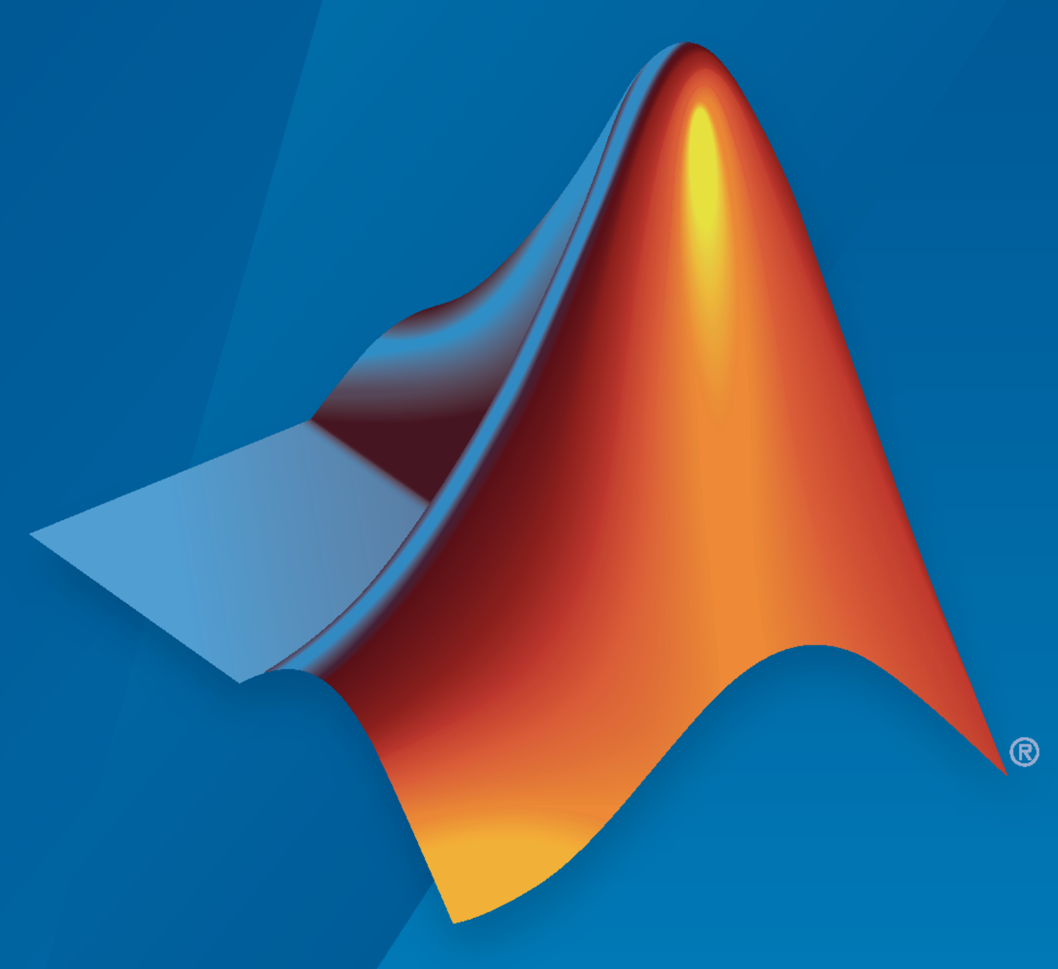
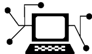
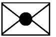
Polyspace® Code Prover™ Release Notes



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.
1 Apple Hill Drive
Natick, MA 01760-2098

Polyspace® Code Prover™ Release Notes

© COPYRIGHT 2013–2020 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.

Contents

R2020b

Verification Setup **1-2**

Compiler Support: Set up Polyspace analysis for code compiled with Renesas SH C compilers **1-2**
Cygwin Support: Create Polyspace projects automatically by using Cygwin 3.x build commands **1-2**
C++17 Support: Run Polyspace analysis on code with C++17 features . . . **1-2**
AUTOSAR Support: Analysis more resilient to ARXML errors **1-3**
AUTOSAR Support: Specify file and folder patterns to exclude from analysis **1-3**
AUTOSAR Support: Specify AUTOSAR software component behaviors and data types using more refined syntax **1-3**
polyspacePackNGo Function: Generate and package Polyspace option files from a Simulink model **1-3**
Polyspace and MATLAB Integration: Integrate Polyspace with MATLAB programmatically without user interaction **1-4**
polyspace.ModelLinkOptions Object: Configure object to analyze code generated as a model reference **1-4**
Configuration from Build System: Generate a project file or analysis options file by using a JSON compilation database **1-5**
Configuration from Build System: Specify how Polyspace imports compiler macro definitions **1-5**
Configuration from Build System: Compiler configuration cached from prior runs for improved performance **1-5**
Changes in analysis options and binaries **1-6**

Verification Results **1-7**

Changes in run-time checks **1-7**
Updated code metrics specifications **1-7**

Reviewing Results **1-9**

Results Export: Export Polyspace results to external formats such as SARIF JSON **1-9**
Simulink Block Annotation: Annotate Simulink blocks from Polyspace user interface to justify Polyspace results **1-9**
User Authentication: Use a credentials file to pass your Polyspace Access credentials at the command line **1-9**
Importing Review Information: Accept information in source or destination results folder in case of merge conflicts **1-10**
Functionality being removed: Polyspace Metrics **1-10**
Functionality being removed: Automatic Orange Tester **1-11**

R2020a

Verification Setup **2-2**

Checking Initialization Code: Analyze initialization code alone before checking remaining program **2-2**
Compiler Support: Set up Polyspace analysis easily for code compiled with MPLAB XC8 C compilers **2-2**
Compiler Support: Set up Polyspace analysis to emulate MPLAB XC16 and XC32 compilers **2-3**
Source Code Encoding: Non-ASCII characters in source code analyzed and displayed without errors **2-3**
Simulink Support: Analyze custom C code in C Function blocks **2-3**
Project Creation from AUTOSAR Configuration: Troubleshoot project creation more easily with resolution hints **2-3**
Changes in analysis options and binaries **2-5**
Changes in MATLAB functions, options object and properties **2-5**

Verification Results **2-6**

Checks on Initialization Code: Verify that global variables are initialized after warm reboot **2-6**
Changes in run-time checks **2-6**

R2019b

Verification Setup **3-2**

Shared Variables Mode: Run a less extensive Code Prover analysis on complete application to compute global variable sharing and usage only **3-2**
Compiler Support: Set up Polyspace analysis easily for code compiled with Cosmic compilers **3-2**
Simulink Support: Analyze generated code by using contextual buttons on the Simulink Editor toolstrip **3-3**
Simulink Support: Verify custom code called from C Caller blocks and Stateflow charts in context of model **3-3**
Simulink Support: Compare two Polyspace result sets and see the effect of changes in model or code generation parameters **3-4**
Configuration from Build System: Compiler version automatically detected from build system **3-5**
Changes in analysis options and binaries **3-6**
Changes in MATLAB functions, options object and properties **3-7**

Verification Results **3-9**

Function Stub Improvements: See fewer orange checks from default conservative assumptions on pointer arguments **3-9**
MISRA C:2012 Directive 4.12: Dynamic memory allocation shall not be used **3-9**

Reviewing Results **3-10**

Code Annotations: Justify Code Prover results by using annotations spread over multiple lines **3-10**

R2019a

Verification Setup **4-2**

Polyspace-only Licenses: Install Polyspace without MATLAB installation **4-2**
New Polyspace Products Supporting Continuous Integration: Perform automated code analysis after code submission with Polyspace Code Prover Server and Polyspace Code Prover Access **4-2**
Offloading Polyspace Analysis to Servers: Use Polyspace desktop products on client side and server products on server side **4-3**
Collaborative Review Support: Upload results from Polyspace user interface to Polyspace Access web interface and share results using web links . . **4-5**
Compiler Support: Set up Polyspace analysis easily for code compiled with ARM v5 and v6 compilers **4-7**
Updated GCC, Clang, and Visual C++ Compiler Support: Set up Polyspace analysis easily for code compiled with GCC versions 7.x, Clang versions 4.x or 5.x, or Microsoft Visual C++ 2017 compilers **4-8**
Simulink Toolstrip: Analyze generated code using contextual buttons in Simulink Editor **4-9**
Changes in analysis options and binaries **4-9**
Changes in MATLAB functions, options object and properties **4-11**

Verification Results **4-13**

Recursion Detection: See list of recursion cycles in C/C++ project **4-13**
Updated code metrics specifications **4-13**

Reviewing Results **4-17**

Source Code Navigation: Keep result pinned while navigating through source code **4-17**
Report Generation: Generate Polyspace reports faster than previous releases **4-19**
Report Generation: Generate single file for HTML reports **4-19**

R2018b

Verification Setup **5-2**

Configuration from Build System: Automatically generate Polyspace configuration modules from build system **5-2**

C11 and C++14 Support: Run Polyspace analysis on code with C11 or C++14 features . . . 5-3
Autodetection of Concurrency Primitives: Multitasking model detected from C11 multithreading functions . . . 5-3
Compiler Support: Set up Polyspace analysis easily for code compiled with Renesas compilers . . . 5-3
AUTOSAR Support: Provide multiple root folders for sources . . . 5-4
AUTOSAR Support: Run Polyspace on AUTOSAR software components by using MATLAB scripts . . . 5-4
AUTOSAR Support: Provide compiler options by tracing your build command . . . 5-4
Function Pointer Calls: Verify functions called through function pointers despite type mismatch . . . 5-5
Check Behavior on Overflows: Fine-tune the behavior of checks based on signedness of integer . . . 5-6
Changes in analysis options and binaries . . . 5-7
Changes in MATLAB option object properties and option values . . . 5-9

Verification Results . . . **5-11**

C++ Specific Checks: View more pertinent results for incorrect object oriented programming and exception handling checks . . . 5-11
Checks on List-Initialization of Arrays: Detect list-initialization with excess initializer clauses (C++11 and beyond) . . . 5-14

Reviewing Results . . . **5-15**

AUTOSAR Support: Focus review to specific software components with queries based on regular expressions . . . 5-15
AUTOSAR Support: See visual representation of runnables and associated files for each software component . . . 5-16
Header Files Access: Open your project header files directly from the point of inclusion . . . 5-19

R2018a

Verification Setup . . . **6-2**

AUTOSAR Support: Set up modular Polyspace analysis for AUTOSAR software components automatically . . . 6-2
MATLAB Coder Support: Run Polyspace on C/C++ code generated from MATLAB code without additional setup . . . 6-3
Compiler Support: Set up Polyspace analysis easily for code compiled with Texas Instruments, IAR or CodeWarrior compilers . . . 6-4
Updated GCC and Clang Compiler Support: Set up Polyspace analysis easily for code compiled with GCC versions 5.x or 6.x, or Clang version 3.x compilers . . . 6-4
Configuration from Build System: Include or exclude sources when generating Polyspace project using polyspace-configure . . . 6-5
Support for IBM Rational Rhapsody to be removed . . . 6-6
Changes in analysis options and binaries . . . 6-6
Changes in MATLAB option object properties . . . 6-9

Verification Results 6-12

AUTOSAR Support: Check for run-time mismatch between AUTOSAR specifications and code implementation 6-12
MISRA C++ Support: Check for overriding of standard library functions, missing const qualifiers and other MISRA C++ rules 6-13
MISRA C:2012 Directives: Detect opportunities for data hiding 6-13
Rule for Source Line Length: Constrain number of characters per line in your code 6-13

Reviewing Results 6-14

Concurrency Modeling: View all tasks and interrupts extracted from code and Polyspace configuration in one view 6-14
Variables Reporting: Export variable list to text file for automated reading 6-15

R2017b

Verification Setup 7-2

Green Hills Compiler Support: Set up Polyspace analysis easily for code compiled with Green Hills Compiler 7-2
OSEK Multitasking Support: Detect the multitasking configuration for your OSEK application automatically 7-2
Polyspace API in MATLAB: Configure analysis, run analysis, and read analysis results with a single MATLAB object 7-3
Compiler-Specific Keywords: Nonstandard compiler-specific keywords are only supported when you specify compiler 7-4
POSIX and BSD Standards: Use functions from these standards without additional setup 7-4
Changes in analysis options and binaries 7-5

Verification Results 7-8

Stack Size Computation: Determine maximum stack usage by a C program and individual functions 7-8
MISRA C:2012 Directive 1.1: Detect instances of implementation-specific behavior in your code 7-8
CERT C Support: Identify CERT C violations using run-time error checks 7-8
Overlapping Memory Detection: Find cases where source and destination arguments of memcpy overlap 7-9
Changes to coding rule checking 7-9

Reviewing Results 7-11

Run-Time Error Cause: Navigate to and view the cause of red nonterminating loops or function calls 7-11
Results Review Workflow: Sort and filter results by subtype 7-12
Result Review Workflow: Hide results that you reviewed once and justified through source code annotations 7-13

Code Annotations: Justify results or define your own format with a new annotation format 7-14
MISRA Comments and Code Annotations: Import your existing MISRA C:2004 justifications to MISRA C:2012 results 7-15
Variable Relationships in Tooltips: Check if variables in operation are related from previous operation 7-16
Result Status: Assign statuses that directly correspond to stages of development workflow 7-17
Function Call Hierarchy: View and navigate to function callers and callees by clicking function name 7-18

R2017a

Verification Setup 8-2

Unified User Interface: Create and maintain a single Polyspace project for Bug Finder and Code Prover analysis 8-2
Improved Speed and Precision: Run analysis faster and receive fewer orange checks as compared to previous releases 8-5
TASKING Compiler Support: Set up Polyspace analysis easily for code compiled with Altium TASKING compiler 8-5
Updated Visual C++ Support: Set up Polyspace analysis easily for code compiled with Microsoft Visual C++ 2015 compiler 8-5
Autodetection of Concurrency Primitives: Multitasking model detected from Windows or µC/OS II multithreading functions 8-6
Manual Multitasking Setup: Functions beginning and ending critical sections do not need to be defined 8-6
Manual Multitasking Setup: main Function Not Required 8-6
Specifying Function Names for Options: Choose from prepopulated list in user interface instead of entering manually 8-6
Polyspace API in MATLAB: Create MATLAB objects from Polyspace projects to run analysis 8-7
Improved support for user implementations of standard library functions 8-8
Improvement in automatic project creation from build systems 8-8
Changes in analysis options and binaries 8-9
Changes in MATLAB options object 8-11
Change in temporary folder location 8-12

Verification Results 8-13

Integers in Floating Point: See improved analysis precision for floating point variables that always take integer values 8-13
New Code Metrics: See number of lines in header files and number of local variables per function 8-13
Checks Green by Definition: Distinguish operations that are safe by definition from operations that are proven safe 8-14
Function Pointer Signature Mismatch: View orange checks instead of red when the mismatch cannot be proven 8-14
Structures with Volatile Fields: See improved analysis precision and apply constraints if necessary 8-15
Changes to coding rule checking 8-15

Reviewing Results 8-17

Easier Review: View verification assumptions, see unreachable and aliased function calls in call graph 8-17
Folder Names in Results: Filter or group analysis results by source folder names 8-18
Code to Model Traceability: Switch easily between identifiers in generated code and corresponding blocks in model 8-18
Polyspace API in MATLAB: Read Polyspace analysis results from MATLAB 8-20

R2016b

Verification Setup 9-2

Diab Compiler Support: Set up Polyspace verification easily for code compiled with Wind River Diab compiler 9-2
Multitasking Code Verification Setup: Specify cyclic tasks and nonpreemptable interrupts directly as verification options 9-2
Improved source and include folder management 9-2
Writable Examples: Modify example projects and restore original versions 9-3
Run verification on .psprj file from the command line 9-3
Polyspace API in MATLAB: Configure and run Polyspace using MATLAB objects 9-3
Configuration Parameters Help: View descriptions of Polyspace options in Simulink configuration parameters 9-4
Eclipse Build Support: Set up Polyspace verification from Eclipse build command 9-4
Visual Studio 2010 add-in support to be removed from installation 9-5
Support for Rhapsody 8.1 9-5
DOS Mode Warning on Linux: Compilation warning for DOS inconsistencies 9-5
Faster Restart for Remote Verification: Reuse compilation results from a previous analysis 9-6
Internal Memory Limits Removed: Expect fewer analysis failures from memory-intensive processes 9-6
Support for local threads 9-6
Changes in Target & Compiler analysis options 9-6
Changes in analysis options and binaries 9-7

Verification Results 9-10

Subnormal Float Detection: Identify loss of precision from operations that lead to subnormal results 9-10
Local Variable Size Estimation: Find total size of local variables in a function 9-10
Changes to coding rule checking 9-10
Metrics for C++ Templates: View code complexity metrics for instances of C ++ templates 9-12
Mutual Exclusion Support: View precise ranges for shared variables protected by critical sections and temporally exclusive tasks 9-12

Improved Embedded Coder Support: View more precise results when generated code uses lookup tables or large data structures **9-14**
Precise Buffer Manipulation Functions: View more precise results on complete copying of structures **9-14**
Assumption for Stubbed Pointers: Review fewer warnings from pointers coming from external code **9-14**
Assumption for Structures with Volatile Fields: Review fewer warnings from partly volatile structures **9-15**
Expected Infinite Loop Detection: Avoid justifying run-time errors on infinite loops that you introduce deliberately **9-15**
Mapping to Standard Functions: View precise results by mapping imprecisely analyzed functions to corresponding standard functions . . **9-16**

Reviewing Results **9-17**

Interactive Graphical Display: Click graphs on Dashboard to filter results **9-17**
Float Range Display: View float variables with narrow ranges more clearly **9-17**
Event History for Coding Rules: Navigate easily between two locations in code that together cause a rule violation **9-17**
Subcheck Display for Standard Library Routines: Determine easily from visual inspection which subcheck failed **9-18**
Results from Macros: Coding rule violations highlighted on macro definitions instead of macro instances **9-18**
Verification Objectives in Eclipse: Create review scopes to focus your review **9-19**
Filtered Report: Reuse result filters for generated report **9-19**
Results Export: Export results to text file for computing graphs and statistics **9-19**
Coding Rule Graphs in Report: View breakdown of coding rules violations by rule number and file **9-19**
Constraints in Report: Add comments about external constraints and view comments in report **9-20**
English Reports in Non-English Locales: Generate English reports on operating systems with a different language **9-20**
Improved PDF report generation **9-21**
Change in report template location **9-21**
Changes in Polyspace User Interface **9-21**

R2016a

Verification Setup **10-2**

Files to Review: Generate results for only specified files and folders **10-2**
Faster MISRA Rule Checking: Check coding rules more quickly and efficiently **10-2**
S-Function Analysis: Launch analysis of S-Function code from Simulink **10-2**
Polyspace Metrics Tomcat Upgrade: Use upgraded default Tomcat server or custom Tomcat version **10-3**

Project Language Flexibility: Change your project language at any time 10-3
External Constraint on Pointers: Specify certain initialization with full range for pointer arguments and return values of stubbed functions 10-3
Source Code Search: Search large applications more quickly 10-4
Polyspace TargetLink plug-in supports data from structures 10-5
Polyspace Eclipse plug-in results location moved 10-5
Improvements in automatic project creation from build command 10-5
Improvements in checking of previously supported MISRA C rules 10-6
Variables with constraints not counted as orange sources 10-6
Changes in analysis options 10-7

Verification Results **10-9**

Floating-Point Support: Propagate ranges more precisely for long double variables and enable verification mode to incorporate infinities and NaNs 10-9
Absolute address usage valid by default 10-11
Run-time checks renamed 10-11

Reviewing Results **10-13**

Autocompletion for Review Comments: Partially type previous comment to select complete comment 10-13
Default Layouts: Switch easily between project setup and results review in user interface 10-13
Persistent Filter States: Apply filters once and view filtered results across multiple runs 10-13
Updated Polyspace Metrics Interface: View summary of project and metrics 10-14
Improved Result Display for File-by-File Verification: View combined summary of results for all files in user interface 10-14
Simplified Variable Access: View task names instead of aliases 10-14

R2015b

Verification Setup **11-2**

Option to Suppress Non-initialization Checks: Customize verification by suppressing non-initialization checks 11-2
Autodetection of Multitasking Primitives: Analyze source code with multitasking primitives from POSIX or VxWorks without manual setup 11-2
Microsoft Visual C++ 2013 Support: Analyze code developed in Microsoft Visual C++ 2013 11-3
GNU 4.9 and Clang 3.5 Support: Analyze code compiled with GCC 4.9 or Clang 3.5 11-3
Improvements in automatic project creation from build command 11-3
Start Page: Get quickly familiar with Polyspace Code Prover 11-4
Saved Layouts: Save your preferred layouts of the Polyspace user interface 11-4
Renaming of labels in Polyspace user interface 11-5

Including options multiple times . . . 11-5
Updated Support for TargetLink . . . 11-6
Improved handling of __declspec . . . 11-6
Changes in analysis options . . . 11-6
Binaries removed . . . 11-11
Support for Visual Studio 2008 to be removed . . . 11-11
Import Visual Studio project removed . . . 11-12

Verification Results . . . **11-13**

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 . . . 11-13
Additional MISRA C:2012 Support: Detect violations of all MISRA C:2012 rules except rules 22.x . . . 11-15
Improved precision for mathematical functions . . . 11-16
Improvements in checking of previously supported MISRA C rules . . . 11-16
Change in Correctness Condition Check . . . 11-18

Reviewing Results . . . **11-19**

Improved Review Capability: View result details and add review comments in one window . . . 11-19
Enhanced Review Scope: Filter coding rule violations from display in one click . . . 11-19
Additional Call Graph Showing Task Creation . . . 11-19
Improvements in Polyspace Metrics workflow . . . 11-20
Improvements in Polyspace Plugin for Eclipse . . . 11-20
Improvements in Report Templates . . . 11-20
Configuration Associated with Result Not Opened by Default . . . 11-21
XML and RTF report formats removed . . . 11-21

R2015a

Verification Setup . . . **12-2**

Simplified workflow for project setup and results review with a unified user interface . . . 12-2
Improvements in search capability in the user interface . . . 12-3
Support for GCC 4.8 . . . 12-4
Polyspace plug-in for Simulink improvements . . . 12-4
Polyspace binaries being removed . . . 12-4
Import Visual Studio project being removed . . . 12-5

Verification Results . . . **12-6**

Detection of stack pointer dereference outside scope . . . 12-6
Isolated ellipsis for variable number of function arguments supported . . . 12-6
Improvement in pointer comparisons . . . 12-7
Improvements in coding rules checking . . . 12-8

Reviewing Results . . . **12-10**

Context-sensitive help for code complexity metrics, MISRA-C:2012, and custom coding rules **12-10**
Review of code complexity metrics and global variable usage in user interface **12-10**
Review of latest results compared to the last run **12-11**
Guidance for reviewing Polyspace Code Prover checks in C code **12-11**
Simplified results infrastructure **12-12**

R2014b

Verification Setup **13-2**

Improved verification speed **13-2**
Support for Mac OS **13-2**
Support for C++11 . . . . . **13-2**
Code Editor for editing source files in Polyspace user interface **13-3**
Local file-by-file verification **13-3**
Simulink plug-in support for custom project files **13-3**
TargetLink support updated **13-3**
AUTOSAR support added **13-4**
Default verification level changed **13-4**
Default mode changed for C++ code verification in user interface **13-4**
Improved global menu in user interface **13-5**
Improved Project Manager perspective **13-5**
Changed analysis options **13-6**
Remote launcher and queue manager renamed **13-6**
Polyspace binaries being removed **13-7**
Import Visual Studio project being removed **13-7**

Verification Results **13-8**

Support for MISRA C:2012 . . . . . **13-8**
Improved verification precision for non-initialized variables **13-8**
New checks for functions not called **13-10**
Improved precision level **13-11**

Reviewing Results **13-12**

Context-sensitive help for verification options and checks **13-12**
Updated Software Quality Objectives **13-12**
Improved Results Manager perspective **13-12**
Error mode removed from coding rules checking **13-14**

R2014a

Verification Setup **14-2**

Automatic project setup from build systems **14-2**

Support for GNU 4.7 and Microsoft Visual Studio C++ 2012 dialects . . . 14-2
Documentation in Japanese 14-2
Preferences file moved 14-3
Support for batch analysis security levels 14-3
Interactive mode for remote verification 14-3
Default text editor 14-3
Support for Windows 8 and Windows Server 2012 14-3
Check model configuration automatically before analysis 14-4
Function replacement in Simulink plug-in 14-4
Polyspace binaries being removed 14-4

Verification Results **14-6**

Support for additional Coding Rules (MISRA C:2004 Rule 18.2, MISRA C++ Rule 5-0-11) 14-6
Improvement of floating point precision 14-6

Reviewing Results **14-7**

Results folder appearance in Project Browser 14-7
Results Manager improvements 14-8
Simplification of coding rules checking 14-9
Additional back-to-model support for Simulink plug-in 14-10

R2013b

Verification Results **15-2**

Proven absence of certain run-time errors in C and C++ code 15-2
Identification of variables exceeding specified range limits 15-2
Graphical display of variable reads and writes 15-2
Calculation of range information for variables, function parameters and return values 15-2

Reviewing Results **15-3**

Color-coding of run-time errors directly in code 15-3
Quality metrics for tracking conformance to software quality objectives 15-3
Web-based dashboard providing code metrics and quality status 15-3
Guided review-checking process for classifying results and run-time error status 15-4
Comparison with R2013a Polyspace products 15-4

R2020b

Version: 10.3

New Features

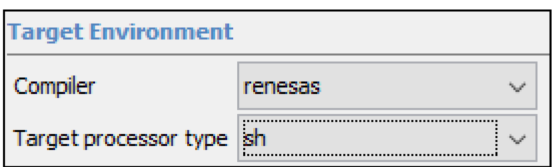
Bug Fixes

Compatibility Considerations

Verification Setup

Compiler Support: Set up Polyspace analysis for code compiled with Renesas SH C compilers

Summary: If you build your source code by using Renesas® SH C compilers, in R2020b, you can specify the target name `sh`, which corresponds to SuperH targets, for your Polyspace analysis.



See also `Renesas Compiler (-compiler renesas)`.

Benefits: You can now set up a Polyspace project without knowing the internal workings of Renesas SH C compilers. If your code compiles with your compiler, it will compile with Polyspace in most cases without requiring additional setup. Previously, you had to explicitly define macros that were implicitly defined by the compiler and remove unknown language extensions from your preprocessed code.

Cygwin Support: Create Polyspace projects automatically by using Cygwin 3.x build commands

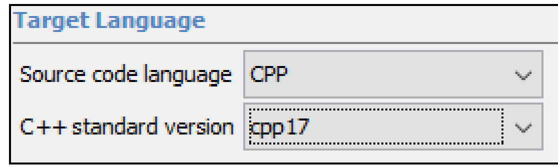
Summary: In R2020b, the `polyspace-configure` command supports version 3.x of Cygwin™ (versions 3.0, 3.1, and so on).

See also “Check if Polyspace Supports Build Scripts”.

Benefits: Using the `polyspace-configure` command, you can trace build scripts that are executed on a Cygwin 3.x command line and create a Polyspace project with the source files and compilation options automatically specified.

C++17 Support: Run Polyspace analysis on code with C++17 features

Summary: In R2020b, Polyspace can interpret the majority of C++17-specific features.



See also:

- `C++ standard version (-cpp-version)`
- C/C++ Language Standard Used in Polyspace Analysis
- “C++17 Language Elements Supported in Polyspace”

Benefits: You can now set up a Polyspace analysis for code containing C++17-specific language elements. Previously, some C++17 specific language elements were not recognized and caused compilation errors.

AUTOSAR Support: Analysis more resilient to ARXML errors

Summary: In R2020b, specific types of ARXML errors do not stop a Code Prover analysis. Despite the errors, the analysis attempts to model the software component behaviors as far as possible and continue into the code extraction and code verification phases.

See also “Interpret Errors and Warnings in Polyspace Analysis of AUTOSAR Code”..

Benefits: You can run a Code Prover analysis more easily on in-progress and incomplete ARXMLs. The ARXML parsing phase reports the errors for each software component behavior. If any of these errors lead to downstream errors during the code extraction phase, you can return to the reports for each software component behavior and track down and fix the ARXML errors.

AUTOSAR Support: Specify file and folder patterns to exclude from analysis

Summary: In R2020b, you can avoid errors in Polyspace analysis from AUTOSAR projects by excluding specific subfolders and files in the source folders up front. For each source folder that you specify, using a Linux-`find`-like syntax, you can specify patterns for file paths that must be excluded.

You can also use a similar file exclusion strategy to exclude files from the ARXML folder.

For more information, see:

- “Select AUTOSAR XML (ARXML) and Code Files for Polyspace Analysis”
- `polyspace-autosar Command`

Benefits: Previously, you could only specify a root folder for your ARXML and source files. The finer file-selection allows you to avoid folders that might cause errors in project setup.

AUTOSAR Support: Specify AUTOSAR software component behaviors and data types using more refined syntax

Summary: In R2020b, you can use a more refined syntax when specifying AUTOSAR software component behaviors to analyze or when importing data types. Using a Linux-`find`-like syntax, you can specify inclusion or exclusion patterns for the fully qualified names of behaviors or types.

For more information, see `polyspace-autosar Command`.

polyspacePackNGo Function: Generate and package Polyspace option files from a Simulink model

Summary: In R2020b, you can package Polyspace option files along with code generated from a Simulink® model, and then analyze the code on a different machine in a distributed workflow. After packaging the generated code, create and archive options files required for a Polyspace analysis by using the `polyspacePackNGo` function.

See also:

- `polyspacePackNGo`
- “Run Polyspace Analysis on Generated Code by Using Packaged Options Files”

Benefits: In a distributed workflow, a Simulink user generates code from a model and sends the code to another development environment. In this environment, a Polyspace user analyzes the generated code by using design ranges and other model-specific information. Previously, in this distributed workflow, you configured the Polyspace analysis options manually. Starting in R2020b, you do not have to manually create the option files when analyzing generated code by using Polyspace in a distributed workflow.

Polyspace and MATLAB Integration: Integrate Polyspace with MATLAB programmatically without user interaction

Summary: In R2020b, use simpler steps to integrate Polyspace and MATLAB®. Instead of browsing to a specific subfolder of the Polyspace installation folder, and then running the `polyspacesetup` function, run `polyspacesetup` from any folder:

```
polyspacesetup('install', 'polyspaceFolder', folder);
```

`folder` is the location of the Polyspace installation in your machine. To integrate Polyspace with MATLAB without user interaction, use:

```
polyspacesetup('install', 'polyspaceFolder', folder, 'silent', true);
```

See:

- `polyspacesetup`
- “Integrate Polyspace with MATLAB and Simulink”

Benefits: Previously, integrating Polyspace with MATLAB required user interaction. Starting in R2020b, you can perform the integration programmatically and silently.

polyspace.ModelLinkOptions Object: Configure object to analyze code generated as a model reference

Summary: In R2020b, you can configure a `polyspace.ModelLinkOptions` object to analyze code generated as a model reference by using the new optional argument `asModelRef`. To run a Polyspace analysis on the code generated as a model reference, create a `polyspace.ModelLinkOptions` object and set the `asModelRef` flag to `true`. See also:

- `polyspace.ModelLinkOptions`
- “Analyze Code Generated as Model Reference”

Benefits: Previously, the class `polyspace.ModelLinkOptions` did not support analyzing code generated as model reference. Starting in R2020b, you can run a Polyspace analysis on code generated as a model reference by using the class `polyspace.ModelLinkOptions`. You can also set the options for the Polyspace analysis by using a `pslinkoptions` object.

Configuration from Build System: Generate a project file or analysis options file by using a JSON compilation database

Summary: In R2020b, if your build system supports the generation of a JSON compilation database, you can create a Polyspace project file or an analysis options file from your build system without tracing your build process. After you generate the JSON compilation database file, pass this file to `polyspace-configure` by using the option `-compilation-database` to extract your build information.

For more information on compilation databases, see JSON Compilation Database.

Benefits: Previously, you had to invoke your build command and trace your build process to extract the build information. For some build systems such as Bazel, `polyspace-configure` could not always trace the build process, resulting in errors when running an analysis by using the generated options file.

Configuration from Build System: Specify how Polyspace imports compiler macro definitions

Summary: In R2020b, when you use `polyspace-configure` to create a Polyspace project file or to generate an analysis options file from your build system, you can specify how Polyspace imports the compiler macro definitions.

Use option `-import-macro-definitions` and specify:

- `none` — Skip the import of macro definition. You can provide macro definitions manually instead.
- `from-whitelist` — Use a Polyspace white list to query your compiler for macro definitions.
- `from-source-token` — Use all non-keyword tokens in your source files to query your compiler for macro definitions.

See also `polyspace-configure`.

Benefits: Previously, Polyspace used all non-keyword tokens in your source files to query your compiler for macro definitions each time that you traced your build command. You now have greater control on the import of macro definitions.

Configuration from Build System: Compiler configuration cached from prior runs for improved performance

Summary: In R2020b, when you use `polyspace-configure` to create a Polyspace project file or to generate an analysis options file from your build system, Polyspace caches your compiler configuration. If your compiler configuration does not change, Polyspace reuses the cached configuration during subsequent runs of `polyspace-configure`.

See also `polyspace-configure`.

Benefits: Previously, Polyspace did not cache your compiler configuration. Instead, during every run of `polyspace-configure`, Polyspace queried your compiler for the size of fundamental types, compiler macro definitions, and other compiler configuration information. Starting R2020b, the caching improves the later `polyspace-configure` runs.

Changes in analysis options and binaries

Option -consider-external-arrays-as-unsafe also applies to C code

In R2020b, the option `-consider-external-arrays-as-unsafe` also applies to C code. The option removes the default Code Prover assumption that external arrays of unspecified size can be safely accessed at any index. Previously, the option was available only for C++ code.

See also `-consider-external-array-access-unsafe`.

Verification Results

Changes in run-time checks

Summary: In R2020b, you see these changes in the results of Code Prover run-time checks.

Check	Change

`Non-initialized variable` and `Non-initialized local variable`	If all fields of a structure are unused and uninitialized, checks for initialization on this structure are orange.
	
	Previously, if none of the fields of a structure were used later, the checks considered the structure as initialized. For instance, in this code:
	
	`typedef struct { int a; char c; } S;`
	
	`void foo(void) {`
	`  S s;`
	`  S s1;`
	
	`  s = s1;`
	`}`
	
	the **Non-initialized local variable** check on `s1` in `s = s1` is orange. Prior to R2020b, the check was green because even though the structure fields are uninitialized, they are not used later.

Compatibility Considerations

You can see a change in the number of results flagged by the updated run-time checks.

Updated code metrics specifications

Summary: In R2020b, these code metrics specifications have been updated.

<table>
<tr><th>Code Metric</th><th>Update</th></tr>
<tr><td><code>Number of Called Functions</code></td><td>These metrics now accounts for function calls in a C++ constructor initializer list.

For instance, in this code snippet, the number of called functions of <code>Derived::Derived()</code> is one. Previously, the number was computed as zero.

<pre>class Base
{
  int b;
  public:
      Base() {
          b = 0;
      };
};
class Derived : public Base
{
  int d;
  public:
      Derived() : Base() {
          d = 0;
      };
};</pre></td></tr>
</table>

Compatibility Considerations

If you compute these code metrics, you can see a difference in results compared to previous releases.

Reviewing Results

Results Export: Export Polyspace results to external formats such as SARIF JSON

Summary: In R2020b, you can use the new `polyspace-results-export` command to export Polyspace results to formats such as JSON and CSV.

- The JSON object follows the Static Analysis Results Interchange Format or SARIF notation.
- The CSV file has the same fields as produced by using the earlier `polyspace-report-generator` command with the `-generate-results-list-file` option.

  Use the `polyspace-report-generator` command to generate PDF or Word reports in a predefined format. To package results using your own format, export them using the `polyspace-results-export` command and read the resulting JSON object or CSV file.

You can use this command with results generated locally or with results uploaded to Polyspace Access.

See also `polyspace-results-export`.

Benefits: Using the JSON object or CSV file, you can display results in a convenient format. For instance, you can group defects found by Bug Finder based on their impact. Because the JSON object follows a standard notation, you can also use this format to display Polyspace results with results from other tools.

Simulink Block Annotation: Annotate Simulink blocks from Polyspace user interface to justify Polyspace results

Summary: In R2020b, you can annotate a Simulink block directly from the Polyspace user interface. See “Annotate Blocks to Justify Issues”.

Benefits: Previously, when annotating a check on generated code from the Polyspace user interface, you had to locate the corresponding block in the Simulink Editor and annotate the block again. Starting in R2020b, you can annotate a check in the Polyspace user interface and have the annotations carry over to the Simulink blocks by using the traceable elements of the code. You do not have to go back to the model to re-enter the annotation.

User Authentication: Use a credentials file to pass your Polyspace Access credentials at the command line

Summary: In R2020b, if you use a command that requires your Polyspace Access credentials, you can save these credentials in a file that you pass to the command. If you use that command inside a script, you no longer need to store your credentials in the script.

To create a credentials file, enter a set of credentials, either as `-login` and `-encrypted-password` entries on separate lines, for example:

```
-login jsmith
-encrypted-password LAMMMEACDMKEFELKMNDCONEAPECEEKPL
```

Or as a `-api-key` entry:

```
-api-key keyValue123
```

For more information on generating API keys, see "Configure User Manager" (Polyspace Code Prover Access).

Save the file and pass it to the command by using the `-credentials-file` flag. You can use the credentials file with these Polyspace commands:

- `polyspace-access`
- `polyspace-results-export`
- `polyspace-report-generator`

For increased security, restrict the read/write permissions for the credentials file.

Benefits: Previously, you could provide your Polyspace Access credentials in a script only by passing them directly to the command. Starting R2020b, when the command that requires the credentials runs, someone who is inspecting currently running processes, for instance, by using the command `ps aux` on Linux, can no longer see your credentials.

Importing Review Information: Accept information in source or destination results folder in case of merge conflicts

Summary: In R2020b, when importing review information such as severity, status, and comments at the command line, if the same result has different review information in the source and destination folder, you can choose one of the following:

- That the review information in the destination folder is retained.

  This behavior is the default behavior of the `polyspace-comments-import` command.
- That the review information in the source folder overwrites the information in the destination folder.

  You can switch to this behavior using the new option `-overwrite-destination-comments`.

See also `polyspace-comments-import`.

Benefits: Previously, newer review information in the destination folder was retained and could not be overwritten. Now, when merging review information, you can choose whether the source or destination folder takes precedence in case of merge conflicts.

Functionality being removed: Polyspace Metrics

Summary: The Polyspace Metrics web dashboard will be removed in a future release.

Compatibility Considerations

To continue monitoring the quality of your code in a web browser, use Polyspace Access instead. In addition to a more intuitive dashboard, with Polyspace Access you can:

- Review and justify results directly from your web browser.

- Integrate a bug tracking tool such as Jira with the web interface and create tickets to track Polyspace findings.
- Monitor the quality of your code against coding standards such as AUTOSAR C++14, CERT® C/C++, and MISRA C®.
- Define custom Quality Objectives definitions and apply them to specific projects.

For more information, see “Polyspace Code Prover Access”.

See also “Migrate Results from Polyspace Metrics to Polyspace Access” (Polyspace Code Prover Access).

Functionality being removed: Automatic Orange Tester

The Automatic Orange Tester will be removed in a future release.

Compatibility Considerations

If you use the Automatic Orange Tester with your Polyspace project, unset the corresponding options. In the desktop interface, you set these options in the **Configuration** pane under the **Advanced Settings** node.

If you use these command-line options in your scripts, remove them:

- `-automatic-orange-tester`
- `-automatic-orange-tester-loop-max-iteration`
- `-automatic-orange-tester-tests-number`
- `-automatic-orange-tester-timeout`

If you use these properties in your MATLAB scripts, remove them (`opts=polyspace.Options('C')`):

- `opts.Advanced.AutomaticOrangeTester`
- `opts.Advanced.AutomaticOrangeTesterLoopMaxIteration`
- `opts.Advanced.AutomaticOrangeTesterTestsNumber`
- `opts.Advanced.AutomaticOrangeTesterTimeout`

R2020a

Version: 10.2

New Features

Bug Fixes

Compatibility Considerations

Verification Setup

Checking Initialization Code: Analyze initialization code alone before checking remaining program

Summary: In R2020a, you can mark off a section of code as initialization code and check for run-time errors only in this section.

For instance, in this example, the initialization code starts from the beginning of `main` and continues up to the pragma `polyspace_end_of_init`. The verification stops when the pragma is encountered.

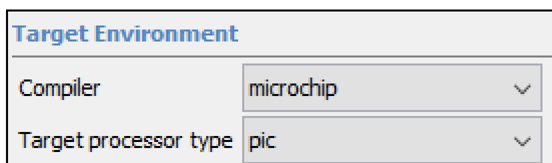
```
#include <limits.h>

int aVar;
const int aConst = INT_MAX;
int anotherVar;

int main() {
      aVar = aConst + 1;
#pragma polyspace_end_of_init
      anotherVar = aVar - 1;
      return 0;
}
```

For more information, see `Verify initialization section of code only (-init-only-mode)`.

Benefits: Often, issues in the initialization code can invalidate the analysis of the remaining code. For instance, in the preceding example, the overflow in the line `aVar = aConst+1` must be fixed first before the value of `aVar` is used in subsequent code. Now, you can check the initialization code alone and fix the issues found before verifying the remaining program.

Compiler Support: Set up Polyspace analysis easily for code compiled with MPLAB XC8 C compilers

Summary: If you build your source code by using MPLAB XC8 C compilers, in R2020a, you can specify the compiler name for your Polyspace analysis.



See also `MPLAB XC8 C Compiler (-compiler microchip)`.

Benefits: You can now set up a Polyspace project without knowing the internal workings of MPLAB XC8 C compilers. If your code compiles with your compiler, it will compile with Polyspace in most cases without requiring additional setup. Previously, you had to explicitly define macros that were implicitly defined by the compiler and remove unknown language extensions from your preprocessed code.

Compiler Support: Set up Polyspace analysis to emulate MPLAB XC16 and XC32 compilers

Summary: If you use MPLAB XC16 or XC32 compilers to build your source code, in R2020a, you can easily emulate these compilers by using the Polyspace GCC compiler options. See Emulate Microchip MPLAB XC16 and XC32 Compilers.

For each compiler, you can emulate these target processor types:

- **MPLAB XC16**: Targets PIC24 and dsPIC.
- **MPLAB XC32**: Target PIC32.

Benefits: You can copy the analysis options required for emulating MPLAB XC16 or XC32 compilers and paste into your Polyspace options file (or specify in a Polyspace project in the user interface), and avoid compilation errors from issues specific to these compilers.

Source Code Encoding: Non-ASCII characters in source code analyzed and displayed without errors

Summary: In R2020a, if your source code contains non-ASCII characters, for instance, Japanese or Korean characters, the Polyspace analysis can interpret the characters and later display the source code correctly.

If you still have compilation errors or display issues from non-ASCII characters, you can explicitly specify your source code encoding using the option `Source code encoding (-sources-encoding)`.

Simulink Support: Analyze custom C code in C Function blocks

Summary: In R2020a, Polyspace can check custom C code in C Function blocks for bugs and run-time errors.

The analysis checks the C code in context of the model. In other words, the analysis uses design ranges and other context information specified in the model.

To analyze custom C code in C Function block, select **Custom Code Used in Model** instead of **Code Generated as Top Model** (meant for generated code) on the **Polyspace** tab in Simulink and then start the analysis. In addition to functions called from C Caller blocks and Stateflow charts, the custom code in C Function blocks are also checked for run-time errors. See Run Polyspace Analysis on Custom Code in C Function Block.

Benefits: The Polyspace analysis of custom code now includes individual scripts in C Function blocks (block introduced in Simulink in R2020a). In a single run, you can analyze all handwritten C code invoked from your model and check for bugs, run-time errors or coding rule violations.

Project Creation from AUTOSAR Configuration: Troubleshoot project creation more easily with resolution hints

Summary: In R2020a, when creating a Polyspace project from your ARXML and source files, if you run into errors during code extraction, the analysis provides helpful resolution hints for these common code extraction issues:

- Undefined data types
- Include files not found

To see the resolution hints, in the file `psar_project.xhtml`, click the ☰ button on the upper left, then click **Behaviors**. On the **Behaviors** tab, below the errors in the code extraction phase, click the link to see a summary of code-extraction diagnostics with possible resolution hints.

Extract implementation code for **89** AUTOSAR behaviors with proof artifacts:

- noRunnableImplementation (30)
- error_noRunnableImplementationTopFileError (3)
- error_atLeastOneRunnableInFileThatDoesNotCompile (23)
- subsetOfRunnablesImplementation (3)
- allRunnablesImplementation (30)

See summary of code-extraction diagnostics with possible resolution hints

Execution reported errors and warnings. Reported errors See detailed log messages

The resolution hints are of two types:

- For undefined data types, the analysis proposes matches from the ARXML that could possibly be used for the data types. You can specify these data types with the `polyspace-autosar` option `-autosar-datatype`.
- For missing include files, the analysis proposes matches for the include files from subfolders of the source folder. You can specify these include files with the `polyspace-autosar` option `-I`.

Instead of fixing individual code extraction errors using the resolution hints, you can download a file with all options that implement the hints. On the summary page, click the link **Download polyspace-autosar options**.

Summary of polyspace-autosar code-extraction diagnostics

Lists diagnostics that are reported when extracting the implementation-code of one or more AUTOSAR behaviors.
Each diagnostic may have "resolution-hints" which are specific to the class of error.
Resolution-hints can translate to polyspace-autosar options that you may add to your project Download polyspace-autosar options

You can use the downloaded text file with the `polyspace-autosar` option `-options-file` to implement the resolution hints in one shot.

For information on the options, see `polyspace-autosar`.

Benefits: Most issues during the code extraction phase of Polyspace project creation come from data types not found or missing include files. You can now address a significant fraction of these issues with a simple one-click resolution.

Changes in analysis options and binaries

Option -function-behavior-specifications renamed to -code-behavior-specifications and capabilities extended

Warns

The option `-function-behavior-specifications` has been renamed to `-code-behavior-specifications`.

Using this option, you could previously map your functions to standard library functions to work around analysis imprecisions or specify thread creation routines. Now, you can use the option to define a blacklist of functions to forbid from your source code.

See also `-code-behavior-specifications`.

Changes in MATLAB functions, options object and properties

polyspaceCodeProverNodesktop removed

Errors

Use `polyspaceCodeProver(`*`projectFile`*`, '-nodesktop')` instead of `polyspaceCodeProverNodesktop(`*`projectFile`*`)`.

pslinksetup removed

Errors

Use `polyspacesetup` instead of `pslinksetup` to integrate between Polyspace and Simulink (in the same release or across releases). See Integrate Polyspace with MATLAB and Simulink.

Verification Results

Checks on Initialization Code: Verify that global variables are initialized after warm reboot

Summary: In R2020a, you can mark off a section of a C program as initialization code and verify if all non-const global variables are explicitly initialized at declaration or in that section.

For instance, in this simple example, the initialization code starts from the beginning of `main` and continues up to the pragma `polyspace_end_of_init`. The global variable `aVar` is initialized in this section but the variable `anotherVar` is not.

```
int aVar;
const int aConst = -1;
int anotherVar;

int main() {
      aVar = aConst;
#pragma polyspace_end_of_init
      return 0;
}
```

For more information, see:

- `Check that global variables are initialized after warm reboot (-check-globals-init)`
- `Global variable not assigned a value in initialization code`

Benefits: In a warm reboot, to save time, the bss segment of a program, which might hold variable values from a previous state, is not loaded. Instead, the program is supposed to explicitly initialize all non-const variables without default values before execution. You can now delimit this initialization code and verify that all non-const global variables are indeed initialized in a warm reboot.

Changes in run-time checks

Summary: In R2020a, you see these changes in the results of Code Prover run-time checks.

<table>
<tr><th>Check</th><th>Change</th></tr>
<tr><td><code>Uncaught exception</code></td><td>The check no longer flags the case where a function throws an exception whose data type is not in the list of exception types in the function declaration.

For instance, the function <code>foo</code> is declared to throw exceptions of type <code>int</code> and <code>std::exception</code>:

<code>void foo2() throw(std::exception, int);</code>

Code Prover used to check if the function can throw exceptions outside the specified types. The check is not performed from R2020a onwards.

Dynamic exception specification using the above syntax is deprecated in C++11 and removed in the later standard C++17. See also Dynamic exception specification in the C++ standard.</td></tr>
</table>

Compatibility Considerations

You can see a change in the number of results flagged by the updated run-time checks.

R2019b

Version: 10.1

New Features

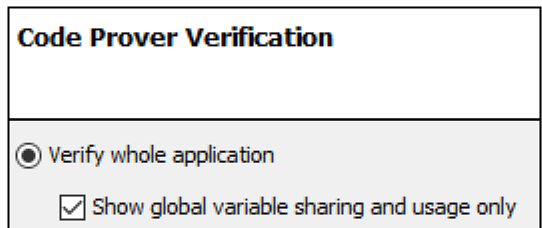
Bug Fixes

Compatibility Considerations

Verification Setup

Shared Variables Mode: Run a less extensive Code Prover analysis on complete application to compute global variable sharing and usage only

Summary: In R2019b, you can run a less extensive Code Prover analysis on your complete application to see a list of all global variables and their sharing and usage.



In this mode, the analysis stops before the run-time error detection phase and the results contain:

- Global variables (shared, unshared, used, unused)
- Coding rules, if coding rule checking is enabled
- Code metrics, if code metrics computation is enabled

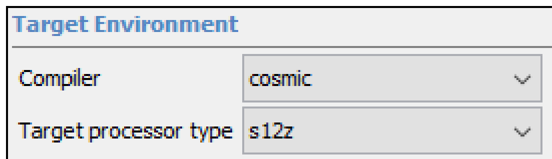
The **Variable Access** pane shows all read and write operations on global variables.

See also `Show global variable sharing and usage only (-shared-variables-mode)`.

Benefits: You can now run Code Prover on your entire application to see global variable sharing and usage, and then run Code Prover component-by-component for run-time error detection. Previously, global variables were reported only in the full analysis that included run-time error detection. Run-time error detection can sometimes take significantly longer for complete applications. If you want to review only the global variable sharing and usage in your complete application, you no longer require the full analysis.

Compiler Support: Set up Polyspace analysis easily for code compiled with Cosmic compilers

Summary: If you build your source code by using Cosmic compilers, in R2019b, you can specify the compiler name for your Polyspace analysis.



See also `Cosmic Compiler (-compiler cosmic)`.

Benefits: You can now set up a Polyspace project without knowing the internal workings of Cosmic compilers. If your code compiles with your compiler, it will compile with Polyspace in most cases

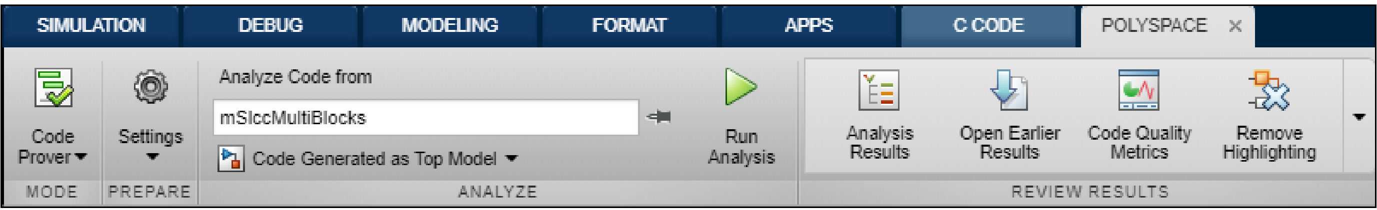
without requiring additional setup. Previously, you had to explicitly define macros that were implicitly defined by the compiler and remove unknown language extensions from your preprocessed code.

Simulink Support: Analyze generated code by using contextual buttons on the Simulink Editor toolstrip

Summary: In R2019b, a toolstrip with contextual buttons replaces the menus and toolbars in the Simulink Editor. For details, see release notes.

Code generation and verification tasks appear in separate tabs on the Simulink toolstrip.

- To generate code, open the **C Code** tab. To access this tab, on the **Apps** tab, select **Embedded Coder**.
- To analyze the generated code, open the **Polyspace** tab. To access this tab, on the **Apps** tab, select **Polyspace Code Verifier**.



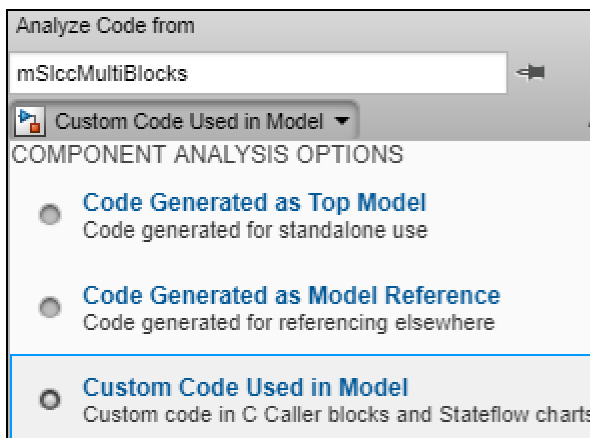
Benefits: The Simulink toolstrip includes contextual tabs, which appear only when you need them.

Additional Considerations

All menu items available earlier in the submenu **Code** > **Polyspace** now appear on the **Polyspace** tab. See Changes in Polyspace Analysis Workflows in Simulink in R2019b.

Simulink Support: Verify custom code called from C Caller blocks and Stateflow charts in context of model

Summary: In R2019b, Polyspace can check functions called from C Caller blocks and Stateflow® charts for bugs and run-time errors. The analysis extracts the functions' inputs and other call context information from the model.



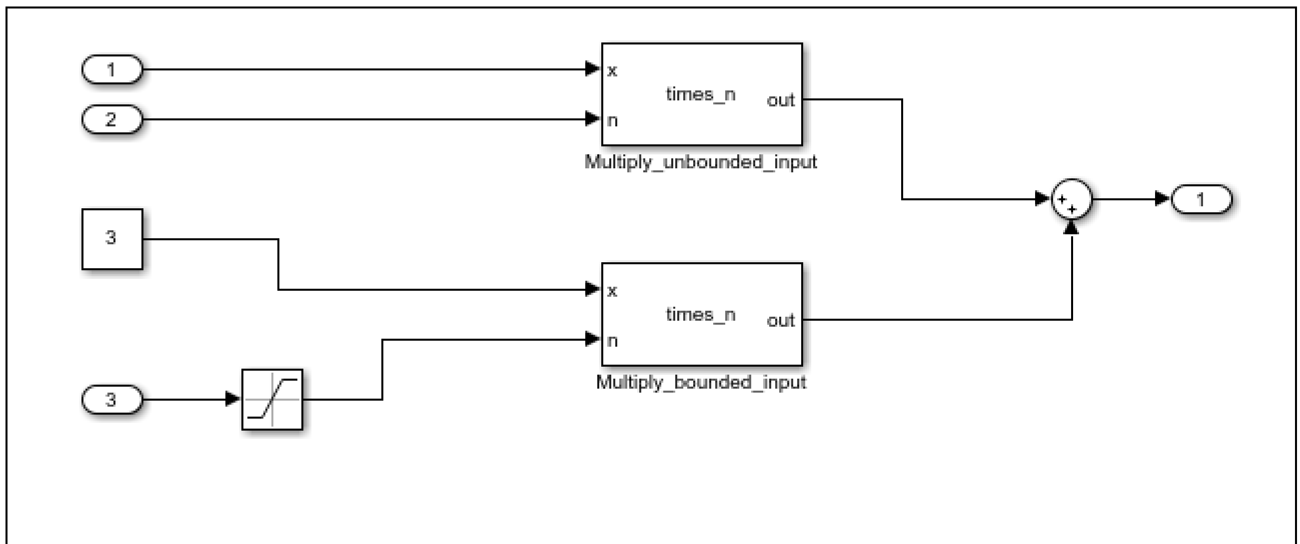
See Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts.

Benefits:

- Check whether handwritten code called from model has issues:

  You typically use model verification software such as Simulink Design Verifier™ to check for bugs and run-time errors in a model. The model verification software shows only a small subset of run-time errors in handwritten code loaded on C Caller blocks and Stateflow charts. With Polyspace, you can check for bugs, run-time errors, coding standard violations and many other issues in handwritten code directly from your Simulink model and supplement the checks at the model level.
- Use call context information for handwritten functions from signal ranges in model:

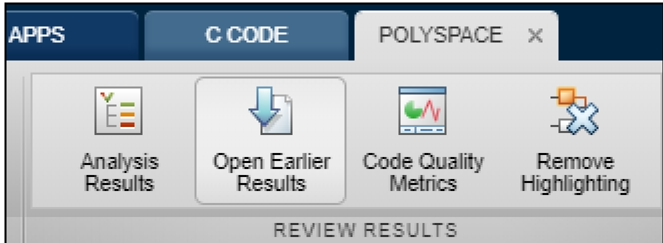
  The analysis uses call context information from the model. For instance, in this simple model, the function `times_n` is called in two C caller blocks (named `Multiply_unbounded_input` and `Multiply_bounded_input`).



When you analyze custom code, in this case the function `times_n`, the analysis shows that an operation in the custom code can overflow. From the analysis results, you can determine that the overflow occurs only when the function is called in the `Multiply_unbounded_input` block but not when it is called from the `Multiply_bounded_input` block.

Simulink Support: Compare two Polyspace result sets and see the effect of changes in model or code generation parameters

Summary: In R2019b, you can open previous Polyspace results on a model directly from the Simulink Editor. You can look at two Polyspace result sets for side-by-side comparison.

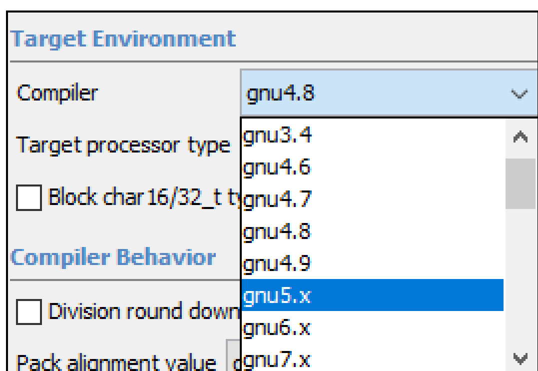


Benefits: Previously, you could open only the latest result from the Simulink editor. To open a previous result, you had to locate the result outside Simulink in your file explorer and open the result in the Polyspace user interface. You can now perform these actions more easily:

- Change a section of the model or a code generation option, regenerate code, rerun Polyspace, open the new results, and compare with a previous result.
- Change a Polyspace analysis option, rerun Polyspace, open the new results, and compare with a previous result.

Configuration from Build System: Compiler version automatically detected from build system

Summary: In R2019b, if you create a Polyspace analysis configuration from your build system by using the `polyspace-configure` command or in the user interface, the analysis uses the correct compiler version for the option `Compiler (-compiler)` for GNU® C, Clang, and Microsoft® Visual C++® compilers. You do not have to change the compiler before starting the Polyspace analysis.



Benefits: Previously, if you traced your build system to create a Polyspace analysis configuration, the latest supported compiler version was used in the configuration. If your code was compiled with an earlier version, you might encounter compilation errors and have to explicitly specify an earlier compiler version before starting the analysis.

For instance, if the Polyspace analysis configuration uses the version GCC 4.9 and some of the standard headers in your GCC version include the file `x86intrin.h`, you can see a compilation error such as this error:

```
/usr/lib/gcc/x86_64-linux-gnu/6/include/avx512bwintrin.h, line 2427:
                                        error: invalid type conversion
|    return (__m512i) __builtin_ia32_packssdw512_mask ((__v16si) __A,
|
```

You had to connect the error to the incorrect compiler version, and then explicitly set a different version. Now, the compiler version is automatically detected when you create a project from your build command.

Changes in analysis options and binaries

Check MISRA C:2012 (-misra3) option values CERT-rules, CERT-all, and ISO-17961 are removed

Warns

`Check MISRA C:2012 (-misra3)` option values `CERT-rules`, `CERT-all`, and `ISO-17961` are removed. Previously, you used **Check MISRA C:2012 (-misra3)** with these options values to check your code for violations of the CERT C and ISO/IEC TS 17961 coding standards. Use a Polyspace Bug Finder™ analysis with the new **Coding Standards & Code Metrics** analysis options `Check SEI CERT-C (-cert-c)` and `Check ISO/IEC TS 17961 (-iso-17961)` instead.

The new analysis options simplify checking for violations of coding standards CERT C and ISO/IEC TS 17961. For more information, see Changes in Coding Standard Checking in R2019a (Polyspace Bug Finder) (Polyspace Bug Finder)

You get a warning when you use the removed option values. Polyspace automatically replaces the removed option value with the value indicated in this table.

Option	Replace by
`-misra3 CERT-rules`	`-misra3 all`
`-misra3 CERT-all`	
`-misra3 ISO-17961`	

Check MISRA C++ rules (-misra-cpp) option values CERT-rules and CERT-all are removed

Warns

`Check MISRA C++:2008 (-misra-cpp)` option values `CERT-rules` and `CERT-all` are removed. Previously, you used **Check MISRA C++ rules (-misra-cpp)** with these options values to check your code for violations of the CERT C++ coding standards. Use a Polyspace Bug Finder analysis with the new **Coding Standards & Code Metrics** analysis option `Check SEI CERT-C++ (-cert-cpp)` instead.

The new analysis option simplifies checking for violations of the CERT C++ coding standard. For more information, see Changes in Coding Standard Checking in R2019a (Polyspace Bug Finder) (Polyspace Bug Finder)

You get a warning when you use the removed option values. Polyspace automatically replaces the removed option value with the value indicated in this table.

Option	Replaced by
`-misra-cpp CERT-rules`	`-misra-cpp all-rules`
`-misra-cpp CERT-all`	

Changes in MATLAB functions, options object and properties

Direct file specification not allowed for CodingRulesCodeMetrics properties that denote rule subsets
Errors

The properties of a `polyspace.Project` object that indicate coding rule subsets no longer take a text file as argument. To specify a custom subset of rules, instead of specifying a text file directly, use the value `from-file` and then specify an XML file using the `CheckersSelectionByFile` property. For instance, if `proj` is a `polyspace.Project` object, instead of:

```
proj.Configuration.CodingRulesCodeMetrics.MisraCppSubset = 'C:\rules.txt';
```

use:

```
proj.Configuration.CodingRulesCodeMetrics.MisraCppSubset = 'from-file';
proj.Configuration.CodingRulesCodeMetrics.EnableCheckersSelectionByFile = true;
proj.Configuration.CodingRulesCodeMetrics.CheckersSelectionByFile = 'C:\rules.xml';
```

where `rules.xml` contains the same specifications as `rules.txt`.

You can convert existing text files into XML files in the Polyspace user interface. In the **Coding Standards & Code Metrics** node of the **Configuration** pane, click . In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML file, and saves this file as `*filename*.xml`, where *`filename`* is the name of the first selected file alphabetically. For instance, if you select the text files `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

The change affects these subproperties of the `CodingRulesCodeMetrics` property:

- `AcAgcSubset`
- `JsfSubset`
- `MisraC3Subset`
- `MisraCSubset`
- `MisraCppSubset`

See also `polyspace.Project.Configuration Properties`.

Format for specifying properties of polyspace.CodingRulesOptions object changed
Errors

The properties of the `polyspace.CodingRulesOptions` object are now grouped into sections. Instead of specifying a rule directly, specify the containing section first and then the rule.

For instance, if `rules` is a `polyspace.CodingRulesOptions` object that specifies MISRA C:2012 rules, instead of:

```
rules.rule_2_1 = false;
```

use:

```
rules.Section_2_Unused_code.rule_2_1 = false;
```

To find the section number for a rule, see Coding Standards. To find the property corresponding to the section name, use auto-completion for MATLAB object properties.

See also `polyspace.CodingRulesOptions`.

Using checkers selection file required for polyspace.CodingRulesOptions object

Errors

If you assign a `polyspace.CodingRulesOptions` object to an analysis configuration, for instance:

```
misraRules = polyspace.CodingRulesOptions('misraC2012');
proj = polyspace.Project;
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = misraRules;
```

You must also enable the use of a checkers selection file, for instance:

```
proj.Configuration.CodingRulesCodeMetrics.EnableCheckersSelectionByFile = true;
```

You have to enable checkers selection by file because the Polyspace run uses an XML file underneath to enable the coding rule checkers. The XML file is saved in a `.settings` subfolder of the results folder.

See also `polyspace.CodingRulesOptions`.

Verification Results

Function Stub Improvements: See fewer orange checks from default conservative assumptions on pointer arguments

Summary: In R2019b, a Code Prover analysis assumes that stubbed function arguments passed by reference or pointer cannot remain uninitialized on return from the function. A function is stubbed if its definition is not available for the analysis. See Stubbed Functions.

Benefits: You see fewer orange checks from the previous default assumption that stubbed function arguments that are not initialized might remain uninitialized on return from the function.

For instance, in the following example, Code Prover assumes that `i` is initialized on return from the function `stub`. With this assumption, the non-initialized variable check on `i` in the line `j=i` appears green.

```
int main(void)
{
    int i, j;
    stub(&i);
    j = i;
    return 0;
}
```

Compatibility Considerations

You see fewer orange non-initialized variable checks compared to previous releases. To revert to the previous conservative assumptions for specific function stubs, specify external constraints. See External Constraints for Polyspace Analysis.

MISRA C:2012 Directive 4.12: Dynamic memory allocation shall not be used

Summary: In R2019b, you can look for violations of MISRA C:2012 Directive 4.12. The directive states that dynamic memory allocation and deallocation packages provided by the Standard Library or third-party packages shall not be used. The use of these packages can lead to undefined behavior.

See `MISRA C:2012 Dir 4.12`.

Reviewing Results

Code Annotations: Justify Code Prover results by using annotations spread over multiple lines

Summary: In R2019b, you can enter multi-line code annotations to justify Polyspace results. Subsequent runs can use these annotations and automatically populate the severity, status, and comments fields for previously reviewed results.

See Annotate Code and Hide Known or Acceptable Results.

Benefits: Previously, the entire Polyspace annotation could span one line only. With the single-line constraint removed, you can add more detailed explanations in code annotations and view the entire annotation in your code editor, or let your code editor wrap the annotations. For instance, you can enter a code annotation like this annotation:

```
x++; /* polyspace RTE:OVFL "This operation
                           cannot overflow
                           because of
                           external constraints" */
```

R2019a

Version: 10.0

New Features

Bug Fixes

Compatibility Considerations

Verification Setup

Polyspace-only Licenses: Install Polyspace without MATLAB installation

Summary: In R2019a, you can install the Polyspace products without a MATLAB installation.

Benefits: If you use Windows® or Linux® binaries to automate your Polyspace analysis and do not otherwise use MATLAB in your workflow, you do not require a MATLAB installation. However, if you want to use the conveniences of MATLAB scripting such as easy reading and visualization of Polyspace results and syntax completion for functions, you can install MATLAB separately and link with your Polyspace installation.

Compatibility Considerations

If you use MATLAB scripts to run Polyspace, you can continue to run your scripts as before. However, your initial set up is different from previous releases:

- Run the installer twice with separate licenses to install MATLAB and Polyspace in separate folders.
- Perform a setup step to link your Polyspace installation with your MATLAB installation.

See Integrate Polyspace with MATLAB and Simulink.

New Polyspace Products Supporting Continuous Integration: Perform automated code analysis after code submission with Polyspace Code Prover Server and Polyspace Code Prover Access

Summary: R2019a brings new Polyspace products for automated runs on server class machines:

- Polyspace Bug Finder Server and Polyspace Bug Finder Access
- Polyspace Code Prover Server and Polyspace Code Prover Access

The current products, Polyspace Bug Finder and Polyspace Code Prover, can be used by individual developers on their desktops.

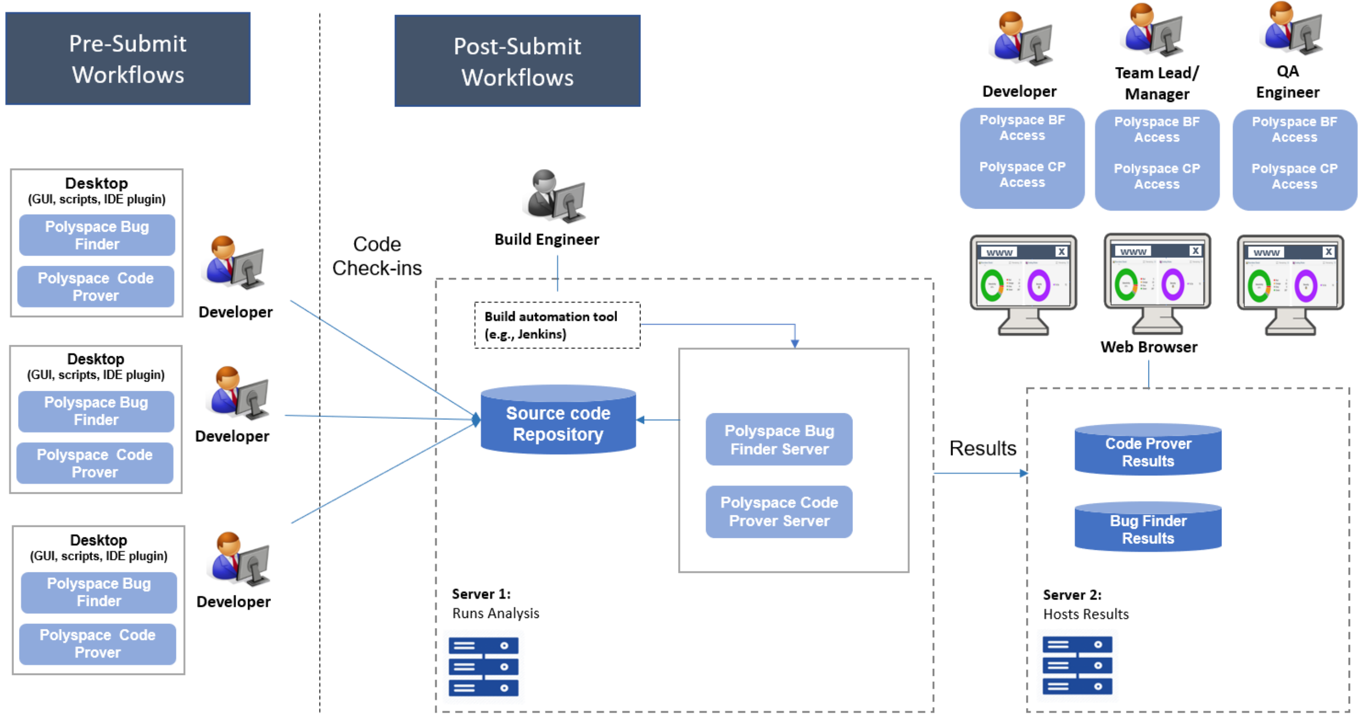
Benefits: The new Polyspace products are designed for automated runs in a continuous integration workflow. With the new products, the Polyspace suite of products now supports all phases of a software development process:

- *Prior to code submission*:

  Developers can run the Polyspace desktop products to check their code during development or right before submission to meet predefined quality goals.

  The desktop products can be plugged in IDEs such as Eclipse™ or run with scripts, for instance during compilation. The analysis results can be reviewed in IDEs such as Eclipse or in the graphical user interface of the desktop products.
- *After code submission*:

The Polyspace server products can run automatically on newly committed code as a build step in a continuous integration process (using tools such as Jenkins). The analysis runs on a server using the product Polyspace Bug Finder Server™ or Polyspace Code Prover Server and the results are uploaded to the Polyspace Access web interface for collaborative review.



Note: Depending on the specifications, the same computer can serve as both Server 1 and Server 2.

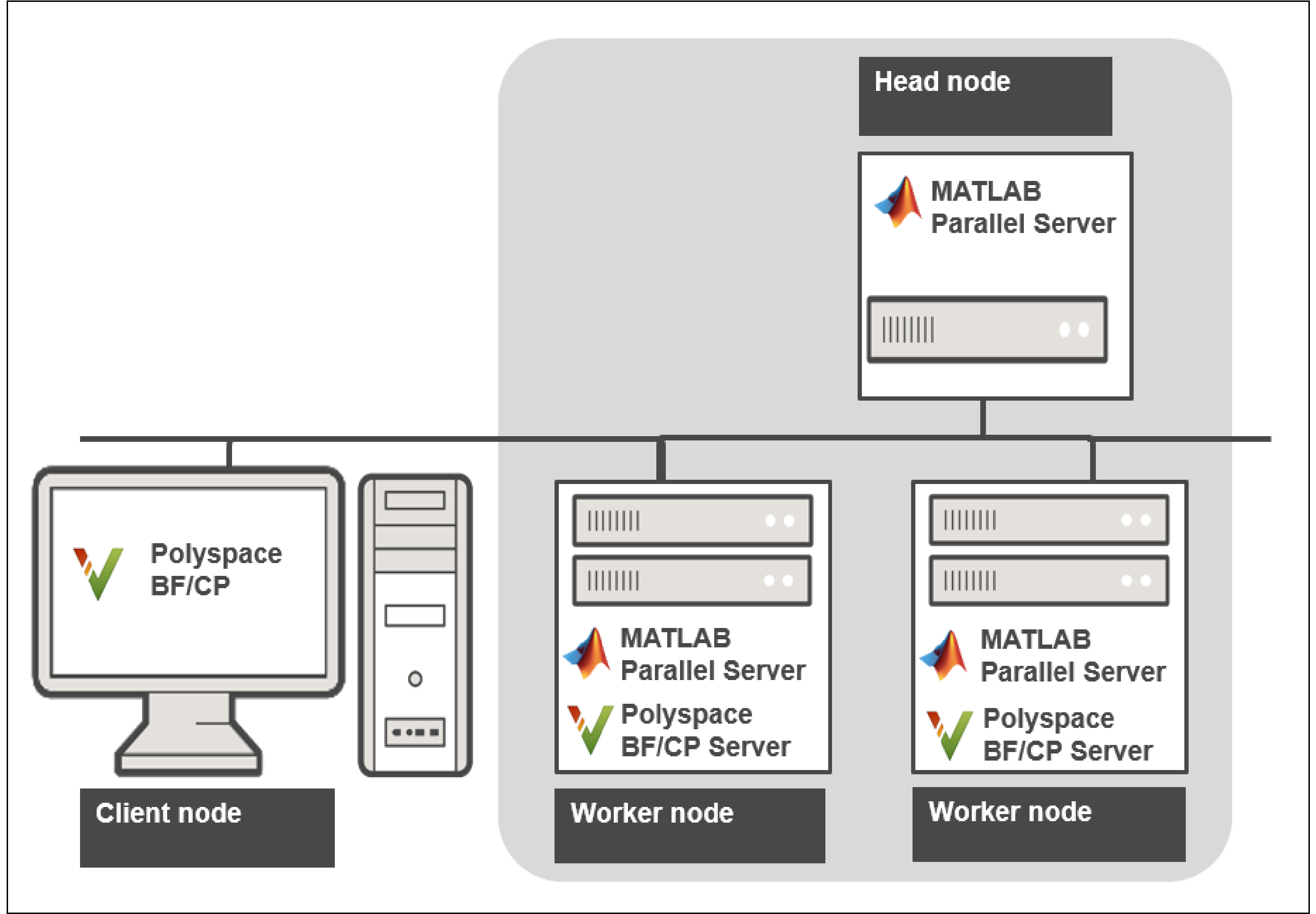
See Polyspace Products for Code Analysis and Verification.

For more information on the new products, see:

- Polyspace Bug Finder Server
- Polyspace Code Prover Server
- Polyspace Bug Finder Access
- Polyspace Code Prover Access

Offloading Polyspace Analysis to Servers: Use Polyspace desktop products on client side and server products on server side

Summary: In R2019a, you can offload a Polyspace analysis from your desktop to remote servers by installing the Polyspace desktop products on the client side and the Polyspace server products on the server side. After analysis, the results are downloaded to the client side for review. You must also install MATLAB Parallel Server™ on the server side to manage submissions from multiple client desktops.



See Install Products for Submitting Polyspace Analysis from Desktops to Remote Server.

You can also follow a workflow where Polyspace runs on a dedicated server after code submission and uploads results to a web interface for review. In this case, you require one or more Polyspace Code Prover Server license for running the analysis on dedicated servers and Polyspace Code Prover Access licenses to review the results.

Benefits: The Polyspace desktop products have a graphical user interface. You can configure options in the user interface with assistance from features such as auto-population of option arguments and contextual help. To save processing time on your desktop, you can then offload the analysis to remote servers.

Compatibility Considerations

If you offloaded analysis results from your desktop to remote servers prior to R2019a, your initial setup is different from previous releases.

- On the client side, you do not require Parallel Computing Toolbox™. You only require the Polyspace desktop product, Polyspace Code Prover.

- On the server side, instead of the desktop product, Polyspace Code Prover, you must install the server product, Polyspace Code Prover Server. You still require MATLAB Parallel Server (previously called MATLAB Distributed Computing Server).

  You install the Polyspace server products and MATLAB Parallel Server in separate folders and link between them.

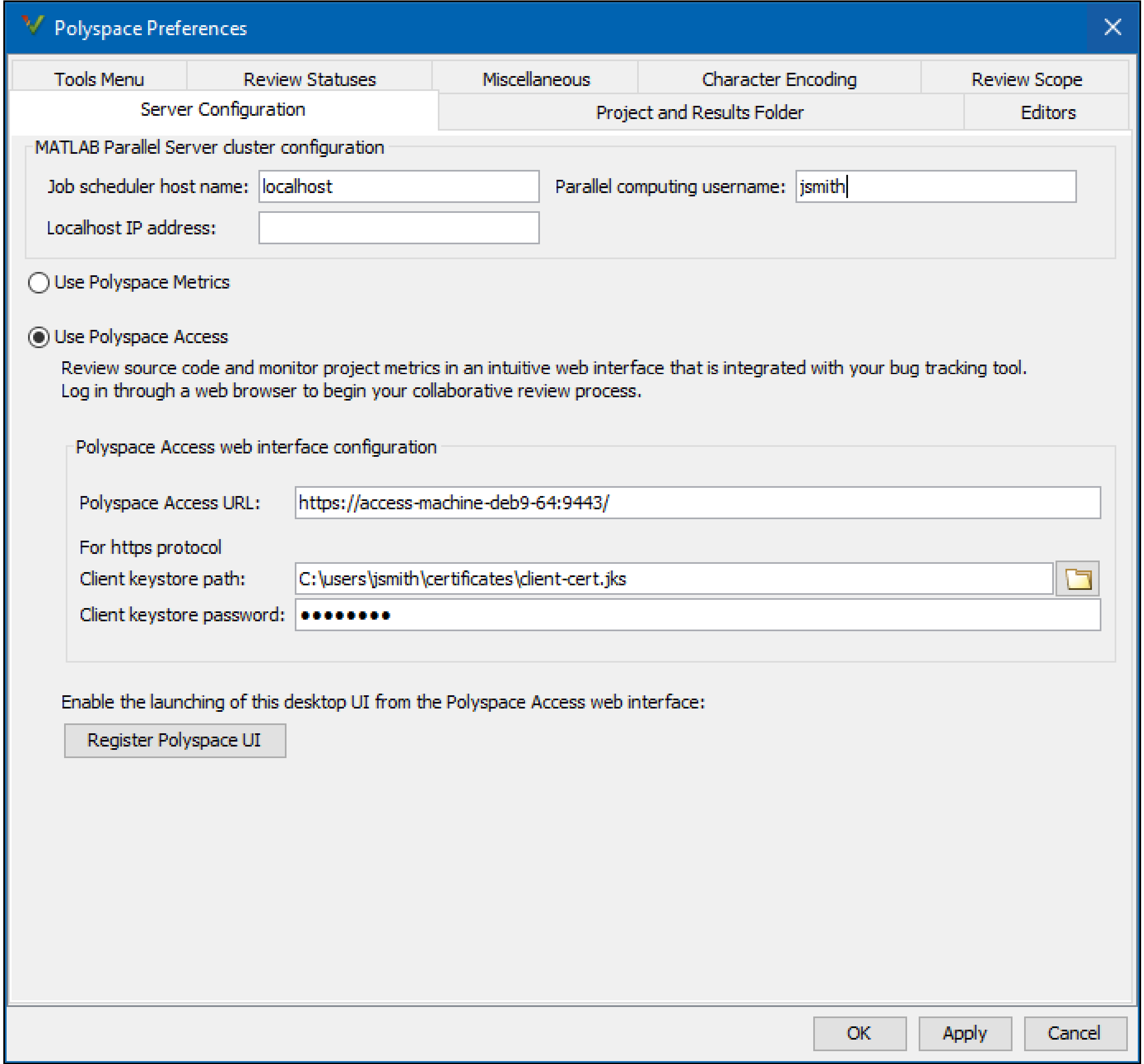
  See Install Products for Submitting Polyspace Analysis from Desktops to Remote Server.
- You do not have the quick start option to start the server with one worker (the **Metrics and Remote Analysis Server Settings** interface). Instead you must use the **Admin Center** interface in MATLAB Parallel Server. In this workflow, you first start the services on all remote computers, then assign responsibilities to these computers as either the head node that schedules jobs or worker nodes that run the analysis.

  See Install Products for Submitting Polyspace Analysis from Desktops to Remote Server.

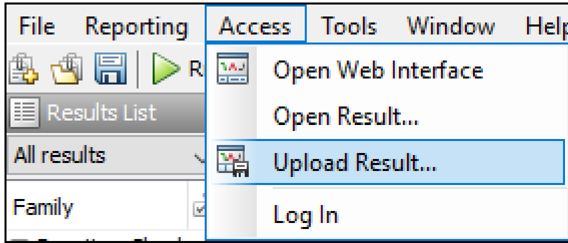
Collaborative Review Support: Upload results from Polyspace user interface to Polyspace Access web interface and share results using web links

Summary: In R2019a, you can upload Polyspace Code Prover results from the user interface of the desktop products to the Polyspace Access web interface. Developers with a Polyspace Code Prover Access license can review these results in the web interface and share the results using web links.

To upload results from the Polyspace user interface, select **Tools** > **Preferences**. On the **Server Configuration** tab, enter the URL of the Polyspace Access web interface and the client keystore path and password.



After setting up communication between the Polyspace user interface and the Polyspace Access web interface, the **Access** menu appears in the Polyspace user interface. You can use this menu to open the web interface, open results from the web interface in the user interface of the desktop product or upload results from the desktop product to the web interface.



For details about setting up and reviewing results in the Polyspace Access web interface, see Polyspace Code Prover Access documentation.

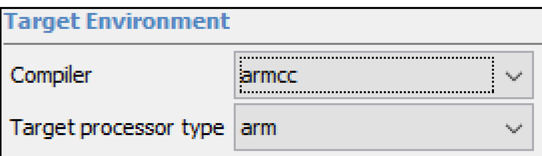
Benefits::

- *Facilitate collaborative review:* The web interface streamlines the review efforts of your team. For instance:
  - During a team meeting, findings can be assessed and assigned to developers.
  - Developers can log into the web interface to review findings assigned to them, and determine whether to justify the findings or fix them.
  - A project manager can track the progress of the review by filtering the list of results for findings that are still open.
- *Authenticate client access:* The web interface is behind a login. Only users with a Polyspace Code Prover Access license and the appropriate credentials can view the results from their web browser.

Compiler Support: Set up Polyspace analysis easily for code compiled with ARM v5 and v6 compilers

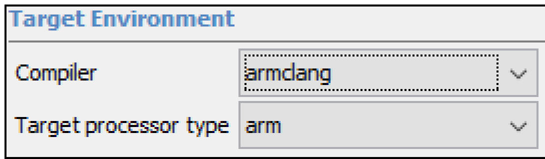
Summary: If you build your source code using these compilers, in R2019a, you can specify the compiler name for your Polyspace analysis:

- ARM® v5

  You can specify target `arm`.

  See `ARM v5 Compiler (-compiler armcc)`.
- ARM v6

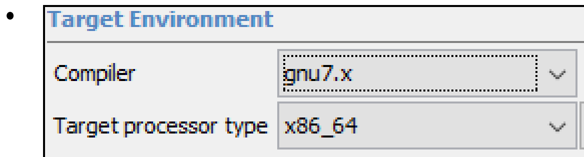
  You can specify targets `arm` and `arm64`.

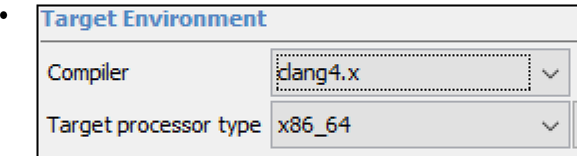
See `ARM v6 Compiler (-compiler armclang)`.

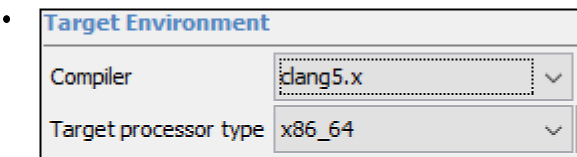
Benefits:You can now set up a Polyspace project without knowing the internal workings of these compilers. If your code compiles with your compiler, it will compile with Polyspace in most cases without requiring additional setup. Previously, you had to explicitly define macros that were implicitly defined by the compiler and remove unknown language extensions from your preprocessed code.

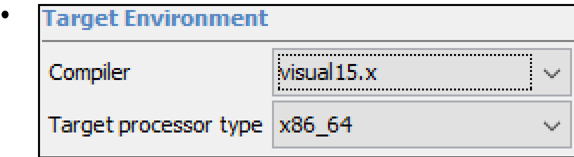
Updated GCC, Clang, and Visual C++ Compiler Support: Set up Polyspace analysis easily for code compiled with GCC versions 7.x, Clang versions 4.x or 5.x, or Microsoft Visual C++ 2017 compilers

Summary: In R2019a, if you build your source code using these version of GCC, Clang, or Microsoft Visual C++ compilers, you can specify the following compiler option values to setup your Polyspace analysis:

- 

  `gnu7.x` for GCC release 7.1, 7.2, and 7.3.
- 

  `clang4.x` for LLVM release 4.0.0, and 4.0.1.
- 

  `clang5.x` for LLVM release 5.0.0, and 5.0.1.
- 

  `visual15.x` for Microsoft Visual C++ 2017 versions 15.0 to15.7.

The analysis can interpret macros that are implicitly defined by the compiler and compiler-specific language extensions such as keywords and pragmas.

For more information, see `Compiler (-compiler)`.

Simulink Toolstrip: Analyze generated code using contextual buttons in Simulink Editor

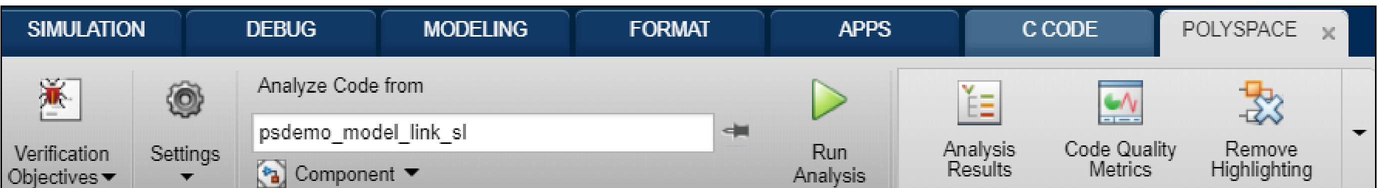
Summary: In R2019a, you have the option to turn on the Simulink Toolstrip.

- To enable the toolstrip, select **File** > **Simulink Preferences**. On the **Editor** node, select **Replace menus and toolbars with the Simulink Toolstrip (Tech Preview)**.
- To disable the toolstrip, on the **Modeling** tab, select **Environment** > **Simulink Preferences**. Clear the previous selection.

See Simulink Toolstrip Tech Preview replaces menus and toolbars in the Simulink Desktop for more details.

Benefits: The Simulink Toolstrip includes contextual tabs, which appear only when you need them. The Polyspace contextual tab includes options for completing actions that apply only to Polyspace.

- To generate code, open the **C Code** tab. To access this tab, on the **Apps** tab, select **Embedded Coder**.
- To analyze the generated code, open the **Polyspace** tab. To access this tab, on the **Apps** tab, select **Polyspace Code Verifier**.



On the **Polyspace** tab:

1. After code generation, from the **Verification Objectives** menu, choose **Find Bugs** (Bug Finder) or **Prove Code** (Code Prover).
2. Optionally, configure code analysis options. To configure the basic options related to the model, select **Settings** > **Polyspace Settings**. To configure advanced options related to the generated code, select **Settings** > **Project**.
3. To start an analysis, select **Run Analysis**. The analysis runs on the model element selected, provided code has been generated earlier from the same element. The selected element appears in the **Analyze Code from** field. To select the entire model, click anywhere on the canvas outside a model element.

Compatibility Considerations

The Simulink Toolstrip included with R2019a is a tech preview. You may encounter performance issues when you enable the toolstrip. Documentation does not reflect the addition of the Simulink Toolstrip and toolstrip customization is not available.

Changes in analysis options and binaries

polyspace-code-prover-nodesktop renamed to polyspace-code-prover
Warns

The command-line options available with `polyspace-code-prover` are the same as those with `polyspace-code-prover-nodesktop` (with the exception of changes mentioned below). Simply replace `polyspace-code-prover-nodesktop` with `polyspace-code-prover` in your batch files or shell scripts.

-report-template arguments changed for coding standard templates

Warns

If you used the template `CodingRules.rpt` for the option `-report-template`, use the new `CodingStandards.rpt` template instead.

Check MISRA C:2012 (-misra3) option values CERT-rules, CERT-all, and ISO-17961 are removed

Warns

`Check MISRA C:2012 (-misra3)` option values `CERT-rules`, `CERT-all`, and `ISO-17961` are removed. Previously, you used **Check MISRA C:2012 (-misra3)** with these options values to check your code for violations of the CERT C and ISO/IEC TS 17961 coding standards. Use a Polyspace Bug Finder analysis with the new **Coding Standards & Code Metrics** analysis options `Check SEI CERT-C (-cert-c)` and `Check ISO/IEC TS 17961 (-iso-17961)` instead.

The new analysis options simplify checking for violations of coding standards CERT C and ISO/IEC TS 17961. For more information, see Changes in Coding Standard Checking in R2019a (Polyspace Bug Finder)

You get a warning when you use the removed option values.

Check MISRA C++ rules (-misra-cpp) option values CERT-rules and CERT-all are removed

Warns

`Check MISRA C++:2008 (-misra-cpp)` option values `CERT-rules` and `CERT-all` are removed. Previously, you used **Check MISRA C++ rules (-misra-cpp)** with these options values to check your code for violations of the CERT C++ coding standards. Use a Polyspace Bug Finder analysis with the new **Coding Standards & Code Metrics** analysis option `Check SEI CERT-C++ (-cert-cpp)` instead.

The new analysis option simplifies checking for violations of the CERT C++ coding standard. For more information, see Changes in Coding Standard Checking in R2019a (Polyspace Bug Finder)

You get a warning when you use the removed option values.

Syntax changed for options that take C++ templates as arguments

Errors

If you use options that take instantiations of C++ templates as arguments, you have to provide the option arguments differently. For instance, for this template function `getMax`:

```
template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}
template int GetMax<int>(int, int); // explicit instantiation
```

If you previously specified the instantiation as argument for the option `Functions to call (-main-generator-calls)` in this way:

```
-main-generator-calls custom="T1 getMax<T1>(T1,T1) [with T1=int]"
```

You now specify the option argument as:

```
-main-generator-calls custom="T1 getMax<int>(T1,T1)"
```

This syntax change applies to all options that take instantiations of templates as arguments.

Changes in MATLAB functions, options object and properties

Initial setup required for running Polyspace from MATLAB

Behavior change

If you use MATLAB scripts to run Polyspace, you can continue to run your scripts as before. However, your initial setup is different compared to previous releases:

- Run the MathWorks® installer twice with separate licenses to install MATLAB and Polyspace in separate folders.
- Perform a setup step to link your Polyspace installation with your MATLAB installation.

See Integrate Polyspace with MATLAB and Simulink (Polyspace Bug Finder).

polyspaceCodeProverNodesktop removed

Warns

Use `polyspaceCodeProver(`*`projectFile`*`, '-nodesktop')` instead of `polyspaceCodeProverNodesktop(`*`projectFile`*`)`.

CodeProverReportTemplate property value changed for coding standard compliance reports

Warns

To update your MATLAB code, use the new template `CodingStandards` for the property `CodeProverReportTemplate`:

```
proj = polyspace.Project;
proj.Configuration.MergedReporting.BugFinderReportTemplate = 'CodingStandards';
```

instead of the old template `CodingRules`.

Property CustomRulesSubset is removed

Errors

`CodingRulesCodeMetrics` property `CustomRulesSubset` is removed. Previously, you used this property to specify the path to the file where you defined custom naming conventions to check against. Use the new property `CheckersSelectionByFile` instead.

With the new property, you specify a file in `.xml` format where you define custom rules to match identifiers in your code, and custom selections of checkers for all the coding standards that Polyspace supports. See `Set checkers by file (-checkers-selection-file)`.

To update your MATLAB code, see this table.

```
opts = polyspace.Project;
```

Property	Use Instead
opts.CodingRulesCodeMetrics... .EnableCustomRules=1; opts.CodingRulesCodeMetrics... .CustomRulesSubset='custom_rules.txt';	opts.CodingRulesCodeMetrics... .EnableCustomRules=1; opts.CodingRulesCodeMetrics... .EnableCheckersSelectionByFile=1; opts.CodingRulesCodeMetrics... .CheckersSelectionByFile='custom_rules.xml';

For more information, see `polyspace.Project.Configuration Properties`.

Option values CERT-rules, CERT-all, and ISO-17961 are removed for CodingRulesCodeMetrics property MisraCSubset

Errors

`MisraCSubset` option values `CERT-rules`, `CERT-all`, and `ISO-17961` are removed. Previously, you used `MisraCSubset` with these options values to check your code for violations of the CERT C and ISO/IEC TS 17961 coding standards. Use a Polyspace Bug Finder analysis with the new `CodingRulesCodeMetrics` properties `CertC` and `EnableIso17961` instead.

The new `CodingRulesCodeMetrics` properties simplify checking for violations of coding standards CERT C and ISO/IEC TS 17961.

For more information, see `polyspace.Project.Configuration Properties`.

Option values CERT-rules and CERT-all are removed for CodingRulesCodeMetrics property MisraCppSubset

Errors

`MisraCppSubset` option values `CERT-rules` and `CERT-all` are removed. Previously, you used `MisraCSubset` with these options values to check your code for violations of the CERT C++ coding standard. Use a Polyspace Bug Finder analysis with the new `CodingRulesCodeMetrics` property `CertCpp` instead.

The new `CodingRulesCodeMetrics` property simplifies checking for violations of the CERT C++ coding standard.

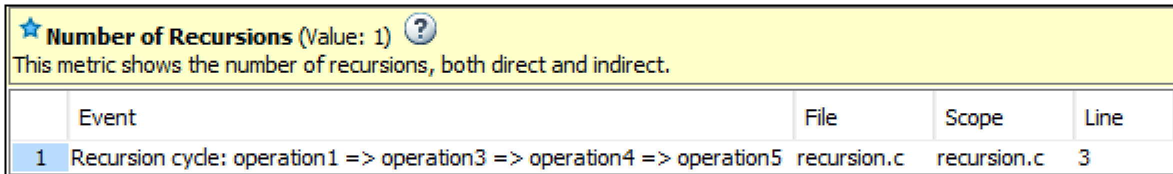
For more information, see `polyspace.Project.Configuration Properties`.

Verification Results

Recursion Detection: See list of recursion cycles in C/C++ project

Summary: In R2019a, the code metrics `Number of Recursions` and `Number of Direct Recursions` are displayed along with a list of recursion cycles in your project.

- For the metric **Number of Direct Recursions**, the list shows all direct recursions (self recursive functions or functions calling themselves).
- For the metric **Number of Recursions**, the list shows all direct recursions plus a partial list of indirect recursion cycles. For details, see `Number of Recursions`.



Benefits:

- *Easier navigation to recursion cycles*: Each row in the list shows one recursion cycle. You can click a row to navigate to one of the functions involved in the recursion cycle.
- *Checking metric computation*: You can check the value of the code metrics `Number of Recursions` and `Number of Direct Recursions`.

Compatibility Considerations

A slightly different algorithm is used to compute the number of recursions. You can see a different value of this metric compared to previous releases. For computation details, see `Number of Recursions`.

Updated code metrics specifications

Summary: In R2019a, these code metric specifications have been updated.

Code Metric	Update
`Number of Function Parameters`	In cases where a C++ function returns an object, you can see a decrease in number of function parameters. Previously, the metric incorrectly included additional parameters corresponding to Polyspace internal variables.

Code Metric	Update
Number of Recursions	You can see a change in the number of recursions in your project. The algorithm to compute recursions is slightly different from previous releases. The metric reports the number of direct recursions plus the number of strongly connected components formed by the indirect recursion cycles. The metric is also supported with events showing the recursion cycles. For details, see the release note about Recursion Detection.
Number of Paths	You can see a high value of the metric in some cases where the metric value was previously reported as zero. The number of paths increases exponentially with the branching in the code. If the number of paths exceeds an internal limit, the metric calculation stops and reports the value 9223372036854775807 (indicating the hexadecimal value 0x7fffffffffffffff). Previously, the metric value was reported as zero in those cases.

<table>
<tr><th>Code Metric</th><th>Update</th></tr>
<tr><td>Code complexity metrics for C++ templates</td><td>

If you use C++ templates, you can see a difference in the value of certain metrics.

Each instantiation of a C++ template is considered as a separate function. Code complexity metrics are reported separately for each instantiation.

For instance, consider the function template `GetMax` instantiated twice in `main`:

```
// function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

In R2019a, the two instantiations of `GetMax` are considered as separate functions. All code metrics are reported separately for the two instantiations. Further, the number of called functions in `main` is 2.

Previously, the two instantiations were considered as one.

</td></tr>
<tr><td>Stack usage metrics, for instance, <code>Minimum Stack Usage</code> and <code>Maximum Stack Usage</code>.</td><td>Stack usage takes into account variable parameters of variable-argument or variadic functions. Previously, only the fixed parameters were taken into account.</td></tr>
</table>

<table>
<tr><th>Code Metric</th><th>Update</th></tr>
<tr><td>Sizes of local variables and stack usage metrics</td><td>You see a decrease in the metrics for a function if a local variable is an instance of a C++ class that inherits virtually from another class. Previously, a Polyspace internal variable was used to keep track of the virtual inheritance and the internal variable was taken into account in the size metrics. The calculation no longer considers the internal variable.

For instance, consider this example:

<code>class A { virtual void f(); };</code>
<code>class B : virtual A { };</code>

Previously, the size of an object of type <code>A</code> was shown as 8 and <code>B</code> as 16. Now both sizes are calculated as 8.</td></tr>
</table>

Compatibility Considerations

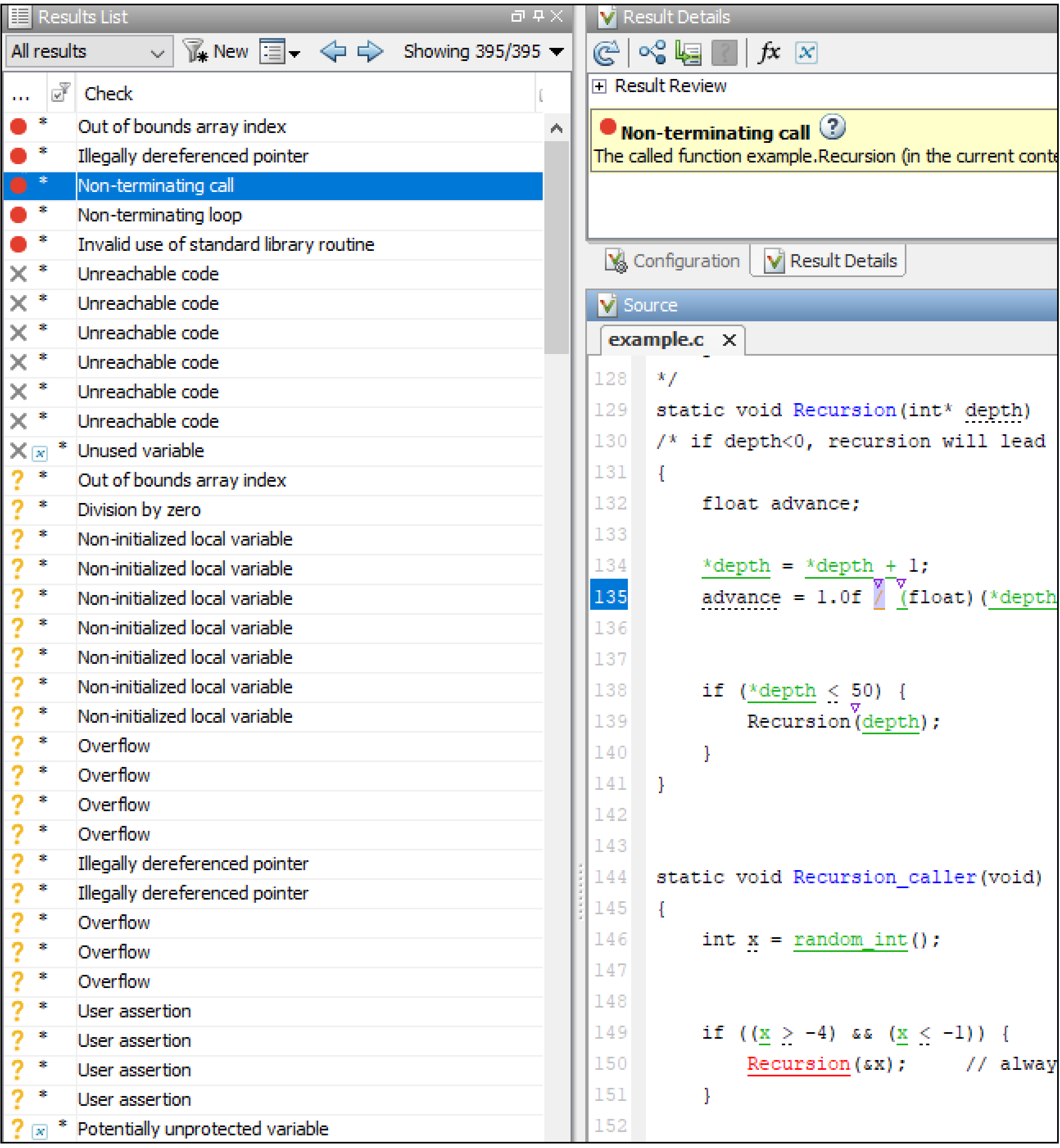
If you compute these code metrics, you can see a difference in results compared to previous releases.

Reviewing Results

Source Code Navigation: Keep result pinned while navigating through source code

Summary: In R2019a, clicking a result in the source code does not change the result selection on the **Results List** and the details on the **Result Details** pane.

For instance, in this example, the result **Non-terminating call** is selected on the **Results List** pane. The corresponding source code (line 150) appears on the **Source** pane and further details about the result on the **Result Details** pane. If you then navigate through the source code and select a token highlighting another result (for instance, the orange / operator in line 135), the selection in the results list and the details still show the **Non-terminating call** result.



Benefits: To find the root cause of a result, you have to navigate through the source code. You can keep the result pinned on the **Results List** and **Result Details** pane during this navigation.

Compatibility Considerations

Previously, if you clicked a token in the source code showing a result, the selection on the **Results List** pane and the information on the **Result Details** pane changed to the clicked result. To emulate this behavior, `Ctrl`-click the token in the source code or right-click and select **Select Results At This Location**.

Report Generation: Generate Polyspace reports faster than previous releases

Summary: In R2019a, Polyspace report generation uses a more optimized algorithm.

Benefits: You can now generate PDF, HTML or Microsoft Word reports from Polyspace results much faster than before. For large reports, report generation can be more than ten times faster than before.

Report Generation: Generate single file for HTML reports

Summary: In R2019a, if you generate an HTML report, a single HTML file is created.

Benefits: The single HTML file allows easier archiving. Previously, several companion files were generated in HTML reporting. You had to archive all files together to be able to view the HTML report.

Compatibility Considerations

The structure of the new HTML report is different from prior releases. If you used scripts to parse the HTML reports, you might have to adapt the scripts to the new HTML structure.

R2018b

Version: 9.10

New Features

Bug Fixes

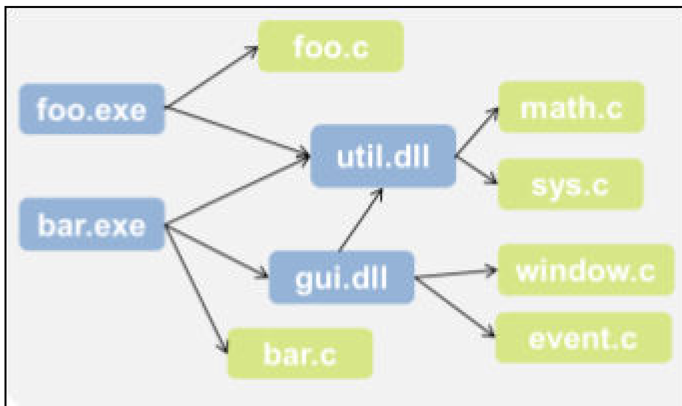
Compatibility Considerations

Verification Setup

Configuration from Build System: Automatically generate Polyspace configuration modules from build system

Summary: In R2018b, you can create a separate Polyspace analysis module for each binary in your build system.

Suppose a build system has the following dependencies and creates four binaries: the executables `foo.exe` and `bar.exe`, and the dynamic libraries `util.dll` and `gui.dll`.



Previously, you created a single Polyspace options file from this build system. You can now create a separate Polyspace options file for each binary created in your build system.

See also:

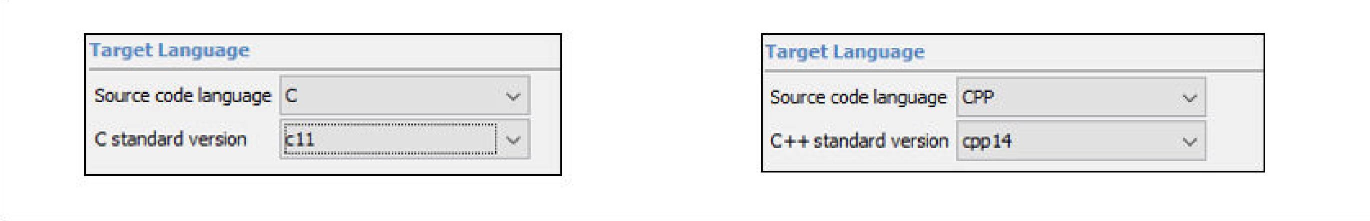
- Modularize Polyspace Analysis by Using Build Command
- `polyspace-configure`

Benefits:

- *More precise analysis*: You can perform a separate Polyspace analysis for each binary in your build system. The analysis does not mix files from distinct binaries.
- *Automated modularization*: You can reuse the modularization in your build system to create the Polyspace analysis modules.
- *Focused analysis*: You can analyze only specific modules instead of your entire code base.
- *Minimal knowledge of build system required*: You do not need to know the details of your build system. With a `-module` flag, a separate options file is created for each binary in your build system. You can analyze only the code implementation of the binaries that you are interested in.

C11 and C++14 Support: Run Polyspace analysis on code with C11 or C++14 features

Summary: In R2018b, Polyspace can interpret the majority of C11 or C++14-specific features.



See also C/C++ Language Standard Used in Polyspace Analysis.

Benefits: You can now setup a Polyspace analysis for code containing C11 or C++14-specific features. Previously, some features were not recognized and caused compilation errors.

Autodetection of Concurrency Primitives: Multitasking model detected from C11 multithreading functions

Summary: In R2018b, if you use C11 functions for multitasking, the Polyspace analysis can interpret them semantically.

Polyspace interprets the following functions:

- `thrd_create`: Thread is created.
- `mtx_lock`: Critical section begins.
- `mtx_unlock`: Critical section ends.

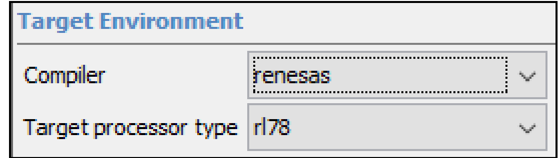
See also Auto-Detection of Thread Creation and Critical Section in Polyspace.

Benefits: You do not have to adapt your code or specify your multitasking model manually through analysis options. The analysis determines your multitasking model from the functions in your code and finds data races or other concurrency defects.

Compiler Support: Set up Polyspace analysis easily for code compiled with Renesas compilers

Summary: If you build your source code with the Renesas compiler, in R2018b, you can specify the compiler name for your Polyspace analysis. The analysis can interpret macros that are implicitly defined by the compiler and compiler-specific language extensions such as keywords and pragmas.

You can specify these target processors directly: `rl78`, `rh850`, or `rx`. See Renesas Compiler (`-compiler renesas`).



Benefits: You can now set up a Polyspace project without knowing the internal workings of the Renesas compilers. If your code compiles with your compiler, it will compile with Polyspace in most cases without requiring additional setup. Previously, you had to explicitly define macros that were implicitly defined by the compiler and remove unknown language extensions from your preprocessed code.

AUTOSAR Support: Provide multiple root folders for sources

Summary: In R2018b, you can specify multiple root folders for source code when running Polyspace Code Prover on AUTOSAR software components.

```
polyspace-autosar -create-project PROJECT_FOLDER
                  -arxml-dir AUTOSAR_FOLDER
                  -sources-dir CODE_FOLDER1 -sources-dir CODE_FOLDER2 ...
                  [OPTIONS]
```

See `polyspace-autosar`.

Benefits: If you have source folders outside your main source hierarchy, for instance source code libraries, you can specify them with the `polyspace-autosar` command.

AUTOSAR Support: Run Polyspace on AUTOSAR software components by using MATLAB scripts

Summary: In R2018b, you can run Polyspace Code Prover on code implementation of AUTOSAR software components using MATLAB scripts.

```
exampleDir = fullfile(matlabroot,'help',...
    'toolbox','codeprover','examples','polyspace_autosar');
arxmlDir = fullfile(exampleDir, 'arxml');
sourceDir = fullfile(exampleDir, 'code');

tempDir = tempdir;

projectDir = fullfile(tempDir, 'polyspace');
status = polyspaceAutosar('-create-project', projectDir, ...
    '-arxml-dir', arxmlDir, ...
    '-sources-dir', sourceDir);
```

See also `polyspaceAutosar`.

Benefits: You can use MATLAB scripts to automate the modularized Polyspace analysis of AUTOSAR code.

AUTOSAR Support: Provide compiler options by tracing your build command

Summary: In R2018b, you can trace your build command to gather compiler options, macro definitions and paths to include folders, and provide this information for analysis of code implementation of AUTOSAR software components.

1. Trace your build command (for instance, `make`) with `polyspace-configure` and generate an options file for subsequent Code Prover analysis. Suppress inclusion of sources in the options file with the `-no-sources` option.

```
polyspace-configure -output-options-file options.txt -no-sources make
```

2 Run Code Prover on AUTOSAR code with `polyspace-autosar`. Provide your ARXML folder, source folders and other options. In addition, provide the earlier generated options file with the `-extra-options-file` option.

```
polyspace-autosar -create-project projFolder \
-arxml-dir arxmlFolder \
-sources-dir codeFolder \
-extra-options-file options.txt
```

See also:

- Run Polyspace on AUTOSAR Code Using Build Command
- `polyspace-configure` and `polyspace-autosar`

Benefits: You can automate the Code Prover analysis of AUTOSAR code by reusing the infrastructure you have already set up.

- *Reusing build command*: You reuse the compiler options specified in your build command for the Code Prover analysis. Code Prover can emulate your compiler and recognize compiler-specific macros and language extensions.
- *Reusing ARXML specifications*: You reuse the ARXML specifications to perform a modular Code Prover analysis. Code Prover can modularize your code based on the software components in the ARXML specifications. See also Benefits of Polyspace for AUTOSAR.

Function Pointer Calls: Verify functions called through function pointers despite type mismatch

Summary: In R2018b, Code Prover can verify functions called through function pointers even if there is a type mismatch between the function argument(s) and/or return type and the function pointer argument(s) and/or return type.

For instance, in this example, function pointer `obj_fptr` has an argument that is a pointer to a four-element array. `obj_fptr` points to a function `foo` whose corresponding argument is a pointer to a three-element array.

```
typedef int array_four_elements[4];
typedef void (*fptr)(array_four_elements*);

typedef int array_three_elements[3];
void foo(array_three_elements*);

void main() {
 array_four_elements arr[4] = {0,0,0,0};
 array_four_elements *ptr;
 fptr obj_fptr;

 ptr = &arr;
 obj_fptr = &foo;

 obj_fptr(&ptr);
}

void foo(array_three_elements* x) {
```

```
    ...
}
```

By default, the analysis does not check functions called through mismatched function pointers. Use the option `Permissive function pointer calls (-permissive-function-pointer)`.

Only type mismatches between pointer types are allowed. Type mismatches between nonpointer types cause compilation errors.

Benefits: If you cannot fix type mismatches between a function pointer and the pointed function, you can still analyze the function body. Previously, Code Prover displayed an orange check on the function call and did not check the function body for run-time errors.

Check Behavior on Overflows: Fine-tune the behavior of checks based on signedness of integer

Behavior change

Summary: In R2018b, you can set different behaviors for checks of operations on signed or unsigned integers that may overflow. For instance, signed integers should truncate and unsigned integers should wrap-around.

Use the new options `Overflow mode for signed integer (-signed-integer-overflows)` and `Overflow mode for unsigned integer (-unsigned-integer-overflows)` to fine-tune the check behavior.

The options `-scalar-overflows-checks` and `-scalar-overflows-behavior` are no longer supported. You get a warning if you use these options. For Polyspace projects starting R2014a, instances of the `-scalar-overflows-*` options are replaced with the new options according to the mapping of this table.

Option	Use instead

`-scalar-overflows-checks none -scalar-overflows-behavior truncate-on-error`	`-signed-integer-overflows allow`
	
	`-unsigned-integer-overflows allow`

`-scalar-overflows-checks signed -scalar-overflows-behavior truncate-on-error`	`-signed-integer-overflows forbid`
	
	`-unsigned-integer-overflows allow`

`-scalar-overflows-checks signed-and-unsigned -scalar-overflows-behavior truncate-on-error`	`-signed-integer-overflows forbid`
	
	`-unsigned-integer-overflows forbid`

`-scalar-overflows-checks none -scalar-overflows-behavior wrap-around`	`-signed-integer-overflows allow`
	
	`-unsigned-integer-overflows allow`

`-scalar-overflows-checks signed -scalar-overflows-behavior wrap-around`	`-signed-integer-overflows warn-with-wrap-around`
	
	`-unsigned-integer-overflows allow`

Option	Use instead
-scalar-overflows-checks signed-and-unsigned -scalar-overflows-behavior wrap-around	-signed-integer-overflows warn-with-wrap-around -unsigned-integer-overflows warn-with-wrap-around

For Polyspace projects prior to R2014a, all combinations of the old `-scalar-overflows-*` options map to `-signed-integer-overflows forbid` `-unsigned-integer-overflows allow`.

Changes in analysis options and binaries

Polyspace Code Prover has new Target & Compiler options

Behavior change

Polyspace Code Prover has new **Target & Compiler** configuration options `C standard version (-c-version)` and `C++ standard version (-cpp-version)`.

Use these options to specify the C and C++ language standards you follow in your source code.

Polyspace Code Prover has new Check Behavior options

Behavior change

Polyspace Code Prover has new **Check Behavior** configuration options `Overflow mode for signed integer (-signed-integer-overflows)` and `Overflow mode for unsigned integer (-unsigned-integer-overflows)`.

Use these options to specify the behavior of checks for signed and unsigned integer overflows.

-compiler option has new value renesas

Behavior change

`Compiler (-compiler)` option has new value `renesas`. When you specify this option value, The analysis can interpret macros that are implicitly defined by the compiler and compiler-specific language extensions such as keywords and pragmas.

-no-def-init-glob overrides -main-generator-writes-variables

Behavior change

`Ignore default initialization of global variables (-no-def-init-glob)` option overrides `Variables to initialize (-main-generator-writes-variables)` option. If you specify the option `-no-def-init-glob`, global variables are considered as uninitialized until you explicitly initialize them in the code. Even if you use the option `-main-generator-writes-variables` to specify that the generated `main` initialize global variables, the analysis ignores the initialization.

Target & Compiler options Respect C90 standard (-no-language-extensions), C++11 extensions (-cpp11-extension), and are removed

Warns

Options **Respect C90 standard** (`-no-language-extensions`) and **C++11 extensions** (`-cpp11-extension`) are removed. Use options `C standard version (-c-version)` and `C++ standard version (-cpp-version)` instead.

In the Polyspace user interface, if an option is replaced by another option, the replacement occurs automatically in your configuration. To update your scripts, see this table.

Option	Use Instead
Respect C90 standard (`-no-language-extensions`)	Set the option `C standard version (-c-version)` to `c90`.
C++11 extensions (`-cpp11-extension`)	Set the option `C++ standard version (-cpp-version)` to `cpp11`.

You get a warning when you use the removed options at the command line.

Check Behavior options Detect overflow (-scalar-overflows-checks), Overflow computation mode (-scalar-overflows-behavior wrap-around), and Ignore overflowing computations on constants (-ignore-constant-overflows) are removed

Warns

Options **Detect overflow** (`-scalar-overflows-checks`) and **Overflow computation mode** (`-scalar-overflows-behavior wrap-around`) are removed. Use options `Overflow mode for signed integer (-signed-integer-overflows)` and `Overflow mode for unsigned integer (-unsigned-integer-overflows)` instead.

Option **Ignore overflowing computations on constants** (`-ignore-constant-overflows`) is removed. There is no replacement.

In the Polyspace user interface, if an option is replaced by another option, the replacement occurs automatically in your configuration. To update your scripts, see this table.

Option	Use Instead
Detect overflow (`-scalar-overflows-checks`)	See Check Behavior on Overflows release note.
Overflow computation mode (`-scalar-overflows-behavior wrap-around`)	See Check Behavior on Overflows release note.
Ignore overflowing computations on constants (`-ignore-constant-overflows`)	Overflows involving integer constants are wrapped around by default without warning. To detect integer constant overflows, use these Bug Finder checkers: • `Integer constant overflow` • `Unsigned integer constant overflow`

You get a warning when you use the removed options at the command line.

polyspace-configure option -lang is removed

Warns

Starting in R2018b, `polyspace-configure` detects the language of your source code.

Option `-lang` will be removed in a future release. You get a warning when you use this option and there is no replacement. To update your code, remove instances of `-lang`.

-compiler option value clang3.5 is removed

Errors

`Compiler (-compiler)` option value `clang3.5` is removed. Use `clang3.x` instead.

In the Polyspace user interface, if an option value is replaced by another option value, the replacement occurs automatically in your configuration. To update your scripts, see this table.

Option	Use Instead
`-compiler clang3.5`	`-compiler clang3.x`

You get an error when you use the removed option at the command line.

Changes in MATLAB option object properties and option values

polyspace.Project.Configuration has new TargetCompiler properties
Behavior change

`polyspace.Project.Configuration` has new `TargetCompiler` properties `CVersion` and `CppVersion`. Use these properties in your MATLAB code to specify the C and C++ language standards you follow in your source code.

For more information, see `Properties`.

polyspace.Project.Configuration has new ChecksAssumption properties
Behavior change

`polyspace.Project.Configuration` has new `ChecksAssumption` properties `SignedIntegerOverflows` and `UnsignedIntegerOverflows`. Use these properties in your MATLAB code to specify the behavior of checks for signed and unsigned integer overflows.

For more information, see `Properties`.

TargetCompiler properties NoLanguageExtensions and Cpp11Extension will be removed
Still runs

Properties `NoLanguageExtensions` and `Cpp11Extension` will be removed. Use `CVersion` and `CppVersion` instead.

To update your MATLAB code, see this table.

```
opts = polyspace.Project;
```

Property	Use Instead
`opts.Configuration.TargetCompiler...` `.NoLanguageExtensions = true;`	`opts.Configuration.TargetCompiler...` `.CVersion = 'c90';`
`opts.Configuration.TargetCompiler...` `.Cpp11Extension = true;`	`opts.Configuration.TargetCompiler...` `.CppVersion = 'cpp11';`

Unlike `NoLanguageExtensions` and `Cpp11Extension` which let you specify one version of the C and C++ language standards, the new object properties `CVersion` and `CppVersion` let you specify different versions of these standards.

For more information, see `Properties`.

ChecksAssumption properties ScalarOverflowsBehavior and ScalarOverflowsChecks will be removed

Still runs

Properties `ScalarOverflowsBehavior` and `ScalarOverflowsChecks` will be removed. Use `SignedIntegerOverflows` and `UnsignedIntegerOverflows` instead.

To update your MATLAB code, see this table.

```
opts = polyspace.Project;
```

Property	Use Instead

`opts.Configuration.ChecksAssumption... .ScalarOverflowsBehavior`
 	
`opts.Configuration.ChecksAssumption... .ScalarOverflowsChecks`
	See Compatibility Considerations in the Check Behavior on Overflow release note.

For more information, see `Properties`.

polyspaceConfigure option -lang is removed

Warns

Starting in R2018b, `polyspaceConfigure` detects the language of your source code.

Option `-lang` will be removed in a future release. You get a warning when you use this option and there is no replacement. To update your code, remove instances of `-lang`.

ChecksAssumption property IgnoreConstantOverflows is removed

Errors

Property `IgnoreConstantOverflows` is removed. There is no replacement.

Overflows involving integer constants are wrapped around by default without warning. To detect integer constant overflows, use these Bug Finder checkers:

- `Integer constant overflow`
- `Unsigned integer constant overflow`

For more information, see `Properties`.

Verification Results

C++ Specific Checks: View more pertinent results for incorrect object oriented programming and exception handling checks

In R2018b, you see more pertinent results for the checks **Incorrect object oriented programming** and **Uncaught exception**.

Object Oriented Programming Checks

Summary: In R2018b, the **Incorrect object oriented programming** check detects member function call using an incorrect `this` pointer only if the member function is `virtual`.

You might call a member function using an incorrect `this` pointer, for instance, after you cast between pointers to two objects. In this example of incorrect C++ polymorphism, you might intend to call the derived class version of `funcFoo` but depending on your compiler, you call the base class version or encounter a segmentation fault.

Result Details

Result Review

Incorrect object oriented programming
Error: incorrect this-pointer type of funcFoo

Source

file.cpp

```
1  #include <new>
2
3  class Foo {
4  public:
5    virtual void funcFoo() {}
6  };
7
8
9  class Bar: public Foo {
10 public:
11   void funcFoo() {}
12 };
13
14 void main() {
15   Foo *FooPtr = new Foo;
16   Bar *BarPtr = (Bar*)(void*)FooPtr;
17   BarPtr->funcFoo();
18 }
```

For more examples, see `Incorrect object oriented programming`.

Benefits: Calling a non-`virtual` function via an incorrect `this` pointer is a less likely scenario. Code Prover now restricts the error detection to cases that are more likely to occur and might not be caught at compile time.

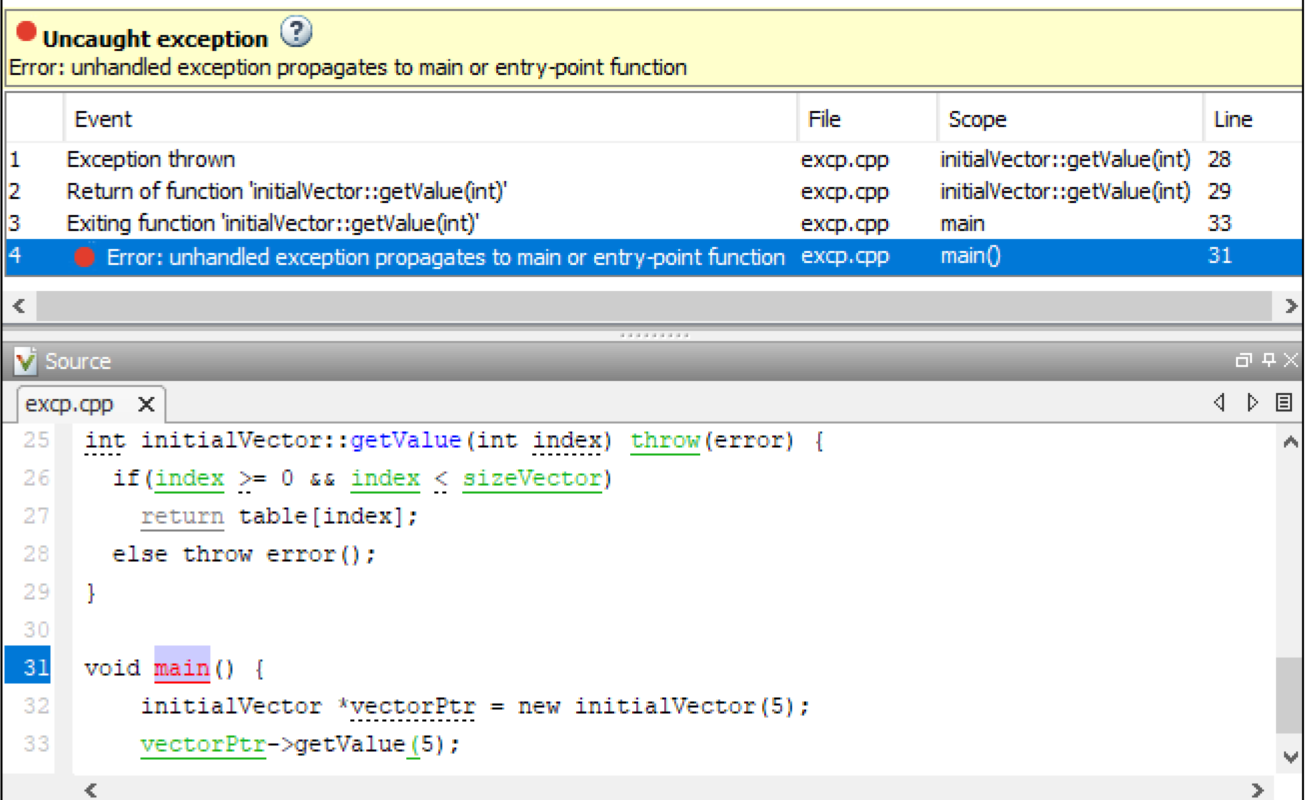
Exception Handling Checks

Summary: In R2018b, the **Uncaught exception** check shows only those cases where the exception handling fails and the std::terminate function is called. For instance:

- An exception is thrown and propagates uncaught to the `main`.
- An exception is thrown during construction of a global variable.

For the full list of cases, see `Uncaught exception`.

For instance, if your code has exceptions that propagate uncaught up to the `main`, the **Uncaught exception** check shows the result only on the `main`. In the event traceback associated with the check, you see the origin of the exception and its propagation up the function call tree to the `main` or another entry-point function. Click each event to navigate to the corresponding point in the source code.



Benefits:

- *Fewer results from same root cause*: If an exception propagates uncaught up to the `main` function, you see only one result on the `main` function. Previously, you saw a separate result for each function in the call tree from the point where the exception is thrown up to the `main`. You had to review several results originating from the same root cause.
- *Easier tracing of exception propagation*: You can use the event traceback to track down an uncaught exception to its origin. Previously, to find the root cause of a red **Uncaught exception** check, you manually navigated the function call tree from the `main` or another entry-point function.

Compatibility Considerations

- *Object oriented programming checks*: You see fewer red or orange **Incorrect object oriented programming** checks in C++ code. Because the blocking red or orange checks no longer appear, subsequent code is now verified. You can see an increase in checks of other types.
- *Exception handling checks*: You see fewer **Uncaught exception** checks in C++ code.

Checks on List-Initialization of Arrays: Detect list-initialization with excess initializer clauses (C++11 and beyond)

Summary: In R2018b, the **Invalid C++ specific operations** check detects if the number of array initializer clauses exceeds the number of elements to initialize.

For instance, the check detects an error if `size` is less than two.

```
arr_const = new int[size]{0,1};
```

See also `Invalid C++ specific operations`.

Benefits: Compilers detect excess initializer clauses if the array size is a compile-time constant. If the array size is dynamically determined at run-time, compilers can fail to detect the error. With Code Prover, you can detect such situations.

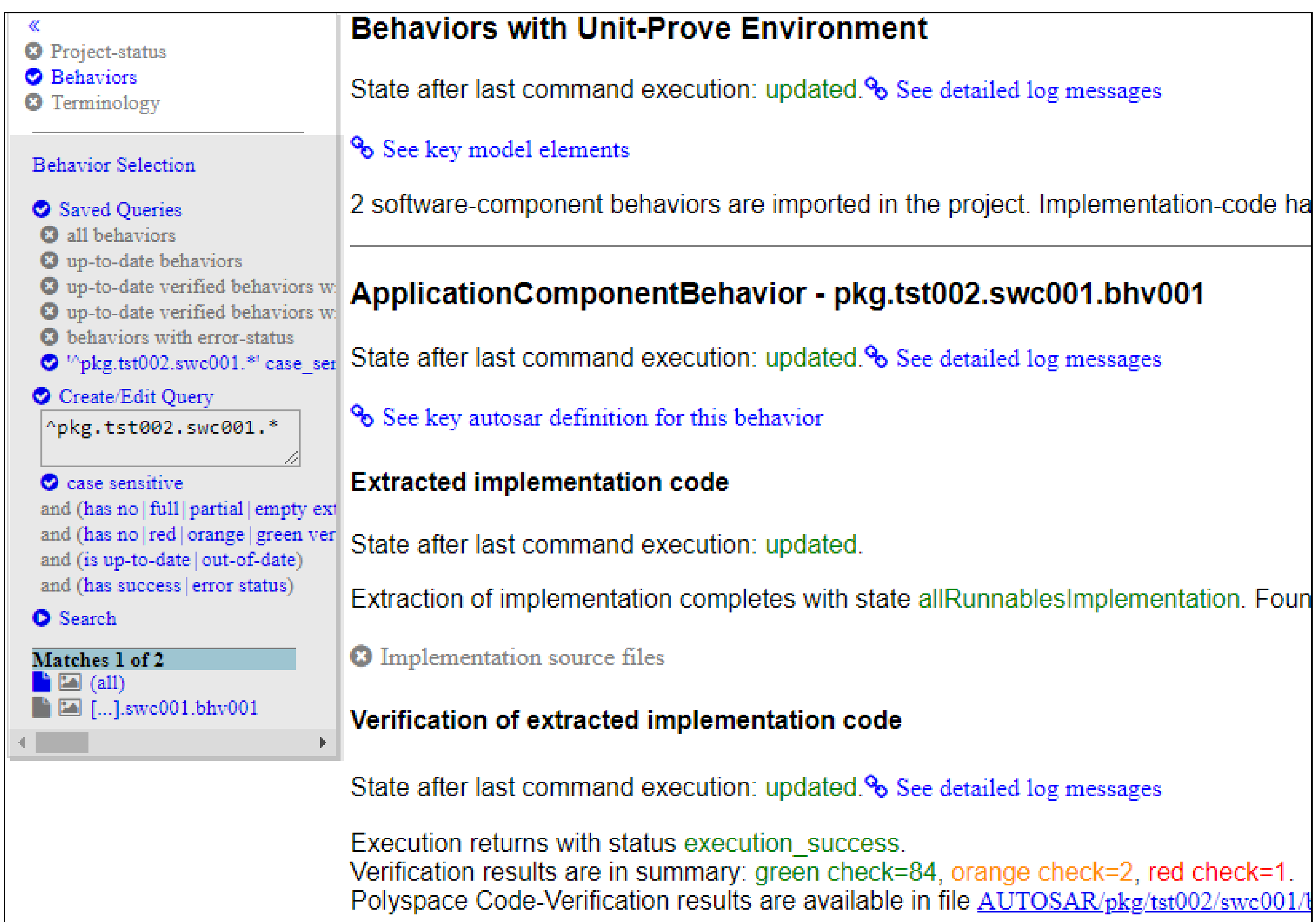
Reviewing Results

AUTOSAR Support: Focus review to specific software components with queries based on regular expressions

Summary: In R2018b, you can quickly filter specific software components when reviewing Code Prover results for your complete AUTOSAR project. For instance, you can:

- Filter software components whose qualified names start with a specific string.

  You can use regular expressions for the pattern matching.
- Filter software components whose code implementation compiled successfully.
- Filter software components that have red checks in the Code Prover results.



See also Review Polyspace Results on AUTOSAR Code.

Benefits:

- *More focused review*: Because you provide your top level ARXML folder for analysis, the results show all software components in your AUTOSAR project. You can quickly focus your review on the software component-s that you are interested in.
- *Easier troubleshooting*: You can quickly find which software components have errors in ARXML parsing or code compilation. See Troubleshoot Polyspace Analysis of AUTOSAR Code.
- *Easier maintenance*: You can quickly find which software components have results that have not changed since the previous analysis.

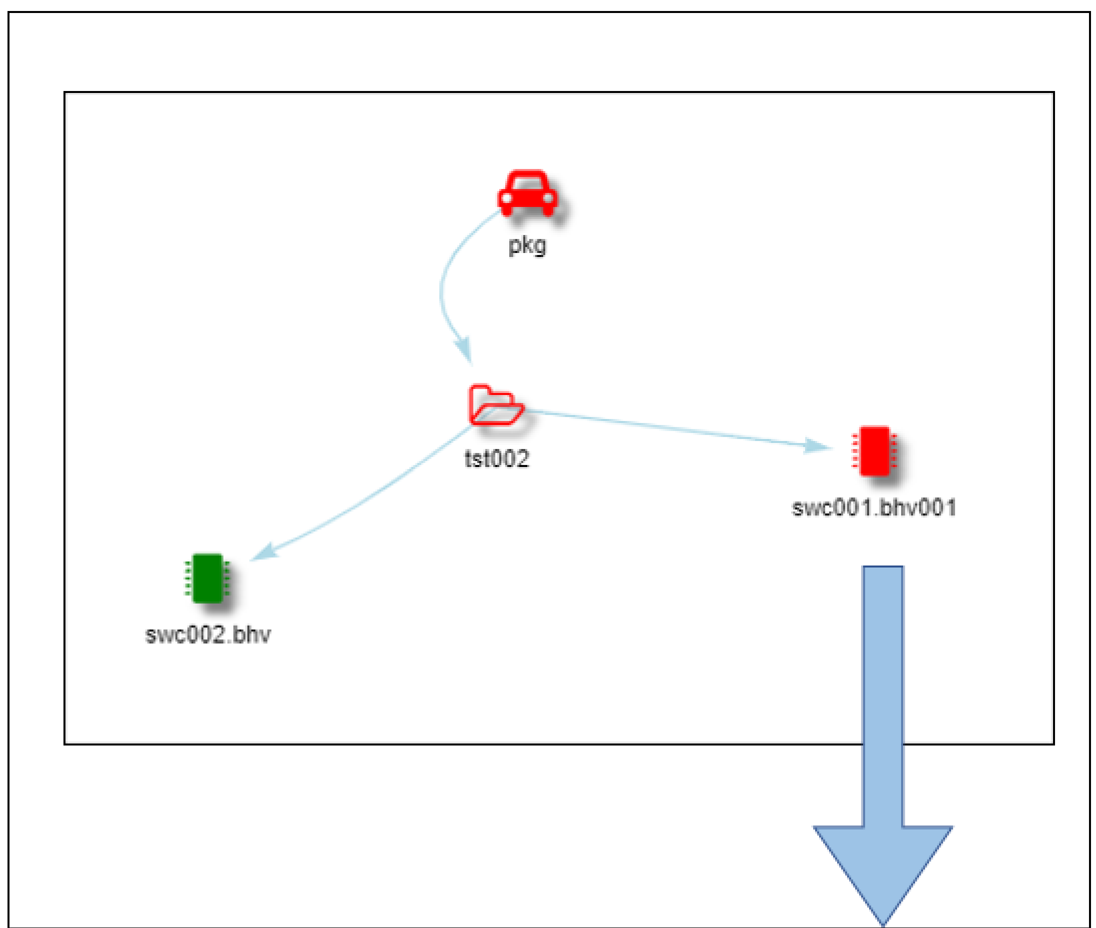
AUTOSAR Support: See visual representation of runnables and associated files for each software component

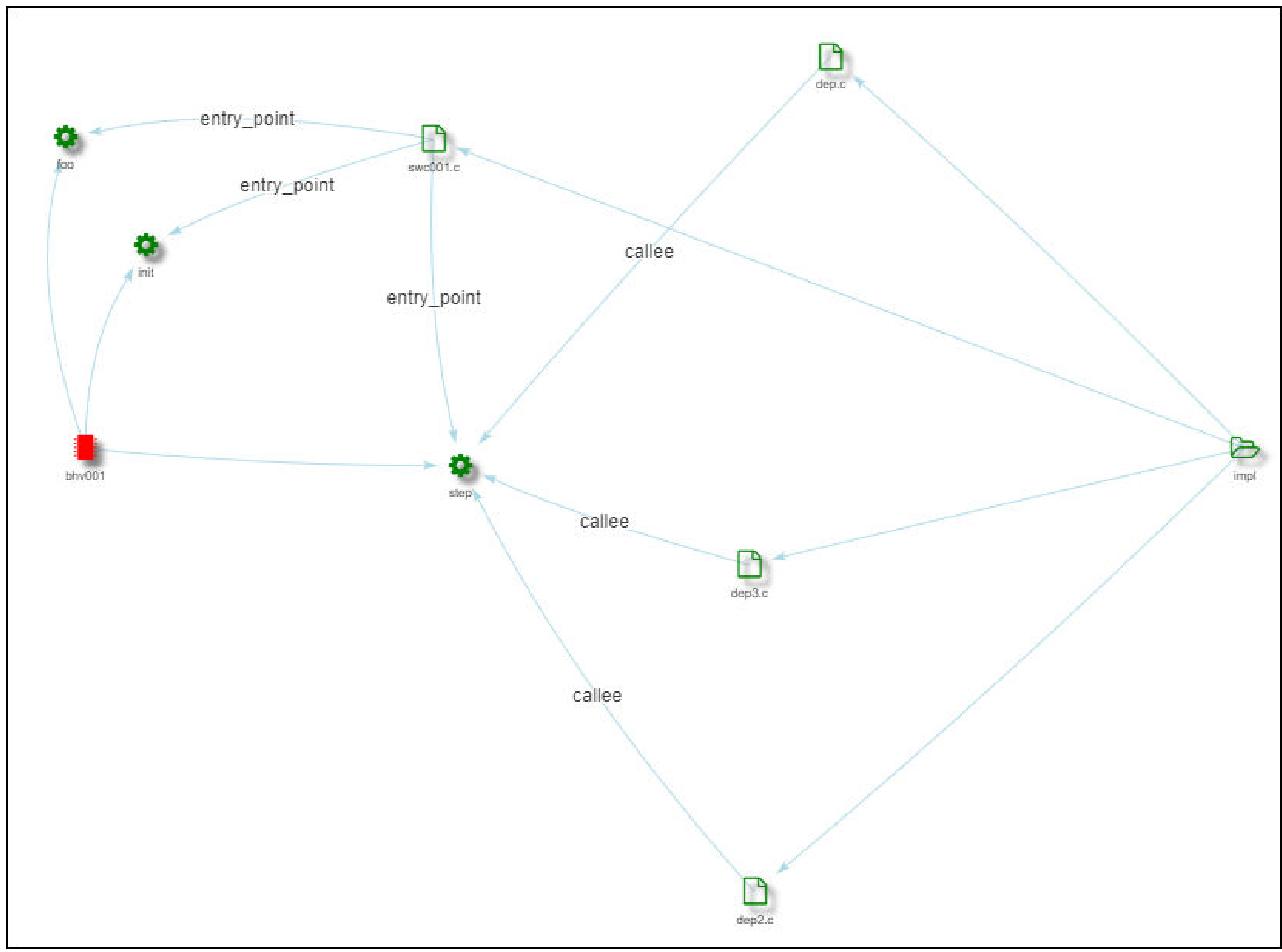
Summary: In R2018b, you can see a visual representation of your complete AUTOSAR project with all software components along with an overview of Code Prover results for each software component.

You can drill down to these details for each software component behavior:

- Entry-point functions and their callees
- Files containing these functions

For instance, the project below contains two software components with full name `pkg.tst002.swc001.bhv001` and `pkg.tst002.swc002.bhv`.


pkg
tst002
swc001.bhv001
swc002.bhv



The code implementation of `pkg.tst002.swc001.bhv001` has three entry point functions: `foo`, `init` and `step`. All three functions are defined in the file `swc001.c`. These entry-point functions call other functions defined in the files `dep.c`, `dep2.c` and `dep3.c`.

See also Review Polyspace Results on AUTOSAR Code.

Benefits:

- *More focused review*: Each software component is shown with the color of the worst verification result. For instance, if the code implementation of a software component contains red checks, the corresponding icon is red. You can use the icon colors to quickly focus on the software components that need attention.
- *Easier tracking of information*: You can see at a glance the files involved in the code implementation of software components.

Header Files Access: Open your project header files directly from the point of inclusion

Summary: In R2018b, you can open header files you reference in your code by right-clicking on the include directive in the **Source** pane.

Source

programming.cpp

```
22  #include <stdint.h> /* C99 standard types */
23  #include <limits.h>
24  #include <errno.h>
25  #include <float.h>
26  #include <signal.h>
27  #include <sys/types.h>
28  #include <sys/socket.h>
29  #include <arpa/inet.h>
30  #include <unistd.h>
31  #include <math.h>
32  #include "bf_example_types.h"
33
34  #define fatal_error() abort()
35
36  volatile int some_condition = 1;
37
38  enum {
39      SIZE4  = 4,
40      SIZE5  = 5,
41      SIZE6  = 6,
42      SIZE20 = 20,
43      SIZE1024 = 1024
44  };
```

If Polyspace determines that the header file is available, the `#include`, `#import`, or `#include_next` preprocessor directive is underlined in the source code.

Benefits: When you review results, you can quickly see the contents of a header file without leaving the Polyspace user interface.

R2018a

Version: 9.9

New Features

Bug Fixes

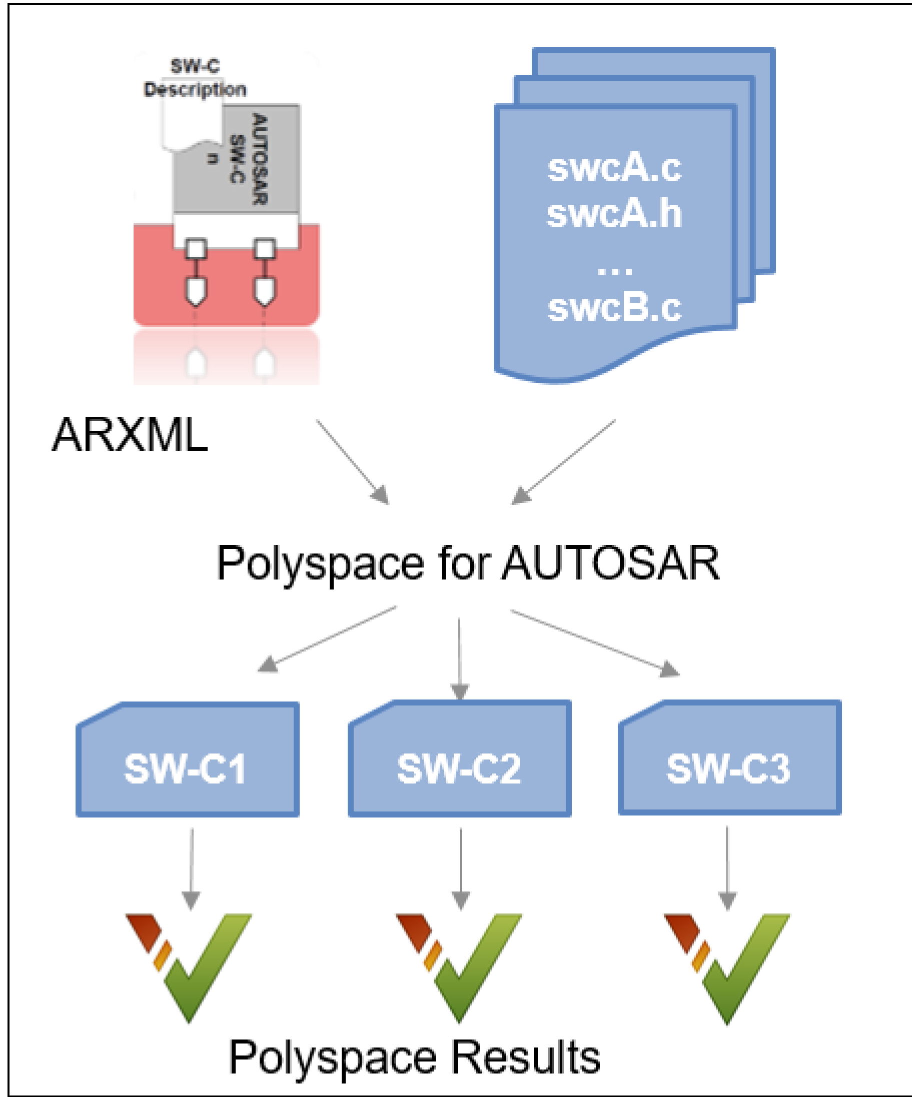
Compatibility Considerations

Verification Setup

AUTOSAR Support: Set up modular Polyspace analysis for AUTOSAR software components automatically

Summary: In R2018a, Polyspace can read specifications for AUTOSAR software components (SWCs) and modularize the corresponding C code implementation. A separate module is created for each software component and checked for run-time errors. Polyspace also detects certain kinds of mismatch that can happen between AUTOSAR specifications and the code implementation at run time.

To perform this modular Polyspace analysis, you simply provide the folders containing the AUTOSAR specifications (`.arxml` files) and the code implementation (`.c` files).



For details, see:

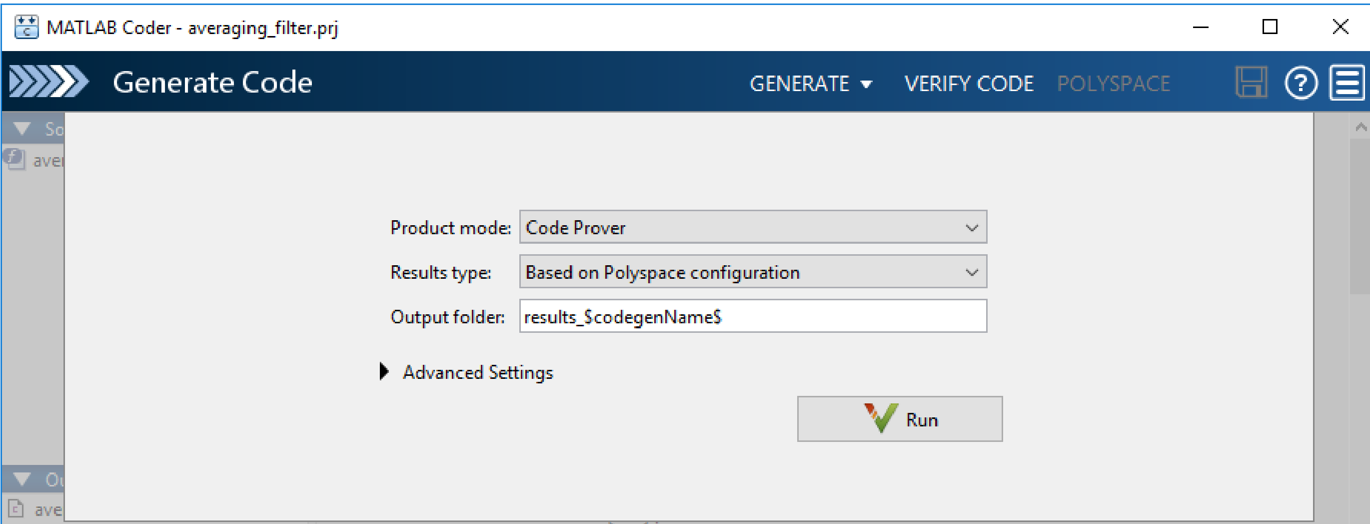
- Using Polyspace in AUTOSAR Software Development
- Run Polyspace on AUTOSAR Code

Benefits:

- *Modularization based on AUTOSAR specs*: The software reuses the modularization already present in your AUTOSAR specifications and verifies the code implementation of each software component (SWC) behavior independently. Previously, to emulate your AUTOSAR modularization, you manually copied files to modules.
- *Minimal effort and knowledge required for configuration*: You do not need to know the details of the AUTOSAR specifications or the code implementation for running a Polyspace analysis. You simply provide the two folders containing your `.arxml` and `.c` files. The software extracts the code implementation of each SWC behavior, creates a prove environment to exercise each runnable with all allowed inputs and checks for run-time errors or mismatch with AUTOSAR specifications.
- *Automatic range specification and precise analysis*: The data type specification in `.arxml` files allows specifying a range constraint on values. The software reuses this range constraint to constraint input variables to allowed values, allowing for a more precise analysis. Previously, for a precise analysis, you had to manually specify range information for all input variables in your code.

MATLAB Coder Support: Run Polyspace on C/C++ code generated from MATLAB code without additional setup

Summary: In R2018a, if you install Embedded Coder® and Polyspace, you can run Polyspace directly on C/C++ code generated from MATLAB code and check for defects (Bug Finder) or run time errors (Code Prover).



For details, see:

- Run Polyspace on C/C++ Code Generated from MATLAB Code
- Configure Advanced Polyspace Options in MATLAB Coder App

Benefits:

- *Seamless integration*: You do not have to configure the Polyspace analysis manually, in the Polyspace user interface or otherwise. The Polyspace analysis is seamlessly integrated with the workflow in the MATLAB Coder™ App.
- *Easier scripting*: You do not have to know or specify names of files generated from your MATLAB code. You can simply use a specific folder for code generation output and provide that folder for code analysis. In this way, you can have end-to-end scripting for the code generation and analysis.

Compiler Support: Set up Polyspace analysis easily for code compiled with Texas Instruments, IAR or CodeWarrior compilers

Summary: If you build your source code using these compilers, in R2018a, you can specify the compiler name for your Polyspace analysis:

- Texas Instruments™

  You can specify these target processors: `c28x`, `c6000`, `arm` and `msp430`.

  See Texas Instruments Compiler (`-compiler ti`) .
- IAR

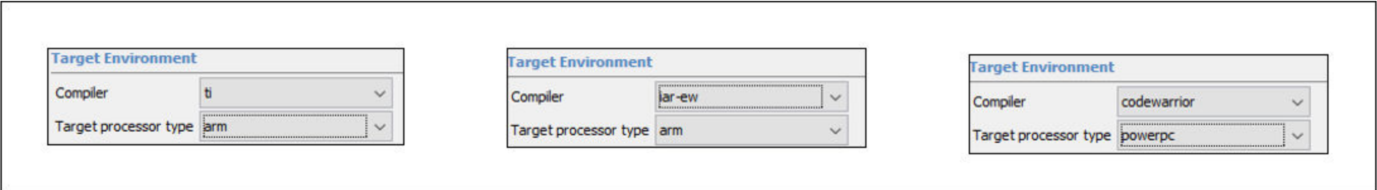
  You can specify these target processors: `arm`, `avr`, `msp430`, `rh850` and `rl78`.

  See IAR Embedded Workbench Compiler (`-compiler iar-ew`).
- CodeWarrior

  You can specify these target processors: `s12z` or `powerpc`.

  NXP CodeWarrior Compiler (`-compiler codewarrior`)

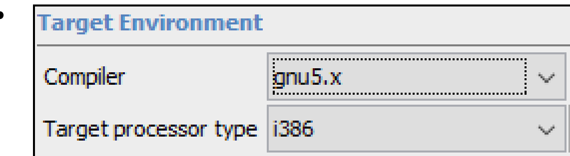
The analysis can interpret macros that are implicitly defined by the compiler and compiler-specific language extensions such as keywords and pragmas.

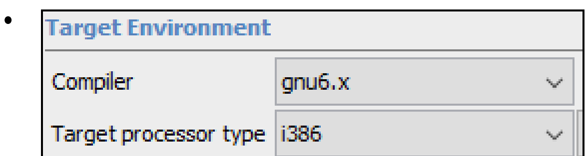


Benefits: You can now set up a Polyspace project without knowing the internal workings of these compilers. If your code compiles with your compiler, it will compile with Polyspace in most cases without requiring additional setup. Previously, you had to explicitly define macros that were implicitly defined by the compiler and remove unknown language extensions from your preprocessed code.

Updated GCC and Clang Compiler Support: Set up Polyspace analysis easily for code compiled with GCC versions 5.x or 6.x, or Clang version 3.x compilers

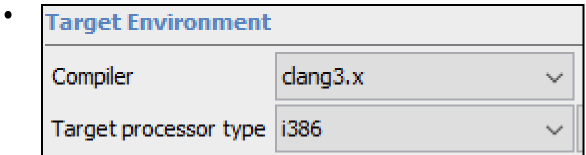
Summary: In R2018a, if you build your source code using these versions of GCC or Clang compilers, you can specify the following compiler option values to setup your Polyspace analysis:

- 

  `gnu5.x`, for GCC release 5.1, 5.2, 5.3, and 5.4.
- 

  `gnu6.x`, for GCC release 6.1, 6.2, and 6.3.

Starting GCC version 5, the version number increases by one for each major release, for instance,.from 5.x to 6.x. Polyspace follows this new naming convention.

- 

  `clang3.x`, for LLVM release 3.5, 3.6, 3.7, 3.8, and 3.9.

The analysis can interpret macros that are implicitly defined by the compiler and compiler-specific language extensions such as keywords and pragmas.

For more information, see `Compiler (-compiler)`.

Configuration from Build System: Include or exclude sources when generating Polyspace project using polyspace-configure

Summary: In R2018a, you can include or exclude source files or folders when generating a Polyspace project from your build system.

To create a Polyspace project that does not contain all files from your build system:

1. Trace your build command. Do not create a project yet. Optionally store the build trace and cache in specific locations (instead of the default).

   ```
   polyspace-configure -no-project make -B \
     -build-trace trace.txt -cache-path /tmp/cache
   ```
2. Create a Polyspace project using the build trace and cache. Include or exclude files as needed using shell GLOB patterns.

   ```
   polyspace-configure -no-build \
     -build-trace trace.txt -cache-path /tmp/cache \
     -include-sources 'src/' -exclude-sources '*_test.c'
   ```

   The preceding example includes sources in folder paths containing `src` and excludes `.c` files ending with `_test`.

3 Delete the build trace and cache.

For more information, see polyspace-configure.

Benefits:

- *Exclusion of irrelevant files*: You can avoid cluttering your Polyspace project with files that you do not want to analyze, for instance, files used for testing.
- *Modular analysis*: You can create a separate Polyspace project for each module covered by your build system. Trace your build command once. When creating a Polyspace project, include only files belonging to a specific module. Repeat the project creation step for each module.

Support for IBM Rational Rhapsody to be removed

The Polyspace integration with the IBM® Rational Rhapsody environment will be removed after R2018b.

Compatibility Considerations

To continue using the latest releases of Polyspace, run code analysis in the Polyspace user interface or using scripts.

Changes in analysis options and binaries

Polyspace Code Prover has a new Multitasking option

Behavior change

Polyspace Bug Finder has a new **Multitasking** configuration option `ARXML files selection (-autosar-multitasking)`.

Use this option to automatically detect the multitasking configuration from your AUTOSAR specification.

Polyspace Code Prover has new -compiler option values

Behavior change

Use the new `Compiler (-compiler)` option values to interpret macros that are implicitly defined by the compilers and compiler-specific language extensions such as keywords and pragmas..

Option	New Value
Compiler (-compiler)	• New value ti added. See Compiler Support release note. • New value iar-ew added. See Compiler Support release note. Use this value to emulate IAR compilers. For older Polyspace projects, you can still use option value iar. • New value codewarrior added. See Compiler Support release note. • New value gnu5.x added. See Updated GCC and Clang Compiler Support release note. • New value gnu6.x added. See Updated GCC and Clang Compiler Support release note. • New value clang3.x added. See Updated GCC and Clang Compiler Support release note.

-compiler option value clang3.5 is removed
Warns

Compiler (-compiler) option value clang3.5 is removed. Use clang3.x instead.

In the Polyspace user interface, if an option value is replaced by another option value, the replacement occurs automatically in your configuration. To update your scripts, see this table.

Option	Use Instead
-compiler clang3.5	-compiler clang3.x

You get a warning when you use the removed option value at the command line.

-compiler option values iso, none, gnu, and visual through visual10 are removed
Errors

`Compiler (-compiler)` option values `iso`, `none`, `gnu`, `visual`, `visual6`, `visual7.0`, `visual7.1`, `visual8`, and `visual10` are removed.

In the Polyspace user interface, if an option value is replaced by another option value, the replacement occurs automatically in your configuration. To update your scripts, see this table.

Option	Use Instead
`-compiler iso` `-compiler none`	`-compiler generic`
`-compiler gnu`	`-compiler gnu3.4`

`-compiler visual`
`-compiler visual6`
`-compiler visual7.0`
`-compiler visual7.1`
`-compiler visual8`
`-compiler visual10`
	`-compiler visual10.0`

You get a error when you use the removed options at the command line.

Check Behavior options Allow incomplete or partial allocation of structures (-size-in-bytes) and Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct) are updated
Behavior change

Option `Allow incomplete or partial allocation of structures (-size-in-bytes)` is available for C++ projects.

Option `Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)` does not automatically enable `Allow incomplete or partial allocation of structures (-size-in-bytes)`.

Target&Compiler options Set wchar_t to unsigned long (-wchar-t-is-unsigned-long) and Set size_t to unsigned long (-size-t-is-unsigned-long) are removed
Errors

Option **Set wchar_t to unsigned long** `(-wchar-t-is-unsigned-long)` is removed. Set Management of wchar_t (`-wchar-t-type-is`) to `unsigned-long` instead.

Option **Set size_t to unsigned long** `(-size-t-is-unsigned-long)` is removed. Set Management of size_t (`-size-t-type-is`) to `unsigned-long` instead.

In the Polyspace user interface, if an option is replaced by another option, the replacement occurs automatically in your configuration. To update your scripts, replace each instance of the removed option with the corresponding new option.

You get an error when you use the removed options at the command line.

Command-line option -static-headers-object is removed
Errors

Analysis option `-static-headers-object` is removed. There is no replacement. The permissive linking introduced by this option now happens by default.

In the Polyspace user interface **Configuration** pane, under **Advanced Settings**, remove the option for the Other field. To update your scripts, remove instances of this option. If you use this option, you get an error.

-enum-type-definition option value defined-by-standard is removed

Errors

Enum type definition (`-enum-type-definition`) option value `defined-by-standard` is removed. Use `defined-by-compiler` instead.

In the Polyspace user interface, if an option value is replaced by another option value, the replacement occurs automatically in your configuration. To update your scripts, see this table.

Option	Use Instead
`-enum-type-definition defined-by-standard`	`-enum-type-definition defined-by-compiler`

You get an error when you use the removed option value at the command line.

Changes in MATLAB option object properties

polyspace.Project.Configuration has new Multitasking properties

Behavior change

`polyspace.Project.Configuration` has new `Multitasking` properties `EnableExternalMultitasking`, `ExternalMultitaskingType`, and `ArxmlMultitasking`. Use these properties to set up the multitasking configuration of your project from external files you provide.

For more information, see `Properties`.

TargetCompiler property has a new Compiler option values

Behavior change

Use the new `Compiler` option values to interpret macros that are implicitly defined by the compilers and compiler-specific language extensions such as keywords and pragmas.

```
opts=polyspace.Project;
```

Property	Description
`opts.Configuration...` `.TargetCompiler.Compiler`	• New value `ti` added. See Compiler Support release note. • New value `iar-ew` added. See Compiler Support release note. Use this value to emulate IAR compilers. For older Polyspace projects, you can still use property value `iar`. • New value `codewarrior` added. See Compiler Support release note. • New value `gnu5.x` added. See Updated GCC and Clang Compiler Support release note. • New value `gnu6.x` added. See Updated GCC and Clang Compiler Support release note. • New value `clang3.x` added. See Updated GCC and Clang Compiler Support release note.

For more information, see `Properties`.

Multitasking property EnableOsekMultitasking is removed

Errors

Property `EnableOsekMultitasking` is removed. To update your MATLAB code, see this table.

```
opts=polyspace.Project;
```

Property	Description

`opts.Configuration.Multitasking...`
`.EnableOsekMultitasking`
	`opts.Configuration.Multitasking...`
	`.EnableExternalMultitasking=1;`
	`opts.Configuration.Multitasking...`
	`.ExternalMultitaskingType='osek';`

If you use the removed property, you get an error.

For more information, see `Properties`.

TargetCompiler properties WcharTIsUnsignedLong and SizeTIsUnsignedLong are removed

Errors

Properties `WcharTIsUnsignedLong` and `SizeTIsUnsignedLong` are removed. To update your MATLAB code, see this table.

```
opts=polyspace.Project;
```

Property	Description
`opts.Configuration.TargetCompiler...` `.WcharTIsUnsignedLong`	`opts.Configuration.TargetCompiler...` `.WcharTTypeIs="unsigned-long"`

Property	Description
opts.Configuration.TargetCompiler... .SizeTIsUnsignedLong	opts.Configuration.TargetCompiler... .SizeTTypeIs="unsigned-long"

If you use the removed property, you get an error.

For more information, see `Properties`.

EnumTypeDefinition option value defined-by-dialect is removed

Errors

`EnumTypeDefinition` option value `defined-by-dialect` is removed. To update your MATLAB code, see this table.

```
opts=polyspace.Project;
```

Property	Description
opts.Configuration.TargetCompiler... .EnumTypeDefinition="defined-by-dialect"	opts.Configuration.TargetCompiler... .EnumTypeDefinition="defined-by-compiler"

If you use the removed property, you get an error.

For more information, see `Properties`.

Verification Results

AUTOSAR Support: Check for run-time mismatch between AUTOSAR specifications and code implementation

Summary: In R2018a, in addition to regular checks for run-time errors, Polyspace can detect certain kinds of mismatch between AUTOSAR specifications and the corresponding code implementation. For instance, the software detects if certain variables in your code can acquire values outside their specifications at run time.

The software performs these checks for two classes of variables.

- **Invalid use of AUTOSAR runtime environment function**: The check applies to arguments of functions supplied by the Run Time Environment (functions beginning with `Rte_`). See `Invalid use of AUTOSAR runtime environment function`.
- **Invalid result of AUTOSAR runnable implementation**: The check applies to output arguments and return value from runnable entities (functions provided by the software components). See `Invalid result of AUTOSAR runnable implementation`.

```
Invalid use of AUTOSAR runtime environment function
Error: Function 'Rte_Write_outRef_colorCount' is called with invalid argument(s)
 • Conditions on data (see spec):
   ✓ data meets its specification:
       Specification: non-NULL

   ✓ data meets its specification:
       Specification: allocated

   ! data->color does not meet its specification:
       Specification: {4U,5U,9U}
       Actual value (const unsigned int 8): 12

 • Conditions on self (see spec):
   ✓ self meets its specification:
       Specification: non-NULL
```

Benefits:

- *Runtime errors detected*: Using static analysis, the software detects all possible invalid argument values that can occur at run time. The analysis represents the most comprehensive testing possible for this kind of mismatch with AUTOSAR specs.
- *Easy navigation between code and spec*: If the software detects a mismatch for a specific variable, you can navigate to an extract of the AUTOSAR specification that describes the variable data type and allowed values. You can also see the application data type (with units) from which the implementation data type and the software base type is derived and the computation method used for this derivation.
- *Easy collaboration between AUTOSAR spec development and coding*: If an argument acquires invalid values, it is easy to see the variable data type in an extract of the AUTOSAR specs. In situations where the code requires a change in the specs, for instance, if an enumeration requires an additional value, it is easy to use the extract to request this change. In this way, the AUTOSAR specification can be kept up-to-date with requirements from the code implementation. See also Using Polyspace in AUTOSAR Software Development.

MISRA C++ Support: Check for overriding of standard library functions, missing const qualifiers and other MISRA C++ rules

Summary: In R2018a, you can look for violations of these MISRA® C++ rules.

Rule	Description
0-1-3	A project shall not contain unused variables.
0-1-5	A project shall not contain unused type declarations.
4-10-1	NULL shall not be used as an integer value.
4-10-2	Literal zero (0) shall not be used as the null-pointer constant.
7-1-1	A variable which is not modified shall be const qualified.
7-1-2	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.
9-3-3	If a member function cannot be made static then it shall be made static, otherwise if it can be made const then it shall be made const.
15-5-3	The terminate() function shall not be called implicitly.
17-0-3	The names of standard library functions shall not be overridden.

See also MISRA C++ Coding Rules.

MISRA C:2012 Directives: Detect opportunities for data hiding

Summary: In R2018a, you can look for violations of MISRA C:2012 Directive 4.8. The directive states that if a pointer to a structure is never dereferenced in a translation unit, its implementation must be hidden in that unit.

See `MISRA C:2012 Directive 4.8`.

Benefits: Using this check, you can find opportunities for defining opaque data types that hide the implementation of a structure.

Rule for Source Line Length: Constrain number of characters per line in your code

Summary: In R2018a, you can define a limit for number of characters per line in your code and use Polyspace to check for lines that fall outside that limit.

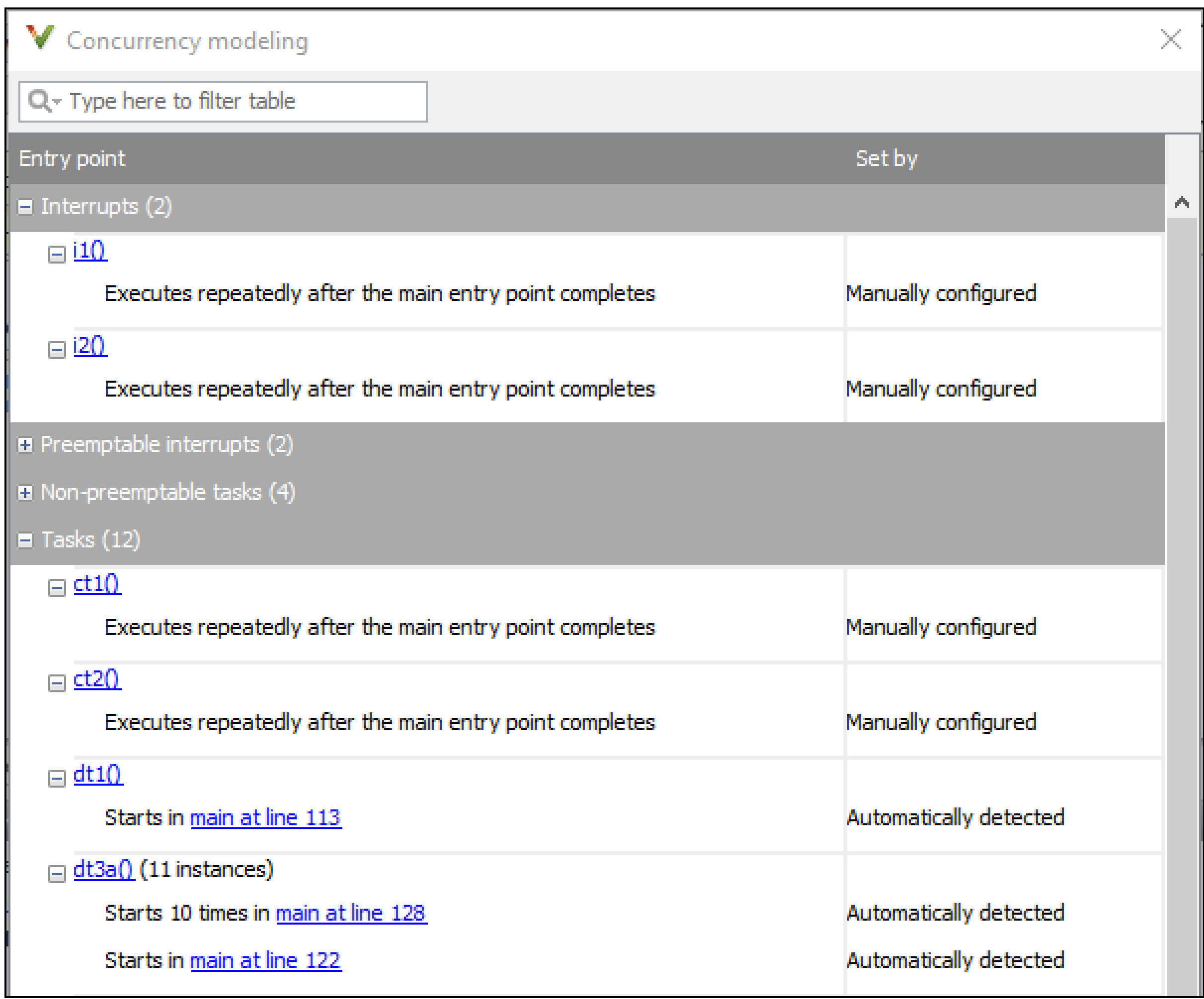
Use custom rule 20.1 and specify the character limit as the rule pattern. See Group 20: Style.

Reviewing Results

Concurrency Modeling: View all tasks and interrupts extracted from code and Polyspace configuration in one view

Summary: In R2018a, you can see the tasks and interrupts extracted from your code and configuration in one view.

After analysis, click the **Concurrency modeling** link on the **Dashboard**.



Benefits: You can verify if Polyspace correctly detected your multitasking configuration from your code. For instance, if you know a priori that a specific function acts as an interrupt, you can spot-check whether Polyspace considers the function as an interrupt.

This information is also included in reports you generate from the analysis results.

Variables Reporting: Export variable list to text file for automated reading

Summary: In R2018a, you can export the list of global variables in your code to a text file, along with the read and write operations on them.

Benefits: You can parse the text file by using MATLAB or Excel® and generate graphs or statistics about your global variables that you cannot readily obtain from the user interface. See Export Global Variable List.

R2017b

Version: 9.8

New Features

Bug Fixes

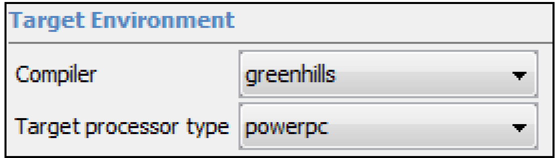
Compatibility Considerations

Verification Setup

Green Hills Compiler Support: Set up Polyspace analysis easily for code compiled with Green Hills Compiler

Summary: If you build your source code with the Green Hills® compiler, in R2017b, you can specify the compiler name for your Polyspace analysis. The analysis can interpret macros that are implicitly defined by the compiler and compiler-specific language extensions such as keywords and pragmas.

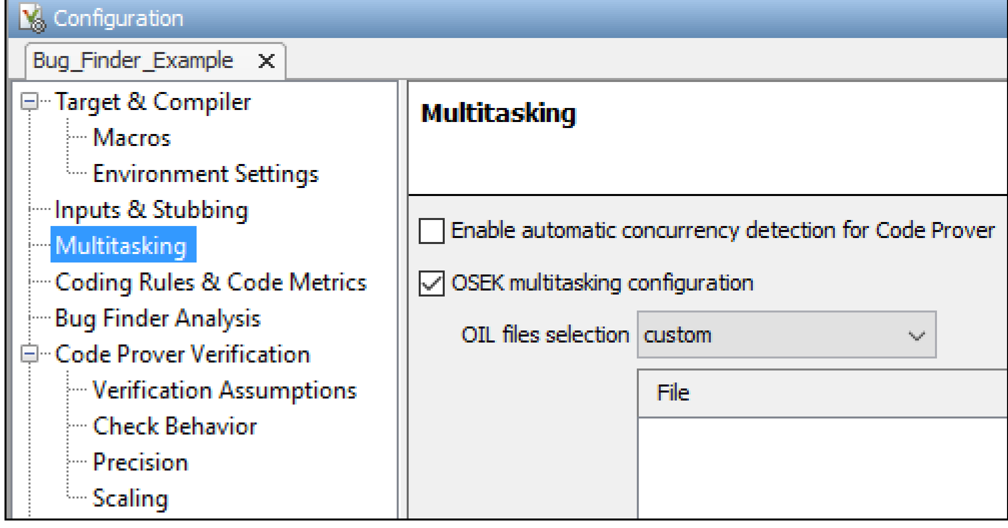
You can specify these target processors directly: `arm64`, `arm`, `i386`, `x86_64`, `powerpc`, `powerpc64`, `rh850` or `tricore`. See Green Hills Compiler (`-compiler greenhills`).



Benefits: You can now set up a Polyspace project without knowing the internal workings of your Green Hills compiler. If your code compiles with your compiler, it will compile with Polyspace in most cases without requiring additional setup. Previously, you had to explicitly define macros that were implicitly defined by the compiler and remove unknown language extensions from your preprocessed code.

OSEK Multitasking Support: Detect the multitasking configuration for your OSEK application automatically

Summary: In R2017b, you can provide an OIL file that Polyspace parses to detect the multitasking configuration for your OSEK application. Polyspace can interpret the OIL file definitions to set up your concurrency model.



For more information, see `OSEK multitasking configuration (-osek-multitasking)`.

Benefits: You no longer need to configure multitasking manually to analyze your OSEK application. Polyspace detects the tasks, interrupts, and critical sections of your model.

Polyspace API in MATLAB: Configure analysis, run analysis, and read analysis results with a single MATLAB object

Summary: In R2017b, you can use a single MATLAB object for the entire Polyspace analysis. The analysis has two subobjects, one for configuring the analysis and another for reading the results.

```
obj = polyspace.Project

% Configure analysis
obj.Configuration.Sources = {fullfile(matlabroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
obj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
obj.Configuration.ResultsDir = fullfile(pwd,'results');
obj.Configuration.CodeProverVerification.MainGenerator = true;


% Run analysis
cpStatus = obj.run('codeProver');

% Read results
cpSummary = obj.Results.getSummary();
```

For more information, see `polyspace.Project`.

Benefits: You need fewer variables for the Polyspace analysis. You can also use the same object for reading both Bug Finder and Code Prover results.

Additional Considerations

Are the pre-R2017b ways of scripting a Polyspace analysis still supported?

The objects `polyspace.Options`, `polyspace.BugFinderResults` and `polyspace.CodeProverResults` are still supported. For easier scripting, it is recommended that you make the following replacements:

- To configure analysis, instead of the `polyspace.Options` object, use the `Configuration` subobject of the `polyspace.Project` object.

  For instance, instead of:

  ```
  opts = polyspace.Options

  opts.ResultsDir = fullfile(pwd,'results');
  ```

  Use:

  ```
  obj = polyspace.Project

  obj.Configuration.ResultsDir = fullfile(pwd,'results');
  ```
- To read results, instead of the `polyspace.BugFinderResults` and `polyspace.CodeProverResults` objects, use the `Results` subobject of the `polyspace.Project` object.

  For instance, instead of:

  ```
  resultsFolder = fullfile(pwd,'results');

  opts = polyspace.Options;
  ```

```
opts.Sources = {fullfile(matlabroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
opts.CodeProverVerification.MainGenerator = true;
opts.ResultsDir = resultsFolder;

polyspaceCodeProver(opts);

resObj = polyspace.CodeProverResults(resultsFolder);
resSummary = resObj.getSummary();
```

Use:

```
resultsFolder = fullfile(pwd,'results');

obj = polyspace.Project;

obj.Configuration.Sources = {fullfile(matlabroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
obj.Configuration.CodeProverVerification.MainGenerator = true;
obj.Configuration.ResultsDir = resultsFolder;

cpStatus = obj.run('codeProver');

resSummary = obj.Results.getSummary();
```

Compiler-Specific Keywords: Nonstandard compiler-specific keywords are only supported when you specify compiler

Summary: In R2017b, compiler-specific keywords are enabled only when you specify a supporting compiler. For instance, `far` is a keyword for certain compilers but not a keyword for others.

Benefits: When configuring your Polyspace project, it is sufficient to specify your compiler. Previously, certain keywords were disabled irrespective of your compiler choice. If your compiler supported those keywords, you had to explicitly enable them.

Compatibility Considerations

In existing projects that use the compiler option `none` (now `generic`), you can see compilation errors. Previously, certain nonstandard keywords such as `data` were removed during preprocessing because they were not relevant for the analysis. This syntax did not cause compilation errors.

```
data int tab[10];
```

Now, the nonstandard keywords are recognized based only on your choice of compiler. If you use a generic compiler, the analysis does not recognize the nonstandard keywords as keywords and does not remove them during preprocessing. For instance, the preceding syntax causes compilation errors. For workarounds, see Errors Related to Generic Compiler.

POSIX and BSD Standards: Use functions from these standards without additional setup

Summary: In R2017b, you can run analysis on code containing POSIX or BSD-specific functions without additional setup, for instance, defining macros such as `_POSIX_SOURCE`. As an example, you can analyze code that uses functions from `unistd.h` out of the box. You do not have to specify the location of `unistd.h` or perform additional configuration.

Benefits: You can quickly run analysis on code that uses functions specific to POSIX or BSD. If you do not provide the headers, Polyspace uses its own implementation of the functions for analysis.

Changes in analysis options and binaries

In R2017b, the following options have been added, changed, or removed.

New Options

Option	Description
`OSEK multitasking configuration (-osek-multitasking)`	See OSEK Multitasking Support release note.
`-xml-annotations-description`	See Code Annotations release note.
Compiler options: • Management of size_t (`-size-t-type-is`) • Management of wchar_t (`-wchar-t-type-is`)	Replaces previous options related to `size_t` and `wchar_t`.

Updated Options

Option	Change
Compiler (`-compiler`)	• Option value `none` changed to `generic`. • New value `greenhills` added. See Green Hills Compiler Support. • Option value `iso` removed. Use `generic` instead. • Option values `visual`, `visual6`, `visual7.0`, `visual7.1`, `visual8` and `visual10` removed. Use `visual10.0` instead. • Option value `gnu` removed. Use `gnu3.4` instead.
Target processor type (`-target`)	Target `powerpc64` added for Diab compiler. See Diab Compiler (`-compiler diab`).
Options related to packing of data structures: • Ignore pragma pack directives (`-ignore-pragma-pack`) • Pack alignment value (`-pack-alignment-value`)	Available for all compilers.
Enum type definition (`-enum-type-definition`)	Option value `defined-by-standard` changed to `defined-by-compiler`.
`-asm-begin` and `-asm-end`	Available for all compilers.

Removed Options

Option	Status	More Information
Management of 'for loop' index scope (`-for-loop-index-scope`)	Warning	Your choice of compilers determines the specification of `for` loop index variables. If you specify an older version of the Microsoft Visual C++ compiler such as `visual6`, `visual7.0` or `visual7.1`, the analysis considers that a `for` loop index is visible outside the loop. Otherwise, the analysis considers that the index is visible only inside the `for` loop.
Set size_t to unsigned long (`-size-t-is-unsigned-long`)	Warning	Use the option Management of size_t (`-size-t-type-is`).
`-wchar-t-is-unsigned-long` and `-wchar-t-is`	Warning `-wchar-t-is` has been removed from the user interface only.	Use the option Management of wchar_t (`-wchar-t-type-is`).
`-static-headers-object`	Warning	The permissive linking introduced by `-static-headers-object` now happens by default. The option is not required.

Compatibility Considerations

If you use scripts that contain the removed or updated options, update your scripts accordingly. In the Polyspace user interface, if an option is replaced by another option, the replacement occurs automatically in your configuration.

Verification Results

Stack Size Computation: Determine maximum stack usage by a C program and individual functions

Summary: In R2017b, the analysis computes the stack usage by each function in your program and the entire program. The maximum stack usage by a function is the total size of all local variables in the function plus the maximum stack usage by the function callees.

For more information, see:

- Maximum Stack Usage and Minimum Stack Usage
- Program Maximum Stack Usage and Program Minimum Stack Usage

See also Determination of Program Stack Usage.

Benefits: You can determine if the stack requirements of your program exceed the available size on the call stack. If the stack requirements exceed the available stack size, you can determine which variable or function is responsible and increase the available stack size or reduce the stack requirements.

MISRA C:2012 Directive 1.1: Detect instances of implementation-specific behavior in your code

Summary: In R2017b, you can detect possible violations of MISRA C:2012 Directive 1.1. The directive requires that you understand and document any implementation-defined behavior that affects the program output. See MISRA C:2012 Dir 1.1.

Benefits: The analysis detects constructs that can have implementation-defined behavior. If you have such constructs in your code, you can find how your compiler implements them. Once you understand and document all implementation-defined behavior, you can be assured that all output of your program is intentional and not produced by chance.

CERT C Support: Identify CERT C violations using run-time error checks

Summary: In R2017b, CERT C rules and recommendations are mapped to Code Prover run-time checks. If you run a Code Prover analysis, you can identify CERT C violations by using the mapping.

..		Check	CERT ID
●	*	Out of bounds array index	DCL38-C ARR30-C STR31-C STR32-C MSC15-C
●	*	Illegally dereferenced pointer	DCL38-C EXP08-C EXP34-C EXP36-C EXP39-C ARR30-C ARR37-C MEM10-C MEM35-C MSC15-C
●	*	Non-terminating call	
●	*	Non-terminating loop	MSC21-C
●	*	Invalid use of standard library routine	FLP32-C STR03-C STR07-C STR31-C STR32-C
X	*	Unreachable code	MSC07-C MSC12-C
X	*	Unreachable code	MSC07-C MSC12-C
X	*	Unreachable code	MSC07-C MSC12-C
X	*	Unreachable code	MSC07-C MSC12-C
X	*	Unreachable code	MSC07-C MSC12-C
X	*	Unreachable code	MSC07-C MSC12-C

Benefits: You can comply with the CERT C standard with Code Prover. Use a combination of run-time checks and MISRA C:2012 checkers. See:

- Check C/C++ Code for Security Standards
- CERT C Coding Standard and Polyspace Results

Overlapping Memory Detection: Find cases where source and destination arguments of memcpy overlap

Summary: In R2017b, Code Prover can detect `memcpy` usage where the source and destination memory regions overlap.

For instance, in this example, Code Prover shows a red **Invalid use of standard library routine** check on the use of `memcpy`.

```
#include <string.h>

int main() {
  char arr[4];
  memcpy (arr, arr + 3, sizeof(int));
}
```

Benefits: According to the Standard, overlap in source and destination arguments of `memcpy` lead to undefined behavior. Using Code Prover, you can detect these cases.

Changes to coding rule checking

Updated Specifications

In R2017b, the following changes have been made in checking of previously supported MISRA C and MISRA C++ rules.

Rule	Description	Improvement
MISRA C: 2004 Rule 17.4 and MISRA C++ Rule 5-0-15	Array indexing shall be the only allowed form of pointer arithmetic.	The rule checker flags array indexing on nonarray pointers. Previously, the checker flagged only explicit pointer arithmetic on pointers.

Rule	Description	Improvement

MISRA C:2004 Rule 8.9, MISRA C:2012 Rule 8.6 and MISRA C++ Rule 3-2-4	An identifier with external linkage shall have exactly one external definition.	The rule checkers flag multiple definitions only if the definitions occur in different files. The checkers do not consider tentative definitions as definitions.
		
		For instance, this code does not violate the rule:
		
		`int val;`
		`int val=1;`

MISRA C:2004 Rule 20.3 and MISRA C:2012 Directive 4.11	The validity of values passed to library functions shall be checked.	The rule checker uses more precise Code Prover analysis to determine if the input to a standard library function is within the allowed domain.

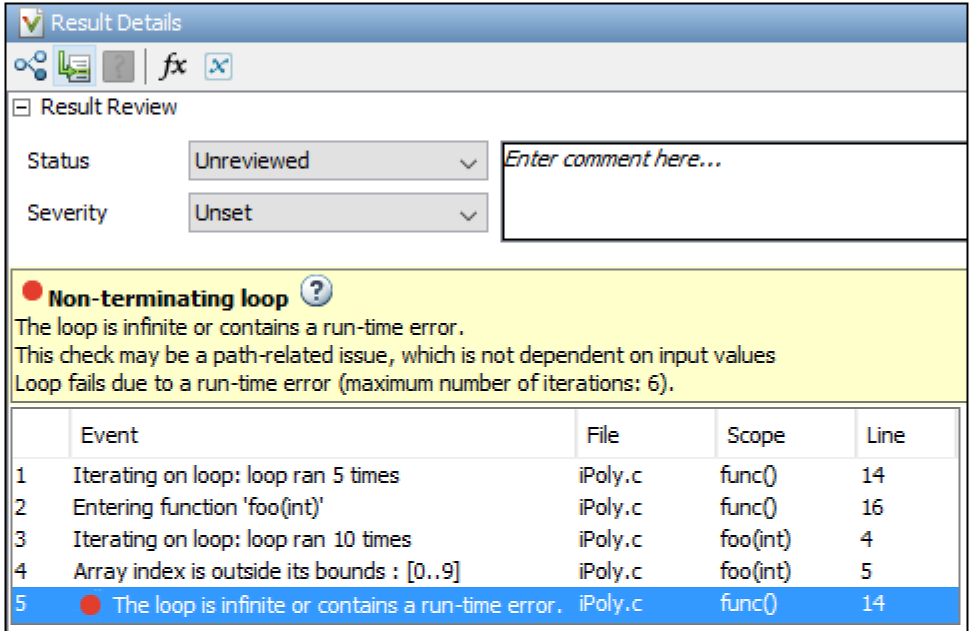
Reviewing Results

Run-Time Error Cause: Navigate to and view the cause of red nonterminating loops or function calls

Summary: In R2017b, you can determine the cause of a nonterminating loop over a few iterations, or a nonterminating function call, if it is due to a run-time error. To navigate to the cause, right-click the result, and select **Go to Cause**.

```
1   int a[10];
2
3   void foo(int x){
4       for (int i=0; i<=x+5; i++){
5       a[i]=i;
6       }
7   }
8
9   void func(){
10
11
12    int x, i;
13    x = 0;
14    for (i = 0; i <= 10; i++) {
15       [i+1]=0;
16      foo(i);
17
18    }
19  }
20
```

Benefits: You can view the sequence of events leading to the error, including the number of successful iterations, in the **Results Details** pane.

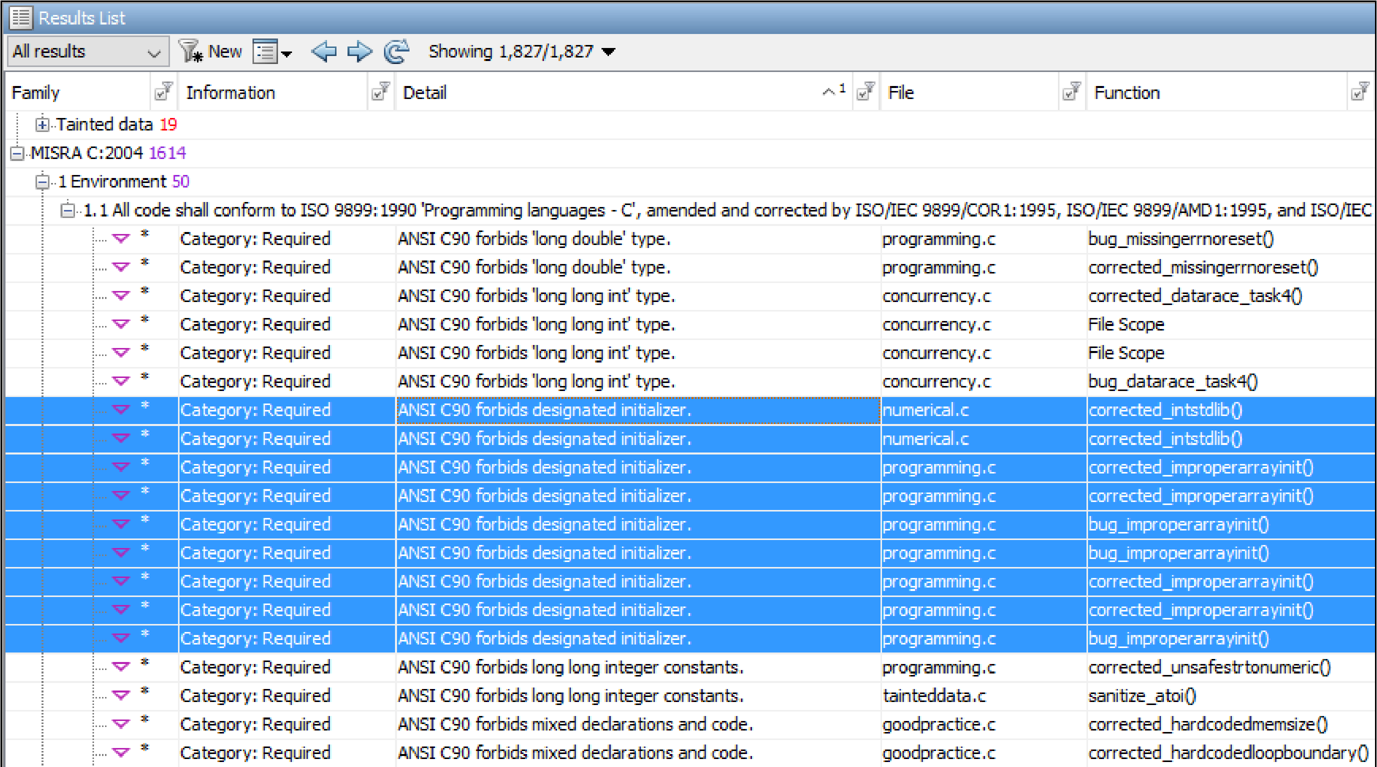


See also Identify Loop Operation with Run-Time Error.

Results Review Workflow: Sort and filter results by subtype

Summary: In R2017b, you can group your results by subtype through the new **Detail** column in the **Results list** pane. This column shows the first line from the **Results Details** pane, which has additional information about a result.

For instance, multiple issues can trigger the same coding rule violation. The **Detail** column shows the specific issue that triggered the rule violation.



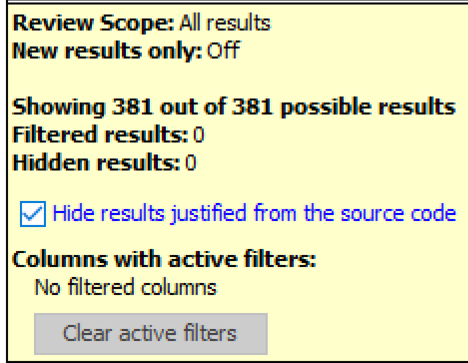
Benefits: You can easily mass-edit statuses or comments for results of the same subtype. In the **Results List** pane, group results by family, then within a result family use the **Detail** column to sort and select a subset.

Result Review Workflow: Hide results that you reviewed once and justified through source code annotations

Summary: In R2017b, if you justify a result through source code annotations, subsequent analyses do not redisplay result again. Although the result still appears in your source code, it does not appear in your results list.

```
static int get_oil_pressure(void)
{
    volatile int vol_i;
    int i;
    i = vol_i;  /* polyspace RTE:NIVL */
    assert(i > 0);
    return i;
}
```

If you want to revisit those justified results, you can make them visible in one-click.



Benefits: When you decide not to fix a finding, you can justify it through source code annotations. That finding does not clutter your subsequent analysis results.

Suppose the analysis flags an error-handling statement as unreachable code. You do not want to remove the statement because future code can trigger the error and make the error-handling necessary. You can justify the unreachable code and choose not to see it again.

Additional Considerations

- *How can I use source code annotations to justify a result?*

  You can directly type source code annotations in the correct format. See Justify Results Through Code Annotations.

  Alternatively, you can copy annotations from information in the user interface.

  - In Eclipse, right-click the result to insert a justification directly in the source code.
  - In Eclipse and the Polyspace user interface, assign one of the statuses `Justified`, `No action planned`, or `Not a defect` to a result. Right-click the result to copy your justification and paste it in a source code editor. See Justify Results Through Code Annotations.
- *Will the hidden results still appear in the report?*

  The hidden results still appear in the report. The results are hidden from view to save review effort. The reports are meant for complete documentation of your results. You cannot hide analysis results from the reports.

Code Annotations: Justify results or define your own format with a new annotation format

Summary: In R2017b, you can justify your results with the new Polyspace annotation syntax, or by using your own custom format. Polyspace also interprets existing code annotations that use a different syntax.

Benefits:

- *Easier results review:* With the new annotation format, you can provide a justification for multiple types of results on the same line. Previously, you had to enter the justification for different types of results, such as defects and coding rules violations, on different lines.

- *Custom annotation format:* You can use an XML file to define any annotation format and map it to the Polyspace syntax. When you analyze your code, Polyspace can interpret the annotations regardless of the format.

Additional Considerations:

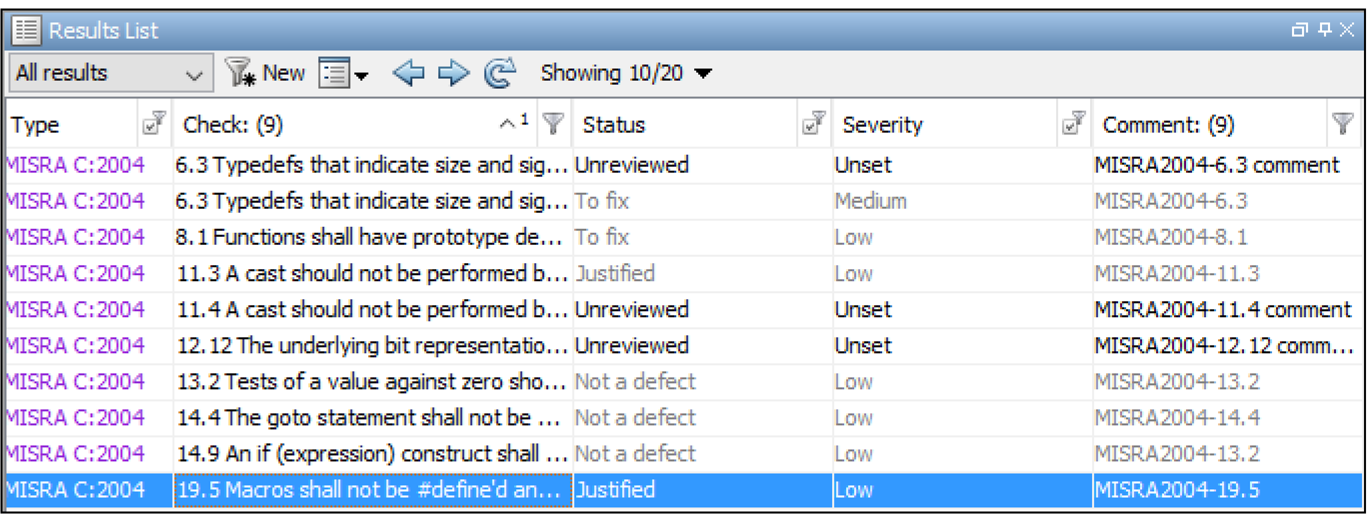
If you use the new annotation format and place your annotation on the line above the result you annotate, the annotation is ignored.

To apply the annotation to the line of code below, add `+1` after the `polyspace` keyword.

Polyspace still supports annotations that use the old syntax.

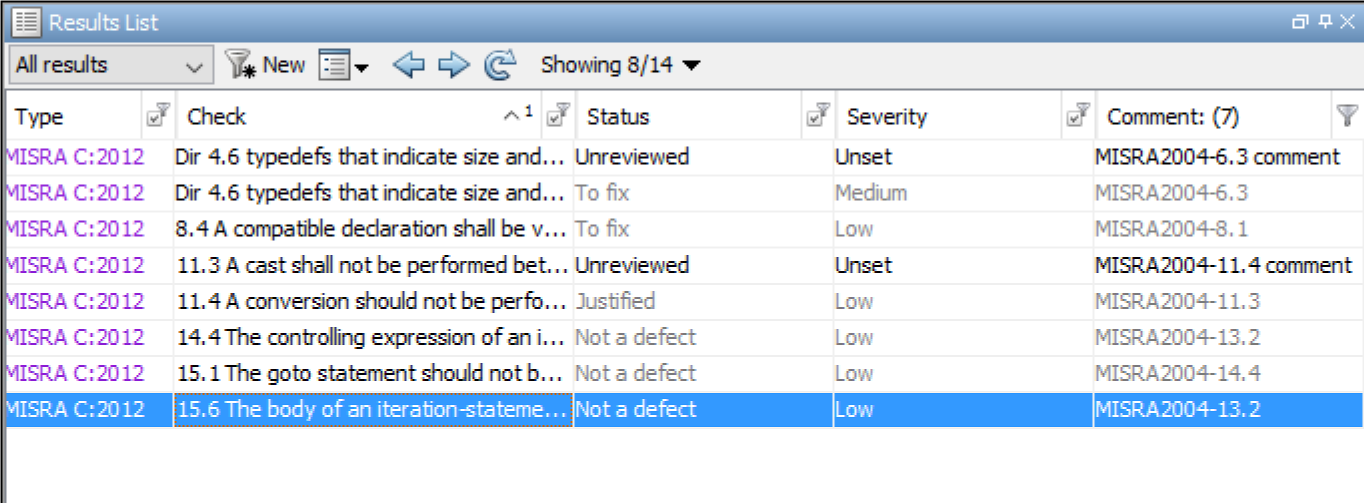
MISRA Comments and Code Annotations: Import your existing MISRA C:2004 justifications to MISRA C:2012 results

Summary: In R2017b, when you check your code against MISRA C:2012 rules, Polyspace imports existing justifications for MISRA C: 2004 violations.



Type	Check: (9)	Status	Severity	Comment: (9)
MISRA C:2004	6.3 Typedefs that indicate size and sig…	Unreviewed	Unset	MISRA2004-6.3 comment
MISRA C:2004	6.3 Typedefs that indicate size and sig…	To fix	Medium	MISRA2004-6.3
MISRA C:2004	8.1 Functions shall have prototype de…	To fix	Low	MISRA2004-8.1
MISRA C:2004	11.3 A cast should not be performed b…	Justified	Low	MISRA2004-11.3
MISRA C:2004	11.4 A cast should not be performed b…	Unreviewed	Unset	MISRA2004-11.4 comment
MISRA C:2004	12.12 The underlying bit representatio…	Unreviewed	Unset	MISRA2004-12.12 comm…
MISRA C:2004	13.2 Tests of a value against zero sho…	Not a defect	Low	MISRA2004-13.2
MISRA C:2004	14.4 The goto statement shall not be …	Not a defect	Low	MISRA2004-14.4
MISRA C:2004	14.9 An if (expression) construct shall …	Not a defect	Low	MISRA2004-13.2
MISRA C:2004	19.5 Macros shall not be #define'd an…	Justified	Low	MISRA2004-19.5

The analysis maps these justifications to the corresponding MISRA C: 2012 rules, if they exist.


Results List

All results | New | Showing 8/14

Type	Check	Status	Severity	Comment: (7)
MISRA C:2012	Dir 4.6 typedefs that indicate size and…	Unreviewed	Unset	MISRA2004-6.3 comment
MISRA C:2012	Dir 4.6 typedefs that indicate size and…	To fix	Medium	MISRA2004-6.3
MISRA C:2012	8.4 A compatible declaration shall be v…	To fix	Low	MISRA2004-8.1
MISRA C:2012	11.3 A cast shall not be performed bet…	Unreviewed	Unset	MISRA2004-11.4 comment
MISRA C:2012	11.4 A conversion should not be perfo…	Justified	Low	MISRA2004-11.3
MISRA C:2012	14.4 The controlling expression of an i…	Not a defect	Low	MISRA2004-13.2
MISRA C:2012	15.1 The goto statement should not b…	Not a defect	Low	MISRA2004-14.4
MISRA C:2012	15.6 The body of an iteration-stateme…	Not a defect	Low	MISRA2004-13.2

For more information, see Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results.

Benefits: You can transition from MISRA C:2004 to MISRA C:2012 compliance. If you have already justified a coding rule violation for MISRA C: 2004, you do not need to review the same result for the corresponding MISRA C:2012 rule.

Variable Relationships in Tooltips: Check if variables in operation are related from previous operation

Summary: In R2017b, you can determine if the variables in any operation are related from some previous operation.

For instance, if you want to know if the variables `var1` and `var2` in the operation `return(var1 – var2)` are related, you can insert a pragma before the line and rerun the analysis:

```
#pragma Inspection_Point var1 var2
```

In the results, you see a tooltip on `var2` in the pragma, which shows the relation between them, if one exists.

```
#pragma Inspection_Point wheel_speed_old wheel_speed
    temp = wheel_speed - wheel_speed_old;

    if (temp < 0)
      out = 1;
    else
      out = 0;

    wheel_speed_old = wheel_speed;
  }

}
```

Inspection point on external variable 'wheel_speed' (int 32): [0 .. 65000]

Relation(s): wheel_speed_old<=wheel_speed

Press 'F2' for focus

Benefits: You can use the pragmas as an additional tool for diagnosing results. At any point in your code, you can tell if certain variables are related to each other. You do not have to manually inspect your code to find if the variables have been previously related.

See Find Relations Between Variables in Code.

Result Status: Assign statuses that directly correspond to stages of development workflow

Summary: In R2017b, you can assign these statuses to a result. Each status corresponds to a stage in your code analysis workflow.

- `Unreviewed` (default status)
- `To investigate`
- `To fix`
- `Justified`
- `No action planned`
- `Not a defect`
- `Other`

Benefits: You can follow your review progress more easily.

Additional Considerations

- *How can I use the statuses to follow my review progress?*

  You can follow your progress in the Polyspace user interface or the Polyspace Metrics web interface.

  - Polyspace user interface: You can filter all results that have a certain status.
  - Polyspace Metrics: You can see the percentage of results reviewed and justified. If you assign a status other than `Unreviewed` to a result, the software considers the result as reviewed. If you assign one of these statuses, the software considers the result as justified: `Justified`, `No action planned`, or `Not a defect`.
- *Can I create my own status?*

  You can still create custom statuses. Select **Tools** > **Preferences** and create your own statuses on the **Review Statuses** tab.

Compatibility Considerations

If you open results from a previous release, the statuses are updated to the new release. The updates are:

- `Fix` or `Investigate` → `To fix` or `To investigate`
- `Improve` → `To fix`
- `Undecided` → `Unreviewed`.

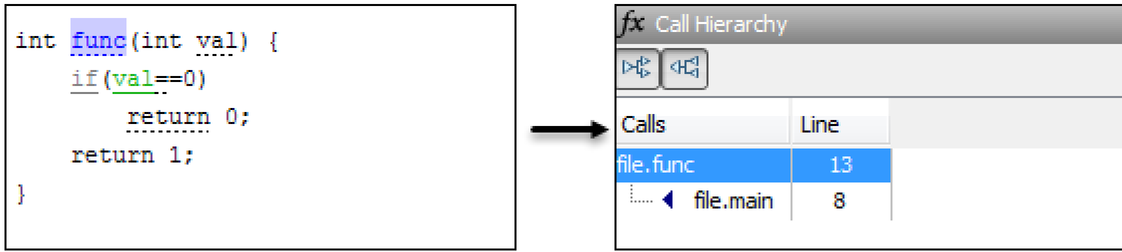
If you open results from a previous release, the severity `Not a defect` is updated to `Unset`.

If your source code annotations use statuses from a previous release, the software reads your annotations using the updates. The software does not change the annotations themselves.

Function Call Hierarchy: View and navigate to function callers and callees by clicking function name

Summary: In R2017b, you can click function names in your source code to see callers and callees of the function. You can then click a caller or callee name to go to their definitions in the source code. The **Call Hierarchy** pane shows the callers and callees.

When a function is defined, the source code shows the function name in blue. To see callers and callees on the **Call Hierarchy** pane, click the function name. For details, see Call Hierarchy.



Benefits: Previously, the **Call Hierarchy** pane was updated only when you clicked on a run-time check. You can now navigate the function call hierarchy more naturally by using function names in your source code.

Additional Considerations

Can I also click function calls to see the callers and callees?

When a function is called, the function call sometimes shows a run-time check color. If the function does not have a run-time check color (see `func2` below), click the function name to update the **Call Hierarchy** pane.

```
int main() {
    int val = INIT_VAL;
    int checkSuccess = func (val);
    if (checkSuccess == 0)
        func2();
}
```

Call Hierarchy

Calls	Line
file.func2	19
file.main	10

If the function has a run-time check color (see `func` above), right-click the function and select **Go To Definition**. The **Call Hierarchy** pane is updated to show the callers and callees.

R2017a

Version: 9.7

New Features

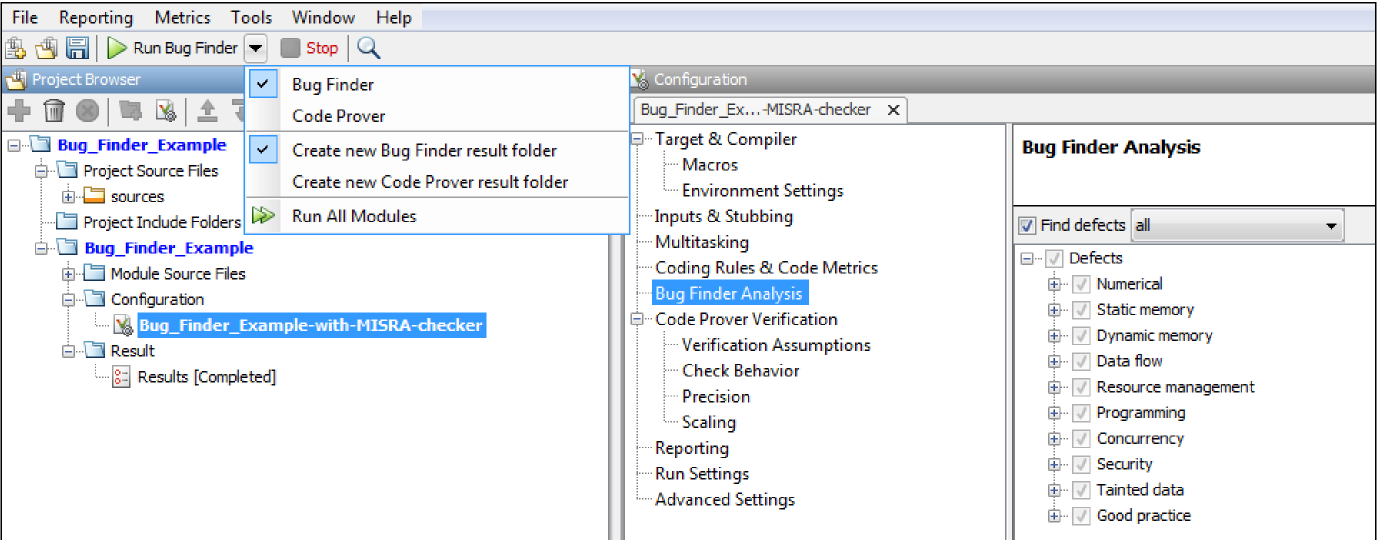
Bug Fixes

Compatibility Considerations

Verification Setup

Unified User Interface: Create and maintain a single Polyspace project for Bug Finder and Code Prover analysis

Summary: In R2017a, you can run Bug Finder and Code Prover analysis on the same Polyspace project in the same user interface.



Benefits:

- *Single entry point for two products*: You launch the Polyspace user interface only once from one icon on your desktop.
- *Easier switching between products*: After you run a Bug Finder analysis, you can switch to the more rigorous Code Prover analysis in one click.
- *One project, one configuration*: Add source files and specify your analysis options only once. After you set up your project, you can switch between the products without having to reconfigure.

Additional Considerations:

- *What if I only want to run a Bug Finder analysis?*

  You have to set the options that apply to a Bug Finder analysis. Most options are common between Bug Finder and Code Prover. So, you still have the benefit that most of your options will be set if you ever switch to Code Prover.

  The options specific to Bug Finder appear in the **Bug Finder Analysis** node, and the ones specific to Code Prover in the **Code Prover Verification** node and the nodes underneath.
- *If I run analysis in the two products, will the two sets of results appear together?*

  Yes, but not in the same view. The two sets of results appear under the same project, both in the user interface and in the physical folder locations.

  - In the user interface, in the **Project Browser**, the Bug Finder results appear with the icon and the Code Prover results appear with the icon.

- In your file explorer, you find the result folders for both analysis under one project folder.

  However, after you run the two analyses, you have to open the two sets of analysis results separately to review them. In the user interface, double-click one of the two result icons to open the results corresponding to that product.

- *Besides analysis options, are there other changes from pre-R2017a that I should be aware of?*

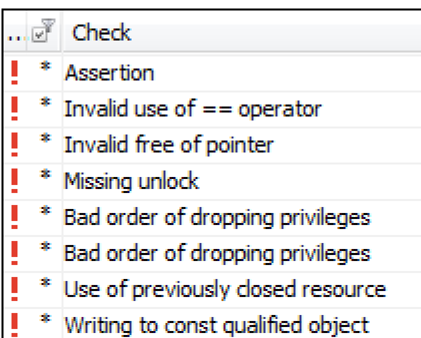
  If you were previously using only one of the two products, you will now notice the following differences.

  Bug Finder User:

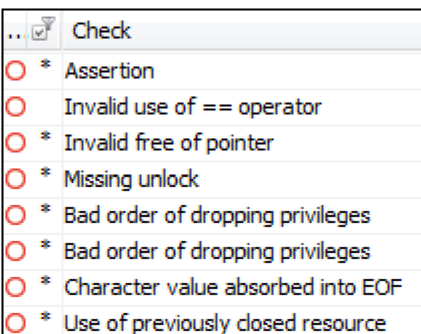
  - You can now create multiple modules in your Polyspace project to analyze separate components of your source code.

    When you create a project and add your source files, they are automatically added to the first module. If you add source files later, you have to select them and using the right-click option **Copy to Module_*n***, copy them to the module that you want.
  - You can now choose to create a new result folder for a second analysis on the same module. Use the option **Create new Bug Finder result folder** from the **Run** button dropdown. Prior to R2017a, there was one result folder for Bug Finder. If you ran a second analysis, it overwrote the previous results. Note that the overwriting is still *the default behavior*.
  - A new icon is used to denote defects.

    Before R2017a:



    R2017a:



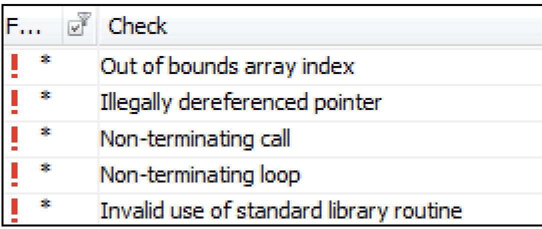
  Code Prover User:

- If you run a second analysis on the same module, by default, it overwrites the previous results. Prior to R2017a, a new result folder was created by default every time you ran an analysis.

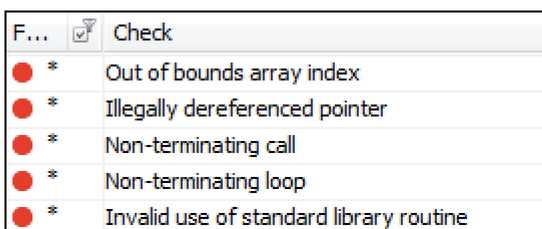
  You can change this default behavior and create a new result folder for the second analysis. Use the option **Create new Code Prover result folder** from the **Run** button dropdown.
- If some of your files do not compile, the analysis continues with the remaining files. If a file with compilation errors contains a function definition, the analysis considers the function as undefined and uses a function stub instead. You can see which files did not compile on the **Output Summary** pane and also in the report generated from the verification results.

  Previously, the default analysis required that all of your files must compile. To revert to this default behavior, use the option Stop analysis if a file does not compile (`-stop-if-compile-error`).
- A new icon is used to denote definite run-time errors or red checks.

  Before R2017a:

  R2017a:

- *I use DOS/UNIX®/MATLAB scripts to launch the analysis. How does this change affect me?*

  The change does not affect you directly. For instance, you still use two separate commands `polyspace-bug-finder-nodesktop` and `polyspace-code-prover-nodesktop` to run analysis from the DOS/UNIX command line. However, if you specify your options in a Polyspace project in the user interface and then create a script from the project, you have to specify your options only once for both products.

  Once you specify your options in the Polyspace project, you can easily create a script for the individual products. For instance, to create a Windows batch file that runs a Code Prover analysis, run the command:

  ```
  polyspace -generate-launching-script-for myproject.psprj
  ```

  To create a Windows batch file that runs a Bug Finder analysis, run the command:

  ```
  polyspace -bug-finder -generate-launching-script-for myproject.psprj
  ```

Improved Speed and Precision: Run analysis faster and receive fewer orange checks as compared to previous releases

Summary: In R2017a, Polyspace analysis uses many improvements that increase precision and reduce analysis time significantly, sometimes by as much as 30%. For instance, in presence of arrays and nested structures, the analysis is faster and more precise.

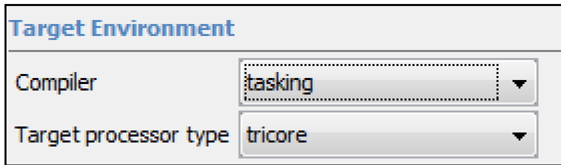
Benefits:

- *Less wait time*: You are likely to spend less time waiting for the analysis to complete.
- *Less review time*: For most applications, you are likely to have fewer orange checks and spend less time manually reviewing them.

TASKING Compiler Support: Set up Polyspace analysis easily for code compiled with Altium TASKING compiler

Summary: If you build your source code with the Altium® TASKING compiler, in R2017a, you can specify the compiler name for your Polyspace analysis. The analysis can interpret macros that are implicitly defined by the compiler and compiler-specific language extensions such as keywords and pragmas.

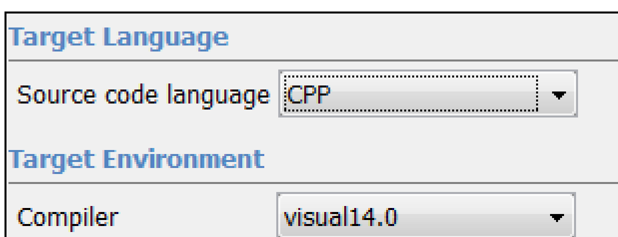
You can specify the following target processors directly: `tricore`, `c166`, `rh850` or `arm`. See TASKING Compiler (`-compiler tasking`).



Benefits: You can now set up a Polyspace project without knowing the internal workings of your TASKING compiler. If your code compiles with your compiler, it will compile with Polyspace in most cases without requiring additional setup. Previously, you had to explicitly define macros that were implicitly defined by the compiler and remove unknown language extensions from your preprocessed code.

Updated Visual C++ Support: Set up Polyspace analysis easily for code compiled with Microsoft Visual C++ 2015 compiler

Summary: If you build your source code with the Microsoft Visual C++ 2015 compiler, in R2017a, you can specify the compiler name for your Polyspace analysis. The analysis can interpret macros that are implicitly defined by the compiler and compiler-specific language extensions such as keywords and pragmas.



For more information, see Compiler (`-compiler`).

Benefits:

- *Easier compilation*: You can now set up a Polyspace project without knowing the internal workings of your Microsoft Visual C++ 2015 compiler.
- *More precise analysis*: The analysis provides precise results when you use compiler-specific extensions.

Autodetection of Concurrency Primitives: Multitasking model detected from Windows or µC/OS II multithreading functions

Summary: In R2017a, if you use the Windows or µC/OS II functions for multitasking, the Polyspace analysis can interpret them semantically.

Polyspace interprets the following functions:

Family	Thread Creation	Critical Section Begins	Critical Section Ends
Windows	`CreateThread`	`EnterCriticalSection`	`LeaveCriticalSection`
µC/OS II	`OSTaskCreate`	`OSMutexPend`	`OSMutexPost`

Benefits: You do not have to adapt your code or specify your multitasking model manually through analysis options. The analysis determines your multitasking model from the functions in your code and checks if shared variables are sufficiently protected.

Manual Multitasking Setup: Functions beginning and ending critical sections do not need to be defined

Summary: In R2017a, if you specify that certain functions begin and end critical sections, you do not have to provide their definitions to Polyspace.

Benefits: If you use functions provided by your operating system whose definitions are not readily accessible, you do not have to provide the definitions.

Manual Multitasking Setup: main Function Not Required

Summary: In R2017a, you can run verification on multitasking applications that do not have a `main` function.

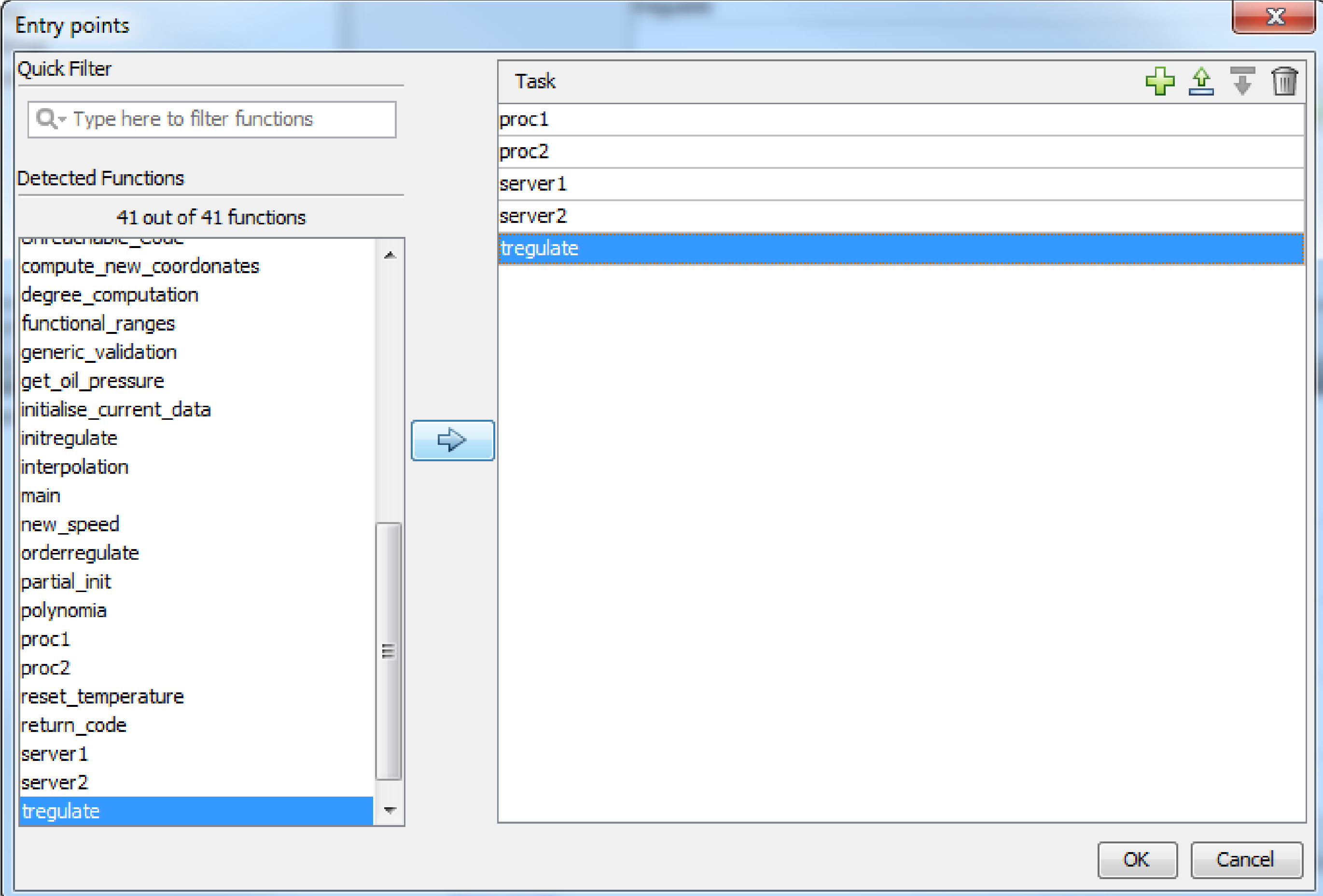
Benefits: Previously, Code Prover analysis required a `main` function for multitasking. If your code did not have a `main` function, you had to add a `main` to your source code or preprocessed code just for the Code Prover analysis.

The software now adds an empty `main` function for you. If your code has a `main` function, the software continues to use that `main` function for analysis.

Specifying Function Names for Options: Choose from prepopulated list in user interface instead of entering manually

Summary: In R2017a, for options that take function names, you can choose the names from a list.

For instance, to specify which functions act as entry points to your multitasking application, you can choose the names from a list as follows:



Benefits: You do not have to enter the names manually. If the functions list is long, you can start typing the function name to reduce the list.

Polyspace API in MATLAB: Create MATLAB objects from Polyspace projects to run analysis

Summary: In R2017a, you can create a MATLAB object from a Polyspace project (`.psrpj` file). For instance, if you have a file `myProject.psprj` in the current working folder, enter:

```
opts = polyspace.loadProject('myProject.psprj')
```

Use the object `opts` in MATLAB scripts to run a Polyspace analysis:

```
polyspaceCodeProver(opts);
```

Benefits:

You can now consider the following workflows:

- *Set options in GUI and script analysis*: Use the Polyspace user interface to specify options in your Polyspace project, or adjust options based on results from a trial run. After the options are stable, create a MATLAB object `opts` from the project and store it in a MAT-file. As you move along in your development cycle, simply load `opts` from your MAT-file, update `opts.Sources` to add new source files, update other properties as required, and use `opts` to run analysis. For all properties of the object, see `polyspace.Options`.
- *Create project from your build command and script analysis*: Use the function `polyspaceConfigure` to create a `.psrpj` file from your build command (makefile). Create a MATLAB object from that file to run analysis. In this way, you can use a MATLAB script for the entire Polyspace analysis workflow beginning from your makefile.

Additional Considerations:

- *A single Polyspace project works for both Bug Finder and Code Prover. Can I likewise use the object to run both a Bug Finder and Code Prover analysis?*

  Yes, once you create the MATLAB object from a Polyspace project, you can use it with both functions `polyspaceBugFinder` and `polyspaceCodeProver`.
- *Can I create an object from a project that I have from a pre-R2017a version of Polyspace?*

  Yes, you can.

Improved support for user implementations of standard library functions

Summary: If the arguments or return value of a standard library function have data types that are defined in your header files, Polyspace compilation now uses your type definitions. Polyspace compilation uses its own implementation of standard library functions and previously, looked for specific type definitions in specific header files. Compilation errors occurred if the definitions could not be found in those specific header files.

For instance, the `fopen` function returns a `FILE*` pointer.

```
FILE * fopen ( const char * filename, const char * mode );
```

Suppose, you define `FILE` using a `typedef` in an included header file that is not `stdio.h`, as follows:

```
typedef int FILE[4];
```

Polyspace compilation uses this definition of `FILE`. Previously, the compilation looked for the definition of `FILE` only in `stdio.h`.

Benefits:

- *Compilation errors avoided*: You see fewer compilation errors due to your implementations of standard library functions.
- *Better analysis*: The analysis uses data types for your standard library functions the way you have defined them. Therefore, it interprets your code more accurately.

Improvement in automatic project creation from build systems

Summary: In R2017a, by default, automatic project creation will throw an error if a project with the same name exists in the output folder.

If you encounter an error, avoid the name conflict: change the project name, output folder, or remove your older project.

Benefits: You cannot overwrite existing projects by accident. If you use scripts that are intended to overwrite existing projects, use the additional option `-allow-overwrite`.

Changes in analysis options and binaries

In R2017a, these options have been added, changed, or removed.

Updated Options

Option	Change	More Information
Report template	Renamed in user interface	New name: **Code Prover report** The command-line name is still `-report-template`.
Batch	Renamed in user interface	New name: **Run Code Prover analysis on a remote cluster** The option is now in the **Run Settings** node in your project configuration. The command-line name is still `-batch`.
Add to results repository	Renamed in user interface	New name: **Upload results to Polyspace Metrics** The option is now in the **Run Settings** node in your project configuration. The command-line name is still `-add-to-results-repository`.
Compiler (`-compiler`)	New value added	You can specify the following arguments: • `tasking` See TASKING Compiler Support on page 8-5. • `visual14.0` See Microsoft Visual C++ Support on page 8-5.
Infinities (`-check-infinite`)	Available in user interface	Previously, this advanced option was available only on the command line.
NaNs (`-check-nan`)	Available in user interface	Previously, this advanced option was available only on the command line.

Removed Options

Option	Status	More Information
Optimize large static initializers (-no-fold)	Removed	The benefits that came with `-no-fold` now appear by default, without the associated costs in precision. So the option is not required.
Continue with compile error (-continue-with-compile-error)	Removed	The option is enabled by default in Code Prover. In other words, if some files have compilation errors, by default, the analysis continues with the remaining files. If you want analysis to stop from even a single compilation error, use the option Stop analysis if a file does not compile (`-stop-if-compile-error`).
Green absolute address checks (-green-absolute-address-checks)	Removed	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)	Removed	Use the option Do not generate results for (`-do-not-generate-results-for`) to suppress results from headers and sources in certain files or folders.
Ignore float rounding (-ignore-float-rounding)	Removed	
`-retype-pointer`	Removed	
`-retype-int-pointer`	Removed	
`-lwtm`	Removed	
`-support-FX-option-results`	Removed	
No automatic stubbing (-no-automatic-stubbing)	Error	Option will be removed in a future release.
`-easy-setup-preprocess`	Error	Option will be removed in a future release.
`gui-api`	Error	Binary will be removed in a future release. Use `polyspace-comments-import` instead.
`polyspace-automatic-verification`	Error	Binary will be removed in a future release.
`polyspace-remote`	Error	Binary will be removed in a future release.
`polyspace-verifier`	Error	Binary will be removed in a future release.
`rte-kernel`	Error	Binary will be removed in a future release.
Dialect (-dialect)	Error	Option will be removed in a future release. Use Compiler (`-compiler`) instead.

Option	Status	More Information
Target operating system (-OS-target)	Error	Option will be removed in a future release. If you use this option in scripts, see the list below for replacements: • Linux: If you get compilation errors, use Compiler (-compiler) gnu*x.x*. Sometimes, you might also have to set Preprocessor definitions (-D) to linux, unix, or __linux__. • Visual: Use Compiler (-compiler) visual*x.x* • Vxworks: Use the VxWorks® configured template. For more information, see Create Project Using Configuration Template. • Solaris: Remove -OS-target. • no-predefined-OS: Remove -OS-target.
Import folder (-import-dir)	Warning	Option will be removed in a future release.

Compatibility Considerations

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

Changes in MATLAB options object

These classes will be removed in a future release.

- polyspace.CodeProverOptions: To customize Polyspace analysis of handwritten code, use polyspace.Options instead.
- polyspace.ModelLinkCodeProverOptions: To customize Polyspace analysis of generated code, use polyspace.ModelLinkOptions instead.

The properties and methods of the new classes are almost the same as the original classes. If optsOld is an object of the original class and optsNew is an object of the new class, the following properties have changed.

Reporting

Removed	Use instead
optsOld.Reporting. EnableReportGeneration	optsNew.MergedReporting. EnableReportGeneration
optsOld.Reporting.ReportTemplate	optsNew.MergedReporting. CodeProverReportTemplate
optsOld.Reporting. ReportOutputFormat	optsNew.MergedReporting. ReportOutputFormat

ComputingSettings

Removed	Use instead
optsOld.ComputingSettings.Batch	optsNew.MergedComputingSettings. BatchCodeProver
optsOld.ComputingSettings. AddToResultsRepository	optsNew.MergedComputingSettings. AddToResultsRepositoryCodeProver

Compatibility Considerations

Replace instances of the old class names in your MATLAB scripts with the new class names. Then, replace the properties accordingly.

Even if you continue to use the old class names, you must change the properties, as described above.

Change in temporary folder location

In R2017a, Polyspace looks for standard environment variables such as `TMPDIR` to store temporary files during an analysis. Previously, Polyspace used the folders `/tmp` or `C:\Temp` during analysis.

You can also store Polyspace temporary files in a folder different from the standard temporary folders. To learn how Polyspace determines the temporary folder location, see Storage of Temporary Files.

Compatibility Considerations

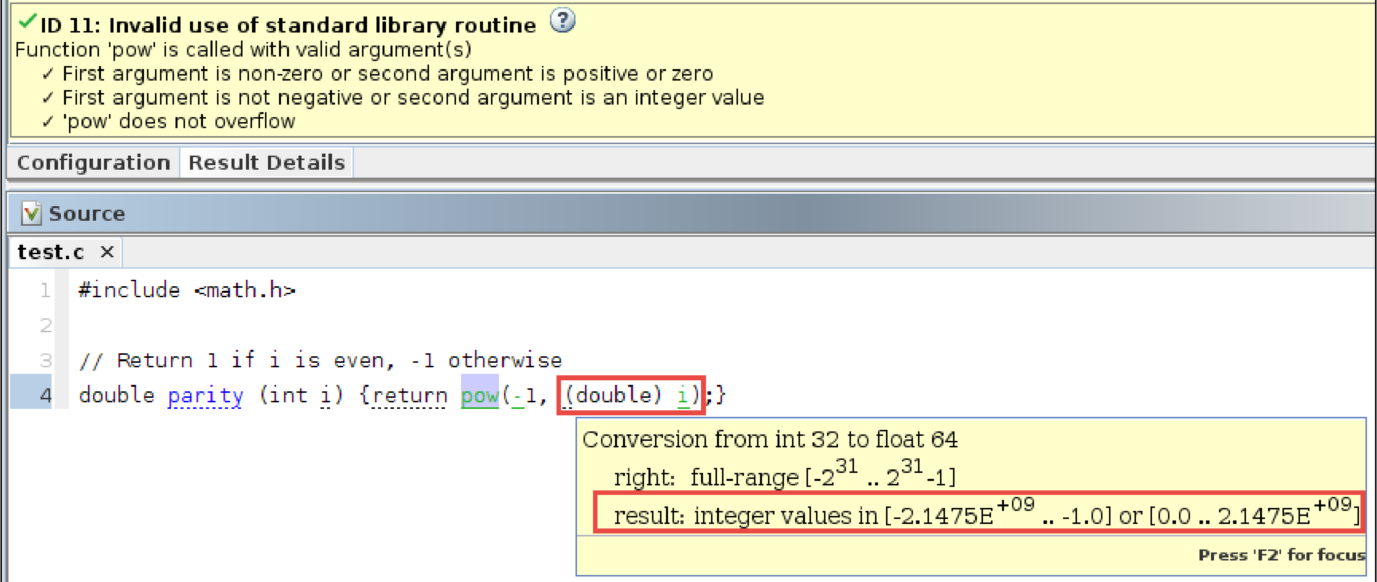
If your analysis seems slower than before, check if the new temporary folder is on a network drive. For faster analysis, use a folder on a local drive instead.

Verification Results

Integers in Floating Point: See improved analysis precision for floating point variables that always take integer values

Summary: In R2017a, the analysis can detect `float` or `double` variables that take integer values.

For instance, in the following code, despite the cast to `double`, the verification detects that `i` takes integer values.



Benefits:

- *Improved analysis precision*: The analysis uses more precise integer arithmetic for these variables.
- *Better understanding of results*: The range tooltip on these variables show that they take integer values only. You can use this information to interpret certain results.

New Code Metrics: See number of lines in header files and number of local variables per function

Summary: In R2017a, Polyspace can provide the following new code complexity metrics:

- Number of lines and number of lines without comments in header files
- Number of local non-static variables for every function and method
- Number of local static variables for every function and method

Benefits: You can determine the memory footprints of your code using these new metrics (along with other already existing metrics).

Checks Green by Definition: Distinguish operations that are safe by definition from operations that are proven safe

Summary: In R2017a, certain numerical run-time checks clearly indicate whether the check is green by definition.

The messages for such checks state that the operation is safe with respect to the run-time check, whatever the operand values. For instance, the `sqrt` or `cbrt` function cannot return subnormal values.

Benefits: If an operation is safe by definition, you do not need to protect against unsafe behavior. If an operation is safe only in the current context, you need to recheck the operation when reusing it in another context. Being able to identify operations that are safe by definition helps you determine if you need to protect against later unsafe behavior.

Additional considerations:

- *Can I tell by visual inspection that an operation is safe by definition?*

  Sometimes, you can. In other cases, rigorous mathematical calculations are required to prove that an operation is safe by definition. Polyspace verification shows you all such operations in green, providing you the assurance that your usage is safe with respect to the run-time check.
- *Which checks can be green by definition?*

  The following checks can be green by definition.

  - Subnormal float: Checks green by definition highlight operations that cannot return subnormal results, whatever the operand values.
  - Overflow: Checks green by definition highlight operations that cannot overflow, whatever the operand values.
  - Invalid operation on floats: Checks green by definition operations that cannot return NaN, whatever the operand values.

  The meaning of green by definition depends on your analysis mode. For instance, in the `warn-first` mode of the **Subnormal float** check, green means that an operation cannot return subnormal results *unless the operands themselves are subnormal*. Green by definition incorporates this change in meaning of green.

Function Pointer Signature Mismatch: View orange checks instead of red when the mismatch cannot be proven

Summary: In R2017a, the Correctness condition check on calls through function pointers aligns more closely with the general semantics of Code Prover checks. The check is red only if the verification proves that the function pointer does not point to a function with matching signature.

Benefits: You can follow the same review policy with **Correctness condition** checks as with other checks. Previously, the **Correctness condition** checks could be red even if the analysis did not prove a mismatch between the function pointer and the function that it points to. These red checks indicated that the verification cannot identify which function to call, because of imprecisions or lack of external information. In these cases, the checks are now orange.

Compatibility Considerations

You can see a change in the number of red and orange **Correctness condition** checks.

Structures with Volatile Fields: See improved analysis precision and apply constraints if necessary

Summary: In R2017a, the software analyzes `volatile`-qualified structure fields more precisely.

- The analysis can distinguish between volatile and nonvolatile fields. Previously, if one field of a structure was volatile, the analysis either considered all fields as volatile or ignored the `volatile` qualifier for all fields, depending on the option you select.

  For instance, in the following code, the software previously considered both `val1` and `val2` as volatile or not.

  ```
  typedef struct myStruct
  {
    volatile int val1;
    int val2;
  };
  ```

  If the analysis considers the `volatile` qualifier for structure fields, they can take any value allowed by their data type at any point in the code.
- You can specify permanent constraints on `volatile`-qualified structure fields to narrow down their assumed range. See Constraints.

  This improvement also applies to `volatile`-qualified arrays.

Benefits: You have to review fewer orange checks from imprecise analysis of structures with volatile fields.

Compatibility Considerations

Unless you use the default assumption to ignore the `volatile` qualifier on structure fields, you can see a reduction in the number of orange checks.

Changes to coding rule checking

In R2017a, the following changes have been made in checking of previously supported MISRA C rules.

Rule	Rule	Improvement

MISRA C: 2004 Rule 5.1	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.	The rule checker shows all identifiers that have the same first 31 characters as one rule violation. Previously, every pair of identifiers with same 31 characters was shown as a separate violation.
		
		For instance, in the following code snippet, the rule violation appears only once.
		
		`extern int engine_exhaust_gas_temperature_raw;`
		`static int engine_exhaust_gas_temperature_scaled;`
		`static int engine_exhaust_gas_temperature_cutoff;`
		
		Previously, the violation was shown three times.
		
		You have to review only one rule violation for every group of identifiers with the same 31 characters. You can still see all instances of conflicting identifier names in the event history of that rule violation.

`MISRA C:2012 Rule 8.5`	An external object or function shall be declared once in one and only one file.	The rule checker considers that variables or functions declared `extern` in a non-header file violates this rule.

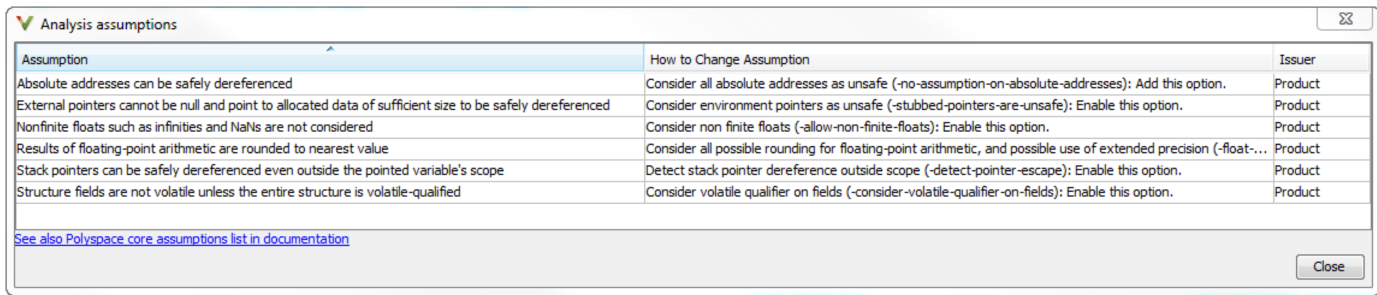
Reviewing Results

Easier Review: View verification assumptions, see unreachable and aliased function calls in call graph

Summary: In R2017a, you can review Polyspace Code Prover checks more easily using new features in the Polyspace user interface.

- *Verification assumptions*: You can see the assumptions that the software makes, collected in one place.

  If an assumption can be changed, the **Analysis assumptions** pane shows the assumption.

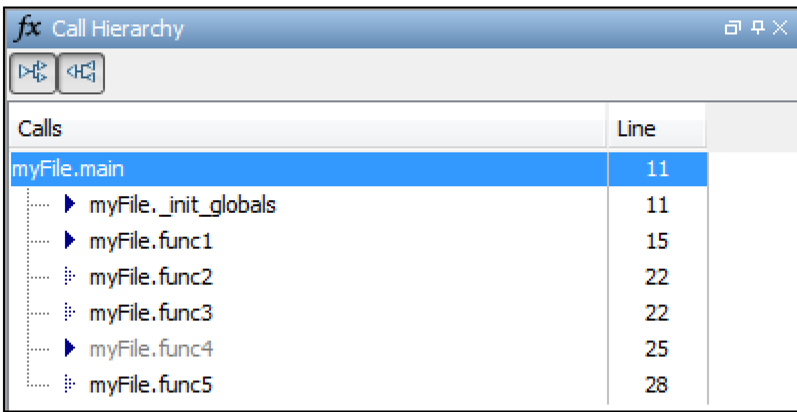


  The Polyspace documentation lists the core assumptions that you cannot change. See Polyspace Software Assumptions.

  You can also see the modifiable assumptions in reports generated using a Code Prover template.

- *Improved function call hierarchy*: The **Call Hierarchy** pane shows which function calls are unreachable (shown in gray) and which calls are made through function pointers (shown with the ⁞ icon).

  For instance, in the following figure, the function `main` calls `func1` directly, and calls `func2`, `func3` and `func5` indirectly via function pointers. The call to `func4` is unreachable.

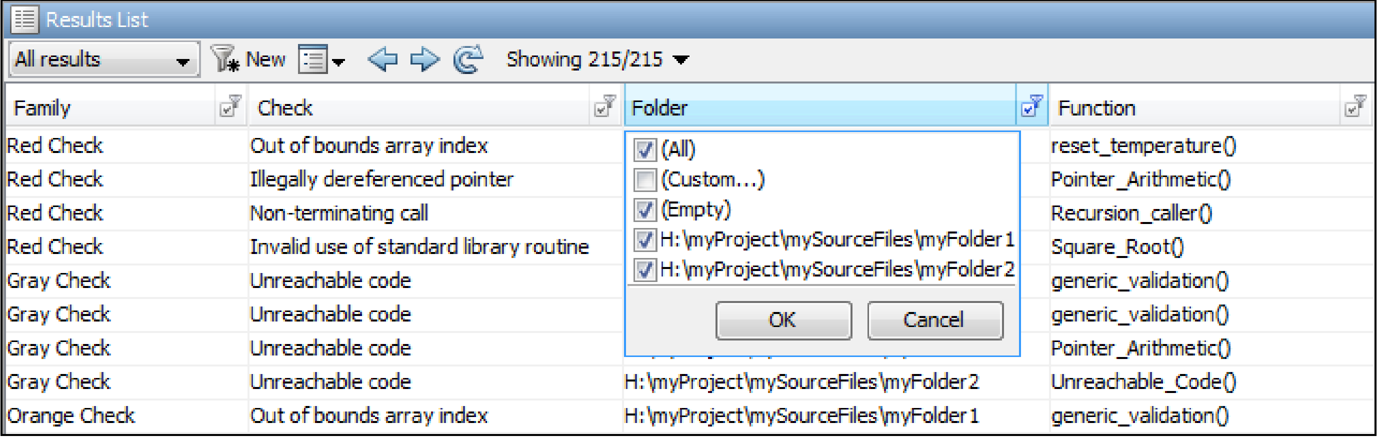


Benefits: The new features help you interpret analysis results more easily.

- *Verification assumptions*: To interpret certain analysis results, you can now browse through the list of analysis assumptions. If an analysis option is available to change the assumption, you can find the option more easily.
- *Improved function call hierarchy*: To interpret certain analysis results, you can now check quickly if an expected function call does not occur in practice.

Folder Names in Results: Filter or group analysis results by source folder names

Summary: In R2017a, the source folder name is shown in the list of analysis results.



Benefits: You can order your results by folders or filter results belonging to specific folders. Using custom filters, you can filter out the subfolders of a folder in one click.

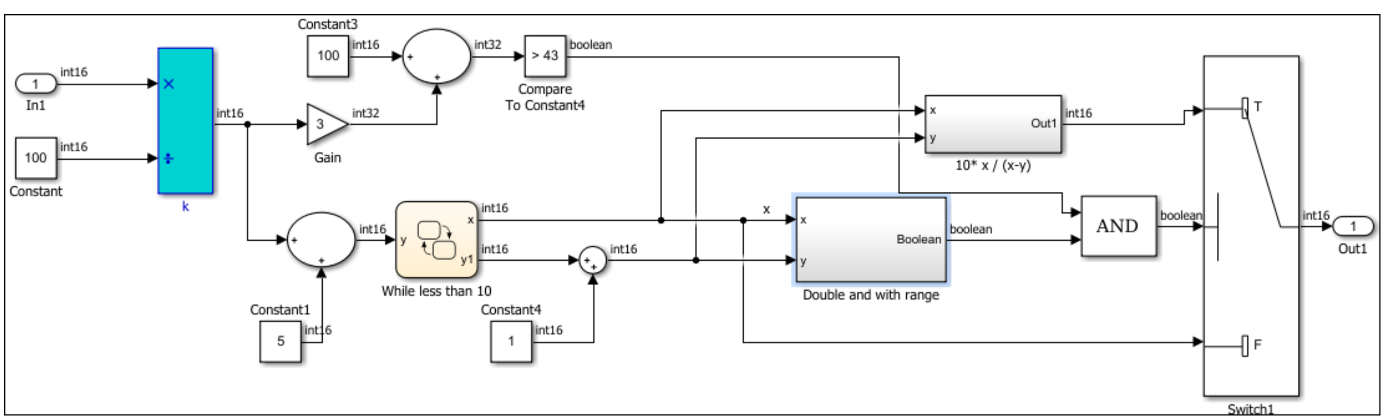
Code to Model Traceability: Switch easily between identifiers in generated code and corresponding blocks in model

Summary: In R2017a, you can trace an instance of a variable in generated code back to your model.

```
void WhereAreTheErrors(const int16_T *rtu_In1, int16_T *rty_Out1)
{
  int16_T rtb_k;
  int16_T rtb_x;
  int16_T rtb_y1;

  /* Product: '<Root>/k' incorporates:
   *  Constant: '<Root>/Constant'
   */
  rtb_k = (int16_T)(*rtu_In1 / 100);
  /*                                                        s:
   *
   *
   */
  /*
  /*
  /*
  /*
  rtb
  rtb
  whi
    /
    /
    r
```

Print Source Code: WhereAreTheErrors.c
Search For "rtb_k" in Current Source File   Ctrl+F
Search For "rtb_k" in All Source Files
Search For All References
Go To Definition
Go To Line   Ctrl+L
Go To Model
Open Editor
Highlights the corresponding block in Model
Add Pre-Justification To Clipboard
Expand All Macros
Collapse All Macros
Create Duplicate Code Window

The model shows the corresponding block highlighted in blue. If the block is in a subsystem, both the subsystem and the block are highlighted in blue.



Benefits:

- *More convenient navigation*: Previously, you traced back from code to model via links in code comments. You can now navigate from the code operations themselves.

- *More fine-grained navigation*: You can easily identify which block in your model leads to which operation in the generated code.

Polyspace API in MATLAB: Read Polyspace analysis results from MATLAB

Summary: You can read your Polyspace analysis results into a MATLAB table. For instance, if the folder `C:\MyResults` contains results of a Polyspace analysis, enter the following:

```
resObj = polyspace.CodeProverResults('C:\MyResults')
resSummary = getSummary(resObj)
resTable = getResults(resObj)
```

`resSummary` and `resTable` are two MATLAB tables containing summary and details of the Polyspace results.

See also `polyspace.CodeProverResults`.

Benefits: You can use the capabilities of MATLAB to obtain graphs and statistics about your Polyspace results.

R2016b

Version: 9.6

New Features

Bug Fixes

Compatibility Considerations

Verification Setup

Diab Compiler Support: Set up Polyspace verification easily for code compiled with Wind River Diab compiler

If you build your source code using the Wind River® Diab compiler, in R2016b, you can easily set up a Polyspace project to verify your code. After you specify the Diab compiler and your target processor, the verification:

- Implicitly defines macros that are defined for the Diab compiler. Previously, you defined the macros in your Polyspace project explicitly to avoid compilation errors.
- Understands language extensions such as keywords and pragmas that are specific to the Diab compiler. Previously, you removed unknown language extensions explicitly from the preprocessed code in your Polyspace project to avoid compilation errors.

You can now set up a Polyspace project manually without knowing the internal workings of your Diab compiler. Specify the Diab compiler and your target processor, and run verification without facing compilation errors. See Diab Compiler (-compiler diab).

The software supports version 5.9 and older versions of the Diab compiler.

Multitasking Code Verification Setup: Specify cyclic tasks and nonpreemptable interrupts directly as verification options

In R2016b, you can specify which entry points in your code represent cyclic tasks and nonpreemptable interrupts. Previously, to emulate the cyclic behavior of a task, you embedded instructions in a loop. To emulate a nonpreemptable interrupt, you specified temporally exclusive pairs where the interrupt was paired with all other interrupts.

For more information, see Cyclic tasks (-cyclic-tasks) and Interrupts (-interrupts).

Improved source and include folder management

Before R2016b, when you created a project, you added and removed source files and include folders individually. If you moved your source files or added new files to your programming project, you re-added the files into your Polyspace project.

Starting in R2016b, you create Polyspace projects with root source folders and include folders. The root folder location represents the top of the hierarchy for your source files. Polyspace shows all files relative to the root source locations. When you add a root source location, you can:

- See all source files under the root folder (and subfolders)
- Exclude files and subfolders in the hierarchy to change the active list of source files to analyze.
- Refresh the source file list to see new files or folders in the root source hierarchy.
- Modify the root source folder path.
- If you use a revision control system, change the root folder location to point to different versions of your source files.

For include folders, instead of adding individual folders, you add a root include folder location. Polyspace adds all include folders underneath the root include location that contains include files. You can refresh and modify the include folder path.

For more information, see Create Project.

Writable Examples: Modify example projects and restore original versions

The examples projects under **Help** > **Examples** are now easier to use. The first time that you open an example project, a writable version is saved in your `Polyspace_Workspace`. In the writable project, you can test configuration options, change sources, and rerun the example. If you want to refresh the example with a clean version, select **Help** > **Examples** > **Restore Default Examples**.

Run verification on .psprj file from the command line

If you already have a project created in the Polyspace Interface, you can now use that `.psprj` file to run your verification from a command line.

DOS or UNIX Command Line

Use the new option `polyspace-code-prover -generate-launching-script-for <PSPRJ FILE>` to generate the files to run the analysis from the command line. These files are generated:

- `source_command.txt` — List of source files in the project
- `options_command.txt` — List of analysis option settings
- `launchingCommand.sh` or `launchingCommand.bat` — Script that runs the verification using `options_command.txt`, `source_command.txt`. The script can also take additional analysis options as parameters.

For more information, see Create Command-Line Script from Project File.

MATLAB Command Prompt

At the MATLAB command prompt, you can now give a `.psprj` file as an argument to `polyspaceCodeProver`.

The syntax `polyspaceCodeProver(PSPRJ file,'nodesktop')` runs a verification on the project. If you have multiple modules or configurations, the active module and active configuration are verified.

Polyspace API in MATLAB: Configure and run Polyspace using MATLAB objects

In R2016b, Polyspace scripting from MATLAB is easier and more MATLAB-friendly. R2016b introduces a set of classes, methods, and function improvements to help you run Polyspace from MATLAB. For more information and examples, see the linked reference pages.

Classes

Name	Description
`polyspace.CodeProverOptions`	An options object with properties that map to the Polyspace environment configuration options. Use this object to customize analysis options and run analysis.
`polyspace.ModelLinkCodeProverOptions`	Another version of the `CodeProverOptions` object with properties specifically for model generated code. Use this object to customize analysis options and run analysis.
`polyspace.GenericTargetOptions`	A helper object for the `CodeProverOptions` classes. Use this object to customize a generic target.
`polyspace.CodingRulesOptions`	A helper object for the `CodeProverOptions` object. Use this object to customize the list of coding rules checked during the analysis.

Methods

Name	Description
`polyspace.Options.copyTo`	Copy settings between options objects. You can use this method to copy options from a `CodeProverOptions` object to a `BugFinderOptions` object and vice versa.
`polyspace.Options.generateProject`	Generate a `.psprj` file from an options object to open in the Polyspace interface.

Functions

Name	Description
`polyspaceCodeProver`	Run an analysis using `CodeProverOptions` objects or `.psprj` files.

Configuration Parameters Help: View descriptions of Polyspace options in Simulink configuration parameters

When you use the Simulink plugin, you must set Simulink configuration parameters to run your analysis. If you need help setting the configuration parameters, you can now right-click a configuration parameter and get `What’s This` help. When you select `What’s This`, a help window opens with details about the different settings and limitations of the parameter.

For more information about the configuration parameters, see Configure Code Verification.

Eclipse Build Support: Set up Polyspace verification from Eclipse build command

In R2016b, if you use a build command to build your source code in Eclipse or an IDE based on Eclipse, you can easily set up your Polyspace verification. To obtain the compiler options required for the verification, trace the build command inside the IDE. For more information, see Configure Verification.

Visual Studio 2010 add-in support to be removed from installation

In a future release, the Polyspace add-in for Visual Studio® 2010 will not be included with the installation.

To run Polyspace on code from Visual Studio, use the automatic configuration tool instead. See Create Project Using Visual Studio Information.

If you still want to use the add-in, you will be able to download the add-in from MATLAB Answers.

Support for Rhapsody 8.1

Starting in R2016b, the Polyspace plugin for IBM Rational® Rhapsody® supports Rhapsody 8.1. For more information, see Verify Code in IBM Rational Rhapsody Environment.

DOS Mode Warning on Linux: Compilation warning for DOS inconsistencies

When using Polyspace on Linux, a new compilation warning may appear. On Windows, DOS is case-insensitive meaning you cannot have two files with the same name but different capitalization. If you select the option Code from DOS or Windows file system (-dos)Code from DOS or Windows file system (-dos), Polyspace simulates this DOS behavior on Linux. If your source files include header files with inconsistent capitalization and it is unclear which file should be included, Polyspace issues a compilation warning.

For example, consider these two situations:

	Include Statements	Include Files

Situation 1	`#include "myheader.h"`
	`#include "MYHEADER.h"`
	`#include "MyHeader.h"`
		`myheader.h`

Situation 2	`#include "myheader.h"`
	`#include "MYHEADER.h"`
	`#include "MyHeader.h"`
		`myheader.h`
		`MYHEADER.h`

In the first situation, only one file exists with the name `myheader.h`. Because these include statements can only refer to one file, it is obvious which file to include. A warning is not issued.

In the second situation, two files exist: `myheader.h` and `MyHeader.h`. Because they have the same name and different capitalization, the capitalization in the include statement affects which file is included. Polyspace can find perfect matches for the first and second include statements. The last include statement is not a perfect match, so could refer to either header file. Because there is ambiguity with the last include statement, Polyspace issues this compilation warning: `warning: could not find include file "MyHeader.h"`.

In a future release, this compilation warning will become a compilation error.

Faster Restart for Remote Verification: Reuse compilation results from a previous analysis

In R2016b, if a remote verification stops after compilation, for instance because of communication problems between the server and client computers, you do not have to restart the verification from the beginning. You can reuse compilation results from the previous failed analysis.

For more information, see `-submit-job-from-previous-compilation-results`.

Internal Memory Limits Removed: Expect fewer analysis failures from memory-intensive processes

In R2016b, several internal limits on memory usage have been removed. Previously, if certain processes consumed memory above a certain limit (around 5 GB per process), those processes were stopped and the overall analysis failed. Now you are less likely to see failures from memory-intensive processes.

If you were unable to complete analysis on large or complex projects because of failures from memory-intensive processes, you are more likely to succeed in R2016b.

Support for local threads

Starting in R2016b, Polyspace adds support for these local thread modifiers:

- `__thread` — requires Compiler (-compiler) `gnu4.8`
- `__declspec(thread)` — requires **Compiler** `(-compiler)` `visual`
- `thread_local` — only for C++ code.

This support may eliminate compilation errors or change variable from shared to non-shared.

Changes in Target & Compiler analysis options

In R2016b, these **Target & Compiler** options have been added, changed, or removed.

Option	Change	More Information
Compiler (-compiler)	New option	
Dialect (`-dialect`)	Removed from the user interface. If you use the option in your scripts, you see a warning.	Option will be permanently removed in a future release. Replace `-dialect` with `-compiler` while retaining the option argument. In the user interface, this replacement is done automatically for existing projects. If you use the Wind River Diab compiler to build your source code, use the option Compiler (-compiler) with argument `diab`.
Target processor type (-target)	Updated for the Wind River Diab compiler.	In the user interface, if you select `diab` for Compiler (-compiler), you see target processors that are tailored to the Diab compiler. For the processor specifications, see the contextual help.

Option	Change	More Information
Target operating system (`-OS-target`)	Removed from the user interface. If you use the option in your scripts, you see a warning.	Option will be permanently removed in a future release. Remove the option from your scripts. For some option arguments, you might have to perform these additional steps: • `Linux`: If you get compilation errors, use a `gnux.x` argument for Compiler (-compiler). Sometimes, you might have to explicitly define operating-system-specific macros such as `linux`, `unix`, or `__linux__`. See Preprocessor definitions (-D). • `Visual`: Use a `visualx.x` argument for Compiler (-compiler). • `Vxworks`: Use the options from the VxWorks templates. Create a Polyspace project using one of the VxWorks templates and generate a script from your project. Copy the options related to the VxWorks template from this script. For more information, see Create Project Using Configuration Template and the reference page for `-generate-launching-scripts-for`. • `Solaris`: Just remove the option `-OS-target`. • `no-predefined-OS`: Just remove the option `-OS-target`.

Changes in analysis options and binaries

In R2016b, these options have been added, changed, or removed.

For changes to **Target & Compiler** options, see “Changes in Target & Compiler analysis options” on page 9-6. For other options, see the following table.

New Options

Option	Description
Cyclic tasks (-cyclic-tasks)	Specify functions that represent cyclic tasks.
Interrupts (-interrupts)	Specify functions that represent nonpreemptable interrupts.
Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)	Specify that stubbed pointers coming from external code can be unsafe to dereference, unless otherwise constrained.
Consider volatile qualifier on fields (-consider-volatile-qualifier-on-fields)	Consider that structures with `volatile`-qualified fields can change between consecutive accesses.
Subnormal detection mode (-check-subnormal)	Detect operations that result in subnormal floating point values.
Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions)	Stub autogenerated functions that use lookup tables and model them more precisely.
-preemptable-interrupts	Specify functions that represent preemptable interrupts.
-non-preemptable-tasks	Specify functions that represent nonpreemptable tasks.
-function-behavior-specifications	• Map your library functions to standard library functions recognized by Polyspace. • Specify functions that contain lookup tables with linear interpolation and no extrapolation.
-submit-job-from-previous-compilation-results	Specify that the analysis job must be resubmitted without recompilation.

Updated Options

Option	Change	More Information
Detect stack pointer dereference outside scope (-detect-pointer-escape)	Option now available in user interface.	
Coding rule subsets `single-unit-rules` and `system-decidable-rules`	Subsets now available in the Polyspace interface.	These subsets are available for Check MISRA C:2004 (-misra2), Check MISRA AC AGC (-misra-ac-agc), and Check MISRA C:2012 (-misra3)
-check-infinite and -check-nan	Option argument renamed.	Use `warn-first` as option argument instead of `warn`.

Removed Options

Option	Status	More Information
Green absolute address checks (`-green-absolute-address-checks`)	Error	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`)	Error	Use the option Do not generate results for (`-do-not-generate-results-for`) to suppress results from headers and sources in certain files or folders.
Ignore float rounding (`-ignore-float-rounding`)	Error	Option will be removed in a future release.
`-retype-pointer`	Error	Option will be removed in a future release.
`-retype-int-pointer`	Error	Option will be removed in a future release.
`-lwtm`	Error	Option will be removed in a future release.
`-support-FX-option-results`	Error	Option will be removed in a future release.
`polyspace-vcproj`	Removed	Use `polyspace-configure` or the Polyspace Add-In for Visual Studio instead.
`polyspace-automatic-verification`	Warning	Binary will be removed in a future release.
`polyspace-verifier`	Warning	Binary will be removed in a future release.
`rte-kernel`	Warning	Binary will be removed in a future release.
`polyspace-remote`	Warning	Binary will be removed in a future release.
Import folder (`-import-dir`)	Warning	Option will be removed in a future release.
No automatic stubbing (`-no-automatic-stubbing`)	Warning	Option will be removed in a future release.
`-easy-setup-preprocess`	Warning	Option will be removed in a future release.
`gui-api`	Warning	Binary will be removed in a future release. Use `polyspace-comments-import` instead.

Compatibility Considerations

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

Verification Results

Subnormal Float Detection: Identify loss of precision from operations that lead to subnormal results

In R2016b, the verification detects operations that result in subnormal floating-point values. The presence of subnormal numbers indicates loss of significant digits. This loss can accumulate over subsequent operations and eventually result in unexpected values. Subnormal numbers can also slow down the execution on targets without hardware support. If you run a Polyspace verification, you can choose to see one of the following:

- All operations that lead to subnormal results.
- Only those operations that lead to subnormal results from normal operands.

  For instance, the numbers `MIN_FLOAT` and `nextabove(MIN_FLOAT)` are normal, but their difference is subnormal.

For more information, see:

- Subnormal detection mode (-check-subnormal): The option to specify subnormal detection.
- Subnormal float: The result of subnormal detection.

Local Variable Size Estimation: Find total size of local variables in a function

In R2016b, you can compute the total size of local variables in a function by using these two metrics:

- Lower Estimate of Local Variable Size: Total size of local variables taking nested scopes into account.

  If a function has variable definitions in nested scopes, the software computes the total variable size in each scope and uses whichever total is greatest. For instance, if a conditional statement has variables definitions, the software computes the total variable size in each branch, and then uses whichever total is greatest.
- Higher Estimate of Local Variable Size: Total size of all local variables.

Changes to coding rule checking

Expanded MISRA C++ Support

The following MISRA C++:2008 rules are now supported.

- 0-1-9: There shall be no dead code.
- 0-1-11: There shall be no unused parameters (named or unnamed) in nonvirtual functions.
- 0-1-12: There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.
- 0-2-1: An object shall not be assigned to an overlapping object.
- 16-6-1: All uses of the #pragma directive shall be documented.

Updated Specifications

The Polyspace specifications for these rules have been updated.

Standard	Rule	Change
MISRA C++:2008	2–10–3	The violation is on the second instance of the duplicate identifier instead of the first.
	2–10–4	The violation is on the second instance of the duplicate identifier instead of the first.
	5–0–3	If two types have the same size in the target configuration, Polyspace does not raise a violation.
	5–0–6	If two types have the same size in the target configuration, Polyspace does not raise a violation.
	5–0–8	If two types have the same size in the target configuration, Polyspace does not raise a violation.
MISRA C:2004 and MISRA AC AGC	5.3	The violation is on the second instance of the duplicate identifier instead of the first.
	5.4	The violation is on the second instance of the duplicate identifier instead of the first.
	10.1	If two types have the same size in the target configuration, Polyspace does not raise a violation.
	10.2	If two types have the same size in the target configuration, Polyspace does not raise a violation.
	10.3	If two types have the same size in the target configuration, Polyspace does not raise a violation.
	10.4	If two types have the same size in the target configuration, Polyspace does not raise a violation.
MISRA C:2012	5.3	The violation is on the second instance of the duplicate identifier instead of the first.
	5.4	The violation is on the second instance of the duplicate identifier instead of the first.
	10.3	If two types have the same size in the target configuration, Polyspace does not raise a violation.
	10.6	If two types have the same size in the target configuration, Polyspace does not raise a violation.
	10.7	If two types have the same size in the target configuration, Polyspace does not raise a violation.
	10.8	If two types have the same size in the target configuration, Polyspace does not raise a violation.

Metrics for C++ Templates: View code complexity metrics for instances of C++ templates

In R2016b, you can compute code complexity metrics for C++ templates. If you instantiate a C++ template function and specify the option Calculate code metrics (-code-metrics), you now see function metrics for the template in your analysis results.

The metrics appear on the template definition. The software uses the first instance of the template to calculate the metrics. If you specialize a template, you see separate metrics for the original template and its specialization.

For more information, see Code Metrics.

Mutual Exclusion Support: View precise ranges for shared variables protected by critical sections and temporally exclusive tasks

In R2016b, when you check multitasking code for run-time errors, the error checking uses the protections that you specify. Previously, the software only determined if the protections were sufficient to prevent concurrent access of shared variables. The run-time error checking did not use those protections and considered all shared variables as unprotected. The improved support for protections in R2016b reduces the number of orange checks in multitasking code.

The following example illustrates the change. For a more detailed tutorial, see Manually Protect Shared Variables from Concurrent Access.

<table>
<tr><th>Prior to R2016b</th><th>R2016b</th></tr>
<tr><td>

In the following code, if you specify that `task` and `interrupt_handler` are temporally exclusive, the shared variable `shared_var` is protected from concurrent access. However, the **Overflow** check on the addition `shared_var += 2;` does not consider this protection. The check is orange because the verification considers that the addition can directly follow the assignment `shared_var = INT_MAX;`.

```
#include <limits.h>
int shared_var;

void inc() {
 /* Orange overflow */
 shared_var += 2;
}

void reset() {
 shared_var = 0;
}

void task() {
  volatile int randomValue = 0;
  while(randomValue) {
    reset();
    inc();
    inc();
  }
}

void interrupt() {
 shared_var = INT_MAX;
}

void interrupt_handler() {
  volatile int randomValue = 0;
  while(randomValue) {
   interrupt();
  }
}

 void main() {
}
```

</td><td>

In the following code, the **Overflow** check on the addition considers the temporal exclusion of `task` and `interrupt_handler`. The check is green because the verification considers that the addition cannot directly follow the assignment `shared_var = INT_MAX;`. The assignment `shared_var=0` takes place in between the two operations.

```
#include <limits.h>
int shared_var;

void inc() {
 /* Green overflow */
 shared_var+=2;
}

void reset() {
 shared_var = 0;
}

void task() {
  volatile int randomValue = 0;
  while(randomValue) {
    reset();
    inc();
    inc();
  }
}

void interrupt() {
 shared_var = INT_MAX;
}

void interrupt_handler() {
  volatile int randomValue = 0;
  while(randomValue) {
   interrupt();
  }
}

 void main() {
}
```

</td></tr>
</table>

If the shared variable is a pointer or an array, this change in behavior does not occur. Run-time error checking on shared pointers and arrays does not consider the protections.

Compatibility Considerations

If you use protections such as critical sections and temporal exclusion of tasks, you can see a reduction from previous releases in the number of orange checks.

Improved Embedded Coder Support: View more precise results when generated code uses lookup tables or large data structures

Lookup Tables

In R2016b, the verification assumes more precise return values for generated functions that use a lookup table in their body. If your model has Lookup Table blocks, such functions are generated . Previously, the software assumed full range for the return values of those functions. To avoid orange checks from this overapproximation, for certain kinds of lookup tables, the software now assumes that the function return values are within the bounds of the lookup table. The software makes this assumption only if the lookup table in the function uses linear interpolation and does not allow extrapolation. For more information, see Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions).

If the software does not detect functions that use lookup tables of this kind, you can also explicitly specify such functions. For instance, if you use a lookup table function in an S-Function block, the function name might not adhere to the naming convention for lookup table functions. If the software assumes full range for its return value, you can explicitly specify that the function uses a lookup table with linear interpolation and no extrapolation. For more information, see -function-behavior-specifications.

Large Data Structures Accessed Via Pointers

In R2016b, when your code has a large global data structure and a function accesses its fields via pointers, the verification is more precise than before. Previously, if a function modified one field via pointers, in some cases, the verification lost precision on other fields and assumed full range for their values.

Code generated from models can have large structures because all inputs to or outputs from the model are placed in one structure. With this improved precision, you can see more precise results for generated code in many cases.

Compatibility Considerations

If you run verification on generated code, you can see a reduction from previous releases in the number of orange checks.

Precise Buffer Manipulation Functions: View more precise results on complete copying of structures

In R2016b, if your code uses the `memcpy`, `memmove`, or `bcopy` function to copy structures, you can see fewer orange checks. Previously, if you copied one structure to another with these functions, the software assumed that each field of the destination structure had full range of values. The software now considers precise values for the result of the copy.

Assumption for Stubbed Pointers: Review fewer warnings from pointers coming from external code

In R2016b, the default verification for C code assumes that stubbed pointers coming from external code are safe to dereference. For instance, the pointer does not have a `NULL` value and pointer dereference is within allowed bounds.

Previously, the default verification assumed that stubbed pointers were unsafe to dereference. For instance, if you dereferenced the pointer return value of a stubbed function without checking for `NULL`, the **Illegally dereferenced pointer** check showed a warning in orange. With the change in default assumption, orange **Illegally dereferenced pointer** checks in your verification results are more likely to have a root cause within your code.

Compatibility Considerations

If you run verification on a Polyspace project from a previous release, you can see a reduction in the number of orange checks.

You can also see an increase in the number of gray checks. If your code contains protections against `NULL` values of environment pointers, for instance conditions such as `if(ext_ptr!=NULL)`, a gray check appears on these protections.

To revert to the previous default assumption, use the option Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe). Alternately, you can individually change the default assumption on certain pointers only in the Constraint Specification window. See the description for **Initialize Pointer** in Constraints.

Assumption for Structures with Volatile Fields: Review fewer warnings from partly volatile structures

In R2016b, the default verification ignores the `volatile` qualifier on fields of a structure. Previously, if a structure had a `volatile`-qualified field, the verification considered all fields of the structure as volatile. As a result, the verification assumed that their values always spanned the full range of their data types. This overapproximation sometimes caused false warnings in orange from the non-`volatile` fields.

If you use a structure whose fields represent values read from hardware, add the `volatile` qualifier to the structure definition instead of individual field definitions.

Compatibility Considerations

If you run verification on a Polyspace project from a previous release, you can see a reduction in the number of orange checks. Occasionally, you can also see changes in red or gray checks. For instance, if the field `structInstance.field` is volatile, branches of the condition `if(structInstance.field)` were previously always reachable. With the new assumption, depending on the value of `structInstance.field`, some of the branches can be unreachable.

To revert to the previous default assumption, use the option Consider volatile qualifier on fields (-consider-volatile-qualifier-on-fields).

Expected Infinite Loop Detection: Avoid justifying run-time errors on infinite loops that you introduce deliberately

In R2016b, the verification detects if an infinite loop is intentional. For these infinite loops, the verification does not produce a red **Non-terminating loop** error on the loop statement. If you deliberately introduce infinite loops, for instance, to emulate cyclic tasks, you do not have to justify red checks. For example, if a loop has a trivial predicate `while(1)` and there are no exit statements in the loop body, the verification considers the loop as intentional.

For more information, see Non-terminating loop.

Compatibility Considerations

If you run verification on a pre-R2016b project, you see a reduction from previous releases in the number of red **Non-terminating loop** checks. However, as in previous releases, any code following the infinite loop shows gray checks. Though the infinite loop is expected, the verification considers code following the infinite loop as unreachable.

Mapping to Standard Functions: View precise results by mapping imprecisely analyzed functions to corresponding standard functions

In R2016b, if you encounter imprecisions in analysis of your custom library function, you can map the function to a standard library function for more precise analysis.

Polyspace Code Prover cannot analyze certain code constructs because of inherent limitations with static verification. If your custom library function uses one of those constructs, to avoid missing a run-time error, the verification assumes all possible results from the function call. To avoid orange checks from this overapproximation, you can map your custom library function to a standard library function. Although the software does not analyze the body of your library function, in the various call sites, the software emulates your function behavior more precisely. For instance, the software assumes a more precise range for the function return value. The reason for this precise analysis is that the software models effects of standard library functions extremely precisely.

For instance:

- If you have an implementation of a trigonometric function and the software assumes full range for the return value, map your implementation to the corresponding standard library trigonometric function.
- If you have a function that copies contents of one memory location to another and the software assumes that the destination location is still uninitialized, map your function to the `memcpy` function.

You can map to only certain standard library functions from `math.h` and copying functions such as `memcpy`. Additionally, you can map your functions to some internal Polyspace functions for more precise analysis. For more information, see -function-behavior-specifications.

Reviewing Results

Interactive Graphical Display: Click graphs on Dashboard to filter results

In R2016b, you can narrow down the scope of your review by using a graphical display of analysis results. Previously you used the graphs to obtain an overview of the analysis results and determine which results to focus on. Now you can also select elements in the graphs to view only those results that you want to focus on. To see all results again, clear your filters in one click.

To filter results, use the following graphs:

- **Check distribution**: If you click a colored region on this pie chart, the **Results List** pane shows checks of that color only.
- **Top 5 coding rule violations**: If you click a column corresponding to a rule, the **Results List** pane shows violations of that rule only.
- **Top 5 orange sources**: If you click a column corresponding to an orange source, the **Results List** pane shows orange checks caused by that source only.

For more information, see Filter and Group Results.

Float Range Display: View float variables with narrow ranges more clearly

In R2016b, the tooltips on float variables show an improved display of the variable ranges. For instance:

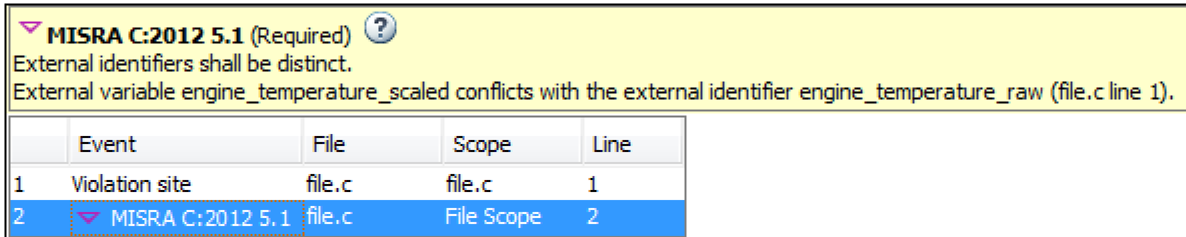
- If the lower and upper bounds of a float variable are close, the tooltip displays as many digits as required to distinguish between them.
- The tooltips clearly indicate which values are shown with rounding. For instance, the value `1.0` does not involve rounding but `1.2345...` shows a variable that is displayed with rounding towards zero.

  When rounded, at least 5 significant digits are displayed.

For more information, see Source.

Event History for Coding Rules: Navigate easily between two locations in code that together cause a rule violation

In R2016b, for certain coding rules, the **Result Details** pane shows previous events causing the rule violation. You can click an event and navigate to the corresponding location in the source code.



This event history is shown for those rules which are related to more than one location in the code. For instance, the event history appears for the following rules:

- MISRA C:2004 Rule 5.2: Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
- MISRA C:2012 Rule 5.1: External identifiers shall be distinct.
- MISRA C++ Rule 2-10-1: Different identifiers shall be typographically unambiguous.
- JSF® C++ Rule 139: External objects will not be declared in more than one file.

Subcheck Display for Standard Library Routines: Determine easily from visual inspection which subcheck failed

The **Invalid use of standard library routine** check consists of multiple subchecks. In R2016b, you can determine visually from the **Result Details** pane which subchecks failed.

- If a subcheck passes, it is marked with a ✓.
- If a subcheck fails in all the cases that the verification considers, it is marked with a **!**.
- If a subcheck fails only in some of the cases, it is marked with a **?**.

For instance, in the following example, the first subcheck passed but the second one failed.

! ID 22: Invalid use of standard library routine
Error: function 'memcpy' is called with invalid argument(s)
- Checks on first argument (destination):
  - ✓ Not null.
  - ! Is not a memory area that is accessible within the boundary given by the third argument.

In the following example, the subchecks on the first argument passed but the first subcheck on the second argument failed sometimes.

? ID 31: Invalid use of standard library routine
Warning: function 'memcpy' is called with possibly invalid argument(s)
- Checks on first argument (destination):
  - ✓ Not null.
  - ✓ Is a memory area that is accessible within the boundary given by the third argument.
- Checks on second argument (source):
  - ? May be null.
  - ✓ Is a memory area that is accessible within the boundary given by the third argument.

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

Results from Macros: Coding rule violations highlighted on macro definitions instead of macro instances

When you run coding rules checking, violations from macro definitions can propagate throughout your code causing many results. In R2016b, coding rule violations caused by a macro are now highlighted on the macro definition. This change reduces the number of coding rule violations with the same root cause, simplifying your review process.

Verification Objectives in Eclipse: Create review scopes to focus your review

From the Eclipse plugin, you can now create custom review scopes. Review scopes filter your results to only the run-time checks, coding rules, or code metrics that you want to see. For more information, see Limit Display of Results.

Filtered Report: Reuse result filters for generated report

In R2016b, if you apply filters to your results, you can reuse those filters for the generated report. For instance, you can use filters to view only the following subset of results on the **Results List** pane and then reuse those filters for the report.

- View only critical checks (red, gray, and critical orange) and create a report with those checks only.
- View only new results found since the last analysis and create a report with the new results only.
- View only code metrics that exceed specified thresholds and create a report with those metrics only.

On the **Results List** pane, you can apply complicated filtering criteria to show only the results that are most meaningful to you. You can reuse these criteria for your generated report and show only the results that you want the report reviewer to focus on. For more information on the filters you can use, see Filter and Group Results.

The report shows which filters you have applied. Another person reviewing your report can see your filtering criteria.

Results Export: Export results to text file for computing graphs and statistics

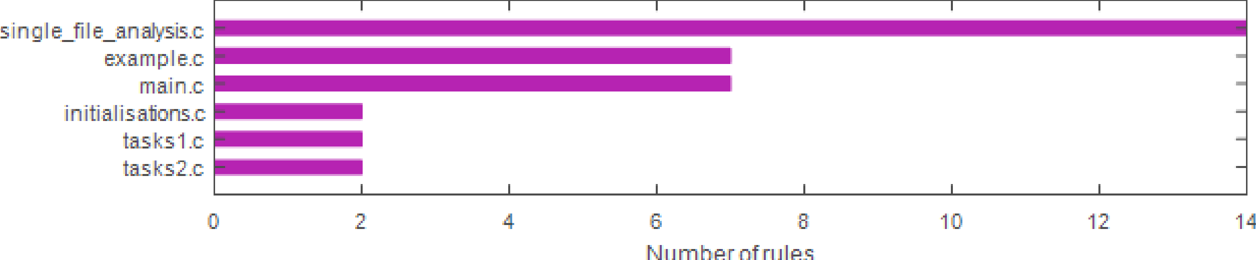
In R2016b, you can export your results to a tab delimited text file. You can parse the text file by using MATLAB or Excel and generate graphs or statistics about your results that you cannot readily obtain from the user interface. For instance, for each check type (**Division by zero**, **Overflow**), you can calculate how many checks are red, orange, or green.

For more information, see Export Results to Text File.

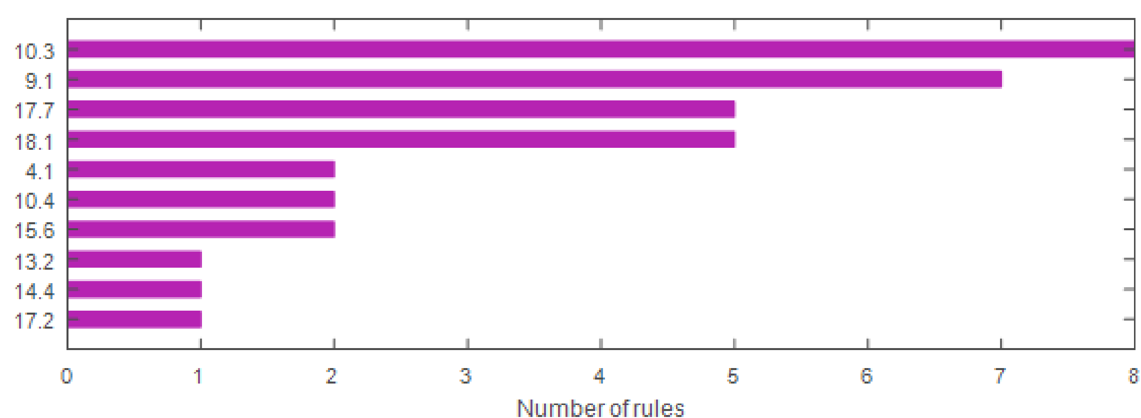
Coding Rule Graphs in Report: View breakdown of coding rules violations by rule number and file

In R2016b, if you choose to report coding rule violations, the report contains two new graphs.

- The first graph shows the number of coding rule violations broken down by file.



- The second graph shows the number of violations broken down by rule number.



Constraints in Report: Add comments about external constraints and view comments in report

In R2016b, when you specify external constraints for verification and add comments in the Constraint Specification window, the comments appear in the generated report. Another person reviewing your report can see your comments. You can use the comments to provide explanations for your constraints.

The constraints, along with your comments, appear in the report appendix that lists your verification options.

For more information, see:

- Constraints
- Generate Report

English Reports in Non-English Locales: Generate English reports on operating systems with a different language

In R2016b, even if your operating system has a display language (Windows) or locale (Linux) such as Japanese or Korean, you can still generate English reports. See Generate Report After Verification.

Improved PDF report generation

In R2016b, the generation of PDF reports is improved.

- The report generation is faster. For large results, the report generation is much less likely to cause out-of-memory errors.
- The reports use an improved visual display.

Change in report template location

The location of the report template files has changed to `*matlabroot*/toolbox/polyspace/psrptgen/templates`. Here, *`matlabroot`* is the MATLAB installation folder.

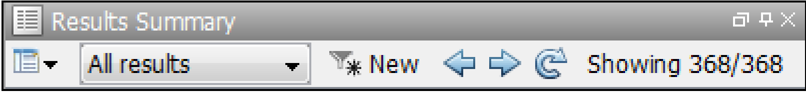
If you use the report templates provided by Polyspace, the change does not impact you. If you use MATLAB Report Generator™ to modify the Polyspace report templates, you can open the templates from this new location.

Changes in Polyspace User Interface

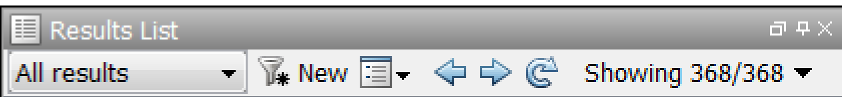
The following table lists minor changes to the user interface including new pane names and new icons.

- **Results List** — Window showing list of results, previously called **Results Summary**.
- — Button to remove items in the configuration or projects.
- The icons on the **Results List** pane have been rearranged.

  In R2016a, the icons were arranged as follows.



  In R2016b, the same icons are arranged as follows.



R2016a

Version: 9.5

New Features

Bug Fixes

Compatibility Considerations

Verification Setup

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`.

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.

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`

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 10-7.

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

Without constraints, Polyspace assumes that `x` in `bar` and and `bar_array` are potentially noninitialized when you read them. You cannot specify that the functions `foo` and `foo_array` initialize `x` with full-range values.
 	
`#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]);`
`}`
	If you specify the following constraints in the **Init Allocated** column, Polyspace considers that `x` in `bar` and `bar_array` are initialized.
	
	• `foo`: Specify `SINGLE_CERTAIN_WRITE` for the argument of `foo`. In other words, `foo` writes a value to `*ptr`.
	• `foo_array`: Specify `MULTI_CERTAIN_WRITE` for the argument of `foo_array`. In other words, `ptr` points to an array and `foo_array` writes a value to the array elements.
	
	`#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]);`
	`}`

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.

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.

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.	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. For instance, in your code, you define a type `myBool` using a `typedef` and cast the result of `(a && b)` to `myBool`. If you specify to Polyspace that `myBool` 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).
`MISRA C:2004 Rule 12.2`	The value of an expression shall be the same under any order of evaluation that the standard permits.	The rule checker no longer flags expressions with the comma operator that can be evaluated in only one order. For instance, the statement `ans = (val++, val++)` does not violate this rule.

MISRA C:2012 Rules

Rule	Description	Improvement
`MISRA C:2012 Rule 13.2`	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.	The rule checker no longer flags expressions with the comma operator that can be evaluated in only one order. For instance, the statement `ans = (val++, val++)` does not violate this rule.

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 Analysis.

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-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.

Verification 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.

<table>
<tr><th>Prior to R2016a</th><th>R2016a</th></tr>
<tr><td>In the following code, Polyspace produces a Division by zero error and stops verification.

<pre>double func(void) {
    double x=1.0/0.0;
    double y=1.0/x;
    double z=x-x;
    return z;
}</pre></td><td>In the following code, if you specify the option Allow non finite floats, Polyspace does not check for a Division by zero error.

<pre>double func(void) {
    double x=1.0/0.0;
    double y=1.0/x;
    double z=x-x;
    return z;
}</pre>The verification assumes that dividing by zero results in:
Value of <code>x</code> equal to <code>Inf</code>
Value of <code>y</code> equal to 0.0
Value of <code>z</code> equal to <code>NaN</code>
In your verification results in the Polyspace user interface, if you place your cursor on <code>y</code> and <code>z</code>, you can see the nonfinite values <code>Inf</code> and <code>NaN</code> respectively in the tooltip.</td></tr>
</table>

Rounding modes

Polyspace supports verification that considers all possible rounding modes when rounding the results of floating point arithmetic. Using the option Float rounding mode (`-float-rounding-mode`), you can enable a verification mode that allows these forms of rounding: round-to-nearest, round-towards-zero, round-towards-positive-infinity and round-towards-negative-infinity. The default rounding mode is round-to-nearest only.

Previously, the default verification assumed all rounding modes to determine the results of floating-point arithmetic. The verification used the round-to-nearest mode only to determine if an **Overflow** occurs.

The following table illustrates the change.

<table>
<tr><th>Prior to R2016a</th><th>R2016a</th></tr>
<tr><td>

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.

```
#include <float.h>

void func(void) {
    double base = DBL_MAX;
    double acc = 1.247400193459199882
2852329456480241037921570377722e+291;

    base = acc + base;
}
```

</td><td>

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.

```
#include <float.h>

void func(void) {
    double base = DBL_MAX;
    double acc = 1.247400193459199882
2852329456480241037921570377722e+291;

    base = acc + base;
}
```

</td></tr>
</table>

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.

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

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

Reviewing Results

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**.

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.

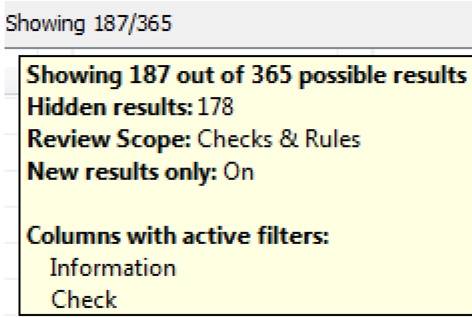
When setting up your projects, select **Window** > **Reset Layout** > **Project Setup**. When reviewing results, select **Window** > **Reset Layout** > **Results Review**.

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

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 New filter to suppress results found in a previous verification.
- Filters on the **Information** and **Check** columns.

For more information, see Filter and Group Results.

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.

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.

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.

R2015b

Version: 9.4

New Features

Bug Fixes

Compatibility Considerations

Verification Setup

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++).

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++).

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++).

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++).

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`.

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**.

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.

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.

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.

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.

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++).

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 `-class-analyzer` *`param`* has been updated. Use `-class-analyzer custom=`*`param`*
Signed right shift (C/C++) (`-logical-signed-right-shift`)	Now available in C ++ projects	
Division round down (C/C++) (`-div-round-down`)	Now available in C ++ projects	
(`-no-def-init-glob`)	Now available in C ++ projects	
Optimize large static initializers (C/C++) (`-no-fold`)	Now available in C ++ projects	
`-lightweight-thread-model`	No longer available in the user interface	
Targets: • `tms320c3x` • , `sharc21x61` • `necv850` • `hc08` • `hc12` • `mpc5xx` • `c18`	Now available in C ++ projects	
Enum type definition (C/C++)(`-enum-type-definition`)	Possible values updated	The possible values for `-enum-type-definition` are the same for C and C++. Available values: • `defined-by-standard (default)` • `auto-signed-first` • `auto-unsigned-first`
`-asm-begin -asm-end`	Now available in C ++ projects	

Option	Status	More information
`-support-FX-option-results`	No longer available in the user interface	
`-pointer-is-24bits`	Available in C++ projects	Availably only if you use the **Target** setting `c18`.
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.
`-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.

Option	Status	More information
`-discard-asm`	Removed	This option is no longer supported. Remove this option from existing configurations.
`-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.

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.

Verification Results

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

<table>
<tr><th>Prior to R2015b</th><th>R2015b</th></tr>
<tr><td>

In the following code, if you specify `task1` and `task2` as entry points, the verification determines that `var_1` and `var_2` are shared, potentially unprotected variables. However, because of the `if` statement, `task1` can operate only on `var_1` and `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.

```
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;
}
```

</td><td>

In the following code, if you specify `task1` and `task2` as entry points, the verification determines that `var_1` and `var_2` are not shared.

```
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;
}
```

</td></tr>
</table>

Shared Structures

<table>
<tr><th>Prior to R2015b</th><th>R2015b</th></tr>
<tr><td>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 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>.
<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></td><td>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 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.
<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></td></tr>
</table>

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 `<fenv.h>` should not be used.
MISRA C:2012 Rule 22.5	A pointer to a `FILE` object shall not be dereferenced.

Improved precision for mathematical functions

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

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 declared in one file and only one file.	Polyspace considers that variables or functions declared `extern` in a non-header file violate this rule.
`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.

Rule	Description	Improvement
MISRA C:2004 Rule 19.2	Nonstandard characters should not occur in header file names in #include directives.	Polyspace no longer raises violation of this rule if the character \ or \\ occurs between the < and > in #include <filename> (or between " and " in #include "filename"). Therefore, you can use Windows paths to files in place of filename 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 .c 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 .c file 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.	• 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. For instance, the statement u8b = u8a + 3;, where u8a and u8b are unsigned char variables, does not violate the rule because the constants 3 and 3U have the same representation.

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.

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.

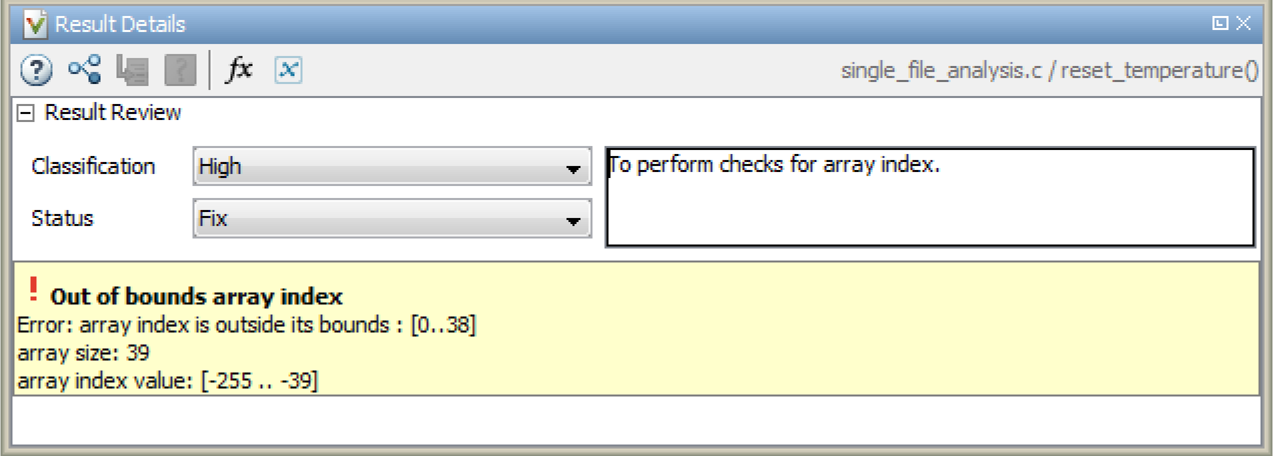
Compatibility Considerations

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.

Reviewing Results

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.



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.

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.

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 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++).

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**.

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

Verification Setup

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.

<table>
<tr><th>Project Manager</th><th>Results Manager</th></tr>
<tr><td>

- **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.

</td><td>

- **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.

  To open the log associated with a result, with the results open, select **Window** > **Show/Hide View** > **Run Log**.
- **Settings**: Same information as **Configuration**, but associated with results instead of a project. This pane has been removed.

  To open the configuration associated with a result, with the results open, select **Window** > **Show/Hide View** > **Configuration**.
- **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.

</td></tr>
</table>

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.

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 used 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
`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`

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.

Verification Results

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

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

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.
```

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.

<table>
<tr><th>Before R2015a</th><th>R2015a</th></tr>
<tr><td>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.
<pre>
void foo(...) {
  /* Function body */
}

void main() {
  foo();
  foo(1,2); //Red COR
}
</pre></td><td>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.
<pre>
void foo(...) {
  /* Function body */
}

void main() {
  foo();
  foo(1,2); //No COR
}
</pre></td></tr>
</table>

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.

<table>
<tr><th>Before R2015a</th><th>R2015a</th></tr>
<tr><td>

In the following code, Polyspace does not check the comparison `ptr==&invalid`. Therefore, it considers that `check` can return either 0 or 1. In the `main` function, it verifies both branches of the `if-else` statement.

```
#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 */
    }
}
```

</td><td>

In the following code, Polyspace checks the comparison `ptr===&invalid` and determines that it is always true. Therefore, it considers that the `if` test is redundant and the function `check` returns 1 only. In the `main` function, it verifies the `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 */
    }
}
```

</td></tr>
</table>

Improvements in coding rules checking

MISRA C:2004 and MISRA AC AGC

Rule Number	Effect	More Information
Rule 12.6	More results on noncompliant `#if` 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 `float`	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 `sizeof` operator and on expressions with `?` 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 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 `try` block.	MISRA C++ Rules — Chapter 15

Reviewing Results

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

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

- “Code Complexity Metrics” on page 12-10
- “Global Variables” on page 12-10

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.

<table>
<tr><th colspan="2">Category</th><th>Color</th><th>Meaning</th></tr>
<tr><td rowspan="2">Shared</td><td>Potentially unprotected</td><td>Orange</td><td>Global variables shared between multiple tasks but possibly not protected from concurrent access by the tasks</td></tr>
<tr><td>Protected</td><td>Green</td><td>Global variables shared between multiple tasks and protected from concurrent access by the tasks</td></tr>
<tr><td rowspan="2">Not shared</td><td>Used</td><td>Black</td><td>Global variables used in a single task</td></tr>
<tr><td>Unused</td><td>Gray</td><td>Global variables declared but not used</td></tr>
</table>

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.

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.

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.

R2014b

Version: 9.2

New Features

Bug Fixes

Compatibility Considerations

Verification Setup

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.

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++).

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 Parallel 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.

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 `polypsace-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 `polypsace-code-prover-nodesktop` command.

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`

Option	Value
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.

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.

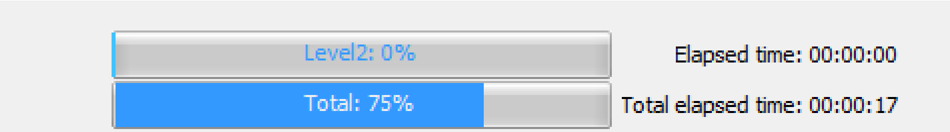
Goal	Prior to R2014b	R2014b
Open the Polyspace Metrics interface in your web browser.	**File > Open Metrics Web Interface**	**Metrics > Open Metrics**
Upload results from the Polyspace user interface to Polyspace Metrics.	**File > Upload in Polyspace Metrics repository**	**Metrics > Upload to Metrics**
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.

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`	`polyspace-job-monitor.exe`	The binary and the interface titles have changed. Interface labels have changed in the Polyspace interface and its plug-ins.
Queue Manager or **Spooler**	**Job Monitor**	
`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`
`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	`matlabroot/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.

Verification Results

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 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
• 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.	When you read structured variables, a check for initialization occurs. The check turns: • 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 uninitialized fields have the full range of values allowed by their type. 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. To determine which fields Polyspace checked for initialization: 1 Select the NIV or NIVL check on the **Results Summary** pane or **Source** pane. 2 View the message on the **Check Details** pane.

Example:
 	
`typedef struct S {`
`   int a;`
`   int b;`
`}S;`
 	
`void func1(S);`
`void func2(int);`
 	
`void main() {`
`  S varS;`
`  func1(varS);`
`  func2(varS.a);`
`}`
 	
A check was not performed when the non-initialized structure `varS` was read. When the field `a` of `varS` was read, a red NIVL check appeared.
	Example:
	
	`typedef struct S {`
	`   int a;`
	`   int b;`
	`}S;`
	
	`void func1(S);`
	`void func2(int);`
	
	`void main() {`
	`  S varS;`
	`  func1(varS);`
	`  func2(varS.a);`
	`}`
	
	When the non-initialized structure `varS` is read, a red NIVL check appears.
	
	For more examples, see:
	• 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
If you initialized only the high bits of a variable through a pointer, an orange check for initialization appeared when the variable was read.	If you initialize only the high bits of a variable through a pointer, a green check for initialization appears when the variable is read.
If you performed an operation on a C++ object after it was destroyed, a red check for initialization appeared on the operation. The check indicated that the object was destroyed.	If you perform an operation on a C++ object after it is destroyed, the check for initialization has the same color as 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.

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.	`none`
Detect functions that are defined but not called.	`never-called`
Detect functions that are defined and called only from an unreachable part of the code.	`called-from-unreachable`
Detect all uncalled functions.	`all`

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++).

Reviewing Results

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 (i).

   The documentation for that check appears in a docked window.

For more information, see Getting Help.

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 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 ✕ test_file.c ✕

```
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.

R2014a

Version: 9.1

New Features

Bug Fixes

Compatibility Considerations

Verification Setup

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 Run. 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 https://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.

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 Parallel 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.

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.

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
`PolyspaceAnnotation`	Warning	`pslinkfun('annotations',...)`
`PolySpaceGetTemplateCFGFile`	Warning	`pslinkfun('gettemplate')`
`PolySpaceHelp`	Warning	`pslinkfun('help')`
`PolySpaceEnableCOMServer`	Warning	`pslinkfun('enablebacktomodel')`
`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`

Verification Results

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.

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,

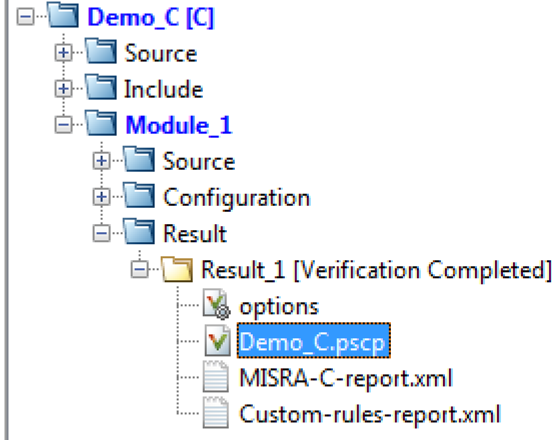
- Pre-R2013b, `arr = [9.9999E^-2,1.0001E-1]`.
- Now, `arr = 0.1`.

Reviewing Results

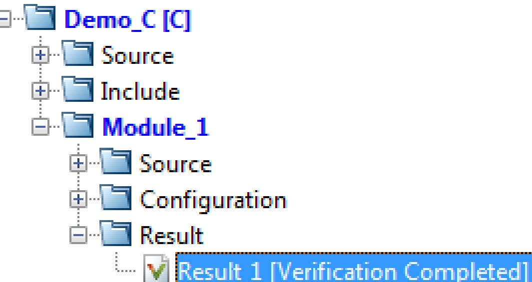
Results folder appearance in Project Browser

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.

  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.	For each coding rule, you can select three results: • **Error**: Analysis stops if the rule is violated. The rule violation is displayed on the **Output Summary** tab in the Project Manager perspective. • **Warning**: Analysis continues even if the rule is violated. The rule violation is displayed on the **Results Summary** pane in the Result Manager perspective. • **Off**: Polyspace does not check for violation of the rule.	For each coding rule, you can select two results: • **On**: Analysis continues even if the rule is violated. The rule violation is displayed on the **Results Summary** pane in the Result Manager perspective. • **Off**: Polyspace does not check for violation of the rule.

Coding Rules Feature	2013b	2014a

Format of the custom coding rules file.	Each line in the file must have the syntax:
	`rule off|error|warning #comments`
	For example:
	`# MISRA configuration - Proj1`
	`10.5 off #don't check 10.5`
	`17.2 error`
	`17.3 warning`
		Each line in the file must have the syntax:
		`rule off|warning #comments`
		For example:
		`# MISRA configuration - Proj1`
		`10.5 off #don't check 10.5`
		`17.2 warning`
		`17.3 warning`

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.

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.

R2013b

Version: 9.0

New Features

Verification Results

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.

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.

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.

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.

Reviewing Results

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.

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.

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: • Polyspace Client for C/C++ on local computer • Polyspace Server for C/C++ on network computers, which are configured as Queue Manager and CPUs.	Install: • MATLAB, Polyspace Bug Finder, and Parallel Computing Toolbox on local computer. • MATLAB, Polyspace Bug Finder, Polyspace Code Prover, and MATLAB Parallel 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	On the **Polyspace Preferences > Server Configuration** tab:
	• 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.
		On the **Polyspace Preferences > Server Configuration** tab:
		• 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**:
		  • Specify host computer for Polyspace Metrics server and communication port.
		  • Specify other settings for Polyspace Metrics.

Category	R2013a	R2013b
	In the Remote Launcher Manager dialog box: 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**.	In the Metrics and Remote Server Settings dialog box: 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: • **Start the Polyspace MDCE service** — Selected. The `mdce` 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 Parallel Server cluster security, see Cluster Security. 3 To start the Polyspace Metrics and `mdce` services, click **Start Daemon**. Use the Metrics and Remote Server Settings dialog box to start and stop `mdce` services only if you configure the MDCS head node as the Polyspace Metrics server. Otherwise, clear the **Start the Polyspace MDCE service** check box, and use the MDCS Admin Center. To open the MDCS Admin Center, run: `matlabroot/toolbox/distcomp/bin/admincenter` For information about the MDCS Admin Center, see Cluster Processes and Profiles.

Category	R2013a	R2013b
Running a remote verification	In the Project Manager perspective: 1 On the **Configuration** > **Machine Configuration** pane, select the following check boxes: • **Send to Polyspace Server** • **Add to results repository —** Allows viewing of results through Polyspace Metrics. 2 On the toolbar, click **Run**. 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.	In the Project Manager perspective: 1 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. 2 On the toolbar, click **Run**. 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 Parallel Server cluster.
Managing remote verifications	Use the Queue Manager to monitor and manage submitted jobs from Polyspace clients. 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 .	Use the Queue Manager to monitor and manage jobs submitted through Parallel Computing Toolbox clients.
Accessing results of remote verifications	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. 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.	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.

Results review

Category	R2013a	R2013b
Results Explorer	Available. Allows navigation through checks by the file and function where they occur. To view, select **Window** > **Show/Hide View** > **Results Explorer**.	Removed. To navigate through checks by file and function, on **Results Summary** pane, from the drop-down menu, select `File/Function`.
Filters on the **Results Summary** pane	Filters appear as icons on the **Results Summary** pane. You can filter by: • Run-time error category • Coding rules violated • Check color • Check justification • Check classification • Check status	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 appear. You can also define your own filters. The filters appear as the icon on each column header. To apply a filter using the information in a column: **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. For more information, see Organize Check Review Using Filters and Groups.
Code Coverage Metrics	In the **Results Explorer** view, the software displays two metrics for the project: • `unp` — Number of unreachable functions as a ratio of total number of functions • `cov` — Percentage of elementary operations covered by verification The unreachable procedures are marked gray in the **Results Explorer** view.	The new **Results Statistics** pane displays the code coverage metrics through the **Code covered by verification** column graph. To see a list of unreachable procedures, click this column graph. For more information, see Results Statistics.

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	`polyspace-c ...`
		
		`polyspace-cpp ...`
			`polyspace-code-prover-nodesktop ...`

	Project configuration file extension	*`project_name`*`.cfg`	*`project_name`*`.psprj`
	Results file extension	*`results_name`*`.rte`	*`results_name`*`.pscp`
	Configuration > **Machine Configuration** pane	Available	Replaced by **Configuration** > **Distributed Computing** pane.
	Configuration > **Post Verification** pane	Available	Renamed **Configuration** > **Advanced Settings**
	`goto` blocks	Not supported	Supported
	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.

Product	Feature	R2013a	R2013b
	Running a verification	**Code > Polyspace > Polyspace for Embedded Coder/Target Link** • Verify Generated Code • Verify Generated Model Reference Code Also right-clicking on a subsystem and selecting **Polyspace > Polyspace for Embedded Coder/ Target Link**	**Code > Polyspace > Verify Code Generated for** • Selected Subsystem • Model • Referenced Model • Selected Target Link Subsystem Also right-clicking on a subsystem and selecting **Polyspace > Verify Code Generated for > Selected Subsystem / Selected Target Link Subsystem**
	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.
	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: • Lambda functions • Rvalue references for `*this` 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-metrics`	Available. Not selected by default.	Removed. Code complexity metrics computed by default.
	`-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`.
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.