1-71
Command-Line Information	1-71
Variable/function range setup (C/C++)	1-72
Settings	1-72
Command-Line Information	1-72
Ignore default initialization of global variables (C)	1-74
Settings	1-74
Tips	1-74
Command-Line Information	1-74
No automatic stubbing (C/C++)	1-76
Settings	1-76
Tips	1-76
Command-Line Information	1-76
Functions to stub (C)	1-78
Settings	1-78
Command-Line Information	1-78
Respect types in fields (C/C++)	1-79
Settings	1-79
Command-Line Information	1-79

Respect types in global variables (C/C++)	1-81
Settings	1-81
Command-Line Information	1-81
Ignore float rounding (C/C++)	1-83
Settings	1-83
Command-Line Information	1-83
Green absolute address checks (C/C++)	1-84
Settings	1-84
Tips	1-84
Command-Line Information	1-84
Ignore overflowing computations on constants (C/C++) ...	1-85
Settings	1-85
Tips	1-85
Command-Line Information	1-85
Allow negative operand for left shifts (C/C++)	1-86
Settings	1-86
Command-Line Information	1-86
Detect overflows (C/C++)	1-87
Settings	1-87
Tips	1-87
Command-Line Information	1-88
Detect Overflows in Buffer Size Computation	1-89
Overflow computation mode (C/C++)	1-91
Settings	1-91
Command-Line Information	1-92
Enable pointer arithmetic across fields (C)	1-93
Settings	1-93
Tips	1-93
Command-Line Information	1-93
Allow incomplete or partial allocation of structures (C) ..	1-95
Settings	1-95
Tips	1-96
Command-Line Information	1-96

Permissive function pointer calls (C)	1-97
Settings	1-97
Tips	1-97
Command-Line Information	1-97
Detect uncalled functions (C/C++)	1-98
Settings	1-98
Command-Line Information	1-98
Precision level (C/C++)	1-99
Settings	1-99
Tips	1-99
Command-Line Information	1-99
Verification level (C)	1-101
Settings	1-101
Tips	1-101
Dependency	1-103
Command-Line Information	1-103
Verification time limit (C/C++)	1-104
Settings	1-104
Command-Line Information	1-104
Retype variables of pointer types (C)	1-105
Settings	1-105
Command-Line Information	1-105
Retype symbols of integer types (C)	1-106
Settings	1-106
Dependencies	1-106
Tips	1-107
Command-Line Information	1-107
Sensitivity context (C/C++)	1-108
Settings	1-108
Command-Line Information	1-108
Improve precision of interprocedural analysis (C/C++) ..	1-110
Settings	1-110
Tips	1-110
Command-Line Information	1-110

Specific precision (C)	1-111
Settings	1-111
Command-Line Information	1-111
Optimize large static initializers (C)	1-112
Settings	1-112
Command-Line Information	1-112
Reduce task complexity (C)	1-113
Settings	1-113
Command-Line Information	1-113
Inline (C)	1-114
Settings	1-114
Tips	1-114
Command-Line Information	1-115
Depth of verification inside structures (C/C++)	1-116
Settings	1-116
Command-Line Information	1-116
Generate report (C/C++)	1-117
Settings	1-117
Tips	1-117
Command-Line Information	1-117
Report template (C/C++)	1-119
Settings	1-119
Dependencies	1-121
Command-Line Information	1-121
Output format (C/C++)	1-122
Settings	1-122
Tips	1-122
Dependencies	1-122
Command-Line Information	1-122
Batch (C/C++)	1-124
Settings	1-124
Dependency	1-125
Command-Line Information	1-125

Add to results repository (C/C++)	1-126
Settings	1-126
Dependency	1-126
Command-Line Information	1-126
Command/script to apply after the end of the code	
verification (C/C++)	1-127
Settings	1-127
Command-Line Information	1-127
Automatic Orange Tester (C)	1-128
Settings	1-128
Tips	1-128
Command-Line Information	1-129
Number of automatic tests (C)	1-130
Settings	1-130
Dependencies	1-130
Command-Line Information	1-130
Maximum loop iterations (C)	1-131
Settings	1-131
Dependencies	1-131
Command-Line Information	1-131
Maximum test time (C)	1-132
Settings	1-132
Dependencies	1-132
Command-Line Information	1-132
Other (C)	1-133
-extra-flags	1-133
-c-extra-flags	1-133
-cfe-extra-flags	1-133
-il-extra-flags	1-134

Target processor type (C++)	2-3
Settings	2-3
Tips	2-4
Command-Line Information	2-4
Dialect (C++)	2-5
Settings	2-5
Dependencies	2-6
Limitations	2-6
Command-Line Information	2-8
C++11 Extensions (C++)	2-10
Settings	2-10
Dependencies	2-10
Command-Line Information	2-10
Block char16/32_t types (C++)	2-11
Settings	2-11
Dependencies	2-11
Command-Line Information	2-11
Enum type definition (C++)	2-12
Settings	2-12
Command-Line Information	2-12
Pack alignment value (C++)	2-13
Settings	2-13
Dependencies	2-13
Command-Line Information	2-13
Ignore pragma pack directives (C++)	2-14
Settings	2-14
Dependencies	2-14
Command-Line Information	2-14
Support managed extensions (C++)	2-15
Settings	2-15
Dependencies	2-15
Command-Line Information	2-15

Import folder (C++)	2-16
Settings	2-16
Dependencies	2-16
Command-Line Information	2-16
Management of scope of 'for loop' variable index (C++) ...	2-17
Settings	2-17
Command-Line Information	2-17
Management of wchar_t (C++)	2-18
Settings	2-18
Command-Line Information	2-18
Set wchar_t to unsigned long (C++)	2-19
Settings	2-19
Command-Line Information	2-19
Set size_t to unsigned long (C++)	2-20
Settings	2-20
Command-Line Information	2-20
Ignore link errors (C++)	2-21
Settings	2-21
Command-Line Information	2-21
Check MISRA C++ rules	2-22
Settings	2-22
Command-Line Information	2-23
Check JSF C++ rules	2-24
Settings	2-24
Tips	2-25
Command-Line Information	2-25
Files and folders to ignore (C++)	2-26
Settings	2-26
Dependencies	2-26
Command-Line Information	2-26
Main entry point (C++)	2-28
Settings	2-28
Dependencies	2-28
Command-Line Information	2-28

Verify module (C++)	2-30
Settings	2-30
Command-Line Information	2-30
Class (C++)	2-32
Settings	2-32
Dependencies	2-32
Tips	2-32
Command-Line Information	2-32
Functions to call within the specified classes (C++)	2-34
Settings	2-34
Dependencies	2-35
Command-Line Information	2-35
Analyze class contents only (C++)	2-37
Settings	2-37
Dependencies	2-37
Tips	2-37
Command-Line Information	2-37
Skip member initialization check (C++)	2-39
Settings	2-39
Dependencies	2-39
Command-Line Information	2-39
Functions to call (C++)	2-40
Settings	2-40
Dependencies	2-40
Tips	2-40
Command-Line Information	2-41
Variables to initialize (C++)	2-42
Settings	2-42
Dependencies	2-42
Command-Line Information	2-42
Initialization functions (C++)	2-44
Settings	2-44
Command-Line Information	2-44
Dependencies	2-44

Parameters (C++)	2-46
Settings	2-46
Command-Line Information	2-46
Inputs (C++)	2-48
Settings	2-48
Command-Line Information	2-48
Initialization functions (C++)	2-50
Settings	2-50
Command-Line Information	2-50
Step functions (C++)	2-51
Settings	2-51
Tips	2-51
Command-Line Information	2-51
Termination functions (C++)	2-53
Settings	2-53
Tips	2-53
Command-Line Information	2-53
No STL stubs (C++)	2-54
Settings	2-54
Tips	2-54
Command-Line Information	2-54
Functions to stub (C++)	2-55
Settings	2-55
Command-Line Information	2-55
Tuning Precision and Scaling Parameters	2-57
Precision versus Time of Verification	2-57
Precision versus Code Size	2-57
Verification level (C++)	2-59
Settings	2-59
Tips	2-59
Dependency	2-60
Command-Line Information	2-60
Inline (C++)	2-61
Settings	2-61

Tips	2-61
Command-Line Information	2-61
Other (C++)	2-62
-extra-flags	2-62
-cpp-extra-flags	2-62
-il-extra-flags	2-62

Polyspace Analysis Options — Command Line Only

3

Check Reference

4

Approximations Used During Verification

5

Why Polyspace Verification Uses Approximations	5-2
What is Static Verification	5-2
Exhaustiveness	5-3
Approximations Made by Polyspace Verification	5-4
Volatile Variables	5-4
Structures with Volatile Fields	5-4
Absolute Addresses	5-5
Pointer Comparison	5-5
Shared Variables	5-5
Trigonometric Functions	5-6
Unions	5-6
Constant Pointer	5-7
Variable Cast as Void Pointer	5-7
Limitations of Polyspace Verification	5-9

Examples

6

Complete Examples	6-2
Simple C Example	6-2
Apache Example	6-2
cxref Example	6-3
T31 Example	6-3
Dishwasher1 Example	6-3
Satellite Example	6-4

Functions

7

Option Descriptions

- “Target operating system (C/C++)” on page 1-4
- “Target processor type (C)” on page 1-6
- “Generic target options (C/C++)” on page 1-9
- “Dialect (C)” on page 1-13
- “Sfr type support (C)” on page 1-16
- “Division round down (C)” on page 1-17
- “Enum type definition (C)” on page 1-18
- “Signed right shift (C)” on page 1-19
- “Preprocessor definitions (C/C++)” on page 1-20
- “Disabled preprocessor definitions (C/C++)” on page 1-21
- “Code from DOS or Windows file system (C/C++)” on page 1-22
- “Command/script to apply to preprocessed files (C/C++)” on page 1-23
- “Continue with compile error (C/C++)” on page 1-25
- “Include (C/C++)” on page 1-26
- “Include folders (C/C++)” on page 1-27
- “Multitasking (C/C++)” on page 1-28
- “Entry points (C/C++)” on page 1-30
- “Critical section details (C/C++)” on page 1-32
- “Temporally exclusive tasks (C/C++)” on page 1-34
- “Check MISRA C:2004” on page 1-36
- “Check MISRA AC AGC” on page 1-38
- “Check MISRA C:2012” on page 1-40
- “Use generated code requirements (C)” on page 1-42
- “Check custom rules (C/C++)” on page 1-44
- “Files and folders to ignore (C)” on page 1-47

- “Effective boolean types (C)” on page 1-49
- “Allowed pragmas (C)” on page 1-51
- “Verify whole application (C/C++)” on page 1-52
- “Verify module (C)” on page 1-53
- “Variables to initialize (C)” on page 1-55
- “Initialization functions (C)” on page 1-57
- “Functions to call (C)” on page 1-59
- “Verify files independently (C/C++)” on page 1-61
- “Common source files (C/C++)” on page 1-63
- “Parameters (C)” on page 1-64
- “Inputs (C)” on page 1-66
- “Initialization functions (C)” on page 1-68
- “Step functions (C)” on page 1-69
- “Termination functions (C)” on page 1-71
- “Variable/function range setup (C/C++)” on page 1-72
- “Ignore default initialization of global variables (C)” on page 1-74
- “No automatic stubbing (C/C++)” on page 1-76
- “Functions to stub (C)” on page 1-78
- “Respect types in fields (C/C++)” on page 1-79
- “Respect types in global variables (C/C++)” on page 1-81
- “Ignore float rounding (C/C++)” on page 1-83
- “Green absolute address checks (C/C++)” on page 1-84
- “Ignore overflowing computations on constants (C/C++)” on page 1-85
- “Allow negative operand for left shifts (C/C++)” on page 1-86
- “Detect overflows (C/C++)” on page 1-87
- “Detect Overflows in Buffer Size Computation” on page 1-89
- “Overflow computation mode (C/C++)” on page 1-91
- “Enable pointer arithmetic across fields (C)” on page 1-93
- “Allow incomplete or partial allocation of structures (C)” on page 1-95
- “Permissive function pointer calls (C)” on page 1-97

- “Detect uncalled functions (C/C++)” on page 1-98
- “Precision level (C/C++)” on page 1-99
- “Verification level (C)” on page 1-101
- “Verification time limit (C/C++)” on page 1-104
- “Retype variables of pointer types (C)” on page 1-105
- “Retype symbols of integer types (C)” on page 1-106
- “Sensitivity context (C/C++)” on page 1-108
- “Improve precision of interprocedural analysis (C/C++)” on page 1-110
- “Specific precision (C)” on page 1-111
- “Optimize large static initializers (C)” on page 1-112
- “Reduce task complexity (C)” on page 1-113
- “Inline (C)” on page 1-114
- “Depth of verification inside structures (C/C++)” on page 1-116
- “Generate report (C/C++)” on page 1-117
- “Report template (C/C++)” on page 1-119
- “Output format (C/C++)” on page 1-122
- “Batch (C/C++)” on page 1-124
- “Add to results repository (C/C++)” on page 1-126
- “Command/script to apply after the end of the code verification (C/C++)” on page 1-127
- “Automatic Orange Tester (C)” on page 1-128
- “Number of automatic tests (C)” on page 1-130
- “Maximum loop iterations (C)” on page 1-131
- “Maximum test time (C)” on page 1-132
- “Other (C)” on page 1-133

Target operating system (C/C++)

Specify the operating system of your target application. This option is available on the **Configuration** pane under the **Target & Compiler** node.

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

A generic set of includes is provided with Polyspace[®]. These are automatically included when the operating system is set to **no-predefined-OS** or **Linux**. For projects developed for other operating systems, analyze these projects using the corresponding include files for that operating system.

Settings

Default: no-predefined-OS

no-predefined-OS

Analyzes with a general operating system set up. Use with preprocessor macros (-U or -D) to specify the system flags at compilation time.

Linux

Analyzes with the Linux[®] system definitions.

Solaris

Analyzes with the Solaris[™] system definitions.

This option requires you to add a path to the Solaris include folder in your project, or use the -I option at the command line.

VxWorks

Analyzes with the VxWorks[®] system definitions.

This option requires you to add a path to the VxWorks include folder in your project, or use the -I option at the command line.

Visual

Analyzes with the Visual Studio[®] system definitions. Used for Microsoft[®] Windows[®] systems.

This option requires you to add a path to the Visual Studio include folder in your project, or use the -I option at the command line.

Dependencies

Setting this parameter changes the available **Dialect** options. All options are available with the `no-predefined-OS` option. The other operating systems only show usable dialects for that system.

Command-Line Information

Parameter: `-os-target`

Value: `no-predefined-OS` | `Linux` | `Solaris` | `VxWorks` | `Visual`

Default: `no-predefined-OS`

Example: `polyspace-code-prover-nodesktop -os-target Linux`

See Also

“Dialect (C)” on page 1-13 | “Dialect (C++)” on page 2-5

Related Examples

- “Specify Analysis Options”

More About

- “Compile Operating System Dependent Code”

Target processor type (C)

Specify the target processor type. This option is available on the **Configuration** pane under the **Target & Compiler** node.

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

Settings:

Default: i386

You can modify some default attributes by selecting the browse button to the right of the **Target processor type** drop-down menu. The optional settings for each target are shown in [brackets] in the table.

Target	char	short	int	long	long long	float	double	long double	ptr	sign of char	endian	align
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 / ColdFire ^a	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	40 ^b	32	signed	Little	32
sharc21x61	32	32	32	32	64	32	32 [64]	32 [64]	32	signed	Little	32
NEC-V850	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]
hc12	8	16	16 [32]	32	32	32	32 [64]	32 [64]	32 ^e	signed	Big	32 [16]
mpc5xx	8	16	32	32	64	32	32 [64]	32 [64]	32	signed	Big	32 [16]

Target	char	short	int	long	long long	float	double	long double	ptr	sign of char	endian	align
c18	8	16	16	32 [24] ^e	32	32	32	32	16 [24]	signed	Little	8
x86_64	8	16	32	64 [32]	64	32	64	128	64	signed	Little	64 [32]
mcpu... (Advanced)	8 [16]	8 [16]	16 [32]	32	32 [64]	32	32 [64]	32 [64]	16 [32]	signed	Little	32 [16, 8]

- a. The M68k family (68000, 68020, etc.) includes the “ColdFire” processor
- b. Operations on long double values will be imprecise.
- 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.
- f. `mcpu` is a reconfigurable Micro Controller/Processor Unit target. You can use this type to configure one or more generic targets.

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 processor described above, contact MathWorks® technical support for advice.

Command-Line Information

Parameter: `-target`

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

Default: `i386`

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

See Also

“Generic target options (C/C++)” on page 1-9

Related Examples

- “Specify Analysis Options”

- “Modify Predefined Target Processor Attributes”
- “Define Generic Target Processors”

Generic target options (C/C++)

The **Generic target options** dialog box is only available when you select a `mcpu` target for **Target processor type**. The option **Target processor type** is available on the **Configuration** pane under the **Target & Compiler** node.

Allows the specification of a generic “Micro Controller/Processor Unit” target. Use the dialog box to specify the name of a new `mcpu` target — e.g., *MyTarget*.

The generic target option is incompatible with either:

- **Target operating system** set to `Visual`
- **Dialect** set to `visual*`

That new target is added to the **Target processor type** option list. The default characteristics of the new target are (using the *type [size, alignment]* format):

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

Changing the genetic target has consequences for:

- Detection of overflow
- Computation of `sizeof` objects

Command-Line Options

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

Option	Description	Available With...	Example
-little-endian	<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>	mcpu	polyspace-code-prover-nodesktop -lang c -target mcpu -little-endian
-big-endian	<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	polyspace-code-prover-nodesktop -target mcpu -big-endian
-default-sign-of-char [signed unsigned]	<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	polyspace-code-prover-nodesktop -default-sign-of-char unsigned -target mcpu
-char-is-16bits	<p>char defined as 16 bits and all objects have a minimum alignment of 16 bits</p> <p>Incompatible with -short-is-8bits and -align 8</p>	mcpu	polyspace-code-prover-nodesktop -target mcpu -char-is-16bits

Option	Description	Available With...	Example
<code>-short-is-8bits</code>	Define <code>short</code> as 8 bits, regardless of sign	mcpu	<code>polyspace-code-prover-nodesktop -target mcpu -short-is-8bits</code>
<code>-int-is-32bits</code>	Define <code>int</code> as 32 bits, regardless of sign. Alignment is also set to 32 bits.	mcpu, hc08, hc12, mpc5xx	<code>polyspace-code-prover-nodesktop -target mcpu -long-long-is-64bits</code>
<code>-long-long-is-64bits</code>	Define <code>long long</code> 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 <code>double</code> and <code>long double</code> as 64 bits, regardless of sign. Alignment is also set to 64 bits.	mcpu, sharc21x6, hc08, hc12, mpc5xx	<code>polyspace-code-prover-nodesktop -target mcpu -double-is-64bits</code>
<code>-pointer-is-32bits</code>	Define <code>pointer</code> as 32 bits, regardless of sign. Alignment is also 32 bits.	mcpu	<code>polyspace-code-prover-nodesktop -target mcpu -pointer-is-32bits</code>
<code>-align [32 16 8]</code>	Specifies the largest alignment of struct or array objects to the 32, 16 or 8 bit boundaries. Consequently, the array or struct storage is strictly determined by the size of the individual data objects without member and end padding.	mcpu, Only 16 or 32 bits for: hc08, hc12, mpc5xx	<code>polyspace-code-prover-nodesktop -target mcpu -align 16</code>

See Also

“Target processor type (C)” on page 1-6 | “Target processor type (C++)” on page 2-3

Related Examples

- “Define Generic Target Processors”

More About

- “Common Generic Targets”

Dialect (C)

Allow syntax associated with C language extensions. This option is available on the **Configuration** pane under the **Target & Compiler** node.

Using this option allows additional structure types as keywords of the language, such as `sfr`, `sbit`, and `bit`. These structures and associated semantics are part of the compiler that extends the ANSI[®] C language.

Settings

Default: none

none

Analysis allows only ANSI C standard syntax.

gnu4.6

Analysis allows GCC 4.6 dialect syntax.

gnu4.7

Analysis allows GCC 4.7 dialect syntax.

visual10

Analysis allows Visual C++[®] 2010 syntax.

visual11.0

Analysis allows Visual C++ 2012 syntax.

keil

Analysis allows non-ANSI C syntax and semantics associated with the Keil[™] 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).

Dependency

This parameter is dependant on the value of **Target operating system**. The dialect options work only with the applicable operating systems. You can use every dialect with the **Target operating system** option, `no-predefined-OS`.

Limitations

Polyspace does not support certain aspects of the GNU[®] 4.7 dialect. These limitations can cause compilation errors, incomplete results, or false positives.

- **Vector types and attributes** — Not supported, ignores attributes.

Workaround: To reduce compilation issues

- At the command line, use the option `-D _EMMINTRIN_H_INCLUDED -D _XMMINTRIN_H_INCLUDED`.
- In the Polyspace environment, in **Macros > Preprocessor definitions**, add two rows: `_EMMINTRIN_H_INCLUDED` and `_XMMINTRIN_H_INCLUDED`.
- **Visibility attributes** — Not supported, ignored. This limitation can cause C++ linkage problems in Polyspace Code Prover[™].


Workaround: Remove all attributes during preprocessing,

- At the command line, use the option `-D __attribute__(x)=`.
- In the Polyspace environment, in **Macros > Preprocessor definitions**, add a row: `__attribute__(x)=`.
- **Complex types** — Only floating complex types supported, integral complex types cause an error.
- **Using built-in library function on complex types** — Not supported, stubbed during analysis. Calls to these functions will return variables with full ranges.

Workaround: To make the analysis more precise, add an include file that defines the functions for complex variables.

- **Computed goto** — Not supported, causes an error in Code Prover.


Workaround: To ignore the computed gotos in Code Prover, stub the functions containing the computed gotos:

- At the command line, use the option `-functions-to-stub funcList` where `funcList` is the list of functions containing the computed gotos.
- In the Polyspace environment, in the **Inputs & Stubbing > Functions to stub** table, use the  button to add a row for each function containing the computed gotos.

- **Nested functions** — Not supported, causes an error.
- **Using built-in library functions on atomic operators** — Not supported, Polyspace stubs the functions. This limitation can cause imprecise results.
- **IEEE® floating point library functions** — Not supported, causes compilation error.

This limitation includes `isnan`, `isnanf`, `isnanl`, `isinf`, `isinf`, `isinfl`, `isnormal`, and `isfinite`.

Workaround: In each of your source files, include a file containing the function definitions or declarations:

- At the command line, use the option `-include filename`.
- In the Polyspace environment, in **Environment Settings > Include**, use the  button to add a row for your definition/declaration file.

Command-Line Information

Parameter: `-dialect`

Value: `none` | `gnu4.6` | `gnu4.7` | `visual10` | `visual11.0` | `keil` | `iar`

Default: `none`

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

See Also

“Target operating system (C/C++)” on page 1-4 | “Target processor type (C)” on page 1-6

Related Examples

- “Verify Keil or IAR Dialects”

Sfr type support (C)

Specify the `sfr` types. This option is available on the **Configuration** pane under the **Target & Compiler** node.

If the code uses `sfr` keywords, you must declare each `sfr` type using this option.

Settings

Default: None

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

Dependency

Setting **Dialect** to `keil` or `iar` enables this parameter.

Command-Line Information

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

Name Value: an `sfr` name

Size Value: `8 | 16 | 32`

Default: None

Example: `polyspace-code-prover-nodesktop -lang c -dialect iar -sfr-types sfr=8,sfr32=32,sfrb=16 ...`

Division round down (C)

Specify how division and modulus of a negative numbers is interpreted by the analysis. This option is available on the **Configuration** pane under the **Target & Compiler** node.

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

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

Settings

Default: Off

Off

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

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

On

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

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

Command-Line Information

Parameter: `-div-round-down`

Default: Off

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

Enum type definition (C)

Allow the analysis to use different base types to represent an enumerated type, depending on the enumerator values and the selected definition. This option is available on the **Configuration** pane under the **Target & Compiler** node.

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

Settings

Default: signed-int

signed-int

Uses the signed integer type for all dialects except gnu.

For the gnu dialects, it uses the first type that can hold all of the enumerator values from the following list: signed int, unsigned int, signed long, unsigned long, signed long long, unsigned long long.

auto-signed-first

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

auto-unsigned-first

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

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

Command-Line Information

Parameter: -enum-type-definition

Value: signed-int | auto-signed-first | auto-unsigned-first

Default: signed-int

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

Signed right shift (C)

Choose between arithmetical and logical computation. This option is available on the **Configuration** pane under the **Target & Compiler** node.

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
```

Command-Line Information

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

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

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


Preprocessor definitions (C/C++)

Define macro compiler flags to be used during compilation phase. This option is available on the **Configuration** pane under the **Macros** node.

Some defines are applied by default, depending on your **Target operating system**.

Settings

Default: None

Using the  button, add a new row for each macro flag. The flag must be in the format *Flag=Value*. If you want Polyspace to ignore the flag, leave the *Value* blank.

For example,

- `name1=name2` replaces all instances of `name1` by `name2`.
- `name=` tells the software to ignore `name`.
- `name` with no equals sign or value replaces all instances of `name` by 1.

Command-Line Information

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

Parameter: `-D`

Default: None

Value: *flag=value*

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

See Also

“Disabled preprocessor definitions (C/C++)” on page 1-21


Disabled preprocessor definitions (C/C++)

Disable macro compiler flags. This option is available on the **Configuration** pane under the **Macros** node.

Some **Target operating system** settings enable macro compilation flags by default. This option allows you disable these macros.

Settings

Default: None

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

Command-Line Information

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

Parameter: `-U`

Default: None

Value: *flag*

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

See Also

“Preprocessor definitions (C/C++)” on page 1-20

Code from DOS or Windows file system (C/C++)

Specify that DOS or Windows files are in analysis. This option is available on the **Configuration** pane under the **Environment Settings** node.

Use this options if the contents of the **Include** or **Source** folder come from a DOS or Windows file system. It deals with upper/lower case sensitivity and control character issues.

Settings

Default: On

On

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: On

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

Command/script to apply to preprocessed files (C/C++)

Specify a perl script to run on each source file after the preprocessing phase. This option is available on the **Configuration** pane under the **Environment Settings** node.

When this option is used, the specified script file or command is run just after the preprocessing phase on each preprocessed `.c` file.

The command should be designed to process the standard output from preprocessing and produce its results in accordance with that standard output. Additionally, It is important to preserve the number of lines in the preprocessed `.ci` file. Adding a line or removing one could result in some unpredictable behavior on the location of checks and MACROS in the Polyspace viewer.

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

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

Example Script

This script, called `replace_keywords`, replaces the keyword “Volatile” by “Import”.

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

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

To run this script on preprocessed files:

- On a Linux or Mac workstation: `polyspace-code-prover-nodesktop -post-preprocessing-command 'pwd'/replace_keywords`
- On a Windows workstation you must give the full path to the Perl scripter: `matlabroot\matlab\polyspace\bin\polyspace-code-prover-`

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

Command-Line Information

Parameter: -post-preprocessing-command

Default: None

Value: Path to executable file or command in quotes

Continue with compile error (C/C++)

Continue verification even if some source files do not compile. This option is available on the **Configuration** pane under the **Environment Settings** node.

Settings

Default: Off

Off

If a source file does not compile, the verification stops.

Functions that are used but not specified are stubbed automatically.

On

Continues the verification even if only one file compiles. Files that have compilation errors are not verified. This means that the results may not contain all coding rule violations or errors.

Functions that are used but not specified are stubbed automatically.

Command-Line Information

Parameter: `-continue-with-compile-error`

Default: Off

Example: `polyspace-code-prover-nodesktop -continue-with-compile-error`

Include (C/C++)

Specify files to be included by each C file involved in the analysis. This option is available on the **Configuration** pane under the **Environment Settings** node.

Settings

Default: None

Specify the file name to be included in every C 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`

Include folders (C/C++)

View the include folders used for verification.

- In the Project Manager perspective, to add include folders, on the **Project Browser**, right-click your project. Select **Add Source**.
- In the Results Manager perspective, to view the include folders you used, select **Window > Show/Hide View > Settings**. Under the node **Environment Settings**, you see the folders listed under **Include folders**.

Settings

This is a read-only option available only from the Results Manager perspective. In the Project Manager perspective, 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 (C/C++)”

Multitasking (C/C++)

Specify whether the code is intended for a multitasking application. This option is available on the **Configuration** pane under the **Multitasking** node.

Settings

Default: Off

On

The code is intended for a multitasking application.

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

Off

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 all functions. To verify all functions, Polyspace generates a `main` function and calls functions from the generated `main` in a sequence you specify. For more information, see “Verify module (C)” or “Verify module (C++)”.

Dependencies

To enable multitasking verification, you must also select **Code Prover Verification** > **Verify whole application**. Otherwise, apart from `main`, all entry point functions appear as unreachable.

Command-Line Information

There is no command-line option to solely turn on multitasking verification. However, using the option `-entry-points` turns on multitasking verification.

See Also

“Entry points (C/C++)” | “Critical section details (C/C++)” | “Temporally exclusive tasks (C/C++)”

Related Examples

- “Model Tasks”
- “Model Tasks if main Contains Infinite Loop”
- “Model Execution Sequence in Tasks”

More About

- “Requirements for Multitasking Verification”

Entry points (C/C++)

Specify functions that serve as entry points to your code. Use this option when your code is intended for multitasking. This option is available on the **Configuration** pane under the **Multitasking** node.

Settings

Default: none

Click  to add a field. Enter function name.

Dependencies

This option is enabled only if you select the **Multitasking** box.

To verify your entry point functions, under **Code Prover Verification**, select **Verify whole application**. Otherwise, apart from `main`, all entry point functions appear as unreachable.

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 a function `func` models cyclic tasks or interrupts that can run zero or more times, to specify the multiple cycles for Polyspace:
 - 1 Create a new function `newFunc` of the form


```
void newFunc (void)
```

- 2 In the body of `newFunc`, call `func` inside a loop with unspecified number of runs. Make the loop control variable `volatile int`. For example:

```
void newFunc(void) {
    volatile int randomValue = 0;
    while(randomValue) {
        func();
    }
}
```

- 3 Specify `newFunc` as entry point.

Command-Line Information

Parameter: `-entry-points`

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

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

See Also

“Critical section details (C/C++)” | “Temporally exclusive tasks (C/C++)”

Related Examples

- “Specify Analysis Options”
- “Model Tasks”
- “Model Tasks if main Contains Infinite Loop”
- “Model Execution Sequence in Tasks”

More About

- “Requirements for Multitasking Verification”


Critical section details (C/C++)

When verifying multitasking code, Polyspace considers that a critical section lies between calls to a lock function and an unlock function. Specify the two function names. This option is available on the **Configuration** pane under the **Multitasking** node.

When a task `my_task` calls a lock function `my_lock`, all other tasks calling `my_lock` must wait till `my_task` calls the corresponding unlock function.

Settings

Default: None

Click  to add a field.

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

Dependencies

This option is enabled only if you select the **Multitasking** box.

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 procedure: `func_begin`

Ending procedure: `func_end`

```
void func() {  
    func_begin(1);  
    /* Critical section code */  
    func_end(1);  
}
```

Starting procedure: `func_begin`

Ending procedure: `func_end`

```
void func() {  
    func_begin(2);  
    /* Critical section code */  
    func_end(2);  
}
```

Command-Line Information

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

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

Default: None

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

See Also

[“Multitasking \(C/C++\)”](#) | [“Entry points \(C/C++\)”](#) | [“Temporally exclusive tasks \(C/C++\)”](#)

Related Examples

- [“Specify Analysis Options”](#)
- [“Prevent Concurrent Access Using Critical Sections”](#)

More About


- [“Requirements for Multitasking Verification”](#)

Temporally exclusive tasks (C/C++)

Specify functions that cannot execute concurrently. The execution of the functions cannot overlap with each other. Use this option to implement temporal exclusion in multitasking code. This option is available on the **Configuration** pane under the **Multitasking** node.

Settings

Default: None

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.

Dependencies

This option is enabled only if you select the **Multitasking** box.

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`

Value: Name of temporal exclusions file

Default: None

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

See Also

[“Multitasking \(C/C++\)”](#) | [“Entry points \(C/C++\)”](#) | [“Critical section details \(C/C++\)”](#)

Related Examples

- [“Specify Analysis Options”](#)
- [“Prevent Concurrent Access Using Temporally Exclusive Tasks”](#)

More About

- “Requirements for Multitasking Verification”

Check MISRA C:2004

Specify whether to check for violation of MISRA C[®]:2004 rules. Each value of the option corresponds to a subset of rules to check. This option is available on the **Configuration** pane under the **Coding Rules** node.

After analysis, the **Results Summary** 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:2004)”.

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: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
```

Tips

To reduce unproven results:

- 1 Find coding rule violations in SQ0-subset1. Fix your code to address the violations and rerun verification.
- 2 Find coding rule violations in SQ0-subset2. Fix your code to address the violations and rerun verification.

Command-Line Information

Parameter: -misra2

Value: required-rules | all-rules | SQ0-subset1 | SQ0-subset2 | *file*

Default: required-rules

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

See Also

“Files and folders to ignore (C)”

Related Examples

- “Specify Analysis Options”
- “Activate Coding Rules Checker”
- “Select Specific MISRA or JSF Coding Rules”

More About

- “Polyspace MISRA C 2004 and MISRA AC AGC Checkers”
- “Software Quality Objective Subsets (C:2004)”

Check MISRA AC AGC

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. This option is available on the **Configuration** pane under the **Coding Rules** node.

After analysis, the **Results Summary** pane lists the coding rule violations. On the **Source** pane, for every coding rule violation, 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.

SQ0 -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)”.



SQ0 -subset2

Check a subset of rules including **SQ0 -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
```

Tips

To reduce unproven results:

- 1 Find coding rule violations in SQ0-subset1. Fix your code to address the violations and rerun verification.
- 2 Find coding rule violations in SQ0-subset2. Fix your code to address the violations and rerun verification.

Command-Line Information

Parameter: -misra-ac-agc

Value: OBL-rules | OBL-REC-rules | all-rules | SQ0-subset1 | SQ0-subset2 | *file*

Default: OBL-rules

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

Related Examples

- “Specify Analysis Options”
- “Activate Coding Rules Checker”
- “Select Specific MISRA or JSF Coding Rules”

More About

- “Polyspace MISRA C 2004 and MISRA AC AGC Checkers”
- “MISRA C:2004 Coding Rules”
- “Software Quality Objective Subsets (AC AGC)”

Check MISRA C:2012

Specify whether to check for violations of MISRA C:2012 guidelines. Each value of the option corresponds to a subset of guidelines to check. This option is available on the **Configuration** pane under the **Coding Rules** node.

After analysis, the **Results Summary** 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 mandatory and required guidelines.

mandatory

Check mandatory guidelines.

all

Check mandatory, required, and advisory guidelines.


SQ0-subset1



Check 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)”.

SQ0-subset2

Check a subset of guidelines, **SQ0-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
```

Tips

To reduce unproven results:

- 1 Find coding rule violations in SQ0-subset1. Fix your code to address the violations. Rerun verification.
- 2 Find coding rule violations in SQ0-subset2. Fix your code to address the violations. Rerun verification.

Command-Line Information

Parameter: -misra3

Value: mandatory | mandatory-required | all | SQ0-subset1 | SQ0-subset2 | *file*

Default: mandatory-required

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

See Also

“Files and folders to ignore (C)”

Related Examples

- “Specify Analysis Options”
- “Activate Coding Rules Checker”
- “Select Specific MISRA or JSF Coding Rules”

More About

- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

Use generated code requirements (C)

Specify whether to use the MISRA C:2012 categories for automatically generated code. This option changes which rules are mandatory, required, or advisory. This option is available on the **Configuration** pane under the **Coding Rules** node.

Settings

Default: Off (On for analyses started from the Simulink® plug-in.)

Off

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

On

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** 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

“Files and folders to ignore (C)” | “Check MISRA C:2012” on page 1-40

Related Examples

- “Specify Analysis Options”
- “Activate Coding Rules Checker”

More About

- “Polyspace MISRA C:2012 Checker”

Check custom rules (C/C++)

Define naming conventions for identifiers and check your code against them. This option is available on the **Configuration** pane under the **Coding Rules** node.

After analysis, the **Results Summary** 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

Default: Off

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 **Check 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_`.

- 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

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"`

Related Examples

- “Specify Analysis Options”
- “Activate Coding Rules Checker”
- “Create Custom Coding Rules”

More About

- “Format of Custom Coding Rules File”
- “Custom Naming Convention Rules”

Files and folders to ignore (C)

Specify files and folders to ignore during coding rules checking. This option is available on the **Configuration** pane under the **Inputs & Stubbing** node. The files and folders are **not** ignored during Code Prover verification.

Settings

Default: all-headers


all-headers

Ignore included .h files

all

Ignore all files in include folders

custom

Ignore include files and folders that you specify in the **File/Folder** view. To add files to the custom **File/Folder** list, select  to choose the files and folders to exclude. To remove a file or folder from the list of excluded files and folders, select the row.

Then click .

Dependencies

This option is enabled only if you select one of the options **Check MISRA C:2004**, **Check MISRA C:2012**, **Check MISRA AC AGC** or **Check custom rules**.

Command-Line Information

Parameter: -includes-to-ignore

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

Default: all-headers

Example: polyspace-code-prover-nodesktop -lang c -sources *file_name* -misra2 required-rules -includes-to-ignore "C:\usr\include"

See Also

“Check MISRA C:2004” | “Check MISRA C:2012” | “Check MISRA AC AGC” | “Check custom rules (C/C++)”

Related Examples

- “Specify Analysis Options”
- “Activate Coding Rules Checker”
- “Exclude Files from Rules Checking”

Effective boolean types (C)


Specify data types that you want Polyspace to treat as Boolean. You can specify a data type only if you have defined it through a `typedef` statement in your source code. This option is available on the **Configuration** pane under the **Coding Rules** node.

Use this option to allow Polyspace to check the following MISRA C or MISRA® AC AGC rules:

- 12.6: Operands of logical operators, `&&`, `||`, and `!`, 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 `switch` expression should not represent a value that is effectively Boolean.

Settings

Default: None

Click  to add a field. Enter a type name that you want Polyspace to treat as Boolean.

Dependencies

This option is enabled only if you select one of the options **Check MISRA C:2004**, **Check MISRA AC AGC** or **Check MISRA C:2012**.

Command-Line Information

Parameter: `-boolean-types`

Value: `type1[, type2[, ...]]`

Default: None

Example: `polyspace-code-prover-nodesktop -sources filename -misra2 required-rules -boolean-types boolean1_t,boolean2_t`

See Also

“Check MISRA C:2004” | “Check MISRA AC AGC”

Related Examples

- “Activate Coding Rules Checker”
- “Specify Boolean Types”

More About


- “MISRA C:2004 Coding Rules”

Allowed pragmas (C)

Specify pragma directives for which MISRA C rule 3.4 should not be applied. MISRA C or MISRA AC AGC rule 3.4 requires checking that all pragma directives are documented within the documentation of the compiler. This option is available on the **Configuration** pane under the **Coding Rules** node.

Settings

Default: None

Click  to add a field. Enter the pragma name that you want Polyspace to ignore during MISRA C checking .

Dependencies

This option is enabled only if you select one of the options **Check MISRA C:2004** or **Check MISRA AC AGC**.

Command-Line Information

Parameter: -allowed-pragmas

Value: *pragma1*[,*pragma2*[,*...*]]

Default: None

Example: polyspace-code-prover-nodesktop -sources *filename* -misra2
required-rules -allowed-pragmas pragma_01,pragma_02

See Also

“Check MISRA C:2004” | “Check MISRA AC AGC”

Related Examples

- “Activate Coding Rules Checker”

More About

- “MISRA C:2004 Coding Rules”

Verify whole application (C/C++)

Specify that Polyspace verification must stop if a `main` function is not present in the source files. This option is available on the **Configuration** pane under the **Code Prover Verification** node.

Settings

Default: Off

On

Polyspace verification stops if it does not find a `main` function in the source files.

Off

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

Parameter: `-main`

Default: On

See Also

“Verify module (C)” | “Verify module (C++)”

Related Examples

- “Specify Analysis Options”

More About

- “Main Generator Overview”

Verify module (C)

Specify that Polyspace must generate a `main` function if it does not find one in the source files. This option is available on the **Configuration** pane under the **Code Prover Verification** node.

Settings

Default: On

On

Polyspace generates a `main` function if it does not find one in the source files. The generated `main`:

- Initializes variables that you specify using **Variables to initialize**.
- Calls functions that you specify using **Initialization functions** ahead of other functions.
- Calls functions that you specify using **Functions to call** in arbitrary order.

If you do not specify the above options explicitly, the generated `main`:

- Initializes all global variables except those declared with keywords `const` and `static`.
- Calls in arbitrary order 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 verification if a `main` function is not present in the source files.

Command-Line Information

Parameter: `-main-generator`

Default: Off

See Also

“Verify whole application (C/C++)” | “Variables to initialize (C)” | “Initialization functions (C)” | “Functions to call (C)”

Related Examples

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

More About

- “Main Generator Overview”

Variables to initialize (C)

Specify global variables that you want the generated `main` to initialize. Despite the initialization, Polyspace considers these variables to have any value allowed by their type. This option is available on the **Configuration** pane under the **Code Prover Verification** node.

Settings

Default: `public`

`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

This option is enabled only if you select **Code Prover Verification > Verify module**.

Command-Line Information

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

Value: `none` | `public` | `all` | `custom=variable1[,variable2[,...]]`

Default: `public`

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -main-generator-writes-variables all`

See Also

“Verify module (C)” | “Initialization functions (C)” | “Functions to call (C)”

Related Examples

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

More About

- “Main Generator Overview”

Initialization functions (C)

Specify functions that you want the generated `main` to call ahead of other functions. This option is available on the **Configuration** pane under the **Code Prover Verification** node.

Settings

Default: None

Click  to add a field. Enter the name of a 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[,...]]`

Default: None

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -functions-called-before-main myfunc`

Dependencies

This option is enabled only if you select **Code Prover Verification > Verify module**.

See Also

“Verify module (C)” | “Variables to initialize (C)” | “Functions to call (C)”

Related Examples

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

More About

- “Main Generator Overview”

Functions to call (C)

Specify the functions that you want the generated `main` to call. The `main` calls these functions after the ones you specify through the **Initialization functions** option. This option is available on the **Configuration** pane under the **Code Prover Verification** node.

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. Click  to add a field. Enter the name of a function.

Dependencies

This option is enabled only if you select **Code Prover Verification > Verify module**.

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 (C)” | “Variables to initialize (C)” | “Initialization functions (C)”

Related Examples

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

More About

- “Main Generator Overview”

Verify files independently (C/C++)

Specify that a separate verification job will be created for each source file. Each file is compiled, sent to the remote verification server, and verified individually. Verification results can be viewed for the entire project or for individual files. This option is available on the **Configuration** pane under the **Code Prover Verification** node.

Settings

Default: Off

On

Polyspace creates a separate verification job for each source file.

Off

Polyspace creates a single verification job for all source files in a module.

Dependencies

This option is enabled only if you select **Code Prover Verification > Verify module** on the **Configuration** pane.

Tips

- If you perform a file by file verification, you cannot specify multitasking options.

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 (C/C++)”

Related Examples

- “Specify Analysis Options”

- “Run File-by-File Verification”
- “Run File-by-File Batch Verification”



Common source files (C/C++)

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. This option is available on the **Configuration** pane under the **Code Prover Verification** node.

For instance, if multiple source files call the same function, use this option to specify the file that contains the function definition. Otherwise, Polyspace stubs functions that are called but not defined in the source files.

Settings

Default: None

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**.

Command-Line Information

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

Value: `file1[,file2[,...]]`

Default: None

Example: `polyspace-code-prover-nodesktop -sources file_name -unit-by-unit -unit-by-unit-common-source definitions.c`

See Also

“Verify files independently (C/C++)”

Related Examples

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

Parameters (C)

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. This option is available on the **Configuration** pane under the **Code Prover Verification** node.

Settings

Default: `public`

`none`

The generated `main` does not initialize variables.


`public`

The generated `main` initializes all variables except those declared with keywords `static` and `const`.

`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.

Command-Line Information

Parameter: `-variables-written-before-loop`

Value: `none` | `public` | `all` | `custom=variable1[,variable2[,...]]`

Default: `public`

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -variables-written-before-loop all`

See Also

“Inputs (C)” | “Initialization functions (C)” | “Step functions (C)” | “Termination functions (C)”

Related Examples

- “Specify Analysis Options”

- “Configure Polyspace Analysis Options”

More About

- “Recommended Polyspace options for Verifying Generated Code”
- “Main Generation for Model Verification”

Inputs (C)

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 any value allowed by their type. This option is available on the **Configuration** pane under the **Code Prover Verification** node.

Settings

Default: public

none

The generated `main` does not initialize variables.


public

The generated `main` initializes all variables except those declared with keywords `static` and `const`.

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.

Command-Line Information

Parameter: `-variables-written-in-loop`

Value: `none` | `public` | `all` | `custom=variable1[,variable2[,...]]`

Default: `public`

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -variables-written-in-loop all`

See Also

“Parameters (C)” | “Initialization functions (C)” | “Step functions (C)” | “Termination functions (C)”

Related Examples

- “Specify Analysis Options”
- “Configure Polyspace Analysis Options”

More About

- “Recommended Polyspace options for Verifying Generated Code”
- “Main Generation for Model Verification”

Initialization functions (C)

This option is available only for model-generated code. Specify functions that the generated `main` must call before the cyclic code begins. This option is available on the **Configuration** pane under the **Code Prover Verification** node.

Settings

Default: None

Click  to add a field. Enter function name.

Command-Line Information

Parameter: `-functions-called-before-loop`

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

Default: None

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -functions-called-before-loop myfunc`

See Also

“Parameters (C)” | “Inputs (C)” | “Step functions (C)” | “Termination functions (C)”

Related Examples

- “Specify Analysis Options”
- “Configure Polyspace Analysis Options”

More About

- “Recommended Polyspace options for Verifying Generated Code”
- “Main Generation for Model Verification”

Step functions (C)

This option is available only for model-generated code. Specify functions that the generated `main` must call in each cycle of the cyclic code. This option is available on the **Configuration** pane under the **Code Prover Verification** node.

Settings

Default: unused

none

The generated `main` does not call functions in the cyclic code.


unused

The generated `main` calls all functions that are not called elsewhere in the code. In particular, 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. It also does not call inlined functions.

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.

Tips

- When you select **unused**, the generated `main` does not call a function if it is called elsewhere. However, this rule does not apply to calls through function pointers. The generated `main` calls a function even when it is called elsewhere through a function pointer.
- 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 | unused | all | custom=*function1*[,*function2*[,...]]

Default: unused

Example: polyspace-code-prover-nodesktop -sources *file_name* -main-generator -functions-called-in-loop all

See Also

“Parameters (C)” | “Inputs (C)” | “Initialization functions (C)” | “Termination functions (C)”

Related Examples

- “Specify Analysis Options”
- “Configure Polyspace Analysis Options”

More About

- “Recommended Polyspace options for Verifying Generated Code”
- “Main Generation for Model Verification”

Termination functions (C)

This option is available only for model-generated code. Specify functions that the generated `main` must call after the cyclic code ends. This option is available on the **Configuration** pane under the **Code Prover Verification** node.

Settings

Default: None

Click  to add a field. Enter function name.

Command-Line Information

Parameter: `-functions-called-after-loop`

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

Default: None

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -functions-called-after-loop myfunc`

See Also

“Parameters (C)” | “Inputs (C)” | “Initialization functions (C)” | “Step functions (C)”

Related Examples

- “Specify Analysis Options”
- “Configure Polyspace Analysis Options”

More About


- “Recommended Polyspace options for Verifying Generated Code”
- “Main Generation for Model Verification”

Variable/function range setup (C/C++)

Specify range for global variables or function outputs using a **Data Range Specifications** template file. The template file can be either a text or an XML file. This option is available on the **Configuration** pane under the **Inputs & Stubbing** node.

Settings

Default: None

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

Command-Line Information

Parameter: -data-range-specifications

Value: *file*

Default: None

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

See Also

“Functions to stub (C)” on page 1-78 | “Ignore default initialization of global variables (C)”

Related Examples

- “Specify Analysis Options”
- “Create Data Range Specification Template”
- “Specify Data Ranges Using Existing Template”
- “Specify Data Ranges Using Text Files”

More About

- “Data Range Specifications”
- “DRS Configuration Settings”

- “Variable Scope”
- “XML Format of DRS File”

Ignore default initialization of global variables (C)

Specify that Polyspace must not treat global variables as initialized. This option is available on the **Configuration** pane under the **Inputs & Stubbing** node.

Settings

Default: Off

On

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

Off

Polyspace considers global variables to be initialized according to ANSI 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**.
- If you initialize a global variables using the generated `main`, 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 (C)”.

Command-Line Information

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

Default: Off

See Also

“Non-initialized variable”

Related Examples

- “Specify Analysis Options”

No automatic stubbing (C/C++)

Specify that verification must stop if a function is not defined in the source files. This option is available on the **Configuration** pane under the **Inputs & Stubbing** node.

Settings

Default: Off

On

Polyspace displays a list of undefined functions and stops verification.

Off

Polyspace stubs all undefined functions.

Tips

Use this option when:

- The code you are verifying must be complete. This option allows you to find functions that are not defined in your source.
- You prefer to stub undefined functions manually.

Command-Line Information

Parameter: `-no-automatic-stubbing`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources filename -no-automatic-stubbing`

See Also

“Functions to stub (C)” | “Functions to stub (C++)” | “No STL stubs (C++)”

Related Examples

- “Specify Analysis Options”
- “Specify Functions to Stub Automatically”
- “Constrain Data with Stubbing”

More About

- “Stubbing Overview”
- “When to Provide Function Stubs”
- “Stubbing Examples”

Functions to stub (C)

Specify functions that you want the software to stub. This option is available on the **Configuration** pane under the **Inputs & Stubbing** node.

Settings

Default: None

Click  to add a field. Enter function name.

Command-Line Information

Parameter: -functions-to-stub

Default: None

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

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

See Also

“No automatic stubbing (C/C++)” | “Variable/function range setup (C/C++)” on page 1-72
| “Functions to stub (C++)”

Related Examples

- “Specify Analysis Options”
- “Specify Functions to Stub Automatically”
- “Constrain Data with Stubbing”

More About

- “Stubbing Overview”
- “When to Provide Function Stubs”
- “Stubbing Examples”

Respect types in fields (C/C++)

Specify that structure fields not declared initially as pointers will not be cast to pointers later. This option is available on the **Configuration** pane under the **Verification Assumptions** node.

Settings

Default: Off

On

The verification assumes that structure fields not declared initially as pointers will not be cast to pointers later.

For instance, in the following code, the structure field `S1.x1` is not declared as a pointer. However, it is cast to a pointer and used to point to `y`. If you select this option, the line `assert(y==0);` causes a green **User assertion** check even though `y` is assigned a value of 1 through `S1.x1`.

```
struct {
    unsigned x1;
    unsigned x2;
} S1;

void funct2(void) {
    int *tmp;
    int y;
    ((int*)&S1)[0] = &y; /* S1.x1 points to y */
    tmp = (int*)S1.x1
    y=0;
    *tmp = 1; /* Write 1 to y */
    assert(y==0);
}
```

Off

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 (C/C++)”

Related Examples

- “Specify Analysis Options”

Respect types in global variables (C/C++)

Specify that global variables not declared initially as pointers will not be cast to pointers later. This option is available on the **Configuration** pane under the **Verification Assumptions** node.

Settings

Default: Off

On

The verification assumes that global variables not declared initially as pointers will not be cast to pointers later.

For instance, in the following code, the variable `x` is not declared as a pointer. However, it is cast to a pointer and used to point to `y`. If you select this option, the line `assert (y==0)`; causes a green check even though `y` is assigned a value of 1 through `x`.

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

Off

The verification assumes that global variables can be cast to pointers even when they are not declared as pointers.

Command-Line Information

Parameter: `-respect-types-in-globals`

Default: Off

See Also

“Respect types in fields (C/C++)”

Related Examples

- “Specify Analysis Options”

Ignore float rounding (C/C++)

Specify that operations involving `float` and `double` variables do not involve rounding. This option is available on the **Configuration** pane under the **Verification Assumptions** node.

Settings

Default: Off

On

The verification considers that operations involving `float` and `double` variables do not involve rounding.

Off

The verification assumes that results of operations involving `float` and `double` are rounded to the nearest value according to the IEEE 754 standard:

- Simple precision on 32-bit targets
- Double precision on 64-bit targets

Command-Line Information

Parameter: `-ignore-float-rounding`

Default: Off

Related Examples

- “Specify Analysis Options”

Green absolute address checks (C/C++)

Specify that absolute addresses in your code are valid addresses. This option is available on the **Configuration** pane under the **Verification Assumptions** node.

Settings

Default: Off

On

The verification assumes that the absolute addresses in your code are valid.

Off

The verification generates an orange **Absolute Address** check when an absolute address is assigned to a pointer. The orange check occurs because the software does not have information about the absolute address and cannot verify, for example, the validity of the address and the availability of memory.

Tips

Even if you use this option, you cannot assign an absolute address to a pointer and perform pointer arithmetic using the pointer. As soon as you perform pointer arithmetic, Polyspace cannot verify the validity of the next dereference using this pointer

Command-Line Information

Parameter: `-green-absolute-address-checks`

Default: Off

See Also

“Absolute address”

Related Examples

- “Specify Analysis Options”

Ignore overflowing computations on constants (C/C++)

Specify that the verification must allow overflow in computations involving constants. 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;` This option is available on the **Configuration** pane under the **Check Behavior** node.

Settings

Default: Off

On

The verification allows overflows in computations involving constants.

Off

If an overflow occurs in computations involving constants, the verification generates an **Overflow** error.

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”

Related Examples

- “Specify Analysis Options”

Allow negative operand for left shifts (C/C++)

Specify that the verification must allow shift operations on a negative number. Unless you use this option, following ANSI C standard, the verification generates an error for the shift operations. This option is available on the **Configuration** pane under the **Check Behavior** node.

Settings

Default: Off

On

The verification allows shift operations on a negative number, for instance, `-2 << 2`.

Off

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

“Shift operations”

Related Examples

- “Specify Analysis Options”

Detect overflows (C/C++)

Specify integer overflows to check for. This option is available on the **Configuration** pane under the **Check Behavior** node.

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” | “Overflow computation mode (C/C++)”

Related Examples

- “Specify Analysis Options”
- “Detect Overflows in Buffer Size Computation”

Detect Overflows in Buffer Size Computation

If you are computing the size of a buffer from **unsigned** integers, for the **Detect overflows** option, use **signed-and-unsigned**. Using this option helps you detect an overflow at the buffer computation stage. Otherwise, you might see an error later due to insufficient buffer. This option is available on the **Configuration** pane under the **Check Behavior** node.

For this example, save the following C code in a file `display.c`:

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

int get_value(void);

void display(unsigned int num_items) {
    int *array;
    array = (int *) malloc(num_items * sizeof(int)); // overflow error
    if (array) {
        for (unsigned int ctr = 0; ctr < num_items; ctr++) {
            array[ctr] = get_value();
        }
        for (unsigned int ctr = 0; ctr < num_items; ctr++) {
            printf("Value is %d.\n", ctr, array[ctr]);
        }
        free(array);
    }
}

void main() {
    display(33000);
}
```

- 1 Create a Polyspace project and add `display.c` to the project.
- 2 On the **Configuration** pane, select the following options:
 - **Target & Compiler:** From the **Target processor type** drop-down list, select a type with 16-bit `int` such as `c167`.
 - **Check Behavior:** From the **Detect overflows** drop-down list, select **signed**.
- 3 Run the verification and open the results.

Polyspace detects an orange **Illegally dereferenced pointer** error on the line `array[ctr] = get_value()` and a red **Non-terminating loop** error on the `for` loop.

This error follows from an earlier error. For a 16-bit `int`, there is an overflow on the computation `num_items * sizeof(int)`. Polyspace does not detect the overflow because it occurs in computation with `unsigned` integers. Instead Polyspace wraps the result of the computation causing the **Illegally dereferenced pointer** error later.

- 4 From the **Detect overflows** drop-down list, select **signed-and-unsigned**.
- 5 Polyspace detects a red **Overflow** error in the computation `num_items * sizeof(int)`.

See Also

“Detect overflows (C/C++)” | “Overflow” | “Illegally dereferenced pointer”

Overflow computation mode (C/C++)

Specify whether Polyspace must wrap the result of an integer overflow or restrict it to its extremum value. This option is available on the **Configuration** pane under the **Check Behavior** node.

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

“Overflow”

Related Examples

- “Specify Analysis Options”

Enable pointer arithmetic across fields (C)

Specify that a pointer assigned to a structure field can point outside its bounds as long as it points within the structure. This option is available on the **Configuration** pane under the **Check Behavior** node.

Settings

Default: Off

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

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**.

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

“Illegally dereferenced pointer”

Related Examples

- “Specify Analysis Options”

Allow incomplete or partial allocation of structures (C)

Specify that the verification must allow partial allocation of memory for structures. This option is available on the **Configuration** pane under the **Check Behavior** node.

Settings

Default: Off

On

The verification must allow partial allocation of memory for structures. Such partial allocation can occur during type-casting from a smaller type.

For instance, using this option, in the following code, the verification must produce a red **Illegally dereferenced pointer** check only on the third assignment in the `if` statement.

```
#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

The verification must require complete allocation of memory for structures.

For instance, without the option, in the above code, the verification must produce a red **Illegally dereferenced pointer** check on the first assignment in the `if` statement.

Tips

The verification also allows partial allocation of structures when you select **Enable pointer arithmetic across fields** or **Precision > Retype variables of pointer types**.

Command-Line Information

Parameter: `-size-in-bytes`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -size-in-bytes`

See Also

“Illegally dereferenced pointer”

Related Examples

- “Specify Analysis Options”

Permissive function pointer calls (C)

Specify that the verification must allow function pointer calls where the type of the function pointer does not match the type of the function. This option is available on the **Configuration** pane under the **Check Behavior** node.

Settings

Default: Off

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

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.

Command-Line Information

Parameter: `-permissive-function-pointer`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -permissive-function-pointer`

Related Examples

- “Specify Analysis Options”

Detect uncalled functions (C/C++)

Detect functions that are not called directly or indirectly from `main` or another entry point during run-time. This option is available on the **Configuration** pane under the **Check Behavior** node.

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”

Related Examples

- “Specify Analysis Options”

Precision level (C/C++)

Specify the precision level that the verification must use. Higher precision leads to greater number of proven results but also requires more verification time. Each precision level corresponds to a different algorithm used for verification. This option is available on the **Configuration** pane under the **Precision** node.

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: -00 | -01 | -02 | -03

Default: -02

Example: `polyspace-code-prover-nodesktop -sources file_name -01`

See Also

“Verification level (C)”

Related Examples

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

Verification level (C)

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. This option is available on the **Configuration** pane under the **Precision** node.

Settings

Default: Software Safety Analysis level 2

C 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 thrice 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 level1 (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; level1(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 level1 (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; level1(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.

- 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.

Dependency

You cannot use the C Source Compliance Checking setting with batch or interactive mode.

Command-Line Information

Parameter: -to

Value: c-compile | pass0 | pass1 | pass2 | pass3 | pass4 | other

Default: pass2

Example: polyspace-code-prover-nodesktop -sources *file_name* -to pass2

Related Examples

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

Verification time limit (C/C++)

Specify a time limit for the verification in hours. If the verification does not complete within that limit, it stops. This option is available on the **Configuration** pane under the **Precision** node.

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`

Related Examples

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

Retype variables of pointer types (C)

Specify that the verification must allow pointers to be cast from one type to another. If you select this option, the verification replaces the original type of the pointer by its new type. This option is available on the **Configuration** pane under the **Precision** node.

Settings

Default: Off

On

The verification allows pointers to be cast from one type to another. It replaces the original type of the pointer by its new type. For instance, using this option, the software produces a green check on the `assert` statement in the following code:

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

Off

The verification retains the declaration type of a pointer even when it is recast.

Command-Line Information

Parameter: `-retype-pointer`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -retype-pointer`

See Also

“Enable pointer arithmetic across fields (C)” | “Allow incomplete or partial allocation of structures (C)”

Related Examples

- “Specify Analysis Options”

Retype symbols of integer types (C)

Specify that the verification must allow integers to be cast to pointers. This option is available on the **Configuration** pane under the **Precision** node.

Settings

Default: Off

On

The verification allows integers to be cast to pointers. For instance, using this option, the software can prove the `assert` statements in the following code:

```
void function(void)
{
  struct S1 {
    int x;
    int y;
    int z;
    char t;
  } s1 = {1,2,3,4};
  int addr;
  addr = (int)&s1;
  assert(((struct S1 *)addr)->y == 2);
}
```

Off

The verification does not allow integers to be cast to pointers.

Dependencies

This option:

- Automatically enables **Check Behavior > Allow incomplete or partial allocation of structures**.
- Has no effect on global integers if you select the option **Verification Assumptions > Respect types in global variables**.
- Has no effect on integers that are structure fields if you select the option **Verification Assumptions > Respect types in fields**.

Tips

- Use this option for:
 - Code with memory mapping
 - Code close to the communication layer API – When your code contains low level drivers, it tends to perform generic pointer casts using `(void *)`.
- If you set this option:
 - Some of the `Illegally dereferenced pointer` checks can change
 - Some of the `Non-initialized variable` checks can change to `Non-initialized pointer` checks.

Command-Line Information

Parameter: `-retype-int-pointer`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -retype-int-pointer`

See Also

“Illegally dereferenced pointer”

Related Examples

- “Specify Analysis Options”

Sensitivity context (C/C++)

Specify that the software must store call context information during verification. If a function contains a red and green check in the same line for two different invocations, both checks will be displayed. This option is available on the **Configuration** pane under the **Precision** node.

Settings

Default: none

none


The software does not store call context information for functions.

auto

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

- 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. Click  to enter the name of a function.

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 decide which functions receive call context storage, use the option `-context-sensitivity-auto`.

Related Examples

- “Specify Analysis Options”

- “Identify Function Call Causing Orange Check”
- “Improve Verification Precision”

Improve precision of interprocedural analysis (C/C++)

Use this option to propagate greater information about function arguments into the called function. This option is available on the **Configuration** pane under the **Precision** node.

Settings

Default: Off

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

Tips

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

Command-Line Information

Parameter: `-path-sensitivity-delta`

Value: Positive integer

Example: `polyspace-code-prover-nodesktop -sources file_name -path-sensitivity-delta 1`

Related Examples


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

Specific precision (C)

Specify source files that you want to verify at a **Precision level** higher than that for the entire verification. This option is available on the **Configuration** pane under the **Precision** node.

Settings

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

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

Command-Line Information

Parameter: `-modules-precision`

Value: `file:00 | file:01 | file:02 | file:03`

Example: `polyspace-code-prover-nodesktop -sources file_name -01 -modules-precision My_File.c:02`

See Also

“Precision level (C/C++)”

Related Examples

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

Optimize large static initializers (C)

Specify that the verification must approximate statically initialized `int`, `float` and `char` arrays if required. If you do not specify this option, for static initialization of large arrays, scaling problems can occur during the compilation phase. This option is available on the **Configuration** pane under the **Scaling** node.

Settings

Default: Off

On

The verification approximates statically initialized `int`, `float` and `char` arrays if required. Using this option can speed up verification, but can decrease precision for some applications.

Off

The verification does not approximate statically initialized `int`, `float` and `char` arrays.

Command-Line Information

Parameter: `-no-fold`

Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -no-fold`

Related Examples

- “Specify Analysis Options”

Reduce task complexity (C)

Specify that the verification must use a slightly less precise model than default for interaction between tasks. This option is available on the **Configuration** pane under the **Scaling** node.

Settings

Default: Off

On

The verification uses a slightly less precise model than default for interaction between tasks. Using this option, you can speed up verification, but have greater number of unproven results. There is also a loss of precision when variables shared between tasks are read through pointers.

Off

The verification uses the default model for interaction between tasks.

Command-Line Information

Parameter: `-lightweight-thread-model`

Default: Off

See Also

“Entry points (C/C++)”

Related Examples


- “Specify Analysis Options”
- “Reduce Verification Time”

Inline (C)

Specify the functions that the verification must clone for every function 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. This option is available on the **Configuration** pane under the **Scaling** node.

Settings

Default: No function is inlined.

Click  to enter function name.

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**.

Command-Line Information

Parameter: `-inline`

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

Default: None

Example: `polyspace-code-prover-nodesktop -sources file_name -inline func1,func2`

Related Examples

- “Specify Analysis Options”
- “Reduce Procedure Complexity”

Depth of verification inside structures (C/C++)

Specify a limit to the depth of analysis for nested structures. This option is available on the **Configuration** pane under the **Scaling** node.

Settings

Default: Full depth of nested structures is analysed.

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.

Command-Line Information

Parameter: `-k-limiting`

Value: *positive integer*

Default: `polyspace-code-prover-nodesktop -sources file_name -k-limiting 1`

Related Examples

- “Specify Analysis Options”

Generate report (C/C++)

Specify whether to generate a report after the analysis. This option is available on the **Configuration** pane under the **Reporting** node.

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

Settings

Default: Off

On

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

Off

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

Tips

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

Command-Line Information

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

See Also

“Report template (C/C++)” | “Output format (C/C++)”

Related Examples

- “Specify Analysis Options”

- “Generate Report from User Interface”
- “Generate Report from Command Line”
- “Open Report”

Report template (C/C++)

Specify template for generating analysis report. This option is available on the **Configuration** pane under the **Reporting** node.

.rpt files for the report templates are available in the folder `MATLAB_Install\polyspace\toolbox\psrptgen\templates\`.

Settings

Default: Developer

CodeMetrics

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

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.

Developer

The report lists information useful to developers, including:

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

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

DeveloperReview

The report lists the same information as the **Developer** report. However, the reviewed results are sorted by review classification and status, and unreviewed results are sorted by file location.

Developer_withGreenChecks

The report lists the same information as the **Developer** report. In addition, the report lists code proven to be error-free or green checks.

Quality

The report lists information useful to quality engineers, including:

- Summary of results
- Statistics about the code
- Graphs showing distributions of checks per file

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

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.

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.

Dependencies

- This option is enabled only if you select the **Generate report** box.
- The templates `SoftwareQualityObjectives` and `SoftwareQualityObjectives_Summary` are available only if you generate a report from results downloaded from the web interface. To generate these reports:
 - 1 Download results from the Polyspace Metrics interface.
 - 2 In the Results Manager perspective, select **Reporting > Run Report**.
 - 3 Select the template that you want.

Command-Line Information

Parameter: `-report-template`

Value: `template.rpt`

Default: `Developer.rpt`

Example: `polyspace-code-prover-nodesktop -sources file_name -report-template Developer.rpt`

See Also

“Generate report (C/C++)” | “Output format (C/C++)”

Related Examples

- “Specify Analysis Options”
- “Generate Report from User Interface”
- “Generate Report from Command Line”
- “Open Report”
- “Customize Report Templates”

Output format (C/C++)

Specify output format of generated report. This option is available on the **Configuration** pane under the **Reporting** node.

Settings

Default: RTF

RTF

Generate report in `.rtf` format

HTML

Generate report in `.html` format

PDF

Generate report in `.pdf` format

Word

Generate report in `.doc` format. Not available on UNIX[®] platforms.

XML

Generate report in `.xml` format.

Tips

- You must have Microsoft Office installed to view `.rtf` format reports containing graphics, such as the **Quality** report.
- If the table of contents or graphics in a `.doc` 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: RTF | HTML | PDF | Word | XML

Default: RTF

Example: `polyspace-code-prover-nodesktop -sources file_name -report-output-format pdf`

See Also

“Generate report (C/C++)” | “Report template (C/C++)”

Related Examples

- “Specify Analysis Options”
- “Generate Report from User Interface”
- “Generate Report from Command Line”
- “Open Report”
- “Customize Report Templates”

Batch (C/C++)

Enable or disable batch remote analysis. This option is available on the **Configuration** pane under the **Distributed Computing** node.

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

Settings

Default: Off

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 “Manage Remote Analyses 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

Do not run batch analysis on a remote computer.

Dependency

You cannot use **Batch** mode with the **Verification Level** options **C** source compliance checking or **C++** source compliance checking.

Command-Line Information

To run a remote verification 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`

See Also

“Add to results repository (C/C++)” on page 1-126 | `-interactive` | `-scheduler`

Related Examples

- “Specify Analysis Options”
- “Set Up Remote Verification and Analysis”

Add to results repository (C/C++)

Specify upload of analysis results to the Polyspace Metrics results repository, allowing Web-based reporting of results and code metrics. This option is available on the **Configuration** pane under the **Distributed Computing** node.

Settings

Default: Off

On

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

Off

Analysis results are stored locally.

Dependency

This option is only available for remote verifications. For local verification, you can manually upload your results to Polyspace Metrics by right-clicking on your results file and selecting **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

See Also


“Set Up Remote Verification and Analysis” | “Set Up Polyspace Metrics” | “Set Up Verification to Generate Metrics” | “Batch (C/C++)” on page 1-124

Command/script to apply after the end of the code verification (C/C++)

Specify a command or script to be executed after the verification. This option is available on the **Configuration** pane under the **Advanced Settings** node.

Settings

Default: None

Enter full path to the command or script, or click  to navigate to the location of the command or script. For example, you can enter the path to a script that sends an email. After the verification, this script will be executed.

Command-Line Information

Parameter: -post-analysis

Value: Path to executable file or command in quotes

Default: None

Example: polyspace-code-prover-nodesktop -sources *file_name* -post-analysis-command `pwd`/send_email

Related Examples

- “Specify Analysis Options”

Automatic Orange Tester (C)

Specify that the Automatic Orange Tester must be executed at the end of the verification. This option is available on the **Configuration** pane under the **Advanced Settings** node.

You must select this option before verification if you want to run the Automatic Orange Tester after verification. During verification, Polyspace generates additional source code that tests each orange check for run-time errors. The software compiles this instrumented code. When you run the Automatic Orange Tester later, the software tests the resulting binary code.

Settings

Default: Off

On

After verification, when you run the Automatic Orange Tester, Polyspace creates tests for unproven code and runs them.

Off

You cannot launch the Automatic Orange Tester after verification.

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 **Variable/function range setup** under **Inputs & Stubbing**.

Command-Line Information

Parameter: -automatic-orange-tester

Default: Off

Example: polyspace-code-prover-nodesktop -sources *file_name* -automatic-orange-tester

See Also

“Number of automatic tests (C)” | “Maximum loop iterations (C)” | “Maximum test time (C)”

Related Examples

- “Specify Analysis Options”
- “Test Orange Checks for Run-Time Errors”

More About

- “Limitations of Automatic Orange Tester”

Number of automatic tests (C)

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. This option is available on the **Configuration** pane under the **Advanced Settings** node.

Settings

Default: 500

Enter number of tests up to a maximum of 100,000.

Dependencies

This option is enabled only if you select the **Automatic Orange Tester** box.

Command-Line Information

Parameter: `-automatic-orange-tester-tests-number`

Value: *positive integer*

Default: 500

Example: `polyspace-code-prover-nodesktop -sources file_name -automatic-orange-tester -automatic-orange-tester-tests-number 500`

See Also

“Automatic Orange Tester (C)” | “Maximum loop iterations (C)” | “Maximum test time (C)”

Related Examples

- “Specify Analysis Options”
- “Test Orange Checks for Run-Time Errors”

Maximum loop iterations (C)

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. This option is available on the **Configuration** pane under the **Advanced Settings** node.

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 select the **Automatic Orange Tester** box.

Command-Line Information

Parameter: `-automatic-orange-tester-loop-max-iteration`

Value: *positive integer*

Default: 1000

Example: `polyspace-code-prover-nodesktop -sources file_name -automatic-orange-tester -automatic-orange-tester-loop-max-iteration 500`

See Also

“Automatic Orange Tester (C)” | “Number of automatic tests (C)” | “Maximum test time (C)”

Related Examples

- “Specify Analysis Options”
- “Test Orange Checks for Run-Time Errors”

Maximum test time (C)

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. This option is available on the **Configuration** pane under the **Advanced Settings** node.

Settings

Default: 5

Enter time in seconds. The maximum value that the software supports is 60.

Dependencies

This option is enabled only if you select the **Automatic Orange Tester** box.

Command-Line Information

Parameter: `-automatic-orange-tester-timeout`

Value: *time*

Default: 5

Example: `polyspace-code-prover-nodesktop -sources file_name -automatic-orange-tester -automatic-orange-tester-test-timeout 10`

See Also

“Automatic Orange Tester (C)” | “Number of automatic tests (C)” | “Maximum loop iterations (C)”

Related Examples

- “Specify Analysis Options”
- “Test Orange Checks for Run-Time Errors”

Other (C)

In this section...
“-extra-flags” on page 1-133
“-c-extra-flags” on page 1-133
“-cfe-extra-flags” on page 1-133
“-il-extra-flags” on page 1-134

Specify special options for C verification, which are provided by MathWorks if required.

-extra-flags

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-code-prover-nodesktop -extra-flags -param1 -extra-flags -  
param2 \  
  
-extra-flags 10 ...
```

-c-extra-flags

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-code-prover-nodesktop -c-extra-flags -param1 -c-extra-  
flags -param2 -c-extra-flags 10
```

-cfe-extra-flags

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-code-prover-nodesktop -cfe-extra-flags -param1 -cfe-extra-flags -param2
```

-il-extra-flags

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-code-prover-nodesktop -il-extra-flags -param1 -il-extra-flags -param2 -il-extra-flags 10
```


Option Descriptions specific to C++ Code

- “Target processor type (C++)” on page 2-3
- “Dialect (C++)” on page 2-5
- “C++11 Extensions (C++)” on page 2-10
- “Block char16/32_t types (C++)” on page 2-11
- “Enum type definition (C++)” on page 2-12
- “Pack alignment value (C++)” on page 2-13
- “Ignore pragma pack directives (C++)” on page 2-14
- “Support managed extensions (C++)” on page 2-15
- “Import folder (C++)” on page 2-16
- “Management of scope of 'for loop' variable index (C++)” on page 2-17
- “Management of wchar_t (C++)” on page 2-18
- “Set wchar_t to unsigned long (C++)” on page 2-19
- “Set size_t to unsigned long (C++)” on page 2-20
- “Ignore link errors (C++)” on page 2-21
- “Check MISRA C++ rules” on page 2-22
- “Check JSF C++ rules” on page 2-24
- “Files and folders to ignore (C++)” on page 2-26
- “Main entry point (C++)” on page 2-28
- “Verify module (C++)” on page 2-30
- “Class (C++)” on page 2-32
- “Functions to call within the specified classes (C++)” on page 2-34
- “Analyze class contents only (C++)” on page 2-37
- “Skip member initialization check (C++)” on page 2-39

- “Functions to call (C++)” on page 2-40
- “Variables to initialize (C++)” on page 2-42
- “Initialization functions (C++)” on page 2-44
- “Parameters (C++)” on page 2-46
- “Inputs (C++)” on page 2-48
- “Initialization functions (C++)” on page 2-50
- “Step functions (C++)” on page 2-51
- “Termination functions (C++)” on page 2-53
- “No STL stubs (C++)” on page 2-54
- “Functions to stub (C++)” on page 2-55
- “Tuning Precision and Scaling Parameters” on page 2-57
- “Verification level (C++)” on page 2-59
- “Inline (C++)” on page 2-61
- “Other (C++)” on page 2-62

Target processor type (C++)

Specify the target processor type. This option is available on the **Configuration** pane under the **Target & Compiler** node.

Specifying the target processor type informs Polyspace of the size of fundamental data types and of the endianness of the target machine. You can analyze code intended for an unlisted processor type using one of the listed processor types, if they share common data properties.

Settings

Default: i386

You can modify some default attributes by selecting the browse button to the right of the **Target processor type** drop-down menu. The optional settings for each target are shown in [brackets] in the table.

Target	char	short	int	long	long long	float	double	long double	ptr	sign of char	endian	align
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 / ColdFire ^a	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
x86_64	8	16	32	64 [32] ^b	64	32	64	128	64	signed	Little	64 [32]
mcpu... (Advanced)	8 [16] [16]	8 [16]	16 [32]	32	32 [64]	32	32 [64]	32 [64]	16 [32]	signed	Little	32 [16, 8]

- a. The M68k family (68000, 68020, etc.) includes the “ColdFire” processor
- b. Use option `-long-is-32bits` to support Microsoft C/C++ Win64 target
- c. `mcpu` is a reconfigurable Micro Controller/Processor Unit target. You can use this type to configure one or more generic targets.

Tips

If your processor is not listed, use a similar processor that shares the same characteristics, or create an `mpcu` generic target processor. If your target processor does not match the characteristics of a processor described above, contact MathWorks technical support for advice.

Command-Line Information

Parameter: `-target`

Value: `i386` | `m68k` | `powerpc` | `c-167` | `x86_64` | `mpcu`

Default: `i386`

Example: `polyspace-code-prover-nodesktop -lang cpp -target powerpc`

See Also

“Generic target options (C/C++)” on page 1-9

Related Examples

- “Specify Analysis Options”
- “Modify Predefined Target Processor Attributes”
- “Define Generic Target Processors”

Dialect (C++)

Allow syntax associated with C++ language extensions. This option is available on the **Configuration** pane under the **Target & Compiler** node.

Settings

Default: none

none

Analysis allows for ISO/IEC 14882:2003 C++ (C++ 2003) syntax.

If you want to allow ISO/IEC 14882:2011 C++ (C++ 2011) syntax, also select **C++ 11 extensions**.

gnu3.4

Analysis allows GCC 3.4 dialect syntax.

gnu4.6

Analysis allows GCC 4.6 dialect syntax.

gnu4.7

Analysis allows GCC 4.7 dialect syntax.

For more information, see “Limitations” on page 2-6.

iso

Analysis allows for ISO/IEC 14882:2003 C++ (C++ 2003) syntax.

If you want to allow ISO/IEC 14882:2011 C++ (C++ 2011) syntax, also select **C++ 11 extensions**.

cfront2

Analysis allows for Cfront 2.0 language extensions.

cfront3

Analysis allows for Cfront 3.0 language extensions.

visual

Analysis allows Visual C++ .NET 2003 syntax.

visual6

Analysis allows Visual C++ 6.0 (VC6) syntax.

visual7.0

Analysis allows Visual C++ .NET 2002 syntax.

`visual7.1`

Analysis allows Visual C++ .NET 2003 syntax.

`visual8`

Analysis allows Visual C++ 2005 syntax.

`visual9.0`

Analysis allows Visual C++ 2008 syntax.

`visual10`

Analysis allows Visual C++ 2010 syntax.

This option automatically adds the option `-no-stl-stubs`.

`visual11.0`

Analysis allows Visual C++ 2012 syntax.

This option automatically adds the option `-no-stl-stubs`.

Dependencies

This parameter is dependent on the value of **Target operating system**. The dialect options work only with the applicable operating systems. You can use every dialect with the **Target operating system** option, `no-predefined-OS`.

If you enable **Check JSF C++ Rules** with a dialect other than `iso` or `none`, Polyspace cannot completely check some JSF coding rules. For example, AV Rule 8: “All code shall conform to ISO/IEC 14882:2002(E) standard C++.”

Limitations

Polyspace does not support certain aspects of the GNU 4.7 dialect. These limitations can cause compilation errors, incomplete results, or false positives.

- **Priority attributes** — Not supported, ignores priorities and uses standard initialization instead.

Example

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

```

struct A{
    int a;
    A():a(1) {
        fprintf(stderr, "A constructor\n");
    }
};

struct B{
    int b;

    B():b(1) {
        fprintf(stderr, "B constructor\n");
    }
};

A a __attribute__((init_priority (100)));
B b __attribute__((init_priority (50)));

```

The expected output from the above code is:

```

B constructor
A constructor

```

However, Polyspace preserves the standard initialization. So the actual output is:

```

A constructor
B constructor

```

Workaround: To use the desired priority, change the order of the declarations to match the desired order.

- **Vector types and attributes** — Not supported.
- **Visibility attributes** — Not supported, ignored. This limitation can cause C++ linkage problems in Polyspace Code Prover.


Workaround: Remove all attributes during preprocessing,

- At the command line, use the option `-D __attribute__(x)=`.
- In the Polyspace environment, in **Macros > Preprocessor definitions**, add a row: `__attribute__(x)=`.
- **Complex types** — Only floating complex types supported, integral complex types cause an error.
- **Using built-in library function on complex types** — Not supported, stubbed during analysis. Calls to these functions will return variables with full ranges.

Workaround: To make the analysis more precise, add an include file that defines the functions for complex variables.


- **Computed goto** — Not supported, causes an error in Code Prover.

Workaround: To ignore the computed gotos in Code Prover, stub the functions containing the computed gotos:

- At the command line, use the option `-functions-to-stub funcList` where *funcList* is the list of functions containing the computed gotos.
- In the Polyspace environment, in the **Inputs & Stubbing > Functions to stub** table, use the  button to add a row for each function containing the computed gotos.
- **Nested functions** — Not supported, causes an error.
- **Using built-in library functions on atomic operators** — Not supported, Polyspace stubs the functions. This limitation can cause imprecise results.
- **IEEE floating point library functions** — Limited support, can cause imprecise results.

This limitation includes `isnan`, `isnanf`, `isnanl`, `isinf`, `isinff`, `isinfl`, `isnormal`, and `isfinite`.

Workaround: In each of your source files, include a file containing the function definitions or declarations:

- At the command line, use the option `-include filename`.
- In the Polyspace environment, in **Environment Settings > Include**, use the  button to add a row for your definition/declaration file.

Command-Line Information

Parameter: `-dialect`

Value: `none` | `gnu3.4` | `gnu4.6` | `gnu4.7` | `iso` | `cfront2` | `cfront3` | `visual` | `visual6` | `visual7.0` | `visual7.1` | `visual8` | `visual9.0` | `visual10` | `visual11.0`

Default: `none`

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

See Also

“Target operating system (C/C++)” on page 1-4 | “Target processor type (C++)” on page 2-3 | “C++11 Extensions (C++)” on page 2-10 | “Block char16/32_t types (C++)” on page 2-11

Related Examples

- “Verify Keil or IAR Dialects”

More About

- “Supported C++ 2011 Standards”

C++11 Extensions (C++)

Allow for C++11 language extensions. This option is available on the **Configuration** pane under the **Target & Compiler** node.

If your code uses any C++11 language constructs, select this option to allow this syntax during your analysis.

Settings

Default: Off

Off

The analysis does not allow C++11 syntax.

On

The analysis allows C++11 syntax.

Dependencies

You can only select this option when the **Dialect** option is `none`, `gnu4.6`, or `gnu4.7`.

Command-Line Information

Parameter: `-cpp-11-extension`

Default: `off`

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

See Also

“Dialect (C++)” on page 2-5 | “Block `char16/32_t` types (C++)” on page 2-11

More About

- “Supported C++ 2011 Standards”

Block char16/32_t types (C++)

The analysis does not allow `char16_t` or `char32_t` types. This option is available on the **Configuration** pane under the **Target & Compiler** node.

If you have defined `char16_t` and/or `char32_t` through a `typedef` statement or using includes, this option allows you to turn off the standard Polyspace definition of `char16_t` and `char32_t`.

Settings

Default: Off

Off

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

On

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

Dependencies

You can only select this option when:

- **Dialect** option is `none`, `gnu4.6`, or `gnu4.7`.
- **C++ 11 Extensions** is selected.

Command-Line Information

Parameter: `-no-uliterals`

Default: `off`

Example: `polyspace-code-prover-nodesktop -dialect gnu4.7 -lang cpp -cpp-11-extension -no-uliterals`

See Also

“Dialect (C++)” on page 2-5 | “C++11 Extensions (C++)” on page 2-10

More About

- “Supported C++ 2011 Standards”

Enum type definition (C++)

Allow the analysis to use different base types to represent an enumerated type, depending on the enumerator values and the selected definition. This option is available on the **Configuration** pane under the **Target & Compiler** node.

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

Settings

Default: auto-signed-int-first

auto-signed-int-first On

Uses the first type that can hold all of the enumerator values from the following list: signed int, unsigned int, signed long, unsigned long, signed long long, unsigned long long

auto-signed-first

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

auto-unsigned-first

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

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

Command-Line Information

Parameter: -enum-type-definition

Value: auto-signed-int-first | auto-signed-first | auto-unsigned-first

Default: auto-signed-int-first

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

Pack alignment value (C++)

Specify the default packing alignment for an analysis. This option is available on the **Configuration** pane under the **Target & Compiler** node.

If an invalid value is given, analysis will halt and display an error message. with a bad value or if this option is used in non visual mode (**Target operating system Visual** or **Dialect visual***).

Settings

Default: 8

- 1
- 2
- 4
- 8
- 16

Dependencies

This analysis option is available only when,

- **Target operating system** is set to no-predefined-OS or Visual.
- and **Dialect** is set to one of the **visual*** options.

Command-Line Information

Parameter: -pack-alignment-value

Value: 1 | 2 | 4 | 8 | 16

Default: 8

Example: polyspace-code-prover-nodesktop -lang cpp -pack-alignment-value 4

Ignore pragma pack directives (C++)

Specifies C++ #pragma packing alignment for structure, union, and class members. This option is available on the **Configuration** pane under the **Target & Compiler** node.

Settings

Default: Off

Off

Keeps C++ #pragma directives in the analysis

On

Allows C++ #pragma directives to be ignored in order to prevent link errors

Analysis will halt and display an error message with a bad value or if this option is used in non visual mode (**Target operating system** Visual or **Dialect** visual*).

Dependencies

This analysis option is available only when,

- **Target operating system** is set to no-predefined-OS or Visual.
- and **Dialect** is set to one of the visual* options.

Command-Line Information

Parameter: -ignore-pragma-pack

Default: Off

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

Support managed extensions (C++)

Visual C++ /FX option allows the partial translation of sources making use of managed extensions to Visual C++ sources without managed extensions. This option is available on the **Configuration** pane under the **Target & Compiler** node.

Settings

Default: Off

Off

Do not support managed extensions

On

Allows the analysis of a project containing translated sources obtained by compilation of a Visual project using the /FX Visual option.

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

These extensions are currently not taken into account by Polyspace analysis and can be considered as a limitation to analyze this kind of code. Managed files need to be located in the same folder as the original ones and Polyspace software will analyze managed files instead of the original ones without intrusion, and will permit you to remove part of the limitations due to specific extensions.

Dependencies

This analysis option is available only when,

- **Target operating system** is set to `no-predefined-OS` or `Visual`.
- and **Dialect** is set to one of the `visual*` options.

Command-Line Information

Parameter: `-support-FX-option-results`

Default: `off`

Example: `polyspace-code-prover-nodesktop -lang cpp -OS-target Visual -support-FX-option-results`

Import folder (C++)

Specifies a single directory to be included by *#import* directive. This option is available on the **Configuration** pane under the **Target & Compiler** node.

Settings

No default

Give the location of *.tlh files generated by a Visual Studio compiler when encountering *#import* directive on *.tlb files.

Dependencies

This analysis option is available only when,

- **Target operating system** is set to `no-predefined-OS` or `Visual`.
- and **Dialect** is set to one of the `visual*` options.

Command-Line Information

Parameter: `-import-dir`

Value: File location

Example: `polyspace-code-prover-nodesktop -OS-target Visual -dialect visual8 -import-dir /com1/inc`

Management of scope of 'for loop' variable index (C++)

Specify the scope of the index variable declared within a `for` loop. This option is available on the **Configuration** pane under the **Target & Compiler** node.

For example:

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

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

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

Settings

Default: `defined-by-dialect`

`defined-by-dialect`

Default behavior specified by selected dialect

`out`

The index variable is usable outside the scope of the `for` loop.

Default behavior for the dialect options `cfront2`, `crfront3`, `visual6`, `visual7` and `visual 7.1`

`in`

The index variable is not usable outside the scope of the `for` loop.

Default behavior for all other dialects, including `visual8`. The C++ standard specifies that the index is treated as `in`.

Command-Line Information

Parameter: `-for-loop-index-scope`

Value: `defined-by-dialect` | `out` | `in`

Default: `defined-by-dialect`

Example: `polyspace-code-prover-nodesktop -lang cpp -for-loop-index-scope in`

Management of `wchar_t` (C++)

Specify how to treat `wchar_t`. This option is available on the **Configuration** pane under the **Target & Compiler** node.

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

Settings

Default: `defined-by-dialect`

`defined-by-dialect`

Default behavior specified by selected dialect

`typedef`

Use according to `typedef` statement specified by Microsoft Visual C++ 6.0/7.0/7.1 dialects.

Default behavior for the dialect options `visual16`, `visual17.0` and `visual17.1`

`keyword`

Use as a keyword as given by the C++ standard

Default behavior for all other dialects, including `visual8`.

Command-Line Information

Parameter: `-wchar-t-is`

Value: `defined-by-dialect` | `typedef` | `keyword`

Default: `defined-by-dialect`

Example: `polyspace-code-prover-nodesktop -for-loop-index-scope keyword`

Set wchar_t to unsigned long (C++)

Specify the underlying type of `wchar_t` to be unsigned long. This option is available on the **Configuration** pane under the **Target & Compiler** node.

Settings

Default: Off

Off

Use the default underlying type of `wchar_t` as defined by the dialect or the **Management of wchar_t** option.

On

Set the type of `size_t` to unsigned long, as defined in the C++ standard.

For example, `sizeof(L'W')` will have the value of `sizeof(unsigned long)` and the `wchar_t` field will be aligned in the same way as the unsigned long field.

Command-Line Information

Parameter: `-wchar-t-is-unsigned-long`

Default: off

Example: `polyspace-code-prover-nodesktop -lang cpp -wchar-t-is-unsigned-long`

Set `size_t` to unsigned long (C++)

Force the underlying type of `size_t` to be `unsigned long`. This option is available on the **Configuration** pane under the **Target & Compiler** node. If you use this option, you can only redefine `size_t` with a `typedef` statement to `unsigned long`.

For example, Polyspace correctly applies this `typedef` statement because the type is `unsigned long`:

```
typedef unsigned long size_t;
```

However, Polyspace ignores this `typedef` statement,

```
typedef unsigned int size_t;
```

because the **Set `size_t` to unsigned long** option allows only `unsigned long`.

Settings

Default: Off

Off

Use the default underlying type of `size_t`, `unsigned int`

On

Set the type of `size_t` to `unsigned long`

Command-Line Information

Parameter: `-size-t-is-unsigned-long`

Default: `off`

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

Ignore link errors (C++)

Ignore linkage errors. This option is available on the **Configuration** pane under the **Environment Settings** node.

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

Default: Off

Off

Stop analysis for linkage errors.

On

Ignore the linkage errors if possible.

Command-Line Information

Parameter: `-no-extern-C`

Default: `off`

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

Check MISRA C++ rules

Specify whether to check for violation of MISRA C++ rules. Each value of the option corresponds to a subset of rules to check. This option is available on the **Configuration** pane under the **Coding Rules** node.

After analysis, the **Results Summary** 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
```

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

Related Examples


- “Specify Analysis Options”
- “Activate Coding Rules Checker”
- “Select Specific MISRA or JSF Coding Rules”

More About

- “Polyspace MISRA C++ Checker”
- “Software Quality Objective Subsets (C++)”
- “MISRA C++ Coding Rules”

Check JSF C++ rules

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. This option is available on the **Configuration** pane under the **Coding Rules** node.

After analysis, the **Results Summary** 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 dialect other than ISO, some rules might not be completely checked. For example, AV Rule 8: “All code shall conform to ISO/IEC 14882:2002(E) standard C++.”

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`

Related Examples

- “Specify Analysis Options”
- “Activate Coding Rules Checker”
- “Select Specific MISRA or JSF Coding Rules”

More About

- “Polyspace JSF C++ Checker”
- “JSF C++ Coding Rules”

Files and folders to ignore (C++)

Specify files and folders to ignore during coding rules checking. This option is available on the **Configuration** pane under the **Inputs & Stubbing** node. The files and folders are **not** ignored during Code Prover verification.

Settings

Default: all-headers


all-headers

 Ignores .h or .hpp files

all

 Ignores all files in include folders

custom

 Ignore include files and folders that you specify in the **File/Folder** view. To add files to the custom **File/Folder** list, select  to choose the files and folders to exclude. To remove a file or folder from the list of excluded files and folders, select the row.

 Then click .

Dependencies

This option is enabled only if you select one of the options **Check MISRA C++ rules**, **Check JSF C++ rules** or **Check custom rules**.

Command-Line Information

Parameter: -includes-to-ignore

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

Default: all-headers

Example: polyspace-code-prover-nodesktop -lang cpp -sources *file_name* -jsf-coding-rules required-rules -includes-to-ignore "C:\usr\include"

See Also

“Check MISRA C++ rules” | “Check JSF C++ rules” | “Check custom rules (C/C++)”

Related Examples

- “Specify Analysis Options”
- “Activate Coding Rules Checker”

Main entry point (C++)

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`.

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 when you select:

- Visual for **Target & Compiler > Target operating system**
- **Code Prover Verification > 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 (C++)”

Related Examples

- “Specify Analysis Options”

Verify module (C++)

Specify that Polyspace must generate a `main` function during verification if it does not find one in the source files.

Settings

Default: On

On

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**.
- 2 Calls functions specified by **Initialization functions** ahead of other functions.
- 3 Calls functions specified by **Functions to call** in arbitrary order.
- 4 Calls class methods specified by **Class** and **Functions to call within the specified classes**.

If you do not specify the above options explicitly, the generated `main`:

- Initializes all global variables except those declared with keywords `const` and `static`.
- Calls in arbitrary order all functions and class methods that are not called anywhere in the source files. Polyspace considers that global variables can be written between two consecutive function or methods calls. Therefore, in each called function or method, global variables initially have the full range of values allowed by their type.

Off

Polyspace stops verification if it does not find a `main` function in the source files.

Command-Line Information

Parameter: `-main-generator`

Default: Off

See Also

“Variables to initialize (C++)” | “Functions to call (C++)” | “Initialization functions (C++)” | “Class (C++)” | “Functions to call within the specified classes (C++)”

Related Examples

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

More About

- “Main Generator Overview”

Class (C++)

Specify classes that Polyspace uses to generate a `main`.

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

This option is enabled only if you select **Code Prover Verification > Verify module**.

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 (C++)” | “Functions to call within the specified classes (C++)” | “Analyze class contents only (C++)” | “Skip member initialization check (C++)”

Related Examples

- “Specify Analysis Options”
- “Simple Class”
- “Simple Inheritance”
- “Multiple Inheritance”
- “Abstract Classes”
- “Virtual Inheritance”
- “Other Types of Classes”

More About

- “Why Provide a Class Analyzer”
- “How the Class Analyzer Works”
- “Sources Verified”
- “Architecture of Generated Main”
- “Class Verification Log File”
- “Characteristics of Class and Log File Messages”
- “Behavior of Global Variables and Members”
- “Methods and Class Specifics”

Functions to call within the specified classes (C++)

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.

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.

Dependencies

This option is enabled only if you select **Code Prover Verification > Verify module**.

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 (C++)” | “Class (C++)” | “Analyze class contents only (C++)” | “Skip member initialization check (C++)”

Related Examples

- “Specify Analysis Options”
- “Simple Class”
- “Simple Inheritance”
- “Multiple Inheritance”
- “Abstract Classes”
- “Virtual Inheritance”
- “Other Types of Classes”

More About

- “Why Provide a Class Analyzer”
- “How the Class Analyzer Works”
- “Sources Verified”

- “Architecture of Generated Main”
- “Class Verification Log File”
- “Characteristics of Class and Log File Messages”
- “Behavior of Global Variables and Members”
- “Methods and Class Specifics”

Analyze class contents only (C++)

Specify that Polyspace must verify only methods of classes that you specify using the **Class** option.

Settings

Default: Off

On

Polyspace verifies the class methods only. It stubs functions out of class scope even if the functions are defined in your code.

Off

Polyspace verifies functions out of class scope in addition to class methods.

Dependencies

This option is enabled only if you select **Code Prover Verification > Verify module**. If you select this option, you must specify the classes using the **Class** 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 (C++)” | “Class (C++)” | “Functions to call within the specified classes (C++)” | “Skip member initialization check (C++)”

Related Examples

- “Specify Analysis Options”
- “Simple Class”
- “Simple Inheritance”
- “Multiple Inheritance”
- “Abstract Classes”
- “Virtual Inheritance”
- “Other Types of Classes”

More About

- “Why Provide a Class Analyzer”
- “How the Class Analyzer Works”
- “Sources Verified”
- “Architecture of Generated Main”
- “Class Verification Log File”
- “Characteristics of Class and Log File Messages”
- “Behavior of Global Variables and Members”
- “Methods and Class Specifics”

Skip member initialization check (C++)

Specify that Polyspace must not check whether each class constructor initializes all class members.

Settings

Default: Off

On

Polyspace does not check whether each class constructor initializes all class members.

Off

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

This option is enabled only if you select **Code Prover Verification > Verify module**. If you select this option, you must specify the classes using the **Class** option.

Command-Line Information

Parameter: `-no-constructors-init-check`

Default: Off

See Also

“Verify module (C++)” | “Class (C++)”

Related Examples

- “Specify Analysis Options”

Functions to call (C++)

Specify functions that you want the generated `main` to call. You can use this option only to specify functions that are not members of a class.

Settings

Default: `unused`

`none`

The generated `main` does not call any function.


`unused`

The generated `main` calls only those functions that are not being 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. Click  to enter function name.

Dependencies

This option is enabled only if you select **Verify module**.

Tips

- Select `unused` when you use the option **Run unit by unit verification**.
- If you want the generated `main` to call an inlined function, select `custom` and specify the function name.
- Select `none`:
 - If you do not want to verify uncalled functions. For applications that are not multitasking, Polyspace cannot verify a function unless it can be reached from `main`.
 - To verify a multitasking application without a `main`.

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 unused`

Related Examples

- [“Specify Analysis Options”](#)
- [“Automatically Generate a Main”](#)

More About

- [“Main Generator Overview”](#)

Variables to initialize (C++)

Specify global variables that you want the generated `main` to initialize.

If you use the generated `main` to initialize a global variable, inside a function, before the first write operation on the variable, Polyspace considers it to have any value allowed by its type.

Settings

Default: `uninit`

`uninit`

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 the keywords `static` and `const`.

`all`

The generated `main` initializes all global variables except those declared with the keyword `const`.

`custom`

The generated `main` only initializes global variables that you specify. Click  to enter variable name.

Dependencies

This option is enabled only if you select **Verify module**.

Command-Line Information

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

Value: `none` | `public` | `all` | `custom=variable1[,variable2[,...]]`

Default: `uninit`

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -main-generator-writes-variables all`

Related Examples

- [“Specify Analysis Options”](#)
- [“Automatically Generate a Main”](#)

More About


- [“Main Generator Overview”](#)

Initialization functions (C++)

Specify functions that you want the generated `main` to call ahead of other functions.

Settings

Default: None

Click  to add a field.

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);
}
```

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.

Command-Line Information

Parameter: `-functions-called-before-main`

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

Default: None

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -functions-called-before-main myClass::init`

Dependencies

This option is enabled only if you select **Verify module**.

Related Examples

- [“Specify Analysis Options”](#)
- [“Automatically Generate a Main”](#)

More About

- [“Main Generator Overview”](#)

Parameters (C++)

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.

Settings

Default: `public`


`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 class members, use the syntax `className::variableName`.

Command-Line Information

Parameter: `-variables-written-before-loop`

Value: `none` | `public` | `all` | `custom=variable1[,variable2[,...]]`

Default: `public`

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -variables-written-before-loop all`

See Also

“Inputs (C++)” on page 2-48 | “Initialization functions (C++)” on page 2-50 | “Step functions (C++)” on page 2-51 | “Termination functions (C++)” on page 2-53

Related Examples

- “Specify Analysis Options”
- “Configure Polyspace Analysis Options”

More About

- “Recommended Polyspace options for Verifying Generated Code”
- “Main Generation for Model Verification”

Inputs (C++)

This option is available only for model-generated code. Specify variables that the generated `main` must write to, at the beginning of every iteration of the cyclic code loop. At the beginning of every loop iteration, Polyspace considers these variables to have any value allowed by their type.

Settings

Default: `public`


`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 class members, use the syntax `className::variableName`.

Command-Line Information

Parameter: `-variables-written-in-loop`

Value: `none` | `public` | `all` | `custom=variable1[,variable2[,...]]`

Default: `public`

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -variables-written-in-loop all`

See Also

“Parameters (C++)” on page 2-46 | “Initialization functions (C++)” on page 2-50 | “Step functions (C++)” on page 2-51 | “Termination functions (C++)” on page 2-53

Related Examples

- “Specify Analysis Options”
- “Configure Polyspace Analysis Options”

More About


- “Recommended Polyspace options for Verifying Generated Code”
- “Main Generation for Model Verification”

Initialization functions (C++)

This option is available only for model-generated code. Specify functions that the generated main must call before the cyclic code begins.

Settings

Default: None

Click  to add a field. Enter function name. For class methods, use the syntax `className::functionName`.

Command-Line Information

Parameter: `-functions-called-before-loop`

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

Default: None

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -functions-called-before-loop myfunc1,myfunc2`

See Also

“Parameters (C++)” on page 2-46 | “Inputs (C++)” on page 2-48 | “Step functions (C++)” on page 2-51 | “Termination functions (C++)” on page 2-53

Related Examples

- “Specify Analysis Options”
- “Configure Polyspace Analysis Options”

More About

- “Recommended Polyspace options for Verifying Generated Code”
- “Main Generation for Model Verification”

Step functions (C++)

This option is available only for model-generated code. Specify functions that the generated `main` must call in each cycle of the cyclic code.

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.

custom

The generated `main` calls functions that you specify. Click  to add a field. Enter function name. For class methods, use the syntax `className::functionName`.

Tips

- If you specify a function for the option **Initialization functions** or **Termination functions**, you cannot specify it for **Step functions**.

Command-Line Information

Parameter: `-functions-called-in-loop`

Value: `none` | `unused` | `all` | `custom=function1[,function2[,...]]`

Default: none

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -functions-called-in-loop all`

See Also

“Parameters (C++)” on page 2-46 | “Inputs (C++)” on page 2-48 | “Initialization functions (C++)” on page 2-50 | “Termination functions (C++)” on page 2-53

Related Examples

- “Specify Analysis Options”

- “Configure Polyspace Analysis Options”

More About


- “Recommended Polyspace options for Verifying Generated Code”
- “Main Generation for Model Verification”

Termination functions (C++)

This option is available only for model-generated code. Specify functions that the generated `main` must call after the cyclic code loop.

Settings

Default: None

Click  to add a field. Enter function name. For class methods, use the syntax `className::functionName`.

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`

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

Default: None

Example: `polyspace-code-prover-nodesktop -sources file_name -main-generator -functions-called-after-loop myfunc1,myfunc2`

See Also

“Parameters (C++)” on page 2-46 | “Inputs (C++)” on page 2-48 | “Initialization functions (C++)” on page 2-50 | “Step functions (C++)” on page 2-51

Related Examples

- “Specify Analysis Options”
- “Configure Polyspace Analysis Options”

More About

- “Recommended Polyspace options for Verifying Generated Code”
- “Main Generation for Model Verification”

No STL stubs (C++)

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

Settings

Default: Off

On

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

Off

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

Tips

Use this option when Polyspace implementation of the Standard Template Library causes linking errors.

Command-Line Information

Parameter: `-no-stl-stubs`

Default: Off

Related Examples

- “Specify Analysis Options”


Functions to stub (C++)

Specify functions to stub during verification. This option is available on the **Configuration** pane under the **Inputs & Stubbing** node.

For these 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.

Settings

Click  to enter function name.

When entering function names, use one of the following syntaxes:

- Basic syntax, with extensions for classes and templates:

Function Type	Syntax
Simple function	<code>test</code>
Class method	<code>A::test</code>
Template method	<code>A<T>::test</code>

- Syntax with function arguments, to differentiate overloaded functions. Function arguments are separated with semicolons:

Function Type	Syntax
Simple function	<code>test()</code>
Class method	<code>A::test(int;int)</code>
Template method	<code>A<T>::test<T>::test(T;T)</code>

Command-Line Information

Parameter: `-functions-to-stub`

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

Default: None

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

See Also

“No automatic stubbing (C/C++)” | “Variable/function range setup (C/C++)” | “Functions to stub (C)”

Related Examples

- “Specify Analysis Options”
- “Specify Functions to Stub Automatically”
- “Constrain Data with Stubbing”

More About

- “Stubbing Overview”
- “When to Provide Function Stubs”
- “Stubbing Examples”

Tuning Precision and Scaling Parameters

Precision versus Time of Verification

There is a compromise to be made to balance the time required to obtain results, and the precision of those results. Consequently, launching Polyspace verification with the following options will allow the time taken for verification to be reduced but will compromise the precision of the results. It is suggested that the parameters should be used in the sequence shown - that is, if the first suggestion does not increase the speed of verification sufficiently then introduce the second, and so on.

- switch from `-O2` to a lower precision;
- set the “Respect types in global variables (C/C++)” and “Respect types in fields (C/C++)” options;
- set the option “Depth of verification inside structures (C/C++)” to 2, then 1, or 0;
- stub manually missing functions which write into their arguments.

Precision versus Code Size

Polyspace verification can make approximations when computing the possible values of the variables, at any point in the program. Such an approximation will use a superset of the actual possible values.

For instance, in a relatively small application, Polyspace verification might retain very detailed information about the data at a particular point in the code, so that for example the variable VAR can take the values `{ -2; 1; 2; 10; 15; 16; 17; 25 }`. If VAR is used to divide, the division is green (because 0 is not a possible value). If the program being analyzed is large, Polyspace verification would simplify the internal data representation by using a less precise approximation, such as `[-2; 2] U {10} U [15 ; 17] U {25}` . Here, the same division appears as an orange check.

If the complexity of the internal data becomes even greater later in the verification, Polyspace verification might further simplify the VAR range to (say) `[-2; 20]`.

This phenomenon leads to the increase or the number of orange warnings when the size of the program becomes large.

Note: The amount of simplification applied to the data representations also depends on the required precision level (O0, O2), Polyspace verification will adjust the level of simplification:

- -O0: shorter computation time. You only need to focus on red and gray checks.
 - -O2: less orange warnings.
 - -O3: less orange warnings and bigger computation time.
-

Verification level (C++)

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

Settings

Default: Software Safety Analysis level 2

C++ 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 thrice 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 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.

Dependency

You cannot use the C++ Source Compliance Checking setting with batch or interactive mode.

Command-Line Information

Parameter: -to

Value: cpp-compliance | pass0 | pass1 | pass2 | pass3 | pass4 | other

Default: pass2

Example: polyspace-code-prover-nodesktop -sources *file_name* -to pass2

Related Examples

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

Inline (C++)

Specify functions that the verification must clone for every function call. For instance, if you specify the function `func` for inlining and `func` is called twice, the software creates two copies of `func` during verification. The copies are named using the convention `funcver_pst_cloned_tot` where *ver* is the version number and *tot* is the total number of copies.

Settings

Default: No function is inlined.

Click  to enter function name.

Tips

- 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`.
- This option applies to all overloaded methods of a class.

Command-Line Information

Parameter: `-inline`

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

Default: None

Example: `polyspace-code-prover-nodesktop -sources file_name -inline my_func`

Related Examples

- “Specify Analysis Options”
- “Reduce Procedure Complexity”

Other (C++)

In this section...
“-extra-flags” on page 2-62
“-cpp-extra-flags” on page 2-62
“-il-extra-flags” on page 2-62

Specify special options for C++ verification, which are provided by MathWorks if required.

-extra-flags

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-code-prover-nodesktop -extra-flags -param1 -extra-flags -  
param2
```

-cpp-extra-flags

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-code-prover-nodesktop -cpp-extra-flags -stubbed-new-may-  
return-null
```

-il-extra-flags

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-code-prover-nodesktop -il-extra-flags flag
```


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.

Mark the offending code block by two `#pragma` directives, one at the beginning of the `asm` 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`.

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`

Related Examples

- “Run Verification from 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`.

Note: In the Polyspace environment, 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
```

Related Examples

- “Run Verification from 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

Related Examples

- “Run Verification from Command Line”

-from

Specify which verification phase to start from

Syntax

`-from verification-phase`

Description

`-from verification-phase` specifies which verification phase to start from. You can use this option only on an existing verification to elaborate on the results that you have already obtained.

For example, if a verification has been completed `-to pass1`, Polyspace verification can be restarted `-from pass1` and hence save on verification time. You usually use the option after one run with the `-to` option, although you can also use it to recover after a power failure. Possible values are as described in the `-to verification-phase` section, with the addition of the `scratch` option.

Limitations

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

Examples

Run a verification to the second pass, and then restart the verification from the same pass.

```
polyspace-code-prover-nodesktop -to pass2
```

polyspace-code-prover-nodesktop -from pass2

See Also

“Verification level (C)” on page 1-101 | polyspaceCodeProver

Related Examples

- “Run Verification from 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

Related Examples

- “Run Verification from Command Line”

-I

Specify include folder for compilation

Syntax

`-I folder`

Description

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

Polyspace software 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`

Related Examples

- “Run Verification from 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.

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`

Related Examples

- “Run Verification from Command Line”

-interactive

Enable interactive remote analysis

Syntax

-interactive

Description

-interactive enables the interactive remote analysis mode. In this remote analysis mode, the analysis is tethered to your local computer. Therefore, on your local computer:

- If you are running the analysis from the Polyspace user interface, using the **Advanced Settings > Other** text box, you cannot close the user interface while the analysis is running.
- If you are running the analysis from the command line, you cannot close the command-line window while the analysis is running.

In this mode, the analysis is not queued on the cluster. Therefore, if a worker is not available on the cluster, the analysis aborts. The software downloads the results to your local computer after the analysis.

For interactive remote analysis, you need:

- MATLAB Distributed Computing Server on the cluster
- MATLAB, Polyspace and Parallel Computing Toolbox on your local computer

Dependency

You cannot use -interactive with the **Verification Level** (-to) option set to C source compliance checking or C++ source compliance checking.

Examples

If you do not have remote verification setup already, to run an interactive remote verification from the command line, use with the -scheduler option.

```
polyspace-code-prover-nodesktop -interactive -scheduler NodeHost  
polyspace-code-prover-nodesktop -interactive -scheduler  
MJSName@NodeHost
```

See Also

“Batch (C/C++)” | -scheduler | polyspaceCodeProver

Related Examples

- “Run Verification from Command Line”
- “Set Up Remote Verification and Analysis”

-keep-all-files

Retain intermediate results and associated working files

Syntax

`-keep-all-files`

Description

`-keep-all-files` retains all intermediate results and associated working files. If the source code remains unchanged, you can restart a verification from the end of a completed pass. If you do not specify this option, intermediate results are erased at the end of a verification.

Tips

- When you select this option, you can restart a Polyspace verification from the end of a complete pass (if the source code is unchanged). If you do not use this option, you must restart the verification from the beginning.
- This option is applicable only to client verifications. Before you download results from the server, intermediate results are removed.
- This option uses up a lot of disk space to store the intermediate files. Therefore, use this option sparingly for debugging.

Examples

Run verification to pass1 and keep intermediate files.

```
polyspace-code-prover-nodesktop -keep-all-files -to pass1
```

See Also

`polyspaceCodeProver`

Related Examples

- “Run Verification from Command Line”

-known-NTC

Ignore known non-terminating calls

Syntax

```
-known-NTC "func1, func2..."
```

Description

-known-NTC "*func1, func2...*" renames known non-terminating calls as Known Non-Terminating Calls in the results. The listed functions, *func1* and *func2* appear as Known Non-Terminating Calls instead of Non-Terminating Calls, enabling easy filtering.

By default, non-terminating calls are listed as **Non-Terminating Call** in the verification results. After a few verifications, it is possible that a few functions "do not terminate". Some functions, such as tasks and threads, contain infinite loops by design, while functions that exit the program, such as `kill_task`, `exit` or `Terminate_Thread`, are often stubbed by means of an infinite loop. If you use these functions often or if the results are for presentation to a third party, you can filter non-terminating calls (K_NTC) of that kind in the Viewer.

Examples

Run a verification and ignore known infinite calls to `kill_task` and `exit`.

```
polyspace-code-prover-nodesktop -lang c -known-NTC "kill_task,exit"
```

See Also

“Known non-terminating call” | `polyspaceCodeProver`

Related Examples

- “Run Verification from Command Line”

-lang

Specify code language for the project

Syntax

```
-lang [c|cpp]
```

Description

`-lang [c|cpp]` specifies the code language for the project, either `c` for C code or `cpp` for C++ code.

If you do not specify a language, Polyspace tries to detect the language from the source files.

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

Examples

Define the language of your Polyspace Project as C++.

```
polyspace-code-prover-nodesktop -lang cpp -sources...
```

See Also

`polyspaceCodeProver`

Related Examples

- “Run Verification from Command Line”

-less-range-information

Limit range information displayed in results

Syntax

`-less-range-information`

Description

`-less-range-information` limits the amount of range information displayed in the results.

By enabling this option, range information is available only for assignments, not read operations.

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

Examples

Consider the following code:

```
x = y + z
```

By enabling this option:

```
polyspace-code-prover-nodesktop -less-range-information
```

range information is available only when you place your cursor over `x`. Without this option enabled, range information is available for `x`, `y`, and `z`.

See Also

`polyspaceCodeProver`

Related Examples

- “Run Verification from Command Line”

- “Use Range Information in Results Manager”

-max-processes

Specify the maximum number of processes that can run simultaneously on a multicore system.

Syntax

`-max-processes num`

Description

`-max-processes num` specifies the maximum number of processes that can run simultaneously on a multicore system. The valid range of *num* is 1 to 128. The default is 4.

Examples

Disable parallel processing during the analysis.

```
polyspace-code-prover-nodesktop -max-processes 1
```

See Also

`polyspaceCodeProver`

Related Examples

- “Run Verification from Command Line”

-no-pointer-information

Turn off pointer information in your results

Syntax

```
-no-pointer-information
```

Description

`-no-pointer-information` turns off the pointer information in your analysis results. When you select this option, the software does not provide pointer information through tooltips. As computing pointer information can take a long time, selecting this option can significantly reduce analysis time.

Examples

Consider the following example:

```
x = *p;
```

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

```
polyspace-code-prover-nodesktop -no-pointer-information
```

See Also

`polyspaceCodeProver`

Related Examples

- “Run Verification from Command Line”

-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 all 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
-dialect none
-dos
-misra2 required-rules
-includes-to-ignore 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`

Related Examples

- “Run Verification from 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`

Related Examples

- “Run Verification from 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`.

Examples

Specify the name of the analysis report.

```
polyspace-code-prover-nodesktop -report-template Developer  
-report-output-name Airbag_v3.rtf
```

See Also

“Output format (C/C++)” on page 1-122 | “Report template (C/C++)” | `polyspaceCodeProver`

Related Examples

- “Run Verification from Command Line”
- “Generate Report from Command Line”

-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. In the user interface, the default location is `C:Polyspace_Results`.

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`

Related Examples

- “Run Verification from Command Line”

-scheduler

Specify cluster or job scheduler

Syntax

`-scheduler schedulingOption`

Description

`-scheduler schedulingOption` specifies the head node of the MDCS 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`

Related Examples

- “Run Verification from Command Line”
- “Manage Remote Verifications”

-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. The list must be in quotations and separated by commas. 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`

Related Examples

- “Run Verification from Command Line”

-sources-list-file

Specify file containing list of sources

Syntax

```
-sources-list-file "filename"
```

Description

`-sources-list-file "filename"` specifies a text file that lists each file name that you want to analyze.

The file must list only one source file per line, and each file name must be given with its absolute path.

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`

Related Examples

- “Run Verification from 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` keeps temporary files in the results folder. By default, temporary files are stored in the standard `/temp` or `C:\Temp` folder. This option stores the temporary files in a subfolder of the results folder. Use this option only when the temporary folder partition does not have enough disk space. If the results folder is mounted on a network drive, this option can slow down your processor.

Examples

Store temporary files in the results folder.

```
polyspace-code-prover-nodesktop -tmp-dir-in-results-dir
```

See Also

`polyspaceCodeProver`

Related Examples

- “Run Verification from 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 of your Polyspace product.

```
polyspace-code-prover-nodesktop -v  
produces output such as:
```

```
Polyspace Code Prover 9.0 (R2013b)
```

```
Copyright 1990-2013 The Mathworks, Inc.
```

See Also

polyspaceCodeProver

Related Examples

- “Run Verification from 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`

Related Examples

- “Run Verification from Command Line”

Check Reference

Absolute address

Absolute address is assigned to pointer

Description

This check determines whether an absolute address is assigned to a pointer.

Examples

Absolute address assigned to pointer

```
void main() {  
    int *p = (int *)0x32;  
    int x = *p;  
    p++;  
    x = *p;  
}
```

In this example, `p` is assigned an absolute address. The check is orange because the software does not have information about the absolute address and cannot verify, for example, the validity of the address and the availability of memory.

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** check even though `p` still points to an absolute address.

Correction — Use Polyspace analysis option

You can use absolute addresses in your code and not produce an orange **Absolute address** error. To allow absolute addresses, on the **Configuration** pane, under **Verification Assumptions**, select **Green absolute address checks**.

```
void main() {
```

```
int *p = (int *)0x32;  
int x = *p;  
p++;  
x = *p;  
}
```

Check Information

Category: Static memory

Language: C | C++

Acronym: ABS_ADDR

More About

- “Review Orange Check”

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.

Examples

Array is mapped to larger array

```
typedef int smallArray[10];
typedef int largeArray[100];

void main() {
    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() {
    float val = 0.0;
    myFuncStruct.func(val);
}

```

In this example, because the global variable `myFuncStruct` is not initialized, the function pointer `myFuncStruct.func` contains `NULL`. Therefore, 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() {
    for (int i=1; i<=MAX_MEMSEG; i++)
        operation(i);
}

```

In this example, the function pointer `operation` is cast to the contents of a location in memory. Because Polyspace cannot determine whether the location contains a variable or a function code, the **Absolute address** check produces an orange error on the cast. Subsequently, when the pointer `operation` is dereferenced, the **Correctness condition** check produces a red error.

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 = func;
    int arg = 0, result = funcPtr(&arg);
}
```

In this example, the function pointer `funcPtr` points to a function with argument type `complex*`. However, it is assigned the function `func` whose argument type is `int*`. Because of this type mismatch, the **Correctness condition** check produces a red error.

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 produces a red error.

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 produces a red error.

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

Category: Other

Language: C | C++

Acronym: COR

See Also

“Variable/function range setup (C/C++)”

More About

- “Check Global Variable Ranges with Global Assert”
- “Review Orange Check”

C++ specific checks

C++ specific invalid operations occur

Description

This check 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.

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 in two categories:

- `n = 0`: When the `new` operator uses this argument, the **C++ specific checks** produce an error.
- `n > 0`: When the `new` operator uses this argument, the **C++ specific checks** is green.

Combining the two categories of arguments, the **C++ specific checks** 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 `shapePtr` is used with the `typeid` operator, the **C++ specific checks** produce an error.
- **Not NULL**: When `shapePtr` is used with the `typeid` operator, the **C++ specific checks** is green.

Combining these two cases, the **C++ specific checks** produce an orange error on the `typeid` operator in the first `if` statement branch in `main`.

Following this orange error, Polyspace considers that `shapePtr` is not `NULL`. Therefore, the **C++ specific checks** on the `typeid` operator in the second `if` statement branch is green.

Check Information

Category: C++

Language: C++

Acronym: CPP

More About

- “Review Orange Check”

Division by zero

Division by zero occurs

Description

This check determines whether the right operand of a division or modulus operation is zero.

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.

Correction — Check for zero denominator

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.

Correction — Check for zero denominator

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.

Correction — Check for zero denominator

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>
#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;
        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.

Correction — Check for zero denominator

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>
#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;
        if(divisor < -0.00001 || divisor > 0.00001)
            printf(" .2f ", 1.0/divisor);
        else
            printf(" Infinite ");
    }
}
```

Check Information

Category: Numerical

Language: C | C++

Acronym: ZDV

More About

- “Review Orange Check”

Exception handling

Exception handling

Description

This check determines whether:

- A function call throws an exception.
- The exception is caught.

This check appears on both a function call as well as the function body. Use this check to follow the propagation of error from an entry-point function down the branches of the call tree.

Examples

Exception in calls 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 **Exception handling** error on both the function call and function body. A red **Exception handling** error also appears on `main` because a function call inside `main` throws an exception.

Exception handled through try/catch construct

```
class error {
    error() { }
    error(const error&) { }
};

void funcNegative() {
    try {
        throw error();
    }
    catch (error NegativeError) {
    }
}

void funcPositive() {
    try {
    }
    catch (error PositiveError) {
    }
}
```

```
int input();
void main()
{
    int val=input();
    if(val < 0)
        funcNegative();
    else
        funcPositive();
}
```

In this example:

- The call to `funcNegative` throws an exception. However, the exception is placed inside a `try` block. Therefore, the exception propagates to the corresponding `catch` block and does not continue farther. The **Exception handling** 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 calls to constructor

```
class error {
};

class X
{
public:
    X() {
        throw error();
    }
    ~X() {
        ;
    }
};

int main() {
    try {
        X * px = new X ;
        delete X;
    } 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 **Exception handling** error on the constructor body and the `new` operator. The exception then propagates to the `catch` block and does not continue farther. Therefore the **Exception handling** check on the `main` function appears green.

The green `assert` statement shows that the exception has propagated to the `catch` block.

Exception in calls 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 **Exception handling** 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 arguments to `throw` in the `Account::debitAccount` method are declared to be either `int` or `char`. However, the method throws an exception with type `negativeBalance`. Therefore, the **Exception handling** check produces a red error on `throw`.

Check Information

Category: C++

Language: C++

Acronym: EXC

More About

- “Review Orange Check”

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

To detect functions that are called from unreachable code, on the **Configuration** pane, select **Check Behavior**. For **Detect uncalled functions**, select **all** or **called-from-unreachable**.

To find where the function is called, use the **Call Hierarchy** pane. For more information, see “View Call Tree for Functions”.

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 checkRatio(double ratio) {
    checkUpperBound(ratio);
    checkLowerBound(ratio);
}

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 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

Category: Data flow

Language: C | C++

Acronym: FNR

See Also

“Detect uncalled functions (C/C++)” | “Function not called” | “Unreachable code”

Function returns a 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.

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 returns a value** check returns a red error on the definition of `reply()`.

Correction — Return value for all inputs

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 returns a value** check returns an orange error on the definition of `reply()`.

Correction — Return value for all inputs

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

Category: C++

Language: C++

Acronym: FRV

See Also

“Initialized return value”

More About

- “Review Orange Check”

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.

To detect functions that are not called, on the **Configuration** pane, select **Check Behavior**. For **Detect uncalled functions**, select **all** or **never-called**.

Examples

Function not called

```
#define max 100
int var;
int getValue(void);
int getSaturation(void);

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;
    }
}

void reset() {
    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`.

Correction: Call Function

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 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();
    }
}

void reset() {
    var=0;
}
```

Function Called from Another Uncalled Function

```
#define max 100
int var;
int numberOfResets;
int getValue();
int getSaturation();

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;
    }
}

void reset() {
    updateCounter();
    var=0;
}

void updateCounter() {
    numberOfResets++;
}

```

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`.

Correction: Call Function

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 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();
    }
}

void reset() {

```

```
    updateCounter();  
    var=0;  
}  
  
void updateCounter() {  
    numberOfResets++;  
}
```

Check Information

Category: Data flow

Language: C | C++

Acronym: FNC

See Also

“Detect uncalled functions (C/C++)” | “Function not reachable”

Illegally dereferenced pointer

Pointer is dereferenced outside bounds

Description

This check on a pointer dereference determines whether the pointer points outside its bounds.

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

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.

Correction — Remove illegal dereference

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.

Correction — Avoid memory access outside structure field

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.

Correction — Avoid NULL pointer dereference

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.

Correction — Use correct type for bit fields

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

```
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.

Correction — Check return value of `malloc` for `NULL`

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.

Correction — Assign sufficient memory to pointer

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 = 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.

Correction — Allocate full memory

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 = malloc(sizeof(cuboid));
    if(cuboidPtr!=NULL) {
        cuboidPtr->length = 10;
        cuboidPtr->breadth = 10;
    }
}
```

Correction — Use Polyspace analysis option

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**.

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



```
typedef struct {
    int length;
    int breadth;
} rectangle;

typedef struct {
    int length;
    int breadth;
    int height;
} cuboid;

void main() {
    cuboid *cuboidPtr = 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 = &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.

Correction — Assign address of structure instead of field

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 = &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;
}
```

Correction — Use Polyspace analysis option

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**.

```
#include<stdlib.h>
typedef struct {
    int length;
    int breadth;
} square;
```

```
void main() {
    square mySquare;
    char* squarePtr = &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;
}
```

Check Information

Category: Static memory

Language: C | C++

Acronym: IDP

See Also

“Allow incomplete or partial allocation of structures (C)” | “Enable pointer arithmetic across fields (C)”

Initialized return 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 every function call.

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 **Initialized return value** check returns a red error. The second call `reply(ch)` always returns a value. Therefore, the check on this call is green.

Correction — Return value for all inputs

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 **Initialized return value** check returns an orange error on `reply(ch)`.

Correction — Return value for all inputs

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

Category: Data flow

Language: C

Acronym: IRV

See Also

“Function returns a value”

More About

- “Review Orange Check”

Inspection points

Variable range information appears

Description

This user-specified check provides range information on specified variables. If you want to know the range of the variables `var1`, `var2`, ... at a certain point in the code, place the line `#pragma var1 var2 ...` at that point. After verification, to see the variable range, place your cursor on the variable name.

Note: The tooltip indicates the range that Polyspace considers, not the actual variable range. Because of approximations, the variable range that Polyspace considers can sometimes be a superset of the actual variable range. Use this check to help understand the cause of other Polyspace checks.

Examples

View range of variable

```
int input();

void main() {
    int num=input();
    int i;
    if(num>0 && num<10) {
        for(i=0; i<20; i++)
            num+=i;
        #pragma Inspection_Point num
    }
    #pragma Inspection_Point num
}
```

In this example, if you place your cursor on the variable `num` in the `#pragma` statements, you can view its range. In the first case, the tooltip shows the range `[191 .. 199]`. In the second case, the tooltip shows the range `[-231 .. 0]` or `[10 .. 231 - 1]`. The

second range shows that Polyspace considers the return value of `input()` to be in the full range of type `int`.

Check Information

Category: Other

Language: C

Acronym: IPT

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.

Examples

Invalid use of standard library float routine

```
#include<assert.h>
#include<math.h>
#define HALF_PI 1.5707963267948966
#define LARGE_EXP 710

enum operation {
    ASIN,
    ACOS,
    TAN,
    SQRT,
    LOG,
    EXP,
    ACOSH,
    ATANH };

enum operation getOperation();
double getVal();

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 TAN:  assert( myVal == HALF_PI);
        res = tan(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;
    case ACOSH: assert(myVal < 1.0);
        res = acosh(myVal);
        break;
    case ATANH: assert(myVal <= -1.0 || myVal >= 1.0);
        res = atanh(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.

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;
}
```

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.

Correction — Call function with valid arguments

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

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

char* Copy_First_Six_Letters(void) {
    char str1[10],str2[6];
    printf("Enter string:\n");
    scanf("%s",str1);
    memcpy(str2,str1,6);
    return str2;
}
```

Invalid use of standard library string routine

```
#include <stdio.h>

char* Copy_String(void)
{
    char *res;
    char gbuffer[5],text[20]="ABCDEFGHijkl";
    res=strcpy(gbuffer,text);
    return(res);
}
```

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.

Correction — Call function with valid arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

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

```
char* Copy_String(void)
{
    char *res;
    char gbuffer[20],text[20]="ABCDEFGHijkl ";
    res=strcpy(gbuffer,text);
    return(res);
}
```

Check Information

Category: Other

Language: C | C++

Acronym: STD_LIB

More About

- “Review Orange Check”

Known non-terminating call

Called function specified with `-known-NTC` does not return to calling context

Description

This check appears on a function call if:

- The function does not return to its calling context.
- You have specified the function name as an argument of the option `-known-NTC`. Use this option to specify functions that contain a known infinite loop or another error.

Examples

Known non-terminating call error

```
enum boolean {TRUE, FALSE};

void task();

void executeTask (enum boolean res) {
    do {
        task();
    } while(res==TRUE);
}

int inputCh();

void main() {
    int ch = inputCh();
    if(ch==1)
        executeTask(TRUE);
    else
        executeTask(FALSE);
}
```

In this example, in the first `if` statement branch, `executeTask` does not return to the calling context because of an infinite loop in the function body. If you:

- Run verification on the command-line with option `-known-NTC "executeTask"`
- Specify `-known-NTC "executeTask"` for **Advanced Settings > Other** on the **Configuration** pane

a red **Known non-terminating call** appears on the call to `executeTask`.

Check Information

Category: Control flow

Language: C

Acronym: K_NTC

See Also

“Non-terminating call”

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.

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.

Correction — Initialize variable before using on right side of assignment

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.

Correction — Initialize variable before using with relational operator

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()`.

Correction — Initialize variable before passing to function

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.

Correction — Initialize variable before passing to function

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;
    }
    printf("The array elements are: \n");
    for(indexFront = 0; indexFront< arrSize; indexFront ++)
        printf("Element[%d]: %d", indexFront, arr[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.

Correction— Initialize structure

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 Summary** pane or **Source** pane.
- 2 View the message on the **Check 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 Summary** pane or **Source** pane.
- 2 View the message on the **Check Details** pane.

Check Information

Category: Data flow

Language: C | C++

Acronym: NIVL

See Also

“Non-initialized pointer” | “Non-initialized variable”

More About

- “Review Orange Check”

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.

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()`.

Correction — Initialize pointer before passing to function

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 = 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.

Correction — Initialize pointer before dereference

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 = 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.

Correction — Use char array instead of char* pointer

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.

Correction — Initialize `char*` pointers through `calloc`

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

Category: Data flow

Language: C | C++

Acronym: NIP

See Also

“Non-initialized local variable” | “Non-initialized variable”

More About

- “Review Orange Check”

Non-initialized variable

Variable other than local variable is not initialized before being read

Description

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

By default, Polyspace considers that global variables are initialized according to ANSI C standards. For instance, the default initial value of an `int` variable is 0.

To prevent this default assumption during analysis, on the **Configuration** pane, select **Inputs & Stubbing**. Select **Ignore default initialization of global variables**. This option is not available for C++ code.

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.

Correction — Initialize global variable before use

One possible correction is to initialize the global variable `globVar` before use.

```
int globVar;
int getVal();
```

```
void main() {  
    int val = getVal();  
    if(val>=0 && val<= 100)  
        globVar += val;  
}
```

Check Information

Category: Data flow

Language: C | C++

Acronym: NIV

See Also

“Ignore default initialization of global variables (C)” | “Non-initialized local variable” | “Non-initialized pointer”

More About

- “Review Orange Check”

Non-null this-pointer in method

this pointer is null during member function call

Description

This check on a `this` pointer dereference determines whether the pointer is `NULL`.

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 **Non-null this-pointer in method** produces a red error.

Correction — Initialize pointer with valid address

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

Category: C++

Language: C++

Acronym: NNT

More About

- “Review Orange Check”

Non-terminating call

Called function does not return to calling context

Description

This check on a function call determines whether the called function returns to its calling context. A function does not return to its calling context if it contains a run-time error.

Depending on the context, a non-terminating call appears in the user interface in two different ways:

- A dashed red underline on the function call. The dashed red underline indicates that you can find the line containing the error in the function body.
 - To find the source of error, place your cursor on the function call.
 - To navigate to the source of error, right-click the function call and select **Go to Cause**.
- A red error on the function call. The red indicates that there is at least one other call to the same function that does not produce a **Non-terminating call** error. You find the error, which is orange, inside the function body.

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

Category: Control flow

Language: C | C++

Acronym: NTC

See Also

“Known non-terminating call”

Non-terminating loop

Loop does not terminate or contains an error

Description

This check on a loop determines whether the loop terminates or contains an error in one of its iterations. If the check fails, a red error appears on the loop command.

For a red **Non-terminating loop** check, on the **Source** pane, place your cursor on the red loop command. A tooltip appears explaining the possible reason for the red check.

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.

Correction — Ensure Loop Termination

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.

Correction — Avoid loop iteration containing 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.

Correction — Avoid loop iteration containing error

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.

Correction — Reduce loop iterations

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

Category: Control flow

Language: C | C++

Acronym: NTL

Object oriented programming

Dynamic type of `this` pointer is incorrect

Description

This check on dereference of a `this` pointer or pointer to method determines whether the dereference is legal.

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 **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 **Object oriented programming** check produces a red error.

Check Information

Category: C++

Language: C++

Acronym: OOP

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.

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]);
}
```

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]`.

Correction — Keep array index less than array size

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]);
}
```

Check Information

Category: Static memory

Language: C | C++

Acronym: OBAI

See Also

“Illegally dereferenced pointer”

More About

- “Review Orange Check”

Overflow

Arithmetic operation causes overflow

Description

This check on an arithmetic operation determines whether the result overflows. An overflow occurs when the value of a variable falls outside the range allowed by its type.

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} . Since the maximum value that the type `int` can hold on a 32-bit target is $2^{31} - 1$, the **Overflow** check on the multiplication produces a red error.

Overflow due to left shift on signed integers

```
void main(void)  
{  
    unsigned int i;  
  
    i = 1090654225 << 1;  
}
```

In this example, an **Overflow** error occurs because of 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 can be represented by `float` on a 32-bit target. Therefore, the operation `val * 2` results in an **Overflow** error.

Negative overflow

```

#define FLT_MAX 3.40282347e+38F
#define FLT_MIN 1.17549435e-38

int input();

void main() {
    int choice=input();

    if(choice==0)
        float_negative_overflow();
    else
        int_negative_overflow();
}

void float_negative_overflow() {
    float zer_float = FLT_MIN;
    float min_float = -FLT_MAX;

    zer_float = zer_float * zer_float;
    min_float = -min_float * min_float;
}

void int_negative_overflow() {
    int min_int = -2147483648;
}

```

In this example:

- In `float_negative_overflow`, there are two cases of underflow:
 - In the first case, `zer_float` contains the closest possible number to zero that can be represented by the type `float`. Because the operation `zer_float * zer_float` produces a number that is even closer to zero, it cannot be represented

by the type `float`. However, the **Overflow** check does not detect this kind of underflow.

- In the second case, `min_float` contains the most negative number that can be represented by the type `float`. Because the operation `-min_float * min_float` produces a number that is further negative, it cannot be represented by the type `float`. Therefore, 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.

Since the maximum value that a signed integer can have is `2147483647`, a overflow occurs in the second step. Therefore, 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 can be represented by the type `char`. Therefore 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	<code>int, long, unsigned long</code>
Hexadecimal	<code>int, unsigned int, long, unsigned long</code>
Float	<code>float, double</code>

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. Because it is an unsigned integer, it can take values in `[0, 3]`. 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, because `myBit` is an `enum` variable, it is implemented through a signed integer according to the ANSI C90 standard. `myBit` occupies 2 bits. Because it is a signed integer, it can take values in `[-2, 1]`. 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**.

Check Information

Category: Numerical

Language: C | C++

Acronym: OVFL

See Also

“Detect overflows (C/C++)” | “Ignore overflowing computations on constants (C/C++)” | “Overflow computation mode (C/C++)”

More About

- “Review Orange Check”

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.

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 **Shift operations** check produces a red error.

Correction — Keep shift amount within bounds

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.

Correction — Use Polyspace analysis option

You can use left shifts on negative numbers and not produce a red **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;  
}
```

Check Information

Category: Numerical

Language: C | C++

Acronym: SHF

See Also

“Allow negative operand for left shifts (C/C++)”

More About

- “Review Orange Check”

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.

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.

Correction — Remove Redundant Test

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.

Correction — Remove Redundant Test

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.

Correction — Combine Tests with &&

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

Category: Data flow

Language: C | C++

Acronym: UNR

See Also

“Function not called” | “Function not reachable”

More About

- “Gray Checks”

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.

Examples

Red assert on array index

```
#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;
}

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.

Correction — Correct cause of assert failure

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`.

Correction — Check return value for 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>

double getNumber();
void squareRootOfDifference(double firstNumber, double 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

Category: Other

Language: C | C++

Acronym: ASRT

More About

- “Review Orange Check”

Approximations Used During Verification

- “Why Polyspace Verification Uses Approximations” on page 5-2
- “Approximations Made by Polyspace Verification” on page 5-4
- “Limitations of Polyspace Verification” on page 5-9

Why Polyspace Verification Uses Approximations

In this section...

“What is Static Verification” on page 5-2

“Exhaustiveness” on page 5-3

What is Static Verification

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

Polyspace verification works by approximating the software under verification, using representative approximations of software operations and data.

For example, consider the following code:

```
for (i=0 ; i<1000 ; ++i)
{   tab[i] = foo(i);
}
```

To check that the variable `i` never overflows the range of `tab`, a traditional approach would be to enumerate each possible value of `i`. One thousand checks would be required.

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

An approximation, by definition, leads to information loss. For instance, the information that `i` is incremented by one every cycle in the loop is lost. However, the important fact is that this information is not required to ensure that range errors will not occur; it is only necessary to prove that the variation domain of `i` is smaller than the range of `tab`. Only one check is required to establish that – and hence the gain in efficiency compared to traditional approaches.

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

Exhaustiveness

Nothing is lost in terms of exhaustiveness. The reason is that Polyspace verification works by performing upper approximations. In other words, the computed variation domain of a program variable is a superset of its actual variation domain. As a result, Polyspace verifies run-time error items that require checking.

Approximations Made by Polyspace Verification

In this section...

“Volatile Variables” on page 5-4
“Structures with Volatile Fields” on page 5-4
“Absolute Addresses” on page 5-5
“Pointer Comparison” on page 5-5
“Shared Variables” on page 5-5
“Trigonometric Functions” on page 5-6
“Unions” on page 5-6
“Constant Pointer” on page 5-7
“Variable Cast as Void Pointer” on page 5-7

Volatile Variables

Volatile variables are potentially uninitialized and their content is full range.

```
2 int volatile_test (void)
3 {
4   volatile int tmp;
5   return(tmp); // NIV orange: the variable content is full range
6   [-2^31;2^31-1]
6 }
```

In the case of a global variable the content would also be full range, but the NIV check would be green.

Structures with Volatile Fields

In this example, although only the `b` field is declared as volatile, in practice, a read access to the `a` field is full range and orange.

```
2 typedef struct {
3   int a;
4   volatile int b;
5 } Vol_Struct;
```

Absolute Addresses

Both reading from, and writing to, an absolute address leads to warning checks on the pointer dereference. An absolute address is considered as a volatile variable.

```
Val = *((char *) 0x0F00); // NIV and IDP orange: access to an
absolute address
```

Pointer Comparison

Polyspace verification is a static tool verifying source code. Memory management concerns dynamic considerations, and the characteristics of particular compilers and targets. Polyspace verification therefore doesn't consider where objects are actually implanted in memory

```
5 int *i, *j, k;
6 i = (int *) 0x0F00;
7 j = (int *) 0x0FF0;
8
9 if ( i < j) // the condition can be true or false
10 k = 12; // this line is reachable
11 else
12 k = 23; // this line is reachable too.
```

Its the same situation if “i” and “j” points to real variable

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

Shared Variables

At a minimum, the range of a shared variable is the union of all ranges of the variable in the application. At a maximum, the variable is full range.

```
12 void p_task1(void)
13 {
14 begin_cs();
15 X = 0;
16 if (X) {
17 Y = X; // Verified NIV, although it should be gray
18 assert (Y == 12); // Warning assert, although it should be gray
19 }
```

```
20 end_cs();
21 }
22
23 void p_task2(void)
24 {
25     begin_cs();
26     X = 12;
27     Y = X + 1;      // Polyspace considers [Y==1] or [Y==13]
28     if (Y == 13)
29         Y = 14;
30     else
31         Y = X - 1 ;    // this line should be gray
32     end_cs();
33 }
```

Trigonometric Functions

With trigonometric functions, such as sines and cosines, verification sometimes assumes that the return value is bound between the limits of that function, regardless of the parameter passed to it. Consider the following example, which uses `acos`, `sin` and `asin` functions.

```
7 double res;
8
9 res = sin(3.141592654);
10 assert(res == 0.0); // Range is [-1..1]
11
12 res = acos(0.0);
13 assert(res == 0.0); // Range always in [0..pi]
14
15 res = asin(0.0);
16 assert(res == 0.0); // Always gives [0.0]
```

Unions

In some situations, unions can help you construct efficient code. However, unions can cause issues for code verification, for example:

- **Padding** – Padding might be inserted at the end of an union.
- **Alignment** – Members of structures within a union might have different alignments.
- **Endianness** – Whether the most significant byte of a word could be stored at the lowest or highest memory address.

- **Bit-order** – Bits within bytes could have both different numbering and allocation to bit fields.

These issues can cause Polyspace verification to lose precision when structure unions are considered. In fact, these kinds of implementation are compiler dependant. Conversions from one type a union to another will cause a loss of precision on two checks:

- Is the other field initialized? Orange NIV
- What is the content of the other field? Full range

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

```
X.b[0] = 1; X.b[1] = 1; X.b[2] = 1; X.b[1] = 1;
if (X.A == 0x1111)
else // both branches are reachable
```

Constant Pointer

To increase Polyspace precision where pointers are analyzed, replace

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

with:

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

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 the following code:

```
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` should be assigned the value 5. However, the software treats `p->x1` as full-range.

In some cases, you can avoid this loss of precision by running your verification with the option `-retype-pointer`. For this example, if you specify `-retype-pointer`, the software assigns the value 5 to `x` in the function `g`.

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:

`MATLAB_Install\polyspace\verifier\code_prover`

Examples

Complete Examples

In this section...

“Simple C Example” on page 6-2
“Apache Example” on page 6-2
“exref Example” on page 6-3
“T31 Example” on page 6-3
“Dishwasher1 Example” on page 6-3
“Satellite Example” on page 6-4

Simple C Example

```
polyspace-code-prover-nodesktop \  
-prog myCproject \  
-O1 \  
-I /home/user/includes \  
-D SUN4 -D USE_FILES \  

```

Apache Example

Here is a script for verifying the code for Apache (after formatting). The source code is in C and the compilation is for an Oracle® Sun™ Microsystems SPARC® processor.

Note: The use of O0 to reduce verification time.

```
polyspace-code-prover-nodesktop \ \  
-target sparc \  
-prog Apache \  
-keep-all-files \  
-continue-with-red-error \  
-O0 \  
-D PST \  
-D __GNUC_MINOR__=6 -D SOLARIS2=270 -D USE_EXPAT \  
-D NO_DL_NEEDED \  
-I sources \  
-I /usr/local/pst/include.sparc \  

```

```
-I /usr/include \
-results-dir RESULTS
```

cxref Example

Here is another C launch command. The compilation is for Linux. Note the escape characters, allowing quoted strings to be used as compiler defines.

```
polyspace-code-prover-nodesktop \
  -OS-target linux \
  -prog cxref \
  -OO \
  -I `pwd` \
  -I sources \
  -I <Polyspace_Install>/include/include.linux \
  -D CXREF_CPP="'\"/usr/local/gcc/bin/cpp\"' ' ' \
  -D PAGE="'\"A4\"' ' ' \
  -results-dir RESULTS
```

T31 Example

Another simple C launcher. There are a couple of tasks and compilation is for an m68k.

```
polyspace-code-prover-nodesktop \
  -target m68k \
  -entry-points task_callback_main,task_tcp_main,cdtask_depm_main,
task_receiver \
  -to pass1 \
  -prog T31 \
  -OO \
  -results-dir `pwd`/RESULTS_31 \
  -keep-all-files
```

Dishwasher1 Example

Another C example. This one is for the c-167 and has tasks protected by critical section.

```
polyspace-code-prover-nodesktop \
  -target c-167 \
  -entry-points periodic,pst_main \
  -D PST -D const= -D water= \
  -from scratch \
```

```
-to pass4 \  
-critical-section-begin "critical_enter:cs1" \  
-critical-section-end "critical_exit:cs1" \  
-prog dishwasher1 \  
-I `pwd`/sources \  
-O0 \  
-keep-all-files \  
-results-dir RESULTS
```

Satellite Example

A C example with tasks and critical sections.

```
polyspace-code-prover-nodesktop  
-target c-167 \  
-entry-points ctask0,ctask1,ctask2,ctask3,interrupts \  
-O2 \  
-keep-all-files \  
-from scratch \  
-critical-section-begin "DisableInterrupts:sc1" \  
-critical-section-end "EnableInterrupts:sc1" \  
-ignore-constant-overflows \  
-include `pwd`/sources/options.h \  
-to pass4 \  
-prog satellite \  
-I `pwd`/sources \  
-results-dir RESULTS
```

Functions

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 priority classification, an action status, or other comments using `Name,Value` pairs.

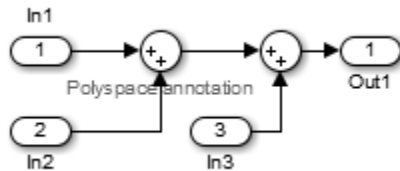
In the generated code associated with the annotated block, Polyspace adds code comments before and after the lines of code. Polyspace reads these comments and marks Polyspace results of the specified `kind` with the annotated information.

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:Fix>*/
annotate_y.Out1=(annotate_u.In1+annotate_u.In2)+annotate_u.In3;
/* polyspace:end<RTE:OVFL:Medium: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. If analysis results do not exist for `systemName`, Polyspace opens to the Project Manager perspective.

`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_v2`, 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_v2';  
open(model)  
gcb = 'WhereAreTheErrors_v2/10* x // (x-y)/Divide';  
pslinkfun('annotations','type','RTE','kind','ZDV','status',...  
         'justify with annotation','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_v2` and open the advanced options window.

```
model = 'WhereAreTheErrors_v2';
```



```
load_system(model)
pslinkfun('advancedoptions')
```

In the **Distributed Computing** pane, select the options **Batch** and **Add to results repository**.

Set the configuration template for new Polyspace analyses to have these options.

```
pslinkfun('settemplate',fullfile(cd,'pslink_config',...
    'WhereAreTheErrors_v2_config.psprj'))
```

View the current Polyspace template.

```
template = pslinkfun('gettemplate')

template =
C:\ModelLinkDemo\pslink_config\WhereAreTheErrors_v2_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_v2`, create a Polyspace options object, set the verification mode, and open the advanced options window.

```
model = 'WhereAreTheErrors_v2';
load_system(model)
slbuild(model)
opts = pslinkoptions(model);
opts.VerificationMode = 'CodeProver';
pslinkfun('advancedoptions')
```

In the **Distributed Computing** pane, select the **Batch** and **Add to results repository** options.

Run Polyspace, then Queue Manager 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 Check Reference”.
'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 Rules”.
'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 Rules”.

type Value	kind Values
'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_v2')`

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','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' — classification of the check

`'high' | 'medium' | 'low' | 'not a defect' | 'unset'`

Classification of the check specified as `high`, `medium`, `low`, `not a defect`, or `unset`.

Example: `'class', 'high'`

'status' — action status

`'undecided' | 'investigate' | 'fix' | 'improve' | 'restart with different options' | 'justify with annotation' | 'no action planned' | 'other'`

Action status of the check specified as `undecided`, `investigate`, `fix`, `improve`, `restart with different options`, `justify with annotation`, `no action planned`, or `other`.

The statuses, `justify with annotation` and `no action planned`, also mark the result as justified.

Example: `'status', 'no action planned'`

'comment' — additional comments

string

Additional comments specified as a string. The comments provide more information about why the results are justified.

Example: `'comment', 'defensive code'`

See Also

`pslinkrun` | `pslinkoptions` | `gcb`

pslinkoptions

Create options object to customize Polyspace runs from MATLAB command line

Syntax

```
opts = pslinkoptions(codegen)
opts = pslinkoptions(model)
```

Description

`opts = pslinkoptions(codegen)` returns an options object with the configuration options for code generated by codegen.

`opts = pslinkoptions(model)` returns an options object with the configuration options for the Simulink model.

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_v2');
model_opt = pslinkoptions('psdemo_model_link_sl_v2')

model_opt =
```

```

        ResultDir: 'results_$ModelName$'
VerificationSettings: 'PrjConfig'
  OpenProjectManager: 0
AddSuffixToResultDir: 0
EnableAdditionalFileList: 0
  AdditionalFileList: {}
  VerificationMode: 'CodeProver'
EnablePrjConfigFile: 0
  PrjConfigFile: ''
  InputRangeMode: 'DesignMinMax'
  ParamRangeMode: 'None'
```

```
        OutputRangeMode: 'None'  
        ModelRefVerifDepth: 'Current model only'  
    ModelRefByModelRefVerif: 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: ''  
        InputRangeMode: 'DesignMinMax'  
        ParamRangeMode: 'None'  
        OutputRangeMode: 'None'  
        ModelRefVerifDepth: 'Current model only'  
    ModelRefByModelRefVerif: 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: ''
        InputRangeMode: 'DesignMinMax'
        ParamRangeMode: 'None'
        OutputRangeMode: 'None'
        ModelRefVerifDepth: 'Current model only'
    ModelRefByModelRefVerif: 0
    CxxVerificationSettings: 'PrjConfig'
    CheckConfigBeforeAnalysis: 'OnWarn'

```

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: ''
        InputRangeMode: 'DesignMinMax'
        ParamRangeMode: 'None'
        OutputRangeMode: 'None'
        ModelRefVerifDepth: 'Current model only'
    ModelRefByModelRefVerif: 0
    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: ''
  InputRangeMode: 'DesignMinMax'
  ParamRangeMode: 'None'
  OutputRangeMode: 'None'
  AutoStubLUT: 0
```

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: ''
  InputRangeMode: 'DesignMinMax'
  ParamRangeMode: 'None'
  OutputRangeMode: 'None'
  AutoStubLUT: 0
```

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 Properties](#).

```
Example: ec_opt = pslinkoptions('ec')
```

```
Example: tl_opt = pslinkoptions('tl')
```

Data Types: char

model — Simulink model

model name

Simulink model, specified by the model name. Creates a Polyspace options object with the configuration options of that model. If you do not set any options, the object has the default configuration options. If a code generator has been set, the object has the default options for that code generator.

For a description of all configuration options and their values, see [pslinkoptions Properties](#).

```
Example: model_opt = pslinkoptions('my_model')
```

Data Types: char

Output Arguments

opts — Polyspace configuration options

options object

Polyspace configuration options, returned as an options object. The object is used with `pslinkrun` to run Polyspace from the MATLAB command line.

For the list of object properties, see [pslinkoptions Properties](#).

```
Example: opts= pslinkoptions('ec')  
opts.VerificationSettings = 'Misra'
```

More About

- `pslinkoptions` Properties

See Also

`pslinkfun` | `pslinkrun`

pslinkrun

Run Polyspace analysis on generated code from MATLAB command line

Syntax

```
resultsFolder = pslinkrun
resultsFolder = pslinkrun(system)
resultsFolder = pslinkrun(system,opts)
resultsFolder = pslinkrun(system,opts,asModelRef)
```

Description

`resultsFolder = pslinkrun` on generated code from the current system and returns the location of the results folder. It uses the analysis options associated with the current system. The current system, or model, is the system returned by the command `bdroot`.

`resultsFolder = pslinkrun(system)` runs Polyspace on the code generated from the model or subsystem specified by `system`. It uses the analysis options associated with `system`.

`resultsFolder = pslinkrun(system,opts)` analyzes `system` using the analysis options from the options object `opts`.

`resultsFolder = pslinkrun(system,opts,asModelRef)` uses `asModelRef` to specify which type of generated code to analyze, standalone code or model reference code. This option is useful when you want to analyze only a referenced model instead of an entire model hierarchy.

Examples

Run Polyspace from the Command Line

Use a Simulink model to generate code, set configuration options, and then run an analysis from the command line.

Load and build the model `WhereAreTheErrors_v2` to generate code.

```
model = 'WhereAreTheErrors_v2';
```

```
load_system(model)
slbuild(model)
```

Create a Polyspace options object from the model and change the configuration to run a Code Prover verification.

```
opts = pslinkoptions(model);
opts.VerificationMode = 'CodeProver';
```

Run Polyspace using your options object:

```
results = pslinkrun(model,opts)
```

The results are saved to the `results_WhereAreTheErrors_v2` folder, listed in the `results` variable.

Build and Analyze Referenced Model Code from the Command Line

Use a Simulink model to generate reference code, set configuration options, and then run an analysis from the command line.

Load and build the model `WhereAreTheErrors_v2` to generate code as if it is referenced by another model:

```
model = 'WhereAreTheErrors_v2';
load_system(model)
slbuild(model, 'ModelReferenceRTWTargetOnly')
```

Create a Polyspace options object from the model and change the configuration to run a Code Prover verification.

```
opts = pslinkoptions(model);
opts.VerificationMode = 'CodeProver';
```

Run Polyspace using your options object:

```
results = pslinkrun(model,opts,true)
```

The results are saved to the `results_mr_WhereAreTheErrors_v2` folder, listed in the `results` variable.

Input Arguments

system — Model or system

bdroot (default) | model or system name

Model or system that you want to analyze, specified as a string, with the model or system name in single quotes. The default value is the system returned by `bdroot`.

Example: `resultsFolder = pslinkrun('demo')` where `demo` is the name of a model.

Data Types: `char`

opts — Analysis options

options associated with system (default) | Polyspace options object

Analysis options for the analysis, specified as an options object or the options already associated with the model or system. The function `pslinkoptions` creates an options object. You can customize the options object by changing the

Example: `pslinkrun('demo', opts_demo)` where `demo` is the name of a model and `opts_demo` is an options object.

asModelRef — Indicator for model reference analysis

false (default) | true

Indicator for model reference analysis, specified as true or false.

- If `asModelRef` is false (default), Polyspace analyzes code generated as standalone code. This option is equivalent to choosing **Verify Code Generated For > Model** in the Simulink Polyspace options.
- If `asModelRef` is true, Polyspace analyzes code generated as model referenced code. This option is equivalent to choosing **Verify Code Generated For > Referenced Model** in the Simulink Polyspace options.

Data Types: `logical`

Output Arguments

resultsFolder — Variable for location of the results folder

string

Variable for location of the results folder, specified as a string. The default value of this variable is `results_$(modelName)`. You can change this value in the configuration options using `pslinkoptions`.

Data Types: `char`

See Also

pslinkfun | pslinkoptions

polyspaceCodeProver

Run Polyspace Code Prover verification from MATLAB

Syntax

```
polyspaceCodeProver
```

```
polyspaceCodeProver(projectFile)
```

```
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(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, you open the project file `Demo_C.psprj` from the folder `Matlab_Install\polyspace\examples\cxx\Demo_C`.

Assign the full path to the project file to a MATLAB variable `prjFile`.

```
prjFile = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', ...  
                  'Demo_C', 'Demo_C.psprj');
```

Use `prjFile` to 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 `Matlab_Install\polyspace\examples\cxx\Demo_C\Module_1\Result_1`.

Assign the full path to the folder to a MATLAB variable `resFolder`.

```
resFolder = fullfile(matlabroot, 'polyspace', 'examples', ...  
                    'cxx', 'Demo_C', 'Module_1', 'Result_1');
```

Use `resFolder` to open the results.

```
polyspaceCodeProver('-results-dir', resFolder)
```

Run Polyspace Verification from MATLAB

This example shows how to run a Polyspace verification on a single source file from the MATLAB command-line. 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 following command on the MATLAB command line.

```
polyspaceCodeProver('-sources', 'C:\Polyspace_Sources\source.c', ...
```



```
'-I', 'C:\Polyspace_Includes', ...
'-results-dir', 'C:\Polyspace_Results')
```

Polyspace runs on the file `C:\Polyspace_Sources\source.c` and stores the result in `C:\Polyspace_Results`.

To view the results from the MATLAB command line, enter:

```
polyspaceCodeProver('-results-dir', 'C:\')
```

Run Polyspace Verification with Coding Rules Checking

This example shows how to run a Polyspace verification with additional options. You can specify as many additional options as you want as “Name-Value Pair Arguments” on page 7-23. Here you specify:

- Checking of MISRA C coding rules using the option `-misra2`. For more information, see “Check MISRA C:2004”.
- Excluding header files from coding rules checking using the option `-includes-to-ignore`. For more information, see “Files and folders to ignore (C)”.
- Automatic generation of `main` function using the option `-main-generator`. For more information, see “Verify module (C)”.

Assign the source file path to a MATLAB variable `sourceFileName`.

```
sourceFileName = fullfile(matlabroot, 'polyspace', ...
'examples', 'cxx', 'Demo_C_Single-File', 'sources', 'example.c')
```

Assign the include file path to a MATLAB variable `includeFileName`.

```
includeFileName = fullfile(matlabroot, 'polyspace', ...
'examples', 'cxx', 'Demo_C_Single-File', 'sources', 'include.h')
```

Assign the results folder path to a MATLAB variable `resFolder`.

```
resFolder = fullfile('C:\', 'Polyspace_Results')
```

Run Polyspace Code Prover verification with additional options `-misra2`, `-includes-to-ignore` and `-main-generator`.

```
polyspaceCodeProver('-sources', sourceFileName, ...
'-I', includeFileName, ...
'-results-dir', resFolder, '-misra2', 'required-rules', ...
'-includes-to-ignore', 'all-headers', '-main-generator')
```

Open the results file.

```
polyspaceCodeProver('-results-dir',resFolder)
```

- “Specify Options from MATLAB Command Line”

Input Arguments

projectFile — Name of .psprj file

string

Name of project file with extension `.psprj`, specified as a string.

If the file is not in the current folder, `projectFile` must include a full or relative path. Use `pwd` to identify the current folder and `cd` to change the current folder.

Example: `'C:\Polyspace_Projects\myProject.psprj'`

resultsFile — Name of .pscp file

string

Name of results file with extension `.pscp`, specified as a string.

If the file is not in the current folder, `resultsFile` must include a full or relative path. Use `pwd` to identify the current folder and `cd` to change the current folder.

Example: `'myResults.psbf'`

resultsFolder — Name of result folder

string

Name of result folder, specified as a string. The folder must contain the results file with extension `.psbf`. 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. Use `pwd` to identify the current folder and `cd` to change the current folder.

Example: `'C:\Polyspace\Results\'`

sourceFiles — Comma-separated names of .c or .cpp files

string

Comma-separated source file names with extension `.c` or `.cpp`, specified as a single string.

If the files are not in the current folder, `sourceFiles` must include a full or relative path. Use `pwd` to identify the current folder and `cd` to change the current folder.

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','-dialect','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 dialect.

- For options that can also be set from the user interface, see the **Command-Line Information** section in:
 - “Analysis Options for C Code”
 - “Analysis Options for C++ Code”
- For options that cannot be set from the user interface, see the **Polyspace Analysis Options** section in “Command-Line Verification”.

polyspaceConfigure

Create Polyspace project from your build system

Syntax

```
polyspaceConfigure buildCommand
```

```
polyspaceConfigure buildCommand -option value
```

Description

`polyspaceConfigure buildCommand` traces your build system and creates a Polyspace project with information gathered from your build system.

`polyspaceConfigure buildCommand -option value` traces your build system and uses the flag `-option value` to modify the default operation of `polyspaceConfigure`.

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` 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` 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
```

Trace Incremental Makefile Builds

This example shows how to trace incremental makefile builds to keep your Polyspace project updated. If you use this approach, `polyspaceConfigure` does not have to trace the entire makefile every time you make a change to it.

Create a Polyspace project from your makefile using `polyspaceConfigure`. For this first project creation:

- Use the `-B` option with `make` so that all prerequisite targets in the makefile are remade.
- Use the `-incremental` option so that the build trace information is saved.

```
polyspaceConfigure -prog myProject ...
                   -incremental make -B targetName buildOptions
```

After you add, remove or change source files, to keep your Polyspace project updated, rerun `polyspaceConfigure` with the same options. Do not use the `-B` option with `make`.

```
polyspaceConfigure -prog myProject ...
                   -incremental make targetName buildOptions
```

The `polyspaceConfigure` function uses the previous build trace information to incrementally add or remove the updated files to your Polyspace project. It does not trace the entire makefile.

- “Create Projects Automatically from Your Build System”

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`

-option value — Options for changing default operation of `polyspaceConfigure`

single option starting with -, followed by argument | multiple space-separated option-argument pairs

Option	Argument	Description
-author	Author name	Name of project author. Example: <code>-author jsmith</code>
-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>
-cache-all-files	None	Option to cache all files read by <code>polyspaceConfigure</code> including binaries
-cache-path	Path	Location of folder where cache information is stored. Example: <code>-cache-path ../cache</code>
-compiler-configuration	Path and file name	Location and name of compiler configuration file. 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 “Your Compiler Is Not Supported”.

Option	Argument	Description
		Example: -compiler-configuration myCompiler.xml
-debug	None	Option used by MathWorks technical support
-help	None	Option to display the full list of polyspaceConfigure commands
-incremental	None	Option to save build trace information for reuse in incremental builds
-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 polyspaceConfigure with the -no-project option. If you use this option, you do not need to specify the buildCommand argument.
-no-cache	None	Option to specify that a cache of your files must not be created.
-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 polyspaceConfigure with the -no-build option.
-output-dump-file	None	Option to save build trace information in a text file.
-output-options-file	None	Option to create a Polyspace analysis options file. Use this file for command-line analysis using polyspaceCodeProver.

Option	Argument	Description
-output-project	Path	Location for saving Polyspace project. The default is the current folder. Example: -output-project ../myProjects/
-prog	Project name	Name of project. The default is polyspace.psprj. Example: -output-project myProject

More About

- “Requirements for Project Creation from Build Systems”
- “Your Compiler Is Not Supported”

polyspaceJobsManager

Manage Polyspace jobs on MDCS 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, '-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, '-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', ...  
    'Demo_C', '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')
```

```
ID AUTHOR APPLICATION LOCAL_RESULTS_DIR WORKER STATUS DATE LANG CLUSTER_MODE  
...  
19 user Polyspace C:\psdemo\res1  queued Wed Mar 16 16:48:38 EST 2014 C Batch  
20 user Polyspace C:\psdemo\res2  queued Wed Mar 16 16:48:38 EST 2014 C Batch
```

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')
```

```
ID AUTHOR APPLICATION LOCAL_RESULTS_DIR WORKER STATUS DATE LANG CLUSTER_MODE
...
19 user Polyspace C:\psdemo\res1 cancelled Wed Mar 16 16:48:38 EST 2014 C Batch
20 user Polyspace C:\psdemo\res2 running Wed Mar 16 16:48:38 EST 2014 C Batch
```

Remove job 19.

```
polyspaceJobsManager('remove', '-job', '19', '-scheduler', ...
    'myMJS@myCompany.com')
polyspaceJobsManager('listjobs', '-scheduler', 'myMJS@myCompany.com')
```

```
ID AUTHOR APPLICATION LOCAL_RESULTS_DIR WORKER STATUS DATE LANG CLUSTER_MODE
...
20 user Polyspace C:\psdemo\res2 completed Wed Mar 16 16:48:38 EST 2014 C Batch
```

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

string

Number of the queued job that you want to manage, specified as a string in single quotes.

Example: '-job', '10'

resultsFolder — Path to results folder

string

Path to results folder specified as a string in single quotes. This folder stores the downloaded results files.

Example: `'-results-folder', 'C:\psdemo\myresults'`

scheduler — job scheduler

head node of your MDCS 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 MDCS 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'`

More About

- “Clusters and Cluster Profiles”
- “Manage Remote Analyses at the Command Line”

See Also

polyspaceCodeProver

PolyspaceAnnotation

Annotate Simulink blocks with known Polyspace results

Compatibility

PolyspaceAnnotation will be removed in a future release. Use “`pslinkfun('annotations',...)`” instead.

Syntax

`PolyspaceAnnotation('type',typeValue,'kind',kindValue,Name,Value)`

Description

`PolyspaceAnnotation('type',typeValue,'kind',kindValue,Name,Value)` adds an annotation of type `typeValue` and kind `kindValue` to the currently selected block in the model. You can also specify a different block using a `Name,Value` pair argument. You can also add notes about a priority classification, an action status, or other comments using `Name,Value` pairs.

In the generated code associated with the annotated block, code comments are added before and after the lines of code. Polyspace reads these comments and marks Polyspace results of the specified `kind` with the annotated information.

When you add annotations, you can identify known errors and coding rule violations to focus on new results.

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.

At the MATLAB command line, load and open the example model
WhereAreTheErrors_v2:

```
open(WhereAreTheErrors_v2)
```

Set the current block to the division block of the $10^* x // (x-y)$ subsystem:

```
gcb = 'WhereAreTheErrors_v2/10* x // (x-y)/Divide';
```

Add an annotation to the current block to mark division by zero (DIV) errors as justified with the annotation.

```
PolyspaceAnnotation('type','RTE','kind','ZDV','status',...  
'justify with annotation','comment','verified not an error')
```

In Simulink, the division block of the $10^* x // (x-y)$ subsystem now has a Polyspace annotation.

Back at the MATLAB command line, generate code for the model:

```
slbuild('WhereAreTheErrors_v2')
```

Run a Polyspace Code Prover verification on your model:

```
pslinkrun('WhereAreTheErrors_v2')
```

After the analysis has finished, open the result in the Polyspace environment:

```
PolySpaceViewer('WhereAreTheErrors_v2')
```

If you look at orange division by zero error, the check is justified and includes the status and comments from your annotation.

Annotate a Block and Run a Polyspace Bug Finder Analysis

Use the Polyspace annotation function to annotate a block and see the annotation in the analysis results.

At the MATLAB command line, load and open the example model
WhereAreTheErrors_v2:

```
WhereAreTheErrors_v2
```

Add an annotation to the switch block to annotate violations to MISRA C rule 13.7. Also, add to the annotation a comment, a classification, and a status.

```
PolyspaceAnnotation('type','Misra-C', 'kind', '13.7','block',...
'WhereAreTheErrors_v2/Switch1', 'status','improve', 'comment', 'look into later');
```

In the WhereAreTheErrors_v2 model in Simulink, you can see a Polyspace annotation added to the switch block.

At the MATLAB command line, generate code for the model:

```
slbuild('WhereAreTheErrors_v2')
```

Run an analysis on your model:

```
pslinkrun('WhereAreTheErrors_v2')
```

After the analysis is finished, open the results in the Polyspace environment:

```
PolySpaceViewer('WhereAreTheErrors_v2')
```

Results 10–14 are MISRA C 13.7 rule violations. The annotation information that you added to the switch block appears in these four results, because all four results are from the switch block.

Input Arguments

typeValue — type of result

'RTE' | 'MISRA-C' | '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-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 Check Reference”.
'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 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: `PolyspaceAnnotation('type','MISRA-CPP','kind','1-2-3')`

Data Types: char

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'block','MyModel\Sum','status','fix'`

'block' — **block to be annotated**

gcb (default) | block name

Block to be annotated specified by the block name. If you do not use this option, the block returned by the function `gcb` is annotated.

Example: `'block', 'MyModel\Sum'`

'class' — classification of the check

`'high' | 'medium' | 'low' | 'not a defect' | 'unset'`

Classification of the check specified as `high`, `medium`, `low`, `not a defect`, or `unset`.

Example: `'class', 'high'`

'status' — action status

`'undecided' | 'investigate' | 'fix' | 'improve' | 'restart with different options' | 'justify with annotation' | 'no action planned' | 'other'`

Action status of the check specified as `undecided`, `investigate`, `fix`, `improve`, `restart with different options`, `justify with annotation`, `no action planned`, or `other`.

The statuses, `justify with annotation` and `no action planned`, also mark the result as justified.

Example: `'status', 'no action planned'`

'comment' — additional comments

`string`

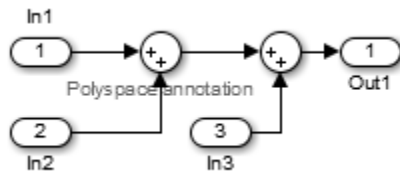
Additional comments specified as a string. The comments provide more information about why the results are justified.

Example: `'comment', 'defensive code'`

Limitations

- You can have only one annotation per block. If a block produces both a rule violation and an error, only one type can be annotation.
- Even though you apply annotations to individual blocks, the scope of the annotation may be larger. The generated code from one block can overlap with another causing the annotation to also overlap.

For example, consider this model and its associated generated code.



```

/*
 * polyspace:begin<RTE:OVFL:Medium:Fix>
 */
annotate_y.Out1 = (annotate_u.In1 + annotate_U.In2) + annotate_U.In3;

/* polyspace:end<RTE:OVFL:Medium:Fix> */

```

The first summation block has a Polyspace annotation, but the second does not. However, the associated generated code adds all three inputs in one line of code. Therefore, the annotation justifies both summations

See Also

[pslinkoptions](#) | [pslinkrun](#) | [PolySpaceViewer](#) | [gcb](#)

PolySpaceViewer

Open analysis results in the Polyspace environment

Compatibility

PolySpaceViewer will be removed in a future release. Use “`pslinkfun('openresults',...)`” instead.

Syntax

```
PolySpaceViewer(system)
```

Description

`PolySpaceViewer(system)` opens the Polyspace results associated with the model or subsystem `system` in the Polyspace environment. If `system` has not been analyzed, Polyspace opens to the Project Manager perspective.

Examples

Open Results in the Polyspace environment from the Command Line

Use the preconfigured model `WhereAreTheErrors_v2` to run a Polyspace analysis and open the results in the Polyspace environment.

Load the model `WhereAreTheErrors_v2`:

```
load_system('WhereAreTheErrors_v2')
```

Open the Polyspace Viewer:

```
PolySpaceViewer('WhereAreTheErrors_v2')
```

The Polyspace environment opens to the Project Manager page because the model does not yet have Polyspace results.

Build the model to generate C code:

```
slbuild('WhereAreTheErrors_v2');
```

Create a Polyspace options object to set the configuration options:

```
config = pslinkoptions('WhereAreTheErrors_v2')
config =
    ResultDir: 'results_$ModelName$'
    VerificationSettings: 'PrjConfig'
    OpenProjectManager: 0
    AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
    AdditionalFileList: {0x1 cell}
    InputRangeMode: 'DesignMinMax'
    ParamRangeMode: 'None'
    OutputRangeMode: 'None'
    VerificationMode: 'CodeProver'
    ModelRefVerifDepth: 'Current model only'
    ModelRefByModelRefVerif: 0
    CxxVerificationSettings: 'PrjConfig'
```

Change the analysis options to also check for MISRA coding rule violations:

```
config.VerificationSettings = 'PrjConfigAndMisra';
```

Run Polyspace on WhereAreTheErrors_v2 using the configuration options object that you created:

```
pslinkrun('WhereAreTheErrors_v2', config);
```

Open the results in the Polyspace environment:

```
PolySpaceViewer('WhereAreTheErrors_v2');
```

The analysis results of WhereAreTheErrors_v2 appear in the Polyspace Results Manager.

Input Arguments

system — Simulink model

system | subsystem

Simulink model specified by the system or subsystem name.

Example: `PolySpaceViewer('myModel')`

See Also

`pslinkoptions` | `pslinkrun` | `PolyspaceAnnotation`

pslinkoptions Properties

Properties for the `pslinkoptions` object

Before running Polyspace from the command-line, use these properties to customize your analysis.

Analysis Configuration

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' – Use all options from the project configuration.
- 'PrjConfigAndMisraAGC' – Use all options from the project configuration and enable MISRA AC AGC rule checking.
- 'PrjConfigAndMisra' – Use all options from the project configuration and enable MISRA C:2004 rule checking.
- 'PrjConfigAndMisraC2012' – Use all 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 .psprj 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 verification 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 results. 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 switch to the Results Manager perspective to review the results.

Example: `opt.OpenProjectManager = 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 use 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 all options from project configuration and run complete analysis.
- 'PrjConfigAndMisraCxx' – Inherit all options from project configuration, enable MISRA C++ rule checking, and run complete analysis.
- 'PrjConfigAndJSF' – Inherit all options from project configuration, enable JSF rule checking, and run complete analysis.
- 'MisraCxx' – Enable MISRA C++ rule checking, and run compilation phase only.
- '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`, to use Lookup Table code during the analysis, or `false`, to not.

Only for TargetLink

```
Example: opts.AutoStubLUT = true;
```

See Also

`pslinkoptions` | `pslinkrun`