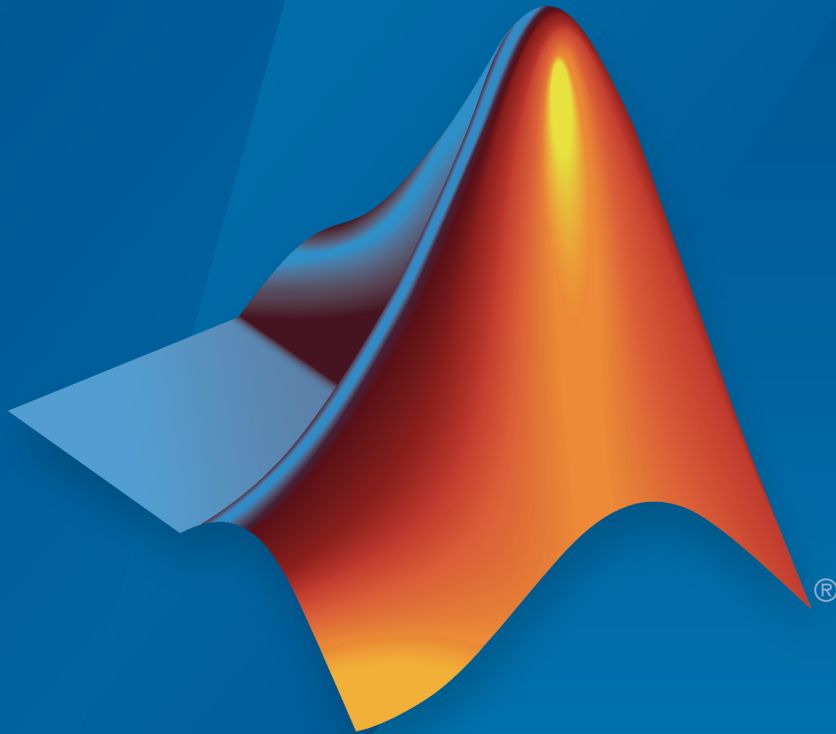


Polyspace[®] Code Prover[™] Release Notes



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[™] Release Notes

© COPYRIGHT 2013–2016 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.

Files to Review: Generate results for only specified files and folders	1-2
Autocompletion for Review Comments: Partially type previous comment to select complete comment	1-2
Faster MISRA Rule Checking: Check coding rules more quickly and efficiently	1-3
S-Function Analysis: Launch analysis of S-Function code from Simulink	1-3
Persistent Filter States: Apply filters once and view filtered results across multiple runs	1-3
Floating-Point Support: Propagate ranges more precisely for long double variables and enable verification mode to incorporate infinities and NaNs	1-4
Long Doubles	1-4
Nonfinites in floating-point arithmetic	1-5
Rounding modes	1-6
Polyspace Metrics Tomcat Upgrade: Use upgraded default Tomcat server or custom Tomcat version	1-7
Updated Polyspace Metrics Interface: View summary of project and metrics	1-7
Project Language Flexibility: Change your project language at any time	1-7

External Constraint on Pointers: Specify certain initialization with full range for pointer arguments and return values of stubbed functions	1-8
Improved Result Display for File-by-File Verification: View combined summary of results for all files in user interface	1-9
Source Code Search: Search large applications more quickly	1-10
Default Layouts: Switch easily between project setup and results review in user interface	1-10
Simplified Variable Access: View task names instead of aliases	1-10
Polyspace TargetLink plug-in supports data from structures	1-10
Polyspace Eclipse plug-in results location moved	1-10
Improvements in automatic project creation from build command	1-11
Improvements in checking of previously supported MISRA C rules	1-12
MISRA C:2004 Rules	1-12
MISRA C:2012 Rules	1-12
Absolute address usage valid by default	1-13
Variables with constraints not counted as orange sources	1-13
Changes in analysis options	1-14
Run-time checks renamed	1-16

Option to Suppress Non-initialization Checks: Customize verification by suppressing non-initialization checks . . .	2-2
Improved Concurrency Detection: View more precise sharing and protection results based on dynamic information such as data flow in branching statements and protection on individual fields of a structure	2-2
Data Flow in Branch Statements	2-2
Shared Structures	2-3
Microsoft Visual C++ 2013 Support: Analyze code developed in Microsoft Visual C++ 2013	2-5
Additional MISRA C:2012 Support: Detect violations of all MISRA C:2012 rules except rules 22.x	2-5
GNU 4.9 and Clang 3.5 Support: Analyze code compiled with GCC 4.9 or Clang 3.5	2-6
Improved Review Capability: View result details and add review comments in one window	2-6
Saved Layouts: Save your preferred layouts of the Polyspace user interface	2-6
Autodetection of Multitasking Primitives: Analyze source code with multitasking primitives from POSIX or VxWorks without manual setup	2-7
Start Page: Get quickly familiar with Polyspace Code Prover	2-7
Enhanced Review Scope: Filter coding rule violations from display in one click	2-8
Additional Call Graph Showing Task Creation	2-8
Improved precision for mathematical functions	2-9

Improved handling of <code>__declspec</code>	2-9
Improvements in Polyspace Metrics workflow	2-9
Updated Support for TargetLink	2-10
Improvements in Polyspace Plugin for Eclipse	2-10
Improvements in automatic project creation from build command	2-11
Improvements in checking of previously supported MISRA C rules	2-12
MISRA C:2004 Rules	2-12
MISRA C:2012 Rules	2-13
Checking Coding Rules Using Text Files	2-14
Including options multiple times	2-14
Renaming of labels in Polyspace user interface	2-15
Configuration Associated with Result Not Opened by Default	2-16
Improvements in Report Templates	2-16
Changes in analysis options	2-16
Change in Correctness Condition Check	2-21
Binaries removed	2-22
Support for Visual Studio 2008 to be removed	2-22
Import Visual Studio project removed	2-23
XML and RTF report formats removed	2-23

Simplified workflow for project setup and results review with a unified user interface	3-2
Review of code complexity metrics and global variable usage in user interface	3-3
Code Complexity Metrics	3-3
Global Variables	3-4
Context-sensitive help for code complexity metrics, MISRA-C:2012, and custom coding rules	3-5
Detection of stack pointer dereference outside scope	3-5
Review of latest results compared to the last run	3-6
Guidance for reviewing Polyspace Code Prover checks in C code	3-7
Improvements in search capability in the user interface ...	3-7
Isolated ellipsis for variable number of function arguments supported	3-8
Improvement in pointer comparisons	3-8
Improvements in coding rules checking	3-9
Simplified results infrastructure	3-11
Support for GCC 4.8	3-11
Polyspace plug-in for Simulink improvements	3-11
Integration with Simulink projects	3-11
DRS file format changed to XML	3-12
Back-to-model available when Simulink is closed	3-12
Polyspace binaries being removed	3-12
Import Visual Studio project being removed	3-13

Support for MISRA C:2012	4-2
Improved verification speed	4-2
Support for Mac OS	4-3
Improved verification precision for non-initialized variables	4-3
Read Operations on Structures	4-3
Other Operations	4-5
Support for C++11	4-6
Context-sensitive help for verification options and checks .	4-6
Code Editor for editing source files in Polyspace user interface	4-7
Local file-by-file verification	4-7
Simulink plug-in support for custom project files	4-8
TargetLink support updated	4-8
AUTOSAR support added	4-8
New checks for functions not called	4-9
Default verification level changed	4-9
Improved precision level	4-10
Default mode changed for C++ code verification in user interface	4-11
Updated Software Quality Objectives	4-11
Improved global menu in user interface	4-11

Improved Project Manager perspective	4-12
Changed analysis options	4-13
Improved Results Manager perspective	4-13
Error mode removed from coding rules checking	4-15
Remote launcher and queue manager renamed	4-15
Polyspace binaries being removed	4-16
Import Visual Studio project being removed	4-17

R2014a

Automatic project setup from build systems	5-2
Support for GNU 4.7 and Microsoft Visual Studio C++ 2012 dialects	5-2
Documentation in Japanese	5-3
Support for additional Coding Rules (MISRA C:2004 Rule 18.2, MISRA C++ Rule 5-0-11)	5-3
Preferences file moved	5-3
Support for batch analysis security levels	5-3
Interactive mode for remote verification	5-4
Default text editor	5-4
Results folder appearance in Project Browser	5-4
Results Manager improvements	5-6

Simplification of coding rules checking	5-8
Support for Windows 8 and Windows Server 2012	5-9
Check model configuration automatically before analysis .	5-10
Additional back-to-model support for Simulink plug-in ...	5-10
Function replacement in Simulink plug-in	5-10
Polyspace binaries being removed	5-11
Improvement of floating point precision	5-11

R2013b

Proven absence of certain run-time errors in C and C++ code	6-2
Color-coding of run-time errors directly in code	6-2
Calculation of range information for variables, function parameters and return values	6-2
Identification of variables exceeding specified range limits	6-3
Quality metrics for tracking conformance to software quality objectives	6-3
Web-based dashboard providing code metrics and quality status	6-4
Guided review-checking process for classifying results and run-time error status	6-4
Graphical display of variable reads and writes	6-5
Comparison with R2013a Polyspace products	6-5

R2016a

Version: 9.5

New Features

Bug Fixes

Compatibility Considerations

Files to Review: Generate results for only specified files and folders

In R2016a, you have greater control over the files on which you want analysis results. The default project configuration displays coding rule violations and code metrics on the set of files that are likely to be most relevant to you. You can add files or folders to this set based on your requirements.

For instance, by default, coding rule violations and code metrics are generated on header files that are located in the same folder as the source files. Often, other header files belong to a third-party library. Though these header files are required for a precise analysis, you are not interested in reviewing findings in those headers. Therefore, by default, results are not generated for those headers. If you are interested in certain headers from third-party libraries, you can add those headers to the subset on which results are generated.

For more information, see:

- Generate results for sources and (-generate-results-for)
- Do not generate results for (-do-not-generate-results-for)

Compatibility Considerations

In R2016a, by default, coding rule violations and code metrics are not generated for headers unless they are in the same location as source files. Previously, if you ran verification at the command line, by default, results were generated for all headers.

Due to the change in default behavior, if you rerun verification on a pre-R2016a project without changing the options, you can lose review comments on findings in some header files. To avoid losing the comments, set the option Generate results for sources and (-generate-results-for) to all-headers.

Autocompletion for Review Comments: Partially type previous comment to select complete comment

In R2016a, on the **Results Summary** or **Result Details** pane, if you start typing a review comment that you have previously entered, a drop-down list shows the previous entry. Select the previous comment from this list instead of retyping the comment.

If you want the autocompletion to be case sensitive, select **Tools > Preferences**. On the **Miscellaneous** tab, select **Autocomplete on Results Summary or Details is case sensitive**.

Faster MISRA Rule Checking: Check coding rules more quickly and efficiently

In R2016a, you can use two predefined subsets to perform a quicker and more efficient check for coding rule violations. The new subsets turn on rules that have the same scope.

- `single-unit-rules` — Check rules that apply only to single translation units.
- `system-decidable-rules` — Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules can be checked only at the integration level because the rules involve more than one translation unit.

Polyspace[®] finds these subsets of rules in the early phases of the analysis. If your project is large, before checking all rules, you can check these subsets of rules for a preliminary analysis.

For more information, see “Coding Rule Subsets Checked Early in Analysis”.

S-Function Analysis: Launch analysis of S-Function code from Simulink

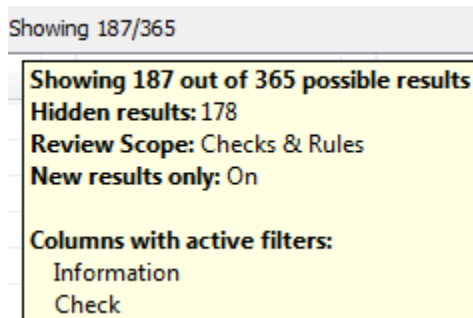
With the Polyspace plug-in for Simulink[®], you can now start a Polyspace verification on S-Functions directly from an S-Function block.

To analyze an S-Function, right-click the S-Function block and select **Polyspace > Verify S-Function**. If the S-Function occurs in your model multiple times, you can choose to analyze all instances of the S-Function by verifying all signal range inputs, or just a single instance of the S-Function by verifying the specific signal ranges for that block.


Persistent Filter States: Apply filters once and view filtered results across multiple runs

In R2016a, if you apply a set of filters to your verification results and rerun verification on the project module, your filters are also applied to the new results. You can specify your filters once and suppress results that are not relevant for you across multiple runs.

The **Results Summary** pane shows the number of results filtered from the display. If you place your cursor on this number, you can see the applied filters.



For instance, in the image, you can see that the following filters have been applied:

- The **Checks & Rules** filter to suppress code metrics and global variables.
- The  filter to suppress results found in a previous verification.
- Filters on the **Information** and **Check** columns.

For more information, see “Filter and Group Results”.

Floating-Point Support: Propagate ranges more precisely for long double variables and enable verification mode to incorporate infinities and NaNs

In R2016a, there are the following improvements on analysis of code involving floating-point variables.

Long Doubles

If your code contains computations involving `long double` variables, you can see fewer orange checks resulting from overapproximation. Previously, Polyspace assumed full-range value for `long double` variables, irrespective of the actual values assigned to them. This assumption led to orange checks that indicated potential numerical and other errors in computations involving `long double` variables.

Polyspace now propagates ranges more precisely for `long double` variables. For information on the number of bits that Polyspace uses for computations involving `long double` variables, see Target processor type (`-target`).

Nonfinites in floating-point arithmetic

Polyspace verification supports nonfinite results such as infinities and NaNs from computations involving floating-point variables. Using the option `Allow non finite floats` (`-allow-non-finite-floats`), you can enable a verification mode that incorporates infinities and NaNs.

In this mode, Polyspace assumes that:

- Floating-point operations can produce results such as infinities and NaNs.

Using options `-check-infinite` and `-check-nan`, you can choose to highlight operations that produce nonfinite results and stop the execution paths where the nonfinite results occur.

- Floating-point variables with unknown values, such as `volatile` variables and return values of stubbed functions, can be infinite or NaN.

The following table illustrates the change.

Prior to R2016a	R2016a
<p>In the following code, Polyspace produces a Division by zero error and stops verification.</p> <pre>double func(void) { double x=1.0/0.0; double y=1.0/x; double z=x-x; return z; }</pre>	<p>In the following code, if you specify the option Allow non finite floats, Polyspace does not check for a Division by zero error.</p> <pre>double func(void) { double x=1.0/0.0; double y=1.0/x; double z=x-x; return z; }</pre> <p>The verification assumes that dividing by zero results in:</p> <ul style="list-style-type: none">• Value of <code>x</code> equal to <code>Inf</code>• Value of <code>y</code> equal to <code>0.0</code>• Value of <code>z</code> equal to <code>NaN</code> <p>In your verification results in the Polyspace user interface, if you place your cursor on <code>y</code></p>

Prior to R2016a	R2016a
	and Z, you can see the nonfinite values Inf and NaN respectively in the tooltip.

Rounding modes

Polyspace supports verification that considers all possible rounding modes and extended precision when rounding the results of floating point arithmetic. Using the option `Float rounding mode (-float-rounding-mode)`, you can enable a verification mode with the following assumptions:

- These rounding modes are possible: round-to-nearest, round-towards-zero, round-towards-positive-infinity and round-towards-negative-infinity.

The default assumption is round-to-nearest only.

- Extended precision can be used for rounding.

The default assumption emulates the GCC flag `-ffloat-store` and disallows extended precision.

Previously, the default verification assumed all rounding modes and extended precision to determine the results of floating-point arithmetic. The verification used the round-to-nearest mode without extended precision only to determine if an **Overflow** occurs.

The following table illustrates the change.

Prior to R2016a	R2016a
<p>In the following code, Polyspace produces a green Overflow check because the addition does not overflow if the result is rounded in to-nearest mode.</p> <pre>#include <float.h> void func(void) { double base = DBL_MAX; double acc = 1.247400193459199882 285232945648024103792 1570377722e+291; base = acc + base;</pre>	<p>In the following code, if you specify all for the option Float rounding mode, Polyspace produces an orange Overflow check because the addition overflows if the result is rounded towards positive infinity.</p> <pre>#include <float.h> void func(void) { double base = DBL_MAX; double acc = 1.247400193459199882 285232945648024103792 1570377722e+291;</pre>

Prior to R2016a	R2016a
}	base = acc + base; }

Polyspace Metrics Tomcat Upgrade: Use upgraded default Tomcat server or custom Tomcat version

Polyspace Metrics now uses Tomcat 8.0.22 to run the Polyspace Metrics web interface.

If you want to use your own version of Tomcat, you can now specify a custom Tomcat server in the daemon configuration file. To add your custom tomcat web server, add the following line to the daemon configuration file.

```
tomcat_install_dir = <path/to/tomcat>
```

The daemon configuration file is located in:

- Windows — \%APPDATA%\Polyspace_RLDatas\polyspace.conf
- Linux — /etc/Polyspace/polyspace.conf

Updated Polyspace Metrics Interface: View summary of project and metrics

You can now view project-level metric summaries from the main Polyspace Metrics page using one of the following methods:

- On the **Projects** tab, roll your mouse over the list of projects to open a window displaying a summary of the project and project metrics.
- On the **Projects** or **Runs** tab, right-click the column headers to add new columns to the table. new columns you can add include Coding Rules, Bug-Finder Checks, Code Metrics, and Review Progress.

Project Language Flexibility: Change your project language at any time

Projects in the Polyspace interface are no longer fixed to C or to C++. When you create a project, you can add any file to the project. After you add files, select the language for your analysis using the Source code language (-lang) option. If you add or change the files in your project, you can change the language to reflect the most suitable analysis type.

Many options that were C only or C++ only are now available for both languages. To see which analysis options have changed, see “Changes in analysis options” on page 1-14.

External Constraint on Pointers: Specify certain initialization with full range for pointer arguments and return values of stubbed functions

In R2016a, if a stubbed function in your code has a pointer argument or a return value, you can specify certain constraints on the pointer outside your code. Using the constraints, you can reduce the number of Non-initialized local variable checks. A function is stubbed if you do not provide the function definition or if you specify the function name for the option Functions to stub (`-functions-to-stub`). For instance, if you declare a function `func` and do not provide the function definition, `func` is stubbed.

```
int* func (int* ptr);
```

You can specify the new external constraints for the pointer argument and the pointer return value of `func`.

You can specify one of the following:

- The pointer points to a non-array variable and the variable is initialized.

The **Init Allocated** column in the constraint specification file supports a new entry `SINGLE_CERTAIN_WRITE` that allows you to specify this constraint.

- The pointer points to an array and all elements of the array are initialized.

The **Init Allocated** column in the constraint specification file supports a new entry `MULTI_CERTAIN_WRITE` that allows you to specify this constraint.

The following table illustrates the change.

Prior to R2016a	R2016a
<p>Without constraints, Polyspace assumes that <code>x</code> in <code>bar</code> and <code>bar_array</code> are potentially noninitialized when you read them. You cannot specify that the functions <code>foo</code> and <code>foo_array</code> initialize <code>x</code> with full-range values.</p> <pre>#define SIZE 5 void foo(int *ptr);</pre>	<p>If you specify the following constraints in the Init Allocated column, Polyspace considers that <code>x</code> in <code>bar</code> and <code>bar_array</code> are initialized.</p> <ul style="list-style-type: none"> • <code>foo</code>: Specify <code>SINGLE_CERTAIN_WRITE</code> for the argument of <code>foo</code>. In other words, <code>foo</code> writes a value to <code>*ptr</code>. • <code>foo_array</code>: Specify <code>MULTI_CERTAIN_WRITE</code> for the

Prior to R2016a	R2016a
<pre>int bar (void) { int x; foo(&x); return x; } void foo_array(int *ptr); void display(int val); void bar_array(void) { int x[SIZE],sum=0; foo_array(x); for(int i=0; i<SIZE; i++) display(x[i]); }</pre>	<p>argument of <code>foo_array</code>. In other words, <code>ptr</code> points to an array and <code>foo_array</code> writes a value to the array elements.</p> <pre>#define SIZE 5 void foo(int *ptr); int bar (void) { int x; foo(&x); return x; } void foo_array(int *ptr); void display(int val); void bar_array(void) { int x[SIZE],sum=0; foo_array(x); for(int i=0; i<SIZE; i++) display(x[i]); }</pre>

For more information, see “Constraints”.

If your project uses a constraint specification file from a previous release, you do not see any change in the verification results. If you generate a constraint specification file, by default, pointer arguments of stubbed functions are constrained to point to an array that is initialized. Applying the default constraint specification file can reduce the number of orange **Non-initialized local variable** checks.

Improved Result Display for File-by-File Verification: View combined summary of results for all files in user interface

In R2016a, if you perform a file-by-file verification, you can see a summary of results for all files on the **Dashboard** pane. You can open the results for each file directly from this summary table. Previously, you obtained this synthesis in an external `html` file.

For more information, see “Run File-by-File Local Verification”.

Source Code Search: Search large applications more quickly

In R2016a, search results are produced more quickly. If you search for a string in a large application, it takes less time for search results to appear.

You can search for a string either by entering the search string in the box on the **Search** pane, or by right-clicking a word in your code on the **Source** pane, and then selecting a search option.

Default Layouts: Switch easily between project setup and results review in user interface

In R2016a, you have two default layouts of panes in the Polyspace user interface, one for project setup and another for results review.

When setting up your projects, select **Window > Reset Layout > Project Setup**. When reviewing results, select **Window > Reset Layout > Results Review**.

For more information, see “Organize Layout of Polyspace User Interface”.

Simplified Variable Access: View task names instead of aliases

In R2016a, on the **Variable Access** pane, in the **Written by task** and **Read by task** columns, you see the task names. Previously, the columns contained aliases such as `t1`, `t2`, `t3`, . . . You viewed the task names using a legend for the aliases.

Polyspace TargetLink plug-in supports data from structures

The Polyspace plug-in for TargetLink[®] can now import data from structures in the constraint specifications (formerly called DRS) for your analysis.

Polyspace Eclipse plug-in results location moved

When you analyze projects using the Polyspace plug-in for Eclipse[™], your results used to be stored inside your Eclipse project under *eclipse project folder\polyspace*. For new Eclipse projects, Polyspace now stores results in the Polyspace Workspace under *Polyspace_Workspace\EclipseProjects\Eclipse Project Name*, where *Polyspace_Workspace* is the default project location specified in your Polyspace Interface preferences. For more information, see “Results Location”.

Improvements in automatic project creation from build command

In R2016a, automatic project creation from build command is improved.

- If you trace your build command and create a Polyspace project from the command line, you do not have to specify a product name or project language. You can open the project in Polyspace Bug Finder™ or Polyspace Code Prover™. The project language is determined by using the following rules:
 - If all your files are compiled as C, as C++03, or C++11, the corresponding language is assigned to the project.

Language	Options Set in Project
C	Source code language: c
C++03	Source code language: cpp
C++11	Source code language: cpp C++11 Extensions: On

- If some files are compiled as C and the remaining files as C++03 or C++11, the **Source code language** option is set to `cpp`.

The option **C++11 Extensions** is also enabled.

For more information, see `Source code language (-lang)` and `C++11 Extensions (-cpp11-extensions)`.

Previously, you specified the product name by using options `-bug-finder` or `-code-prover`. If you did not specify a project language and your source code consisted of both `.c` and `.cpp` files, the language `cpp` was assigned to the project. The options `-bug-finder` and `-code-prover` have been removed.

For more information, see “Create Project Automatically”.

- If header files in your project contain constructs that are not supported in Polyspace Code Prover, a compilation error occurs. In R2016a, when you trace your build, Polyspace detects such header files and does not add them to your project. Later, when you run verification on the project, you do not face compilation errors because of unsupported constructs in header files.
- The support for IAR compilers has improved. All variations of IAR compilers are now supported for automatic project creation from build command.

Improvements in checking of previously supported MISRA C rules

In R2016a, the following changes have been made in checking of previously supported MISRA C[®] rules.

MISRA C:2004 Rules

Rule	Description	Improvement
MISRA C:2004 Rule 10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.	<p>The rule checker no longer raises a violation of this rule if an expression with a Boolean result is cast to a type that is also effectively Boolean.</p> <p>For instance, in your code, you define a type <code>myBool</code> using a <code>typedef</code> and cast the result of <code>(a && b)</code> to <code>myBool</code>. If you specify to Polyspace that <code>myBool</code> is effectively Boolean, the rule checker does not consider this cast as a violation of rule 10.3. For more information on how to specify effectively Boolean types, see Effective boolean types (-boolean-types).</p>
MISRA C:2004 Rule 12.2	The value of an expression shall be the same under any order of evaluation that the standard permits.	<p>The rule checker no longer flags expressions with the comma operator that can be evaluated in only one order.</p> <p>For instance, the statement <code>ans = (val++, val++)</code> does not violate this rule.</p>

MISRA C:2012 Rules

Rule	Description	Improvement
MISRA C:2012 Rule 13.2	The value of an expression and its persistent side effects	The rule checker no longer flags expressions with the comma operator

Rule	Description	Improvement
	shall be the same under all permitted evaluation orders.	that can be evaluated in only one order. For instance, the statement <code>ans = (val++, val++)</code> does not violate this rule.

Absolute address usage valid by default

In R2016a, the Absolute address usage check is considered valid and therefore green by default. If you assign an absolute address to a pointer in your code, the verification assumes that:

- The address is valid.
- The type of the pointer to which you assign the address determines the initial value stored in the address.

If you assign the address to an `int*` pointer, the memory zone that the address points to is initialized with an `int` value. The value can be anything that is allowed for the data type `int`.

Previously, the **Absolute address usage** check was considered possibly invalid and therefore orange by default. You either justified the checks or turned them green by using the option **Green absolute address checks** (`-green-absolute-address-checks` on command line).

Compatibility Considerations

If the code in your project uses absolute addresses, you see a decrease in the number of orange checks from previous releases of the software. To turn the check orange by default for each absolute address usage, use the command-line option `-no-assumption-on-absolute-addresses`. To use a command-line option in the user interface, enter the option in the Other field.

Variables with constraints not counted as orange sources

In R2016a, once you constrain certain variables outside your code, those variables do not appear as possible causes of orange checks on the **Orange Sources** pane.

This pane lists the variables that you can constrain outside your code to reduce orange checks.

- Previously, the pane listed variables even after you had constrained them, with the assumption that you might constrain them further.
- Starting in R2016a, Polyspace assumes that once you constrain variables to match real-world values, you will not constrain them further.

Therefore, variables already constrained are not shown on the **Orange Sources** pane.

For more information on constraining variables using the **Orange Sources** pane, see “Create Constraint Template After Verification”.

Changes in analysis options

In R2016a, the following options have been added, changed, or removed.

New Options

Option	Description
Generate results for sources and (-generate-results-for)	Specify files on which you want analysis results.
Do not generate results for (-do-not-generate-results-for)	Specify files on which you do not want analysis results.
Allow non finite floats (-allow-non-finite-floats)	Enable a verification mode that incorporates infinities and NaNs.
Float rounding mode (-float-rounding-mode)	Assume all rounding modes and extended precision when determining the results of floating point arithmetic.
-check-infinite	Specify how to handle floating point operations that result in infinity.
-check-nan	Specify how to handle floating point operations that result in NaN.
-no-assumption-on-absolute-addresses	Make Absolute address usage checks orange by default.

Updated Options

Option	Change	More Information
Source code language (-lang)	Added to Polyspace Interface	Select your project language to set compilation rules and enable language specific analysis options.
Dialect (-dialect)	Unified dialects for C and C++ projects. All projects can use any dialect option.	
Target processor type (-target)	Targets i386 and x86_64 now allow any alignment value.	
Sfr type support (-sfr-types)	Allowed for both C and C++	
Pack alignment value (-pack-alignment-value)	Allowed for both C and C++	
Import folder (-import-dir)	Allowed for both C and C++	
Ignore pragma pack directives (-ignore-pragma-pack)	Allowed for both C and C++	
Division round down (-div-round-down)	Allowed for both C and C++	

Removed Options

Option	Status	More Information
Green absolute address checks (-green-absolute-address-checks)	Warning	Absolute address usage checks are green by default. To remove this assumption and produce an orange check, use the option -no-assumption-on-absolute-addresses.
Files and folders to ignore (-includes-to-ignore)	Warning	Use the option Do not generate results for (-do-not-generate-results-

Option	Status	More Information
		for) to suppress results from headers and sources in certain files or folders.
Ignore float rounding (-ignore-float-rounding)	Warning	Option will be removed in a future release.
-retype-pointer	Warning	Option will be removed in a future release.
-retype-int-pointer	Warning	Option will be removed in a future release.
-lwtm	Warning	Option will be removed in a future release.
-support-FX-option-results	Warning	Option will be removed in a future release.
polyspace-vcproj	Error	Binary has been removed.

Compatibility Considerations

If you use scripts that contain the removed or updated options, change your scripts accordingly.

Run-time checks renamed

In R2016a, the following checks have been renamed. The new names state the error in the code instead of what the check looks for.

Old Name	New Name
Absolute address	Absolute address usage
C++ specific checks	Invalid C++ specific operations
Exception handling	Uncaught exception
Function Returns a Value	Function not returning value
Initialized Return Value	Return value not initialized
Non-null this-pointer in method	Null this-pointer calling method

Old Name	New Name
Object Oriented Programming	Incorrect object oriented programming
Shift operations	Invalid shift operations

R2015b

Version: 9.4

New Features

Bug Fixes

Compatibility Considerations

Option to Suppress Non-initialization Checks: Customize verification by suppressing non-initialization checks

In R2015b, you can use an analysis option to turn off the checks for non-initialization. If you turn on this option, Polyspace assumes that, at declaration:

- Variables have full-range of values allowed by their type.
- Pointers can be NULL-valued or point to a memory block at an unknown offset.

When you use this option, the following checks are turned off:

- Non-initialized local variable: Local variable is not initialized before being read.
- Non-initialized variable: Variable other than local variable is not initialized before being read.
- Non-initialized pointer: Pointer is not initialized before being read.
- Return value not initialized: C function does not return value when expected.

For more information, see [Disable checks for non-initialization \(C/C++\)](#).

Improved Concurrency Detection: View more precise sharing and protection results based on dynamic information such as data flow in branching statements and protection on individual fields of a structure

In R2015b, Polyspace Code Prover uses dynamic information such as data flow in branch statements to determine whether a variable is shared and protected. Previously, sharing and protection were determined statically resulting in overapproximation of the actual behavior. For more information on shared variables and multitasking options, see [Multitasking](#).

The following examples illustrate the change. For more examples, see [Global Variables](#).

Data Flow in Branch Statements

Prior to R2015b	R2015b
In the following code, if you specify <code>task1</code> and <code>task2</code> as entry points, the verification determines that <code>var_1</code> and <code>var_2</code> are shared, potentially unprotected variables. However, because of the <code>if</code> statement, <code>task1</code> can operate only on <code>var_1</code> and	In the following code, if you specify <code>task1</code> and <code>task2</code> as entry points, the verification determines that <code>var_1</code> and <code>var_2</code> are not shared. <code>unsigned int var_1;</code>

Prior to R2015b	R2015b
<p>task2 only on var_2. When determining sharing, the verification does not consider the branching in the if statement and therefore determines that var_1 and var_2 are shared.</p> <pre> unsigned int var_1; unsigned int var_2; volatile int randomVal; void task1(void) { while(randomVal) operation(1); } void task2(void) { while(randomVal) operation(2); } void operation(int i) { if(i==1) { var_1++; } else { var_2++; } } int main(void) { return 0; } </pre>	<pre> unsigned int var_2; volatile int randomVal; void task1(void) { while(randomVal) operation(1); } void task2(void) { while(randomVal) operation(2); } void operation(int i) { if(i==1) { var_1++; } else { var_2++; } } int main(void) { return 0; } </pre>

Shared Structures

Prior to R2015b	R2015b
<p>In the following code, if you specify <code>task1</code> and <code>task2</code> as entry points, the verification determines that the structure variable <code>sharedStruct</code> is a potentially unprotected variable. However, <code>task1</code> can operate only</p>	<p>In the following code, if you specify <code>task1</code> and <code>task2</code> as entry points, the verification determines that the structure variable <code>sharedStruct</code> is a shared, protected variable. If you select the result, the</p>

Prior to R2015b	R2015b
<p>on <code>sharedStruct.var_1</code> and <code>task2</code> only on <code>sharedStruct.var_2</code>. The verification considers <code>sharedStruct</code> as a whole and ignores the sharing and protection on individual fields of <code>sharedStruct</code>.</p> <pre> struct S { unsigned int var_1; unsigned int var_2; }; volatile int randomVal; struct S sharedStruct; void task1(void) { while(randomVal) operation1(); } void task2(void) { while(randomVal) operation2(); } void operation1(void) { sharedStruct.var_1++; } void operation2(void) { sharedStruct.var_2++; } int main(void) { return 0; } </pre>	<p>Result Details pane states that all operations on the variable are protected by access pattern. For the variable <code>sharedStruct</code>, the Protection column on the Variable Access pane contains Access pattern.</p> <pre> struct S { unsigned int var_1; unsigned int var_2; }; volatile int randomVal; struct S sharedStruct; void task1(void) { while(randomVal) operation1(); } void task2(void) { while(randomVal) operation2(); } void operation1(void) { sharedStruct.var_1++; } void operation2(void) { sharedStruct.var_2++; } int main(void) { return 0; } </pre>

Microsoft Visual C++ 2013 Support: Analyze code developed in Microsoft Visual C++ 2013

You can analyze code developed in the Microsoft® Visual C++® 2013 dialect.

To analyze code compiled with Microsoft Visual C++ 2013, set your dialect to `visual12.0`. If you specify the dialect, Polyspace allows language extensions specific to Microsoft Visual C++ 2013. Otherwise, it produces a compilation error if you use those extensions. For more information, see [Dialect \(C/C++\)](#) or [Dialect \(C++\)](#).

Additional MISRA C:2012 Support: Detect violations of all MISRA C:2012 rules except rules 22.x

In R2015b, Polyspace Code Prover supports the following MISRA C: 2012 coding rules.

For complete MISRA C: 2012 support, including rules 22.1–22.4 and 22.6, use Polyspace Bug Finder.

Rule	Description
MISRA C:2012 Directive 2.1	All source files shall compile without any compilation errors.
MISRA C:2012 Directive 4.5	Identifiers in the same name space with overlapping visibility should be typographically unambiguous.
MISRA C:2012 Rule 2.6	A function should not contain unused label declarations.
MISRA C:2012 Rule 2.7	There should be no unused parameters in functions.
MISRA C:2012 Rule 17.5	The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.
MISRA C:2012 Rule 17.8	A function parameter should not be modified.
MISRA C:2012 Rule 21.12	The exception handling features of <code><fenv.h></code> should not be used.
MISRA C:2012 Rule 22.5	A pointer to a <code>FILE</code> object shall not be dereferenced.

GNU 4.9 and Clang 3.5 Support: Analyze code compiled with GCC 4.9 or Clang 3.5

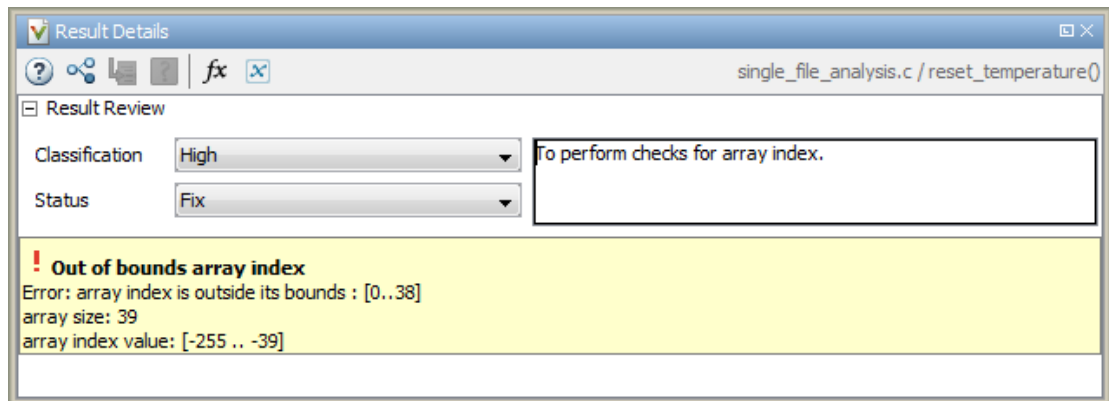
Polyspace now supports the GNU 4.9 and Clang 3.5 dialects for C and C++ projects.

To analyze code compiled with one of these dialects, set the **Target & Compiler > Dialect** option to `gnu4.9` or `clang3.5`.

For more information, see [Dialect \(C/C++\)](#) or [Dialect \(C++\)](#).

Improved Review Capability: View result details and add review comments in one window

In R2015b, the **Check Details** pane is renamed **Result Details**. On this pane, in addition to viewing details about a result, you can now enter review information such as **Classification**, **Status**, and comments. For more information, see [Add Review Comments to Results](#).



Saved Layouts: Save your preferred layouts of the Polyspace user interface

In R2015b, if you reorganize the Polyspace user interface and place the various panes in more convenient locations, you can save your new layout. If you change your layout, you can quickly revert to a saved layout.

With this modification, you can create customized layouts suitable for different requirements and switch between saved layouts. For instance:

- You can have separate layouts for project configuration and results review.
- You can have a minimal layout with only frequently used panes.

For more information, see *Organize Layout of Polyspace User Interface*.

Autodetection of Multitasking Primitives: Analyze source code with multitasking primitives from POSIX or VxWorks without manual setup

If you use POSIX® or VxWorks® to perform multitasking, Polyspace can now interpret your multitasking code without having to change your code or manually set multiple configuration options.

To turn on automatic detection, select the analysis option **Multitasking > Enable automatic concurrency detection**. Polyspace detects thread creation and critical sections from supported multitasking functions.

Functions Polyspace can interpret:

POSIX

- `pthread_create`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`

VxWorks

- `taskSpawn`
- `semTake`
- `semGive`

For more information, see *Enable automatic concurrency detection (C/C++)*.

Start Page: Get quickly familiar with Polyspace Code Prover

In R2015b, when you open Polyspace Code Prover for the first time, a **Start Page** pane appears. From this pane, you can:

- Open recent results and demo examples.
- Start a new project.
- Get additional help using the **Getting Started**, **What's New** and **Learn More** tabs.

If you select the **Show on startup** box on the lower left of this pane, the pane appears each time you open Polyspace Code Prover. Otherwise, if you close the pane once, it does not reopen. To open the pane, select **Window > Show/Hide View > Start Page**.

Enhanced Review Scope: Filter coding rule violations from display in one click

In R2015b, you can suppress a certain number or percentage of coding rule violations from the display using custom options in the **Show** menu on the **Results Summary** pane. You can:

- Suppress violations of coding rules that are not relevant for you.
- Focus your results review by seeing only a certain number of coding rule violations in your display.
- Predefine a percentage of coding rule violations that you intend to review. View only that percentage in your analysis results.

You define an option on the **Show** menu only once. The option is available for one-click use every time that you open your results. For more information, see [Suppress Certain Rules from Display in One Click](#).

Previously, using custom options on the **Show** menu, you suppressed orange checks and code metrics (if they fell below a certain threshold). With this enhancement, you can use the **Show** menu to display only those results that must be justified to reach a certain Software Quality Objective (SQO) level. For instance, you can reach predefined SQO levels 4, 5, and 6 using the options on the **Show** menu. For more information, see [Software Quality Objectives](#).

Additional Call Graph Showing Task Creation

For global variables, the call graph provides a visual representation of the function call sequence leading to operations on the variable. In R2015b, the call graph for shared global variables has been augmented with a supporting call graph that shows task creation.

Previously, Polyspace modeled multitasking code by assuming that all tasks begin after the `main` completes execution. This model has been relaxed for POSIX thread creation functions allowing creation of tasks in the `main` and in functions called from the `main`. Therefore, the call sequence leading to the creation of a task can be nontrivial. The task creation call graph provides you a visual representation of this call sequence.

For more information, see [Review Global Variable Usage](#).

Improved precision for mathematical functions

Polyspace Code Prover has more precise implementations for mathematical functions defined in `math.h`.

Improved handling of `__declspec`

For projects in Visual C, Polyspace Code Prover can now interpret the aligned size specified by the keyword `__declspec(align(...) ...)`.

For example, this structure uses the `__declspec` keyword:

```
struct S1 { __declspec(align(8)) char c; };
```


In R2015b Polyspace correctly interprets the size of `S1` as 8 bytes.

Compatibility Considerations

In previous versions, Polyspace ignored the `__declspec` keyword, so code with the `__declspec(align())` keyword was verifiable using **Dialect > None**. To avoid compilation errors with the R2015b support of `__declspec(align())`, set **Dialect** to one of the Visual C dialects. For the list of supported Visual dialects, see [Dialect \(C/C++\)](#).

Improvements in Polyspace Metrics workflow

In R2015b, the Polyspace Metrics workflow has improved in the following ways:

- You can justify code complexity metrics in the Polyspace user interface and upload the justifications to Polyspace Metrics. If a code metric value violates quality thresholds and appears red, after justification, it appears green with the  icon.

For more information about justifying Polyspace results starting from the Polyspace Metrics interface, see [Compare Metrics Against Software Quality Objectives](#).

- You can define custom SQO levels specific to a project. In the file **Custom-SQO-Definitions.xml**, if you specify a project name, in the Polyspace Metrics web dashboard, the custom SQO level appears only for that project. You can choose this SQO level to compare the project against quality thresholds that you defined. For more information, see [Customize Software Quality Objectives](#).
- In the Polyspace user interface, the same menu item **Metrics > Upload to Metrics** allows you to upload your results initially and also upload comments and justifications in the results later.

Previously, you used a different menu item **Save comments to Metrics** to save your review comments and justifications in a result.

For more information on uploading comments and justifications from the Polyspace user interface to the Polyspace Metrics web interface, see [Review Metrics for Particular Project or Run](#).

Updated Support for TargetLink


The Polyspace plug-in for TargetLink now supports versions 3.5 and 4.0 of the dSPACE® Data Dictionary and TargetLink Code Generator.

dSPACE and TargetLink version 3.4 is no longer supported.

For more information, see [TargetLink Considerations](#).

Improvements in Polyspace Plugin for Eclipse

In R2015b, the following improvements have been made to the Polyspace plugin for Eclipse:

- When you select a result in the **Results Summary** view, the **Result Details** view displays additional information about the result. In the **Result Details** view, if you click the  button next to the result name, you can see a brief description and examples of the result.
- You can switch to a Polyspace perspective that shows only the information relevant to a Polyspace Code Prover verification. To open the perspective, select **Window > Open Perspective > Other**. In the Open Perspective dialog box, select **Polyspace**.

Improvements in automatic project creation from build command

In R2015b, automatic project creation from build command is improved.

- If you build your source code from the Cygwin™ environment (using either a 32 or 64-bit installation), Polyspace can trace your build and create a Polyspace project or options file.
- Support for the following compilers has improved:
 - Texas Instruments™ C2000 compiler

This compiler is available with Code Composer Studio™.
 - Cosmic HC08 C compiler
 - MPLAB XC8 C Compiler
- With certain compilers, the speed of tracing your build command has improved. The software now stores build information in the system temporary folder, thereby allowing faster access during the build.

If you still encounter a slow build, use the advanced option `-cache-path ./ps_cache` when tracing your build. For more information, see [Slow Build Process When Polyspace Traces the Build](#).

- If the software detects target settings that correspond to a standard processor type, it assigns that standard target processor type to your project. The target processor type defines the size of fundamental data types and the endianness of the target machine. For more information, see [Target processor type \(C/C++\)](#).

Previously, when you created a project from your build command, the software assigned a custom target processor type. Although you saw the processor type in the form of an option such as `-custom-target true,8,2,4,-1,4,8,4,8,8,4,8,1,little,unsigned_int,int,unsigned_short`, you could not identify easily how many bits were associated with each fundamental type. With this enhancement, when the software assigns a processor type, you can identify the number of bits for each type. Click the **Edit** button for the option **Target processor type**.

- Automatic project creation uses a configuration file written for specific compilers. If your compiler is not supported, you can adapt one of the existing configuration files for your compiler. The configuration file, written in XML, is now simplified with some new elements, macros and attributes.

- The `preprocess_options_list` element supports a new `$(OUTPUT_FILE)` macro when the compiler does not allow sending the preprocessed file to the standard output.
- A new `preprocessed_output_file` element allows the preprocessed file name to be adapted from the source file name.
- The `semantic_options` element supports a new `isPrefix` attribute. This attribute provides a shortcut to specify multiple semantic options that begin with the same prefix.
- The `semantic_options` element supports a new `numArgs` attribute. This attribute provides a shortcut to specify semantic options that take one or more arguments.

For more information, see [Compiler Not Supported for Project Creation from Build Systems](#).

- Sometimes, the build command returns a non-zero status even when the command succeeds. The non-zero status can result from warnings in the build process. However, Polyspace does not trace the build and create a Polyspace project. You can now use an option `-allow-build-error` to create a Polyspace project even if the build command returns an exit status or error level different from zero. This option helps you understand the error in the build process.

For more information, see `-option value` arguments of `polyspaceConfigure`.

Improvements in checking of previously supported MISRA C rules

In R2015b, the following changes have been made in checking of previously supported MISRA C rules.

MISRA C:2004 Rules

Rule	Description	Improvement
MISRA C:2004 Rule 2.1	Assembly language shall be encapsulated and isolated.	If an assembly language statement is entirely encapsulated in macros, Polyspace no longer considers that the statement violates this rule.
MISRA C:2004 Rule 8.8	An external object or function shall be	Polyspace considers that variables or functions declared <code>extern</code> in a non-header file violate this rule.

Rule	Description	Improvement
	declared in one file and only one file.	
MISRA C:2004 Rule 10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if it is not a conversion to a wider integer type of the same signedness.	Polyspace no longer raises violation of this rule on operations involving pointers.
MISRA C:2004 Rule 19.2	Nonstandard characters should not occur in header file names in <code>#include</code> directives.	Polyspace no longer raises violation of this rule if the character <code>\</code> or <code>\</code> occurs between the <code><</code> and <code>></code> in <code>#include <filename></code> (or between <code>"</code> and <code>"</code> in <code>#include "filename"</code>). Therefore, you can use Windows [®] paths to files in place of <code>filename</code> without triggering a rule violation.

MISRA C:2012 Rules

Rule	Description	Improvement
MISRA C:2012 Directive 4.3	Assembly language shall be encapsulated and isolated.	If an assembly language statement is entirely encapsulated in macros, Polyspace no longer considers that the statement violates this rule.
MISRA C:2012 Rule 1.1	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.	If a rule violation occurs because your <code>.c</code> file contains too many macros, instead of placing the rule violation on the last macro usage, Polyspace places the rule violation at the beginning of the file. Therefore, you can add a comment before the first line of the <code>.c</code> file

Rule	Description	Improvement
		justifying the violation. Previously, you had to place the comment before the last macro usage. If you added another use of the macro later, the comment did not apply. For information on adding code comments to justify results, see Add Review Comments to Code.
MISRA C:2012 Rule 10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.	<ul style="list-style-type: none"> • If one of the operands is the constant zero, Polyspace does not raise a violation of this rule. • If one of the operands is a signed constant and the other operand is unsigned, the rule violation is not raised if the signed constant has the same representation as its unsigned equivalent. <p>For instance, the statement <code>u8b = u8a + 3;</code>, where <code>u8a</code> and <code>u8b</code> are unsigned char variables, does not violate the rule because the constants <code>3</code> and <code>3U</code> have the same representation.</p>

Checking Coding Rules Using Text Files

In R2015b, if your coding rules configuration text file has an incorrect syntax, the analysis stops with an error message. The error message states the line numbers in the configuration file that contain the incorrect syntax.

For more information on checking for coding rules using text files, see [Select Specific MISRA or JSF Coding Rules](#).

Including options multiple times

You can now specify analysis options multiple times. This feature is available only at the command line or using the command-line names in the **Advanced options** dialog

box in the user interface. Customize pre-made configurations without having to find the changed options in the options file.

If you specify an option multiple times, only the last setting is used. For example, if your configuration is:

```
-lang c
-prog test_bf_cp
-verif-version 1.0
-author username
-sources-list-file sources.txt
-OS-target no-predefined-OS
-target i386
-dialect none
-misra-cpp required-rules
-target powerpc
```

Polyspace uses the last target setting, `powerpc`, and ignores the other target specified, `i386`.

The user interface also follows this rule. If you specify `c18` for **Target processor type** and `-target i386` for **Advanced options**, this counts as multiple analysis option specifications. Polyspace uses the target specified in the **Advanced options** box, `i386`.

Compatibility Considerations

If your current configuration specifies analysis options multiple times, change the configuration by either:

- Removing the unnecessary analysis options.
- Moving the desired analysis options to the end of the configuration.

Renaming of labels in Polyspace user interface

In the Polyspace user interface, the following labels have been renamed:

- On the **Configuration** pane, the node **Coding Rules** is changed to **Coding Rules & Code Metrics**. The **Coding Rules & Code Metrics** node contains the option **Calculate Code Metrics**, which appeared previously on the **Advanced Settings** node.
- On the **Results Summary** pane, the **Category** column title is changed to **Group**, avoiding confusion with coding rule categories.

- On the **Results Summary** and **Result Details** pane, the field **Classification** is changed to **Severity**. You assign a **Severity** such as **High**, **Medium** and **Low** to a defect to indicate how critical you consider the issue.
- The labels associated with specifying constraints have changed as follows:
 - On the **Configuration** pane, the field **Variable/function range setup** is changed to **Constraint setup**.
 - When you click **Edit** beside the **Constraint setup** field, a new window opens. The window name is changed from **Polyspace DRS Configuration** to **Constraint Specification**.

For more information, see [Specify Constraints](#).

Configuration Associated with Result Not Opened by Default

In R2015b, when you open your result, the **Configuration** pane does not automatically display a read-only form of the associated configuration.

To view the configuration associated with the result, select the link **View configuration for results** on the **Dashboard** pane. If a corresponding project is open on the **Project Browser** pane, you can also right-click the result under the **Results** node in the project and select **Open Configuration**.

Improvements in Report Templates

In R2015b, the major improvements in report templates include the following:

- Instead of filenames, absolute paths to files appear in the reports.
- If you check for coding rules, the appendix about coding rules configuration states all rules along with the information whether they were enabled or disabled. Previously, the appendix only stated the enabled rules.

For more information on templates, see [Report template \(C/C++\)](#).

Changes in analysis options

In R2015b, the following options have been added or removed.

New Options

Option	Status	More information
Respect C90 Standard (-no-language-extensions)	New	The analysis does not allow C language extensions that do not follow the ISO/IEC 9899:1990 standard.
Dialect visual12.0	New	Allows Microsoft Visual C++ 2013 (visual 12) language extensions.
Dialect gnu4.9	New	Allows GCC 4.9 language extensions.
Dialect clang3.5	New	Allows Clang 3.5 language extensions.
Configure multitasking manually (C/C++)	New	This option enables the previous multitasking options (Entry points , Critical section details , Temporally exclusive tasks) in the user interface.
Enable automatic concurrency detection (C/C++)	New	Enables automatic concurrency detection for POSIX® and VxWorks® threading functions.
Disable checks for non-initialization (C/C++)	New	Disables checks for non-initialization in your code.

Updated Options

Option	Status	More information
Calculate Code Metrics (C/C++)	Moved in user interface	The option has been moved in the Configuration panel from the Advanced Settings pane to the Coding Rules and Code Metrics pane.
-class-analyzer	Updated syntax	The syntax for <code>-class-analyzer param</code> has been updated. Use <code>-class-analyzer custom=param</code>
Signed right shift (C/C++) (-logical-signed-right-shift)	Now available in C++ projects	
Division round down (C/C++)	Now available in C++ projects	

Option	Status	More information
<code>(-div-round-down)</code>		
<code>(-no-def-init-glob)</code>	Now available in C++ projects	
Optimize large static initializers (C/C++) <code>(-no-fold)</code>	Now available in C++ projects	
<code>-lightweight-thread-model</code>	No longer available in the user interface	
Targets: <ul style="list-style-type: none"> • <code>tms320c3x</code> • <code>,sharc21x61</code> • <code>necv850</code> • <code>hc08</code> • <code>hc12</code> • <code>mpc5xx</code> • <code>c18</code> 	Now available in C++ projects	
Enum type definition (C/C++) <code>(-enum-type-definition)</code>	Possible values updated	The possible values for <code>-enum-type-definition</code> are the same for C and C++. Available values: <ul style="list-style-type: none"> • <code>defined-by-standard</code> (default) • <code>auto-signed-first</code> • <code>auto-unsigned-first</code>
<code>-asm-begin -asm-end</code>	Now available in C++ projects	
<code>-support-FX-option-results</code>	No longer available in the user interface	
<code>-pointer-is-24bits</code>	Available in C++ projects	Available only if you use the Target setting <code>C18</code> .

Option	Status	More information
Output format (C/C++) -report-output-format	Possible values updated	The output format RTF is deprecated and not available on the Configuration pane.

Removed Options

Option	Status	More information
-dialect cfront2	Removed	Use a different dialect instead.
-dialect cfront3	Removed	Use a different dialect instead.
-known-NTC	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-desktop	Removed	Use -main-generator instead.
-permissive	Removed	Use -allow-negative-operand-in-shift -ignore-constant-overflows instead.
-automatic-stubbing	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-float-overflows	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-continue-with-exisiting-host	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-allow-unsupported-linux	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-passes-time	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-ignore-missing-headers	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.

Option	Status	More information
-continue-with-red-error	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-voa	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-machine-architecture	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-allow-non-int-bitfield	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-allow-undef-variables	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-allow-unnamed-fields	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-permissive stubber	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-permissive-link	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-allow-language-extensions	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-include-headers-once	Removed	Polyspace includes this behavior by default. Remove this option from existing configurations.
-strict	Removed	This option is no longer supported. Remove this option from existing configurations.
-discard-asm	Removed	This option is no longer supported. Remove this option from existing configurations.

Option	Status	More information
-quick	Removed	Use -to pass0 instead.
-detect-unsigned- overflows	Removed	Use -scalar-overflows-checks-signed- and-unsigned instead.
-misra2 AC-AGC-OBL- subset	Removed	Use -misra-ac-agc OBL-rules instead.

Compatibility Considerations

If you use scripts that contain a removed or updated option, change your scripts accordingly.

Change in Correctness Condition Check

In R2015b, the specification of the **Correctness Condition** check has changed in the following way. For more information on the check, see Correctness condition.

When you use a function pointer to call a function and Polyspace cannot determine which function the pointer points to, the **Correctness Condition** check is orange instead of red. This situation can occur, for instance, if:

- The function pointer points to an *absolute address*. The check is orange because the verification cannot determine from the code whether the absolute address contains a well-typed function.
- The function pointer contains the return value of a stubbed function. For information on stubbing, see Assumptions About Stubbed Functions.

Following the orange check, the verification assumes that the following variables can have the full range of values allowed by their type:

- Variable storing the return value from the function call.
- Variables that can be modified through the function arguments.

Compatibility Considerations

If your code contains function pointers that point to an absolute address for instance, you can see a change in the number of results from a previous version of the product. Because red checks stop further verification of the code in the current block and orange

checks do not, this change of the **Correctness Condition** check from red to orange can expose more of your code to verification. Therefore, the number of checks in your code can change.

Binaries removed

The following binaries have been removed.

Binary name	Use instead
polyspace-automatic -orange-tester.exe	From the Polyspace environment, select Tools > Automatic Orange Tester
polyspace-c.exe	polyspace-code-prover-nodesktop -lang c
polyspace-cpp.exe	polyspace-code-prover-nodesktop -lang cpp
polyspace-remote-c.exe	polyspace-code-prover-nodesktop -lang c -batch
polyspace-remote-cpp.exe	polyspace-code-prover-nodesktop -lang cpp -batch
polyspace-remote.exe	polyspace-code-prover-nodesktop -batch
polyspace-rl-manager.exe	polyspace-server-settings.exe
polyspace-spooler.exe	polyspace-job-monitor.exe
polyspace-ver.exe	polyspace-code-prover-nodesktop -ver

The binaries to use are located in *matlabroot/polyspace/bin*.

Support for Visual Studio 2008 to be removed

The Polyspace Add-In for Visual Studio® 2008 is no longer supported and will be removed in a future release.

Compatibility Considerations

To analyze your Visual Studio projects, use either:

- The Polyspace Add-in for Visual Studio 2010. See *Install Polyspace Add-In for Visual Studio*.

-
- The `polyspace-configure` tool to create a project using your build command. See [Create Project Using Visual Studio Information](#).

Import Visual Studio project removed

The **Tools > Import Visual Studio project** has been removed.

To import your project information from Visual Studio, use the **Create from build system** option during new project creation. For more information, see [Create Project Using Visual Studio Information](#).

XML and RTF report formats removed

The formats XML and RTF for report generation are no longer available from R2016a onwards. If you generated reports using one of these formats, use an alternative format instead.

For more information, see [Output format \(C/C++\)](#).

R2015a

Version: 9.3

New Features

Bug Fixes

Compatibility Considerations

Simplified workflow for project setup and results review with a unified user interface

In R2015a, the Project and Results Manager perspectives are now unified. You can run verification and review results without switching between two perspectives.

The major changes are:

- You can start a new verification during your results review. Previously, you started a new verification only from the Project Manager perspective.
- After a verification, the result opens automatically. If you are looking at a previous result when a verification is over, you can load the new result or retain the previous one on the **Results Summary** pane. If you retain the previous results, you can later open the new results from the **Project Browser**. The new results are highlighted.
- You can have any of the panes open in the unified interface.

Previously, you could open the following panes only in one of the two perspectives.

Project Manager	Results Manager
<ul style="list-style-type: none"> • Project Browser: Set up project. • Configuration: Specify analysis options for your project. • Output Summary: Monitor progress of verification. • Run Log: Find detailed information about a verification. 	<ul style="list-style-type: none"> • Results Summary: View Polyspace results. • Source: View read-only form of source code color coded with Polyspace results. • Check Details: View details of a particular result. • Check Review: Comment on a particular result. • Variable Access: View global variables and read/write operations on them. • Call Hierarchy: View callers and callees of a function. • Results Properties: Same as Run Log, but associated with results instead of a project. This pane has been removed.

Project Manager	Results Manager
	<p>To open the log associated with a result, with the results open, select Window > Show/Hide View > Run Log.</p> <ul style="list-style-type: none"> • Settings: Same information as Configuration, but associated with results instead of a project. This pane has been removed. <p>To open the configuration associated with a result, with the results open, select Window > Show/Hide View > Configuration.</p> <ul style="list-style-type: none"> • Orange Sources: View sources of orange checks. • Sensitivity Context: For a check that has a different color for different function calls, view the check color for each function call.

Review of code complexity metrics and global variable usage in user interface

- “Code Complexity Metrics” on page 3-3
- “Global Variables” on page 3-4

Code Complexity Metrics

In R2015a, you can view code complexity metrics in the Polyspace user interface. For more information, see Code Metrics. Previously, this information was available only in the Polyspace Metrics web interface.

In the user interface, you can:

- Specify a limit for the value of a metric. If the metric value for your source code exceeds this limit, the metric appears red on the **Results Summary** pane.

- Justify the value of a metric. If a metric value exceeds specified limits and appears red, you can add a comment with the rationale.

Combining these actions, you can enforce coding standards across your organization. For more information, see [Review Code Metrics](#).

Reducing the complexity of your code improves code readability, reduces the possibility of coding errors, and allows more precise Polyspace verification.

Global Variables

In R2015a, you can comment and justify global variable usage on the **Results Summary** pane. Previously, you viewed global variable usage on the **Variable Access** pane, but could not comment on them.

On the **Results Summary** pane, global variables are classified into one of the following categories.

Category		Color	Meaning
Shared	Potentially unprotected	Orange	Global variables shared between multiple tasks but possibly not protected from concurrent access by the tasks
	Protected	Green	Global variables shared between multiple tasks and protected from concurrent access by the tasks
Not shared	Used	Black	Global variables used in a single task
	Unused	Gray	Global variables declared but not used

For more information, see [Global Variables](#).

For code that you do not intend for multitasking, all variables are nonshared and can be either used or unused. For code that you intend for multitasking, you can specify tasks and protections through the analysis options for multitasking. For more information, see [Multitasking](#).

You can still view the global variables on the **Variable Access** pane.

- To comment and justify potentially unprotected and unused global variables, use the **Results Summary** pane.
- To find the read and write operations on a global variable, use the **Check Details** or **Variable Access** pane. On the **Variable Access** pane, you can also see the variable range and other information.

For more information, see [Review Global Variable Usage](#).

Context-sensitive help for code complexity metrics, MISRA-C:2012, and custom coding rules

In R2015a, context-sensitive help is available in the user interface for code complexity metrics, MISRA C:2012 rule violations, and custom coding rule violations.

To access the contextual help, see [Getting Help](#).

For information about these results, see:

- [Code Metrics](#)
- [MISRA C:2012 Directives and Rules](#)
- [Custom Coding Rules](#)

Detection of stack pointer dereference outside scope


In R2015a, the **Illegally dereferenced pointer** check can detect stack pointer dereference outside scope. Such dereference can happen, for example, when a pointer to a variable that is local to a function is returned from the function. Because the scope of the variable is limited to the function, dereferencing the pointer outside the function can cause undefined behavior.

This enhancement is not available by default. Use the option `-detect-pointer-escape` to detect such dereferences. To provide command-line options in the user interface:

- 1 On the **Configuration** pane, select **Advanced Settings**.
- 2 Enter the option in the **Other** field.

Before R2015a	R2015a
<p>In the following code, <code>ptr</code> points to <code>ret</code>. Because the scope of <code>ret</code> is limited to <code>func1</code>, when <code>ptr</code> is accessed in <code>func2</code>, the access is illegal. Polyspace Code Prover did not detect such pointer escapes.</p> <pre>void func2(int *ptr) { *ptr = 0; } int* func1(void) { int ret = 0; return &ret ; } void main(void) { int* ptr = func1() ; func2(ptr) ; }</pre>	<p>In the following code, Polyspace Code Prover produces a red Illegally dereferenced pointer check on <code>*ptr</code>.</p> <pre>void func2(int *ptr) { *ptr = 0; } int* func1(void) { int ret = 0; return &ret ; } void main(void) { int* ptr = func1() ; func2(ptr) ; }</pre>

The **Check Details** pane displays a message indicating that `ret` is accessed outside its scope.

 **ID 1: Illegally dereferenced pointer**

Error: pointer is outside its bounds

This check may be a path-related issue, which is not dependent on input values

Dereference of parameter 'ptr' (pointer to int 32, size: 32 bits):

Pointer is not null.

Points to 4 bytes at offset 0 in buffer of 4 bytes, so is within bounds (if memory is allocated).

Pointer may point to variable or field of variable:

'ret', local to function 'func1'. 'ret' is accessed outside its scope.

Review of latest results compared to the last run

In R2015a, you can review only new results compared to the previous run.

If you rerun your verification, the new results are displayed with an asterisk (*) against them on the **Results Summary** pane. To filter only these new checks, select the **New results** box.


If you make changes in your source code, you can use this feature to see only the checks introduced due to those changes. You can avoid reviewing checks in the source code that you did not change.

Guidance for reviewing Polyspace Code Prover checks in C code

In R2015a, the context-sensitive help for checks provides guidance about how to review the check. The help describes:

- Information available in the software for the check.
- In your source code, how to navigate to the root cause of the check.
- Common causes of the check.

To open the context-sensitive help for a check:

- On the **Results Summary** or **Source** pane, select the check.
- Select the  button.
- Select the link in the section **Diagnosing This Check**.

This additional guidance is not available for C++-specific checks.

Improvements in search capability in the user interface

In R2015a, the **Search** pane allows you to search for a string in various panes of the user interface.

To search for a string in the new user interface:

- 1 If the **Search** pane is not visible, open it. Select **Window > Show/Hide View > Search**.
- 2 Enter your string in the search box.
- 3 From the drop-down list beside the box, select names of panes you want to search.

The **Search** pane consolidates the search options previously available.

Isolated ellipsis for variable number of function arguments supported

In R2015a, for C++ code, Polyspace Code Prover supports the ellipsis in the function definition syntax `void foo(...){}` to mean variable number of arguments. Previously, the use of ellipsis in isolation was not supported. You could use only the syntax where the ellipsis was preceded with other parameters.

Before R2015a	R2015a
<p>In the following code, Polyspace considers that <code>foo</code> has no arguments. Therefore, it produces a red Correctness condition error on the second function call. The Check Details pane indicates that the wrong number of arguments were used in the function call.</p> <pre data-bbox="238 777 785 1001"> void foo(...) { /* Function body */ } void main() { foo(); foo(1,2); //Red COR } </pre>	<p>In the following code, Polyspace considers that <code>foo</code> takes a variable number of arguments. It does not produce a red Correctness condition error on the second function call.</p> <pre data-bbox="788 713 1337 939"> void foo(...) { /* Function body */ } void main() { foo(); foo(1,2); //No COR } </pre>

Improvement in pointer comparisons

In R2015a, Polyspace is more precise on pointer comparisons. In certain cases, if the software can determine that a pointer comparison is always true or false, it provides that result. Previously, Polyspace did not check pointer comparisons.

Before R2015a	R2015a
<p>In the following code, Polyspace does not check the comparison <code>ptr==&invalid</code>. Therefore, it considers that <code>check</code> can return either 0 or 1. In the <code>main</code> function, it verifies both branches of the <code>if-else</code> statement.</p> <pre data-bbox="238 1512 785 1531"> #include <stdlib.h> </pre>	<p>In the following code, Polyspace checks the comparison <code>ptr===&invalid</code> and determines that it is always true. Therefore, it considers that the <code>if</code> test is redundant and the function <code>check</code> returns 1 only. In the <code>main</code> function, it verifies the</p>

Before R2015a	R2015a
<pre> typedef unsigned char U8; U8 invalid; #define TEST_DISABLED &invalid U8 check(U8 cnt, U8* ptr) { U8 ret=0; if (ptr == &invalid) { ret=1; } return ret; } void main() { U8 isDisabled; isDisabled = check(1U,TEST_DISABLED); if(isDisabled == 1) { /* Do not perform test */ } else { /* Perform test */ } } </pre>	<pre> if branch and considers the else branch as unreachable. #include <stdlib.h> typedef unsigned char U8; U8 invalid; #define TEST_DISABLED &invalid U8 check(U8 cnt, U8* ptr) { U8 ret=0; if(ptr == &invalid) { ret=1; } return ret; } void main() { U8 isDisabled; isDisabled = check(1U,TEST_DISABLED); if(isDisabled == 1) { /* Do not perform test */ } else { /* Perform test */ } } </pre>

Improvements in coding rules checking

MISRA C:2004 and MISRA AC AGC

Rule Number	Effect	More Information
Rule 12.6	More results on noncompliant <code>#if</code> preprocessor directives Fewer results for variables cast to effective Boolean types.	MISRA C:2004 Rules — Chapter 12: Expressions
Rule 12.12	Fewer results when converting to an array of <code>float</code>	MISRA C:2004 Rules — Chapter 12: Expressions

MISRA C:2012

Rule Number	Effect	More Information
Rules 10.3	Fewer results on enumeration constants when the type of the constant is a named enumeration type. Fewer results on user-defined effective Boolean types.	MISRA C:2012 Rule 10.3
Rule 10.4	Fewer results on enumeration constants when the type of the constant is a named enumeration type. Fewer results for casts to user-defined effective Boolean types.	MISRA C:2012 Rule 10.4
Rule 10.5	Fewer results on enumeration constants when the type of the constant is a named enumeration type. Fewer results on user-defined effective Boolean types.	MISRA C:2012 Rule 10.5
Rule 12.1	More results on expressions with <code>sizeof</code> operator and on expressions with <code>?</code> operators. Fewer results on operators of the same precedence and in preprocessing directives.	MISRA C:2012 Rule 12.1
Rule 14.3	No results for non-controlling expressions.	MISRA C:2012 Rule 14.3

MISRA C++:2008

Rule Number	Effect	More Information
Rule 5-0-3	Fewer results on enumeration constants when the type of the constant is the enumeration type.	MISRA C++ Rules — Chapter 5
Rule 6-5-1	Fewer results on compliant vector variable iterators.	MISRA C++ Rules — Chapter 6

Rule Number	Effect	More Information
Rule 14-8-2	Fewer results for functions contained in the Files and folders to ignore (C++) option.	MISRA C++ Rules — Chapter 14
Rule 15-3-2	Fewer results for user-defined return statements after a <code>try</code> block.	MISRA C++ Rules — Chapter 15

Simplified results infrastructure

Polyspace results folders are reorganized and simplified. Files have been removed, combined, renamed, or moved. The changes do not affect the results that you see in the Polyspace environment.

Some important changes and file locations:

- The main results file is now encrypted and renamed `ps_results.pscp`. You can view results only in the Polyspace environment.
- The log file, `Polyspace_R2015a_project_date-time.log` has not changed.

For more information, see Results Folder Contents.

Support for GCC 4.8

Polyspace now supports the GCC 4.8 dialect for C and C++ projects.

To allow GCC 4.8 extensions in your Polyspace Code Prover verification, set **Target & Compiler > Dialect** option `gnu4.8`.

For more information, see Dialect (C) and Dialect (C++).

Polyspace plug-in for Simulink improvements

In R2015a, there are three improvements to the Polyspace Simulink plug-in.

Integration with Simulink projects

You can now save your Polyspace results to a Simulink project. Using this feature, you can organize and control your Polyspace results alongside your model files and folders.

To save your results to a Simulink project:

- 1 Open your Simulink project.
- 2 From your model, select **Code > Polyspace > Options**.
- 3 In the Polyspace parameter configuration tab, select the **Save results to Simulink project** option.

For more information, see [Save Results to a Simulink Project](#).

DRS file format changed to XML

By default, the DRS files generated in Simulink are saved in XML.

For more information, see [XML File Format for Constraints](#)

If you want to use a customized `.txt` DRS file, contact customer support.

Back-to-model available when Simulink is closed

In the Polyspace plug-in for Simulink, the back-to-model feature now works even when your model is closed. When you click a link in your Polyspace results, MATLAB® opens your Simulink model and highlights the appropriate block.

Note: This feature works only with Simulink R2013b and later.

For more information about the back-to-model feature, see [Identify Errors in Simulink Models](#).

Polyspace binaries being removed

The following Polyspace binaries will be removed in a future release. The binaries are located in `matlabroot/polyspace/bin`. You get a warning if you run them.

Binary name	Use instead
<code>polyspace-automatic -orange-tester.exe</code>	From the Polyspace environment, select Tools > Automatic Orange Tester
<code>polyspace-c.exe</code>	<code>polyspace-code-prover-nodesktop -lang c</code>

Binary name	Use instead
polyspace-cpp.exe	polyspace-code-prover-nodesktop -lang cpp
polyspace-remote-c.exe	polyspace-code-prover-nodesktop -lang c -batch
polyspace-remote-cpp.exe	polyspace-code-prover-nodesktop -lang cpp -batch
polyspace-remote.exe	polyspace-code-prover-nodesktop -batch
polyspace-rl-manager.exe	polyspace-server-settings.exe
polyspace-spooler.exe	polyspace-job-monitor.exe
polyspace-ver.exe	polyspace-code-prover-nodesktop -ver

Import Visual Studio project being removed

The **File > Import Visual Studio project** will be removed in a future release. Instead, use the **Create from build system** option during New Project creation. For more information, see Trace Visual Studio Build.

R2014b

Version: 9.2

New Features

Bug Fixes

Compatibility Considerations

Support for MISRA C:2012

Polyspace can now check your code against MISRA C:2012 directives and coding rules. To check for MISRA C:2012 coding rule violations:

- 1 On the **Configuration** pane, select **Coding Rules**.
- 2 Select **Check MISRA C:2012**.
- 3 The MISRA C:2012 guidelines have different categories for handwritten and automatically generated code.

If you want to use the settings for automatically generated code, also select **Use generated code requirements**.

For more information about supported rules, see MISRA C:2012 Coding Directives and Rules.

Improved verification speed

In R2014b, the following two changes improve the verification speed:

- Polyspace Code Prover can run the compilation phase of your verification in parallel on multiple processors. The software detects available processors and uses them to compile different source files in parallel.

Previously, the software ran post-compilation phases in parallel but compiled the source files sequentially. Starting in R2014b, the software can use multiple processors for the entire verification process.

To explicitly specify the number of processors, use the command-line option `-max-processes`. For more information, see `-max-processes`.

- Polyspace Code Prover has an improved engine for verification. This engine typically improves verification speed by 25%. However, in some cases, verification can take the same amount of time or longer.

Compatibility Considerations

In most cases, you do not see significant change in the number of checks resulting from the improved engine. If you see a major increase in the number of orange checks, contact technical support. For more information, see Obtain System Information for Technical Support.

Support for Mac OS

You can install and run Polyspace on Mac OS X. Polyspace is supported for Mac OS 10.7.4+, 10.8, and 10.9.

You can use Polyspace Metrics on Safari and set up your Mac as a Metrics server. However, if you restart your Mac machine that is setup as a Metrics server, you must restart the Polyspace server daemon.

The Automatic Orange Tester is not supported for Mac.

Improved verification precision for non-initialized variables

Polyspace Code Prover performs the following checks for initialization:

- Non-initialized local variable or NIVL
- Non-initialized variable or NIV

In R2014b, the following changes appear in these checks.

Read Operations on Structures

When you read structured variables, Polyspace Code Prover performs a check for initialization. This check helps detect partially initialized and non-initialized structures earlier in the code.

Prior to R2014b	R2014b
<ul style="list-style-type: none">• When you read structured variable, a check for initialization was not performed.• The checks occurred only when you read individual fields of a structured variable, provided the fields themselves were not structured variables.	<p>When you read structured variables, a check for initialization occurs. The check turns:</p> <ul style="list-style-type: none">• Green, if all fields of the structure that are used are initialized. If no field is used, the check is green by default.• Red, if all fields that are used are not initialized.• Orange, if only some fields that are used are initialized. Following the check, Polyspace considers that the

Prior to R2014b	R2014b
	<p>uninitialized fields have the full range of values allowed by their type.</p> <p>Polyspace considers a field as used if there is a read or write operation on the field anywhere in the code. Polyspace does not check for initialization of fields that are not used.</p> <p>To determine which fields Polyspace checked for initialization:</p> <ol style="list-style-type: none"><li data-bbox="795 661 1299 756">1 Select the NIV or NIVL check on the Results Summary pane or Source pane.<li data-bbox="795 770 1248 836">2 View the message on the Check Details pane.

Prior to R2014b	R2014b
<p>Example:</p> <pre>typedef struct S { int a; int b; }S; void func1(S); void func2(int); void main() { S varS; func1(varS); func2(varS.a); }</pre> <p>A check was not performed when the non-initialized structure <code>varS</code> was read. When the field <code>a</code> of <code>varS</code> was read, a red NIVL check appeared.</p>	<p>Example:</p> <pre>typedef struct S { int a; int b; }S; void func1(S); void func2(int); void main() { S varS; func1(varS); func2(varS.a); }</pre> <p>When the non-initialized structure <code>varS</code> is read, a red NIVL check appears.</p> <p>For more examples, see:</p> <ul style="list-style-type: none"> • Partially initialized structure — All used fields initialized • Partially initialized structure — Some used fields initialized

Other Operations

The specification of **Non-initialized variable** checks has changed for the following operations. These operations are not commonly used. Therefore, it is likely that these changes do not affect your Polyspace verification.

Prior to R2014b	R2014b
<p>If you initialized only the high bits of a variable through a pointer, an orange check for initialization appeared when the variable was read.</p>	<p>If you initialize only the high bits of a variable through a pointer, a green check for initialization appears when the variable is read.</p>
<p>If you performed an operation on a C++ object after it was destroyed, a red check for initialization appeared on the operation.</p>	<p>If you perform an operation on a C++ object after it is destroyed, the check for initialization has the same color as</p>

Prior to R2014b	R2014b
The check indicated that the object was destroyed.	before the destruction. Polyspace does not introduce a red check on this type of access.

Compatibility Considerations

If you use an earlier version of Polyspace Code Prover, it is possible that you see the following changes in your results.

- Read operation on structures: You see an increase in the total number of checks.

However, some red or orange NIV or NIVL checks on the fields of structures turn green. Instead, you see some new red or orange checks on the structures themselves.
- Other operations:
 - If you have operations that initialize only the high bits of a variable through a pointer, you can see a reduction in orange NIV or NIVL checks.
 - If you have operations that access an object after it is destroyed, you can see a reduction in red NIV or NIVL checks.

Support for C++11

Polyspace can now fully analyze C++ code that follows the ISO/IEC 14882:2011 standard, also called C++11.

Use two new analysis options when analyzing C++11 code. On the **Target & Compiler** pane, select:

- **C++11 extensions** to allow the standard C++11 libraries and functions during your analysis.
- **Block char 16/32_t types** to not allow char16_t or char32_t types during the analysis.

For more information, see C++11 Extensions (C++) and Block char16/32_t types (C++).


Context-sensitive help for verification options and checks

In R2014b, contextual help is available for verification options in the Polyspace interface and its plug-ins. To view the contextual help:

-
- 1 Hover your cursor over a verification option in the **Configuration** pane.
 - 2 Inside the tooltip, select the “More Help” link.

The documentation for that option appears in a dockable window.

Contextual help is available in the Polyspace interface for run-time errors. To view the contextual help for checks:

- 1 In the Results Manager perspective, select a run-time error from the results.
- 2 Inside the **Check Details** pane, select .

The documentation for that check appears in a docked window.

For more information, see Getting Help.

Code Editor for editing source files in Polyspace user interface

In R2014b, by default, you can edit your source files inside the Polyspace user interface.

- In the Project Manager perspective, on the **Project Browser** tree, double-click your source file.
- In the Results Manager perspective, right-click the **Source** pane and select **Open Source File**.

Your source files appear on a **Code Editor** tab. On this tab, you can edit your source files and save them.

To use an external text editor, change your preferences.

- 1 Select **Tools > Preferences**.
- 2 Specify an external editor on the **Editors** tab.

For more information, see Specify External Text Editor.

Local file-by-file verification

In R2014b, you can verify your source code file by file on your local installation of Polyspace Code Prover. Each file is verified independently of the other files in your module. Previously, you performed file-by-file verification only on a remote server. The verification required:

- Parallel Computing Toolbox™ on the client side
- MATLAB Distributed Computing Server™ on the server side

For more information on file-by-file verification, see:

- Run File-by-File Verification
- Open Results of File-by-File Verification

For information on file-by-file verification in batch mode, see:

- Run File-by-File Batch Verification
- Open Results of File-by-File Batch Verification

Simulink plug-in support for custom project files

With the Polyspace plug-in for Simulink, you can now use a project file to specify the verification options.

On the **Polyspace** pane of the Configuration Parameters window, with the **Use custom project file** option you can enter a path or browse for a `.psprj` project file.

For more information, see [Configure Polyspace Analysis Options](#).

TargetLink support updated

The Polyspace plug-in for Simulink now supports TargetLink 3.4 and 3.5. Older versions of TargetLink are not supported.

For more information, see [TargetLink Considerations](#).

AUTOSAR support added

In R2013b, the Polyspace plug-in for Simulink added support for AUTOSAR generated code with Embedded Coder®. If you use `autosar.tlc` as your **System target file** for code generation, when you run Polyspace, the verification can use the data range information from AUTOSAR.

The Polyspace verification uses the same default options and parameters as it does for Embedded Coder.

For more information, see *Embedded Coder Considerations*.

New checks for functions not called

Two new checks in Polyspace Code Prover detect C/C++ functions that are defined but not called during execution of the code.

Check	Purpose
Function not called	Detects functions that are defined but not called in the source files.
Function not reachable	Detects functions that are defined but called only from an unreachable part of the source.

You can choose to activate these checks using the following options:

- In the user interface, on the **Configuration** pane, under **Check Behavior**, select a value for the option **Detect uncalled functions**.
- At the command line, use the option `-uncalled-function-checks` with an appropriate argument.

Goal	Option Value
Do not detect uncalled functions.	<code>none</code>
Detect functions that are defined but not called.	<code>never-called</code>
Detect functions that are defined and called only from an unreachable part of the code.	<code>called-from-unreachable</code>
Detect all uncalled functions.	<code>all</code>

Default verification level changed

In R2014b, unless you specify a verification level explicitly, Polyspace Code Prover verification performs two passes on your source code instead of four. For instance:

- In the user interface, on the **Output Summary** tab, you can see that the verification continues to **Level2**. For more passes, on the **Configuration** pane, under the **Precision** node, select a higher **Verification level**.

- At the command line, the verification implicitly uses `-to pass2`. For more passes, use the `-to` option explicitly with a higher pass value.

The default verification is completed in much less time.

For more information, see:

- Verification level (C)
- Verification level (C++)

Compatibility Considerations

If you do not specify a verification level explicitly in your `polyspace-code-prover-nodesktop` command, your verification runs to **Software Safety Analysis Level 2**. In most cases, this verification level produces only slightly more orange checks than **Software Safety Analysis Level 4**. However, if you see a significant change in your results, to reproduce your earlier results:

- In the user interface, select **Software Safety Analysis Level 4** for **Verification level**.
- At the command line, use the option `-to pass4` with the `polyspace-code-prover-nodesktop` command.

Improved precision level

In R2014b, certain internal limits have been removed from verification that uses a **Precision level** of **3**. Because of this improvement, you can use this **Precision level** to significantly reduce orange checks, especially for multitasking code that uses shared variables. However, if you use this level, the verification can take significantly longer.

To set **Precision level** to **3**, do one of the following:

- In the user interface, on the **Configuration** pane, select **Precision**. From the **Precision level** drop-down list, select **3**.
- At the DOS or UNIX[®] command prompt, use the flag `-O3` with the `polyspace-code-prover-nodesktop` command.
- At the MATLAB command prompt, use the argument `'-O3'` with the `polyspaceCodeProver` function.

For more information, see [Precision level \(C/C++\)](#).

Default mode changed for C++ code verification in user interface

When you create a new Polyspace Code Prover project with C++ as the project language, the following options are selected in the user interface by default. The options appear on the **Configuration** pane under the **Code Prover Verification** node.

Option	Value
Verify Module	On
Class	all
Functions to call within the specified classes	unused
Functions to call	unused
Variables to initialize	uninit

These options replace the default selection of **Verify whole application** on the Polyspace user interface.

If your C++ code does not contain a `main` function, Polyspace generates a `main` by default during verification from the user interface.

For more information on the `main` generation options, see [Provide Context for C++ Code Verification](#).

Updated Software Quality Objectives

In R2014b, the Software Quality Objectives or SQOs have been updated to include MISRA[®] C++: 2008 coding rule violations.

Using the predefined SQO levels, you can specify quality thresholds for your project or individual files in your project. With the updated SQOs, you can now specify that your project must not violate certain MISRA C++ rules.

For more information, see [Predefined SQO Levels](#).

Improved global menu in user interface

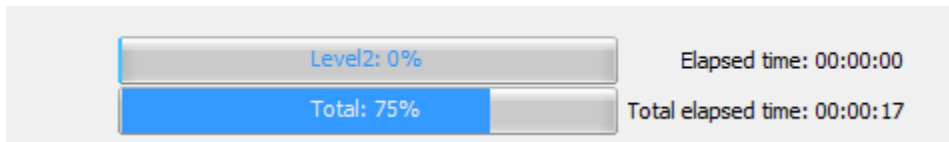
The global menu in the Polyspace user interface has been updated. The following table lists the current location for the existing global menu options.


Goal	Prior to R2014b	R2014b
Open the Polyspace Metrics interface in your web browser.	File > Open Metrics Web Interface	Metrics > Open Metrics
Upload results from the Polyspace user interface to Polyspace Metrics.	File > Upload in Polyspace Metrics repository	Metrics > Upload to Metrics
Update results stored in Polyspace Metrics with your review comments and justifications.	File > Save in Polyspace Metrics repository	Metrics > Save comments to Metrics
Generate a report from results after verification.	Run > Run Report > Run Report	Reporting > Run Report
Open generated report.	Run > Run Report > Open Report	Reporting > Open Report
Partition source code into modules.	Run > Run Modularize	Tools > Run Modularize
Import review comments from previous verification.	Review > Import	Tools > Import Comments
Specify code generator for generated code.	Review > Code Generator Support	Tools > Code Generator Support
Specify settings that apply to all Polyspace Code Prover projects.	Options > Preferences	Tools > Preferences
Specify settings for remote verification.	Options > Metrics and Remote Server Settings	Metrics > Metrics and Remote Server Settings

Improved Project Manager perspective

The following changes have been made in the Project Manager perspective:

- The **Progress Monitor** tab does not exist anymore. Instead, after you start a verification, you can view its progress on the **Output Summary** tab.
- Instead of a single progress bar showing all the stages of verification, you can see two progress bars. The top bar shows progress in the current stage of verification and the lower bar shows overall progress.



After verification, you can see the overall time taken. To see the time taken in each stage of verification, click the  icon.

- In the **Project Browser**, projects appear sorted in alphabetical order instead of order of creation.

Changed analysis options

Changes have been made to the following analysis options:

- On the **Configuration** pane, the analysis option **Files and folders to ignore** has been moved from **Coding Rules Checking** to **Inputs & Stubbing**. The functionality in Polyspace Code Prover has not changed.
- On the **Configuration** pane, the **Interactive** option has been removed from the graphical interface. To use interactive mode, use the `-interactive` flag at the command line or in the **Advanced Settings > Other** text field.
- You cannot use batch mode or interactive mode with **Verification Level > C/C++ source compliance checking**.

To run only to code compliance, run Polyspace Code Prover locally.

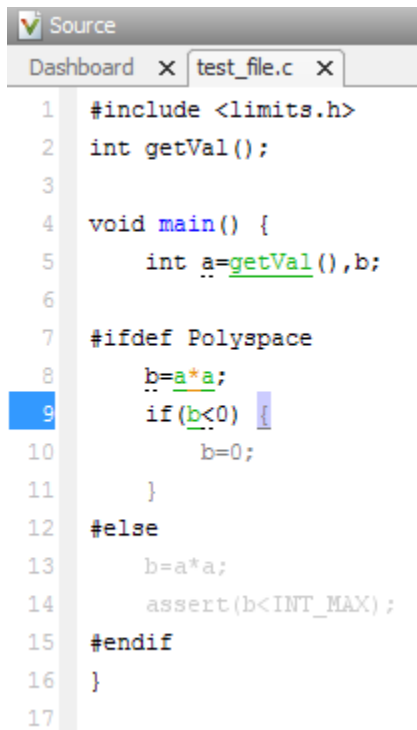
To perform batch or interactive verifications, use **Software Safety Analysis level 0** or higher.

Improved Results Manager perspective

The following changes have been made in the Results Manager perspective:

- On the **Source** pane, the following code appears in gray:
 - Code deactivated due to conditional compilation. Polyspace assigns a lighter shade of gray to this code.
 - Code in an unreachable branch. Polyspace assigns a darker shade of gray to this code.

For the difference between the two cases, see the code below. To reproduce the colors, before verification, on the **Configuration** pane, enter **Polyspace=** for **Preprocessor definitions**.



```
Source
Dashboard x test_file.c x
1 #include <limits.h>
2 int getVal();
3
4 void main() {
5     int a=getVal(),b;
6
7     #ifdef Polyspace
8         b=a*a;
9         if(b<0) {
10             b=0;
11         }
12     #else
13         b=a*a;
14         assert(b<INT_MAX);
15     #endif
16 }
17
```

- To prioritize your orange check review, use the **Show** menu on the **Results Summary** pane. This menu replaces the previously available methodologies for the same purpose.
 - To display red, gray, and orange checks likely to be run-time errors, from the **Show** menu, select **Critical checks**. This option replaces the **First checks to review** methodology.
 - To display all checks, from the **Show** menu, select **All checks**. This option replaces the **All checks** methodology.
 - The methodologies **Methodology for C/C++ > Light** and **Methodology for C/C++ > Moderate** have been removed.

-
- To create your own subset of orange checks to review, select **Tools > Preferences**. On the **Review Scope** tab, specify the number or percentage of orange checks of each type to review. The options on this tab replace the options on the **Review Configuration** tab.
 - To group your checks, use the **Group by** menu on the **Results Summary** pane.
 - To leave your checks ungrouped, instead of **List of Checks**, select **Group by > None**.
 - To group checks by check color and type, instead of **Checks by Family**, select **Group by > Family**.
 - To group checks by file and function, instead of **Checks by File/Function**, select **Group by > File**.
 - To view the percentage of checks that you have justified, instead of the **Review Statistics** pane, use the **Justified** column on the **Results Summary** pane. On this pane:
 - To view the percentage of checks that you justified broken down by color/type, select **Group by > Family**.
 - To view the percentage of checks that you justified broken down by file/function, select **Group by > File**.

Error mode removed from coding rules checking

In R2014b, the **Error** mode has been removed from coding rules checking. Therefore, coding rule violations cannot stop a verification.

Compatibility Considerations

For existing coding rules files, rules having the keyword **error** are treated in the same way as the keyword **warning**. For more information on **warning**, see [Format of Custom Coding Rules File](#).

Remote launcher and queue manager renamed

Polyspace has renamed the remote launcher and the queue manager.

Previous name	New Name	More information
polyspace-rl-manager.exe	polyspace-server-settings.exe	Only the binary name has changed. The interface title, Metrics and Remote Server Settings , is unchanged.
polyspace-spooler.exe Queue Manager or Spooler	polyspace-job-monitor.exe Job Monitor	The binary and the interface titles have changed. Interface labels have changed in the Polyspace interface and its plug-ins.
pslinkfun('queuemanager')	pslinkfun('jobmonitor')	See pslinkfun.

Compatibility Considerations

If you use the old binaries or functions, you receive a warning.

Polyspace binaries being removed

The following Polyspace binaries will be removed in a future release. Unless otherwise noted, the binaries to use are located in *matlabroot/polyspace/bin*.

Binary name	What happens	Use instead
polyspace-automatic -orange-tester.exe	Warning	From the Polyspace environment, select Tools > Automatic Orange Tester
polyspace-c.exe	Warning	polyspace-code-prover-nodesktop -lang c
polyspace-cpp.exe	Warning	polyspace-code-prover-nodesktop -lang cpp
polyspace-remote-c.exe	Warning	polyspace-code-prover-nodesktop -lang c -batch
polyspace-remote-cpp.exe	Warning	polyspace-code-prover-nodesktop -lang cpp -batch

Binary name	What happens	Use instead
polyspace-remote.exe	Warning	polyspace-code-prover-nodesktop - batch
polyspace-rl-manager.exe	Warning	polyspace-server-settings.exe
polyspace-spooler.exe	Warning	polyspace-job-monitor.exe
polyspace-ver.exe	Warning	polyspace-code-prover-nodesktop -ver
setup-remote-launcher.exe	Warning	<i>matlabroot</i> /toolbox/polyspace / psdistcomp/bin/setup-polyspace-cluster

Import Visual Studio project being removed

The **File > Import Visual Studio project** will be removed in a future release. Instead, use the **Create from build system** option during New Project creation. For more information, see Trace Visual Studio Build.

R2014a

Version: 9.1

New Features

Bug Fixes

Compatibility Considerations


Automatic project setup from build systems

In R2014a, you can set up a Polyspace project from build automation scripts that you use to build your software application. The automatic project setup runs your automation scripts to determine:

- Source files.
- Includes.
- **Target & Compiler** options.

To set up a project from your build automation scripts:

- On the DOS or UNIX command line: Use the `polyspace-configure` command. For more information, see [Create Project from DOS and UNIX Command Line](#).
- In the user interface: When creating a new project, in the **Project – Properties** window, select **Create from build command**. In the following window, enter:
 - The build command that you use.
 - The directory from which you run your build command.
 - Additional options. For more information, see [Create Project in User Interface](#).

Click . In the **Project Browser**, you see your new Polyspace project with the required source files, include folders, and **Target & Compiler** options.

- On the MATLAB command line: Use the `polyspaceConfigure` function. For more information, see [Create Project from MATLAB Command Line](#).

Support for GNU 4.7 and Microsoft Visual Studio C++ 2012 dialects

Polyspace supports two additional dialects: Microsoft Visual Studio C++ 2012 and GNU[®] 4.7. If your code uses language extensions from these dialects, specify the corresponding analysis option in your configuration. From the **Target & Compiler > Dialect** menu, select:

- `gnu4.7` for GNU 4.7
- `visual11.0` for Microsoft Visual Studio C++ 2012

For more information about these and other supported dialects, see [Dialects for C](#) or [Dialects for C++](#).

Documentation in Japanese

The Polyspace product, including the documentation, is available in Japanese.

To view the Japanese version of Polyspace Code Prover documentation, go to <http://www.mathworks.co.jp/jp/help/codeprover/>. If the documentation appears in English, from the country list beside the globe icon at the top of the page, select Japan.

Support for additional Coding Rules (MISRA C:2004 Rule 18.2, MISRA C++ Rule 5-0-11)

The Polyspace coding rules checker now supports two additional coding rules: MISRA C 18.2 and MISRA C++ 5-0-11.

- MISRA C 18.2 is a required rule that checks for assignments to overlapping objects.
- MISRA C++ 5-0-11 is a required rule that checks for the use of the plain `char` type as anything other than storage or character values.
- MISRA C++ 5-0-12 is a required rule that checks for the use of the signed and unsigned `char` types as anything other than numerical values.

For more information, see MISRA C:2004 Coding Rules or MISRA C++ Coding Rules.

Preferences file moved

In R2014a, the location of the Polyspace preferences file has been changed.

Operating System	Location before R2014a	Location in R2014a
Windows	%APPDATA%\Polyspace	%APPDATA%\MathWorks\MATLAB\R2014a\Polyspace
Linux®	/home/\$USER/.polyspace	/home/\$USER/.matlab/\$RELEASE/Polyspace

For more information, see Storage of Polyspace Preferences.

Support for batch analysis security levels

When creating an MDCS server for Polyspace batch analyses, you can now add additional security levels through the **MATLAB Admin Center**. Using the **Metrics and Remote Server Settings**, the MDCS server is automatically set to security level

zero. If you want additional security for your server, use the **Admin Center** button. The additional security levels require authentication by user name, cluster user name and password, or network user name and password.

For more information, see MDCS documentation.

Interactive mode for remote verification

In R2014a, you can select an additional **Interactive** mode for remote verification. In this mode, when you run Polyspace Code Prover on a cluster, your local computer is tethered to the cluster through Parallel Computing Toolbox and MATLAB Distributed Computing Server.

To run verification in this mode

- In the user interface: On the **Configuration** pane, under **Distributed Computing**, select **Interactive**.
- On the DOS or UNIX command line, append `-interactive` to the `polyspace-code-prover-nodesktop` command.
- On the MATLAB command line, add the argument `'-interactive'` to the `polyspaceCodeProver` function.

For more information, see [Interactive](#).

Default text editor

In R2014a, Polyspace uses a default text editor for opening source files. The editor is:

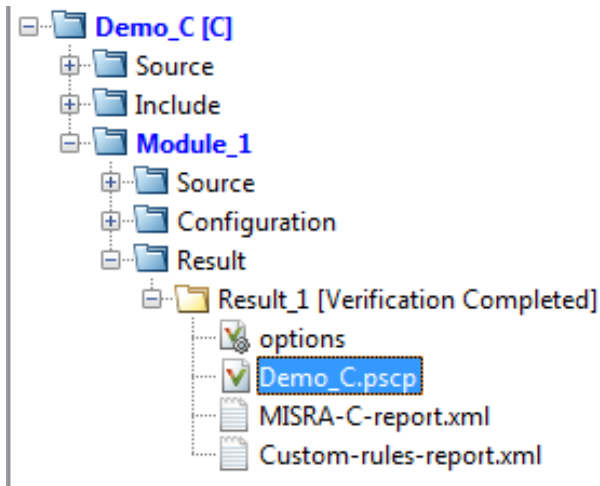
- WordPad in Windows
- vi in Linux

You can change the text editor on the **Editors** tab under **Options > Preferences**. For more information, see [Specify Text Editor](#).

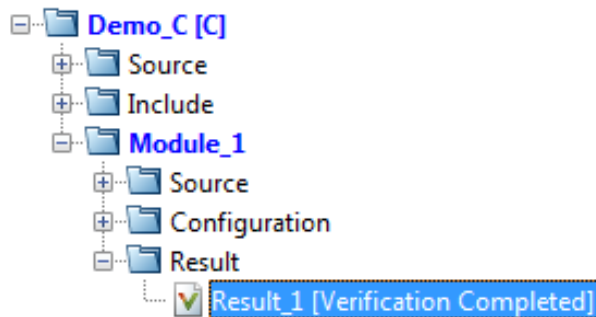
Results folder appearance in Project Browser

In R2014a, the results folder appears in a simplified form in the **Project Browser**. Instead of a folder containing several files, the result appears as a single file.

- Format before R2014a:



- Format in R2014a:



The following table lists the changes in the actions that you can perform on the results folder.

Action	2013b	2014a
Open results.	In the result folder, double-click result file with extension .pscp.	Double-click result file.
Open analysis options used for result.	In the result folder, select options .	Right-click result file and select Open Configuration .

Action	2013b	2014a
Open metrics page for batch analyses if you had used the analysis option Distributed Computing > Add to results repository .	In the result folder, select Metrics Web Page .	Double-click result file. If you had used the option Distributed Computing > Add to results repository , double-clicking the results file for the first time opens the metrics web page instead of the Result Manager perspective.
Open results folder in your file browser.	Navigate to results folder. To find results folder location, select Options > Preferences . View result folder location on the Project and Results Folder tab.	Right-click result file and select Open Folder with File Manager .

Results Manager improvements

- In R2014a, you can view the extent of a code block on the **Source** pane by clicking either its opening or closing brace.

```
Source
Dashboard x tasks1.c x
47 static void initregulate(void)
48 {
49     int tmp = 0;
50     while (random_int() < 1000) {
51         tmp = orderregulate();
52         Begin_CS();
53         tmp = SHR + SHR2 + SHR6;
54         End_CS();
55         tmp = Get_PowerLevel();
56         Compute_Injection();
57     } /* end loop; */
58 }
```

Note: This action does not highlight the code block if the brace itself is already highlighted. The opening brace can be highlighted, for instance, if there is an **Unreachable code** error on the code block.

- In R2014a, the **Verification Statistics** pane in the Project Manager and the **Results Statistics** pane in the Results Manager have been renamed **Dashboard**.

On the **Dashboard**, you can obtain an overview of the results in a graphical format. For more information, see [Dashboard](#).

- In R2014a, on the **Results Summary** pane, you can distinguish between violations of predefined coding rules such as MISRA C or C++ and custom coding rules.

- The predefined rules are indicated by ▼ .
- The custom rules are indicated by ▼ .

In addition, when you click on the **Check** column header on the **Results Summary** pane, the rules are sorted by rule number instead of alphabetically.

- In R2014a, you can double-click a variable name on the **Source** pane to highlight all instances of the variable.

Simplification of coding rules checking

In R2014a, the **Error** mode has been removed from coding rules checking. This mode applied only to:

- The option **Custom** for:
 - **Check MISRA C rules**
 - **Check MISRA AC AGC rules**
 - **Check MISRA C++ rules**
 - **Check JSF C++ rules**
- **Check custom rules**

The following table lists the changes that appear in coding rules checking.

Coding Rules Feature	2013b	2014a
New file wizard for custom coding rules.	<p>For each coding rule, you can select three results:</p> <ul style="list-style-type: none"> • Error: Analysis stops if the rule is violated. <p>The rule violation is displayed on the Output Summary tab in the Project Manager perspective.</p> <ul style="list-style-type: none"> • Warning: Analysis continues even if the rule is violated. 	<p>For each coding rule, you can select two results:</p> <ul style="list-style-type: none"> • On: Analysis continues even if the rule is violated. <p>The rule violation is displayed on the Results Summary pane in the Result Manager perspective.</p> <ul style="list-style-type: none"> • Off: Polyspace does not check for violation of the rule.

Coding Rules Feature	2013b	2014a
	<p>The rule violation is displayed on the Results Summary pane in the Result Manager perspective.</p> <ul style="list-style-type: none"> • Off: Polyspace does not check for violation of the rule. 	
Format of the custom coding rules file.	<p>Each line in the file must have the syntax:</p> <pre><i>rule</i> off error warning #<i>comments</i></pre> <p>For example:</p> <pre># MISRA configuration - Proj1 10.5 off #don't check 10.5 17.2 error 17.3 warning</pre>	<p>Each line in the file must have the syntax:</p> <pre><i>rule</i> off warning #<i>comments</i></pre> <p>For example:</p> <pre># MISRA configuration - Proj1 10.5 off #don't check 10.5 17.2 warning 17.3 warning</pre>

Compatibility Considerations

For existing coding rules files that use the keyword **error**:

- If you run analysis from the user interface, it will be treated in the same way as the keyword **warning**. The verification will not stop even if the rule is violated. The rule violation will however be reported on the **Results Summary** pane.
- If you run analysis from the command line, the verification will stop if the rule is violated.

Support for Windows 8 and Windows Server 2012

Polyspace supports installation and analysis on Windows Server[®] 2012 and Windows 8.

For installation instructions, see Installation, Licensing, and Activation.

Check model configuration automatically before analysis

For the Polyspace Simulink plug-in, the **Check configuration** feature has been enhanced to automatically check your model configuration before analysis. In the **Polyspace** pane of the Model Configuration options, select:

- **On, proceed with warnings** to automatically check the configuration before analysis and continue with analysis when only warnings are found.
- **On, stop for warnings** to automatically check the configuration before analysis and stop if warnings are found.
- **Off** to never check the configuration automatically before an analysis.

If the configuration check finds errors, Polyspace always stops the analysis.

For more information about **Check configuration**, see [Check Simulink Model Settings](#).

Additional back-to-model support for Simulink plug-in

As you click the different links, the corresponding block is highlighted in the model. Because of internal improvements, the back-to-model feature is more stable. Additionally, support has been added for Stateflow[®] charts in Target Link and Linux operating systems.

For more information about the back-to-model feature, see [Identify Errors in Simulink Models](#).

Function replacement in Simulink plug-in

The following functions have been replaced in the Simulink plug-in by the function `pslinkfun`. These functions be removed in a future release.

Function	What Happens?	Use This Function Instead
<code>PolyspaceAnnotation</code>	Warning	<code>pslinkfun('annotations',...)</code>
<code>PolySpaceGetTemplateCFGFile</code>	Warning	<code>pslinkfun('gettemplate')</code>
<code>PolySpaceHelp</code>	Warning	<code>pslinkfun('help')</code>
<code>PolySpaceEnableCOMServer</code>	Warning	<code>pslinkfun('enablebacktomodel')</code>

Function	What Happens?	Use This Function Instead
PolySpaceSpooler	Warning	pslinkfun('queuemanager')
PolySpaceViewer	Warning	pslinkfun('openresults',...)
PolySpaceSetTemplateCFGFile	Warning	pslinkfun('settemplate',...)
PolySpaceConfigure	Warning	pslinkfun('advancedoptions')
PolySpaceKillAnalysis	Warning	pslinkfun('stop')
PolySpaceMetrics	Warning	pslinkfun('metrics')

Polyspace binaries being removed

The following Polyspace binaries will be removed in a future release:

- polyspace-automatic-orange-tester.exe
- polyspace-c.exe
- polyspace-cpp.exe
- polyspace-modularize.exe
- polyspace-remote-c.exe
- polyspace-remote-cpp.exe
- polyspace-remote.exe
- polyspace-report-generator.exe
- polyspace-results-repository.exe
- polyspace-rl-manager.exe
- polyspace-spooler.exe
- polyspace-ver.exe
- setup-remote-launcher.exe

Improvement of floating point precision

In R2013b, Polyspace improved the precision of floating point representation. Previously, Polyspace represented the floating point values with intervals, as seen in the tooltips. Now, Polyspace uses a rounding method.

For example, the verification represents `float arr = 0.1`; as,

- Pre-R2013b, `arr = [9.9999E^-2, 1.0001E-1]`.
- Now, `arr = 0.1`.

R2013b

Version: 9.0

New Features

Proven absence of certain run-time errors in C and C++ code

Use Polyspace Code Prover to prove the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in source code. To verify code, the software uses formal methods-based abstract interpretation techniques. The code verification is static. It does not require program execution, code instrumentation, or test cases. Before compilation and test, you can verify handwritten code, generated code, or a combination of these two types of code.

Color-coding of run-time errors directly in code

Polyspace Code Prover uses color coding to indicate the status of code elements.

- **Green** — Proved to never have a run-time error.
- **Red** — Proved to always have a run-time error.
- **Gray** — Proved to be unreachable, which can indicate a functional issue.
- **Orange** — Unproven, and can have an error.

Errors detected include:

- Overflows, underflows, divide-by-zero, and other arithmetic errors
- Out-of-bounds array access and illegally dereferenced pointers
- Always true/false statement due to dataflow propagation
- Read access operation on uninitialized data
- Dead code
- Access to null `this` pointer (C++)
- Dynamic errors related to object programming, inheritance, and exception handling (C++)
- Uninitialized class members (C++)
- Unsound type conversions

For more information, see Interpret Results.

Calculation of range information for variables, function parameters and return values

Polyspace Code Prover calculates and displays range information associated with, for example, variables, function parameters and return values, and operators. The displayed

range information represents a superset of dynamic values, which the software computes using static methods.

For more information, see [Interpret Results](#).

Identification of variables exceeding specified range limits

By default, Polyspace Code Prover performs a *robustness* verification of your code. The verification proves that the software works under all conditions. As the verification assumes that all data inputs are set to their full range, almost any operation on these inputs can produce an overflow.

To prove that your code works in normal conditions, use the Data Range Specification (DRS) feature to perform contextual verification. You can set constraints on data ranges, and verify your code within these ranges. The use of DRS can substantially reduce the number of orange checks in verification results.

You can use DRS to set constraints on:

- Global variables
- Input parameters for user-defined functions called by the main generator
- Return values for stub functions

For a global variable, if you specify the `globalassert` mode, the software generates a warning when the variable exceeds your specified range.

For more information, see [Data Range Configuration](#).

Quality metrics for tracking conformance to software quality objectives

You can define a quality model with reference to coding rule violations, code complexity, and run-time errors. By observing these metrics, you can track your progress toward predefined software quality objectives as your code evolves from the first iteration to the final version.

By confirming the absence of certain run-time errors and measuring the rate of improvement in code quality, Polyspace Code Prover enables developers, testers, and project managers to produce, assess, and deliver code that is free of run-time errors.

For more information, see [Quality Metrics](#).

Web-based dashboard providing code metrics and quality status

Polyspace Code Prover provides Polyspace Metrics, a Web-based dashboard for tracking submitted verification jobs, reviewing progress, and viewing the quality status of your code. Polyspace Metrics provides an integrated view of project metrics, displaying code complexity, coding rule violations, run-time errors, and other code metrics.

For more information, see [Quality Metrics](#).

Guided review-checking process for classifying results and run-time error status

In the Results Manager perspective, Polyspace Code Prover provides you with several options to organize your review process.

- You can use review methodologies to specify the number and type of checks displayed on the **Results Summary** pane. With each methodology, you review only a subset of checks.

For example, if you are reviewing verification results for the first time, select **First checks to review**. The software displays all red and gray checks but only a subset of orange checks. These orange checks are the ones most likely to be run-time errors. For more information, see [Review Checks Using Predefined Methodologies](#).

- You can group checks by **File/Function** or **Check**:
 - Grouping by **Check** classifies checks by color. Within each color, this grouping classifies checks by categories related to the origin of the check, such as **Control flow**, **Data flow**, and **Numerical**.
 - Grouping by **File/Function** classifies checks by the file where they originated. Within each file, this grouping classifies checks by functions where they originated.
 - For C++ files, you can also group checks by **Class**. This grouping classifies checks by the class definition where they originated.

For more information, see [Organize Check Review Using Filters and Groups](#).

- You can filter checks using any of the column information criteria on the **Results Summary** pane. For example, you can filter out checks that you have already justified using the filter icon on the **Justified** column header. If you have applied a filter, the column heading changes to indicate that all results are not displayed.

You can also define custom filters. For more information, see [Organize Check Review Using Filters and Groups](#).

- You can navigate through the **Results Summary** pane using the keyboard or UI buttons. Both means of navigation respect the grouping, filters, and methodology used to display results.

Graphical display of variable reads and writes

A Polyspace Code Prover verification generates a data dictionary with information about global variables and the read and write access operations on these variables. You can view this information through the **Variable Access** pane of the Results Manager perspective.

For more information, see [Exploring Results Manager Perspective](#).

Comparison with R2013a Polyspace products

Polyspace Code Prover is a single product that replaces the following R2013a products:

- Polyspace Client™ for C/C++
- Polyspace Server™ for C/C++

Polyspace Bug Finder, which is available with the Polyspace Code Prover, incorporates the following R2013a products:

- Polyspace Model Link™ SL
- Polyspace Model Link TL
- Polyspace UML Link™ RH

For a summary of differences and similarities in remote verification, results review and other features and options, expand the following:

Remote verification

Category	R2013a	R2013b
Products required	Install: <ul style="list-style-type: none">• Polyspace Client for C/C++ on local computer	Install: <ul style="list-style-type: none">• MATLAB, Polyspace Bug Finder, and Parallel Computing Toolbox on local computer.

Category	R2013a	R2013b
	<ul style="list-style-type: none"> Polyspace Server for C/C++ on network computers, which are configured as Queue Manager and CPUs. 	<ul style="list-style-type: none"> MATLAB, Polyspace Bug Finder, Polyspace Code Prover, and MATLAB Distributed Computing Server on head node of computer cluster. For information about setting up a cluster, see Install Products and Choose Cluster Configuration.
Configuring and starting services	<p>On the Polyspace Preferences > Server Configuration tab:</p> <ul style="list-style-type: none"> Under Remote configuration, specify host computer for Queue Manager and Polyspace Metrics server and communication port. Under Metrics configuration, specify other settings for Polyspace Metrics. 	<p>On the Polyspace Preferences > Server Configuration tab:</p> <ul style="list-style-type: none"> Under MDCS cluster configuration, specify computer for cluster head node, which hosts the MATLAB job scheduler (MJS). The MJS replaces the R2013a Polyspace Queue Manager. Under Metrics configuration: <ul style="list-style-type: none"> Specify host computer for Polyspace Metrics server and communication port. Specify other settings for Polyspace Metrics.


Category	R2013a	R2013b
	<p>In the Remote Launcher Manager dialog box:</p> <ol style="list-style-type: none"> 1 Under Common Settings, specify Polyspace communication port, user details, and results folder for remote verifications. 2 Under Queue Manager Settings, specify Queue Manager and CPUs. 3 Under Polyspace Server Settings, specify available Polyspace products. 4 To start the Queue Manager and Polyspace Metrics service, click Start Daemon. 	<p>In the Metrics and Remote Server Settings dialog box:</p> <ol style="list-style-type: none"> 1 Under Polyspace Metrics Settings, specify user details, Polyspace communication port, and results folder for remote verifications. 2 Under Polyspace MDCS Cluster Security Settings, you see the following options with default values: <ul style="list-style-type: none"> • Start the Polyspace MDCE service — Selected. The <code>mdce</code> service, which is required to manage the MJS, runs on the MJS host computer and other nodes of the cluster. • MDCE service port — 27350. • Use secure communication – Not selected. Communication is not encrypted. You may want to use communication with security. For information about MATLAB Distributed Computing Server cluster security, see Cluster Security. 3 To start the Polyspace Metrics and <code>mdce</code> services, click Start Daemon. <p>Use the Metrics and Remote Server Settings dialog box to start and stop <code>mdce</code> services only if you configure the MDCS head node as the Polyspace Metrics server. Otherwise, clear the</p>

Category	R2013a	R2013b
		<p>Start the Polyspace MDCE service check box, and use the MDCS Admin Center. To open the MDCS Admin Center, run:</p> <pre><i>MATLAB_Install/toolbox/distcomp/bin/admincenter</i></pre> <p>For information about the MDCS Admin Center, see Cluster Processes and Profiles.</p>
Running a remote verification	<p>In the Project Manager perspective:</p> <ol style="list-style-type: none"> On the Configuration > Machine Configuration pane, select the following check boxes: <ul style="list-style-type: none"> Send to Polyspace Server Add to results repository — Allows viewing of results through Polyspace Metrics. On the toolbar, click Run. <p>The Polyspace client performs code compilation and coding rule checking on the local, host computer. Then the Polyspace client submits the verification to the Queue Manager on your network.</p>	<p>In the Project Manager perspective:</p> <ol style="list-style-type: none"> On the Configuration > Distributed Computing pane, select the Batch check box. By default, the software selects the Add to results repository, which enables the generation of Polyspace Metrics. On the toolbar, click Run. <p>The Polyspace Code Prover software performs code compilation and coding rule checking on the local, host computer. Then the Parallel Computing Toolbox client submits the verification job to the MJS of the MATLAB Distributed Computing Server cluster.</p>
Managing remote verifications	<p>Use the Queue Manager to monitor and manage submitted jobs from Polyspace clients.</p> <p>On the Web, you can monitor jobs through Polyspace Metrics. If you have installed Polyspace Server for C/C++ on your local computer, through Polyspace Metrics, you can open the Queue Manager .</p>	<p>Use the Queue Manager to monitor and manage jobs submitted through Parallel Computing Toolbox clients.</p>

Category	R2013a	R2013b
Accessing results of remote verifications	<p>When you run a verification on a Polyspace server, the Polyspace software automatically downloads the results to your local, client computer. You can view the results in the Results Manager perspective. In addition, you can use the Queue Manager to download results of verifications submitted from other Polyspace clients.</p> <p>On the Web, use Polyspace Metrics to view verification results stored in results repository. If Polyspace Client for C/C++ is installed on your local computer, you can download verification results. For example, in Polyspace Metrics, clicking a cell value in the Run-Time Checks view opens the corresponding verification results in the Results Manager.</p>	<p>On the Web, use Polyspace Metrics to view verification results. If Polyspace Bug Finder is installed on your local computer, you can download verification results. For example, in Polyspace Metrics, clicking a Project cell in the Runs view opens the corresponding verification results in the Results Manager.</p>

Results review

Category	R2013a	R2013b
Results Explorer	<p>Available. Allows navigation through checks by the file and function where they occur. To view, select Window > Show/Hide View > Results Explorer.</p>	<p>Removed. To navigate through checks by file and function, on Results Summary pane, from the drop-down menu, select File/Function.</p>
Filters on the Results Summary pane	<p>Filters appear as icons on the Results Summary pane. You can filter by:</p> <ul style="list-style-type: none"> • Run-time error category • Coding rules violated 	<p>You can filter by the information in all the columns of the Results Summary pane. In addition to existing filters, the new filtering capabilities extend to the file, function and line number where the checks</p>

Category	R2013a	R2013b
	<ul style="list-style-type: none"> • Check color • Check justification • Check classification • Check status 	<p>appear. You can also define your own filters.</p> <p>The filters appear as the  icon on each column header. To apply a filter using the information in a column:</p> <ol style="list-style-type: none"> 1 Place your cursor on the column header. The filter icon appears. 2 Click the filter icon and from the context menu, clear the All box. Select the appropriate boxes to see the corresponding checks. <p>For more information, see Organize Check Review Using Filters and Groups.</p>
Code Coverage Metrics	<p>In the Results Explorer view, the software displays two metrics for the project:</p> <ul style="list-style-type: none"> • unp — Number of unreachable functions as a ratio of total number of functions • cov — Percentage of elementary operations covered by verification <p>The unreachable procedures are marked gray in the Results Explorer view.</p>	<p>The new Results Statistics pane displays the code coverage metrics through the Code covered by verification column graph.</p> <p>To see a list of unreachable procedures, click this column graph.</p> <p>For more information, see Results Statistics.</p>

Other features

Product	Feature	R2013a	R2013b
Polyspace Client and Server for C/C++	Installation	Separate installation process for Polyspace products	Polyspace Code Prover software installed during MATLAB installation process.
	Project configuration	On host, for example, using Polyspace Client for C/C++ software.	On host, using Polyspace Code Prover software.
	Local verification	On host, run Polyspace Client for C/C++ verification. Review results in Results Manager.	On host, run Polyspace Code Prover verification. Review results in Results Manager.
	Export of review comments to Excel [®] , and Excel report generation	Supported	Not supported.
	Line command	<code>polyspace-c ...</code> <code>polyspace-cpp ...</code>	<code>polyspace-code-prover-nodesktop ...</code>
	Project configuration file extension	<code>project_name.cfg</code>	<code>project_name.psprj</code>
	Results file extension	<code>results_name.rte</code>	<code>results_name.pscp</code>
	Configuration > Machine Configuration pane	Available	Replaced by Configuration > Distributed Computing pane.
	Configuration > Post Verification pane	Available	Renamed Configuration > Advanced Settings
	<code>goto</code> blocks	Not supported	Supported

Product	Feature	R2013a	R2013b
	Run verifications from multiple Polyspace environments	Supported	Not supported, produces a license error - 4, 0.
	Non-official options field	Available in Configuration > Machine Configuration pane	Renamed Other and moved to Configuration > Advanced Settings pane
Polyspace Model Link SL and TL	Default includes	Includes specific to the target specified.	Generic includes for C and C++. These includes are target independent.
	Running a verification	Code > Polyspace > Polyspace for Embedded Coder/ Target Link <ul style="list-style-type: none"> • Verify Generated Code • Verify Generated Model Reference Code <p>Also right-clicking on a subsystem and selecting Polyspace > Polyspace for Embedded Coder/ Target Link</p>	Code > Polyspace > Verify Code Generated for <ul style="list-style-type: none"> • Selected Subsystem • Model • Referenced Model • Selected Target Link Subsystem <p>Also right-clicking on a subsystem and selecting Polyspace > Verify Code Generated for > Selected Subsystem / Selected Target Link Subsystem</p>
	Product Mode	Not available.	Choose between Code Prover or Bug Finder depending on the type of analysis you want to run.
	Settings	Available. Called Verification Settings from	Available. Called Settings from . Functionality the same.

Product	Feature	R2013a	R2013b
	Open results	Option Open Project Manager and Results Manager opened the Polyspace Project Manager.	Option Open results automatically after verification opens Polyspace Metrics (batch verifications) or Polyspace Results Manager (local verifications).
Polyspace plug-in for Visual Studio 2010	Support for C++11 features	Partial support.	Added support for: <ul style="list-style-type: none"> • Lambda functions • Rvalue references for <code>*this</code> and initialization of class objects by rvalues • Decltype • Auto keyword for multi-declarator auto and trailing return types • Static assert • Nullptr • Extended friend declarations • Local and unnamed types as template arguments

Options

Product	Option	R2013a	R2013b
	<code>-code-metrics</code>	Available. Not selected by default.	Removed. Code complexity metrics computed by default.

Product	Option	R2013a	R2013b
	-dialect	Available.	Default unchanged, but new value gnu4.6 available for C and C++.
Polyspace Client and Server for C/C++	-max-processes	Specify through Machine Configuration > Number of processes for multiple CPU core systems or command line .	Specify from command line, or through Advanced Settings > Other .
	-allow-language-extensions	Available. Selected by default.	Removed. By default, software supports subset of common C language constructs and extended keywords defined by the C99 standard or supported by many compilers.
	-enum-type-definition	Available with three values. First value called defined-by-standard .	Available with three values. For C, first value renamed signed-int . For C++, first value renamed auto-signed-int-first .

Product	Option	R2013a	R2013b
Polyspace Model Link SL and TL	- scalar - overflows - behavior wrap - around	Available. Not selected by default.	Default. This option identifies generated code from blocks with saturation enabled. However, this option might lead to a loss of precision. For models without saturation, you can choose to remove this option.
	- ignore - constant - overflows	Available. Not selected by default.	Default.

