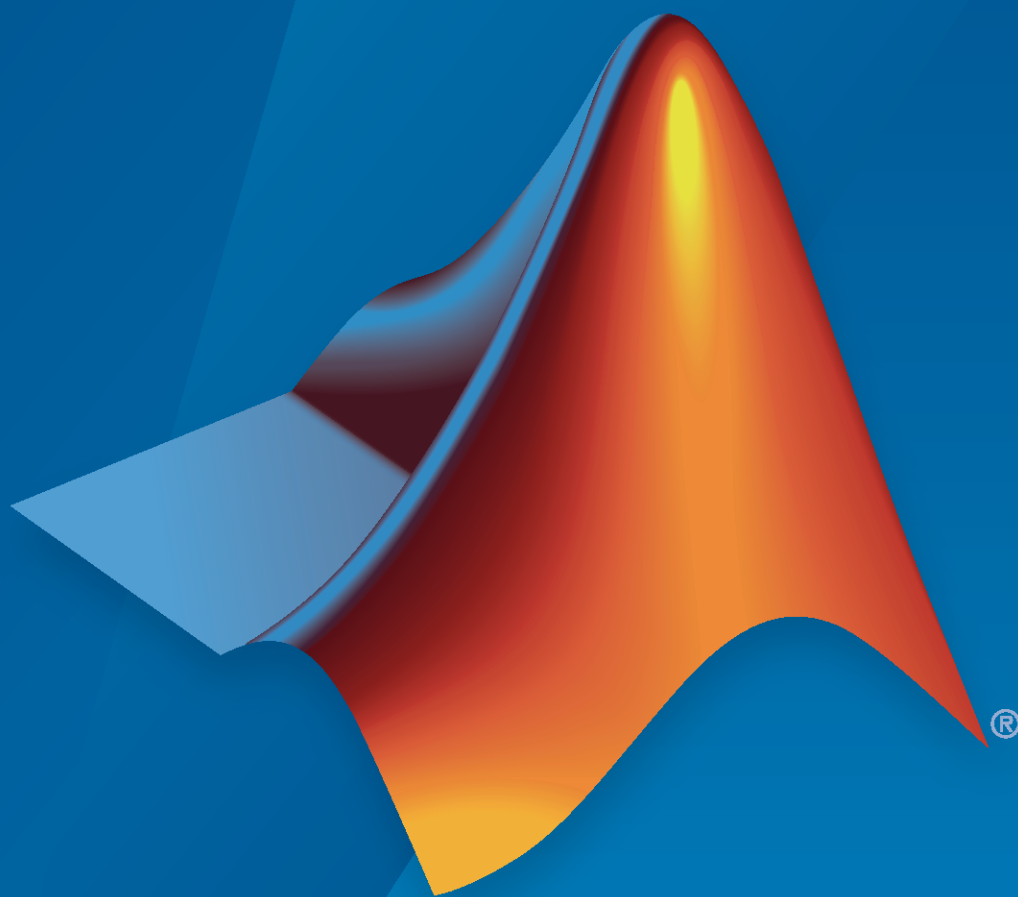


Polyspace[®] Bug Finder[™] Release Notes



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® Bug Finder™ Release Notes

© COPYRIGHT 2013–2023 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.

Analysis Setup	1-2
Project File Classification: Control precisely which files to include in analysis and how to analyze them	1-2
Security Standards Support: Check explicitly for subsets of CWE rules ...	1-2
Updated Clang Compiler Support: Set up Polyspace analysis for code compiled by using Clang version 13.x	1-3
Updated GCC Compiler Support: Set up Polyspace analysis for code compiled by using GCC versions 11.x and 12.x	1-3
Configuration from Build System: Import compiler includes automatically without tracing build	1-4
Target Alignment: New default alignment for target x86_64 and alignment up to 128 bits available for supported targets	1-4
Compilation of Large Projects: Improved performance and scalability when compiling large projects	1-5
polyspace-access Command: View release and license information, upload from file path, view URL of uploaded results	1-5
Changes in analysis options and binaries	1-6
Changes in MATLAB function, options object and properties	1-6
Analysis Results	1-8
Bug Finder Checkers: Check for duplicate code, copy-paste errors, and related issues	1-8
Bug Finder Checkers: Check for security vulnerabilities such as SQL injection or LDAP injection	1-8
Bug Finder Checkers: Check for infinite loops, reference to unnamed temporaries, and other issues	1-8
CERT C++ Support: Check for pointer arithmetic on polymorphic objects and other issues	1-9
MISRA C++:2008 Support: Check for issues caused by unused variables and enum types	1-9
CWE Support: Updated CWE version and support for additional rules ..	1-10
Updated Bug Finder defect checkers	1-10
Changes to coding standards checking	1-13
Reviewing Results	1-33
Security Standards Support: View violations of CWE rules as analysis findings in the Results List and in reports	1-33
Polyspace Extension for Visual Studio Code (VSCoDe): Fix common defects or coding rule violations in one click	1-34
Software Quality Objectives: View aggregated software quality objectives in Polyspace Access	1-35

Report Generation: Use CodingStandards report template to generate report for CWE results	1-35
Coding Standards Support: Display custom comments in Results List and Result Details pane	1-36
Inputs Causing Defect: See example system inputs that lead to non-initialized pointers	1-37
Simulink Support: Justify Known MISRA C++:2008 and AUTOSAR C++14 Violations	1-38
Polyspace user interfaces now available in Chinese	1-38
Reduction in duplicate results on templates	1-38
Polyspace Access Installation	1-40
Polyspace Extension for Visual Studio Code (VSCode): Polyspace as You Code available for macOS	1-40
Polyspace as You Code Plugin for Eclipse: Plugin is not compatible with the Polyspace desktop integration plugin	1-40

R2022b

Analysis Setup	2-2
Compiler Support: Set up Polyspace analysis for code compiled with Intel C++ Compiler Classic (icc/icl) compilers	2-2
Updated Clang Compiler Support: Set up Polyspace analysis for code compiled by using Clang version 12.x	2-2
Modifying Checkers: Create list of macros to prohibit and check for use of macros from the list	2-3
Design Constraints: Specify constraints on function inputs for more precise analysis	2-3
Object Size Limitation Removed: Analyze code containing large data structures	2-4
Changes in analysis options and binaries	2-4
Analysis Results	2-6
AUTOSAR C++14 Support: Check for 370 AUTOSAR C++14 rules including 7 new rules	2-6
Bug Finder Checkers: Check for uncaught exceptions, dangling string_view, and other issues	2-6
CERT C++ Support: Check for issues arising from container iterators, memory management, and std::string objects	2-7
CWE Support: Updated CWE version and support for additional rules related to security, data flow, and other issues	2-8
Custom Coding Rules: Enforce naming convention for floating-point literals	2-9
Changes to coding standards checking	2-10
Updated Bug Finder defect checkers	2-18
Reviewing Results	2-20

Additional Info in Result Details: See expected values and underlying types in constant overflows	2-20
Inputs Causing Defect: See example system inputs that lead to non-initialized variables	2-20
Polyspace Access: Import review details and justifications from existing projects	2-21
polyspace-access Command: Manage review information and compare project runs	2-23
Changes in the polyspace-access Command Options	2-24
Polyspace Extension for Visual Studio Code (VSCode): Review findings in header files	2-25
Polyspace Extension for Visual Studio Code (VSCode): Autocomplete code annotations using catalog of predefined comments	2-26
Polyspace Extension for Visual Studio Code (VSCode): Get started faster with the Polyspace as You Code walkthroughs	2-27
Results Export: Updated color property when you export Code Metrics results to JSON SARIF format	2-27
Changes in number and locations of results showing conflicting declarations	2-28
Changes in locations of results in macro instances or template instantiations	2-28

R2022a

Analysis Setup	3-2
Updated Clang Compiler Support: Set up Polyspace analysis for code compiled by using Clang versions 6.x to 11.x	3-2
Incremental Compilation: Run faster analysis by compiling only files edited since previous analysis	3-3
Configuration from Build System: Import compiler macro definitions automatically without tracing build	3-3
Simulink Support: Polyspace updates generated code when model changes	3-4
MATLAB Coder Support: Polyspace analysis takes into account MATLAB Coder settings for nonfinite numbers	3-4
New Code Behaviors: Specify behaviors such as critical, memory-managing, or real-time to code elements to enable associated checks	3-5
Modifying Checkers: Create list of keywords to prohibit and check for use of keywords from the list	3-6
Ignoring Code Annotations: Perform a worst-case analysis to see all results including previously justified ones	3-6
Functionality Being Removed: Polyspace desktop integration with Eclipse IDE	3-7
Functionality Removed: Polyspace stubs for Standard Template Library	3-7
Functionality Removed: Compilation assistant	3-7
Changes in analysis options and binaries	3-8
Changes in MATLAB function, options object and properties	3-8
Analysis Results	3-10

AUTOSAR C++14 Support: Check for 363 AUTOSAR C++14 rules including 18 new rules	3-10
Bug Finder Checkers: Check for inefficient string methods, critical data leaks, and other issues	3-11
CERT C++ Support: Check for violations associated with exception handling	3-12
MISRA C++:2008 Support: Check for void functions with no external side effects	3-13
CWE Support: Check for violation of 8 new CWE rules about class access, exception, and other issues	3-13
Updated Bug Finder defect checkers	3-14
Changes to coding standards checking	3-15
Reviewing Results	3-23
Results Export: Generate more accurate keys to track results across analysis runs	3-23
Polyspace Extension for Visual Studio Code (VSCode): Track files to analyze and the status of analysis	3-24
Polyspace Extension for Visual Studio Code (VSCode): New views for configuration, results details, baseline	3-25
Polyspace Extension for Visual Studio Code (VSCode): Mass-justify findings with a single click	3-26
Polyspace Access: Redesign of UI dashboard design for consistency and efficiency	3-27
Polyspace Access: Improved performance when viewing aggregate data from large project folders	3-28
Polyspace Access: View code covered by verification in new graph	3-29
Polyspace Access Project Runs: Add labels to analysis runs that you upload to a project	3-29
polyspace-access Command: Assign SQO levels, move or delete a project, and view list of runs for a project	3-30
polyspace-access Command: Improved robustness and error diagnostics	3-31
Functionality Removed: Report generation from pre-R2015a results	3-32
Polyspace Access Installation	3-33
License Management: Use a single license to review Bug Finder, Code Prover, and Ada results in your web browser	3-33
Support for X.509 certificates generated without a SAN extension removed	3-34
Polyspace Extension for Eclipse: Specify path to IDE executable when installing extension on Eclipse-based IDEs	3-34
Changes in Polyspace Access docker containers	3-34

R2021b

Documentation	4-2
Documentation: View combined documentation for all Polyspace Bug Finder products	4-2

Documentation: View web documentation by default	4-3
Contextual Help: View contextual help in web browser	4-3
Analysis Setup	4-5
IAR Embedded Workbench Compiler: Set up Polyspace analysis for code compiled by using RISC-V target	4-5
Updated GCC Compiler Version Support: Set up Polyspace analysis for code compiled with GCC versions 9.x and 10.x	4-5
Simulink Support: Consistent C++ version in Polyspace and Simulink ...	4-6
C17 Support: Run Polyspace analysis on code that follows version C17 of C standard	4-6
Configuration from Build System: Copy console output to log file	4-6
Polyspace as You Code Configuration: Use script in Polyspace extension without specifying path of installation folder	4-7
Polyspace Extension for Visual Studio Code (VSCode): View information about the extension in the status bar	4-7
Polyspace Extension for Visual Studio Code (VSCode): Specify communication port and enable debugging	4-8
Polyspace Extension for Visual Studio Code (VSCode): Configure use of baseline information in fewer steps	4-9
Polyspace Extension for Visual Studio: Open project configuration by using fewer clicks	4-10
Polyspace Extension for Visual Studio: Extract build options more quickly from Visual Studio solution	4-11
Functionality Being Removed: Polyspace stubs for Standard Template Library	4-11
Functionality Being Removed: Compilation assistant	4-11
Changes in analysis options and binaries	4-12
Analysis Results	4-14
AUTOSAR C++14 Support: Check for 345 AUTOSAR C++14 rules including 18 new rules	4-14
Bug Finder Checkers: Check for inefficient C++ algorithms or function usage and other issues	4-15
CERT C++ Support: Check for violations associated with exception handling	4-16
Custom Rules: Check typedef statements that creates aliases for existing typedef aliases	4-17
Changes to coding standards checking	4-18
Updated Bug Finder defect checkers	4-30
Reviewing Results	4-32
Results in Macros: See results in macro expansions when macro parameters cause an issue	4-32
Additional Info in Result Details: See expected and actual values for numerical defects	4-33
Results Review: Open review history, select layout, and open additional panes by using fewer clicks	4-34
Results Review: View relevant information in review panes when you select a finding	4-34
Functionality Removed: Polyspace Metrics	4-34
Functionality Being Removed: Report generation from pre-R2015a results	4-35

Polyspace Access Installation	4-36
User Management: Set project permissions at the group level	4-36
User Management: Update list of users and groups more quickly by reloading web browser	4-36
User Authentication: Authenticate user logins against custom identities and LDAP identities simultaneously	4-36
Polyspace Access Services: Faster results uploads and more responsive source code view	4-37

R2021a

Analysis Setup	5-2
Bug Finder Analysis Engine for Single Source File : Run Polyspace as You Code analysis and view results in your IDE or code editor	5-2
Polyspace Extension for Visual Studio : Run Polyspace As You Code analysis and view results in Visual Studio IDE	5-2
Polyspace Extension for Visual Studio Code : Run Polyspace As You Code analysis and view results in Visual Studio Code code editor	5-3
Polyspace Extension for Eclipse : Run Polyspace As You Code analysis and view results in Eclipse IDE	5-4
Configuration from Build System: Specify options delimiter and suppress console output	5-5
Configuration from Build System: Improved detection of incompatible software	5-6
Updated GCC Compiler Support: Set up Polyspace analysis for code compiled with GCC version 8.x	5-6
Updated Microsoft Visual C++ Support: Set up a Polyspace analysis for code compiled with Visual Studio 2019	5-7
Modifying Checker Behavior: Modify parameters for MISRA C:2012 rules 1.1 and 5.1 to 5.5	5-7
Simulink Support: Start Polyspace analysis without an explicit code generation step	5-8
polyspacesetup Function : Integrate Polyspace with MATLAB in fewer steps	5-9
pslinkrunCrossRelease Function : Analyze code generated in an earlier release of Simulink by using a later release of Polyspace	5-9
Functionality being removed: Compilation assistant	5-9
Changes in analysis options and binaries	5-10
Analysis Results	5-11
AUTOSAR C++14 Support: Check for 327 AUTOSAR C++14 rules including 19 new rules in R2021a	5-11
CERT C++ Support: Check for memory management and programming rule violations.	5-12
MISRA C++:2008 Support: Check for disallowed pointer arithmetic	5-13
MISRA C:2012 Support: Checkers updated to account for MISRA C:2012 Technical Corrigendum 1 and Amendment 2	5-13
Guidelines: New checkers for software complexity defects	5-15

JSF AV C++ Support: Check for cases where pass-by-reference is preferred to pass-by-pointer	5-16
New Bug Finder Checkers: Check for inefficient string operations, noncompliance with AUTOSAR Standard specifications, and other issues	5-17
Changes to coding standards checking	5-18
Updated Bug Finder defect checkers	5-27
Reviewing Results	5-28
Simulink Block Annotation : Add multiple Polyspace annotations corresponding to multiple types of Polyspace results	5-28
Review Information Between Polyspace Access Projects : Download results from Polyspace Access and import review information to a different project	5-28
Results Review Scope : Define and share custom families of filters	5-28
Results Review Layout : Select view to prioritize review of code or results list	5-29
Code Quality Comparison Between Runs: Filter and view information for previous findings fixed in the current run	5-31
Polyspace Access Installation	5-32
License Management : Uploading of results to Polyspace Access no longer requires a license checkout	5-32
User Manager : Enable pagination when requesting large set of users from LDAP server	5-32
Bug Tracking Tool : Create Jira tickets for Jira projects that use single select custom fields	5-32
Admin Interface : Improved logging for Polyspace Access services	5-32

R2020b

Analysis Setup	6-2
Compiler Support: Set up Polyspace analysis for code compiled by Renesas SH C compilers	6-2
Cygwin Support: Create Polyspace projects automatically by using Cygwin 3.x build commands	6-2
C++17 Support: Run Polyspace analysis on code that has C++17 features	6-2
Modifying Checker Behavior: Check for non-initialized buffers when passed by pointer to certain functions	6-3
Configuration from Build System: Generate a project file or analysis options file by using a JSON compilation database	6-3
Configuration from Build System: Specify how Polyspace imports compiler macro definitions	6-4
Configuration from Build System: Compiler configuration cached from prior runs for improved performance	6-4
polyspacePackNGo Function : Generate and package Polyspace option files from a Simulink model	6-4

Polyspace and MATLAB Integration : Integrate Polyspace with MATLAB programmatically without user interaction	6-5
polyspace.ModelLinkOptions Object : Configure object to analyze code generated as a model reference	6-5
Offloading Analysis : Submit Polyspace analysis jobs from CI server to a dedicated analysis cluster	6-6
Offloading Analysis : Server-side errors reported back to client side	6-6
Changes in analysis options and binaries	6-6
Analysis Results	6-8
AUTOSAR C++14 Support: Check for 308 AUTOSAR C++14 rules including 61 new rules in R2020b	6-8
CERT C Support: Check for missing const-qualification and use of hardcoded numbers	6-13
CERT C++ Support: Check for exception handling issues, memory management problems, and other rule violations	6-14
MISRA C++:2008 Support: Check for commented out code, variables used once, exception handling issues, and other rule violations	6-14
JSF AV C++ Support: Check for commented out code and methods that can be inlined	6-15
MISRA C Support: Check for commented out code	6-15
New Bug Finder Defect Checkers: Check for post-C++11 defects such as problematic move operations, missing constexpr, and noexcept violations	6-16
Changes to coding standards checking	6-17
Updated Bug Finder defect checkers	6-30
Updated code metrics specifications	6-31
Reviewing Results	6-33
Results Export: Export Polyspace results to external formats such as SARIF JSON	6-33
Importing Review Information: Accept information in source or destination results folder in case of merge conflicts	6-33
Source Code Tooltips: Display information related to only the currently selected defect	6-34
Simulink Block Annotation : Annotate Simulink blocks from Polyspace user interface to justify Polyspace results	6-34
User Authentication : Use a credentials file to pass your Polyspace Access credentials at the command line	6-35
Code Quality Improvement Progress : Compare results from current run to previous runs and determine progress in code quality improvement	6-35
Code Quality Objectives : Define custom quality objectives definitions and apply them to specific projects	6-36
Source Code Tooltips : Display only information necessary to understand the selected defect	6-37
Project Selection : Find a project in the Project Explorer through a text filter	6-37
Functionality being removed: Polyspace Metrics	6-38
Polyspace Access Installation	6-39
Bug Tracking Tool : Integrate with Jira Software Cloud	6-39
Cluster Admin Settings: Validate values of settings on demand or on save	6-39

HTTPS Configuration : Configure services without specifying ports or SSL certificates	6-39
Functionality Replaced : Polyspace Access embedded LDAP	6-39
Changes in Polyspace Access docker containers, options, and binaries ..	6-40

R2020a

Analysis Setup	7-2
Compiler Support: Set up Polyspace analysis easily for code compiled with MPLAB XC8 C compilers	7-2
Compiler Support: Set up Polyspace analysis to emulate MPLAB XC16 and XC32 compilers	7-2
Source Code Encoding: Non-ASCII characters in source code analyzed and displayed without errors	7-2
Modifying Checkers: Create list of functions to prohibit and check for use of functions from the list	7-3
Simulink Support : Analyze custom C code in C Function blocks	7-3
Jenkins Support: Use sample Jenkins Pipeline script to run Polyspace as part of continuous delivery pipeline	7-3
Changes in analysis options and binaries	7-4
Changes in MATLAB functions, options object and properties	7-4
Analysis Results	7-5
Extending Checkers: Run stricter analysis that considers all possible values of system inputs	7-5
AUTOSAR C++14 Support: Check for 37 new rules related to lexical conventions, standard conversions, declarations, derived classes, special member functions, overloading and other groups	7-6
CERT C Support: Check for CERT C rules related to threads and hardcoded sensitive data, and recommendations related to macros and code formatting	7-9
CERT C++ Support: Check for CERT C++ rule related to hard coded sensitive data, order of initialization in constructor and other issues ..	7-10
CWE Support: Check for CWE rule related to incorrect block delimitation	7-10
New Bug Finder Defect Checkers: Check for possible performance bottlenecks, hardcoded sensitive data and other issues	7-11
Changes to coding standards checking	7-13
Updated Bug Finder defect checkers	7-14
Reviewing Results	7-16
Inputs Causing Defect: See example input values for numerical defects found with stricter analysis	7-16
Simulink Support: Navigate from generated code in Polyspace Access to blocks in model	7-17
Bug Tracking Tool Support : Create Redmine tickets for Polyspace Access results and assign to developers	7-17
Bug Tracking Tool Support : Manage tickets for multiple findings	7-18
Results Review : See review history of findings	7-19

Results Review : See the configuration options used for analysis	7-19
Code Quality Objectives : Customize thresholds used to track the quality of your code	7-21
Project Dashboard: Open results by clicking Dashboard charts	7-21
Exporting Results : Export only results that must be reviewed to satisfy software quality objectives (SQOs)	7-22
Report Generation : Configure report generator to communicate with Polyspace Access over HTTPS	7-22
Report Generation : Navigate to Polyspace Access Results List from report	7-23
Polyspace Access Installation	7-24
Installation and Configuration : New Issue Tracker service	7-24
Installation and Configuration : Change in default location of Polyspace Access data volume and working directories	7-24

R2019b

Analysis Setup	8-2
Compiler Support: Set up Polyspace analysis easily for code compiled with Cosmic compilers	8-2
Configuration from Build System: Compiler version automatically detected from build system	8-2
Simulink Support: Analyze generated code by using contextual buttons on the Simulink Editor toolstrip	8-3
Simulink Support: Verify custom code called from C Caller blocks and Stateflow charts in context of model	8-3
Simulink Support: Compare two Polyspace result sets and see the effect of changes in model or code generation parameters	8-5
Changes in MATLAB functions, options object and properties	8-5
Analysis Results	8-8
AUTOSAR C++14 Support: Check for misuse of lambda expressions, potential problems with enumerations, and other issues	8-8
CERT C++ Support: Check for pointer escape via lambda expressions, exceptions caught by value, use of bitwise operations for copying objects, and other issues	8-9
CERT C Support: Check for undefined behavior from successive joining or detaching of the same thread	8-9
New Bug Finder Defect Checkers: Check for new security vulnerabilities, multithreading issues, missing C++ overloads, and other issues	8-10
MISRA C:2012 Directive 4.12: Dynamic memory allocation shall not be used	8-11
Updated Bug Finder defect checkers	8-11
Reviewing Results	8-12
Code Annotations: Justify Bug Finder results by using annotations spread over multiple lines	8-12

Polyspace Access Installation	8-13
User Authentication: Use LDAP search filters to restrict number of users to authenticate	8-13
User Management : Update list of users from LDAP database or LDIF file	8-13

R2019a

Analysis Setup	9-2
Polyspace-only Licenses: Install Polyspace without MATLAB installation	9-2
New Polyspace Products Supporting Continuous Integration: Perform automated code analysis after code submission with Polyspace Bug Finder Server and Polyspace Bug Finder Access	9-2
Bug Finder Analysis Engine Separated from Viewer : Run Bug Finder analysis on server and view the results from multiple client machines	9-3
Continuous Integration Support: Run Bug Finder on server class computers with continuous upload to Polyspace Access web interface	9-4
Continuous Integration Support : Set up testing criteria based on Bug Finder static analysis results	9-6
Continuous Integration Support : Set up email notification with summary of Bug Finder results after analysis	9-6
Offloading Polyspace Analysis to Servers: Use Polyspace desktop products on client side and server products on server side	9-7
Collaborative Review Support: Upload results from Polyspace user interface to Polyspace Access web interface and share results using web links ..	9-9
Support for Security Standards: Check explicitly for subsets of CERT C, CERT C++ or ISO/IEC TS 17961 rules	9-11
Coding Standard Support: Enforce common standards across team or organization by reusing checker configuration	9-12
Compiler Support: Set up Polyspace analysis easily for code compiled with ARM v5 and v6 compilers	9-13
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	9-14
Simulink Toolstrip: Analyze generated code using contextual buttons in Simulink Editor	9-15
Changes in analysis options and binaries	9-16
Changes in MATLAB functions, options object and properties	9-18
Analysis Results	9-22
AUTOSAR C++14 Support: Check for violations of rules from the AUTOSAR C++14 coding standard	9-22
Improved CERT C++ Support: Check for missing overloads, ambiguous declaration syntax and other rules from CERT C++ Coding Standard	9-22
Recursion Detection: See list of recursion cycles in C/C++ project	9-23

New Bug Finder Defect Checkers: Check for misplaced CV qualifiers, C++ most vexing parse, ill-constructed variadic functions, and other issues	9-24
Updated code metrics specifications	9-24
Updated Bug Finder defect checkers	9-27
Reviewing Results	9-29
Support for Security Standards: See CERT C, CERT C++ or ISO/IEC TS 17961 rule violations explicitly in Polyspace analysis results and reports	9-29
Bug Fix Suggestions: See possible fixes for types of defects found by Bug Finder	9-30
Source Code Navigation: Keep result pinned while navigating through source code	9-30
Report Generation: Generate Polyspace reports faster than previous releases	9-32
Report Generation: Generate single file for HTML reports	9-32
Project Dashboard : Track progress of code quality via Polyspace results	9-32
Collaborative Review Support : Review Polyspace Bug Finder results and source code in web browser	9-34
Collaborative Review Support : Share Polyspace Bug Finder results using web links	9-35
Project Authorization Management: Create and enforce authorization policies for access to project	9-35
Bug Tracking Tool Support : Create JIRA issues for Polyspace Bug Finder results	9-36

R2018b

Analysis Setup	10-2
Configuration from Build System: Automatically generate Polyspace configuration modules from build system	10-2
C11 and C++14 Support: Run Polyspace analysis on code with C11 or C++14 features	10-2
Autodetection of Concurrency Primitives: Multitasking model detected from C11 multithreading functions	10-3
Compiler Support: Set up Polyspace analysis easily for code compiled with Renesas compilers	10-3
Changes in analysis options and binaries	10-4
Changes in MATLAB option object properties and option values	10-5
Analysis Results	10-7
CERT C++ Support: Identify CERT C++ violations by using defect checkers and coding rules	10-7
Improved CERT C Support: Check for precision loss, blocking operations, and other rules from the CERT C Coding Standard	10-8
Constant Overflows: Check for overflows on integer constants	10-9
Updated Bug Finder defect checkers	10-9

Changes to coding standards checking	10-10
Reviewing Results	10-11
Function Call Hierarchy: View call tree of functions in source code	10-11
Header Files Access: Open your project header files directly from the point of inclusion	10-11

R2018a

Analysis Setup	11-2
AUTOSAR Support: Set up Polyspace multitasking configuration automatically from an AUTOSAR description	11-2
MATLAB Coder Support: Run Polyspace on C/C++ code generated from MATLAB code without additional setup	11-2
Compiler Support: Set up Polyspace analysis easily for code compiled with Texas Instruments, IAR or CodeWarrior compilers	11-3
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	11-4
Configuration from Build System: Include or exclude sources when generating Polyspace project using polyspace-configure	11-5
Support for IBM Rational Rhapsody to be removed	11-5
Changes in analysis options and binaries	11-5
Changes in MATLAB option object properties	11-8
Analysis Results	11-11
CERT C Support: Check for information leakage, invalid environment pointers, and other rules from the CERT C Coding Standard	11-11
Cryptography Checkers: Check for security vulnerabilities such as incorrect use of public key cryptography routines	11-12
MISRA C++ Support: Check for overriding of standard library functions, missing const qualifiers, and other MISRA C++ rules	11-13
MISRA C:2012 Directive 4.8: Detect opportunities for data hiding	11-14
Rule for Source Line Length: Constrain number of characters per line in your code	11-14
Improved Fast Analysis: Find some multi-file MISRA C violations in fast analysis	11-14
Reviewing Results	11-15
Concurrency Modeling: View all tasks and interrupts extracted from code and Polyspace configuration in one view	11-15
Data Races: Distinguish write-write conflicts from more benign read-write conflicts	11-16

Analysis Setup	12-2
Green Hills Compiler Support: Set up Polyspace analysis easily for code compiled with Green Hills MULTI Compiler	12-2
OSEK Multitasking Support: Detect the multitasking configuration for your OSEK application automatically	12-2
Incremental Analysis in Eclipse: Detect bugs as you type and save code in your Eclipse IDE	12-3
Polyspace API in MATLAB: Configure analysis, run analysis, and read analysis results with a single MATLAB object	12-3
Compiler-Specific Keywords: Nonstandard compiler-specific keywords are only supported when you specify compiler	12-5
POSIX and BSD Standards: Use functions from these standards without additional setup	12-5
Changes in analysis options and binaries	12-5
Analysis Results	12-8
Security Standards Support: Detect violations of all secure coding guidelines from ISO/IEC Technical Specification 17961:2013 and more guidelines from SEI CERT C Coding Standard	12-8
MISRA C:2012 Directive 1.1: Detect instances of implementation-specific behavior in your code	12-9
Changes to coding rule checking	12-9
Reviewing Results	12-11
Result Review Workflow: Hide results that you reviewed once and justified through source code annotations	12-11
Code Annotations: Justify results or define your own format with a new annotation format	12-12
MISRA Comments and Code Annotations: Import your existing MISRA C:2004 justifications to MISRA C:2012 results	12-12
Results Review Workflow: Sort and filter results by subtype	12-13
Constraint Specification: Navigate easily to the constraint specification interface for Bug Finder results	12-14
Result Status: Assign statuses that directly correspond to stages of development workflow	12-15

Analysis Setup	13-2
Unified User Interface: Create and maintain a single Polyspace project for Bug Finder and Code Prover analysis	13-2
Easier Compliance with Security Standards: Choose CWE, CERT C99, or ISO/IEC TS 17961 coding standard and address corresponding violations through Polyspace results and security reports	13-5

Incremental Analysis of Specific Checks: Analyze only files edited since previous analysis to quickly find new defects and coding rule violations	13-6
TASKING Compiler Support: Set up Polyspace analysis easily for code compiled with Altium TASKING compiler	13-7
Updated Visual C++ Support: Set up Polyspace analysis easily for code compiled with Microsoft Visual C++ 2015 compiler	13-7
Autodetection of Concurrency Primitives: Multitasking model detected from Windows, μ C/OS II or C++11 multithreading functions	13-7
Autodetection of Concurrency Primitives: Map Unsupported Thread Creation Functions to Supported Functions	13-8
Manual Multitasking Setup: Specify routines that disable and reenable all interrupts	13-9
Specifying Function Names for Options: Choose from prepopulated list in user interface instead of entering manually	13-10
Polyspace API in MATLAB: Create MATLAB objects from Polyspace projects to run analysis	13-11
Support for 128-bit variables	13-12
Improvement in automatic project creation from build systems	13-12
Changes in analysis options and binaries	13-12
Changes in MATLAB option object properties	13-15
Change in temporary folder location	13-16
Analysis Results	13-17
Additional Defect Checkers for Security: Check for security vulnerabilities such as incorrect use of cryptographic routines	13-17
MISRA Amendment Support: Check your code for new security guidelines in MISRA C:2012 Amendment 1	13-19
New Code Metrics: See number of lines in header files and number of local variables per function	13-20
Changes to coding rule checking	13-20
Reviewing Results	13-22
Folder Names in Results: Filter or organize analysis results by source folder names	13-22
Code to Model Traceability: Switch easily between identifiers in generated code and corresponding blocks in model	13-22
Polyspace API in MATLAB: Read Polyspace analysis results from MATLAB	13-24
Double Lock and Other Concurrency Defects: Get help investigating the defects using detailed control flow information	13-24
Spreadsheet of Checkers: Use spreadsheet to keep track of checkers that you enable	13-25

R2016b

Analysis Setup	14-2
Diab Compiler Support: Set up Polyspace analysis easily for code compiled with Wind River Diab compiler	14-2

Multitasking Code Analysis Setup: Specify cyclic tasks and nonpreemptable interrupts directly as analysis options	14-2
Improved source and include folder management	14-2
Writable Examples: Modify example projects and restore original versions	14-3
Run analysis on .psprj file from the command line	14-3
Support for local threads	14-3
Polyspace API in MATLAB: Configure and run Polyspace using MATLAB objects	14-4
Configuration Parameters Help: View descriptions of Polyspace options in Simulink configuration parameters	14-4
Eclipse Build Support: Set up Polyspace analysis from Eclipse build command	14-5
Visual Studio 2010 add-in support to be removed from installation	14-5
Support for Rhapsody 8.1	14-5
DOS Mode Warning on Linux: Compilation warning for DOS inconsistencies	14-5
Faster Restart for Remote Verification: Reuse compilation results from a previous analysis	14-6
Changes in Target & Compiler analysis options	14-6
Changes in analysis options and binaries	14-7
Analysis Results	14-9
CERT C Support: Identify CERT C violations using defect checkers and coding rules	14-9
Local Variable Size Estimation: Find total size of local variables in a function	14-10
Metrics for C++ Templates: View code complexity metrics for instances of C++ templates	14-11
Changes to coding rule checking	14-11
Updated Bug Finder defect checkers	14-12
Reviewing Results	14-14
Data Race Graphs: Fix data race defects easily using graphical view of function call sequence	14-14
Interactive Graphical Display: Click graphs on Dashboard to filter results	14-14
Event History for Coding Rules: Navigate easily between two locations in code that together cause a rule violation	14-15
Results in Macros Consolidated: View coding rule violations and defects on macro definitions instead of macro instances	14-15
Analysis Objectives in Eclipse: Create review scopes to focus your review	14-15
Filtered Report: Reuse result filters for generated report	14-16
Results Export: Export results to text file for computing graphs and statistics	14-16
Coding Rules in Report: View improved presentation of coding rules violations in report	14-16
English Reports in Non-English Locales: Generate English reports on operating systems with a different language	14-17
Change in report template location	14-17
Improved PDF Report Generation	14-17
Changes in Polyspace User Interface	14-17

Analysis Setup	15-2
Files to Review: Generate results for only specified files and folders	15-2
Faster MISRA Checking: Check coding rules more quickly and efficiently	15-2
S-Function Analysis: Launch analysis of S-Function code from Simulink	15-2
Import signal ranges from model for generated code analysis	15-3
Polyspace Metrics Tomcat Upgrade: Use upgraded default Tomcat server or custom Tomcat version	15-3
Polyspace Metrics Interface Updated: View project and metrics summary and defect impact	15-3
Source Code Search: Search huge applications more quickly	15-3
Default Layouts: Switch easily between project setup and results review in user interface	15-4
Files Not Compiled: Receive alerts about compilation errors in dashboard and reports	15-4
Project Language Flexibility: Change your project language at any time	15-4
Improvements in automatic project creation from build command	15-4
Polyspace TargetLink plug-in supports data from structures	15-5
Changes in analysis options	15-5
Analysis Results	15-7
Improvements to defect checkers	15-7
Improvements in checking of previously supported MISRA C rules	15-7
Standards Mapped to Defects: Observe coding standards using Polyspace Bug Finder	15-8
Reviewing Results	15-9
More results available in real time	15-9
Autocompletion for Review Comments: Partially type previous comment to select complete comment	15-9
Persistent Filter States: Apply filters once and view filtered results across multiple runs	15-9
Polyspace Eclipse plug-in results location moved	15-9

Bug Fixes

Analysis Setup	17-2
Mixed C/C++ Code: Run analysis on entire project with C and C++ source files	17-2
Autodetection of Multitasking Primitives: Analyze source code with multitasking primitives from POSIX and VxWorks without manual setup	17-2
Microsoft Visual C++ 2013: Analyze code developed in Microsoft Visual C++ 2013	17-3
GNU 4.9 and Clang 3.5 Support: Analyze code compiled with GNU 4.9 or Clang 3.5	17-3
Improvements to automatic project creation from build command	17-3
Start Page: Get oriented with Polyspace Bug Finder	17-4
Saved Layouts: Save your preferred layouts of the Polyspace user interface	17-4
Renaming of labels in Polyspace user interface	17-5
Including options multiple times	17-5
Updated Support for TargetLink	17-6
Changes in analysis options	17-6
Binaries removed	17-8
Support for Visual Studio 2008 to be removed	17-8
Import Visual Studio project removed	17-9
Analysis Results	17-10
More Defect Categories: Detect security vulnerabilities, resource management issues, object oriented design issues	17-10
Complete MISRA C:2012 Support: Detect violations of all MISRA C:2012 rules	17-10
Improvements in checking of previously supported MISRA C rules	17-11
Changes to Bug Finder Defects	17-12
Reviewing Results	17-19
Results in Real Time: View results as they are produced	17-19
Improved Eclipse Support: View results embedded in source code and context-sensitive help	17-19
Defects Classified by Impact: Prioritize defect review by using the impact attribute assigned to each defect type	17-20
Improved Review Capability: View result details and add review comments in one window	17-20
Enhanced Review Scope: Filter coding rule violations from display in one click	17-21
Configuration Associated with Result Not Opened by Default	17-21
Improvements in Report Templates	17-21
XML and RTF report formats removed	17-21

Analysis Setup	18-2
Simplified workflow for project setup and results review with a unified user interface	18-2
Search improvements in the user interface	18-2
Option to specify program termination functions	18-3
Support for GCC 4.8	18-3
Polyspace plug-in for Simulink improvements	18-3
Polyspace binaries being removed	18-4
Import Visual Studio project being removed	18-4
Analysis Results	18-5
Changes to Bug Finder defects	18-5
Improvements in coding rules checking	18-5
Reviewing Results	18-7
Code complexity metrics available in user interface	18-7
Context-sensitive help for code complexity metrics, MISRA-C:2012, and custom coding rules	18-7
Review of latest results compared to the last run	18-7
Simplified results infrastructure	18-7
Default statuses to justify results	18-8
Filters to limit display of results	18-8

Analysis Setup	19-2
Parallel compilation for faster analysis	19-2
Support for Mac OS	19-2
Support for C++11	19-2
Code editor in Polyspace interface	19-2
Ignore files and folders during analysis	19-2
Simulink plug-in support for custom project files	19-3
TargetLink support updated	19-3
AUTOSAR support added	19-3
Remote launcher and queue manager renamed	19-3
Improved global menu in user interface	19-4
Improved Project Manager perspective	19-4
Polyspace binaries being removed	19-4
Import Visual Studio project being removed	19-5
Analysis Results	19-6
Support for MISRA C:2012	19-6

Additional concurrency issue detection (deadlocks, double locks, and others)	19-6
New and updated defect checkers	19-7
Reviewing Results	19-9
Context-sensitive help for analysis options and defects	19-9
Improved Results Manager perspective	19-9
Error mode removed from coding rules checking	19-9

R2014a

Analysis Setup	20-2
Automatic project setup from build systems	20-2
Support for GNU 4.7 and Microsoft Visual Studio C++ 2012 dialects	20-2
Simplification of coding rules checking	20-2
Preferences file moved	20-3
Security level support for batch analysis	20-4
Interactive mode for remote analysis	20-4
Default text editor	20-4
Support for Windows 8 and Windows Server 2012	20-4
Function replacement in Simulink plug-in	20-4
Check model configuration automatically before analysis	20-5
Data range specification support	20-5
Polyspace binaries being removed	20-5
Analysis Results	20-7
Classification of bugs according to the Common Weakness Enumeration (CWE) standard	20-7
Additional coding rules support (MISRA-C:2004 Rule 18.2, MISRA-C++ Rule 5-0-11)	20-7
Additional analysis checkers	20-7
Improvement of floating point precision	20-7
Reviewing Results	20-8
Results folder appearance in Project Browser	20-8
Results manager improvements	20-9
Additional back-to-model support for Simulink plug-in	20-10

R2013b

Analysis Setup	21-2
Introduction of Polyspace Bug Finder	21-2
Fast analysis of large code bases	21-2

Eclipse integration	21-2
Analysis Results	21-3
Detection of run-time errors, data flow problems, and other defects in C and C++ code	21-3
Compliance checking for MISRA-C:2004, MISRA-C++:2008, JSF++, and custom naming conventions	21-3
Cyclomatic complexity and other code metrics	21-3
Reviewing Results	21-4
Traceability of code analysis results to Simulink models	21-4
Access to Polyspace Code Prover results	21-4

R2023a

Version: 3.8

New Features

Bug Fixes

Compatibility Considerations

Analysis Setup

Project File Classification: Control precisely which files to include in analysis and how to analyze them

In R2023a, you can define file sets in your project that need specific treatment during analysis. For instance, you might want to skip the definitions of function bodies in third-party libraries or force analysis of all functions in files that you own. You can enumerate file sets with specific behaviors in a classification XML file and fine-tune the Bug Finder analysis using this classification file.

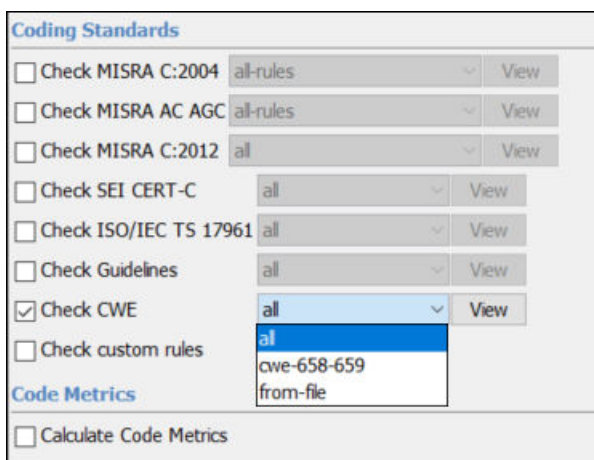
See also:

- `-classification`
- “Classify Project Files Into File Sets for Precise Control of Bug Finder Analysis”

Security Standards Support: Check explicitly for subsets of CWE rules

In R2023a, you can explicitly check for violations of Common Weakness Enumeration (CWE) rules by selecting the CWE rules that you want to enable during your analysis. To check for CWE rule violations, choose from these predefined subsets or create a custom subset that meets your requirements:

- `all` — All supported CWE rules.
- `cwe-658-659` — CWE rules for weaknesses that are specific to C and C++ Software.
- `from-file` — Custom selection of CWE rules that you configure with the **Checkers selection** interface. See “Customize Checker Subsets”.



See also:

- Check CWE (`-cwe`)
- “Common Weakness Enumeration (CWE)”

Products: Polyspace Bug Finder, Polyspace Bug Finder Server™, Polyspace Access™ (Polyspace as You Code).

Compatibility Considerations

In previous releases, you checked for violations of CWE rules by enabling defect checkers that were mapped to CWE rules (`-checkers CWE`). In R2023a, to check for violations of CWE rules, use the option `Check CWE (-cwe)`.

The desktop interface automatically updates your project configuration by replacing the option `-checkers CWE` with the option `and -cwe all`.

To update your scripts, replace all instances of `-checkers CWE` with `-cwe all`. Alternatively, you can enable only a subset of CWE rule checkers instead of all CWE checkers.

Updated Clang Compiler Support: Set up Polyspace analysis for code compiled by using Clang version 13.x

In R2023a, Polyspace supports Clang compiler version 13.x natively. If you build your source code by using this Clang compiler version, you can specify the compiler name for your Polyspace analysis.

You can now set up a Polyspace project without knowing the internal workings of this Clang compiler version. The analysis can interpret macros that are implicitly defined by the compiler and compiler-specific language extensions, such as keywords and pragmas.

Target Environment	
Compiler	clang13.x
Target processor type	x86_64

At the command line, specify a compiler by using the option `-compiler`. For instance:

```
polyspace-bug-finder-server -sources file.c -compiler clang13.x ...
```

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

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Updated GCC Compiler Support: Set up Polyspace analysis for code compiled by using GCC versions 11.x and 12.x

In R2023a, Polyspace supports GCC compiler versions 11.x and 12.x natively. If you build your source code by using these GCC compiler versions, you can specify the compiler name for your Polyspace analysis.

You can now set up a Polyspace project without knowing the internal workings of these compiler versions. The analysis can interpret macros that are implicitly defined by the compiler and compiler-specific language extensions, such as keywords and pragmas.

Target Environment	
Compiler	gnu11.x
Target processor type	x86_64

Target Environment	
Compiler	gnu12.x
Target processor type	x86_64

At the command line, specify a compiler by using the option `-compiler`. For instance:

```
polyspace-bug-finder-server -sources file.c -compiler gnu12.x ...
```

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

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Configuration from Build System: Import compiler includes automatically without tracing build

In R2023a, Polyspace can import the include folder paths of compilers that provide native options to list the compiler includes, for instance `gcc -Xpreprocessor -v`. Polyspace passes the relevant options to the compiler and extracts the paths of the include folders from the output without tracing your build.

With this new import strategy, you can use `polyspace-configure` with a JSON compilation database on Mac OS to create a Polyspace project or to generate an analysis options file from your build system without specifying the compiler includes.

Previously, because the System Integrity Protection (SIP) on Mac OS prevents Polyspace from tracing your build, you specified the compiler includes manually.

See also `polyspace-configure`.

Product: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code).

Target Alignment: New default alignment for target x86_64 and alignment up to 128 bits available for supported targets

In R2023a, the default alignment for target `x86_64` is now 128 bits. Previously, the default alignment for this target was 64 bits. See `Target processor type (-target)`.

In addition, you can now specify that the struct or array objects of a target align with the 128, 64, 32, 16, or 8-bit boundary for any target that Polyspace supports. Use the option `-align` to specify the target alignment. See `Generic target options`. Previously, you could specify only a subset of alignments for each target.

Products: Polyspace Bug Finder, Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code).

Compilation of Large Projects: Improved performance and scalability when compiling large projects

Polyspace analyses now run faster because, during the compilation phase, the storage of information for each translation unit happens in parallel. The parallelization results in performance and scalability improvements, especially during the compilation of large projects. For instance, on a machine that uses an Intel Xeon W-2133 processor with 64GB of RAM, the compilation time is up to 50% faster.

This new behavior is enabled by default and can increase the disk usage during the analysis.

The faster compilation is disabled when you use fast analysis. See `Use fast analysis mode for Bug Finder (-fast-analysis)`.

Products: Polyspace Bug Finder, Polyspace Bug Finder Server.

polyspace-access Command: View release and license information, upload from file path, view URL of uploaded results

In R2023a, `polyspace-access` has these new options and commands behaviors that simplify the management of the Polyspace Access software and results.

- New option `-ver`


Display the release version and license number of a Polyspace Access instance at the command line. Use option `-host` to specify the hostname of the Polyspace Access server. For example:

```
polyspace-access -ver -host myAccessServer $LOGIN
Connecting to https://myAccessServer:9443
Connecting as jsmith
-----
polyspace-access command version: R2023a
Polyspace Access Server version: R2023a
Polyspace Access license number: 12345678
-----
Command Completed
```

The command returns:

- The Polyspace Access server version and license number.
- The version of the `polyspace-access` binary.

Here, `$LOGIN` is a variable that stores the login credentials and other connection information. See “Encrypt Password and Store Login Options in a Variable”

Previously, you could not display this information at the command line. To view the version and license number you logged into Polyspace Access and clicked  > About Polyspace.

- Improved `-upload` command
 - You can now specify the path of a results file when you upload results to Polyspace Access, for example `-upload "C:\Polyspace_Workspace\My Example Project \Module1\ps_results.psbf"`

Previously, you could specify only the path of a folder containing a results file, or the path of a zipped file containing a results file.

- View URL of update results — You can now view the Polyspace Access URL for the results that you upload with the command `polyspace-access -upload`. The URL is available in the command output, for example:

```
polyspace-access %login% -upload C:\myProject\resultsDir\ps_results.psbf -parent-project public/myExample
Zipping results
Connecting to https://myAccessServer:9443
Connecting as jsmith
Upload C:\myProject\resultsDir\ps_results.psbf in public/myExample/PolyspaceProject (Bug Finder)
Upload with IMPORT_ID 1673331466966_6517d659-769c-4233-9e13-c505781c962c.zip
Uploading RunId: 1241 Progress: 10%
Uploading RunId: 1241 Progress: 95%
Upload successful for RUN_ID 1241 and PROJECT_ID 483
You can now view the results via your web browser at
ACCESS_URL https://myAccessServer:9443/metrics/index.html?a=metrics&p=483&r=1241
Upload completed for 1 / 1 runs
Command Completed
```

Use the URL to open the uploaded results in Polyspace Access.

See also `polyspace-access`.

Products: Polyspace Bug Finder, Polyspace Bug Finder Server, Polyspace Access.

Changes in analysis options and binaries

The value `CWE` for the option `Find defects (-checkers -disable-checkers)` will be removed in a future release

Warns

The value `CWE` for the option `Find defects (-checkers -disable-checkers)` will be removed in a future release. If you use the value `CWE`, Polyspace emits a warning and continues the analysis using the value `all` instead. To check your code for `CWE` violations, use the option `Check CWE (-cwe)` instead.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

Option `-function-behavior-specifications` removed

Errors

The option `-function-behavior-specifications` was previously renamed to `-code-behavior-specifications`. Starting in R2023a, the old option has been removed.

If you were using the option `-function-behavior-specifications`, use `-code-behavior-specifications` instead. See also `-code-behavior-specifications`.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

Changes in MATLAB function, options object and properties

CheckersPreset option value `CWE` is removed

Errors

The option `CWE` for property `CheckersPreset` is removed. To update your MATLAB® code, see this table.

```
opts=polyspace.Options;
```

Property	Instead Use
opts.Configuration.BugFinderAnalysis.CheckerOptions.Cwe	opts.CodingRulesCodeMetrics.EnableCwe=1; opts.CodingRulesCodeMetrics.Cwe='all';

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

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

Analysis Results

Bug Finder Checkers: Check for duplicate code, copy-paste errors, and related issues

In R2023a, using these Bug Finder checkers, you can find opportunities for refactoring your code into more maintainable modules.

Defect	Description	Defect Group
Duplicated code	A section of code is duplicated in other places.	Good Practice
Partially duplicated code	A section of code is duplicated in other places with very minor changes.	Good Practice
Possible copy-paste error	A section of code is duplicated in other places with exactly one minor change.	Programming

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

Bug Finder Checkers: Check for security vulnerabilities such as SQL injection or LDAP injection

In R2023a, using these Bug Finder checkers, you can find security vulnerabilities in C/C++ code that queries possibly secure databases.

Defect	Description	Defect Group
LDAP injection	Data read from an untrusted source is used in the construction of an SQL query	Security
SQL injection	Data read from an untrusted source is used in the construction of an LDAP query	Security

For the full list of security-related checkers, see “Security Defects”. See also “Bug Finder Defect Groups”.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

Bug Finder Checkers: Check for infinite loops, reference to unnamed temporaries, and other issues

In R2023a, using new Bug Finder checkers, you can check for these additional issues.

Defect	Description	Defect Group
Infinite loop	Loop termination condition might never be satisfied	Data flow
Reference to un-named temporary	A local reference is declared by using an unnamed temporary variable that is returned by value	Good Practice
Unnecessary construction before reassignment	Instead of directly constructing an object with a value, you construct an object and then immediately assign a value to it, resulting in inefficient code	Performance
Unnecessary implementation of a special member function	Implementing a special member function when the implicit version of the function in equivalent hinders code optimization, resulting in inefficient code	Performance

For the full list of checkers, see “Defects”. See also “Bug Finder Defect Groups”.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

CERT C++ Support: Check for pointer arithmetic on polymorphic objects and other issues

In R2023a, you can look for violations of these CERT C++ rules in addition to previously supported rules.

CERT C++ Rule	Description	Polyspace Checker
CTR56-CPP	Do not use pointer arithmetic on polymorphic objects	CERT C++: CTR56-CPP

See also “CERT C++ Rules”.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

MISRA C++:2008 Support: Check for issues caused by unused variables and enum types

In R2023a, you can look for violations of these MISRA™ C++:2008 rules in addition to previously supported rules.

MISRA C++:2008 Rule	Description	Polyspace Checker
0-1-6	A project shall not contain instances of non-volatile variables being given values that are never subsequently used.	MISRA C++:2008 Rule 0-1-6
7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	MISRA C++:2008 Rule 7-2-1

See also “MISRA C++:2008 Rules”.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

CWE Support: Updated CWE version and support for additional rules

In R2023a, Polyspace supports version 4.9 of the CWE standard. In this release, you can check for violations of this CWE rule in addition to previously supported rules.

CWE Rule	Description	Polyspace Checkers
CWE Rule 1071	Empty code block	CWE Rule 1071

For all CWE rules supported with Polyspace Bug Finder, see “Common Weakness Enumeration (CWE)”.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

Updated Bug Finder defect checkers

Defect	Description	Update
Expensive construction of <code>std::string</code> or <code>std::regex</code> from constant string	A constant <code>std::string</code> or <code>std::regex</code> object is constructed from constant data, resulting in inefficient code	The defect checker reports a defect if you construct an <code>std::regex</code> object by using constant data, such as a string literal or the output of a <code>constexpr</code> function. Previously, this defect was reported on <code>std::string</code> objects only. Now the defect checker supports checking both <code>std::string</code> and <code>std::regex</code> objects for this defect.

Defect	Description	Update
Expensive pass by value	Parameter might be expensive to copy	<p>The defect checker no longer reports a defect when you pass a parameter by value to a <code>virtual</code> method in these instances:</p> <ul style="list-style-type: none"> The method is an overriding method in a derived class. The method is a base class method that takes a <code>non-const</code> parameter. <p>The defect checker still reports a defect when you pass a <code>const</code> parameter by value to <code>virtual</code> method in a base class.</p>
Expensive use of a standard algorithm when a more efficient method exists	Functions from the algorithm library are misused with inappropriate inputs, resulting in inefficient code	<p>The defect checker:</p> <ul style="list-style-type: none"> Reports a defect if you pass an expensive-to-copy functor to a standard algorithm function. Because standard algorithms copy these functors, passing expensive functors to these functions results in inefficient code. Supports standard algorithm function overloads that accommodate <code>std::execution</code> classes.
Hard-coded object size used to manipulate memory	Memory manipulation with hard-coded size instead of <code>sizeof</code>	The defect checker reports a defect if you hard-code the memory size argument of memory manipulation functions such as <code>memcpy</code> , <code>memmove</code> , <code>memcmp</code> , or <code>memset</code> .
Missing <code>constexpr</code> specifier	<code>constexpr</code> specifier can be used on variable or function for compile-time evaluation.	<p>The defect checker:</p> <ul style="list-style-type: none"> No longer does a blanket reporting of variables initialized with <code>constexpr</code> constructors as candidates for <code>constexpr</code>-qualification. The checker now considers the constructor arguments. For instance, if the constructors take arguments that can be evaluated only at runtime, variables initialized with those constructors are not appropriate for <code>constexpr</code>-qualification. Does not report a pointer to a buffer as candidate for <code>constexpr</code>-qualification if it is assigned to another pointer and the buffer written via the second pointer.
Misuse of <code>readlink()</code>	Third argument of <code>readlink</code> does not leave space for null terminator in buffer	The defect checker now reports a Misuse of <code>readlink()</code> even when the value of the variable passed as a third argument of <code>readlink()</code> is unknown, as long as that same variable is also used to dynamically allocate the second argument of <code>readlink()</code> .

Defect	Description	Update
Noexcept function might exit with an exception	Functions specified as <code>noexcept</code> , <code>noexcept(true)</code> or <code>noexcept(<true condition>)</code> exits with an exception, which causes abnormal termination of program execution, leading to resource leak and security vulnerability	A compiler typically generates functions from constructs such as lambdas. The defect checker no longer reports this defect on the compiler generated functions.
Resource leak	File stream not closed before FILE pointer scope ends or pointer is reassigned	The defect checker covers file opening and closing using the <code>open()</code> and <code>close()</code> functions.
Unmodified variable not const-qualified	Variable not const-qualified but variable value not modified during lifetime	The defect checker no longer suggests <code>const</code> qualification for a variable if a reference to the variable is assigned to another variable through a ternary operator (and the later variable is modified). Such a variable cannot be <code>const</code> qualified.
Wrong type used in sizeof	<code>sizeof</code> argument does not match pointed type	The defect checker can now parse more complex expressions passed as argument to the <code>sizeof</code> operator and find more bugs related to incorrect types.
Useless include	An include directive is present but not used	The defect checker no longer reports a violation when you use the header file <code>stdbool.h</code> , even when your code does not contain <code>bool</code> type variables.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access

Compatibility Considerations

If you previously checked your code by enabling these defect checkers, you might see a change in the number of defects.

Changes to coding standards checking

In R2023a, these changes have been made to the checking of previously supported rules.

AUTOSAR C++14

Rule	Description	Change
AUTOSAR C++14 Rule A0-1-2	The value returned by a function having a non-void return type that is not an overloaded operator shall be used	<p>The rule checker reports violations of this rule if:</p> <ul style="list-style-type: none"> A function returns an instantiation of a <code>void</code> overloaded template and the returned object remains unused. In this code, the function <code>foo()</code> returns an object generated from the <code>void</code> overloaded template <code>object</code>. Because the returned object is not used, Polyspace reports a violation. <pre>template<typename T, typename I = int> class object{}; template<typename I> class object<void,I> final{ object(){} ~object();}; object<void> foo(); void bar(){ foo(); //Noncompliant } </pre> <ul style="list-style-type: none"> A function returns a reference which remains unused. For instance,
AUTOSAR C++14 Rule A0-1-6	There should be no unused type declarations	The rule checker no longer reports a violation on <code>public</code> unused types defined in a template.
AUTOSAR C++14 Rule M0-2-1	An object shall not be assigned to an overlapping object.	The rule checker no longer reports a violation in situations where a <code>this</code> pointer in a class or structure is assigned and does not have overlapping memory.
AUTOSAR C++14 Rule M2-10-1	Different identifiers shall be typographically unambiguous	<p>The rule checker no longer reports duplicate violations in templates. Consider this code:</p> <pre>template <class TT> void foo(){} template <class TT> void Foo(){} // Noncompliant </pre> <p>From R2023a, Polyspace reports a violation on the template function <code>foo()</code>. Previously, Polyspace reported violations on both the template and the function.</p>

Rule	Description	Change
AUTOSAR C++14 Rule A2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2014 shall be used	The rule checker does not report a violation of this rule on the sequences <code>\u</code> and <code>\U</code> .
AUTOSAR C++14 Rule A3-1-5	A function definition shall only be placed in a class definition if (1) the function is intended to be inlined (2) it is a member function template (3) it is a member function of a class template	The rule checker does not report a violation of this rule for <code>constexpr</code> and <code>constexpr</code> functions.
AUTOSAR C++14 Rule A5-1-7	A lambda shall not be an operand to <code>decltype</code> or <code>typeid</code>	The rule checker does not report a violation of this rule if you use <code>decltype</code> .
AUTOSAR C++14 Rule A5-2-6	The operands of a logical <code>&&</code> or <code> </code> shall be parenthesized if the operands contain binary operators.	The rule checker no longer reports a violation of this rule for a constant expression (<code>constexpr</code>) when you correctly parenthesize the operands of <code>&&</code> or <code> </code> in the expression.

Rule	Description	Change
AUTOSAR C++14 Rule A7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration	The rule checker checks for violations of this rule only for non-scoped enumerated types.
AUTOSAR C++14 Rule A8-5-0	All memory shall be initialized before it is read	The rule checker reports a violation of this rule if these issues occur: <ul style="list-style-type: none">• Non-initialized variable• Non-initialized pointer• Member not initialized in constructor

Rule	Description	Change
AUTOSAR C++14 Rule M9-3-3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.	<p>The rule checker no longer requires <code>const</code>-qualification for a member functions if the function passes a data member to a constructor that might possibly modify the data member. For instance, in this example:</p> <ul style="list-style-type: none"> • The member function <code>TestClass::Test1()</code> passes a data member <code>mMutex</code> to a constructor that takes non-<code>const</code> references (there is no constructor overload that takes <code>const</code> references). Therefore, this member function is not a candidate for <code>const</code>-qualification. • The member function <code>TestClass::Test2()</code> passes the same data member <code>mMutex</code> to a constructor that takes <code>const</code> references. Therefore, this member function is a candidate for <code>const</code>-qualification. <pre> class Item { }; class Lock1 { public: Lock1(Item& i); }; class Lock2 { public: Lock2(Item& i); Lock2(const Item& i); }; class TestClass { public: void Test1() { Lock1 lk{mMutex}; } void Test2() { Lock2 lk{mMutex}; } private: Item mMutex; }; </pre>

Rule	Description	Change
AUTOSAR C++14 Rule A12-7-1	If the behavior of a user-defined special member function is identical to implicitly defined special member function, then it shall be defined "=default" or be left undefined	<p>The rule checker reports a violation if:</p> <ul style="list-style-type: none"> A class copies or moves <code>std::string</code> objects by using unnecessary user-defined copy and move operators. For example, in this code, the copy and move operators of <code>my_string</code> handle a <code>std::string</code> in a way that is identical to the implicitly defined operators. <pre> #include<string> #include<utility> class my_string { public: my_string& operator=(const my_string& other) { //Noncompl m_name = other.m_name; return *this; } my_string& operator=(my_string&& other) noexcept { //Noncompl m_name = std::move(other.m_name); return *this; } private: std::string m_name; }; </pre> <ul style="list-style-type: none"> A class containing trivial types implements a copy constructor that behaves the same way as an implicit copy constructor.
AUTOSAR C++14 Rule A14-1-1	A template should check if a specific template argument is suitable for this template	<p>The rule checker no longer reports a violation on <code>struct</code> and <code>union</code> templates that contain a <code>static_assert</code> statement. For instance, consider this code:</p> <pre> template<typename T> struct floatStruct //Compliant { static_assert(std::is_floating_point<T>::value, "Expecting a floating }; </pre> <p>Because the <code>struct</code> template <code>floatStruct</code> contain a <code>static_assert</code> statement, the rule checker correctly identifies the template as compliant with this rule and does not report a violation.</p>

Rule	Description	Change
AUTOSAR C++14 Rule A15-4-4	A declaration of non-throwing function shall contain noexcept specification	<p>The rule checker correctly recognizes main() functions that contain the noexcept specification. For instance, the rule checker no longer reports a violation for this code:</p> <pre data-bbox="565 415 1461 703"> #include <vector> int main() noexcept { std::vector<int> c = {1, 2, 3, 4, 5, 6, 7}; //Compliant int x = 5; c.erase(std::remove_if(c.begin(), c.end(), [x](int n) noexcept { return //... } </pre>
AUTOSAR C++14 Rule A15-4-5	Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders	The rule checker does not report a violation of this rule on templates.

Rule	Description	Change
AUTOSAR C++14 Rule A15-5-3	The <code>std::terminate()</code> function shall not be called implicitly	<p>The rule checker provides a more detailed event trace for violations of this rule. For instance, consider this code:</p> <pre>#include <stdexcept> class obj { public: obj(){ //... throw -1; //unhandled exception } }; void main(){ try{ //... obj localObject; //Noncompliant }catch(std::exception& e){ } }</pre> <p>The <code>main()</code> function cannot catch the exception raised by the constructor of <code>obj</code>. The rule checker reports a violation of this rule on the statement that constructs the <code>obj</code> type object. The event trace in Results Details also shows the <code>throw</code> statement where the uncaught exception originates.</p> <p>Previously, the rule checker flagged the <code>main()</code> function if an exception remained uncaught.</p>
AUTOSAR C++14 Rule M16-0-1	<code>#include</code> directives in a file shall only be preceded by other preprocessor directives or comments.	<p>The rule checker does not report a violation of this rule when an <code>extern "C"</code> block contains an <code>#include</code> directive.</p> <pre>#include <file.h> //Compliant extern "C" { #include <file2.h> //Compliant }</pre>
AUTOSAR C++14 Rule M3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility	<p>The rule checker does not report a violation of this rule when a static variable is defined in a function and is only used within an <code>if</code> statement inside the function.</p>

Rule	Description	Change
AUTOSAR C++14 Rule M5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type	The rule checker no longer reports a violation of this defect if you initialize a pointer with these zero constants: <ul style="list-style-type: none">• 0x0• 0U

CERT C

Rule	Description	Change
CERT C: Rec. EXP09-C	Use sizeof to determine the size of a type or variable	The rule checker reports violation of this rule if you hard-code the memory size argument of memory manipulation functions such as memcpy, memmove, memcmp, or memset.
CERT C: Rec. MSC04-C	Use comments consistently and in a readable fashion	The rule checker does not report a violation of this rule when you use a comment that contains ://. For instance, when you add an HTML link in a comment.
CERT C: Rule FI042-C and CERT C++: FI042-C	Close files when they are no longer needed	The rule checker covers file opening and closing using the open() and close() functions.
CERT C: Rule FLP30-C	Do not use floating-point variables as loop counters	<p>The rule checker correctly detects floating-point variables that are not the loop counter in a loop condition. For instance, this code uses the double array buffer in the while loop condition. Polyspace correctly recognizes that buffer is not the loop variable and does not report a violation.</p> <pre> int main(void){ unsigned int iter1, iter2; double buffer[2]; double tmp; //... while((iter1+1 < 2)&& (buffer[iter1]<buffer[iter2])){ //Compliant - 1 // swap buffer[iter1] and buffer[iter2] tmp = buffer[iter2]; buffer[iter2] = buffer[iter1]; buffer[iter1] = tmp; iter2 = iter1; iter1++; } return 1; } </pre>

Rule	Description	Change
CERT C: Rule MEM35-C	Allocate sufficient memory for an object	<p>The rule checker no longer checks for the issue Wrong type used in sizeof during memory allocation. Instead, Polyspace checks for the issue Insufficient memory allocation. Consider this code:</p> <pre>typedef struct S { int a; int b; int c; }S; void foo(){ S* a; a = (S*)malloc(sizeof(S)*2); //Noncompliant } void bar(){ S* a; a = (S*)malloc(sizeof(S)*3); //Compliant }</pre> <p>In both the <code>foo()</code> and <code>bar()</code> functions, you use a pointer as the size argument of <code>malloc</code>. Both of these usages are Wrong type used in sizeof during memory allocation defects. However, <code>bar()</code> allocates 12 bytes of memory, which is sufficient for the object <code>a</code>. The rule checker no longer reports a violation of CERT C: Rule MEM35-C in <code>bar()</code>. In <code>foo()</code>, the eight bytes of allocated memory are insufficient for the object <code>a</code>, causing the issue Insufficient memory allocation. The rule checker reports a violation of this rule.</p>
CERT C: Rule POS30-C	Use the <code>readlink()</code> function properly	<p>The rule checker now reports a Misuse of readlink() even when the value of the variable passed as a third argument of <code>readlink()</code> is unknown, as long as that same variable is also used to dynamically allocate the second argument of <code>readlink()</code>.</p>

CERT C++

Rule	Description	Change
CERT C++: CTR50-CPP	Guarantee that container indices and iterators are within the valid range	<p>The rule checker reports a violation of this rule when:</p> <ul style="list-style-type: none"> The index you use might be negative. Because standard template library (STL) containers expect an unsigned index, the compiler might implicitly cast a negative index into an unsigned index. For example: <pre>std::vector v<int>; int index; //index is negative v[index]; // index is implicitly cast into unsigned int(index)</pre> You explicitly cast a potentially negative value into an unsigned integer and use the converted value as an index of a STL container. For instance: <pre>std::vector v<int>; int index; //index is negative v[static_cast<size_t>(index)]; // index is explicitly cast into unsigned</pre>
CERT C++: FLP30-C	Do not use floating-point variables as loop counters	<p>The rule checker correctly detects floating-point variables that are not the loop counter in a loop condition. For instance, this code uses the double array buffer in the while loop condition. The rule checker correctly recognizes that <code>buffer</code> is not the loop variable and does not report a violation.</p> <pre>int main(void){ unsigned int iter1, iter2; double buffer[2]; double tmp; //... while((iter1+1 < 2)&& (buffer[iter1]<buffer[iter2])){ //Compliant - loop condition is correct // swap buffer[iter1] and buffer[iter2] tmp = buffer[iter2]; buffer[iter2] = buffer[iter1]; buffer[iter1] = tmp; iter2 = iter1; iter1++; } return 1; }</pre>

Rule	Description	Change
CERT C++: MEM35-C	Allocate sufficient memory for an object	<p>The rule checker no longer checks for the issue Wrong type used in sizeof during memory allocation. Instead, Polyspace checks for the issue Insufficient memory allocation. Consider this code:</p> <pre>typedef struct S { int a; int b; int c; }S; void foo(){ S* a; a = (S*)malloc(sizeof(S)*2); //Noncompliant } void bar(){ S* a; a = (S*)malloc(sizeof(S)*3); //Compliant }</pre> <p>In both the <code>foo()</code> and <code>bar()</code> functions, you use a pointer as the size argument of <code>malloc</code>. Both of these usages are Wrong type used in sizeof during memory allocation defects. However, <code>bar()</code> allocates 12 bytes of memory, which is sufficient for the object <code>a</code>. The rule checker no longer reports a violation of CERT C: Rule MEM35-C in <code>bar()</code>. In <code>foo()</code>, the eight bytes of allocated memory are insufficient for the object <code>a</code>, causing the issue Insufficient memory allocation. The rule checker reports this violation.</p>
CERT C++: STR50-CPP	Guarantee that storage for strings has sufficient space for character data and the null terminator	<p>The rule checker reports a violation of this rule if you use an input string that is not null-terminated instead of a null terminated string. For example, in this code, the constructor of the <code>std::string</code> object expects a null-terminated string. Instead, the code uses the input string buffer, which is not null-terminated, and the rule checker reports a violation.</p> <pre>void foo(std::istream& instream) { //... char buffer[10]; instream.read(buffer, sizeof(buffer)); //... std::string str(buffer); //... }</pre>
CERT C++: STR51-CPP	Do not attempt to create a <code>std::string</code> from a null pointer	<p>When you perform string operations that require calling <code>std::char_traits::length()</code>, Polyspace reports a violation of this rule only if the analysis determines that the pointer that <code>std::char_traits::length()</code> operates on is a null pointer. If Polyspace determines only that the pointer might be null, no violation is reported.</p>

MISRA C:2012

Rule	Description	Change
MISRA C:2012 Dir 4.14	The validity of values received from external sources shall be checked	<p>The rule checker reports violations of this rule for these issues:</p> <ul style="list-style-type: none"> Using an externally obtained string that lacks a null-termination in places where a null-terminated string is expected. Such use might result in undefined behavior. For instance, in this code, <code>printf()</code> expects a null-terminated string. Using the character array <code>str</code>, which is not null-terminated, results in undefined behavior. <pre>char str[10]; scanf("%10c", str); printf("%s",str);//Null terminated string expected</pre> Using an externally obtained indeterminate string. When functions from the <code>fgets()</code> family fail, the externally obtained string becomes indeterminate. Each element of this string is a noninitialized variable. Using such an indeterminate string might result in unexpected or undefined behavior. For instance, in this code, if the function <code>fgets()</code> fails, <code>buffer</code> becomes an indeterminate string, consisting of noninitialized elements. Because the function <code>printf()</code> expects a null-terminated string, using <code>buffer</code> with this function results in undefined behavior. <pre>char buffer[10]; fgets(buffer, sizeof(buffer), stdin); printf("%s",buffer);</pre>
MISRA C:2012 Rule 1.2	Language extensions should not be used	The rule checker reports violations of this rule if you use <code>asm</code> functions in your C90 or C99 code.
MISRA C:2012 Rule 2.1	A project shall not contain unreachable code	<p>The rule checker reports a violation of this rule when a static function is not called in the same file where it is defined.</p> <p>For instance, the rule checker flags the function <code>initialize</code> that is defined as <code>static</code> but not called in the same file:</p> <pre>#include <stdlib.h> #include <stdio.h> static int initialize(void) //Noncompliant { int input; printf("Enter an integer:"); scanf("%d",&input); return(input); } void main() { int num; num=0; printf("The value of num is %d",num); }</pre>

Rule	Description	Change
MISRA C:2012 Rule 2.2	There shall be no dead code	<p>Several changes have been made to the rule checker specifications:</p> <ul style="list-style-type: none"> The rule checker reports a violation of this rule when indirectly calling an empty function. For instance, in this code, the function pointer <code>fp</code> holds the pointer value to the empty function creating an indirect function call. <pre>void ex1(void) { } void ex2(void) { void (*fp)(void); fp = ex1; fp(); // Noncompliant }</pre> <ul style="list-style-type: none"> The rule checker no longer ignores potential dead code cases that include <code>asm</code> blocks made of a single <code>NOP</code> instruction. For example, in the function <code>test</code>, if variable <code>b</code> is never read after this point, the rule checker will now flag it as dead code. <pre>void test(void) { int b; b = 0; //Noncompliant __asm("NOP"); }</pre>
MISRA C:2012 Rule 9.1	The value of an object with automatic storage duration shall not be read before it has been set	<p>The rule checker reports a violation of this rule if these issues occur:</p> <ul style="list-style-type: none"> Non-initialized variable Non-initialized pointer
MISRA C:2012 Rule 9.3	Arrays shall not be partially initialized	<p>The rule checker reports a violation when an array is initialized with an incorrect shorthand notation such as <code>{0.0}</code>, <code>{0U}</code> or <code>{}</code> (instead of the usual <code>{0}</code>).</p>

Rule	Description	Change
MISRA C:2012 Rule 10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category	<p>There are multiple changes to the behavior of this checker:</p> <ul style="list-style-type: none"> The rule checker no longer reports a violation when a variable of aggregate type such as an array is initialized using the shorthand notation <code>{0}</code>, for instance: <pre>float dat2[3*3] = {0};</pre> This checker no longer reports a violation on <code>boolean</code> types when you generate code by using the target <code>autosar.tlc</code>.
MISRA C:2012 Rule 13.5	The right hand operand of a logical <code>&&</code> or <code> </code> operator shall not contain persistent side effects	<p>The rule checker no longer reports a violation of this rule in these situations.</p> <ul style="list-style-type: none"> The right side operand of a logical <code>&&</code> or <code> </code> operator contains a call to a function, for example, <code>A()</code>, and Polyspace considers <code>A()</code> to be pure. If <code>A()</code> calls another function, for example, <code>B()</code>, and Polyspace considers <code>B()</code> to be pure. <p>Previously, the checker reported a violation if the right operand of a logical <code>&&</code> or <code> </code> operator contained a function call, whether pure or not. For more information on how the checker detects pure functions, see MISRA C:2012 Rule 13.5.</p>
MISRA C:2012 Rule 14.1	A loop counter shall not have essentially floating type	<p>The rule checker correctly detects floating-point variables that are not the loop counter in a loop condition. For instance, this code uses the <code>double</code> array <code>buffer</code> in the <code>while</code> loop condition. The rule checker correctly recognizes that <code>buffer</code> is not the loop variable and does not report a violation.</p> <pre>int main(void){ unsigned int iter1, iter2; double buffer[2]; double tmp; //... while((iter1+1 < 2)&& (buffer[iter1]<buffer[iter2])){ //Compliant - l // swap buffer[iter1] and buffer[iter2] tmp = buffer[iter2]; buffer[iter2] = buffer[iter1]; buffer[iter1] = tmp; iter2 = iter1; iter1++; } return 1; }</pre>

Rule	Description	Change
MISRA C:2012 Rule 19.1	An object shall not be assigned or copied to an overlapping object	<p>The rule checker reports a violation of this rule if one member of an object of aggregate type is copied to an adjoining member and the number of bytes copied result in an overlap between the source and destination.</p> <p>For instance, in this example, the member <code>init_string</code> of the aggregate structure <code>my_string_aggr</code> consists of 21 characters. But, more than 21 characters of this member are copied to the adjoining member <code>new_string</code>, resulting in an overlapping copy</p> <pre>typedef struct { char init_string[21]; char new_string[101]; } string_aggr; string_aggr my_string_aggr; void copy() { strncpy(&my_string_aggr.new_string[0], &my_string_aggr.init_string[0], 100U); /* Non-compliant */ }</pre>
MISRA C:2012 Rule 21.6	The Standard Library input/output functions shall not be used	<p>The rule checker now reports a violation of this rule when you use any of these Standard Library input/output functions:</p> <ul style="list-style-type: none"> • <code>snprintf()</code> • <code>vsnprintf()</code> • <code>vsscanf()</code>

MISRA C++:2008

Rule	Description	Change
MISRA C++:2008 Rule 0-1-5	A project shall not contain unused type declarations	The rule checker no longer reports a violation on <code>public</code> unused types you define in a template.
MISRA C++:2008 Rule 2-10-1	Different identifiers shall be typographically unambiguous	The rule checker no longer reports duplicate violations in templates. Previously, each typographically ambiguous template function showed two violations - one on the template definition and another on the template instantiation.
MISRA C++:2008 Rule 16-0-1	<code>#include</code> directives in a file shall only be preceded by other preprocessor directives or comments.	The rule checker does not report a violation of this rule when an <code>extern "C"</code> block contains an <code>#include</code> directive. <pre>#include <file.h> //Compliant extern "C" { #include <file2.h> //Compliant }</pre>
MISRA C++:2008 Rule 5-0-18	<code>></code> , <code>>=</code> , <code><</code> , <code><=</code> shall not be applied to objects of pointer type, except where they point to the same array	The rule checker no longer reports violations of this rule when you compare integers that are cast from pointers.
MISRA C++:2008 Rule 5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type	The rule checker no longer reports a violation of this defect if you initialize a pointer with these zero constants: <ul style="list-style-type: none"> • <code>0x0</code> • <code>0U</code>
MISRA C++:2008 Rule 8-5-1	All variables shall have a defined value before they are used	The rule checker reports a violation of this rule if these issues occur: <ul style="list-style-type: none"> • Non-initialized variable • Non-initialized pointer • Member not initialized in constructor

Rule	Description	Change
MISRA C++:2008 Rule 9-3-3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.	<p>The rule checker no longer requires <code>const</code>-qualification for a member functions if the function passes a data member to a constructor that might possibly modify the data member. For instance, in this example:</p> <ul style="list-style-type: none"> • The member function <code>TestClass::Test1()</code> passes a data member <code>mMutex</code> to a constructor that takes non-<code>const</code> references (there is no constructor overload that takes <code>const</code> references). Therefore, this member function is not a candidate for <code>const</code>-qualification. • The member function <code>TestClass::Test2()</code> passes the same data member <code>mMutex</code> to a constructor that takes <code>const</code> references. Therefore, this member function is a candidate for <code>const</code>-qualification. <pre> class Item { }; class Lock1 { public: Lock1(Item& i); }; class Lock2 { public: Lock2(Item& i); Lock2(const Item& i); }; class TestClass { public: void Test1() { Lock1 lk{mMutex}; } void Test2() { Lock2 lk{mMutex}; } private: Item mMutex; }; </pre>

MISRA AC AGC and MISRA C:2004

Rule	Description	Change
9.1	All automatic variables shall have been assigned a value before being used	The rule checker reports a violation of this rule if these additional issues occur: <ul style="list-style-type: none"> • Non-initialized variable • Non-initialized pointer

JSF AV C++ Coding Rules

Rule	Description	Change
41	Source lines will be kept to a length of 120 characters or less.	Starting in R2023a, Polyspace ignores the line splicing character when calculating the number of characters in a line.
142	All variables shall be initialized before use.	The rule checker reports a violation of this rule if these issues occur: <ul style="list-style-type: none"> • Non-initialized variable • Non-initialized pointer • Member not initialized in constructor

Custom Coding Rules

Rule	Description	Change
20.1	Source line length must not exceed specified number of characters.	Starting in R2023a, Polyspace ignores the line splicing character when calculating the number of characters in a line.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

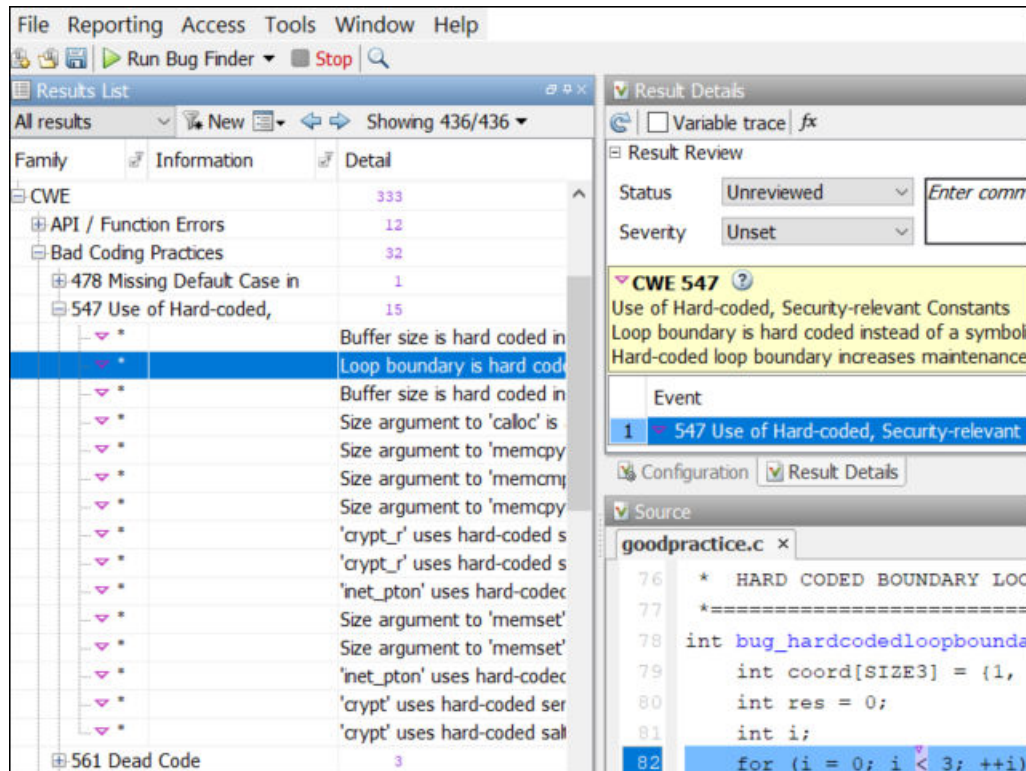
Compatibility Considerations

If you previously checked your code for the preceding rules, you might see a change in the number of violations.

Reviewing Results

Security Standards Support: View violations of CWE rules as analysis findings in the Results List and in reports

In R2023a, when you check for violations of the CWE rules, Polyspace reports violations of these rules as explicit findings in the **Results List** and in reports.



In the results list you see the rule number and you can filter the results by CWE rule for a more focused review.

See also:

- Check CWE (-cwe)
- “Common Weakness Enumeration (CWE)”

Products: Polyspace Bug Finder, Polyspace Access.

Compatibility Considerations

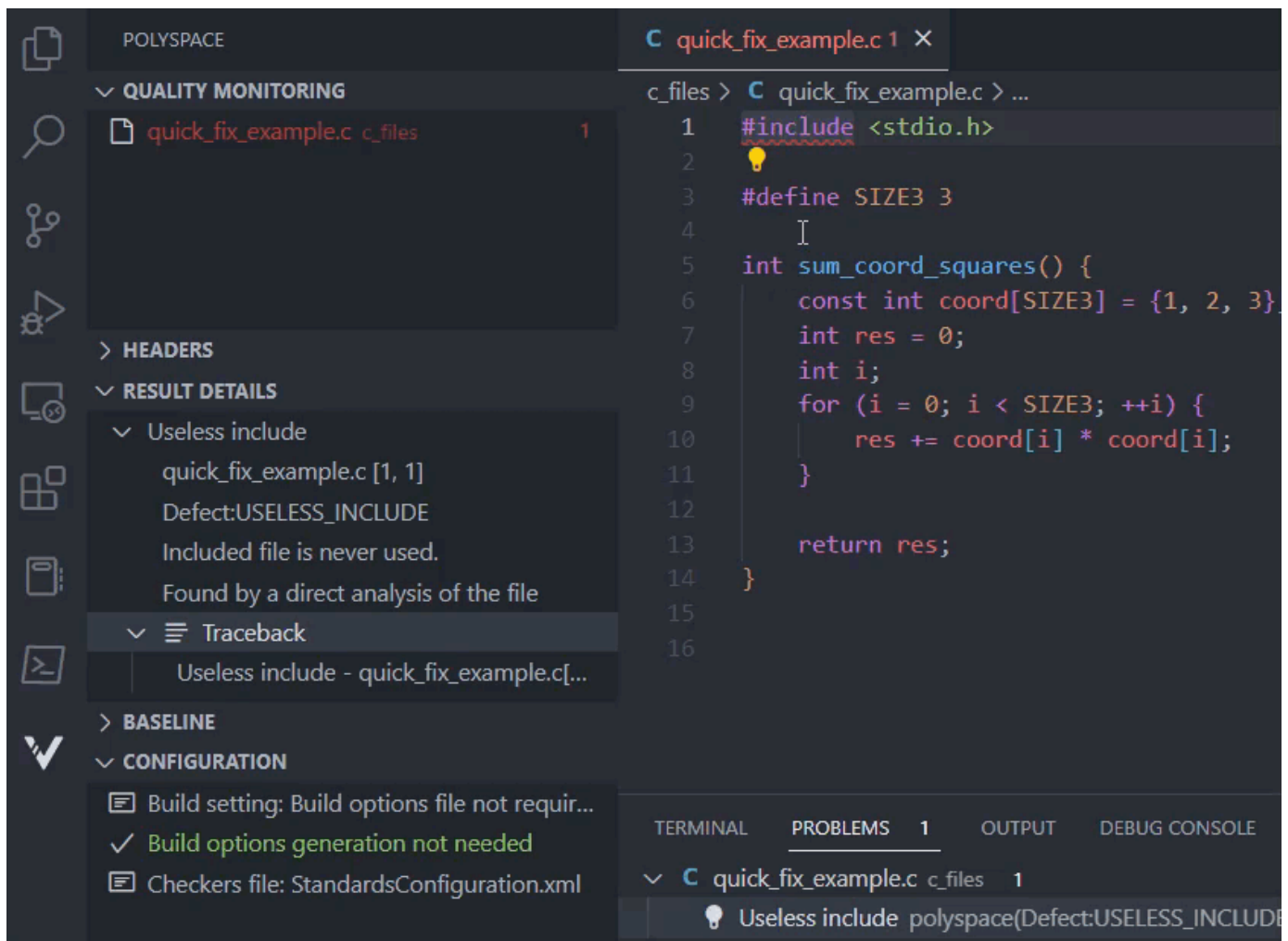
In previous releases, when reviewing violations of CWE rules, you reviewed the Bug Finder defects that were mapped to the rules. When you reviewed results in the results list, CWE rules that corresponded to a defect were listed in a **CWE** column. In R2023a, you can check for violations of CWE rules directly and review findings by CWE rule ID.

If you open results generated with a Polyspace product version R2022b or earlier, the **CWE** column is still available in the **Results List**. Starting with R2023a results, the **CWE** column is no longer available.

Polyspace Extension for Visual Studio Code (VSCode): Fix common defects or coding rule violations in one click

You can now fix some common defects or coding rule violations, such as not declaring an unmodified variable `const`, with a single click. You can apply the fix to one finding or to all instances of that finding in the currently active file.

To apply a fix, select a finding in the editor or the **Problems** panel, and then click the appropriate option in the light bulb menu. Polyspace inserts the fix in your code and removes the corresponding finding from the list of findings in the **Problems** panel.

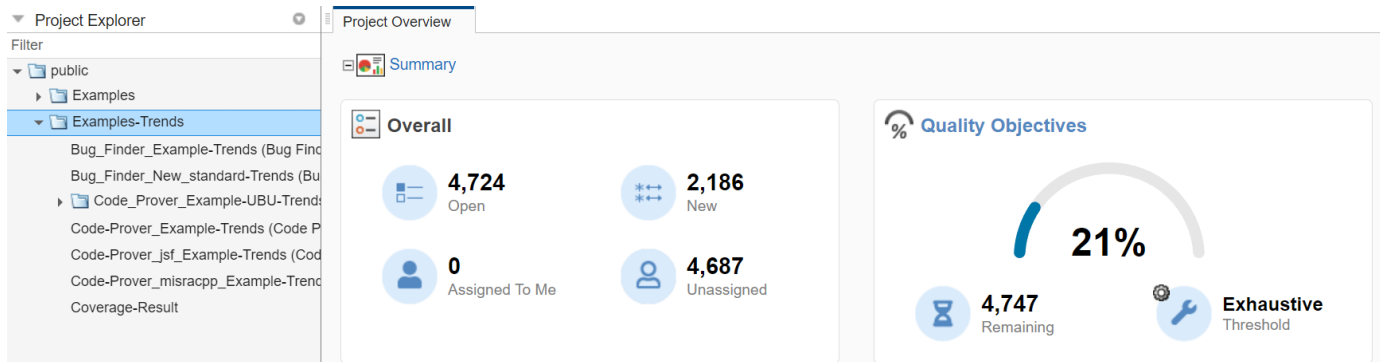


See also “Apply Suggested Fix for Common Defects or Coding Rule Violations”.

Products: , Polyspace Access (Polyspace as You Code).

Software Quality Objectives: View aggregated software quality objectives in Polyspace Access

In R2023a, Polyspace Access shows aggregate software quality objectives (SQO) statistics when you select a project folder. To view aggregate SQO statistics, click on a folder in the **Project Explorer** pane. A **Quality Objectives** card shows in the **Project Overview** dashboard. To view detailed SQO information for all items in the selected folder, click on the title of the **Quality Objectives** card to open the **Quality Objectives** dashboard.



Note these differences between viewing SQO information at the folder level and the project level.

- For folders, the **Progress** and **Details** sections do not contain links to filtered results in the tables.
- You cannot assign quality objective levels to all projects in the folder at once.
- The **Quality Objectives** card for folders does not show **Pass**, **In progress**, or **Incomplete** labels.

Polyspace Access aggregates SQO statistics even if the quality objectives configuration levels are not homogeneous. While individual projects might contain separate definitions of each SQO level, Polyspace Access does not separate the statistics by level details when aggregating the statistics. All SQO1 level projects are aggregated together, as are SQO2 and so forth.

For more information about the **Quality Objectives** dashboard, see “Quality Objectives Dashboard in Polyspace Access”.

Report Generation: Use CodingStandards report template to generate report for CWE results

In R2023a, Polyspace supports a subset of the Common Weakness Enumeration (CWE) list natively and you can use the CodingStandards report template to generate a report for CWE results. You no longer use the SecurityCWE template.

You can find these templates in the folder `polyspaceroot\toolbox\polyspace\psrptgen\templates\`, where `polyspaceroot` is your Polyspace installation folder; for instance `C:\Program Files\Polyspace\R2023a`.

Products: Polyspace Bug Finder, Polyspace Bug Finder Server.

Compatibility Considerations

If you use scripts where you specify a report template to generate a report for CWE results, update the path of the template to point to the CodingStandards template.

The SecurityCWE template is now located under *polyspaceroot\toolbox\polyspace\psrptgen\templates\bug_finder\deprecated*. You can still use this template for CWE results generated with a Polyspace product version R2022b or earlier.

Coding Standards Support: Display custom comments in Results List and Result Details pane

If you use the **Checkers selection** interface to enable coding rule checkers and you enter text in the **Comment** column for these rules, Polyspace now displays that text in the **Result Details** pane and in the **Detail** column of the **Results List**. You can then filter your results based on the text that you entered.

You can add custom comments, if, for instance, you want to customize the categorization of a coding standard. For example, if you want to treat a MISRA C™:2012 advisory rule as required.

The screenshot displays the Polyspace interface with three main components:

- Checkers selection pane:** Shows a tree view of coding standards. The **MISRA C:2012** category is selected, and a table of rules is shown below.

Status	Category	Name	Comment
<input checked="" type="checkbox"/>	required	8.2 Function types shall be in prototype for...	
<input checked="" type="checkbox"/>	required	8.3 All declarations of an object or function...	
<input checked="" type="checkbox"/>	required	8.4 A compatible declaration shall be visible...	
<input checked="" type="checkbox"/>	required	8.5 An external object or function shall be ...	
<input checked="" type="checkbox"/>	required	8.6 An identifier with external linkage shall h...	
<input checked="" type="checkbox"/>	advisory	8.7 Functions and objects should not be de...	Mandatory for this project
<input checked="" type="checkbox"/>	required	8.8 The static storage class specifier shall ...	
<input checked="" type="checkbox"/>	advisory	8.9 An object should be defined at block sc...	
<input checked="" type="checkbox"/>	required	8.10 An inline function shall be declared wit...	
<input checked="" type="checkbox"/>	advisory	8.11 When an array with external linkage is...	
<input checked="" type="checkbox"/>	required	8.12 Within an enumerator list, the value o...	
<input checked="" type="checkbox"/>	advisory	8.13 A pointer should point to a const-qual...	
<input checked="" type="checkbox"/>	required	8.14 The restrict type qualifier shall not be ...	
<input checked="" type="checkbox"/>		9 Initialization	
<input checked="" type="checkbox"/>		10 The essential type model	
- Results List pane:** Shows a summary of results for MISRA C:2012, including 81 total results, with 52 under '8 Declarations and definitions' and 52 under '8.7 Functions and objects should not be de...'. A specific instance of rule 8.7 is highlighted with a custom user comment: 'Mandatory for this project'.
- Result Details pane:** Shows the details for the selected MISRA C:2012 8.7 (Advisory) rule, including the rule text and the custom user comment: 'Mandatory for this project'.

Products: Polyspace Bug Finder, Polyspace Bug Finder Server, Polyspace Access.

Inputs Causing Defect: See example system inputs that lead to non-initialized pointers

If a pointer in your code is initialized only for certain system input values, in R2023a, a Bug Finder analysis can report the defect along with one possible combination of input values causing the defect. Bug Finder shows problematic input values for these defects:

- Non-initialized pointer
- Pointer to non initialized value converted to const pointer

For instance, in this example, the `if` statement has a missing `else` branch. As a result, the pointer `backup` can be non-initialized for certain values of the input size.

```
void checkIfLowCount(int*, int);
void removeFirstElement(int *arr, int size)
{
    int *backup;
    if(size > 2)
    {
        backup = arr;
        for(int i = 0; i < size - 1; i++, arr++)
        {
            *(arr+1) = *arr;
        }
        *arr = 0;
    }
    checkIfLowCount(backup, size);
}
```

If you use the option `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`, the event list below the **Non-initialized pointer** defect shows a possible value of the input size that leads to the defect.

ID 1: Non-initialized pointer (Impact: High) ? 🔗

Local pointer 'backup' is read before being initialized.

Result includes example values that lead to the defect.

	Event	File	Scope	Line
1	Function called by external code with input 'size'	file.c	removeFirstElement()	3
	Possible input value causing defect: -49			
2	Entering function 'removeFirstElement'	file.c	removeFirstElement()	3
3	Declaration of variable 'backup'	file.c	removeFirstElement()	4
4	Not entering if statement (if-condition false)	file.c	removeFirstElement()	5
5	Non-initialized pointer	file.c	removeFirstElement()	12

For the full list of defects that Bug Finder can report along with input values, see “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

Simulink Support: Justify Known MISRA C++:2008 and AUTOSAR C++14 Violations

Starting in R2023a, Polyspace recognizes the justifying code annotations that Embedded Coder® inserts in code generated from Simulink® models.

When generating code from a model, if you set the properties `JustifyMISRAViolations` and `GenerateComments` to `on`, Embedded Coder inserts code annotations that justify known MISRA C++:2008 and AUTOSAR C++14 violations. For instance, the generated code might contain dummy or placeholder functions that are intended to be modified. These functions can violate MISRA C++:2008 rules. By using the preceding properties, you generate code in which Embedded Coder annotates such violations with justifying comments. When you run a Polyspace analysis on such generated code, the MISRA C++:2008 and AUTOSAR C++14 violations caused by the placeholder functions are reported as `Justified`.

For example, in this generated code, the function `test_init()` is noncompliant with MISRA C++:2008 Rule 0-1-8. If you set `JustifyMISRAViolations` and `GenerateComments` to `on`, Embedded Coder inserts the comments `MW:begin` and `MW:end` around the function:

```
// MW:begin MISRA-CPP:0-1-8 "Justify"
void test_init()
{
    int i;
}
// MW:end MISRA-CPP:0-1-8 "Justify"
```

Polyspace recognizes the `MW:begin` and `MW:end` annotations and shows the annotated results as `Justified` in the results list.

See:

- “Include Comments in Generated C/C++ Code” (Embedded Coder)
- “Generate C/C++ Code with Improved MISRA and AUTOSAR Compliance” (Embedded Coder)
- `coder.setupMISRAConfig`

Product: Polyspace Bug Finder (Desktop).

Polyspace user interfaces now available in Chinese

Polyspace software is now available in Chinese. Starting in R2023a, if you work on a machine with a Chinese locale, the labels and messages in the Polyspace user interfaces and IDE extensions are now translated.

Products: Polyspace Bug Finder, Polyspace Bug Finder Server, Polyspace Access.

Reduction in duplicate results on templates

In R2023a, for checkers of certain types, if the same issue is seen in multiple instantiations of a template, you see the result for only one instantiation. You do not have to review the same issue multiple times.

Compatibility Considerations

You might see fewer results related to template instantiations compared to previous releases.

Polyspace Access Installation

Polyspace Extension for Visual Studio Code (VSCode): Polyspace as You Code available for macOS

Starting in R2023a, the Polyspace as You Code extension for Visual Studio Code is available for macOS. If your build system supports the generation of a JSON compilation database file, you can configure the extension to extract the build options from your build system automatically. See “Get Build Configuration from JSON Compilation Database”.

Previously, Polyspace could not extract the configuration of your build system if the System Integrity (SIP) feature was enabled. If you can generate a JSON compilation database file for your build system, you no longer need to disable this feature to extract the build configuration.

See also `polyspace-configure`.

Products: Polyspace Access (Polyspace as You Code).

Polyspace as You Code Plugin for Eclipse: Plugin is not compatible with the Polyspace desktop integration plugin

Starting in R2023a, the Polyspace as You Code plugin for Eclipse™-based IDEs is no longer compatible with the Polyspace desktop plugin for Eclipse.

Compatibility Considerations

If you use the Polyspace as You Code installer to install the plugin, Polyspace uninstalls the desktop plugin automatically. See “Install Polyspace as You Code Using Installer”.

If you install the Polyspace as You Code plugin manually, uninstall the desktop plugin for Eclipse first. See “Uninstall Polyspace Plugin”.

R2022b

Version: 3.7

New Features

Bug Fixes

Compatibility Considerations

Analysis Setup

Compiler Support: Set up Polyspace analysis for code compiled with Intel C++ Compiler Classic (icc/icl) compilers

In R2022b, if you build your source code by using an Intel® C++ Compiler Classic (icc/icl) compiler, you can specify the compiler name for your Polyspace analysis.

Target Environment	
Compiler	intel
Target processor type	x86_64

At the command line, you specify a compiler by using the `-compiler` option. For instance:

```
polyspace-bug-finder-server -sources file.c -compiler intel ...
```

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

See also `Intel C++ Compiler Classic (icc/icl) (-compiler intel)`.

You can now set up a Polyspace project without knowing the internal workings of an Intel C++ Compiler Classic (icc/icl) 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.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Updated Clang Compiler Support: Set up Polyspace analysis for code compiled by using Clang version 12.x

In R2022b, Polyspace supports Clang compiler version 12.x natively. If you build your source code by using Clang compiler version 12.x, you can specify the compiler name for your Polyspace analysis.

Target Environment	
Compiler	clang12.x
Target processor type	x86_64

At the command line, you specify a compiler by using the option `-compiler`. For instance:

```
polyspace-bug-finder-server -sources file.c -compiler clang12.x ...
```

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

You can now set up a Polyspace project without knowing the internal workings of this compiler. The analysis can interpret macros that are implicitly defined by the compiler and compiler-specific language extensions, such as keywords and pragmas.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Modifying Checkers: Create list of macros to prohibit and check for use of macros from the list

In R2022b, you can define a blacklist of macros to forbid from your source code and use Bug Finder to implement your blacklist. The Bug Finder checker **Use of a forbidden macro** checks if a macro from this list appears in your sources.

You can use this checker to impose consistency in macro usage. For instance, if you want to impose the rule that boolean macros must take the form TRUE or FALSE but not the versions true or false, you can use this checker.

See also:

- Use of a forbidden macro
- Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

Design Constraints: Specify constraints on function inputs for more precise analysis

In R2022b, you can specify external constraints on unknown system inputs for a more precise analysis with Bug Finder. Unknown system inputs include inputs to uncalled functions and return values of stubbed functions.

For instance, in this example, you see an Integer division by zero defect on the division operation in `getFlooredNumber()`:

```
typedef struct group {
    int maxAllocation;
    int minGroupSize;
};

int getFlooredNumber(int total, int size) {
    if (size >= 0)
        return total/size;
    else
        return -1;
}

int getMaxGroups(struct group* aGroup) {
    int maxGroupNum = getFlooredNumber(aGroup->maxAllocation, aGroup->minGroupSize);
    return maxGroupNum;
}
```

A Bug Finder analysis determines values of unknown inputs from the input data type. Since the function `getFlooredNumber()` is called only from `getMaxGroups()`, which in turn is uncalled, the inputs to `getFlooredNumber()` are unknown. The analysis assumes that the second input to `getFlooredNumber()` can take any `int` value including zero and lead to a division by zero. Prior to R2022b, you could not constrain the inputs further. You can now specify external constraints on function inputs to narrow down the range of possible values. For instance, in the preceding example, you can specify a constrained range on `getMaxGroups()` so that the second input to `getFlooredNumber()` no longer includes 0:

The screenshot shows a window titled "Constraint Specification - D:\Polyspace\R2022b\Features\DRS in Bug Finder\Module_1_BF_DRS_drs_template.xml". Below the title bar are icons for file operations and buttons for "Update" and "Stop". The main area is a table with three columns: "Name", "Init Range", and "Init Mode". The table is expanded to show a tree view of "User Defined Functions" containing "getFlooredNumber()", "getMaxGroups()", and "getMaxGroups.return". Under "getMaxGroups()", there are three entries: "getMaxGroups.arg1" with "INIT" mode, "getMaxGroups.* arg1" with "MAIN GENERATOR" mode, and "getMaxGroups.* arg1.minGroupSize" with "1..10" range and "INIT" mode.

Name	Init Range	Init Mode
Global Variables		
User Defined Functions		
getFlooredNumber()		
getMaxGroups()		
getMaxGroups.arg1		INIT
getMaxGroups.* arg1		
getMaxGroups.* arg1.maxAllocation		MAIN GENERATOR
getMaxGroups.* arg1.minGroupSize	1..10	INIT
getMaxGroups.return		

If you apply this constraint, Bug Finder no longer shows the division by zero defect.

For more information on external constraints, see:

- Specify External Constraints for Polyspace Analysis
- External Constraints for Polyspace Analysis

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

Object Size Limitation Removed: Analyze code containing large data structures

In R2022b, Polyspace can analyze code containing large objects. Previously, objects with size greater than 256 MB caused the analysis to stop. Such large-sized objects typically result from deep nesting of structures containing array fields or multidimensional arrays of large structures.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

Changes in analysis options and binaries

The argument `data_race_all` for the option `-checkers` is removed

Errors

Starting in R2022b, using the argument `data_race_all` with the option `-checkers` results in an error. To detect data races in atomic operations, use the analysis option `-detect-atomic-data-race`. See:

- `-detect-atomic-data-race`
- Extend Data Race Checkers to Atomic Operations

If you use `-checkers all` to run a Bug Finder analysis, you might see a reduced number of defects because this command no longer checks for data races in atomic operations. To mimic the result of `-checkers all` in previous releases, specify the option `-detect-atomic-data-race`.

The option `-detect-atomic-data-race` is a command-line only option. To use this option from the Polyspace user interface, in the **Advanced Settings** node, specify the option in the **Other** field.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

Analysis Results

AUTOSAR C++14 Support: Check for 370 AUTOSAR C++14 rules including 7 new rules

In R2022b, you can look for violations of these AUTOSAR C++14 rules in addition to previously supported rules. Polyspace Bug Finder now supports 100% of the statically enforceable AUTOSAR C++14 rules. See [Statically Enforceable AUTOSAR C++14 Rules Supported by Polyspace Bug Finder](#).

AUTOSAR Rule	Description	Polyspace Checker		
A8-4-11	A smart pointer shall only be used as a parameter type if it expresses lifetime semantics	AUTOSAR A8-4-11	C++14	Rule
A8-4-12	A <code>std::unique_ptr</code> shall be passed to a function as: (1) a copy to express the function assumes ownership (2) an lvalue reference to express that the function replaces the managed object	AUTOSAR A8-4-12	C++14	Rule
A8-4-13	A <code>std::shared_ptr</code> shall be passed to a function as: (1) a copy to express the function shares ownership (2) an lvalue reference to express that the function replaces the managed object (3) a <code>const</code> lvalue reference to express that the function retains a reference count	AUTOSAR A8-4-13	C++14	Rule
A15-1-5	Exceptions shall not be thrown across execution boundaries	AUTOSAR A15-1-5	C++14	Rule
A16-2-2	There shall be no unused include directives	AUTOSAR A16-2-2	C++14	Rule
A16-2-3	An include directive shall be added explicitly for every symbol used in a file	AUTOSAR A16-2-3	C++14	Rule
A20-8-4	A <code>std::unique_ptr</code> shall be used over <code>std::shared_ptr</code> if ownership sharing is not required	AUTOSAR A20-8-4	C++14	Rule

See also AUTOSAR C++14 Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

Bug Finder Checkers: Check for uncaught exceptions, dangling `string_view`, and other issues

In R2022b, using new Bug Finder checkers, you can check for these additional types of issues.

Defect	Description	Defect Group
Expensive return caused by unnecessary <code>std::move</code>	An unnecessary call to <code>std::move</code> in the return statement hinders return value optimization, resulting in inefficient code	Performance
Expensive use of container's <code>size</code> method	A container's <code>size()</code> method is used for checking emptiness instead of its <code>empty()</code> method, which is more efficient	Performance
Expensive use of <code>map</code> 's bracket operator to insert or assign a value	The bracket operator of <code>std::map</code> or <code>std::unordered_map</code> is used for inserting or assigning a value in the container instead of the <code>insert_or_assign()</code> method, which is more efficient	Performance
Invalid scientific notation format	The use of an invalid format makes the code less readable	Good Practice
Missing call to container's <code>reserve</code> method	A fixed number of items are added to a container without calling the <code>reserve()</code> method of the container beforehand, resulting in inefficient code	Performance
<code>std::string_view</code> initialized with dangling pointer	An <code>std::string_view</code> object is initialized by using an unnamed temporary object	Programming
Uncaught exception	An exception is raised from a function but it is not caught and handled	C++ Exception
Useless include	An include directive is present but not used	Good Practice
Use of a forbidden macro	Use of a macro that appears in a blocklist of forbidden macros	Good practice

For the full list of checkers, see Defects. See also Bug Finder Defect Groups

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access

CERT C++ Support: Check for issues arising from container iterators, memory management, and `std::string` objects

In R2022b, you can look for violations of these CERT C++ rules in addition to previously supported rules.

CERT C++ Rule	Description	Polyspace Checker
CTR52-CPP	Guarantee that library functions do not overflow	CERT C++: CTR52-CPP
CTR53-CPP	Use valid iterator ranges	CERT C++: CTR53-CPP

CERT C++ Rule	Description	Polyspace Checker
CTR54-CPP	Do not subtract iterators that do not refer to the same container	CERT C++: CTR54-CPP
CTR55-CPP	Do not use an additive operator on an iterator if the result would overflow	CERT C++: CTR55-CPP
DCL55-CPP	Avoid information leakage when passing a class object across a trust boundary	CERT C++: DCL55-CPP
DCL56-CPP	Avoid cycles during initialization of static objects	CERT C++: DCL56-CPP
ERR59-CPP	Do not throw an exception across execution boundaries	CERT C++: ERR59-CPP
EXP51-CPP	Do not delete an array through a pointer of the incorrect type	CERT C++: EXP51-CPP
EXP62-CPP	Do not access the bits of an object representation that are not part of the object's value representation	CERT C++: EXP62-CPP
MEM53-CPP	Explicitly construct and destruct objects when manually managing object lifetime	CERT C++: MEM53-CPP
STR51-CPP	Do not attempt to create a <code>std::string</code> from a null pointer	CERT C++: STR51-CPP
STR52-CPP	Use valid references, pointers, and iterators to reference elements of a <code>basic_string</code>	CERT C++: STR52-CPP

See also CERT C++ Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access

CWE Support: Updated CWE version and support for additional rules related to security, data flow, and other issues

In R2022b, Polyspace supports version 4.6 of the CWE standard. In this release, you can check for violations of these CWE rules in addition to previously supported rules.

CWE Rule	Description	Polyspace Checkers
248	Uncaught Exception	Uncaught exception
563	Assignment to Variable without Use	Write without a further read

CWE Rule	Description	Polyspace Checkers
570	Expression is Always False	Dead code Code deactivated by constant false condition
571	Expression is Always True	Dead code Useless if Unreachable code
798	Use of Hard-coded Credentials	Hard-coded sensitive data Constant cipher key
1335	Incorrect Bitwise Shift of Integer	Shift of a negative value Right operand of shift operation outside allowed bounds

For the full mapping between CWE rules and Polyspace Bug Finder defect checkers, see [CWE Coding Standard](#) and [Polyspace Results](#).

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

Custom Coding Rules: Enforce naming convention for floating-point literals

In R2022b, you can check that floating-point literals in your code adhere to a specific naming convention by enabling new custom coding rule 8.5. To enable custom coding rules, see [Create Custom Coding Rules](#).

For instance, you might want to check that floating-point literals are suffixed with characters 'f' or 'F'. The suffix lets the compiler know that the floating-point literal is a `float` and you avoid the overhead of arithmetic conversions.

Consider this code:

```
float func1(float var)
{
    return var * 4.00; // Noncompliant, 4.00 is considered a double
}
float func2(float var)
{
    return var * 4.00F; //Compliant
}
```

- In `func1()`, `var` is converted to a `double` because the literal `4.00` is interpreted as a `double`, and the multiplication is performed with `double` precision. The result is then truncated back to `float`.

- In `func2()`, the literal is suffixed with a capital F. This indicates to the compiler that the literal is a float and the calculation is faster because there is no conversion to a double.

You use regular expressions to specify the desired pattern, for example, `(.*(f|F))|(^0\.\d$)`. Here, the second capturing group `(^0\.\d$)` prevents Polyspace from reporting a violation for the trivial case of floating-point literal 0.0.

See also `Check custom rules (-custom-rules)`.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code).

Changes to coding standards checking

In R2022b, these changes have been made in the checking of previously supported rules.

AUTOSAR C++14

Rule	Description	Change
AUTOSAR C++14 Rule A2-7-3	All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation	The rule checker shows fewer false positives associated with declarations in headers. Previously, if you suppressed results from a source file using the option <code>Do not generate results for (-do-not-generate-results-for)</code> , you might have seen incorrect rule violations in headers included in the source file.
AUTOSAR C++14 Rule A5-1-1	Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead	The rule checker no longer reports a violation on range-based for loops. For instance: <pre>for (auto & l_Button : m_pButtonArray) //Compliant { l_Button = new SelectionButton(this); }</pre> Previously, the rule checker reported a violation on range-based for loops that did not use any literal values.
AUTOSAR C++14 Rule A5-2-2	Traditional C-style casts shall not be used	The rule checker no longer reports a violation on casts that uses the <code>{}</code> notation. For instance: <pre>std::uint32_t var7 = std::uint32_t{0}; //Compliant</pre>
AUTOSAR C++14 Rule A7-1-3	CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name	Polyspace does not report a violation when you use a CV qualifier on the left side of <code>std::string</code> and other typedef names from the <code>std</code> namespace. For example: <pre>//... const std::string Name = "Jdoe"; //Compliant</pre>

Rule	Description	Change
AUTOSAR C++14 Rule M7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function	The rule checker no longer reports a violation if a function returns a pointer or reference that is passed to the function as a parameter. For instance: <pre>int* func(int* param){ int* localCopy = param; //... return localCopy;//Compliant }</pre>
AUTOSAR C++14 Rule A12-1-1	Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members	Polyspace does not report a violation of this rule if a class uses delegating constructors. In this code, Polyspace does not report violations of this rule when the constructor <code>Derived::Derived(int)</code> delegates its initialization duties to <code>Derived::Derived(int, int)</code> . <pre>class Base { public: Base() : a{1} {} virtual void abstractA() const = 0; private: int a; }; class Derived: public Base { public: Derived(int a, int b): Base{}, z{a}, y{b} {} //Compliant Derived(int a): Derived(a, 0) {} //Compliant private: int z; int y; };</pre>
AUTOSAR C++14 Rule A12-1-2	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type	When reporting violations in derived classes, the rule checker no longer takes into consideration initialization in base classes. Initialization in base classes cannot cause the kind of confusion that the rule seeks to avoid.

Rule	Description	Change
AUTOSAR C++14 Rule A20-8-5 and AUTOSAR C++14 Rule A20-8-6	<code>std::make_shared</code> (or <code>std::make_unique</code>) shall be used to construct objects owned by <code>std::shared_ptr</code> (or <code>std::weak_ptr</code>)	The checkers no longer report a violation when a smart pointer is passed by reference to a function and then initialized inside the function.
AUTOSAR C++14 Rule A27-0-2	A C-style string shall guarantee sufficient space for data and the null terminator	The rule checker reports a violation of this rule if the size of a destination buffer in a <code>strcpy()</code> operation is insufficient to accommodate the source buffer and a null terminator.

CERT C

Rule	Description	Change
CERT C: Rule ARR39-C	Do not add or subtract a scaled integer to a pointer	<p>The rule checker now raises a violation when improper pointer scaling is performed in multiple statements. For instance:</p> <pre>#define bigNum unsigned long long struct Collection { bigNum bn_A; bigNum bn_B; }; void foo(void) { size_t offset = offsetof(struct Collection, bn_B); struct Collection *s = (struct Collection *)malloc(sizeof(struct Collection)); //.... memset(s + offset, 0, sizeof(struct Collection) - offset); //Noncompliant }</pre> <p>Previously, the rule checker detected improper pointer scaling performed in a single statement.</p>
CERT C: Rule EXP36-C	Do not cast pointers into more strictly aligned pointer types	<p>The rule checker does not report violations of this rule if you use a source pointer that is known to align correctly to the target type. For instance:</p> <ul style="list-style-type: none"> • The <code>alignas</code> specifier matches the alignment of the source pointer with that of the target pointer. • Pointers returned by functions such as <code>aligned_alloc()</code>, <code>malloc()</code>, and <code>realloc()</code> matches the alignment of the target pointer.
CERT C: Rule MEM30-C	Do not access freed memory	<p>The rule checker now reports a violation when you free memory after a possible zero-size reallocation. In some implementations, zero-size reallocations free the memory. Freeing such reallocated memory might lead to a double free defect.</p>
CERT C: Rule MEM34-C	Only free memory allocated dynamically	<p>The rule checker now reports a violation when you reallocate memory that was not previously allocated dynamically.</p>
CERT C: Rule STR31-C	Guarantee that storage for strings has sufficient space for character data and the null terminator	<p>The rule checker reports a violation of this rule if the size of a destination buffer in a <code>strcpy()</code> operation is insufficient to accommodate the source buffer and a null terminator.</p>
CERT C: Rec. MSC04-C	Use comments consistently and in a readable fashion	<p>The rule checker now detects nested comments and flags confusing code caused by nested comments. For instance:</p> <pre>x = y/**/ z //Noncompliant +0; x = y /*divisor:*/z //Noncompliant +0;</pre>

Rule	Description	Change
CERT C: Rec. STR07-C	Use the bounds-checking interfaces for string manipulation	The rule checker reports a violation of this rule if the size of a destination buffer in a <code>strcpy()</code> operation is insufficient to accommodate the source buffer and a null terminator.
CERT C: Rec. PRE09-C	Do not replace secure functions with deprecated or obsolescent functions	The rule checker reports a violation of this rule if the size of a destination buffer in a <code>strcpy()</code> operation is insufficient to accommodate the source buffer and a null terminator.

CERT C++

Rule	Description	Change
CERT C++: ARR39-C	Do not add or subtract a scaled integer to a pointer	<p>The rule checker now reports a violation when an improper pointer scaling is performed in multiple statements. For instance:</p> <pre>#define bigNum unsigned long long struct Collection { bigNum bn_A; bigNum bn_B; }; void foo(void) { size_t offset = offsetof(struct Collection, bn_B); struct Collection *s = (struct Collection *)malloc(sizeof(struct Collection)); //.... memset(s + offset, 0, sizeof(struct Collection) - offset); //Noncompliant }</pre> <p>Previously, the rule checker detected improper pointer scaling that are performed in a single statement.</p>
CERT C: Rule EXP36-C	Do not cast pointers into more strictly aligned pointer types	<p>The rule checker does not report violations of this rule if you use a source pointer that aligns correctly to the target type. For instance:</p> <ul style="list-style-type: none"> You use the <code>alignas</code> specifier to match the alignment of the source pointer with that of the target pointer. You use pointers returned by functions such as <code>aligned_alloc()</code>, <code>malloc()</code>, and <code>realloc()</code>. These pointers matches the alignment of the target pointer.
CERT C++: STR31-C	Guarantee that storage for strings has sufficient space for character data and the null terminator	<p>The rule checker reports a violation of this rule if the size of a destination buffer in a <code>strcpy()</code> operation is insufficient to accommodate the source buffer and a null terminator.</p>
CERT C++: STR50-CPP	Guarantee that storage for strings has sufficient space for character data and the null terminator	<p>The rule checker reports a violation of this rule if the size of a destination buffer in a <code>strcpy()</code> operation is insufficient to accommodate the source buffer and a null terminator.</p>

MISRA C:2012

Rule	Description	Change
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	The rule checker now reports a violation on these additional issues when checking C90 code: <ul style="list-style-type: none"> • Unnamed unions or empty structs • enum that has a trailing comma
MISRA C:2012 Rule 8.3	All declarations of an object or function shall use the same names and type qualifiers	The rule checker no longer reports a violation of this rule when a function with an empty parameter list is declared as <code>foo(void)</code> in one translation unit, and as <code>foo()</code> in another translation unit. For instance, consider a function that is declared in a header file as <pre>int func(void);</pre> When this function is declared in another source file, the rule checker considers these declaration styles as compliant: <pre>int func();</pre> or <pre>int func(void);</pre>
MISRA C:2012 Rule 17.8	A function parameter should not be modified	The rule checker does not report a violation of this rule when a function parameter is passed by address as <code>const</code> to another function. For instance: <pre>int foo(const int*); void bar(int param){ foo(&param); //Compliant }</pre> Because <code>foo()</code> accepts <code>const int*</code> objects, the parameter <code>param</code> is not modified when it is passed by address to <code>foo</code> .

MISRA C++:2008

Rule	Description	Change
MISRA C++:2008 Rule 7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function	The rule checker no longer reports a violation if a function returns a pointer or reference that is passed to the function as a parameter. For instance: <pre>int* func(int* param){ int* localCopy = param; //... return localCopy;//Compliant }</pre>

CWE

Rule	Description	Change
806	Buffer Access Using Size of Source Buffer	The Polyspace Bug Finder checker <code>Invalid use of standard library memory routine</code> is now mapped to this CWE identifier. The rule checker flags the use of memory library functions with invalid arguments, which might result in buffer overflows. See Mapping Between CWE Identifiers and Polyspace Results.

Custom Coding Rules

Rule	Description	Change
20.1	Source line length must not exceed specified number of characters	The rule checker now reports a violation if the number of characters is greater than the specified number. Previously, the rule checker reported a violation if the number of characters in a line was greater than or equal to the specified number.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access

Compatibility Considerations

If you checked your code for the preceding rules, you might see a change in the number of violations.

Updated Bug Finder defect checkers

In R2022b, these defect checkers have been updated.

Defect	Description	Update
Deallocation of previously deallocated pointer	Memory freed more than once without allocation	The defect checker reports a defect when you free memory after a possible zero-size reallocation. In some implementations, zero-size reallocations free the memory leading to a double-free defect.
Missing break of switch case	case block of switch statement does not end in a break or comment	The defect checker does not flag case blocks that end with a <code>[[fallthrough]]</code> . The <code>[[fallthrough]]</code> attribute is available in C++17 or later.
Missing return statement	Function with non-void return type does not return value on some paths	The defect checker takes into account cases where both the true and false expressions of a ternary operator involves a <code>throw</code> . If an execution path through a function involves such ternary operators, a <code>return</code> statement is not necessary along that path. The checker no longer flags these cases.
Use of dangerous standard function	Dangerous functions cause possible buffer overflow in destination buffer	The defect checker reports a defect if the size of a destination buffer in a <code>strcpy()</code> operation is insufficient to accommodate the source buffer and a null terminator.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access

Compatibility Considerations

If you checked your code for these defect checkers, you might see a change in the number of defects.



Reviewing Results

Additional Info in Result Details: See expected values and underlying types in constant overflows

In R2022b, in results showing constant overflows, you see more information on data types and values involved in the overflow. Unlike variables, constants do not have a data type explicitly specified in the code. When overflows occur in operations involving constants, it is useful to know the underlying data type used in the operation and the allowed range of the operation result.

For instance, the following information appears on the **Result Details** pane for the overflow on the second enum value here:

```
enum {
    a=0x7fffffff,
    b
} MyEnumA;
```

Integer constant overflow (Impact: Medium)  

Overflow on signed constant.

Additional Info:

Expected values: $[-2^{31} .. 2^{31}-1]$. The underlying type of this enum is *int*.

Actual values: 2^{31} (0x80000000).

Risk: Depending on your compiler, overflowing constants might be truncated or wrapped around and cause unexpected results.

You can see that the enum variable uses an underlying type `int`, which results in an expected range $[-2^{31}, 2^{31}-1]$.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access



Inputs Causing Defect: See example system inputs that lead to non-initialized variables

If a variable in your code is non-initialized only for certain system input values, in R2022b, the `Non-initialized variable` result is shown along with one possible combination of input values causing the defect.

For instance, in this example, the `if` statement has a missing catch-all `else` branch. As a result, the variable `sanitizedValue` can be non-initialized for certain values of the input `originalValue`.

```
int sanitize(int originalValue) {
    int sanitizedValue;
    if (originalValue <= -1)
        sanitizedValue = -1;
    else if(originalValue >= 1)
        sanitizedValue = 1;
    return sanitizedValue;
}
```

If you use the option `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`, the event list below the **Non-initialized variable** defect shows a possible value of the input `originalValue` that leads to the defect.

○ Non-initialized variable (Impact: High)  

Local variable `sanitizedValue` is read before it is initialized.

Additional Info:

Risk: Reading non-initialized memory can result in unexpected values.

Fix: Initialize the local variable before use.

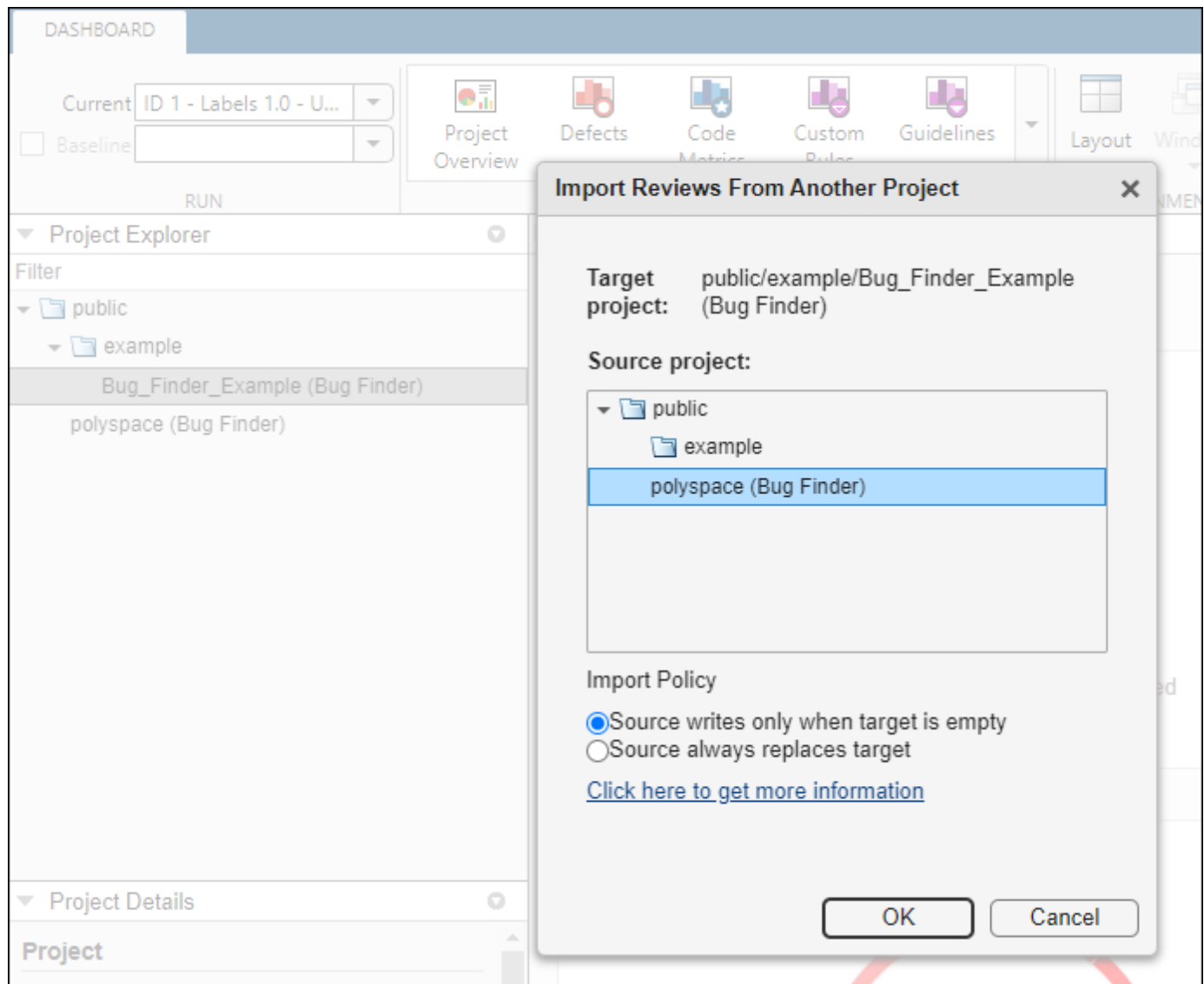
Result includes example values that lead to the defect.

	Event	File	Scope	Line
1	Function called by external code with input 'originalValue' Possible input value causing defect: 0	test.c	sanitize()	1
2	Entering function 'sanitize'	test.c	sanitize()	1
3	Declaration of variable 'sanitizedValue'	test.c	sanitize()	2
4	Entering else branch (if-condition false)	test.c	sanitize()	5
5	○ Non-initialized variable	test.c	sanitize()	7

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

Polyspace Access: Import review details and justifications from existing projects

In R2022b, you can import review information between Polyspace Access projects from the Polyspace Access interface or at the command line. For instance, if you justify findings in the file `test.c` in one project, you can reuse those justifications in another project that also uses file `test.c`.



After you import the review information, you can use filters in the Polyspace Access interface toolbar or view the **Result Details** pane to determine which review information was imported.

At the command-line, use option `polyspace-access -import-reviews` to import review information from a source project to a destination project. For example:

```
polyspace-access -import-reviews <source> -to-project-path <destination> -import-strategy <strategy> -host ...
```

Where `-import-strategy` specifies whether the source always overwrites the destination or not.

After you complete the import, you can also use the command `polyspace-access -export -imported-reviews` to generate a file that lists findings filtered by type of import, for instance all findings where the review information was overwritten during the import.

Previously, to import review information, you had to download results from Polyspace Access, import from previous results at the command line, and then upload the results again. The updated method takes fewer steps. You can also see in the result details which review information is new in the current result and which review information is imported.

See Import Review Information from Existing Polyspace Access Projects.

Products: Polyspace Access

polyspace-access Command: Manage review information and compare project runs

In R2022b, you can manage Polyspace Access projects programmatically by using the `polyspace-access` command to perform the operations in this table.

Operation	Command
Add, edit, or remove review information for a finding	<pre>polyspace-access -review <textFileOrfindingID> -project-path <fullProjectPath> <review-options> -host ...</pre> <p>Use this command to assign review information such as status, severity, comment, owner, or bug tracking tool ticket.</p> <p>To assign the same review information to multiple findings, specify the path of a text file where you store the finding IDs of those findings.</p> <p>For example, to perform a batch assignment for a large number of findings from project <code>public/example/Bug Finder</code>:</p> <ol style="list-style-type: none"> 1 Create a text file <code>findingsIDs.txt</code> where you store the finding IDs of all the findings that you want to review. In the file, list one finding ID per line. 2 Pass the file <code>findingsIDs.txt</code> to the <code>-review</code> command: <pre>polyspace-access -review findingsIDs.txt -project-path "public/example/Bug Finder" -set-status "To investigate" -set-severity Medium -host ...</pre> <p>To assign review information to a single finding, specify its finding ID.</p> <p>See also Make Batch Edits to Review Information for New Findings.</p>

Operation	Command
Compare two runs from the same project	<pre>polyspace-access -export <runID1> -baseline <runID2> - resolution <findingResolution> -output <filePath> - host ...</pre> <p>Here:</p> <ul style="list-style-type: none"> runID1 is the run ID of the run that you use as a current run. runID2 is the run ID of the run that you use as a baseline for the comparison. <p>The baseline run must be older than the current run.</p> <p>The command outputs a file with a list of findings filtered by one of these resolution types:</p> <ul style="list-style-type: none"> New Fixed Resolved Unresolved <p>See also Comparison Mode at the Command Line.</p>

You can use these commands as part of automation scripts when managing your projects. Previously, you performed the operations in this table only from the Polyspace Access user interface.

Products: Polyspace Access.

Changes in the polyspace-access Command Options

Options `-new-findings` and `-review-status` will be removed in a future release

Warns

The polyspace-access command options in this table will be removed in a future release. Use the new options instead.

Current Option	Usage	New Option
<code>-new-findings</code>	Select only new findings when you export or assign an owner to unassigned findings.	<p><code>-resolution new</code></p> <p>This new option lets you filter by additional resolution types such as Fixed, Unresolved, and resolved. See also</p> <p>You get an error if you use this option with a project that contains only one run. In this case, all findings are new and you do not need to specify this option.</p>

Current Option	Usage	New Option
- review-status <i>Unreviewed</i> " <i>To investigate</i> " " <i>To fix</i> " <i>Justified</i> " <i>No action planned</i> " " <i>Not a defect</i> " <i>Other</i>	Select findings with one of the specified review status when you export or assign an owner to unassigned findings.	- status <i>Unreviewed</i> " <i>To investigate</i> " " <i>To fix</i> " <i>Justified</i> " <i>No action planned</i> " " <i>Not a defect</i> " <i>Other</i>

For accuracy and consistency, two filter have been updated in the Polyspace Access interface. The filter labels changes are:

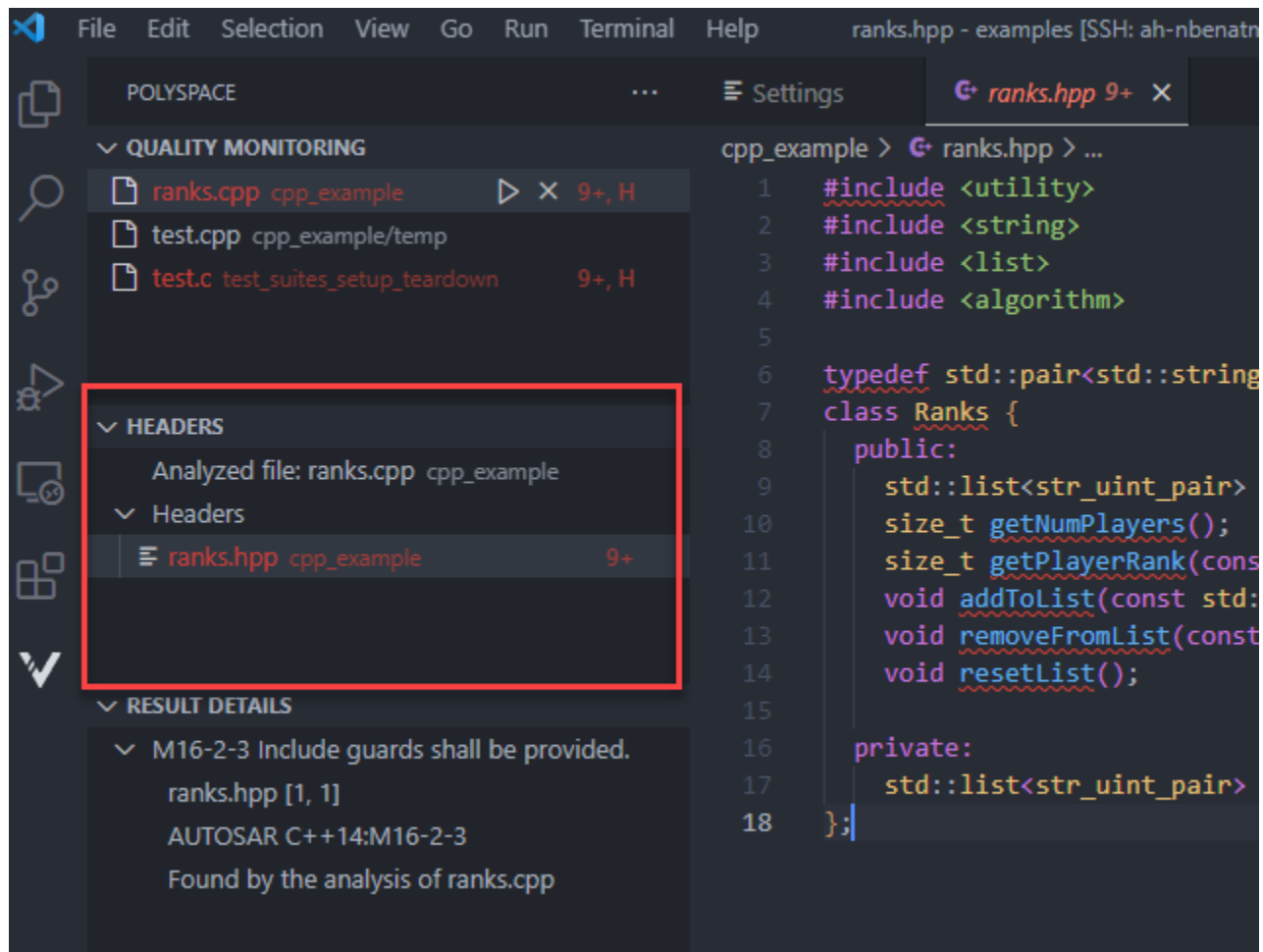
- State → Resolution
- Review Status → Status

See Filter Polyspace Access Results.

Products: Polyspace Access

Polyspace Extension for Visual Studio Code (VSCode): Review findings in header files

In R2022b, when you analyze a file with Polyspace as You Code, you can review and address findings in the header files that are included by the analyzed file if those headers are in the same folder as the analyzed file. To view the headers included by a file, click an already analyzed file in the **Quality Monitoring** view. The new **Headers** view shows included files only if Polyspace reports findings in those files. The reported findings are contextual to the analyzed file that includes the headers.



Select a file in the **Headers** view to open its findings in the **Output** panel. Polyspace also reports findings in a header file if you explicitly analyze that header file.

Previously, Polyspace analyzed but did not report findings for included headers that were in the same folder as the analyzed file. For header files that are not in the same folder as the analyzed file, Polyspace takes those headers into consideration for compilation but does not analyze them.

See also: **HEADERS** view.

Product: Polyspace Access (Polyspace as You Code).

Polyspace Extension for Visual Studio Code (VSCode): Autocomplete code annotations using catalog of predefined comments

In R2022b, if your team or organization uses a predefined set of comments to justify analysis results, you can pass those comments to the Polyspace as You Code extension as a JSON catalog. When you start typing an annotation to justify a Polyspace result, the Polyspace as You Code extension autocompletes the Polyspace annotation syntax and the comments from the catalog are available from a dropdown when you fill in the comment of the annotation.

```

Settings  ranks.cpp 9+ X
cpp_example > ranks.cpp > getNumPlayers()
1  #include "ranks.hpp"
2
3  std::list<str_uint_pair> Ranks::getPlayers() {
4      return playersList;
5
6  size_t Ranks::getNumPlayers() {
7      return playersList.size();
8  }
9  size_t Ranks::getPlayerRank(const std::string& name) {
10     size_t rank;
11     auto pos = std::find_if(playersList.begin(),
12                            playersList.end(),
13                            [name](str_uint_pair i) {
14                                return i.first == name; });
15     (pos != playersList.end())
16     ? rank = std::distance(playersList.begin(), pos) + 1
17     : rank = -1; // If player not found return -1
18     //instead of distance to one past end of list
19     return rank;
20 }
21 void Ranks::addToList(const std::string& name, const size_t& score) {
22     if (playersList.size()) {
23         // Find first instance of smaller score and insert

```

Previously, you had to type your justification each time you added an annotation in the editor.

See also [Use a Justification Catalog to Autocomplete Annotations in Visual Studio Code](#).

Product: Polyspace Access (Polyspace as You Code).

Polyspace Extension for Visual Studio Code (VSCode): Get started faster with the Polyspace as You Code walkthroughs

In R2022b, three new walkthroughs are available in VSCode to help you learn about Polyspace as You Code. The walkthroughs cover initial configuration with customizable options, running an analysis, addressing results, and using a baseline with your results.

For more information, see:

- [Perform Polyspace as You Code Analysis in Visual Studio Code](#)
- [Configure Polyspace as You Code in Visual Studio Code](#)
- [Run and Review Results in Polyspace as You Code for Visual Studio Code](#)
- [Configure and Download Baseline with Polyspace as You Code](#)

Products: Polyspace Access (Polyspace as You Code)

Results Export: Updated color property when you export Code Metrics results to JSON SARIF format

In R2022b, the Polyspace Code Metrics **color** property is empty when you export Polyspace results to a JSON SARIF file.

Previously, the value of this **color** property was **BLACK**. The updated value is more consistent with the Code Metrics color for other Polyspace export formats.

Compatibility Considerations

If you pass the JSON SARIF file to a script or third party tool that uses the Code Metrics color to manage Polyspace results, update the script or tool configuration to filter on an empty **color** field instead of a **BLACK** field value.

Changes in number and locations of results showing conflicting declarations

In R2022b, checkers that involve multiple source locations, such as checkers for conflicting declarations, might show the following changes:

- There might be a change in the result location.

For instance, if a checker flagged two conflicting declarations and the result appeared on one of the declarations, it might now appear on the other one. The algorithm for localizing the results have been updated and the results no longer depend on the order in which files are specified during analysis.

- For certain checkers, if two results were previously shown for the same conflicting pair of declarations, you now only see a single result.

For instance, the checker for AUTOSAR C++14 Rule M10-2-1 flags conflicting names in the same multiple inheritance hierarchy. Previously, the checker showed two results for a pair of conflicting names. You now see only a single result.

Compatibility Considerations

For checkers that involve multiple source locations, you might see one of the following:

- A few of the results might be localized on a different line from before. If you had entered code annotations justifying such a result, you might have to move the annotation to the new location.
- If two results were shown for the same conflicting pair of declarations, you see a single result now. In this case, if you had code annotations justifying the two results, they should continue to work for the single result.

Changes in locations of results in macro instances or template instantiations

In R2022b, results in macro instances are shown on the macro definition for more checker types. Previously, for certain checkers, separate instances of a macro showed essentially the same result (with differences only in the additional information such as expected/actual values and events). Results that are essentially the same but appear in different macro instances now get rolled up to the macro definition. For these results, on the **Result Details** pane:

- The **Additional Info** section of the result message shows one set of additional information that is relevant to one or more instances of the macro.

- The **Event** list below the result shows the macro instances where the above result message is relevant.

The display does not show other macro instances where the additional information is different.

The same logic is used for the result display when a result appears in multiple instantiations of a template. Results that are essentially the same but appear in different instantiations of a template get rolled up to the template definition:

- The **Additional Info** section of the result message contains additional information that is specific to one or more instantiations of the template.
- The **Event** list below the result shows the template instantiations where the above result message is relevant.

The display does not show other template instantiations where the additional information is different.

Compatibility Considerations

You might see a change in the number of results in macro instances or template instantiations.

R2022a

Version: 3.6

New Features

Bug Fixes

Compatibility Considerations

Analysis Setup

Updated Clang Compiler Support: Set up Polyspace analysis for code compiled by using Clang versions 6.x to 11.x

In R2022a, Polyspace supports Clang compiler versions 6.x to 11.x natively. If you build your source code by using these Clang compiler versions, you can specify the corresponding compiler option values for your Polyspace analysis.

- | Target Environment | |
|-----------------------|----------|
| Compiler | clang6.x |
| Target processor type | x86_64 |

clang6.x for LLVM release 6.0.0 and 6.0.1.

- | Target Environment | |
|-----------------------|----------|
| Compiler | clang7.x |
| Target processor type | x86_64 |

clang7.x for LLVM release 7.0.0, 7.0.1, and 7.1.0.

- | Target Environment | |
|-----------------------|----------|
| Compiler | clang8.x |
| Target processor type | x86_64 |

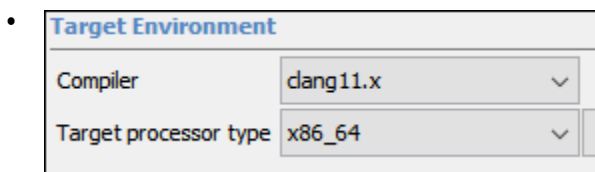
clang8.x for LLVM release 8.0.0 and 8.0.1.

- | Target Environment | |
|-----------------------|----------|
| Compiler | clang9.x |
| Target processor type | x86_64 |

clang9.x for LLVM release 9.0.0 and 9.0.1.

- | Target Environment | |
|-----------------------|-----------|
| Compiler | clang10.x |
| Target processor type | x86_64 |

clang10.x for LLVM release 10.0.0 and 10.0.1.



`clang11.x` for LLVM release 11.0.0, 11.0.1, and 11.1.0.

At the command line, you specify a compiler by using the option `-compiler`. For instance:

```
polyspace-bug-finder-server -sources file.c -compiler clang9.x ...
```

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

Because of the native support, you can now set up a Polyspace project without knowing the internal workings of these compilers. The analysis can interpret macros that are implicitly defined by the compilers and compiler-specific language extensions, such as keywords and pragmas.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Incremental Compilation: Run faster analysis by compiling only files edited since previous analysis

In R2022a, you can run an incremental mode during the compilation phase of a Bug Finder analysis. In this mode, after you run an initial analysis, Polyspace recompiles a file during the compilation phase only if one of the following is true:

- You make edits to a source file.
- You make edits to a header file. Polyspace recompiles all the files that include this header file.
- You make changes to the analysis options. Polyspace recompiles all the files.

To enable this incremental mode, use option `-incremental`. Make sure that you use the same results folder for all later analysis runs. In the Polyspace user interface, specify this option in the `Other` field.

Product: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server .

Configuration from Build System: Import compiler macro definitions automatically without tracing build

In R2022a, Polyspace can import macro definitions from compilers that provide native options to list the compiler predefined macros, for instance `gcc -dm -E`. Polyspace passes the relevant options to the compiler and extracts the macro definitions from the output, without tracing your build.

With this new macro import strategy, when you use `polyspace-configure` to create a Polyspace project or to generate an analysis options file from your build system, Polyspace automatically attempts to import macro definitions by trying import strategies in this order of priority:

- 1 From compiler by using native compiler options to list macro definitions. This strategy is available only for compilers that support listing macro definitions.

- 2 From source code tokens. Polyspace uses every non-keyword token in your source code to query your compiler for macro definitions. This strategy is available only if Polyspace can trace your build. This strategy is not available if you use a JSON compilation database to extract your build configuration.
- 3 From a predefined allow list. Polyspace uses the allow list to query your compiler for macro definitions.

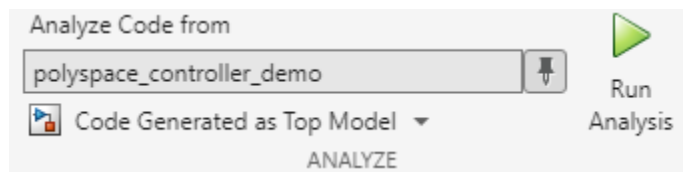
Previously, if you used a JSON compilation database or if a third-party software prevented `polyspace-configure` from tracing your build, for example System Integrity Protection on Mac OS, Polyspace attempted only the allow list strategy to import macro definitions.

See also `polyspace-configure`.

Product: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code).

Simulink Support: Polyspace updates generated code when model changes

In R2022a, the workflow for analyzing code generated from Simulink models is simplified. If you have Embedded Coder, Polyspace checks the model for any new change every time you click **Run Analysis**. If you have changed the model since the last Polyspace analysis, Polyspace invokes Embedded Coder to refresh the generated code before starting the new analysis.



See Run Polyspace Analysis on Code Generated with Embedded Coder.

Previously, to reflect an updated model in your code, you regenerated code explicitly. Now, Polyspace detects changes in your model and regenerates code if the model changes.

Additional Considerations: After updating your model, you still need to generate code explicitly if any of the following is true:

- You do not use Embedded Coder to generate code.
- The model is configured to generate code as a model reference.

Product: Polyspace Bug Finder (Desktop).

MATLAB Coder Support: Polyspace analysis takes into account MATLAB Coder settings for nonfinite numbers

In R2022a, a Polyspace analysis of C/C++ code generated from MATLAB code has better support for nonfinite numbers. If the option **Support nonfinite numbers** is enabled in your code generation settings (default behavior), a subsequent Polyspace analysis of the generated code takes this information into account. The analysis runs with the Polyspace option `Consider non finite floats (-allow-non-finite-floats)` enabled and correctly interprets infinities and NaN-s.

Previously, the analysis ignored the MATLAB Coder™ specification and produced false positives when the generated code explicitly used infinities and NaN-s.

Product: Polyspace Bug Finder (Desktop).

New Code Behaviors: Specify behaviors such as critical, memory-managing, or real-time to code elements to enable associated checks

In R2022a, there are new code behaviors for the analysis option `-code-behavior-specifications`. You can specify objects as critical and functions as memory-managing functions, exception handling functions, and real-time functions. Polyspace checkers can then determine if the variables and functions conform to rules associated with those behaviors. Consider this code:

```
class UserAccount
{
public:
    UserAccount(char *username, char *password)
    {
        //...
    }
    char password[15];
};
```

Here, the critical object `password` is declared as public, perhaps inadvertently. This critical information remains vulnerable to malicious client classes. Because Polyspace does not know that this object is critical, this issue is not flagged. Specify `password` as critical by including this entry in the code behavior specification XML:

```
<member name="password" kind="variable">
  <behavior name="CRITICAL_DATA"/>
</member>
```

After you specify the code behavior, Polyspace raises the defect `Critical data member is not private`.

In R2022a, these code behaviors are new:

- **EXCEPTION_HANDLING:** Specify that a function handles exceptions by using an entry in the code behavior specification XML with this behavior. For instance:

```
<function name="function_name">
  <behavior name="EXCEPTION_HANDLING"/>
</function>
```

See AUTOSAR C++14 Rule A15-0-7.

- **MANAGES_MEMORY:** Specify that a function manages dynamic memory by using an entry in the code behavior specification XML with this behavior. For instance:

```
<function name="function_name">
  <behavior name="MANAGES_MEMORY"/>
</function>
```

See AUTOSAR C++14 Rule A18-5-7.

- **REAL_TIME_FUNC:** Specify that a function runs in real time by using an entry in the code behavior specification XML with this behavior. For instance:

```
<function name="function_name">
  <behavior name="REAL_TIME_FUNC"/>
</function>
```

See AUTOSAR C++14 Rule A18-5-7.

- **CRITICAL_DATA**: Specify a data member as critical by using an entry in the code behavior specification XML with this behavior. For instance:

```
<member name="password" kind="variable">
  <behavior name="CRITICAL_DATA"/>
</member>
```

See Critical data member is not private.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

Modifying Checkers: Create list of keywords to prohibit and check for use of keywords from the list

In R2022a, you can define a blocklist of keywords to forbid from your source code. The Bug Finder checker `Use of a forbidden keyword` checks if a keyword from this list appears in your sources.

A keyword might be blocked for one of these reasons:

- The keyword is prone to misuse or makes the code difficult to maintain.
- The keyword is being deprecated as part of a migration, for instance, from C++98 to C++11.

As part of a migration, you can make a list of keywords that need to be replaced and use this checker to identify their use.

See also `Flag Deprecated or Unsafe Functions or Keywords Using Bug Finder Checkers`.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

Ignoring Code Annotations: Perform a worst-case analysis to see all results including previously justified ones

In R2022a, you can run a Polyspace analysis that ignores all code annotations justifying Polyspace results.

To avoid reviewing a result multiple times, you can add annotations specific to Polyspace to your code with review information such as justification for a result. Later runs take into account these annotations and show the review information in the results. In some cases, you might want to run a clean analysis as if the results have not been previously reviewed. Starting in R2022a, you can use the option `-ignore-code-annotations` to run such an analysis with no history. The analysis ignores the code annotations and shows all annotated results without any review information from the annotations.

See also:

- `-ignore-code-annotations`

- Annotate Code and Hide Known or Acceptable Results

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Functionality Being Removed: Polyspace desktop integration with Eclipse IDE

The Polyspace desktop product integration with Eclipse-based IDEs will be removed in a future release.

Product: Polyspace Bug Finder (Desktop).

Compatibility Considerations

To continue monitoring the quality of your code from your Eclipse-based IDE, use the Polyspace as You Code Eclipse plugin. See [Install Polyspace as You Code Plugin in Eclipse](#)

Functionality Removed: Polyspace stubs for Standard Template Library

Polyspace stubs for the C++ Standard Template Library (STL) have been removed. These stubs conform to the older C++98 Standard and were meant for quickly getting started with a C++ analysis. In most situations, your compiler implementation of the Standard Template Library is required for successful compilation of a C++ project with Polyspace.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

Compatibility Considerations

If you were using the Polyspace stubs when running an analysis, you will encounter a compilation error. To work around the error, provide your compiler implementation of the Standard Template Library for analysis.

Functionality Removed: Compilation assistant

The Polyspace compilation assistant is removed in R2022a. You get an error if you use option `-easy-setup-compile` at the command line.

Product: Polyspace Bug Finder (Desktop).

Compatibility Considerations

If you use the compilation assistant in your Polyspace project:

- The option is automatically removed in the Polyspace user interface.
- Remove option `-easy-setup-compile` at the command line.

Alternatively, when you set up your Polyspace project:

- Use the `Compiler` (`-compiler`) option to specify a compiler that Polyspace supports natively if you compile your code by using that compiler.
- Use `polyspace-configure` to trace your build command and to obtain your compiler configuration. See `polyspace-configure`.

Changes in analysis options and binaries

Option `-no-stl-stubs` no longer supported

Errors

The option **No STL stubs** (`-no-stl-stubs`) is no longer supported. This option disabled use of Polyspace stubs for the C++ Standard Template Library (STL). Since these stubs are no longer provided in R2022a, there is no requirement to disable their use.

The argument `data_race_all` for the option `-checkers` will be removed in a future release

Warns

The argument `data_race_all` for the option `-checkers` will be removed in a future release. To detect data races in atomic operations, use the new analysis options `-detect-atomic-data-race`. See:

- `-detect-atomic-data-race`
- `Extend Data Race Checkers to Atomic Operations`

If you use `-checkers all` to run a Bug Finder analysis, you might see a reduced number of defects because this command no longer checks for data races in atomic operations. To get back to the previous behavior, specify the option `-detect-atomic-data-race`.

The option `-detect-atomic-data-race` is a command-line only option. To use this option from the Polyspace user interface, in the **Advanced Settings** node, specify the option in the **Other** field.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

The code metrics estimating sizes of local variables will be removed in a future release

Warns

The code metrics `Lower Estimate of Size of Local Variables` and `Higher Estimate of Size of Local Variables` will be removed from Bug Finder in a future release. To calculate these metrics, use the option `-stack-usage` in Code Prover. See `Calculate stack usage (-stack-usage)`.

You get a warning if you calculate the higher and lower estimates of local variables by using Bug Finder.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access (Polyspace as You Code)

Changes in MATLAB function, options object and properties

Property `NoStlStubs` no longer supported

Errors

The property `NoStlStubs` is no longer supported. To disable use of Polyspace stubs for the C++ Standard Template Library (STL), you enabled this property as follows:

```
proj = polyspace.Project;  
proj.Configuration.InputsStubbing.NoStlStubs = true;
```

Since these stubs are no longer provided in R2022a, there is no requirement to disable their use.

Analysis Results

AUTOSAR C++14 Support: Check for 363 AUTOSAR C++14 rules including 18 new rules

In R2022a, you can look for violations of these AUTOSAR C++14 rules in addition to previously supported rules.

AUTOSAR Rule	Description	Polyspace Checker		
A0-4-4	Range, domain and pole errors shall be checked when using math functions.	AUTOSAR A0-4-4	C++14	Rule
A5-2-5	An array or container shall not be accessed beyond its range.	AUTOSAR A5-2-5	C++14	Rule
A5-5-1	A pointer to member shall not access non-existent class members.	AUTOSAR A5-5-1	C++14	Rule
A6-5-1	A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used.	AUTOSAR A6-5-1	C++14	Rule
A7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	AUTOSAR A7-2-1	C++14	Rule
A13-1-3	User defined literals operators shall only perform conversion of passed parameters.	AUTOSAR A13-1-3	C++14	Rule
A13-5-4	If two opposite operators are defined, one shall be defined in terms of the other.	AUTOSAR A13-5-4	C++14	Rule
A15-0-2	At least the basic guarantee for exception safety shall be provided for all operations. In addition, each function may offer either the strong guarantee or the nothrow guarantee.	AUTOSAR A15-0-2	C++14	Rule
A15-0-3	Exception safety guarantee of a called function shall be considered.	AUTOSAR A15-0-3	C++14	Rule
A15-0-7	Exception handling mechanism shall guarantee a deterministic worst-case time execution time.	AUTOSAR A15-0-7	C++14	Rule
A17-1-1	Use of the C Standard Library shall be encapsulated and isolated.	AUTOSAR A17-1-1	C++14	Rule
A18-1-4	A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type.	AUTOSAR A18-1-4	C++14	Rule

AUTOSAR Rule	Description	Polyspace Checker		
A18-5-7	If non-real-time implementation of dynamic memory management functions is used in the project, then memory shall only be allocated and deallocated during non-real-time program phases.	AUTOSAR A18-5-7	C++14	Rule
A20-8-7	A <code>std::weak_ptr</code> shall be used to represent temporary shared ownership.	AUTOSAR A20-8-7	C++14	Rule
A23-0-2	Elements of a container shall only be accessed via valid references, iterators, and pointers	AUTOSAR A23-0-2	C++14	Rule
A25-1-1	Non-static data members or captured values of predicate function objects that are state related to this object's identity shall not be copied	AUTOSAR A25-1-1	C++14	Rule
A25-4-1	Ordering predicates used with associative containers and STL sorting and related algorithms shall adhere to a strict weak ordering relation.	AUTOSAR A25-4-1	C++14	Rule
M0-1-8	All functions with void return type shall have external side effect(s).	AUTOSAR M0-1-8	C++14	Rule

See also AUTOSAR C++14 Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access

Bug Finder Checkers: Check for inefficient string methods, critical data leaks, and other issues

In R2022a, using new Bug Finder checkers, you can check for these additional types of issues.

Defect	Description
Critical data member is not private	A critical data member is declared public.
Declaration of catch for generic exception	A catch block handles a generic exception that might have many different subtypes.
Declaration of throw for generic exception	A function is specified to throw a generic exception, which might have many subtypes.
Expensive allocation in loop	Fixed sized memory is allocated or deallocated in a loop
Expensive use of container's insertion methods	One of the insertion methods of a container is used to insert a temporary object.
Expensive use of string functions from the C standard library	String functions from the C standard library are used inefficiently.

Defect	Description
Expensive return of a const object	A function returns objects by copy instead of by move because the returned object is declared as a const
Expensive use of substr() to shorten a std::string	The method std::string::substr() is called to shorten an std::string object.
Improper erase-remove idiom	Container's erase() is not called or called improperly following a call to std::remove().
Inefficient use of for loop	Range-based for loop can perform equivalent iteration more efficiently
Invalid iterator usage	Iterator use is not a valid C++ syntax.
Method not const	A method that can be made const is not marked const.
Public static field is not const	A static and public field of a struct or class is not marked as a const.
Uncertain memory cleaning	The code clears sensitive information from memory but compiler optimization might leave the information untouched
Useless preprocessor conditional directive	Preprocessor conditional directive is always true or always false
Use of a forbidden keyword	A keyword that appears in a blacklist of forbidden keywords is used.

For the full list of checkers, see Defects.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access

CERT C++ Support: Check for violations associated with exception handling

In R2022a, you can look for violations of this CERT C++ rule in addition to previously supported rules.

CERT C++ Rule	Description	Polyspace Checker
CTR51-CPP	Use valid references, pointers, and iterators to reference elements of a container	CERT C++: CTR51-CPP
CTR57-CPP	Provide a valid ordering predicate.	CERT C++: CTR57-CPP
CTR58-CPP	Predicate function objects should not be mutable	CERT C++: CTR58-CPP
ERR56-CPP	Guarantee exception safety	CERT C++: ERR56-CPP
OOP55-CPP	Do not use pointer-to-member operators to access nonexistent members	CERT C++: OOP55-CPP

See also CERT C++ Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access

MISRA C++:2008 Support: Check for void functions with no external side effects

In R2022a, you can look for violations of this MISRA C++:2008 rule in addition to previously supported rules.

CERT C++ Rule	Description	Polyspace Checker
0-1-8	All functions with void return type shall have external side effect(s)	MISRA C++:2008 Rule 0-1-8

See also CERT C++ Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access

CWE Support: Check for violation of 8 new CWE rules about class access, exception, and other issues

In R2022a, you can check for violation of this CWE rule in addition to previously supported rules.

CWE Rule	Description	Polyspace Checkers
14	Compiler Removal of Code to Clear Buffers	Uncertain memory cleaning
396	Declaration of Catch for Generic Exception	Declaration of catch for generic exception
397	Declaration of Throws for Generic Exception	Declaration of throw for generic exception
489	Active debug code	Use of a forbidden function Note that the checker can be used to forbid any function. To adapt the checker to CWE-489, only forbid those functions that correspond to debug code, for instance, functions from the Windows debug API. For your convenience, some debug functions are already defined in the sample XML of code behavior specifications.
500	Public Static Field Not Marked Final	Public static field is not const

CWE Rule	Description	Polyspace Checkers
766	Critical Data Element Declared Public	Critical data member is not private
806	Buffer Access Using Size of Source Buffer	<ul style="list-style-type: none"> Invalid use of standard library string routine Destination buffer overflow in string manipulation
839	Numeric Range Comparison Without Minimum Check	<ul style="list-style-type: none"> Pointer dereference with tainted offset Tainted sign change conversion

For the full mapping between CWE rules and Polyspace Bug Finder defect checkers, see CWE Coding Standard and Polyspace Results.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access

Updated Bug Finder defect checkers

In R2022a, these defect checkers have been updated.

Defect	Description	Update
Pointer or reference to stack variable leaving scope	Pointer to local variable leaves the variable scope	<p>Starting in R2022a, Polyspace assumes that the local objects within a function definition are in the same scope. Consider this code:</p> <pre>void foo(){ int* p; { int tmp = 4; p = &tmp; } int q = *p;//Compliant }</pre> <p>Polyspace assumes that tmp, p and q are in the same scope.</p>
Data race including atomic operations	Multiple tasks perform unprotected operations on shared variable	<p>Previously, you used this checker to detect data races in operations that can be performed in one machine instruction. This checker is no longer supported and it will be removed in a future release.</p> <p>Starting in R2022a, use the option <code>-detect-atomic-data-race</code> to detect data races in operations that can be performed in one machine instruction.</p>

Defect	Description	Update
Invalid use of standard library floating point routine	Wrong arguments to standard library function	<p>Starting in R2022a, Polyspace flags an invalid integer argument when calling the floating point functions defined in the <code>std</code> namespace in C++11 and later. For instance:</p> <pre>#include <cmath> void wrong_usage(){ double res; res = std::sqrt((int)-5);// Defect res = std::log(0);// Defect res = sqrt((int)-5);// Defect res = log(0);// Defect }</pre> <p>Previously, Polyspace did not flag invalid use of integer arguments when calling floating point functions.</p>
Self assignment not tested in operator	Copy assignment operator does not test for self-assignment	<p>Starting in R2022a, Polyspace does not raise this defect when you implement a copy assignment operator by using the copy-and-swap idiom. For instance:</p> <pre>#include <algorithm> class MyClass { public: MyClass& operator=(MyClass& other) { //Compliant MyClass tmp(other); swap(tmp); return *this; } void swap(MyClass& other) noexcept{ std::swap(obj_, other.obj_); } private: int * obj_; };</pre> <p>The use of copy-and-swap idiom avoids the unexpected behavior that might occur when self-assigning.</p>

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access

Compatibility Considerations

If you checked your code for the preceding defect checkers, you might see a change in the number of defects.

Changes to coding standards checking

In R2022a, these changes have been made in the checking of previously supported rules.

AUTOSAR C++14

Rule	Description	Change
AUTOSAR C++14 Rule A3-8-1	An object shall not be accessed outside of its lifetime.	<p>Starting in R2022a, Polyspace assumes that the local objects within a function definition are in the same scope. Consider this code:</p> <pre>void foo(){ int* p; { int tmp = 4; p = &tmp; } int q = *p;//Compliant } </pre> <p>Polyspace assumes that tmp, p and q are in the same scope.</p>
AUTOSAR C++14 Rule M5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	<p>Starting in R2022a, Polyspace no longer raises this checker when the result of a pointer arithmetic operation results in <code>nullptr</code>. For instance, consider this code:</p> <pre>void g(int *p); void add(int* p, int n) { g(p + n); //Compliant } void foo() { add(nullptr, 0); } </pre> <p>The pointer arithmetic in <code>add()</code> results in a <code>nullptr</code>. Polyspace does not flag this operation.</p>
AUTOSAR C++14 Rule M5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression.	<p>Polyspace no longer considers function calls to be cvalues. For instance:</p> <pre>class A { public: double memFunc() const; void foo(); }; void rateControl::foo() { static_cast<int>(memFunc());//Compliant //... } </pre> <p>In the body of the function <code>foo()</code>, return value of the call to <code>A::memfunc()</code> is a <code>double</code>. This return value is cast to <code>int</code>. Because function calls are not cvalue expressions, Polyspace does not flag this operation.</p>

Rule	Description	Change
AUTOSAR C++14 Rule A2-7-3	All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation.	<p>Starting in R2022a, Polyspace no longer raises a violation on declarations local to an object in a function when they are preceded by a comment. For instance:</p> <pre data-bbox="565 415 1015 703"> // A function void f1(int i) { // A const int const N = 20; // struct local to f1 struct s1{ //int int ii = 0; //Compliant }; } </pre>
AUTOSAR C++14 Rule M0-1-4	A project shall not contain non-volatile POD variables having only one use.	<p>Starting in R2022a, Polyspace raises a violation if the only use of a variable is to pass its address to a function. For instance:</p> <pre data-bbox="565 871 1047 1092"> void getReading(int*); void foo() { int sensorData;//Noncompliant getReading(&sensorData); } </pre>

CERT C

Rule	Description	Change
CERT C: Rule DCL30-C	Declare objects with appropriate storage durations.	<p>Starting in R2022a, Polyspace assumes that the local objects within a function definition are in the same scope. Consider this code:</p> <pre>void foo(){ int* p; { int tmp = 4; p = &tmp; } int q = *p;//Compliant } </pre> <p>Polyspace assumes that tmp, p and q are in the same scope.</p>
CERT C: Rule CON43-C	Do not allow data races in multithreaded code.	<p>When checking for data races, Polyspace ignores operations that can be performed in one machine instruction. Previously, to check these atomic instructions for data races, you used <code>-checkers</code> with the argument <code>data_race_all</code>. This argument is no longer supported.</p> <p>Starting in R2022a, use the option <code>-detect-atomic-data-race</code> to detect data races in operations that can be performed in one machine instruction.</p>
CERT C: Rule INT36-C	Converting a pointer to integer or integer to pointer.	<p>Starting in R2022a, Polyspace raises a flag when any integral types are cast into a raw pointer type.</p> <pre>void func(unsigned int flag) { char *ptr; unsigned long long int_same_as_ptr ; unsigned int int_smaller_than_ptr; /*int to ptr of same size*/ ptr = (char *)int_same_as_ptr; //Noncompliant /*int to ptr of different size*/ ptr = (char *)int_smaller_than_ptr; //Noncompliant } </pre> <p>Previously, Polyspace flagged conversion from an integral object to raw pointer when their sizes were mismatched.</p>

CERT C++

Rule	Description	Change
CERT C++: DCL30-C, CERT C++: EXP54-CPP	Declare objects with appropriate storage durations.	<p>Starting in R2022a, Polyspace assumes that the local objects within a function definition are in the same scope. Consider this code:</p> <pre>void foo(){ int* p; { int tmp = 4; p = &tmp; } int q = *p;//Compliant } </pre> <p>Polyspace assumes that tmp, p and q are in the same scope.</p>
CERT C++: CON43-C	Declare objects with appropriate storage durations.	<p>When checking for data races, Polyspace ignores operations that can be performed in one machine instruction. Previously, to check these atomic instructions for data races, you used <code>-checkers</code> with the argument <code>data_race_all</code>. This argument is no longer supported.</p> <p>Starting in R2022a, use the option <code>-detect-atomic-data-race</code> to detect data races in operations that can be performed in one machine instruction.</p>
CERT C++: INT36-C	Converting a pointer to integer or integer to pointer.	<p>Starting in R2022a, Polyspace raises a flag when any integral types are cast into a raw pointer type.</p> <pre>void func(unsigned int flag) { char *ptr; unsigned long long int_same_as_ptr ; unsigned int int_smaller_than_ptr; /*int to ptr of same size*/ ptr = (char *)int_same_as_ptr; //Noncompliant /*int to ptr of different size*/ ptr = (char *)int_smaller_than_ptr; //Noncompliant } </pre> <p>Previously, Polyspace flagged conversion from an integral object to raw pointer when their sizes were mismatched.</p>

ISO/IEC TS 17961

Rule	Description	Change
ISO/IEC TS 17961 [addressc ape]	Declare objects with appropriate storage durations.	<p>Starting in R2022a, Polyspace assumes that the local objects within a function definition are in the same scope. Consider this code:</p> <pre>void foo(){ int* p; { int tmp = 4; p = &tmp; } int q = *p;//Compliant }</pre> <p>Polyspace assumes that tmp, p and q are in the same scope.</p>

MISRA C:2012

Rule	Description	Change
MISRA C:2012 Rule 8.13	A pointer should point to a const-qualified type whenever possible.	<p>Starting in R2022a, Polyspace does not raise a violation of this rule when a nonconst pointer is passed to a function, and its pointed data is modified by using a local copy of the pointer. For instance:</p> <pre>typedef struct { int a; char text[20]; } myStruct; void foo(myStruct* bar)//Compliant { char* ptr; ptr = bar->text; *ptr = 'a'; }</pre> <p>Because the data pointed to by <code>bar</code> is modified by the local copy <code>ptr</code>, <code>bar</code> cannot be a const. Polyspace does not flag the nonconst pointer <code>bar</code>.</p>
MISRA C:2012 Rule 20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.	<p>Starting in R2022a, Polyspace does not raise a violation of this rule when a macro parameter is preceded or followed by an identifier. For instance:</p> <pre>#define TYPEDECL(type, var) static type* var TYPEDECL(const int, foo)//Compliant</pre> <p>Because the macro <code>TYPEDECL</code> expands into a variable declaration, operand precedence is not relevant. Polyspace does not flag the lack of parentheses in the macro.</p>
MISRA C:2012 Rule 14.2	A for loop shall be well-formed	<p>Starting in R2022a, Polyspace does not raise a flag if a for loop tests the condition by using a binary operation. For instance:</p> <pre>int i,b; //... for(i = 0; (i+b)<5;++i)//Compliant</pre>

MISRA C++:2008

Rule	Description	Change
MISRA C++:2008 Rule 5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	<p>Starting in R2022a, Polyspace no longer raises this checker when the result of a pointer arithmetic operation results in <code>nullptr</code>. For instance, consider this code:</p> <pre>void g(int *p); void add(int* p, int n) { g(p + n); } void foo() { add(nullptr, 0); //Compliant } </pre> <p>The pointer arithmetic in <code>add()</code> results in a <code>nullptr</code>. Polyspace does not flag this operation.</p>
MISRA C++:2008 Rule 5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression	<p>Polyspace no longer considers function calls to be cvalues. For instance:</p> <pre>class A { public: double memFunc() const; void foo(); }; void rateControl::foo() { static_cast<int>(memFunc()); //Compliant //... } </pre> <p>In the body of the function <code>foo()</code>, return value of the call to <code>A::memfunc()</code> is a <code>double</code>. This return value is cast to <code>int</code>. Because function calls are not cvalue expressions, Polyspace does not flag this operation.</p>
MISRA C++:2008 Rule 0-1-4	A project shall not contain non-volatile POD variables having only one use.	<p>Starting in R2022a, Polyspace raises a violation if the only use of a variable is to pass its address to a function. For instance:</p> <pre>void getReading(int*); void foo() { int sensorData; //Noncompliant getReading(&sensorData); } </pre>

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access

Compatibility Considerations

If you checked your code for the preceding rules, you might see a change in the number of violations.

Reviewing Results

Results Export: Generate more accurate keys to track results across analysis runs

In R2022a, if you export Polyspace results to a CSV file, you can specify whether the generated result keys are calculated within the result function scope or the file scope. Along with the scope, the key is calculated by using the result names and types to identify each result. You use these keys to track results across analysis runs, for instance to remove duplicate results when you merge results from different modules that have common files.

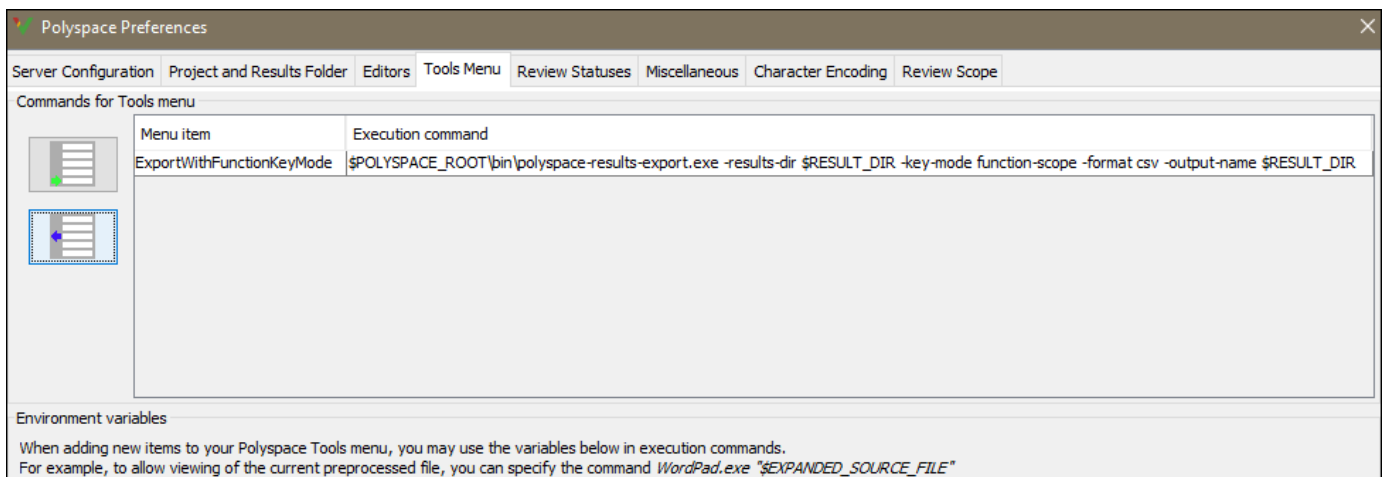
To specify how the keys are calculated, use option `-key-mode flag` at the command line with `polyspace-report-generator` or `polyspace-results-export`, where *flag* is one of these :

- `file-scope`(default) — The entries in the **Key** column are calculated by using the result name, the result type, and the file scope. This corresponds to the current behavior when you export analysis results.
- `function-scope` — The entries in the **Key** column are calculated by using the result name, the result type, and the function location. If the result is not inside a function, the key is calculated by using the file scope. When you enable this mode, the keys of results that are inside functions are prefixed with FN.

In the Polyspace user interface:

- To export results with `function-scope` mode, create a menu item by going to **Tools > Preferences** and entering this command on the **Tools Menu** tab:

```
$POLYSPACE_ROOT\bin\polyspace-results-export.exe -results-dir $RESULT_DIR -key-mode function-scope
-format csv -output-name $RESULT_DIR
```



You can then export results by using the menu item you created from **Tools > External Tools**.

- To export results with `file-scope` mode, use the **Reporting > Export** menu.

When you enable `function-scope` mode, a new result key is generated between analysis runs only if there are changes within the function scope of that result. You can use the more accurate `function-scope` keys, along with the entries of the **File** and **Function** column to track relevant changes.

Previously, edits within the file could trigger a recalculation of result keys, even if there were no changes within the function where the result was located.

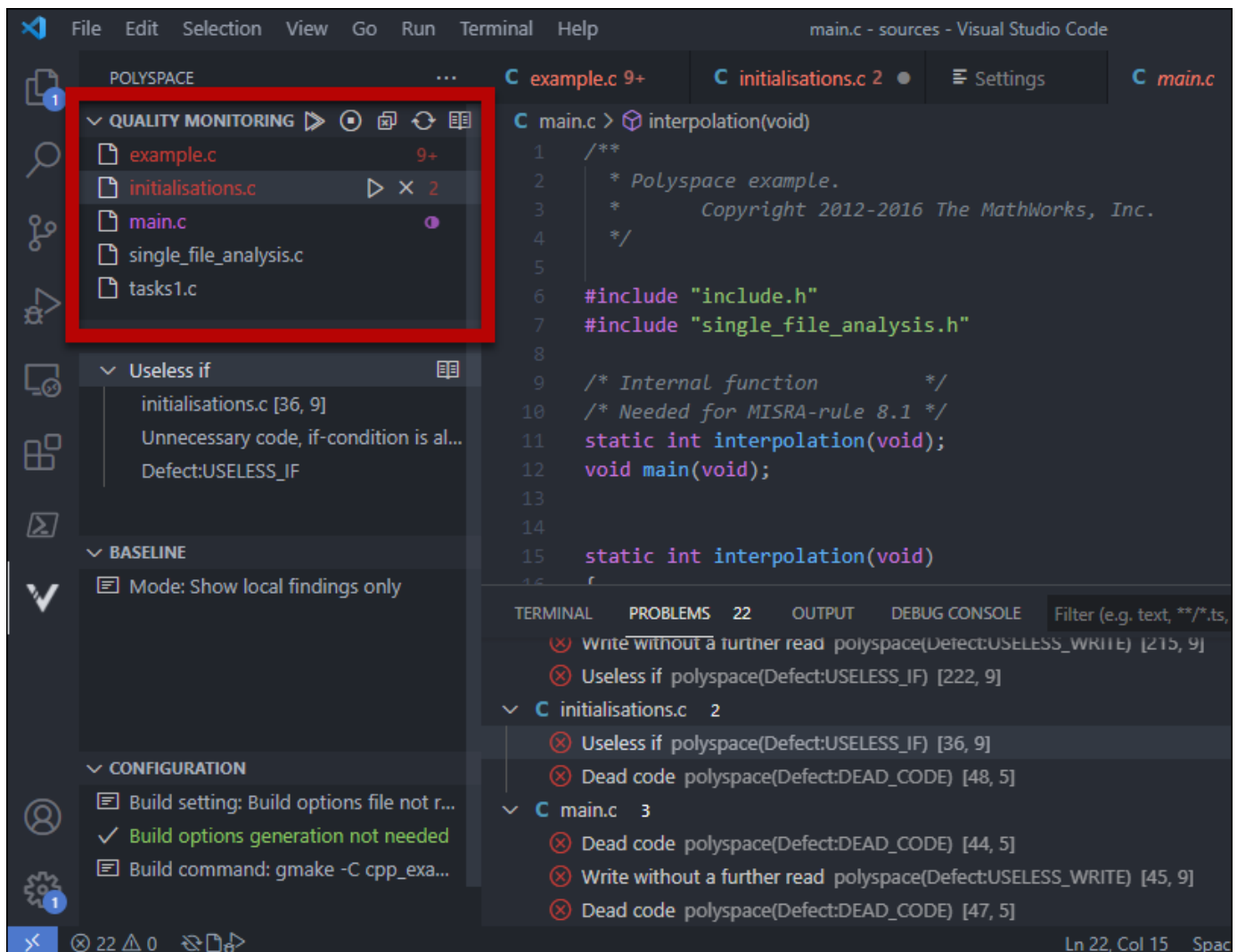
See also [Enable Function Scope for Exported Keys](#).

Product: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Access ..

Polyspace Extension for Visual Studio Code (VSCode): Track files to analyze and the status of analysis

In R2022a, you can track the files that you want to analyze more easily by adding those files to the **Quality Monitoring** view. This view shows all the files that you have selected for analysis, the status of the analysis for each file, and the number of findings. In this view, you can also start a file-by-file analysis of all the files that you added to the view or you can choose to analyze only a specific file.


Use the **Analysis Options: Add To Quality Monitoring On Save** setting to enable adding files to this view on save (Ctrl + S). You can also add files manually by right-clicking the file in the editor, the **EXPLORER** panel, or **SOURCE CONTROL** panel in the side bar.

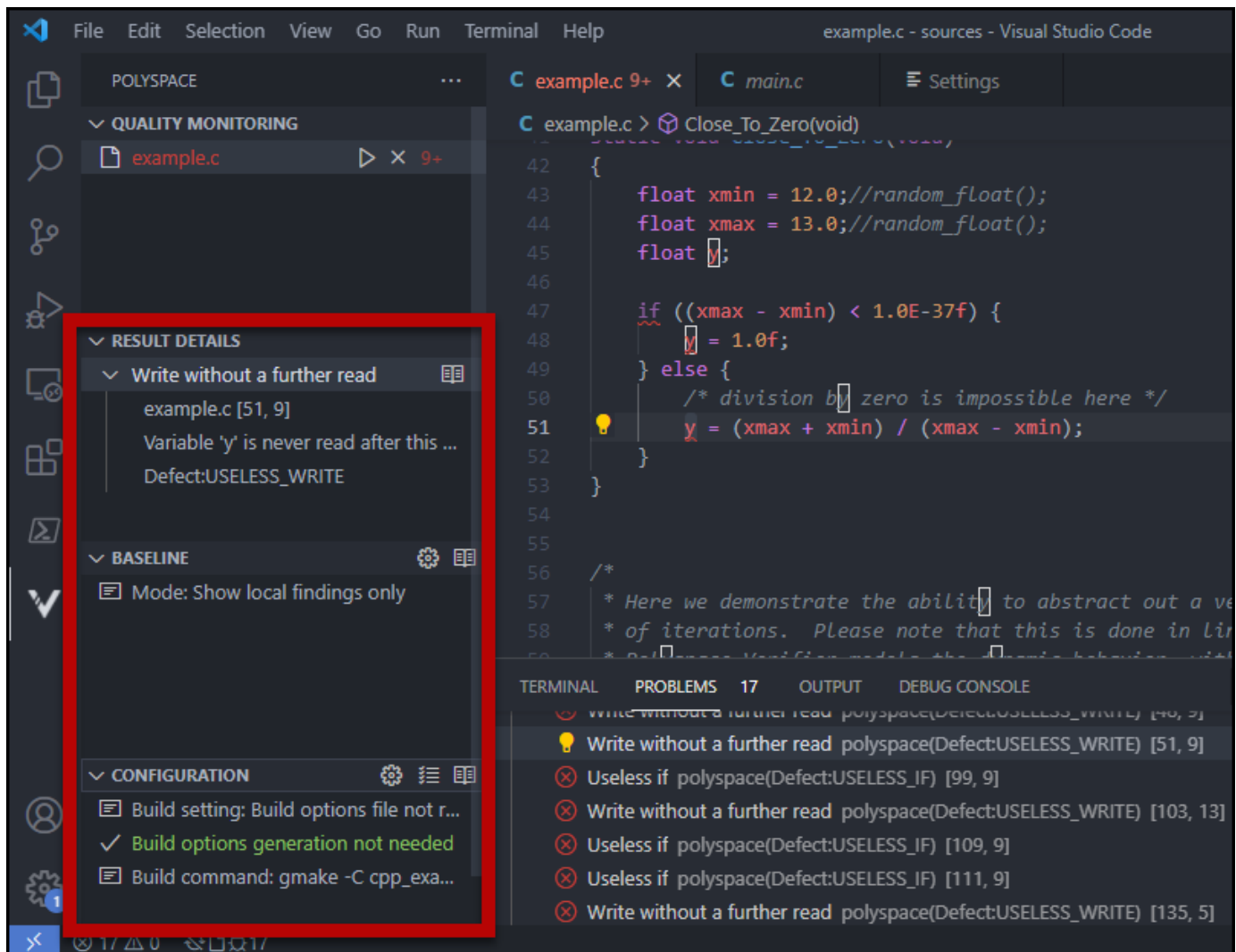


Product: Polyspace Access (Polyspace as You Code).





Polyspace Extension for Visual Studio Code (VSCode): New views for configuration, results details, baseline

In R2022a, you can use the new Polyspace views in Visual Studio Code to review your current build and baseline configurations and see additional details about findings. You can also open relevant settings or the documentation in a single click.

Display or hide these views by clicking  in the activity bar on the left.



The functionalities of the views are described in this table

View	Functionality
RESULT DETAILS	<p>When you select a finding in the OUTPUT panel or in the editor, this panel shows additional information about the finding, including the traceback when available. The OUTPUT panel shows only one line for each finding.</p> <p>Click  to open the contextual help for this finding.</p>
BASELINE	<p>View the current mode for the baseline (whether you are using the baseline or not) and, if you use the baseline, the status of the baseline download.</p> <p>Click  to open the baseline settings.</p>
CONFIGURATION	<p>This view shows your current Analysis Options settings and the status of the generated build options file, if applicable. Click  to open the Analysis Options settings and  to open the Checkers selection interface.</p>

Product: Polyspace Access (Polyspace as You Code).

Polyspace Extension for Visual Studio Code (VSCode): Mass-justify findings with a single click

In R2022a, you can mass-justify findings of the same defect or coding rule violations with a single click. If, for instance, the analysis finds a known or acceptable defect or coding rule violation, you can justify all instances of that finding in the currently active file.

To mass-justify findings, select a finding in the editor or the **Problems** panel, and then from the light bulb menu, click the **Justify all** option. Polyspace inserts an annotation in your code on each line that contains this finding and removes all instances of that finding from the **Problems** panels. Justified findings are suppressed in subsequent analyses.


```
46
47     if ((xmax - xmin) < 1.0E-37f) {
48         y = 1.0f;
49     } else {
50         /* division by zero is impossible here */
51         y = (xmax + xmin) / (xmax - xmin);
52     }
53 }
```

TERMINAL PROBLEMS 30 OUTPUT DEBUG CONSOLE Filter (e.g. text, **/

example.c 17

- ⊗ Dead code polyspace(Defect:DEAD_CODE) [47, 5]
- 💡 Write without a further read polyspace(Defect:USELESS_WRITE) [48, 9]
- ⊗ Write without a further read polyspace(Defect:USELESS_WRITE) [51, 9]
- ⊗ Useless if polyspace(Defect:USELESS_IF) [99, 9]
- ⊗ Write without a further read polyspace(Defect:USELESS_WRITE) [103, 1]
- ⊗ Useless if polyspace(Defect:USELESS_IF) [109, 9]
- ⊗ Useless if polyspace(Defect:USELESS_IF) [111, 9]
- ⊗ Write without a further read polyspace(Defect:USELESS_WRITE) [135, 5]
- ⊗ Dead code polyspace(Defect:DEAD_CODE) [149, 5]

Product: Polyspace Access (Polyspace as You Code).

Polyspace Access: Redesign of UI dashboard design for consistency and efficiency

The Polyspace Access Dashboard has been redesigned by modernizing and enhancing consistency and efficiency. The new global layout affects various cards, charts, graphs, numbers, and icons in

Polyspace Access.

The screenshot displays the Polyspace Access interface. On the left is a 'Project Explorer' with a list of projects, including 'Bug_Finder_Example-Trends (Bug Finder)'. The main area shows a 'Project Overview' dashboard for this project. It includes several summary cards: 'Open Results' (2,477 Open, 2,477 New, 0 Assigned To Me, 2,475 Unassigned), 'Polyspace' (26% progress, 2,278 Remaining, Exhaustive Threshold), 'Code Metrics' (0 Sub Projects, 3,315 Uncommented, 12 Files, 694 Cyclomatic), 'Defects' (77 Density, 254 Open, 254 New, 254 To Do, 0 In Progress, 158 Done), and 'Coding Standards' (671 Density, 2,223 Open, 2,223 New, 2,223 To Do, 0 In Progress, 807 Done). Below these is a 'Trends' line chart titled 'Open findings over time' showing data points from 12/23/2021 8:33:48 to 12/23/2021 9:44:40. At the bottom is a 'Details' table:

Name	Total	To Do	In Progress	Done
Defects	412	254	-	158
Coding Standards	3,030	2,223	-	807

Polyspace Access: Improved performance when viewing aggregate data from large project folders

Polyspace Access shows improved speed performance when opening different Polyspace Access dashboards. This speed improvement for loading dashboards is most notable when viewing aggregated project information for a folder that has many projects in it.

This table shows the improvements in loading times of the **Defects** dashboard for folders containing 500 and 2500 subprojects.

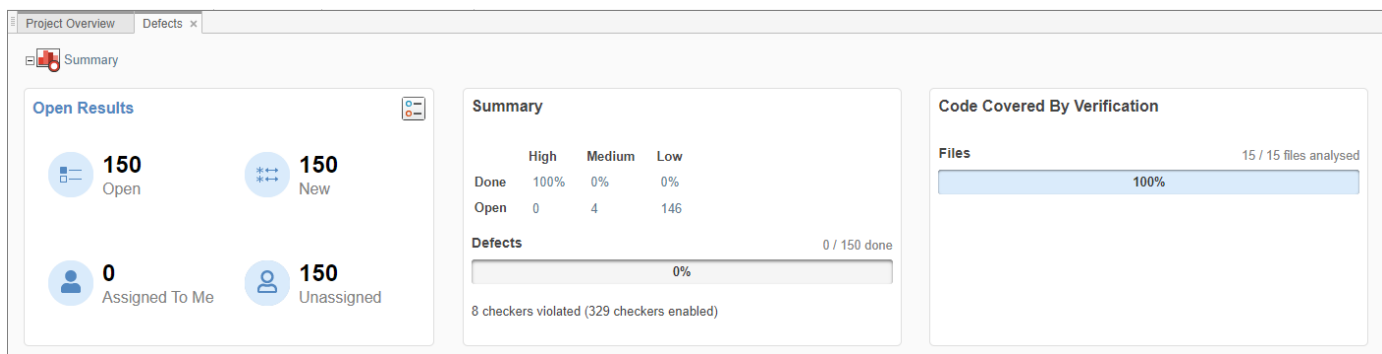
Folder Size	R2021b Loading Time	R2022a Loading Time
Folder containing 500 subprojects	45 seconds	1 second
Folder containing 2500 subprojects	50 seconds	3 seconds

Performance when loading dashboards was timed on a server using an Intel Xeon (Skylake IBRS) 4 core processor with 32GB of RAM and an SSD.

Polyspace Access: View code covered by verification in new graph

In R2022a, a **Code Covered By Verification** graph is now displayed in the **Run-time Checks**, **Defects**, and **Coding Standards** dashboards.

The **Code Covered By Verification** graph shows the number of files that were not analyzed due to situations such as compilation errors. For Code Prover run-time checks, the graph also shows the number of functions and code operations that were not checked because they were proven to be unreachable.



Polyspace Access Project Runs: Add labels to analysis runs that you upload to a project

In R2022a, you can add labels to a project run that you upload to Polyspace Access.

You can use labels to, for instance, identify project runs of interest more easily, or to associate a run with a specific branch or continuous integration build job.

To add or remove a label:



- At the command line, enter:

```
polyspace-access -add-label <LABEL> -run-id <RUN_ID> -host ...
polyspace-access -remove-label <LABEL> -run-id <RUN_ID> -host ...
```

Use the `-list-project` flag to obtain the run ID of the latest run, or the `-list-runs` flag to obtain the run ID of older runs. See `polyspace-access`.

For example, to add label `myLabel` to a project run with run ID 1234, enter:

```
polyspace-access -label myLabel -run-id 1234 -host ...
```

- In the Polyspace Access interface, select a run on the toolstrip drop-down list and click  in the **Project Details** pane. To remove a label, select the label from the **Labels** list and click .

The screenshot displays the Polyspace Access dashboard. At the top, there is a 'DASHBOARD' header and a navigation bar with several icons. Below this, a dropdown menu shows the current project as 'ID 13186 - Labels C...'. A list of projects is visible, with 'ID 13186 - Labels Custom Label, otherCustomLabel - Upload Date 6/13/21, 6:59 PM' selected. The main content area is divided into two columns. The left column, titled 'Project Details', shows information for the selected project: Name 'Bug_Finder_Example (Bug Finder)', Author 'MathWorks', Language 'C', Tools 'Bug Finder', Coding Standards 'Custom Rules, Guidelines', and Number of Runs '3'. Below this, the 'Run (ID 13186)' section shows the Upload Date '6/13/21, 6:59 PM' and a list of labels: 'Custom Label' and 'otherCustomLabel'. The right column, titled 'Summary', displays 'Open Results' with a count of '296 Open' and 'Assigned To Me' with a count of '0'. There are also icons for 'Open Results' and 'Assigned To Me'.

Product: Polyspace Access.

polyspace-access Command: Assign SQO levels, move or delete a project, and view list of runs for a project

In R2022a, you can manage Polyspace Access projects programmatically by using the `polyspace-access` command to perform the operations listed in this table.

Operation	Command
Move or rename a project or folder	<pre>polyspace-access -move-project <SOURCE> -to-project-path <DESTINATION> -host ...</pre> <p>For example, to move project <code>public/myProjects/foo</code> to folder <code>public/myOtherProjects</code>, run this command :</p> <pre>polyspace-access -move-project public/myProjects/foo -to-project-path public/myOtherProjects/foo -host ...</pre>
Delete a project or folder	<pre>polyspace-access -delete-project <PROJECT_PATH> -host ...</pre> <p>The command deletes the project from the Project Explorer but not from the Polyspace Access database. To delete a project from the database, see Delete Outdated Projects.</p>
Assign a Software Quality Objective (SQO) level to a project	<pre>polyspace-access -set-sqo <PROJECT_PATH> [-name <SQO_NAME>] -level <SQO_LEVEL> -host ...</pre> <p>The command assigns the specified <code><SQO_LEVEL></code> (1 through 6 or "exhaustive") for the current SQO definition. You can optionally use the <code>-name</code> flag to assign a different SQO definition. See also Quality Objectives Dashboard in Polyspace Access Web Interface.</p>
View the SQO definition and level currently assigned to a project.	<pre>polyspace-access -get-sqo <PROJECT_PATH> -host ...</pre> <p>The command returns the name of the SQO definition currently assigned to the project along with the assigned SQO level.</p>
View list of available SQO definitions	<pre>polyspace-access -list-sqo -host ...</pre> <p>The command returns a list of the names of all the available SQO definitions.</p>
View list of runs uploaded to a project	<pre>polyspace-access -list-runs <PROJECT_PATH> -host ...</pre> <p>The command returns a list of all the runs uploaded to the specified project. For each run, you see the run ID and any labels added to that run.</p>

You can use these commands as part of automation scripts when managing your projects. Previously, you performed the operations listed in the table only from the Polyspace Access user interface.

Product: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server.

polyspace-access Command: Improved robustness and error diagnostics

In R2022a, you can use the new `-max-retry` option with the `polyspace-access` command when you upload results from a client machine to a server machine that hosts the Polyspace Access database. The option specifies the number of times the client machine attempts to reconnect to the server machine in the event of a sporadic network failure, for instance `-max-retry 5`. The reconnection attempts happen at 10-second intervals. By default, the client machine attempts to reconnect three times.

The `polyspace-access` command also has improved error messages to help you diagnose issues more easily.

Product: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server.

Functionality Removed: Report generation from pre-R2015a results

Report generation from pre-R2015a Polyspace results are no longer supported. In releases R2015a and earlier, Polyspace products generated results in a format that will no longer be supported for report generation.

Compatibility Considerations

Typically, you do not require support for report generation from earlier releases since you presumably have archived reports generated using the earlier release. To generate reports from pre-R2015a releases using a newer release, first rerun the analysis using the newer release and regenerate the results in a supported format, and then generate reports. See also [Generate Reports from Polyspace Results](#).

Polyspace Access Installation

License Management: Use a single license to review Bug Finder, Code Prover, and Ada results in your web browser

In R2022a, Polyspace Bug Finder Access and Polyspace Code Prover™ Access are both renamed Polyspace Access. You require only one license to review Bug Finder, Code Prover, and Ada results in your web browser.

Previously, you required separate licenses to review Bug Finder results, and to review Code Prover or Ada results.

Product: Polyspace Access.

Compatibility Considerations

In the license manager options file, typically named `MLM.opt`, identify all users and groups to which you previously granted right-to-use privileges for only Polyspace Code Prover Access. For instance:

```
Define a group of users
GROUP ACCESS_CP_users user1 user2 user3

# Grant right-to-use privileges to individual users for Bug Finder
INCLUDE Polyspace_BF_Access USER user1
INCLUDE Polyspace_BF_Access USER user2

# Grant right-to-use privileges to individual users for Code Prover
INCLUDE Polyspace_CP_Access USER admin
# Grant right-to-use privileges to group of users for Code Prover
INCLUDE Polyspace_CP_Access GROUP ACCESS_CP_users
```

For all those users and groups, replace all instances of `Polyspace_CP_Access` with `Polyspace_BF_Access`.

```
Define a group of users
GROUP ACCESS_CP_users user1 user2 user3

# Grant right-to-use privileges to individual users
INCLUDE Polyspace_BF_Access USER user1
INCLUDE Polyspace_BF_Access USER user2

# Grant right-to-use privileges to individual users
INCLUDE Polyspace_BF_Access USER admin
# Grant right-to-use privileges to group of users
INCLUDE Polyspace_BF_Access GROUP ACCESS_CP_users
```

If a user or group previously had right-to-use privileges for Polyspace Bug Finder Access, this user or group now has right-to-use privileges to review Code Prover and Ada results.

See also [Configure Polyspace Access License](#).

Support for X.509 certificates generated without a SAN extension removed

In R2022a, Polyspace Access no longer supports certificates that were generated without a subject alternative name (SAN) extension. If the certificate uses only the common name (CN) field to identify the server host, Polyspace Access considers that certificate invalid. You cannot use this certificate to encrypt communications between the Polyspace Access server and client machines.

If you configure your bug tracking tool or LDAP server for HTTPS with certificates that were generated without a SAN extension, those certificates are also considered invalid.

Product: Polyspace Access.

Compatibility Considerations

To encrypt communications between the Polyspace Access server and client machines with HTTPS, generate x.509 certificates that use a SAN extension to specify the fully qualified domain name of the server hosting Polyspace Access. See [Choose Between HTTP and HTTPS Configuration for Polyspace Access](#).

To generate a valid SAN certificate for your bug tracking tool or LDAP server, contact your network administrator.

Polyspace Extension for Eclipse: Specify path to IDE executable when installing extension on Eclipse-based IDEs

In R2022a, when you install the Polyspace as You Code plugin for Eclipse-based IDEs such as Eclipse, Code Composer Studio, HighTech, or Windriver Workbench, you must specify the path to the IDE executable, for instance `C:\Program Files\Eclipse\eclipse.exe`.

Previously, you specified the path to the folder that contained that executable.

Product: Polyspace Access (Polyspace as You Code).

Compatibility Considerations

If you install the Eclipse plugin non-interactively by reusing the `installer.properties` file, edit the file to replace variable `ECLIPSE_INSTALL_FOLDER` with `ECLIPSE_EXECUTABLE_PATH` and specify the path to the executable, for instance:

```
ECLIPSE_EXECUTABLE_PATH=C:\\Program Files\\Eclipse\\eclipse.exe
```

See [Install Polyspace as You Code Noninteractively](#).

Changes in Polyspace Access docker containers

In R2022a, the following docker containers have been renamed:

Previous Container Name	Current Container Name
polyspace-access-web-server-main	polyspace-access-web-server-0-main

Previous Container Name	Current Container Name
polyspace-access-etl-main	polyspace-access-etl-0-main
polyspace-access-db-main	polyspace-access-db-0-main
issuetracker-server-main	issuetracker-server-0-main
issuetracker-ui-main	issuetracker-ui-0-main
usermanager-server-main	usermanager-server-0-main
usermanager-ui-main	usermanager-ui-0-main
usermanager-db-main	usermanager-db-0-main

Product: Polyspace Access.

Compatibility Considerations

In your scripts, replace instances of the previous names with the current names.

R2021b

Version: 3.5

New Features

Bug Fixes

Compatibility Considerations

Documentation

Documentation: View combined documentation for all Polyspace Bug Finder products

In R2021b, the Polyspace Bug Finder documentation covers all workflows for running a Polyspace Bug Finder analysis:

- *Desktop:*

The classic product Polyspace Bug Finder supports desktop workflows. You can run Bug Finder in the Polyspace user interface, by using scripts, or from platforms such as Simulink.

The Polyspace Bug Finder documentation continues to describe these workflows. See:

- Install Polyspace Products on Desktop
 - Set Up Bug Finder Analysis on Desktop
 - Review Polyspace Bug Finder Results in Polyspace User Interface
- *Server and Web Browser:*

The newer products, Polyspace Bug Finder Server and Polyspace Bug Finder Access, released in R2019a, support server-based workflows. Polyspace Bug Finder Server runs a Bug Finder analysis on continuous integration platforms such as Jenkins. Polyspace Bug Finder Access hosts the analysis results on a server so that several users can review them simultaneously on a web browser.

Previously, workflows involving Polyspace Bug Finder Server and Polyspace Bug Finder Access were documented separately. The Polyspace Bug Finder documentation now describes these workflows. See:

- Install Polyspace Products on Server
 - Set Up Bug Finder Analysis on Servers During Continuous Integration
 - Review Polyspace Bug Finder Results in Web Browser
- *Integrated Development Environments (IDEs):*

The latest feature, Polyspace as You Code, released in R2021a, runs a single-file analysis with Bug Finder on the file that is currently open in an IDE such as Eclipse, Visual Studio or Visual Studio Code. Polyspace as You Code is available with a Polyspace Bug Finder Access license.

Previously, workflows involving Polyspace as You Code were documented separately. The Polyspace Bug Finder documentation now describes these workflows. See:

- Install Polyspace Products in IDEs
- Set Up Polyspace Analysis in IDEs
- Review Polyspace as You Code Results in IDEs

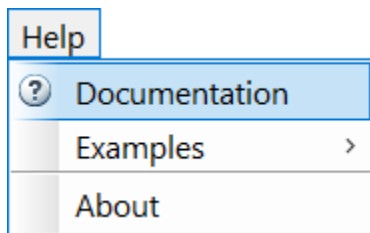
Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server , Polyspace Bug Finder Access.

Documentation: View web documentation by default

In R2021b, if you open the Polyspace documentation from the product while connected to the internet, the web documentation opens in your default web browser. If you open the documentation without an internet connection, or if your internet connection becomes unavailable, the same actions point to the PDF documentation shipped with the products.

This change applies to all methods of opening the documentation:

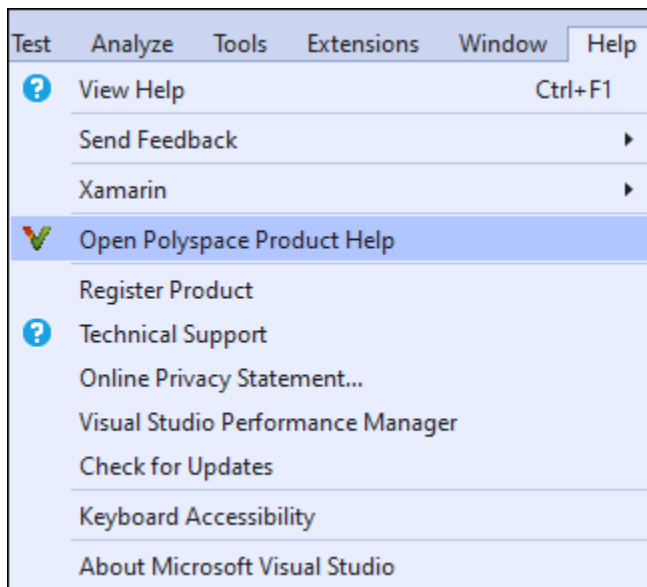
- From the Polyspace user interface, choosing **Help > Documentation**.



- Using the `-doc` option of commands such as `polyspace-bug-finder`.

See `-doc` | `-documentation`.

- From IDEs, using Polyspace as You Code extensions.



Depending on where you open the documentation from, you see documentation pages appropriate to the platform.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server , Polyspace Bug Finder Access.

Contextual Help: View contextual help in web browser

In R2021b, all contextual help in the Polyspace products open in your default web browser. Contextual help includes help for options that can be accessed from the tooltip on the option:

Target Language

Source code language

C standard version

C standard version (-c-version)
Select the version of the C standard to use.

[? More Help](#)

and help for results that can be accessed from the result details:

○ ID 83: Unsafe conversion between pointer and integer (Impact: Medium) [?](#) [🔗](#)

Conversion from constant 0xdeadbeef to unsigned int * is misaligned.
The expected alignment is 4 bytes.
The resulting pointer may point to an invalid object.

[More Help](#)

	Event	File	Scope	Line
1	○ Unsafe conversion between pointer and integer	programming.c	bug_badintptrcast()	619

You can now use all facilities of your web browser to interact with a contextual help page. Note that the contextual help buttons open pages available with your installation. You do not require an internet connection to view these pages.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Access

Analysis Setup

IAR Embedded Workbench Compiler: Set up Polyspace analysis for code compiled by using RISC-V target

In R2021b, Polyspace supports the IAR Embedded Workbench RISC-V target natively. If you build your source code by using the RISC-V target, you can specify the target name for your Polyspace analysis.

Target Environment	
Compiler	iar-ew
Target processor type	riscv

For more information, see [IAR Embedded Workbench Compiler \(-compiler iar-ew\)](#).

At the command line, you specify a compiler target by using the option `Target processor type (-target)`. For instance:

```
polyspace-bug-finder-server -sources file.c -compiler iar-ew -target riscv...
```

Because of the native support, you can now set up a Polyspace project without knowing the internal workings of this compiler target. The analysis can interpret macros that are implicitly defined by the target and compiler-specific language extensions, such as keywords and pragmas.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Updated GCC Compiler Version Support: Set up Polyspace analysis for code compiled with GCC versions 9.x and 10.x

In R2021b, Polyspace supports GCC compiler versions 9.x and 10.x. If you build your source code by using the GCC compiler versions 9.x or 10.x., you can specify the compiler name for your Polyspace analysis.

Target Environment	
Compiler	gnu9.x
Target processor type	x86_64

Target Environment	
Compiler	gnu10.x
Target processor type	x86_64

At the command line, you specify a compiler by using the option `-compiler`. For instance:

```
polyspace-bug-finder-server -sources file.c -compiler gnu9.x ...
```

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

Because of the native support, you can now set up a Polyspace project without knowing the internal workings of these compilers. The analysis can interpret macros that are implicitly defined by the compilers and compiler-specific language extensions, such as keywords and pragmas.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

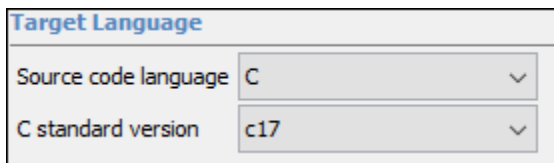
Simulink Support: Consistent C++ version in Polyspace and Simulink

In R2021b, the C++ version used in the Polyspace analysis and the C++ version specified in the Simulink **Configuration Parameter** window are consistent. Polyspace extracts the C++ version specified in Simulink **Configuration Parameter** window and uses the version number in the analysis. Previously, unless you explicitly set the C++ version for Polyspace, the analysis used the default C++03, regardless of the C++ version specified in Simulink. Starting in R2021b, the C++ version specified in Simulink is also used in the Polyspace analysis.

Product: Polyspace Bug Finder (Desktop).

C17 Support: Run Polyspace analysis on code that follows version C17 of C standard

In R2021b, Polyspace supports version C17 of the C Standard (ISO/IEC 9899:2018). This version of the standard addresses issues in the previous version, C11, but it does not introduce new language features.



See also `C standard version (-c-version)`.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Configuration from Build System: Copy console output to log file

In R2021b, you can copy the output of the `polyspace-configure` command to a log file. Use option `-log` to specify the log file path, for example:

```
polyspace-configure -verbose -log pscfg.log make -B
```

You can review the contents of the log file to investigate possible issues with the execution of the `polyspace-configure` command, especially if you run the command as part of an automation script. Previously, to store the console output, you had to redirect the Standard Out (stdout) and the Standard Error (stderr) to a file.

See also `polyspace-configure`.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Polyspace as You Code Configuration: Use script in Polyspace extension without specifying path of installation folder

In R2021b, if you use a script in a Polyspace extension for Visual Studio®, Visual Studio Code, or Eclipse to run a Polyspace as You Code analysis and view results, the extension passes these paths as arguments to the script:

- Path of the currently analyzed file as a first argument.
- Path of working directory where the extension stores analysis results as a second argument.
- Path of the installation folder for Polyspace as You Code as a third argument.

Because you do not need to hard-code the paths of the installation folder, source file, or results folder, you can more easily share the script with team members who are working on different machines.

For example, this simple batch script uses the arguments passed by the extension to run an analysis on the currently active file by using the default Polyspace build options and stores the results in the results folder specified in the extension setting.

```
set INSTALL_DIR=%3
set ANALYZE=%INSTALL_DIR%\polyspace\bin\polyspace-bug-finder-access.exe
set SOURCES=%1
set RESULTS_FOLDER=%2
```

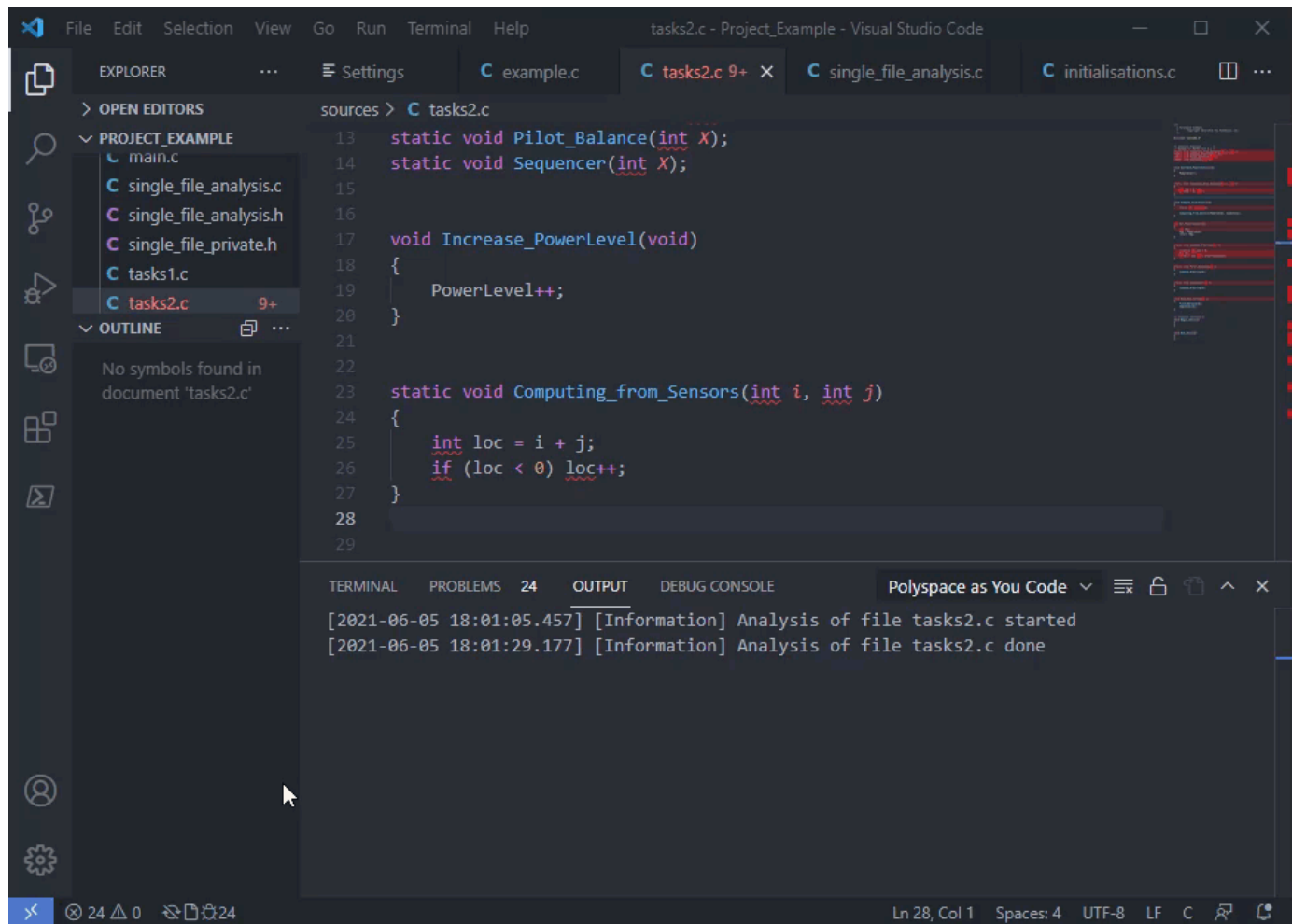
```
"%ANALYZE%" -sources %SOURCES% -results-dir %RESULTS_FOLDER%
```

Typically, you use a script to perform advanced workflows from your IDE. For instance, if analyze files from components that have different build configurations or you use a custom tool to setup your build environment.

Previously, the Polyspace extension did not pass the installation folder to the script. You had to edit the script to use it on a different machine.

Polyspace Extension for Visual Studio Code (VSCode): View information about the extension in the status bar

In R2021b, Polyspace as You Code displays icons in the Visual Studio Code status bar to provide information about the state of the extension, such as errors in the configuration or whether the analysis uses a baseline.

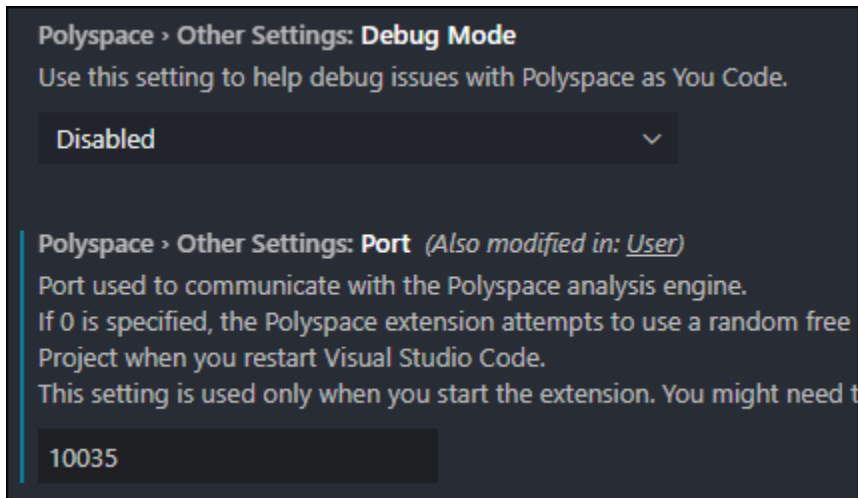


- Place your cursor over an icon to view a tooltip containing additional information.
- When applicable, click an icon to perform common operations, such as opening the extension settings or viewing the **OUTPUT** pane.

Product: Polyspace Bug Finder Access (Polyspace as You Code).

Polyspace Extension for Visual Studio Code (VSCode): Specify communication port and enable debugging

In R2021b, the Polyspace extension for Visual Studio Code has these new settings:



- **Debug Mode** — Enable this setting to display all debugging information in the **OUTPUT** pane and to help you troubleshoot issues that occur when you use the extension. By default, this setting is disabled and the **OUTPUT** pane displays only errors, warnings, and information messages, such as the start or end of an analysis.
- **Port** — Specify the port that the extension uses to communicate with the Polyspace as You Code analysis engine. Use this setting if, for instance, your machine is configured with a firewall and you want to specify an open port in the firewall.

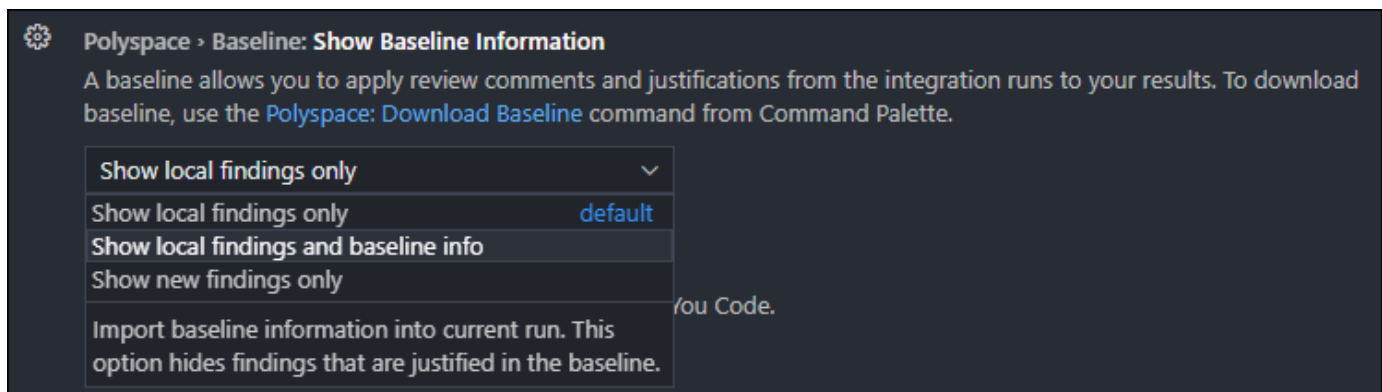
If the setting is set to the default value (0), Polyspace queries your system for an available port and uses whichever port your system returns.

See also Other Settings.

Product: Polyspace Bug Finder Access (Polyspace as You Code).

Polyspace Extension for Visual Studio Code (VSCode): Configure use of baseline information in fewer steps

In R2021b, the settings that you use to specify whether the analysis uses information from a baseline run are consolidated into a single drop-down list for a simpler configuration.



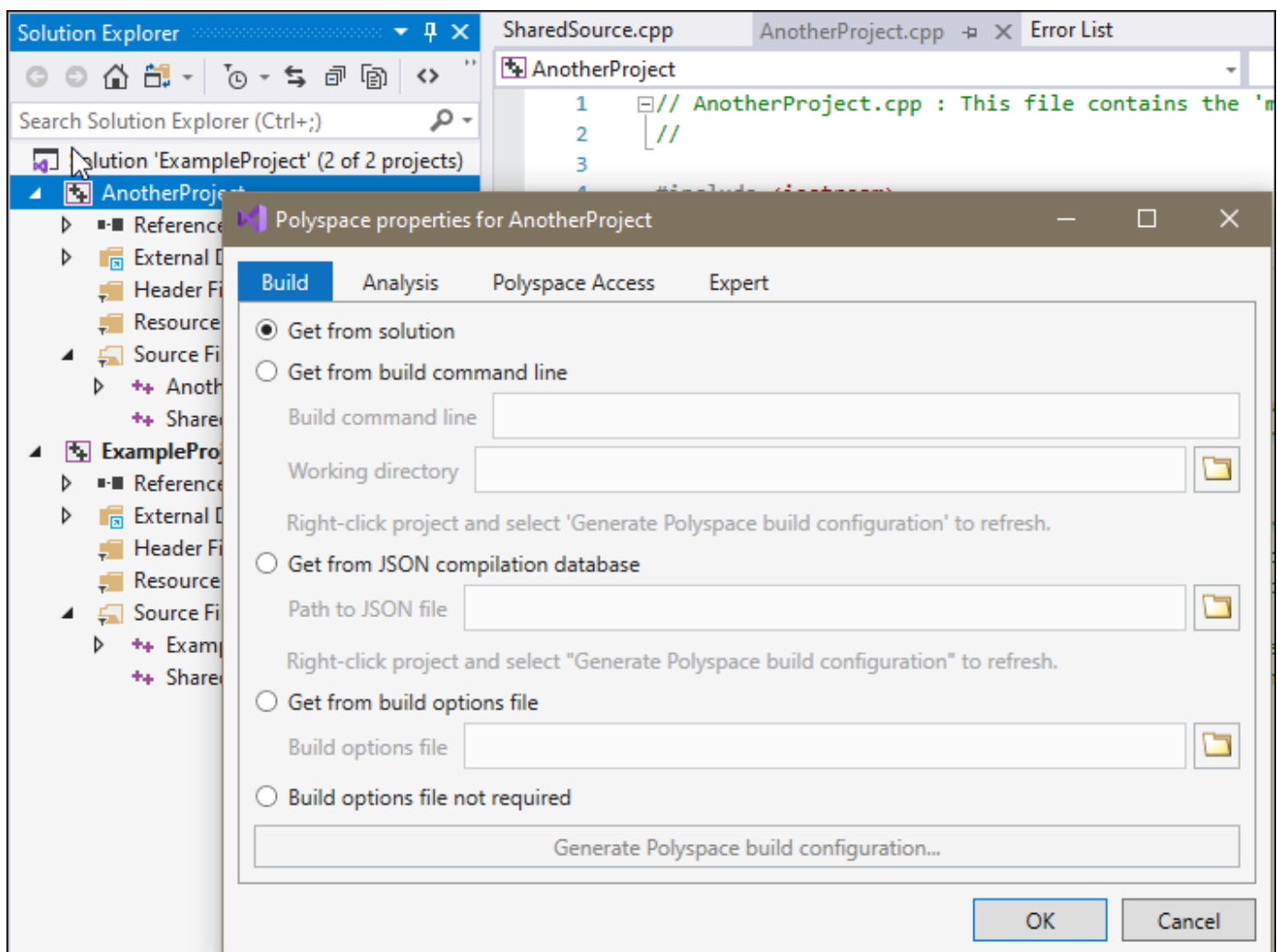
Previously, you used separate settings to specify whether the analysis used the baseline information and whether to show only new findings.

See also [Baseline Polyspace as You Code Results in Visual Studio Code](#).

Product: Polyspace Bug Finder Access (Polyspace as You Code).

Polyspace Extension for Visual Studio: Open project configuration by using fewer clicks

In R2021b, you can right-click a project in the Visual Studio **Solution Explorer** and select **Polyspace properties** to open the Polyspace extension settings for the selected project.



See [Configure Polyspace as You Code Extension in Visual Studio](#).

Previously, you configured the project settings by going to the Polyspace node in the **Tools > Options** menu. You still use this menu to configure the global settings for the Polyspace extension.

Product: Polyspace Bug Finder Access (Polyspace as You Code).

Polyspace Extension for Visual Studio: Extract build options more quickly from Visual Studio solution

In R2021b, if you use Polyspace extension option **Build from solution** and your project configuration type is *Application (.exe)*, *Dynamic Library (.dll)*, or *Static Library (.lib)*, Polyspace extracts your build options directly from the project without building your solution and tracing the build.

If you make changes to your project, Polyspace updates the build options when you start the next analysis. You no longer need to manually update the Polyspace build options when you make changes to your project.

Previously, to extract or update the build options used by the analysis, Polyspace built your solution and traced the build. Starting in R2021b, Polyspace skips the build step and you can analyze your files in less time.

See [Generate Build Options for Polyspace as You Code Analysis in Visual Studio](#).

For other Visual Studio project types, Polyspace still builds your solution and traces your build to extract the build options.

Product: Polyspace Bug Finder Access (Polyspace as You Code).

Functionality Being Removed: Polyspace stubs for Standard Template Library

Polyspace stubs for the C++ Standard Template Library (STL) will be removed in a future release. These stubs conform to the older C++98 Standard and are meant for quickly getting started with a C++ analysis. In most situations, your compiler implementation of the Standard Template Library is required for successful compilation of a C++ project with Polyspace.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Compatibility Considerations

If your project uses STL stubs, you see a warning. Use the option `No STL stubs (-no-stl-stubs)` to prevent use of the stubs, and then provide your compiler implementation of the Standard Template Library for analysis.

Functionality Being Removed: Compilation assistant

The Polyspace compilation assistant will be removed in a future release. You get a warning when you enable this option and run an analysis.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Compatibility Considerations

If you use the compilation assistant in your Polyspace project, clear the corresponding option. To clear this option in the desktop interface, go to **Tools > Preferences**, and then select the **Project and Results Folder** tab.

Alternatively, when you set up your Polyspace project:

- Use the **Compiler** (`-compiler`) option to specify a compiler that Polyspace supports natively if you compile your code by using that compiler.
- Use `polyspace-configure` to trace your build command and to obtain your compiler configuration. See `polyspace-configure`.

Changes in analysis options and binaries

Some arguments for `-misra3`, `-misra-cpp`, and `-checkers` are no longer supported

Errors

Starting in R2021b, some arguments for the preceding analysis options are no longer supported.

- `-misra3`: For this option, the argument values `CERT-all`, `CERT-rules`, and `ISO-17961` are no longer supported. To enable the CERT C or ISO-17961 coding standards, use options such as `-cert-c` and `-iso-17961`.
- `-misra-cpp`: For this option, the argument values `CERT-all` and `CERT-rules` are no longer supported. To enable the CERT C++ coding rules, use `-cert-cpp` instead.
- `-checkers`: For this option, the argument values `CERT-all`, `CERT-rules`, and `ISO-17961` are no longer supported. To enable the CERT C or ISO-17961 coding standards, use options such as `-cert-c` and `-iso-17961`.

See also:

- Check SEI CERT-C (`-cert-c`)
- Check SEI CERT-C++ (`-cert-cpp`)
- Check ISO/IEC TS 17961 (`-iso-17961`)

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

The analysis options `-misra2`, `-misra3`, `-misra-ac-agc`, `-misra-cpp`, and `-jsf-coding-rules` no longer accept text files as valid argument

Errors

Starting in R2021b, the preceding analysis options no longer support a custom selection of coding rules through text files. In previous releases, when you used text files with the preceding options to specify custom selection, the Polyspace desktop interface produced a warning in the log. From R2021b, the analysis stops with an error when you use a text file as an argument for these options. To enable an custom selection of coding rules, use the option `Set checkers by file` (`-checkers-selection-file`) and specify an XML file as an argument. See:

- Check for Coding Standard Violations
- Setting Checkers in Polyspace as You Code

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Analysis Results

AUTOSAR C++14 Support: Check for 345 AUTOSAR C++14 rules including 18 new rules

In R2021b, check for these additional AUTOSAR C++14 rules.

AUTOSAR Rule	Description	Polyspace Checker		
A4-7-1	An integer expression shall not lead to data loss.	AUTOSAR A4-7-1	C++14	Rule
A8-4-3	Common ways of passing parameters should be used.	AUTOSAR A8-4-3	C++14	Rule
A10-4-1	Hierarchies should be based on interface classes.	AUTOSAR A10-4-1	C++14	Rule
A12-1-3	If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead.	AUTOSAR A12-1-3	C++14	Rule
A12-1-5	Common class initialization for non-constant members shall be done by a delegating constructor.	AUTOSAR A12-1-5	C++14	Rule
A12-7-1	If the behavior of a user-defined special member function is identical to implicitly defined special member function, then it shall be defined "=default" or be left undefined.	AUTOSAR A12-7-1	C++14	Rule
A13-2-2	A binary arithmetic operator and a bitwise operator shall return a "prvalue".	AUTOSAR A13-2-2	C++14	Rule
A13-3-1	A function that contains "forwarding reference" as its argument shall not be overloaded.	AUTOSAR A13-3-1	C++14	Rule
A14-1-1	A template should check if a specific template argument is suitable for this template	AUTOSAR A14-1-1	C++14	Rule
A14-5-1	A template constructor shall not participate in overload resolution for a single argument of the enclosing class type.	AUTOSAR A14-5-1	C++14	Rule
A14-7-1	A type used as a template argument shall provide all members that are used by the template.	AUTOSAR A14-7-1	C++14	Rule

AUTOSAR Rule	Description	Polyspace Checker		
A15-1-4	If a function exits with an exception, then before a throw, the function shall place all objects/resources that the function constructed in valid states or it shall delete them.	AUTOSAR A15-1-4	C++14	Rule
A15-2-2	If a constructor is not noexcept and the constructor cannot finish object initialization, then it shall deallocate the object's resources and it shall throw an exception.	AUTOSAR A15-2-2	C++14	Rule
A15-4-5	Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overrides.	AUTOSAR A15-4-5	C++14	Rule
A15-5-2	Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of <code>std::abort()</code> , <code>std::quick_exit()</code> , <code>std::_Exit()</code> , <code>std::terminate()</code> shall not be done.	AUTOSAR A15-5-2	C++14	Rule
A18-5-5	Memory management functions shall ensure the following: (a) deterministic behavior resulting with the existence of worst-case execution time, (b) avoiding memory fragmentation, (c) avoid running out of memory, (d) avoiding mismatched allocations or deallocations, (e) no dependence on non-deterministic calls to kernel.	AUTOSAR A18-5-5	C++14	Rule
A18-5-8	Objects that do not outlive a function shall have automatic storage duration.	AUTOSAR A18-5-8	C++14	Rule
A27-0-1	Inputs from independent components shall be validated.	AUTOSAR A27-0-1	C++14	Rule

See also AUTOSAR C++14 Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Bug Finder Checkers: Check for inefficient C++ algorithms or function usage and other issues

In R2021b, using new Bug Finder checkers, you can check for these additional types of issues.

Defect	Description
Expensive use of a standard algorithm when a more efficient method exists	Functions from the algorithm library are misused with inappropriate inputs, resulting in inefficient code
Expensive use of container's count method	A container's count function is used for checking if a key is present in the container, leading to inefficient code
Unnecessary Padding	Members of a struct are padded to fulfill alignment requirement when rearranging the members to fulfill the alignment requirement saves memory
Inefficient use of sprintf	The function sprintf copies strings instead of the more efficient strcpy
C-string from string::c_str() compared to pointer	The C string obtained from std::string::c_str() is compared to a pointer (or NULL)
Expensive post-increment operation	An object is post-incremented when pre-incrementing is faster
Expensive dynamic cast	Expensive dynamic_cast is used instead of the more efficient static_cast or const_cast
Move operation uses copy	A move constructor or move assignment operator uses copy operations on base classes or data members

For the full list of checkers, see Defects.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

CERT C++ Support: Check for violations associated with exception handling

In R2021b, you can look for violations of this CERT C++ rule in addition to previously supported rules.

CERT C++ Rule	Description	Polyspace Checker
ERR57-CPP	Do not leak resources when handling exceptions	CERT C++: ERR57-CPP

See also CERT C++ Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Custom Rules: Check typedef statements that creates aliases for existing typedef aliases

Starting in R2021b, you can check compliance with naming rule conventions in typedef statements that define a new alias for an existing typedef alias.

This change affects the custom rule 3.1-3.5, 4.2, 5.2, and 6.2. For instance, consider these typedef statements:

```
// pattern for custom rule 3.1: .*_i
typedef short int int8_i; //Compliant
typedef int8_i short_signed_int_typedef; //Noncompliant

// pattern for custom rule 3.2: .*_f
typedef float float32_f; //Compliant
typedef float32_f sLong; //Noncompliant

// pattern for custom rule 3.3: .*_p
typedef int* int_p; //Compliant
typedef int_p p_int; // Noncompliant

// pattern for custom rule 3.4: .*_arr
typedef int int3_arr[3]; //Compliant
typedef int3_arr in3af[3]; //Noncompliant

// pattern for custom rule 3.5: .*_fp
typedef void (*func_fp) (int); //Compliant
typedef func_fp func; //Noncompliant

// pattern for custom rule 4.2: .*_ST
typedef struct my_ST {
    int x;
} my_ST; //Compliant
typedef my_ST int_struct; //Noncompliant

// pattern for custom rule 5.2: .*_cl
class A{};
typedef A a_cl; // Compliant
typedef a_cl bc; //Noncompliant

// pattern for custom rule 6.2: .*_en
typedef enum state {DEAD,ALIVE} State_en; //Compliant
typedef State_en se; //Noncompliant
```

The naming pattern for various types, classes, structures, and enumerators are specified. Polyspace flags the typedef statements that define noncompliant aliases. For instance, the typedef statement for the alias sLong is flagged because it does not end in _f. The identifier sLong is an alias for float32_f, which in turn is an alias for float.

Previously, Polyspace checked the typedef statements that defined aliases for fundamental types. Starting in R2021b, Polyspace also checks the typedef statements that create an alias for a previously defined typedef alias.

See Custom Coding Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Compatibility Considerations

If you checked naming convention of typedef aliases in your code, you might see a change in the number of violations.

Changes to coding standards checking

In R2021b, these changes have been made in the checking of previously supported rules.

AUTOSAR C++14

Rule	Description	Change
AUTOSAR C++14 Rule A5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.	The checker shows more accurate results. For instance: <ul style="list-style-type: none"> The checker reports only one violation when a variable is written and reused within the same expression. Previously, two violations were reported. The checker detects violations involving global variables, for instance, when the same global variable is written and reused in an expression.
AUTOSAR C++14 Rule A7-1-1	Constexpr or const specifiers shall be used for immutable data declaration.	When variables of reference types T& are modified, the checker no longer flags them (or the variables referenced to) as candidates for const-qualification.
AUTOSAR C++14 Rule A8-4-7	"in" parameters for "cheap to copy" types shall be passed by value.	The checker now flags passed-by-value "in" parameters that are: <ul style="list-style-type: none"> Not trivially copyable, for instance <pre>struct B { B(B const&) {} }; void func2(const B b); //Noncompliant</pre> Expensive to copy, for instance <pre>void func(std::string s){ //Noncompliant //... }</pre> In addition, for passed-by-reference parameters, the checker: <ul style="list-style-type: none"> Treats non-const parameters as "in" parameters if they are not modified inside the function . Does not flag move-only type "in" parameters that are passed by reference.
AUTOSAR C++14 Rule A10-3-3	Virtual functions shall not be introduced in a final class.	The checker no longer flags functions specified as <code>virtual final</code> or <code>override final</code> .

Rule	Description	Change
AUTOSAR C++14 Rule A15-5-3	The <code>std::terminate()</code> function shall not be called implicitly.	The checker now flags an <code>atexit</code> handler that raises an unhandled exception. A termination handler that is passed to <code>std::atexit</code> raises an unhandled exception. For instance: <pre>#include <stdexcept> void atexit_handler(){//Noncompliant throw std::runtime_error("Error in atexit function"); } void main(){//Noncompliant try{ //... std::atexit(atexit_handler); }catch(std::exception& e){ //... } }</pre>
AUTOSAR C++14 Rule M0-3-2	If a function generates error information, then that error information shall be tested.	The checker is now raised on sensitive functions that return pointers if the return value is not checked. For instance: <pre>FILE *in; in = fopen (argv[1], "r");</pre> Polyspace now raises a defect if <code>in</code> is not tested before it is used for accessing a file.
AUTOSAR C++14 Rule M11-0-1	Member data in non-POD class types shall be private.	The checker flags nonprivate data members in classes that are not POD types.
AUTOSAR C++14 Rule M16-0-1	<code>#include</code> directives should only be preceded by preprocessor directives or comments.	The checker ignores code that is hidden by using conditional compilation directives such as <code>#if</code> or <code>#ifdef</code> .

CERT C

Rule	Description	Change
CERT C: Rec. DCL00-C	Const-qualify immutable objects	When variables of reference types T& are modified, the checker no longer flags them (or the variables referenced to) as candidates for const-qualification.
CERT C: Rec. EXP10-C	Do not depend on the order of evaluation of subexpressions or the order in which side effects take place	The checker shows more accurate results. For instance: <ul style="list-style-type: none"> The checker reports only one violation when a variable is written and reused within the same expression. Previously, two violations were reported. The checker detects violations involving global variables, for instance, when the same global variable is written and reused in an expression.
CERT C: Rule ARR39-C	Do not add or subtract a scaled integer to a pointer	The checker now checks for incorrect pointer scaling that occurs in the temporary objects used in function calls. For instance, Polyspace now detects this incorrect use of <code>sizeof</code> : <pre>#include <stddef.h> #include <stdlib.h> #include <wchar.h> enum { WCHAR_BUF = 128 }; void func2_ko (void) { wchar_t error_msg[WCHAR_BUF]; wcsncpy (error_msg, L"Error: "); fgetws (error_msg + wcslen (error_msg) * sizeof (wchar_t), WCHAR_BUF - 7, stdin); //Noncompliant }</pre>
CERT C: Rule ERR33-C	Detect and handle standard library errors	The checker is now raised when: <ul style="list-style-type: none"> A sensitive function returns a pointer that is not validated before use. For instance: <pre>FILE *in; in = fopen (argv[1], "r");//Noncompliant</pre> Polyspace now raises a defect if <code>in</code> is not validated before it is used for accessing a file. A pointer is overwritten during memory reallocation. For instance: <pre>p = realloc(p,SIZE);</pre> Reallocation statements that overwrite the original pointer by the return value of <code>realloc()</code> might cause memory leaks and data loss if the <code>realloc()</code> function fails.
CERT C: Rule EXP30-C	Do not depend on the order of evaluation for side effects	The checker shows more accurate results. For instance: <ul style="list-style-type: none"> The checker reports only one violation when a variable is written and reused within the same expression. Previously, two violations were reported. The checker detects violations involving global variables, for instance, when the same global variable is written and reused in an expression.

Rule	Description	Change
CERT C: Rule EXP37-C	Call functions with the correct number and type of arguments	The checker is now raised when an external function is called with an argument that is not compatible with the prototype after integral promotion. For instance: <pre>extern long foo(int); long bar(long i) { return foo(i); //Noncompliant: calls foo(int) with a long }</pre>
CERT C: Rule EXP39-C	Do not access a variable through a pointer of an incompatible type.	The checker now detects situations where you access memory that is reallocated to an object of a different type without reinitializing the contents first. For instance, in the example, the checker raises a flag if you read the memory that <code>Bptr</code> points to, without reinitializing the memory first. <pre>struct A; struct B; struct A *Aptr = (struct A*) malloc(sizeof(struct A)); struct B *Bptr = (struct B*) realloc(Aptr, sizeof(struct B));</pre>
CERT C: Rule EXP43-C	Avoid undefined behavior when using restrict-qualified pointers	The checker now checks for Overlapping access by restrict-qualified pointers . This issue occurs when any of these conditions is true: <ul style="list-style-type: none"> • Two restrict-qualified pointers modify objects that have overlapping memory addresses. • In a function, a <code>const</code> restrict-qualified pointer might modify a <code>nonconst</code> restrict-qualified pointer but when calling the function, both of these arguments are the same or are derived from the same pointer. • A standard library function is called by using restrict-qualified pointers that overlap. • A restrict-qualified pointer is assigned to another restrict-qualified pointer within the same scope.
CERT C: Rule FI040-C	Reset strings on <code>fgets()</code> or <code>fgetws()</code> failure	The checker now flags direct read operations on buffers written by <code>fgets</code> -family functions if the function calls might have failed and the buffers have possibly indeterminate contents. If the buffer written is <code>buf</code> , direct read operations include operations such as <code>buf[0]</code> or <code>*buf</code> .
CERT C: Rule FLP34-C	Ensure that floating-point conversions are within range of the new type	The checker now detects overflows in conversions from floating point to integer types. In addition, if the same variable causes multiple conversion overflows, all overflows are flagged. Previously, only the first overflow in a block was flagged.

Rule	Description	Change
CERT C: Rule POS54-C	Detect and handle POSIX library errors	The checker is now raised on sensitive functions that return pointers if the return value is not checked. For instance: <pre>FILE *in; in = fmemopen (argv[1], strlen (argv[1]), "r")</pre> Polyspace now raises a defect if <code>in</code> is not tested before it is used for accessing a file.

CERT C++

Rule	Description	Change
CERT C++: ARR39-C	Do not add or subtract a scaled integer to a pointer	<p>The checker now checks for incorrect pointer scaling that occurs in the temporary objects used in function calls. For instance, Polyspace now detects this incorrect use of <code>sizeof</code>:</p> <pre>#include <stddef.h> #include <stdlib.h> #include <wchar.h> enum { WCHAR_BUF = 128 }; void func2_ko (void) { wchar_t error_msg[WCHAR_BUF]; wcsncpy (error_msg, L"Error: "); fgetws (error_msg + wcslen (error_msg) * sizeof (wchar_t), WCHAR_BUF - 7, stdin); //Noncompliant }</pre>
CERT C++: ERR33-C	Detect and handle standard library errors	<p>The checker is now raised when:</p> <ul style="list-style-type: none"> A sensitive function returns a pointer that is not validated before use. For instance: <pre>FILE *in; in = fopen (argv[1], "r");//Noncompliant</pre> <p>Polyspace now raises a defect if <code>in</code> is not validated before it is used for accessing a file.</p> A pointer is overwritten during memory reallocation. For instance: <pre>p = realloc(p,SIZE);</pre> <p>Reallocation statements that overwrite the original pointer by the return value of <code>realloc()</code> might cause memory leaks and data loss if the <code>realloc()</code> function fails.</p>
CERT C++: ERR50-CPP	Do not abruptly terminate the program	<p>The checker now flags an <code>atexit</code> handler that raises an unhandled exception. A termination handler that is passed to <code>std::atexit</code> raises an unhandled exception. For instance:</p> <pre>#include <stdexcept> void atexit_handler(){//Noncompliant throw std::runtime_error("Error in atexit function"); } void main(){//Noncompliant try{ //... std::atexit(atexit_handler); }catch(std::exception& e){ //... } }</pre>
CERT C++: EXP37-C	Call functions with the correct number and type of arguments	<p>The checker is now raised when an external function is called with an argument that is not compatible with the prototype after integral promotion. For instance:</p> <pre>extern long foo(int); long bar(long i) { return foo(i); //Noncompliant: calls foo(int) with a long }</pre>

Rule	Description	Change
CERT C++: EXP39-C	Do not access a variable through a pointer of an incompatible type	<p>The checker now detects situations where you access memory that is reallocated to an object of a different type without reinitializing the contents first.</p> <p>For instance, in the example, the checker raises a flag if you read the memory that <code>Bptr</code> points to, without reinitializing the memory first.</p> <pre>struct A; struct B; struct A *Aptr = (struct A*) malloc(sizeof(struct A)); struct B *Bptr = (struct B*) realloc(Aptr, sizeof(struct B));</pre>
CERT C++: EXP50-CPP	Do not depend on the order of evaluation for side effects	<p>The checker shows more accurate results. For instance:</p> <ul style="list-style-type: none"> The checker reports only one violation when a variable is written and reused within the same expression. Previously, two violations were reported. The checker detects violations involving global variables, for instance, when the same global variable is written and reused in an expression.
CERT C++: FI040-C	Reset strings on <code>fgets()</code> or <code>fgetws()</code> failure	The checker now flags direct read operations on buffers written by <code>fgets</code> -family functions if the function calls might have failed and the buffers have possibly indeterminate contents. If the buffer written is <code>buf</code> , direct read operations include operations such as <code>buf[0]</code> or <code>*buf</code> .
CERT C++: FLP34-C	Ensure that floating-point conversions are within range of the new type	<p>The checker now detects overflows in conversions from floating point to integer types.</p> <p>In addition, if the same variable causes multiple conversion overflows, all overflows are flagged. Previously, only the first overflow in a block was flagged.</p>

ISO/IEC TS 17961

Rule	Description	Change
ISO/IEC TS 17961 [liberr]	Failing to detect and handle standard library errors	<p>The checker is now raised on sensitive functions that return pointers if the return value is not checked. For instance:</p> <pre>FILE *in; in = fmemopen (argv[1], strlen (argv[1]), "r")</pre> <p>Polyspace now raises a defect if <code>in</code> is not tested before it is used for accessing a file.</p>

JSF AV C++

Rule	Description	Change
JSF AV Rule 204.1	The value of an expression shall be the same under any order of evaluation that the standard permits.	The checker shows more accurate results. For instance: <ul style="list-style-type: none"> • The checker reports only one violation when a variable is written and reused within the same expression. Previously, two violations were reported. • The checker detects violations involving global variables, for instance, when the same global variable is written and reused in an expression.

MISRA C: 2004

Rule	Description	Change
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 checker shows more accurate results. For instance: <ul style="list-style-type: none"> • The checker reports only one violation when a variable is written and reused within the same expression. Previously, two violations were reported. • The checker detects violations involving global variables, for instance, when the same global variable is written and reused in an expression.

MISRA C: 2012

Rule	Description	Change
MISRA C:2012 Dir 4.7	If a function returns error information, then that error information shall be tested.	<p>The checker is now raised on sensitive functions that return pointers if the return value is not checked. For instance:</p> <pre>FILE *in; in = fmemopen (argv[1], strlen (argv[1]), "r")</pre> <p>Polyspace now raises a defect if <code>in</code> is not tested before it is used for accessing a file.</p>
MISRA C:2012 Rule 10.6	The value of a composite expression shall not be assigned to an object with wider essential type.	<p>The checker now flags unary composite operators when their output is assigned to an object with a wider essential type. For instance:</p> <pre>#include<stdint.h> void foo(uint8_t u8a, uint16_t u16a){ u8a = ~u8a; /* Compliant*/ u16a = ~u8a; /* Noncompliant */ }</pre>
MISRA C:2012 Rule 10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type	<p>The checker now flags unary composite operators when their output is cast to a different essential type category or a wider essential type. For instance:</p> <pre>#include<stdint.h> void foo(uint8_t u8a, uint16_t u16a){ u8a = ~u8a; /* Compliant*/ u16a = (uint16_t)(~u8a); /* Noncompliant */ }</pre>
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.	<p>The checker shows more accurate results. For instance:</p> <ul style="list-style-type: none"> The checker reports only one violation when a variable is written and reused within the same expression. Previously, two violations were reported. The checker detects violations involving global variables, for instance, when the same global variable is written and reused in an expression.

Rule	Description	Change
MISRA C:2012 Rule 20.1	#include directives should only be preceded by preprocessor directives or comments.	Starting in R2021b, Polyspace ignores code that is hidden by using conditional compilation directives such as #if or #ifdef.

MISRA C++:2008

Rule	Description	Change
MISRA C++:2008 Rule 0-3-2	If a function generates error information, then that error information shall be tested.	The checker is now raised on sensitive functions that return pointers if the return value is not checked. For instance: <pre>FILE *in; in = fmemopen (argv[1], strlen (argv[1]), "r")</pre> Polyspace now raises a defect if <code>in</code> is not tested before it is used for accessing a file.
MISRA C++:2008 Rule 5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.	The checker shows more accurate results. For instance: <ul style="list-style-type: none"> The checker reports only one violation when a variable is written and reused within the same expression. Previously, two violations were reported. The checker detects violations involving global variables, for instance, when the same global variable is written and reused in an expression.
MISRA C++:2008 Rule 7-1-1	A variable which is not modified shall be const qualified.	When variables of reference types T& are modified, the checker no longer flags them (or the variables referenced to) as candidates for const-qualification.
MISRA C++:2008 Rule 11-0-1	Member data in non-POD class types shall be private.	The checker flags nonprivate data members in classes that are not POD types.
MISRA C++:2008 Rule 15-5-3	The terminate() function shall not be called implicitly.	The checker now flags an <code>atexit</code> handler that raises an unhandled exception. A termination handler that is passed to <code>std::atexit</code> raises an unhandled exception. For instance: <pre>#include <stdexcept> void atexit_handler(){//Noncompliant throw std::runtime_error("Error in atexit function"); } void main(){//Noncompliant try{ //... std::atexit(atexit_handler); }catch(std::exception& e){ //... } }</pre>

Rule	Description	Change
MISRA C++:2008 Rule 16-0-1	#include directives should only be preceded by preprocessor directives or comments.	The checker ignores code that is hidden by using conditional compilation directives such as #if or #ifdef.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Compatibility Considerations

If you checked your code for the preceding rules, you might see a change in the number of violations.

Updated Bug Finder defect checkers

In R2021b, these defect checkers have been updated.

Defect	Description	Update
Float conversion overflow	Overflow when converting between floating point data types	If the same variable causes multiple conversion overflows, all overflows are flagged. Previously, only the first overflow in a block was flagged.
Invalid use of standard library string routine	Standard library string function called with invalid arguments	<p>The checker now checks C++ <code>std::string</code> methods:</p> <ul style="list-style-type: none"> For methods that take a <code>const char*</code> argument <code>str</code>, the check raises a defect if <code>str</code> is NULL or points to an invalid string. <p>For instance:</p> <pre>std::string str = ""; const char txt[3] = {'n','o','p'}; str.append(txt); //txt is not a valid string</pre> <ul style="list-style-type: none"> For methods that take a <code>const char*</code> argument <code>str</code> and a <code>size_t</code> argument <code>n</code>, the check raises a defect if <code>str</code> is NULL or points to a buffer of more than <code>n</code> characters. <p>For instance:</p> <pre>std::string str = ""; const char txt[3] = {'n','o','p'}; str.append(txt,4); //txt points to a 3-character buffer</pre>
Line with more than one statement	Multiple statements on a line	The checker no longer flags C++17 <code>if</code> and <code>switch</code> statements with initializers.

Defect	Description	Update
Missing <code>constexpr</code> specifier	<code>constexpr</code> specifier can be used on variable or function for compile-time evaluation	The checker: <ul style="list-style-type: none"> No longer flags variables initialized with literal values as potential <code>constexpr</code> variables (since there might be no performance gain in declaring them <code>constexpr</code>). Flags functions that contain only a return statement involving the function parameters as potential <code>constexpr</code> functions.
Returned value of a sensitive function not checked	Sensitive functions called without checking for unexpected return values and errors	The checker now flags sensitive functions that return pointers if the return value is not checked. For instance: <pre>FILE *in; in = fopen(argv[1], "r")</pre> Polyspace now raises a defect if <code>in</code> is not tested before it is used for accessing a file.
Unmodified variable not <code>const</code> -qualified	Variable not <code>const</code> -qualified but variable value is not modified during its lifetime.	The checker no longer shows these false positives: <ul style="list-style-type: none"> When variables of reference types T& are modified, the checker no longer suggests them (or the variables referenced to) as candidates for <code>const</code>-qualification. The checker no longer flags variables declared with <code>decltype(auto)</code> as candidates for <code>const</code>-qualification (even if the variables are unchanged later).

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Compatibility Considerations

If you checked your code for the preceding defect checkers, you might see a change in the number of defects.

Reviewing Results

Results in Macros: See results in macro expansions when macro parameters cause an issue

In R2021b, if a function-like macro causes a defect or coding standard violation, the result is displayed on the root cause of the issue: the macro parameter or the macro definition.

For instance:

- In this example, the definition of macro `LEFTOVER()` contains a lowercase `l` and violates MISRA C:2012 Rule 7.3. This result is shown on the macro definition.

```
#define LEFTOVER(size) 10000ul - size /* Noncompliant */
#define REMAINDER(size) 10000UL - size /* Compliant */

void func(int arrSize) {
    int n = LEFTOVER(arrSize);
    int m = REMAINDER(arrSize);
}
```

- In this example, the definition of macro `COPY_ELEMENT()` results in an ambiguous evaluation order and violates MISRA C:2012 Rule 13.2 only when the parameter `i++` is passed to it. This result is shown on the macro expansion, specifically on the parameter in the expansion.

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);          /* Compliant */
    COPY_ELEMENT (i++);       /* Noncompliant */
}
```

See also Polyspace Results in Lines Containing Macros.

This way of showing results in function-like macros enables you to easily fix them:

- For issues caused by the macro definition, you can implement the fix once. Tools that report on the macro expansion can show multiple violations for one root cause.

In the preceding example, you can change the lowercase `l` in `LEFTOVER()` to fix the issue. The `REMAINDER()` macro shows this fix.

- For issues caused by the macro parameters, you can also implement the fix once.

In the preceding example, you can compute `i++` in a separate step, and then pass `i` to the `COPY_ELEMENT()` macro to fix the issue.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Access

Compatibility Considerations



You might see an increase in the number of defects and coding standard violations associated with function-like macros. If a macro is defined in a header and the header is excluded from the analysis,

previously, an issue caused by macro parameters was associated with the macro definition and suppressed from display. These issues are now shown on the macro expansion and will appear in your results.

To revert to the previous behavior, use the option `-defect-in-macro` definition.

Additional Info in Result Details: See expected and actual values for numerical defects

In R2021b, the result details for several numerical defects provide additional information to improve understanding of the defects. For instance, the result details for `Float conversion overflow` looks like this figure.

○ ID 8478: Integer conversion overflow (Impact: High)  

Conversion from type `int 64` to type `int 16` overflows.

Additional Info:

Expected values: [-32768 .. 32767].

Actual values: { 12479232 , 16674560 }.

Risk: Truncation or wrap-around of *value* to fit destination type might lead to unexpected results.

Fix: Ensure that the destination type is larger than or same as the type of *value*.

	Event	File	Scope	Line
1	Assignment to local variable 'value'	numerical.c	bug_intconvovfl()	108
2	○ Integer conversion overflow	numerical.c	bug_intconvovfl()	112

A numerical defect occurs when some value is expected to be within a range but falls outside that range. For instance, a value being converted to a floating-point data type is expected to be within the range of values that the data type can accommodate. Otherwise, a float conversion overflow occurs. For these kinds of defects, it is useful to know the expected values, and one or more actual values that can occur in context of the code.

The additional information for numerical defects include:

- **Expected values:** This line shows the range of expected values. If one or more actual values fall outside this range, the defect occurs. The reason why certain values are expected and the expected values depend on the defect checker. For instance, when the analysis checks for overflows, a data type determines the expected values.
- **Actual values:** This line shows the values of variables causing the defect along one or more execution paths within the code.
- **Risk:** This line describes why you must fix the defect. You can find an elaboration of this risk in the contextual help for the result.
- **Fix:** This line points to one or more possible fixes. You can find more details on the fixes, and one or more examples in the contextual help for the result.


See also Numerical Defects.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Access

Results Review: Open review history, select layout, and open additional panes by using fewer clicks

In R2021b, use the new **Layout** menu on the toolstrip to select the layout of the Polyspace Access interface.

To open additional panes, such as the **Review History**, use the **Window** menu and make a selection.

You can also open the **Review History** by clicking  in the **Result Details** pane.


Previously, you used **Window > Layout** to select a layout and **Window > Show/Hide View** to open additional panes.

Product: Polyspace Bug Finder Access.

Results Review: View relevant information in review panes when you select a finding

In R2021b, if you select a finding in the **Result Details** or **Source Code** panes, these panes are updated with information about the selected finding (if available):

- **Call Hierarchy**
- **Contextual Help**
- **Data Race Graph**
- **Error Call Graph**
- **Variable Access Graph**

Previously, except for the **Result Details** pane, the panes were not updated when you selected a different finding. For instance, after selecting a different finding, you clicked  in the **Result Details** pane to update the **Contextual Help** pane.

Product: Polyspace Bug Finder Access.

Functionality Removed: Polyspace Metrics

The Polyspace Metrics web dashboard is removed in R2021b. You get an error if you use:

- The `polyspace-results-repository` command.
- Option `-add-to-results-repository` when you run an analysis on a remote cluster.
- A configuration that attempts to start a Polyspace Metrics server automatically. The configuration is typically stored in a `.conf` file in the `%APPDATA%` (Windows) or `/etc/polyspace` (Linux) folder.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

Compatibility Considerations

To continue monitoring the quality of your code in a web browser, use Polyspace Access, which has 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, through 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:

- Review Polyspace Bug Finder Results in Web Browser
- Upload Results to Polyspace Access
- Migrate Results from Polyspace Metrics to Polyspace Access

Functionality Being Removed: Report generation from pre-R2015a results

Report generation from pre-R2015a Polyspace results will not be supported in a future release. In releases R2015a and earlier, Polyspace products generated results in a format that will no longer be supported for report generation.

Compatibility Considerations

Typically, you do not require support for report generation from earlier releases since you presumably have archived reports generated using the earlier release. To generate reports from pre-R2015a releases using a newer release, first rerun the analysis using the newer release and regenerate the results in a supported format, and then generate reports. See also Generate Reports from Polyspace Results.

Polyspace Access Installation

User Management: Set project permissions at the group level

In R2021b, you can use groups to manage project permissions for large sets of users. Import groups from your company Lightweight Directory Access Protocol (LDAP) or create custom groups in the **User Manager** interface, and then assign roles to those groups to authorize or prevent them from viewing projects in Polyspace Access. All members of the group inherit the role that you assign to the group.

To assign a role to a group:

- In the Polyspace Access interface, right-click a project in the **Project Explorer** and select **Manage Project Permissions**.
- At the command line, use `polyspace-access` with options `-set-role` and `-group`.

See Manage Project Permissions.

Previously, you managed project permissions by assigning roles to users individually.

Product: Polyspace Bug Finder Access.

User Management: Update list of users and groups more quickly by reloading web browser

In R2021b, Polyspace Access populates its list of users and groups from the **User Manager** database. If you add new users or groups to the **User Manager** database, refresh the list of users and groups by logging into the Polyspace Access interface or by reloading your web browser if you are already logged in.

Previously, if you added users to the **User Manager** database, you had to restart the Polyspace Access services to refresh the list of users in Polyspace Access.

Product: Polyspace Bug Finder Access.

User Authentication: Authenticate user logins against custom identities and LDAP identities simultaneously

In R2021b, you can configure Polyspace Access to simultaneously authenticate users against credentials from your organization's Lightweight Directory Access Protocol (LDAP) and against custom credentials. If you create custom user profiles, those users still can log into Polyspace Access after you start using your organization's LDAP to authenticate users.

See Configure **User Manager**.

Previously, you configured Polyspace Access to authenticate users against custom credentials or LDAP credentials, but not both at the same time.

Product: Polyspace Bug Finder Access.

Compatibility Considerations

If you configured Polyspace Access to use your company LDAP in a previous release, and you reuse the `settings.json` file in R2021b, in the **Admin** interface, go to the settings and select **Connect an LDAP directory**.

Polyspace Access Services: Faster results uploads and more responsive source code view

In R2021b, when you upload results to the Polyspace Access database for review, the upload is up to 10% faster than in R2021a.

When you review the uploaded results in your web browser, the **Source Code** pane is more responsive as you scroll through the code or switch between different files. In R2021a, the pane takes up to 40% longer to display the source code when you scroll through code or switch between files.

These performance improvements are more noticeable with large files and with source code that contains a large number of findings.

Product: Polyspace Bug Finder Access.

R2021a

Version: 3.4

New Features

Bug Fixes

Compatibility Considerations

Analysis Setup

Bug Finder Analysis Engine for Single Source File : Run Polyspace as You Code analysis and view results in your IDE or code editor

In R2021a, you can use the new Polyspace as You Code capability to check your code for bugs and coding standard violations while you work in your IDE or code editor. The analysis runs on only the currently active file. You can identify and fix issues early in the development cycle.

With Polyspace as You Code, you can:

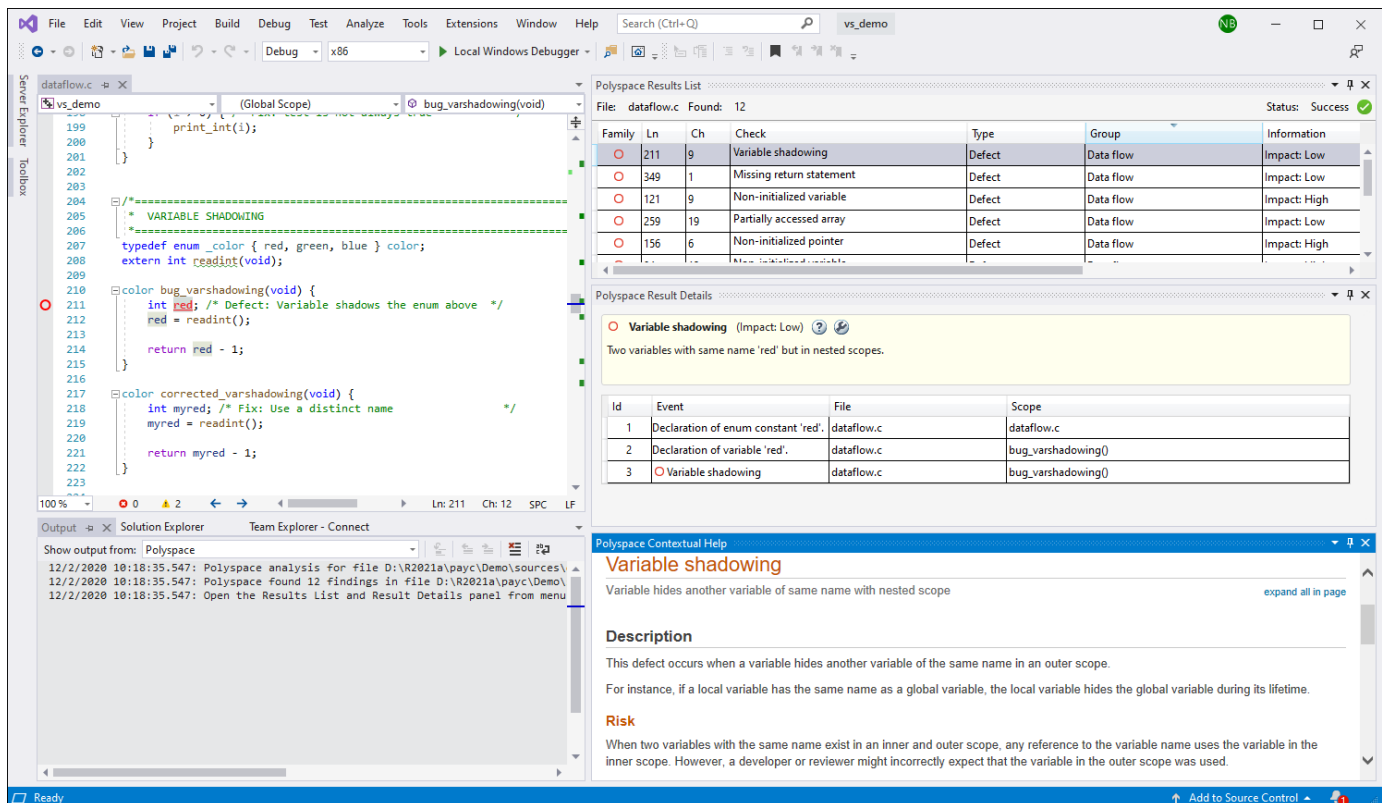
- Start an analysis of the currently active file on save or on demand.
- Extract analysis options from your IDE project, your build command, or your JSON compilation database.
- Import analysis options from a Polyspace PSPRJ project file.
- Leverage results reviews from integration analyses uploaded in Polyspace Access to hide already justified results and focus on new findings.

The Polyspace as You Code analysis engine and IDE extensions are available for download from the Polyspace Access web interface. You must have a valid Polyspace Access license.

Product: Polyspace Bug Finder Access (Polyspace as You Code).

Polyspace Extension for Visual Studio : Run Polyspace As You Code analysis and view results in Visual Studio IDE

In R2021a, you can use the new Polyspace as You Code extension to check your code for bugs and coding standards violations while you code in your Visual Studio IDE.



After you install the extension and the Polyspace as You Code analysis engine, you can:

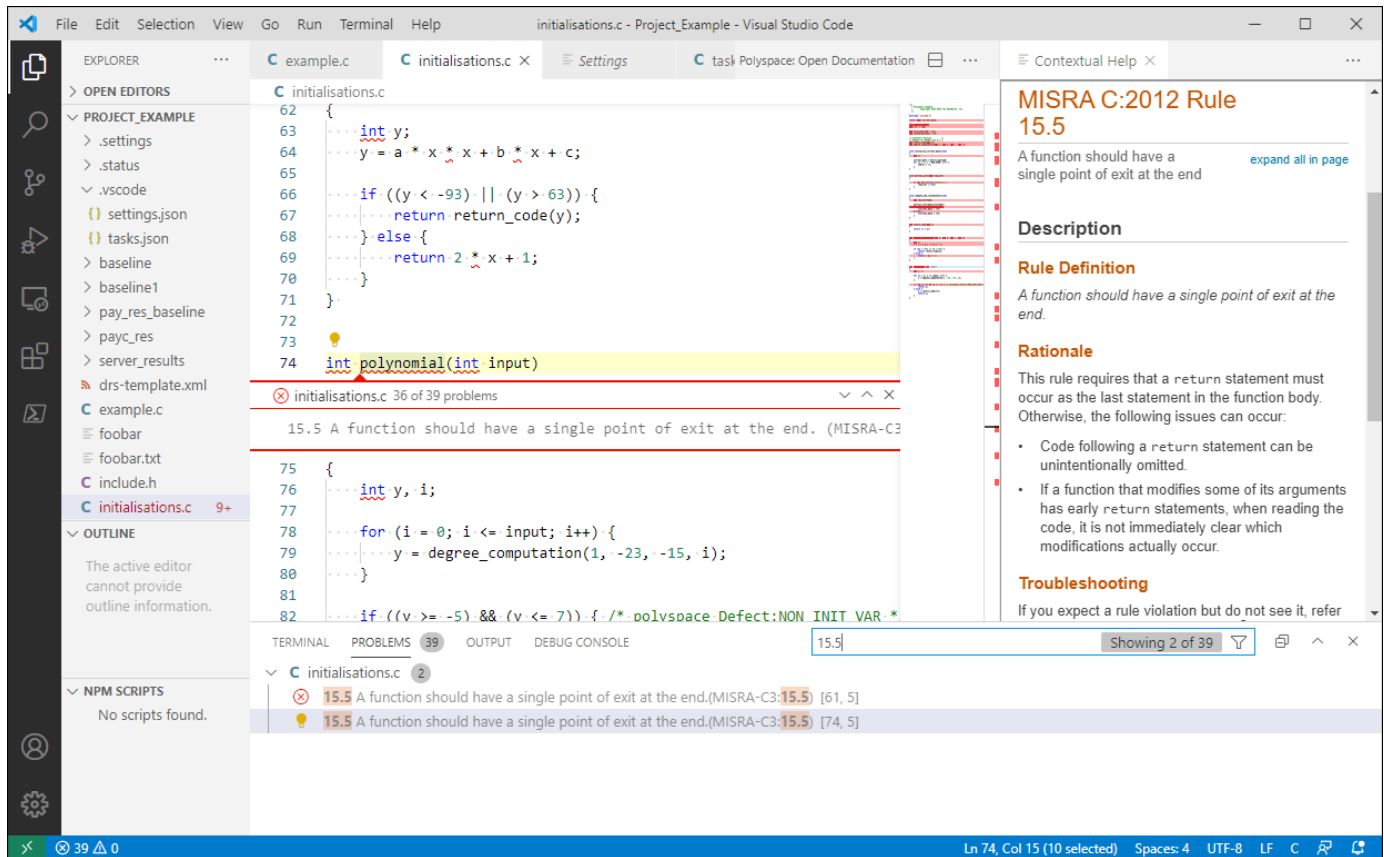
- Start an analysis of the currently active file on save or on demand.
- Extract analysis options from your Visual Studio project or build command.
- Import analysis options from a Polyspace PSPRJ project file.
- View highlighted defects in your source code and apply annotations in one click.
- Sort results in the **Results List** and open the **Result Details** and **Contextual Help** to learn more about a defect.
- Leverage results reviews from integration analyses uploaded in Polyspace Access to hide already justified results and focus on new findings.

The Polyspace as You Code analysis engine and IDE extensions are available for download from the Polyspace Access web interface. You must have a valid Polyspace Access license.

Product: Polyspace Bug Finder Access (Polyspace as You Code).

Polyspace Extension for Visual Studio Code : Run Polyspace As You Code analysis and view results in Visual Studio Code code editor

In R2021a, you can use the new Polyspace as You Code extension to check your code for bugs and coding standards violations while you code in your Visual Studio Code editor.



After you install the extension and the Polyspace as You Code, you can:

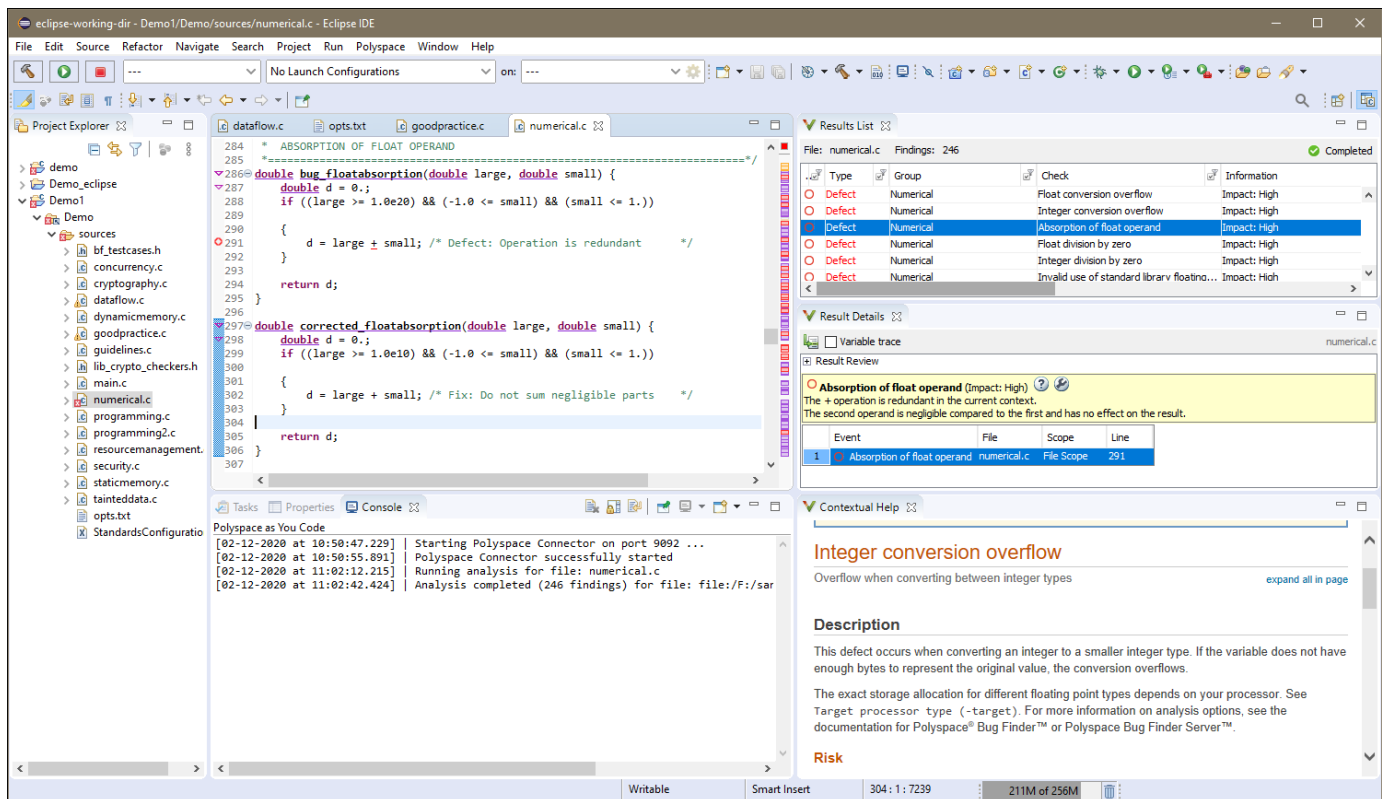
- Start an analysis of the currently active file on save or on demand.
- Extract analysis options from your Visual Studio Code build task or build command.
- Import analysis options from a Polyspace PSPRJ project file.
- View highlighted defects in your source code and apply annotations in one click.
- Filter results in the **Problems** pane and open the **Contextual Help** to learn more about a defect.
- Leverage results reviews from integration analyses uploaded in Polyspace Access to hide already justified results and focus on new findings.

The Polyspace as You Code analysis engine and IDE extensions are available for download from the Polyspace Access web interface. You must have a valid Polyspace Access license.

Product: Polyspace Bug Finder Access (Polyspace as You Code).

Polyspace Extension for Eclipse : Run Polyspace As You Code analysis and view results in Eclipse IDE

In R2021a, you can use the new Polyspace as You Code extension to check your code for bugs and coding standards violations while you code in your Eclipse IDE.



After you install the extension, you can:

- Start an analysis of the currently active file on save or on demand.
- Extract analysis options from your Eclipse project or build command.
- Import analysis options from a Polyspace PSPRJ project file.
- View highlighted defects in your source code and apply annotations in one click.
- Sort results in the **Results List** and open the **Result Details** and **Contextual Help** to learn more about a defect.
- Leverage results reviews from integration analyses uploaded in Polyspace Access to hide already justified results and focus on new findings.

The Polyspace as You Code analysis engine and IDE extensions are available for download from the Polyspace Access web interface. You must have a valid Polyspace Access license.

Product: Polyspace Bug Finder Access (Polyspace as You Code).

Configuration from Build System: Specify options delimiter and suppress console output

In R2021a, `polyspace-configure` has new options to simplify the creation of a Polyspace project or options file:

- `-options-for-sources-delimiter` — Use this option to specify an ASCII character that Polyspace uses as a delimiter between a group of analysis options. You typically use this option in combination with `-options-for-sources`, which associates a group of analysis options with

specific source files. You might want to specify a delimiter if, for instance, the default delimiter (;) is already used inside a macro.

- `-no-console-output` — Use this option to completely suppress the console output of `polyspace-configure`, including error and warning messages. By default, `polyspace-configure` emits errors and warnings only.

See also `polyspace-configure`

The new options allow you to customize the `polyspace-configure` runs without extensive additional scripting.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Configuration from Build System: Improved detection of incompatible software

In R2021a, if you use software that is not compatible with `polyspace-configure` when you trace your build process, `polyspace-configure` emits a message that identifies the software and that provides contextual help if applicable. Software that is not compatible with `polyspace-configure` includes some antivirus software and certain build systems such as Bazel.

For more information, see `polyspace-configure`.

Previously, when `polyspace-configure` could not trace your build process because of incompatible software, the command output did not identify the software. Now, you can easily check if your build system and environment is compatible with `polyspace-configure`.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Updated GCC Compiler Support: Set up Polyspace analysis for code compiled with GCC version 8.x

In R2021a, Polyspace supports the GCC compiler version 8.x natively. If you build your source code by using GCC version 8.x, you can specify the compiler name for your Polyspace analysis.

Target Environment	
Compiler	gnu8.x
Target processor type	x86_64

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

Because of the native support, you can now set up a Polyspace project without knowing the internal workings of this compiler. The analysis can interpret macros that are implicitly defined by the compiler and compiler-specific language extensions such as keywords and pragmas.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Updated Microsoft Visual C++ Support: Set up a Polyspace analysis for code compiled with Visual Studio 2019

In R2021a, Polyspace supports the compiler Visual Studio 2019 natively. If you build your source code by using Visual Studio 2019 (versions 16.x), you can specify the compiler name for your Polyspace analysis.

Target Environment	
Compiler	visual16.x
Target processor type	x86_64

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

Because of the native support, you can now set up a Polyspace project without knowing the internal workings of this compiler. The analysis can interpret macros that are implicitly defined by the compiler and compiler-specific language extensions such as keywords and pragmas.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Modifying Checker Behavior: Modify parameters for MISRA C:2012 rules 1.1 and 5.1 to 5.5

In R2021a, you can modify the thresholds used in the checkers for MISRA C: 2012 Rules 1.1 and 5.1 to 5.5.

Rule	Description	Supported Modification
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.	<p>You can increase or decrease these parameters of the rule checker:</p> <ul style="list-style-type: none"> Maximum depth of nesting allowed in control flow statements Maximum levels of inclusion allowed using include files Maximum number of constants allowed in an enumeration Maximum number of macros allowed in a translation unit Maximum number of members allowed in a structure Maximum levels of nesting allowed in a structure

Rule	Description	Supported Modification
MISRA C:2012 Rule 5.1	These rules require uniqueness of certain types of identifiers. For instance, rule 5.1 requires that external identifiers be distinct.	If the difference between two identifiers occurs beyond the first <i>num</i> characters, the rule checker considers the identifiers as identical. You can modify the parameter <i>num</i> separately for external and internal identifiers.
MISRA C:2012 Rule 5.2		
MISRA C:2012 Rule 5.3		
MISRA C:2012 Rule 5.4		
MISRA C:2012 Rule 5.5		

For more information, see:

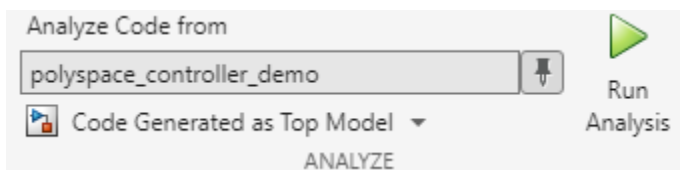
- [Modify Default Behavior of Bug Finder Checkers](#)
- [-code-behavior-specifications](#)

Benefits: You can adapt the checkers for MISRA C: 2012 Rules 1.1 and 5.1 to 5.5 to follow your compiler specifications.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Simulink Support: Start Polyspace analysis without an explicit code generation step

In R2021a, start the Polyspace analysis of generated code without having to explicitly generate the code first. To start the Polyspace analysis of code generated from a model, Click **Run Analysis** in the Simulink toolstrip.



If you have Embedded Coder, Polyspace generates code from the model by using Embedded Coder when there is no previously generated code corresponding to the model. After the code generation is complete, the Polyspace analysis starts.

See [Run Polyspace Analysis on Code Generated from Simulink Model](#).

Previously, you generated code explicitly in a separate step before starting the Polyspace analysis of the generated code. You are no longer required to perform this step.

Additional Considerations: Before starting a Polyspace analysis, you still need to generate code explicitly if any of the following is true:

- You do not use Embedded Coder to generate code.
- The model is configured to generate code as a model reference.

Product: Polyspace Bug Finder (Desktop).

polyspacesetup Function : Integrate Polyspace with MATLAB in fewer steps

In R2021a, you can integrate Polyspace with the current or earlier release of MATLAB in fewer steps. When you run the function `polyspacesetup` at the MATLAB command prompt, the function looks for a Polyspace installation in the default location. If the installation exists, the function integrates Polyspace with MATLAB. Specify the installation location explicitly only when you install Polyspace in a nondefault location.

See Also:

- `polyspacesetup`
- Integrate Polyspace with MATLAB and Simulink

Previously, to integrate Polyspace with Simulink, you provided the location of the Polyspace installation folder. Starting in R2021a, providing the installation location is no longer required if you install Polyspace in the default location.

Product: Polyspace Bug Finder (Desktop).

pslinkrunCrossRelease Function : Analyze code generated in an earlier release of Simulink by using a later release of Polyspace

In R2021a, you can run a Polyspace analysis of generated code from an earlier release of Simulink by using the function `pslinkrunCrossRelease`. To use this cross-release workflow, your Polyspace version must be later than your Simulink version and your Simulink must be R2020b or later.

See :

- `pslinkrunCrossRelease`
- Run Polyspace on Code Generated by Using Previous Releases of Simulink

Previously, you used the function `pslinkrun` in both cross-release and same release workflows. Starting in R2021a, these two workflows are differentiated by introducing the function `pslinkrunCrossRelease` explicitly for the cross-release workflow.

The compatibility of Polyspace with prior releases of Simulink is also simplified. Previously, the compatibility of Polyspace with an earlier Simulink depended on the specific version of Polyspace and Simulink. Starting in R2021a, you can integrate Polyspace with Simulink only if your Polyspace version is later than your Simulink version, and you have Simulink from R2020b or later. See Polyspace Support of MATLAB and Simulink from Different Releases.

Product: Polyspace Bug Finder (Desktop).

Compatibility Considerations

The function `pslinkrun` no longer supports a cross-release workflow. Use the function `pslinkrunCrossRelease` instead.

Functionality being removed: Compilation assistant

The Polyspace compilation assistant will be removed in a future release.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Compatibility Considerations

If you use the compilation assistant in your Polyspace project, clear the corresponding option. To clear this option in the desktop interface, go to **Tools > Preferences** and then select the **Project and Results Folder** tab.

Instead, when you set up your Polyspace project, you can:

- Use the `Compiler (-compiler)` option to specify a compiler that Polyspace supports natively if you compile your code by using that compiler.
- Use `polyspace-configure` to trace your build command and to obtain your compiler configuration. See `polyspace-configure`.

Changes in analysis options and binaries

-code-behavior-specifications takes only one file as argument

Behavior change

Starting in R2021a, this option only takes one XML file as argument. If you were specifying code behaviors in multiple XML files, combine their content into one file and provide this file as argument to the option.

See also `-code-behavior-specifications`.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

-sources-encoding with value other than auto disables automatic detection of encoding

Behavior change

Starting in R2021a, if you explicitly specify a value with the option `-sources-encoding` (or use the default value `system` which uses the default encoding of your OS), the analysis does not perform any automatic detection of source file encoding. For instance, if you use `-sources-encoding shift-jis`, the analysis internally converts your source files from Shift JIS (Shift Japanese Industrial Standards) to UTF-8 encoding before processing them. If you see regressions from previous releases, consider using `-sources-encoding auto` to reenabte the automatic detection of source encoding. Automatic detection is useful when your project contains, for instance, a mix of different encodings.

See also `Source code encoding (-sources-encoding)`.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access (Polyspace as You Code)

Analysis Results

AUTOSAR C++14 Support: Check for 327 AUTOSAR C++14 rules including 19 new rules in R2021a

In R2021a, you can look for violations of these AUTOSAR C++14 rules in addition to previously supported rules.

AUTOSAR C++14 Rule	Description	Polyspace Checker		
A2-7-3	All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation.	AUTOSAR A2-7-3	C++14	Rule
A2-8-1	A header file name should reflect the logical entity for which it provides declarations.	AUTOSAR A2-8-1	C++14	Rule
A2-8-2	An implementation file name should reflect the logical entity for which it provides definitions.	AUTOSAR A2-8-2	C++14	Rule
A8-4-5	"consume" parameters declared as X && shall always be moved from.	AUTOSAR A8-4-5	C++14	Rule
A8-4-6	"forward" parameters declared as T && shall always be forwarded.	AUTOSAR A8-4-6	C++14	Rule
A8-4-8	Output parameters shall not be used.	AUTOSAR A8-4-8	C++14	Rule
A8-4-9	"in-out" parameters declared as T & shall be modified.	AUTOSAR A8-4-9	C++14	Rule
A8-4-10	A parameter shall be passed by reference if it can't be NULL.	AUTOSAR A8-4-10	C++14	Rule
A8-5-4	If a class has a user-declared constructor that takes a parameter of type <code>std::initializer_list</code> , then it shall be the only constructor apart from special member function constructors.	AUTOSAR A8-5-4	C++14	Rule
A12-8-1	Move and copy constructors shall move and respectively copy base classes and data members of a class, without any side effects.	AUTOSAR A12-8-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
A12-8-2	User-defined copy and move assignment operators should use user-defined no-throw swap function.	AUTOSAR A12-8-2	C++14	Rule
A12-8-3	Moved-from object shall not be read-accessed.	AUTOSAR A12-8-3	C++14	Rule
A13-5-3	User-defined conversion operators should not be used.	AUTOSAR A13-5-3	C++14	Rule
A13-6-1	Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits.	AUTOSAR A13-6-1	C++14	Rule
A15-4-1	Dynamic exception-specification shall not be used.	AUTOSAR A15-4-1	C++14	Rule
A15-4-4	A declaration of non-throwing function shall contain noexcept specification.	AUTOSAR A15-4-4	C++14	Rule
A20-8-1	An already-owned pointer value shall not be stored in an unrelated smart pointer.	AUTOSAR A20-8-1	C++14	Rule
A27-0-4	C-style strings shall not be used.	AUTOSAR A27-0-4	C++14	Rule
M5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	AUTOSAR M5-0-16	C++14	Rule

See also AUTOSAR C++14 Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

CERT C++ Support: Check for memory management and programming rule violations.

In R2021a, you can look for violations of these CERT C++ rules in addition to previously supported rules.

CERT C++ Rule	Description	Polyspace Checker
OOP50-CPP	Do not invoke virtual functions from constructors or destructors	CERT C++: OOP50-CPP
EXP63-CPP	Do not rely on the value of a moved-from object	CERT C++: EXP63-CPP

CERT C++ Rule	Description	Polyspace Checker
MEM56-CPP	Do not store an already-owned pointer value in an unrelated smart pointer	CERT C++: MEM56-CPP

See also CERT C++ Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

MISRA C++:2008 Support: Check for disallowed pointer arithmetic

In R2021a, you can look for violation of this MISRA C++:2008 rule in addition to previously supported rules.

Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	MISRA C++:2008 Rule 5-0-16

See also MISRA C++:2008 Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

MISRA C:2012 Support: Checkers updated to account for MISRA C:2012 Technical Corrigendum 1 and Amendment 2

In R2021a, Polyspace supports amendments to MISRA C:2012 rules in Technical Corrigendum 1 and Amendment 2.

MISRA C:2012 Technical Corrigendum 1

MISRA C:2012 Technical Corrigendum 1 adds clarifications to existing rules. The clarifications have led to changes in these checkers:

Rule	Description	Update in Technical Corrigendum 1
MISRA C:2012 Rule 10.1	Operands shall not be of an inappropriate essential type.	The rule now explicitly forbids use of pointer types with logical operands such as &&, and !.
MISRA C:2012 Rule 10.5	The value of an expression should not be cast to an inappropriate essential type.	The rule now forbids casts of integer constants with value 0 or 1 to essentially enum types.

Rule	Description	Update in Technical Corrigendum 1
MISRA C:2012 Rule 11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.	The rule now takes into account only the unqualified types that the pointers point to. For instance, if a pointer is assigned to another and the only difference between the pointed types is a <code>const</code> qualifier, the rule does not consider this assignment as a conversion.
MISRA C:2012 Rule 11.4	A conversion should not be performed between a pointer to object and an integer type.	The rule now applies explicitly to pointers to objects only. Conversions between an integer type and other pointer types such as <code>void*</code> or pointers to functions are flagged by other rules.
MISRA C:2012 Rule 11.9	The macro <code>NULL</code> shall be the only permitted form of integer null pointer constant.	The rule allows the use of <code>{0}</code> to initialize aggregates or unions containing pointers.
MISRA C:2012 Rule 14.2	A for loop shall be well-formed.	The rule allows any form of initialization of the loop counter as long as the initialization does not have other side effects.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

MISRA C:2012 Amendment 2

MISRA C:2012 Amendment 2 addresses the new language features in the C11 standard. All updates in Amendment 2 have been incorporated in the checkers.

Rule	Description	Update in Amendment 2
MISRA C:2012 Rule 1.4	Emergent language features shall not be used.	This rule is new in Amendment 2.
MISRA C:2012 Rule 12.1	The precedence of operators within expressions should be made explicit.	The rule now mandates a violation if the operand of the <code>_Alignof</code> operator is not enclosed in parenthesis.
MISRA C:2012 Rule 21.3	The memory allocation and deallocation functions of <code><stdlib.h></code> shall not be used.	The rule now flags uses of the <code>aligned_alloc</code> function.

Rule	Description	Update in Amendment 2
MISRA C:2012 Rule 21.8	The Standard Library termination functions of <stdlib.h> shall not be used.	The rule no longer flags system. In addition to <code>exit</code> and <code>abort</code> , the rule now flags <code>_Exit</code> and <code>quick_exit</code> .
MISRA C:2012 Rule 21.21	The Standard Library function system of <stdlib.h> shall not be used.	This rule is new in Amendment 2.
MISRA C:2012 Rule 22.1	All resources obtained dynamically by means of Standard Library functions shall be explicitly released.	The rule now flags memory allocation using the <code>aligned_alloc</code> function if the memory is not released.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Guidelines: New checkers for software complexity defects

In R2021a, Polyspace has a new category of checkers called **Guidelines**. This category contains the **Software Complexity** checkers. Reduce the software complexity metrics of your code by activating these new checkers. See Reduce Software Complexity by Using Polyspace Checkers. The **Software Complexity** checkers include:

Defect	Description
Number of Calling Functions Exceeds Threshold	The number of distinct callers of a function is greater than the defined threshold.
Number of Called Functions Exceeds Threshold	The number of distinct function calls within the body of a function is greater than the defined threshold.
Comment density below threshold	The comment density of the module falls below the specified threshold.
Call Tree Complexity Exceeds Threshold	The call tree complexity of a file is greater than the defined threshold.
Number of Lines Within body Exceeds Threshold	The number of lines in the body of a function is greater than the defined threshold.
Number of Executable Lines Exceeds Threshold	The number of executable lines in the body of a function is greater than the defined threshold.
Number of Call Levels Exceeds Threshold	The nesting depth of control structures in a function is greater than the defined nesting depth threshold of a function.
Number of GOTO Statements Exceeds Threshold	The number of <code>goto</code> statements in a function is greater than the defined threshold.
Number of Local Static variables Exceeds Threshold	The number of local static variables in a function is greater than the defined threshold.

Defect	Description
Number of Local Nonstatic Variables Exceeds Threshold	The number of function calls in a function is greater than the defined call occurrence threshold of a function.
Number of Call Occurrences Exceeds Threshold	The number of function calls in a function is greater than the defined call occurrence threshold of a function.
Number of Function Parameters Exceeds Threshold	The number of arguments of a function is greater than the defined threshold.
Number of Paths Exceeds Threshold	The number of static paths in a function is greater than the defined threshold.
Number of Return Statements Exceeds Threshold	The number of return statements in a function is greater than the defined threshold.
Number of Instructions Exceeds Threshold	The number of instructions in a function is greater than the defined threshold.
Number of Lines Exceeds Threshold	The number of total lines in a file is greater than the defined threshold.
Cyclomatic Complexity Exceeds Threshold	The cyclomatic complexity of a function is greater than the defined cyclomatic complexity threshold of a function.
Language Scope Exceeds Threshold	The language scope of a function is greater than the defined threshold.

In the Polyspace user interface, activate these checkers in the **Coding Standard & Code Metric** node of the **Configuration** pane. Alternatively, in the Checkers selection window, select the **Guidelines > Software Complexity** checkers.

To activate these checkers in the command-line, use the analysis option `Check Guidelines (-guidelines)`. To specify a subset of these checkers with modified thresholds by using a checkers selection file, use `Set checkers by file (-checkers-selection-file)`.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Compatibility Considerations

Each of these software complexity checkers corresponds to a code metric. When you import comments from a previous run by using the command `polyspace-comments-import`, Polyspace copies any review information on a code metric in the previous result to the corresponding software complexity checker in the current result. If the current result contains the same code metric, the review information is also copied to the code metric.

JSF AV C++ Support: Check for cases where pass-by-reference is preferred to pass-by-pointer

In R2021a, you can check for this JSF® AV C++ rule in addition to previously supported rules.

Rule	Description
AV Rule 117	Arguments should be passed by reference if NULL values are not possible.

See also JSF AV C++ Coding Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

New Bug Finder Checkers: Check for inefficient string operations, noncompliance with AUTOSAR Standard specifications, and other issues

In R2021a, you can check for these new Bug Finder defects in your code.

Defect	Description
Const rvalue reference parameter may cause unnecessary data copies	The const-ness of an rvalue reference prevents move operation and causes a more expensive copy operation instead.
Expensive use of <code>std::string</code> methods instead of more efficient overload	An <code>std::string</code> method uses a single character string literal, that is, a <code>const char*</code> object of length one, instead of using a single quoted character.
Expensive use of <code>std::string</code> with empty string literal	Use of <code>std::string</code> with empty string literal can be replaced by less expensive calls to <code>std::basic_string</code> member functions.
Expensive use of non-member <code>std::string</code> operator+() instead of a simple append	The non-member <code>std::string</code> operator+() function is called when the append (or +=) method would have been more efficient.
Expensive local variable copy	A local variable is created by copy from a <code>const</code> reference and not modified later.
Expensive logical operation	A logical operation requires the evaluation of both operands because of their order, resulting in inefficient code.
File does not compile	A file has a compilation error.
Noncompliance with AUTOSAR library	A call to an AUTOSAR RTE API function violates AUTOSAR Standard specifications.
Use of <code>new</code> or <code>make_unique</code> instead of more efficient <code>make_shared</code>	Creating a <code>shared_ptr</code> pointer with <code>new</code> or <code>make_unique</code> causes an unnecessary additional memory allocation.

For all defect checkers, see Defects.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Changes to coding standards checking

In R2021a, coding rules checking has improved across various standards. For instance, you can check for both MISRA C:2004 and MISRA C:2012 rules in the same run.

These changes have been made in the checking of previously supported rules.

AUTOSAR C++14

Rule	Description	Change
AUTOSAR C++14 Rule A2-10-4	The identifier name of a non-member object with static storage duration or static function shall not be reused within a namespace.	The checker is no longer raised on unused code such as: <ul style="list-style-type: none"> • Noninstantiated templates • Uncalled <code>static</code> or <code>extern</code> functions • Uncalled and undefined local functions • Unused types and variables
AUTOSAR C++14 Rule A2-10-5	An identifier name of a function with static storage duration or a non-member object with external or internal linkage should not be reused.	The checker is no longer raised on unused code such as: <ul style="list-style-type: none"> • Noninstantiated templates • Uncalled <code>static</code> or <code>extern</code> functions • Uncalled and undefined local functions • Unused types and variables
AUTOSAR C++14 Rule A5-0-3	The declaration of objects shall contain no more than two levels of pointer indirection.	The checker no longer flags the use of objects with more than two levels of pointer indirection.
AUTOSAR C++14 Rule A5-1-1	Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.	The checker no longer flags use of literal values in initializations involving <code>std::initializer_list</code> -s used to call constructors, for instance, when initializing <code>std::vector</code> -s of strings.

Rule	Description	Change
AUTOSAR C++14 Rule A8-5-2	Braced-initialization {}, without equals sign, shall be used for variable initialization.	<p>The checker now adheres more strictly to the AUTOSAR C++14 specifications. The checker flags non-uniform initializations such as:</p> <ul style="list-style-type: none"> • <i>Type</i> obj1 = obj2; • <i>Type</i> obj1(obj2); <p>Even if obj1 and obj2 have the same types. Previously, the checker raised a flag only if the types were different.</p> <p>The checker allows an exception for these cases:</p> <ul style="list-style-type: none"> • Initialization of variables with type <code>auto</code> using a simple assignment • Initialization of reference types • Declarations with global scope using the format <i>Type</i> a() where <i>Type</i> is a class type with default constructor. The analysis interprets a as a function returning the type <i>Type</i>. • Loop variable initialization in OpenMP parallel for loops, that is, in for loop statements that immediately follow <code>#pragma omp parallel for</code>
AUTOSAR C++14 Rule A10-1-1	Classes shall not be derived from more than one base class which is not an interface class.	The checker now expands interface classes to include constructors and destructors set to <code>=default</code> or <code>=delete</code> .
AUTOSAR C++14 Rule A12-6-1	All class data members that are initialized by the constructor shall be initialized using member initializers.	The checker no longer flags constructors that use default member initialization.
AUTOSAR C++14 Rule A12-8-7	Assignment operators should be declared with the ref-qualifier &.	The checker no longer flags deleted assignment operators without the ref-qualifier &.

Rule	Description	Change
<ul style="list-style-type: none"> AUTOSAR C++14 Rule A20-8-5 AUTOSAR C++14 Rule A20-8-6 	<p><code>std::make_unique</code> (<code>std::make_shared</code>) shall be used to construct objects owned by <code>std::unique_ptr</code> (<code>std::shared_ptr</code>).</p>	The checkers now also apply to <code>boost::unique_ptr</code> and <code>boost::shared_ptr</code> .
AUTOSAR C++14 Rule M5-0-15	Array indexing shall be the only form of pointer arithmetic.	The checker no longer flags arithmetic operations such as increment and decrement on iterators that point to elements in containers.
AUTOSAR C++14 Rule M3-2-1	All declarations of an object or function shall have compatible types.	<p>The checker considers two types to be compatible if they have the same size and signedness in the environment that you use. For instance, if you specify <code>-target</code> as <code>i386</code>, Polyspace considers <code>long</code> and <code>int</code> to be compatible types.</p> <p>The checker is no longer raised on unused code such as:</p> <ul style="list-style-type: none"> Noninstantiated templates Uncalled <code>static</code> or <code>extern</code> functions Uncalled and undefined local functions Unused types and variables
AUTOSAR C++14 Rule M3-2-2	The One Definition Rule shall not be violated.	<p>The checker is no longer raised on unused code such as:</p> <ul style="list-style-type: none"> Noninstantiated templates Uncalled <code>static</code> or <code>extern</code> functions Uncalled and undefined local functions Unused types and variables <p>In the declarations that violate these rules, the violations are flagged on the keywords instead of the variable names.</p>
AUTOSAR C++14 Rule M3-2-4	An identifier with external linkage shall have exactly one definition.	<p>The checker is no longer raised on unused code such as:</p> <ul style="list-style-type: none"> Noninstantiated templates Uncalled <code>static</code> or <code>extern</code> functions Uncalled and undefined local functions Unused types and variables

CERT C

Rule	Description	Change
CERT C: Rule EXP37-C	Call functions with the correct number and type of arguments	This checker now flags: <ul style="list-style-type: none"> • Calls with complex arguments to math functions that do not take a complex input • Calls to functions whose provided or deduced prototypes do not match their definitions.
CERT C: Rule INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data	The checker now detects comparisons of <code>time_t</code> variables with variables of other types. <code>time_t</code> is an implementation-defined type, therefore, these comparisons can lead to unexpected results.
CERT C: Rec. MEM04-C	Beware of zero-length allocations	The checker now performs more direct checks for possibilities of zero-length memory allocations and adheres more strictly to the CERT C standard.

CERT C++

Rule	Description	Change
CERT C+ +: DCL53- CPP	Do not write syntactically ambiguous declarations	The checker no longer flags ambiguous declarations with global scope. For instance, the analysis does not flag declarations with global scope using the format <code>Type a()</code> where <code>Type</code> is a class type with a default constructor. The analysis interprets <code>a</code> as a function returning the type <code>Type</code> .
CERT C+ +: DCL60- CPP	Obey the one-definition rule	The checker is no longer raised on unused code such as: <ul style="list-style-type: none"> • Noninstantiated templates • Uncalled <code>static</code> or <code>extern</code> functions • Uncalled and undefined local functions • Unused types and variables <p>In the declarations that violate these rules, the violations are flagged on the keywords instead of the variable names.</p>
CERT C+ +: EXP37-C	Call functions with the correct number and type of arguments	The checker now flags the calls to an <code>extern "C"</code> function if its prototypes does not match the definition.

JSF AV C++

Rule	Description	Change
JSF AV C++ Rule 137	All declarations at file scope should be static where possible.	The checker is raised on declarations of nonstatic objects that you use in only one file. The checker is raised even if you analyze a single file. The checker is not raised on the declarations of objects that remain unused, such as: <ul style="list-style-type: none">• Noninstantiated templates• Uncalled <code>static</code> or <code>extern</code> functions• Uncalled and undefined local functions• Unused types and variables

MISRA C: 2012

Rule	Description	Change
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.	You can now change the thresholds used in the rule checking using the option <code>-code-behavior-specifications</code> .
MISRA C:2012 Rule 1.3	There shall be no occurrence of undefined or critical unspecified behaviour.	The checker no longer flags all instances of the <code>offsetof</code> macro, but only the instances that cause undefined behavior. For instance, if the second argument of <code>offsetof</code> is not a field of the first argument or is a bitfield, the checker raises a violation.
MISRA C:2012 Rule 5.x	Rules that ensure uniqueness of identifiers.	You can now change the thresholds used in the rule checking using the option <code>-code-behavior-specifications</code> .
MISRA C:2012 Rule 10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.	The checker now detects implicit conversions when a structure is initialized using aggregate initialization. For instance, in this code snippet, the initialization of structure <code>a</code> results in an implicit conversion from the essentially signed type of <code>x</code> to the essentially unsigned type of the field <code>a_</code> <pre>typedef struct tag_A { unsigned char a_; unsigned char b_; }tag_A; int x = 1; tag_A a = {x,0}; //Noncompliant</pre>

Rule	Description	Change
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.	The checker treats macros such as TRUE or FALSE that resolve to 0 or 1 as essentially boolean.

MISRA C++:2008

Rule	Description	Change
MISRA C++:2008 Rule 2-10-5	The identifier name of a non-member object or function with static storage duration should not be reused.	The checker is no longer raised on unused code such as: <ul style="list-style-type: none"> • Noninstantiated templates • Uncalled <code>static</code> or <code>extern</code> functions • Uncalled and undefined local functions • Unused types and variables
MISRA C++:2008 Rule 3-2-1	All declarations of an object or function shall have compatible types.	The checker considers two types to be compatible if they have the same size and signedness in the environment that you use. For instance, if you specify <code>-target</code> as <code>i386</code> , Polyspace considers <code>long</code> and <code>int</code> to be compatible types. The checker is no longer raised on unused code such as: <ul style="list-style-type: none"> • Noninstantiated templates • Uncalled <code>static</code> or <code>extern</code> functions • Uncalled and undefined local functions • Unused types and variables
MISRA C++:2008 Rule 3-2-2	The One Definition Rule shall not be violated.	The checker is no longer raised on unused code such as: <ul style="list-style-type: none"> • Noninstantiated templates • Uncalled <code>static</code> or <code>extern</code> functions • Uncalled and undefined local functions • Unused types and variables <p>In the declarations that violate these rules, the violations are flagged on the keywords instead of the variable names.</p>
MISRA C++:2008 Rule 3-2-4	An identifier with external linkage shall have exactly one definition.	The checker is no longer raised on unused code such as: <ul style="list-style-type: none"> • Noninstantiated templates • Uncalled <code>static</code> or <code>extern</code> functions • Uncalled and undefined local functions • Unused types and variables

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Compatibility Considerations

If you checked your code for the preceding rules, you might see a change in the number of violations.

Updated Bug Finder defect checkers

In R2021a, these defect checkers have been updated.

Defect	Description	Update
Ambiguous declaration syntax	Declaration syntax can be interpreted as object declaration or part of function declaration	The checker no longer flags ambiguous declarations with global scope. For instance, the analysis does not flag declarations with global scope using the format <i>Type</i> <i>a()</i> where <i>Type</i> is a class type with a default constructor. The analysis interprets <i>a</i> as a function returning the type <i>Type</i> .
Format string specifiers and arguments mismatch	Format specifiers in printf-like functions do not match corresponding arguments	In cases where integer promotion modifies the perceived data type of an argument, the analysis result shows both the original type and the type after promotion. The format specifier has to match the type after integer promotion.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Reviewing Results

Simulink Block Annotation : Add multiple Polyspace annotations corresponding to multiple types of Polyspace results

In R2021a, you can annotate a Simulink block with multiple annotations for multiple types of Polyspace results through the **Polyspace Annotation** window. For instance, consider a block that is annotated for a MISRA C violation. If this block is then flagged for a defect violation, you can add an annotation corresponding to the defect violation without overwriting the previous annotation for the MISRA C violation. To add these two annotations, open the **Polyspace Annotation** window twice and each time, annotate for a specific type of result. These annotations are appended to each other and can be seen in the **Result Details** pane of Polyspace User Interface. See [Annotate Blocks to Justify Issues](#)

Previously, if you added a new annotation to an already annotated Simulink block, Polyspace overwrote the existing annotation. Starting in R2021a, adding an annotation to a previously annotated Simulink block appends the new annotation to the existing annotation.

Product: Polyspace Bug Finder (Desktop).

Review Information Between Polyspace Access Projects : Download results from Polyspace Access and import review information to a different project

In R2021a, you can merge review information between Polyspace Access projects.

To merge the review information between projects:

- 1 Use the `polyspace-access -download` command to download the results from the project where you already reviewed the findings.
- 2 Use the `-import-comments` option to import the review information as you run an analysis on the other project.
- 3 Upload the analysis results containing the imported review information to Polyspace Access.

See `polyspace-access` (Polyspace Bug Finder Server).

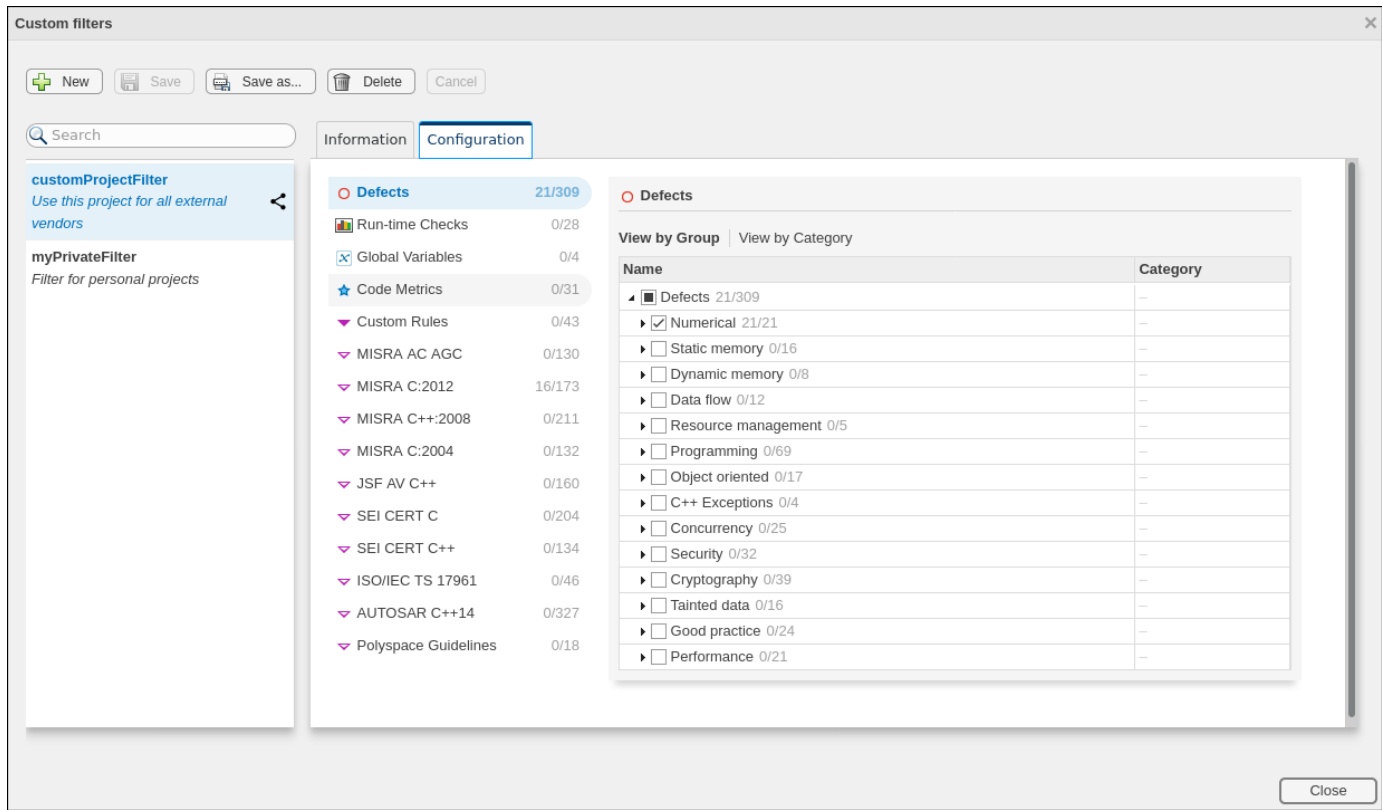
If you review findings in one project and you reuse the source code that contains those findings in another project, you do not need to review those findings again.

Product: Polyspace Bug Finder Access.

Results Review Scope : Define and share custom families of filters

In R2021a, you can create custom families of filters to tailor the scope of your review to results that are relevant to only your project or organization. You can then share the customized review scopes with other Polyspace Access users. See [Create Custom Filter Groups in Polyspace Access Web Interface](#) (Polyspace Bug Finder Access).

For example, you might want to review your code for violations of a subset of only **Numerical** defects and **MISRA C: 2012** rules.

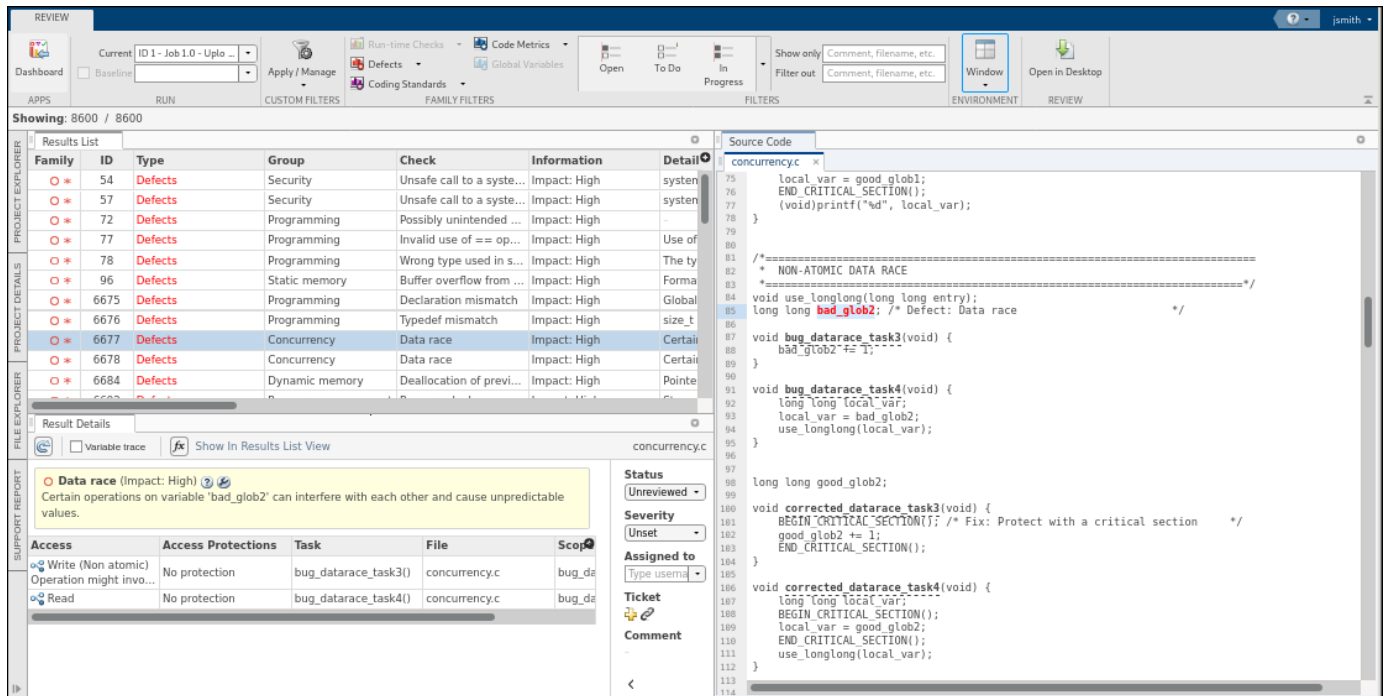


Product: Polyspace Bug Finder Access.

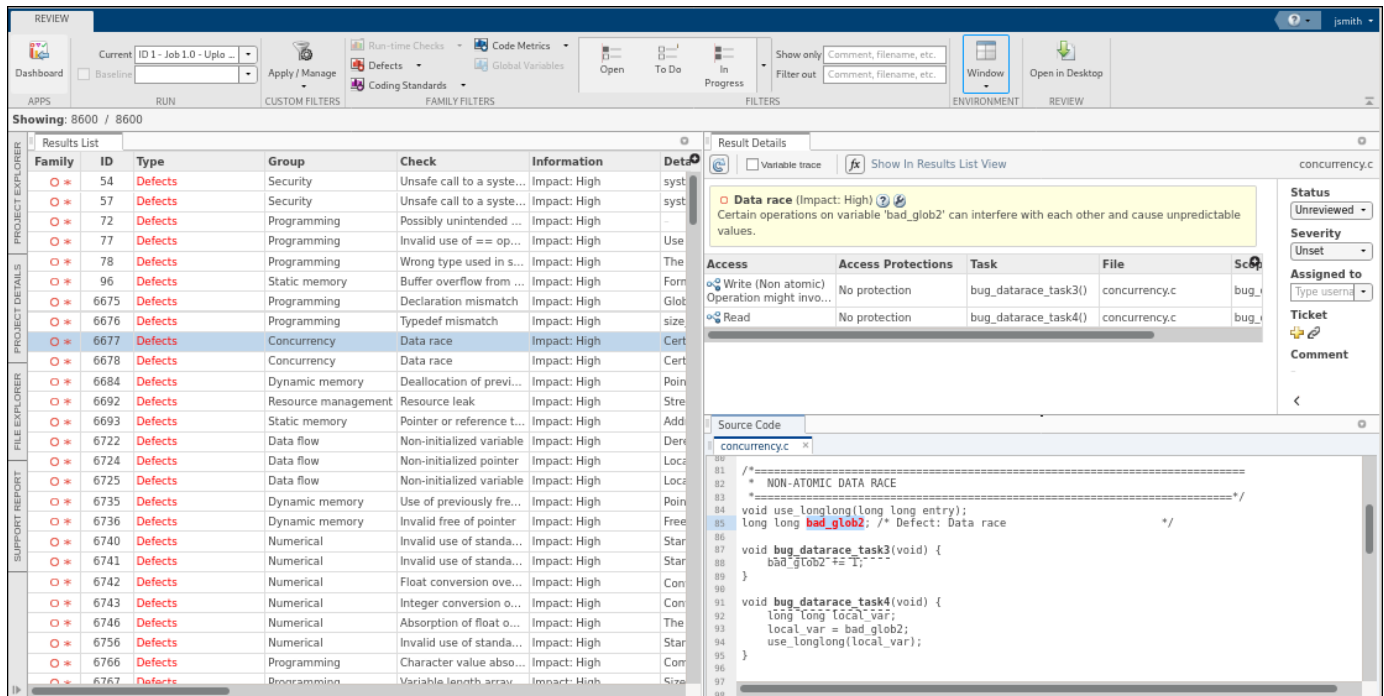
Results Review Layout : Select view to prioritize review of code or results list

In R2021a, the Polyspace Access interface has different layouts to match your results review workflow.

The default **Code Review** layout enables you to focus on the **Source Code** while you investigate issues in your code.



The **Results Review** layout prioritizes the **Results List** and **Result Details** panes as you review and triage findings.



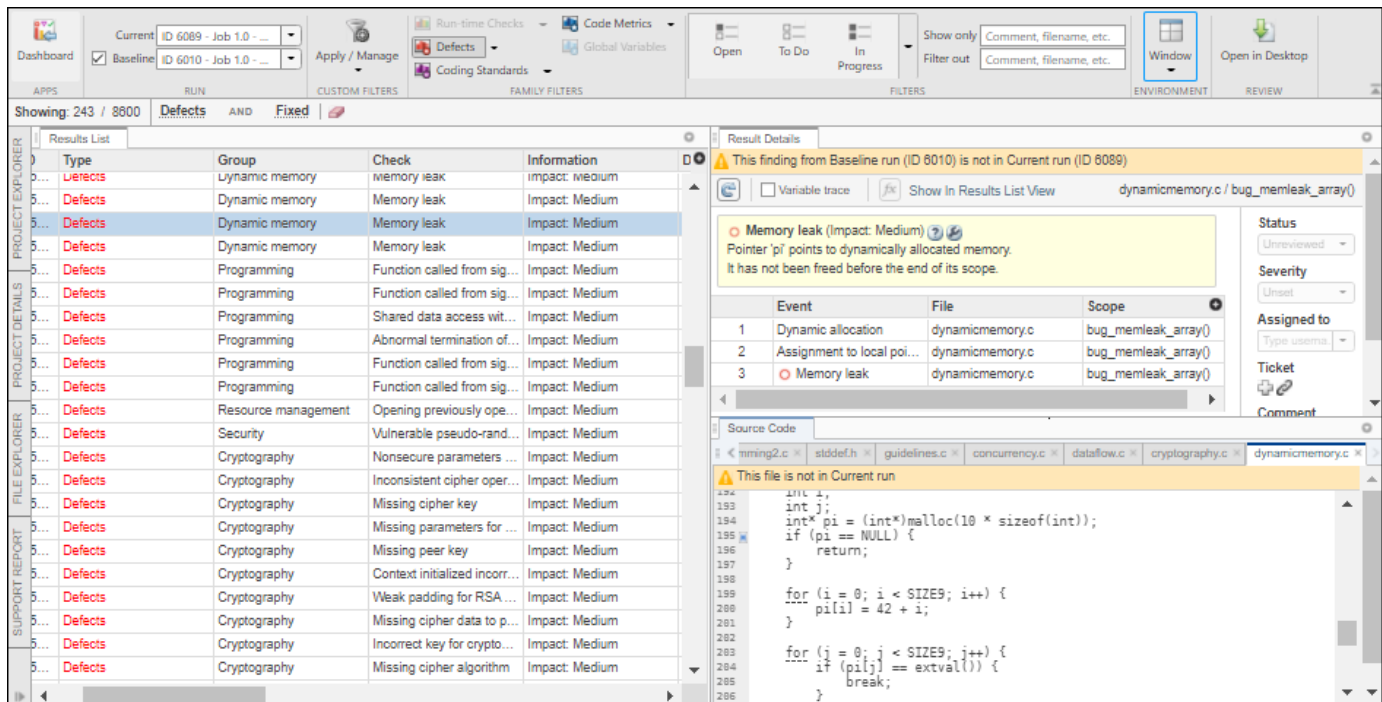
Product: Polyspace Bug Finder Access.

Code Quality Comparison Between Runs: Filter and view information for previous findings fixed in the current run

In R2021a, if you compare two project runs and some of the findings from the **Baseline** run are **Fixed** in the **Current** run, you can filter for and view the source code and result details for these findings. See Compare Analysis Results to Previous Runs (Polyspace Bug Finder Access).

Polyspace Access considers a finding **Fixed** if either:

- You make changes to your code that fix an issue.
- The source code that contains an issue is deleted or is not part of the current analysis.



Previously, you had to open the **Baseline** run as a separate tab to view the source code and result details for **Fixed** findings.

Product: Polyspace Bug Finder Access.

Polyspace Access Installation

License Management : Uploading of results to Polyspace Access no longer requires a license checkout

In R2021a, the upload of analysis results to the Polyspace Access database does not trigger a Polyspace Access license checkout.

If you upload results as part of an automation script, you no longer consume a license when you run the script. Previously, each results upload triggered a license checkout.

Product: Polyspace Bug Finder Access.

User Manager : Enable pagination when requesting large set of users from LDAP server

In R2021a, if you use an LDAP server to retrieve user profiles and authenticate user logins, you can enable pagination to retrieve a large set of users from the LDAP server. See [Authenticate Users from Your Organization LDAP Server \(Polyspace Bug Finder Access\)](#).

Typically, LDAP servers limit the number of entries that they return in a result set. If the number of entries exceed that limit, the result set is truncated. When you enable pagination, the number of results is broken up into smaller sets. You are able to retrieve all entries from the LDAP server when you query a large set of users.

Product: Polyspace Bug Finder Access.

Bug Tracking Tool : Create Jira tickets for Jira projects that use single select custom fields

In R2021a, if you integrate the Jira software bug tracking tool (BTT) with Polyspace Access, you can create Jira tickets for Jira projects that are configured with single select custom fields. See [Configure Jira Software Bug Tracking Tool \(Polyspace Bug Finder Access\)](#).

Previously, Polyspace Access did not support the creation of Jira tickets in projects that used single select custom fields.

Product: Polyspace Bug Finder Access.

Admin Interface : Improved logging for Polyspace Access services

In R2021a, when you view the logs for the Polyspace Access services in the **Admin** user interface, the logs are automatically refreshed. You do not need to reload the page to view new events.

Product: Polyspace Bug Finder Access.

R2020b

Version: 3.3

New Features

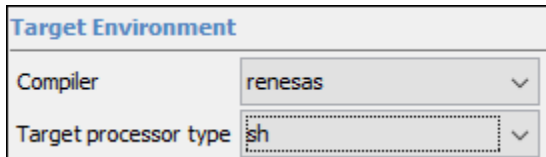
Bug Fixes

Compatibility Considerations

Analysis Setup

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

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.



Target Environment	
Compiler	renesas
Target processor type	sh

See also `Compiler (-compiler renesas)`.

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.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

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

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

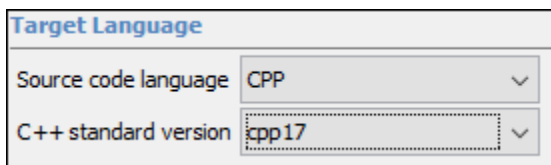
This feature applies to both Polyspace Bug Finder and Bug Finder Server. See also [Check if Polyspace Supports Build Scripts](#).

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

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

C++17 Support: Run Polyspace analysis on code that has C++17 features

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



Target Language	
Source code language	CPP
C++ standard version	cpp17

This feature applies to both Polyspace Bug Finder and Bug Finder Server. See also:

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

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

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

Modifying Checker Behavior: Check for non-initialized buffers when passed by pointer to certain functions

In R2020b, you can indicate that pointer arguments to some functions must point to initialized buffers. By default, the checker `Non-initialized variable` checks a pointer for an initialized buffer only when you dereference the pointer. A function call such as:

```
int var; func(&var);
```

is not flagged for non-initialization because you might initialize the variable `var` in `func`. Starting in R2020b, you can specify a list of functions whose pointer arguments must be checked for initialized buffers.

For more information, see:

- `-code-behavior-specifications`
- Extend Checkers for Initialization to Check Function Arguments Passed by Pointers

Suppose that you consider some function calls as part of the system boundary and you want to make sure that you pass initialized buffers across the boundary. For instance, the Run-Time environment or `Rte_` functions in AUTOSAR allow a software component to communicate with other software components. You might want to ensure that pointer arguments to these functions point to initialized buffers. You can now use Bug Finder to find uninitialized buffers passed through pointers to these functions.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

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

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.

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.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

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

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`.

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.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

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

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`.

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.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

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

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

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.

Product: Polyspace Bug Finder (Desktop).

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

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

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

Product: Polyspace Bug Finder (Desktop).

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

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

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.

Product: Polyspace Bug Finder (Desktop).

Offloading Analysis : Submit Polyspace analysis jobs from CI server to a dedicated analysis cluster

In R2020b, you can set up a continuous integration (CI) system to offload a Polyspace analysis to a dedicated cluster and download the results after analysis. The cluster performing the analysis can be one server or several servers where a head node distributes the jobs to several worker nodes which perform the analysis. In this workflow:

- MATLAB Parallel Server™ is required on all servers involved in distributing jobs or running the analysis.
- You install the Polyspace Server products both on the client and server side, but you do not require licenses for job submission on the client side.

See Offload Polyspace Analysis from Continuous Integration Server to Another Server.

When running static code analysis with Polyspace as part of continuous integration, you might want the analysis to run on a server that is different from the server running your continuous integration (CI) scripts. For instance, you might want to perform the analysis on a server that has more processing power. You can then offload the analysis from your CI server to the other server.

Product: Polyspace Bug Finder Server.

Offloading Analysis : Server-side errors reported back to client side

If you run a Polyspace analysis on a MATLAB Parallel Server cluster, in R2020b, server-side errors are reported back in the client-side log.

The log contains this additional information reported back from the server side:

- Errors that occurred during the server-side analysis.

For instance, if a Polyspace Server license has not been activated, you see a license checkout failure reported from the server side.

- Path to the Polyspace Server instance that runs the analysis.

Information reported from the server side appears in the log between the **Start Diary** and **End Diary** lines.

Starting R2020b, you can troubleshoot server-side errors more easily by using the log reported on the client side.

Product: Polyspace Bug Finder Server.

Changes in analysis options and binaries

XML syntax with option `-code-behavior-specifications` changed

Warns

The option `-code-behavior-specifications` takes an XML file as argument. You can use this XML file to specify whether a certain function must be subjected to special checks. For instance, you can specify that a function must not be used altogether.

In R2020b, the XML syntax changed slightly. To associate the behavior FORBIDDEN_FUNC with a function *funcName*, instead of the syntax:

```
<function name="funcName" behavior="FORBIDDEN_FUNC">
```

Use the syntax:

```
<function name="funcName">  
  <behavior name="FORBIDDEN_FUNC">  
</function>
```

See also `-code-behavior-specifications`.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

Analysis Results

AUTOSAR C++14 Support: Check for 308 AUTOSAR C++14 rules including 61 new rules in R2020b

In R2020b, you can look for violations of these AUTOSAR C++14 rules in addition to previously supported rules.

AUTOSAR C++14 Rule	Description	Polyspace Checker
A0-1-1	A project shall not contain instances of non-volatile variables being given values that are not subsequently used.	AUTOSAR C++14 Rule A0-1-1
A0-1-3	Every function defined in an anonymous namespace, or static function with internal linkage, or private member function shall be used.	AUTOSAR C++14 Rule A0-1-3
A2-7-2	Sections of code shall not be "commented out".	AUTOSAR C++14 Rule A2-7-2
A2-10-4	The identifier name of a non-member object with static storage duration or static function shall not be reused within a namespace.	AUTOSAR C++14 Rule A2-10-4
A2-10-5	An identifier name of a function with static storage duration or a non-member object with external or internal linkage should not be reused.	AUTOSAR C++14 Rule A2-10-5
A3-1-5	A function definition shall only be placed in a class definition if (1) the function is intended to be inlined (2) it is a member function template (3) it is a member function of a class template.	AUTOSAR C++14 Rule A3-1-5
A3-1-6	Trivial accessor and mutator functions should be inlined.	AUTOSAR C++14 Rule A3-1-6
A3-8-1	An object shall not be accessed outside of its lifetime.	AUTOSAR C++14 Rule A3-8-1
A5-1-6	Return type of a non-void return type lambda expression should be explicitly specified.	AUTOSAR C++14 Rule A5-1-6

AUTOSAR C++14 Rule	Description	Polyspace Checker
A5-1-8	Lambda expressions should not be defined inside another lambda expression.	AUTOSAR C++14 Rule A5-1-8
A5-1-9	Identical unnamed lambda expressions shall be replaced with a named function or a named lambda expression.	AUTOSAR C++14 Rule A5-1-9
A5-2-1	dynamic_cast should not be used.	AUTOSAR C++14 Rule A5-2-1
A5-3-1	Evaluation of the operand to the typeid operator shall not contain side effects.	AUTOSAR C++14 Rule A5-3-1
A5-3-2	Null pointers shall not be dereferenced.	AUTOSAR C++14 Rule A5-3-2
A5-10-1	A pointer to member virtual function shall only be tested for equality with null-pointer-constant.	AUTOSAR C++14 Rule A5-10-1
A6-2-1	Move and copy assignment operators shall either move or respectively copy base classes and data members of a class, without any side effects.	AUTOSAR C++14 Rule A6-2-1
A6-2-2	Expression statements shall not be explicit calls to constructors of temporary objects only.	AUTOSAR C++14 Rule A6-2-2
A6-5-3	Do statements should not be used.	AUTOSAR C++14 Rule A6-5-3
A7-1-1	Constexpr or const specifiers shall be used for immutable data declaration.	AUTOSAR C++14 Rule A7-1-1
A7-1-2	The constexpr specifier shall be used for values that can be determined at compile time.	AUTOSAR C++14 Rule A7-1-2
A7-1-5	The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax.	AUTOSAR C++14 Rule A7-1-5

AUTOSAR C++14 Rule	Description	Polyspace Checker
A7-6-1	Functions declared with the [[noreturn]] attribute shall not return.	AUTOSAR C++14 Rule A7-6-1
A8-4-4	Multiple output values from a function should be returned as a struct or tuple.	AUTOSAR C++14 Rule A8-4-4
A8-4-14	Interfaces shall be precisely and strongly typed.	AUTOSAR C++14 Rule A8-4-14
A11-0-1	A non-POD type should be defined as class.	AUTOSAR C++14 Rule A11-0-1
A12-0-2	Bitwise operations and operations that assume data representation in memory shall not be performed on objects.	AUTOSAR C++14 Rule A12-0-2
A12-1-2	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.	AUTOSAR C++14 Rule A12-1-2
A12-1-6	Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors.	AUTOSAR C++14 Rule A12-1-6
A12-4-2	If a public destructor of a class is non-virtual, then the class should be declared final.	AUTOSAR C++14 Rule A12-4-2
A12-8-4	Move constructor shall not initialize its class members and base classes using copy semantics.	AUTOSAR C++14 Rule A12-8-4
A12-8-7	Assignment operators should be declared with the ref-qualifier &.	AUTOSAR C++14 Rule A12-8-7
A13-5-5	Comparison operators shall be non-member functions with identical parameter types and noexcept.	AUTOSAR C++14 Rule A13-5-5
A14-5-2	Class members that are not dependent on template class parameters should be defined in a separate base class.	AUTOSAR C++14 Rule A14-5-2

AUTOSAR C++14 Rule	Description	Polyspace Checker
A14-5-3	A non-member generic operator shall only be declared in a namespace that does not contain class (struct) type, enum type or union type declarations.	AUTOSAR C++14 Rule A14-5-3
A15-1-1	Only instances of types derived from <code>std::exception</code> should be thrown.	AUTOSAR C++14 Rule A15-1-1
A15-1-3	All thrown exceptions should be unique.	AUTOSAR C++14 Rule A15-1-3
A15-2-1	Constructors that are not <code>noexcept</code> shall not be invoked before program startup.	AUTOSAR C++14 Rule A15-2-1
A15-3-3	Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, <code>std::exception</code> and all otherwise unhandled exceptions.	AUTOSAR C++14 Rule A15-3-3
A15-3-4	Catch-all (ellipsis and <code>std::exception</code>) handlers shall be used only in (a) main, (b) task main functions, (c) in functions that are supposed to isolate independent components and (d) when calling third-party code that uses exceptions not according to AUTOSAR C++14 guidelines.	AUTOSAR C++14 Rule A15-3-4
A15-4-2	If a function is declared to be <code>noexcept</code> , <code>noexcept(true)</code> or <code>noexcept(<true condition>)</code> , then it shall not exit with an exception.	AUTOSAR C++14 Rule A15-4-2
A15-4-3	Function's <code>noexcept</code> specification shall be either identical or more restrictive across all translation units and all overriders.	AUTOSAR C++14 Rule A15-4-3

AUTOSAR C++14 Rule	Description	Polyspace Checker
A15-5-1	All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate.	AUTOSAR C++14 Rule A15-5-1
A18-5-9	Custom implementations of dynamic memory allocation and deallocation functions shall meet the semantic requirements specified in the corresponding "Required behaviour" clause from the C++ Standard.	AUTOSAR C++14 Rule A18-5-9
A18-5-10	Placement new shall be used only with properly aligned pointers to sufficient storage capacity.	AUTOSAR C++14 Rule A18-5-10
A18-5-11	"operator new" and "operator delete" shall be defined together.	AUTOSAR C++14 Rule A18-5-11
A18-9-2	Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference.	AUTOSAR C++14 Rule A18-9-2
A18-9-4	An argument to std::forward shall not be subsequently used.	AUTOSAR C++14 Rule A18-9-4
A20-8-2	A std::unique_ptr shall be used to represent exclusive ownership.	AUTOSAR C++14 Rule A20-8-2
A20-8-3	A std::shared_ptr shall be used to represent shared ownership.	AUTOSAR C++14 Rule A20-8-3
A20-8-5	std::make_unique shall be used to construct objects owned by std::unique_ptr.	AUTOSAR C++14 Rule A20-8-5
A20-8-6	std::make_shared shall be used to construct objects owned by std::shared_ptr.	AUTOSAR C++14 Rule A20-8-6
A26-5-2	Random number engines shall not be default-initialized.	AUTOSAR C++14 Rule A26-5-2

AUTOSAR C++14 Rule	Description	Polyspace Checker
A27-0-2	A C-style string shall guarantee sufficient space for data and the null terminator.	AUTOSAR C++14 Rule A27-0-2
A27-0-3	Alternate input and output operations on a file stream shall not be used without an intervening flush or positioning call.	AUTOSAR C++14 Rule A27-0-3
M0-1-4	A project shall not contain non-volatile POD variables having only one use.	AUTOSAR C++14 Rule M0-1-4
M0-3-2	If a function generates error information, then that error information shall be tested.	AUTOSAR C++14 Rule M0-3-2
M7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	AUTOSAR C++14 Rule M7-5-2
M9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit.	AUTOSAR C++14 Rule M9-6-4
M15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	AUTOSAR C++14 Rule M15-1-1
M15-3-1	Exceptions shall be raised only after start-up and before termination.	AUTOSAR C++14 Rule M15-3-1
M15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	AUTOSAR C++14 Rule M15-3-4

See also AUTOSAR C++14 Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

CERT C Support: Check for missing const-qualification and use of hardcoded numbers

In R2020b, you can look for violations of these CERT C recommendations in addition to previously supported rules.

CERT C Rule	Description	Polyspace Checker
DCL00-C	Const-qualify immutable objects	CERT C: Rec. DCL00-C

See also CERT C Rules and Recommendations.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

CERT C++ Support: Check for exception handling issues, memory management problems, and other rule violations

In R2020b, you can look for violations of these CERT C++ rules in addition to previously supported rules.

CERT C++ Rule	Description	Polyspace Checker
ERR58-CPP	Handle all exceptions thrown before main() begins executing	CERT C++: ERR58-CPP
MEM54-CPP	Provide placement new with properly aligned pointers to sufficient storage capacity	CERT C++: MEM54-CPP
MEM55-CPP	Honor replacement dynamic storage management requirements	CERT C++: MEM55-CPP
MSC53-CPP	Do not return from a function declared [[noreturn]]	CERT C++: MSC53-CPP
ERR55-CPP	Honor exception specifications	CERT C++: ERR55-CPP

See also CERT C++ Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

MISRA C++:2008 Support: Check for commented out code, variables used once, exception handling issues, and other rule violations

In R2020b, you can look for violations of these MISRA C++:2008 rules in addition to previously supported rules.

MISRA C++:2008 Rule	Description	Polyspace Checker
0-1-4	A project shall not contain non-volatile POD variables having only one use.	MISRA C++:2008 Rule 0-1-4
0-3-2	If a function generates error information, then that error information shall be tested.	MISRA C++:2008 Rule 0-3-2

MISRA C++:2008 Rule	Description	Polyspace Checker
2-7-2	Sections of code should not be "commented out" using C-style comments.	MISRA C++:2008 Rule 2-7-2
2-7-3	Sections of code should not be "commented out" using C++-style comments.	MISRA C++:2008 Rule 2-7-3
14-5-1	A non-member generic function shall only be declared in a namespace that is not an associated namespace.	MISRA C++:2008 Rule 14-5-1
15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	MISRA C++:2008 Rule 15-1-1
15-3-1	Exceptions shall be raised only after start-up and before termination of the program.	MISRA C++:2008 Rule 15-3-1
15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	MISRA C++:2008 Rule 15-3-4

See also MISRA C++:2008 Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

JSF AV C++ Support: Check for commented out code and methods that can be inlined

In R2020b, you can check for these JSF AV C++ rules in addition to previously supported rules.

Rule	Description
122	Trivial accessor and mutator functions should be inlined.
127	Code that is not used (commented out) shall be deleted.

See also JSF AV C++ Coding Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

MISRA C Support: Check for commented out code

In R2020b, you can look for violations of these MISRA C rules and directives in addition to previously supported rules and directives.

MISRA C Rule	Description	Polyspace Checker
MISRA C:2004 Rule 2.4	Sections of code should not be "commented out".	MISRA C:2004 Rule 2.4 See also MISRA C :2004 and MISRA AC AGC Coding Rules.
MISRA C:2012 Dir 4.4	Sections of code should not be "commented out".	MISRA C:2012 Dir 4.4

See also MISRA C :2012 Directives and Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

New Bug Finder Defect Checkers: Check for post-C++11 defects such as problematic move operations, missing constexpr, and noexcept violations

In R2020b, you can check for these new types of defects.

Defect	Description
A move operation may throw	Throwing move operations might result in STL containers using the corresponding copy operations
Const <code>std::move</code> input may cause a more expensive object copy	Const <code>std::move</code> input cannot be moved and results in more expensive copy operation
Data race on adjacent bit fields	Multiple threads perform unprotected operations on adjacent bit fields of a shared data structure
Expensive <code>std::string::c_str()</code> use in a <code>std::string</code> operation	An <code>std::string</code> operation uses the output of an <code>std::string::c_str</code> method, resulting in inefficient code
Expensive constant <code>std::string</code> construction	A const string object is constructed from constant data resulting in inefficient code
Expensive copy in a range-based for loop iteration	The loop variable of a range-based for loop is copied from the range elements instead of being referenced resulting in inefficient code
Expensive pass by value	Functions pass large parameters by value instead of by reference
Expensive return by value	Functions return large output by value instead of by reference
Incorrect value forwarding	Forwarded object might be modified unexpectedly
Missing <code>constexpr</code> specifier	<code>constexpr</code> specifier can be used on expression for compile-time evaluation

Defect	Description
Noexcept function exits with exception	Functions specified as noexcept, noexcept(true) or noexcept(<true condition>) exit with an exception, which causes abnormal termination of program execution, leading to resource leak and security vulnerability
std::move called on an unmovable type	Result of std::move is not movable
Throw argument raises unexpected exception	The argument expression in a throw statement raises unexpected exceptions, leading to resource leaks and security vulnerabilities

See the full list of defect checkers in Defects.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Changes to coding standards checking

In R2020b, coding rules checking has improved across various coding standards:

- The Polyspace checkers for AUTOSAR C++14 now follow AUTOSAR C++14 release 18-10 (October 2018).
- You can check for MISRA C++ and JSF AV C++ rules in the same run. If the issues that you want to detect span MISRA C++ and JSF AV C++, you can enable rules from both standards and detect issues in a single run.

In addition, these changes have been made in the checking of previously supported rules.

AUTOSAR C++14

Rule	Description	Change
AUTOSAR C++14 Rule A2-13-6	Universal character names shall be used only inside character or string literals.	The checker no longer flags universal character names in code deactivated with a preprocessor directive such as <code>#if</code> . You can enter universal character names for non-string uses in deactivated code.
AUTOSAR C++14 Rule A5-1-1	Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.	The checker now flags use of literal values as template parameters.
AUTOSAR C++14 Rule A5-6-1	The right hand operand of the integer division or remainder operators shall not be equal to zero.	The checker now uses a narrower definition of tainted data. The following are no longer considered as tainted data: <ul style="list-style-type: none"> • Inputs to functions that do not have a visible caller • Return values of undefined (stubbed) functions • Global variables external to the unit See Sources of Tainting in a Polyspace Analysis. To revert to the previous definition, use the option <code>-consider-analysis-perimeter-as-trust-boundary</code> .
AUTOSAR C++14 Rule A21-8-1	Arguments to character-handling functions shall be representable as an unsigned char.	The checker now only detects the use of a signed or plain <code>char</code> variable with a negative value as argument to a character-handling function declared in <code>ctype.h</code> , for instance, <code>isalpha()</code> or <code>isdigit()</code> .

Rule	Description	Change
AUTOSAR C++14 Rule M0-1-4	A project shall not contain non-volatile POD variables having only one use.	<ul style="list-style-type: none">• The checker now considers dynamic assignments of a variable, such as <code>int var = foo()</code> as a single use of the variable.• Some objects are designed to be used only once by their semantics. Polyspace does not flag a single use of these objects:<ul style="list-style-type: none">• <code>lock_guard</code>• <code>scoped_lock</code>• <code>shared_lock</code>• <code>unique_lock</code>• <code>thread</code>• <code>future</code>• <code>shared_future</code> <p>If you use nonstandard objects that provide similar functionality as the objects in the preceding list, Polyspace might flag single uses of the nonstandard objects. Justify their single uses by using comments.</p>

CERT C

Rule	Description	Change
CERT C: Rec. PRE01-C	Use parentheses within macros around parameter names	The checker no longer flags uses of the <code>va_arg</code> macro if the macro parameters are not enclosed in parentheses (in accordance with the exception in the CERT C specifications).
CERT C: Rule EXP36-C	Do not cast pointers into more strictly aligned pointer types	The checker now flags: <ul style="list-style-type: none"> • Conversion of <code>void*</code> pointer into pointer to object. • Source buffer misaligned with destination buffer.
CERT C: Rule EXP43-C	Avoid undefined behavior when using restrict-qualified pointers.	The checker now detects situations where you assign a <code>restrict</code> qualified pointer to another <code>restrict</code> qualified pointer such that they both attempt to point to the same object.
CERT C: Rule EXP46-C	Do not use a bitwise operator with a Boolean-like operand	The checker now flags the use of bitwise operators, such as: <ul style="list-style-type: none"> • Bitwise AND (<code>&</code>, <code>&=</code>) • Bitwise OR (<code> </code>, <code> =</code>) • Bitwise XOR (<code>^</code>, <code>^=</code>) • Bitwise NOT (<code>~</code>) with: <ul style="list-style-type: none"> • Boolean type variables • Outputs of relational or equality expressions
CERT C: Rule MEM30-C	Do not access freed memory	The checker now flags attempts to deallocate a previously freed memory block.
CERT C: Rule MEM35-C	Allocate sufficient memory for an object	The checker now flags the use of a pointer type as the argument of the <code>sizeof</code> operator in a <code>malloc</code> statement. Use the type of the object to which the pointer points as the argument of the <code>sizeof</code> operator.
CERT C: Rule MSC39-C	Do not call <code>va_arg()</code> on a <code>va_list</code> that has an indeterminate value.	The checker flags situations where you might be using a <code>va_list</code> that has an indeterminate value.

Rule	Description	Change
CERT C: Rule STR37-C	Arguments to character-handling functions must be representable as an unsigned char	The checker now only detects the use of a signed or plain char variable with a negative value as argument to a character-handling function declared in <code>ctype.h</code> , for instance, <code>isalpha()</code> or <code>isdigit()</code> .

Rule	Description	Change
<p>Coding rules that involve detection of tainted data, including:</p> <ul style="list-style-type: none"> • CERT C: Rec. INT04-C • CERT C: Rec. INT10-C • CERT C: Rule INT31-C • CERT C: Rule INT32-C • CERT C: Rule INT33-C • CERT C: Rule ARR30-C • CERT C: Rule ARR32-C • CERT C: Rule ARR38-C 		<p>The checkers now use a narrower definition of tainted data. The following are no longer considered as tainted data:</p> <ul style="list-style-type: none"> • Inputs to functions that do not have a visible caller • Return values of undefined (stubbed) functions • Global variables external to the unit <p>See Sources of Tainting in a Polyspace Analysis. To revert to the previous definition, use the option <code>-consider-analysis-perimeter-as-trust-boundary</code>.</p>

Rule	Description	Change
• CERT C: Rec. STR02- C		
• CERT C: Rule STR32- C		
• CERT C: Rec. MEM04- C		
• CERT C: Rec. MEM05- C		
• CERT C: Rule MEM35- C		
• CERT C: Rule FI030- C		
• CERT C: Rec. ENV01- C		
• CERT C: Rec. MSC21- C		
• CERT C: Rec. WIN00- C		

CERT C++

Rule	Description	Change
CERT C++: DCL51-CPP	Do not declare or define a reserved identifier	The checker now flags: <ul style="list-style-type: none"> • Macros or identifiers beginning with underscore followed by an uppercase letter. • User-defined literal operators if the operator names do not begin with an underscore (C++11 and later). <p>By convention, these macros, identifiers and operators are reserved for the Standard Library.</p>
CERT C++: EXP36-C	Do not cast pointers into more strictly aligned pointer types	The checker now flags: <ul style="list-style-type: none"> • Conversion of <code>void*</code> pointer into pointer to object. • Source buffer misaligned with destination buffer.
CERT C++: EXP46-C	Do not use a bitwise operator with a Boolean-like operand	The checker now flags the use of bitwise operators, such as: <ul style="list-style-type: none"> • Bitwise AND (<code>&</code>, <code>&=</code>) • Bitwise OR (<code> </code>, <code> =</code>) • Bitwise XOR (<code>^</code>, <code>^=</code>) • Bitwise NOT (<code>~</code>) <p>with:</p> <ul style="list-style-type: none"> • Boolean type variables • Outputs of relational or equality expressions
CERT C++: EXP52-CPP	Do not rely on side effects in unevaluated operands	The checker now flags <code>decltype</code> operations where the operands have side effects.
CERT C++: MEM30-C	Do not access freed memory	The checker now flags attempts to deallocate a previously freed memory block.
CERT C++: MEM35-C	Allocate sufficient memory for an object	The checker now flags the use of a pointer type as the argument of the <code>sizeof</code> operator in a <code>malloc</code> statement. Use the type of the object to which the pointer points as the argument of the <code>sizeof</code> operator.
CERT C++: MSC39-C	Do not call <code>va_arg()</code> on a <code>va_list</code> that has an indeterminate value	The checker flags situations where you might be using a <code>va_list</code> that has an indeterminate value.

Rule	Description	Change
CERT C+ +: STR37-C	Arguments to character-handling functions must be representable as an unsigned char	The checker now only detects the use of a signed or plain char variable with a negative value as argument to a character-handling function declared in <code>ctype.h</code> , for instance, <code>isalpha()</code> or <code>isdigit()</code> .

Rule	Description	Change
<p>Coding rules that involve detection of tainted data, including:</p> <ul style="list-style-type: none"> • CERT C ++: INT31-C • CERT C ++: INT32-C • CERT C ++: INT33-C • CERT C ++: ARR30-C • CERT C ++: ARR38-C • CERT C ++: STR32-C • CERT C ++: MEM35-C • CERT C ++: FI030-C 		<p>The checkers now use a narrower definition of tainted data. The following are no longer considered as tainted data:</p> <ul style="list-style-type: none"> • Inputs to functions that do not have a visible caller • Return values of undefined (stubbed) functions • Global variables external to the unit <p>See Sources of Tainting in a Polyspace Analysis. To revert to the previous definition, use the option <code>-consider-analysis-perimeter-as-trust-boundary</code>.</p>

ISO/IEC TS 17961

Rule	Description	Change
<p>Coding rules that involve checking of tainted data, including:</p> <ul style="list-style-type: none"> • ISO/IEC TS 17961 [usrfmt] • ISO/IEC TS 17961 [taintstrcpy] • ISO/IEC TS 17961 [taintformatio] • ISO/IEC TS 17961 [taintsink] 		<p>The checkers now use a narrower definition of tainted data. The following are no longer considered as tainted data:</p> <ul style="list-style-type: none"> • Inputs to functions that do not have a visible caller • Return values of undefined (stubbed) functions • Global variables external to the unit <p>See Sources of Tainting in a Polyspace Analysis. To revert to the previous definition, use the option <code>-consider-analysis-perimeter-as-trust-boundary</code>.</p>

MISRA C: 2012

Rule	Description	Change
MISRA C:2012 Dir 4.14	The validity of values received from external sources shall be checked.	<p>The checker now use a broader definition of valid data. The following are no longer considered as invalid data:</p> <ul style="list-style-type: none"> • Inputs to functions that do not have a visible caller • Return values of undefined (stubbed) functions • Global variables external to the unit <p>See Sources of Tainting in a Polyspace Analysis. To revert to the previous definition, use the option <code>-consider-analysis-perimeter-as-trust-boundary</code>.</p>
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.	<p>The checker takes into account header files irrespective of whether you suppress headers using the option <code>Do not generate results for (-do-not-generate-results-for)</code>.</p> <p>For instance, the checker raises a violation if the number of macros in C99 code exceeds 4095. The checker now counts macros in header files irrespective of whether you choose to suppress results in headers. The reason is that the header files are included in a translation unit and the translation unit as a whole is subject to MISRA C: 2012 Rule 1.1. Previously, the headers were taken into account only if unsuppressed.</p>

MISRA C++:2008

Rule	Description	Change
MISRA C++:2008 Rule 0-1-4	A project shall not contain non-volatile POD variables having only one use.	<ul style="list-style-type: none"> • The checker now considers dynamic assignments of a variable, such as <code>int var = foo()</code> as a single use of the variable. • Some objects are designed to be used only once by their semantics. Polyspace does not flag a single use of these objects: <ul style="list-style-type: none"> • <code>lock_guard</code> • <code>scoped_lock</code> • <code>shared_lock</code> • <code>unique_lock</code> • <code>thread</code> • <code>future</code> • <code>shared_future</code> <p>If you use nonstandard objects that provide similar functionality as the objects in the preceding list, Polyspace might flag single uses of the nonstandard objects. Justify their single uses by using comments.</p>
MISRA C++:2008 Rule 2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.	The checker no longer flags class member operators in nested scopes. Class member operators in nested scopes do not hide each other.
MISRA C++:2008 Rule 3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.	The checker no longer flags identifiers used only in a range-based <code>for</code> loop but defined outside the loop.
MISRA C++:2008 Rule 14-6-2	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.	The checker no longer flags calls that use an underlying function call operator.

Rule	Description	Change
MISRA C++:2008 Rule 17-0-1	Reserved identifiers, macros and functions in the Standard Library shall not be defined, redefined or undefined.	The checker raises a violation if you define or redefine a macro beginning with an underscore followed by an uppercase letter. These macros are typically reserved for the Standard Library.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Compatibility Considerations

If you checked your code for the preceding rules, you might see a change in the number of violations.

Updated Bug Finder defect checkers

In R2020b, these defect checkers have been updated.

Defect	Description	Update
Tainted Data Defects	Use of tainted and unvalidated data in critical operations	<p>The checkers now use a narrower definition of tainted data. The following are no longer considered as tainted data:</p> <ul style="list-style-type: none"> • Inputs to functions that do not have a visible caller • Return values of undefined (stubbed) functions • Global variables external to the unit <p>See Sources of Tainting in a Polyspace Analysis. To revert to the previous definition, use the option <code>-consider-analysis-perimeter-as-trust-boundary</code>.</p>

Defect	Description	Update
Deterministic random output from constant seed and Predictable random output from predictable seed	Issues with seeding of random number generator functions	The checkers now support random number generator functions from the C++ Standard Library, for instance, <code>std::linear_congruential_engine<>::seed()</code> and <code>std::mersenne_twister_engine<>::seed()</code> .
Large pass-by-value argument	Functions pass large parameters by value instead of by reference	Checker is removed. Use Expensive pass by value and Expensive return by value instead.
<ul style="list-style-type: none"> • Empty destructors may cause unnecessary data copies • <code>std::endl</code> may cause an unnecessary flush 	Issues that impact performance of C++ code	<p>The Impact attribute of these checkers has been changed from High to Low.</p> <p>These checkers do not have a universally high criticality. The checkers are critical only for code that must be optimized for performance.</p>
Inefficient string length computation	Issue that impacts performance of C++ code	<p>The Impact attribute of this checker has been changed from High to Medium.</p> <p>This checker does not have a universally high criticality. The checker is critical only for code that must be optimized for performance and also promotes a good coding style.</p>
Missing return statement	Issues with data flow	This checker flags nonvoid functions that do not return the flow of execution except if the function is specified as <code>[[noreturn]]</code> .

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Compatibility Considerations

If you check your code for the preceding defects, you might see a difference in the number of issues found.

Updated code metrics specifications

In R2020b, these code metrics specifications have been updated.

Code Metric	Update
Number of Called Functions	<p>These metrics now accounts for function calls in a C++ constructor initializer list.</p> <p>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.</p> <pre> class Base { int b; public: Base() { b = 0; }; }; class Derived : public Base { int d; public: Derived() : Base() { d = 0; }; }; </pre>

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

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

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`.

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.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

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

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`.

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.

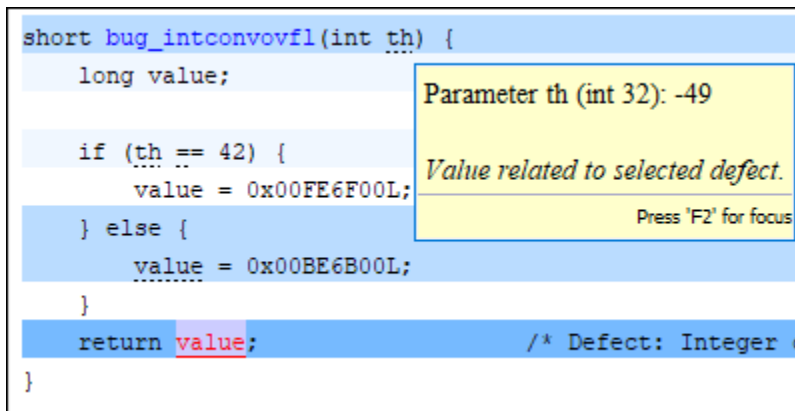
Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Source Code Tooltips: Display information related to only the currently selected defect

In R2020b, Bug Finder tooltips show only information that is necessary to understand the currently selected defect, such as:

- Data types of variables that lead to the current defect.
- One specific value of an input variable that leads to the current defect, if you enable the option Run stricter checks considering all values of system inputs (-checks-using-system-input-values)

In this tooltip, you see that the input parameter is a 32-bit int and the value -49 leads to the currently selected defect.:



```

short bug_intconvovfl(int th) {
    long value;

    if (th == 42) {
        value = 0x00FE6F00L;
    } else {
        value = 0x00BE6B00L;
    }

    return value; /* Defect: Integer c
}
  
```

Previously, tooltips showed range information such as all possible values of a specific variable in the given context. You can still see this range information in Code Prover.

In Bug Finder, tooltips do not appear on any line other than ones related to the current defect. When they appear, they contain only information required to understand the currently selected defect.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Access

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

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

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.

Product: Polyspace Bug Finder (Desktop).

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

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 Bug Finder 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.

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.

Product: Polyspace Bug Finder Access.

Code Quality Improvement Progress : Compare results from current run to previous runs and determine progress in code quality improvement

In R2020b, you can select any two runs of a project in the Polyspace web interface (current and baseline runs) and compare them. You can compare a current run to only older baseline runs.

The screenshot shows the Polyspace Bug Finder interface. The top navigation bar includes 'DASHBOARD', 'Project Overview', 'Defects', 'Code Metrics', 'Custom Rules', 'Polyspace Guidelines', 'Window', 'Open in Desktop', and 'Review'. The main content area is divided into a 'Project Explorer' on the left and a 'Project Overview' on the right. The 'Project Overview' section displays a 'Summary' table comparing 'Baseline Run' and 'Current Run' for the project 'Bug_Finder_Example'.

Comparison	Baseline Run	Current Run
Number of Files	14	14
Number of Lines Without Comm...	5201	5201
Defects - Total	242	-
Defects - Density	36	0
Coding Standards - Total	49	-
Coding Standards - Density	9	0

Below the comparison table is a 'Details' section with a table showing the status of findings:

Name	Resolved	New	Unre
Defects	188	-	
Custom Rules	45	-	
Polyspace Guidelines	4	-	

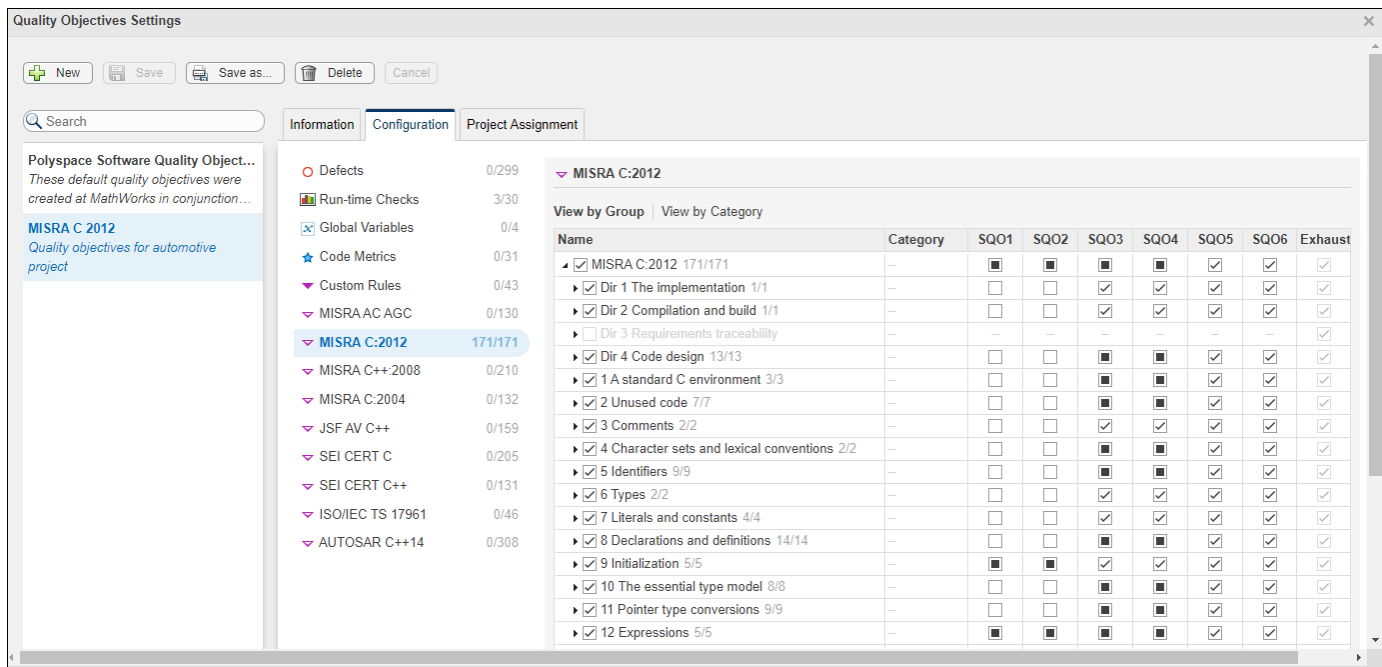
The comparison shows the number of analysis findings that are:

- **Resolved.** Findings from the baseline run no longer found in the current run.
- **New.** Findings in the current run that were not present in the baseline run.
- **Unresolved.** Findings from the baseline run that are still present in the current run.

Product: Polyspace Bug Finder Access.

Code Quality Objectives : Define custom quality objectives definitions and apply them to specific projects

In R2020b, you can create custom quality objectives definitions and apply those definitions to specific projects. For instance, if you want to track the compliance of a project with a coding standard, you can create Quality Objective thresholds for that coding standard and apply them to your project.



To create custom quality objectives definitions, you must be an **Administrator** or **Owner**.

Previously, custom quality objectives applied to all projects.

Product: Polyspace Bug Finder Access.

Source Code Tooltips : Display only information necessary to understand the selected defect

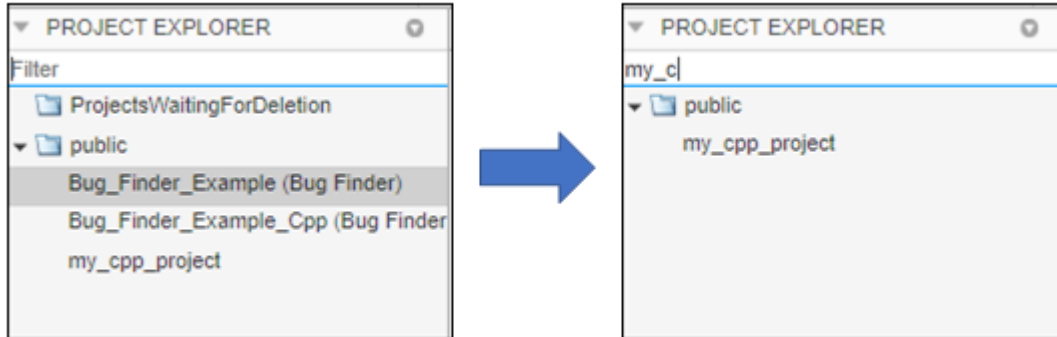
In R2020b, Bug Finder tooltips show only information that is necessary to understand the currently selected defect.

Previously, tooltips showed range information, such as all possible values of a specific variable in the given context. You can still see this range information in Code Prover.

Product: Polyspace Bug Finder Access.

Project Selection : Find a project in the Project Explorer through a text filter

In R2020b, you can use a text filter in the **Project Explorer** to find projects that are not visible in a folder hierarchy. The text filter is not case sensitive.



Product: Polyspace Bug Finder Access.

Functionality being removed: Polyspace Metrics

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 Bug Finder Access](#).

See also [Migrate Results from Polyspace Metrics to Polyspace Access \(Polyspace Bug Finder Access\)](#).

Polyspace Access Installation

Bug Tracking Tool : Integrate with Jira Software Cloud

In R2020b, you can integrate Jira Software Cloud with Polyspace Access. After you configure Polyspace Access, you can create a Jira ticket to track Polyspace findings. The ticket is populated with details of the finding and a link to open that finding in Polyspace Access. See [Configure Issue Tracker \(Polyspace Bug Finder Access\)](#).

Previously, you could integrate Polyspace Access with only self-managed Jira Software.

Product: Polyspace Bug Finder Access.

Cluster Admin Settings: Validate values of settings on demand or on save

In R2020b, the **Cluster Admin** validates the settings that you enter in the **Cluster Settings** when you save those settings. You can also validate the settings before you save by clicking **Validate now** at the bottom of the page.

Product: Polyspace Bug Finder Access.

HTTPS Configuration : Configure services without specifying ports or SSL certificates

In R2020b, if you install Polyspace Access on a single node, the ports of the Polyspace Access services are no longer exposed. You do not need to specify port numbers for the services or to provide SSL private keys and certificates for the HTTPS configuration. See [Configure Polyspace Access for HTTPS \(Polyspace Bug Finder Access\)](#).

Previously, you had to check the availability of the ports for the services, and then you provided a private key and SSL certificate file to enable the HTTPS protocol for Polyspace Access.

Product: Polyspace Bug Finder Access.

Functionality Replaced : Polyspace Access embedded LDAP

The Polyspace Access embedded LDAP is removed in R2020b. To continue using custom login credentials for Polyspace Access, use the **User Manager** internal directory instead. See [Authenticate Users from Internal Directory \(Polyspace Bug Finder Access\)](#).

User Manager
admin ▾

Dashboard
Create

Sign-in ID	Display Name	Email	
admin ADMIN	admin	admin@email.com	✕ ▾
jdoe	John Doe		✕ ▾
jsmith	Jane Smith		✕ ▾
rroll	Richard Roll		✕ ▾

Product: Polyspace Bug Finder Access.

Compatibility Considerations

In the **User Manager** interface, create users to transfer the user names and passwords that you stored in the embedded LDAP LDIF file to the **User Manager** database.

Changes in Polyspace Access docker containers, options, and binaries

In R2020b, the following docker containers, options, and binaries have been renamed:

- The cop-docker-agent binary is now called the admin-docker-agent
- **HTTPS Options**

Previous Option Name	Current Option Name
--https-certificate-file	--ssl-cert-file
--https-private-key-file	--ssl-key-file
--https-trusted-certificates-file	--ssl-ca-file

- **Containers**

Previous Container Name	Current Container Name
polyspace-db	polyspace-access-db-main
polyspace-etl	polyspace-access-etl-main
polyspace-gateway	gateway
polyspace-issuetracker	issuetracker-server-main
polyspace-web-server	polyspace-access-web-server-main

Product: Polyspace Bug Finder Access.

Compatibility Considerations

In your scripts, replace instances of the previous names with the current names. You cannot reuse a settings configuration file (`settings.json`) from a previous release of Polyspace Access with the R2020b software.

R2020a

Version: 3.2

New Features

Bug Fixes

Compatibility Considerations

Analysis Setup

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

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

Target Environment	
Compiler	microchip
Target processor type	pic

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

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.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

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

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.

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.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

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

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)`.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

Modifying Checkers: Create list of functions to prohibit and check for use of functions from the list

In R2020a, you can define a blocklist of functions to forbid from your source code. The Bug Finder checker `Use of a forbidden function` checks if a function from this list appears in your sources.

A function might be blocked for one of these reasons:

- The function can lead to many situations where the behavior is undefined leading to security vulnerabilities, and a more secure function exists.

You can block functions that are not explicitly checked by existing checkers such as `Use of dangerous standard function` or `Use of obsolete standard function`.

- The function is being deprecated as part of a migration, for instance, from C++98 to C++11.

As part of a migration, you can make a list of functions that need to be replaced and use this checker to identify their use.

See also `Flag Deprecated or Unsafe Functions Using Bug Finder Checkers`.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

Simulink Support : Analyze custom C code in C Function blocks

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`.

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.

Product: Polyspace Bug Finder (Desktop).

Jenkins Support: Use sample Jenkins Pipeline script to run Polyspace as part of continuous delivery pipeline

In R2020a, you can start from a template Jenkins Pipeline script to run Polyspace analysis as part of a continuous delivery pipeline.

See `Sample Jenkins Pipeline Scripts for Polyspace Analysis`.

You can make simple replacements to adapt the template to your Polyspace Server and Access installations, and include the script in a new or existing Jenkinsfile to get up and running with Polyspace in Jenkins Pipelines.

Product: Polyspace Bug Finder Server.

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 blocklist of functions to forbid from your source code.

See also `-code-behavior-specifications`.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

Changes in MATLAB functions, options object and properties

`polyspaceBugFinderNodesktop` removed

Errors

Use `polyspaceBugFinder(projectFile, '-nodesktop')` instead of `polyspaceBugFinderNodesktop(projectFile)`.

Product: Polyspace Bug Finder (Desktop).

`pslinksetup` removed

Errors

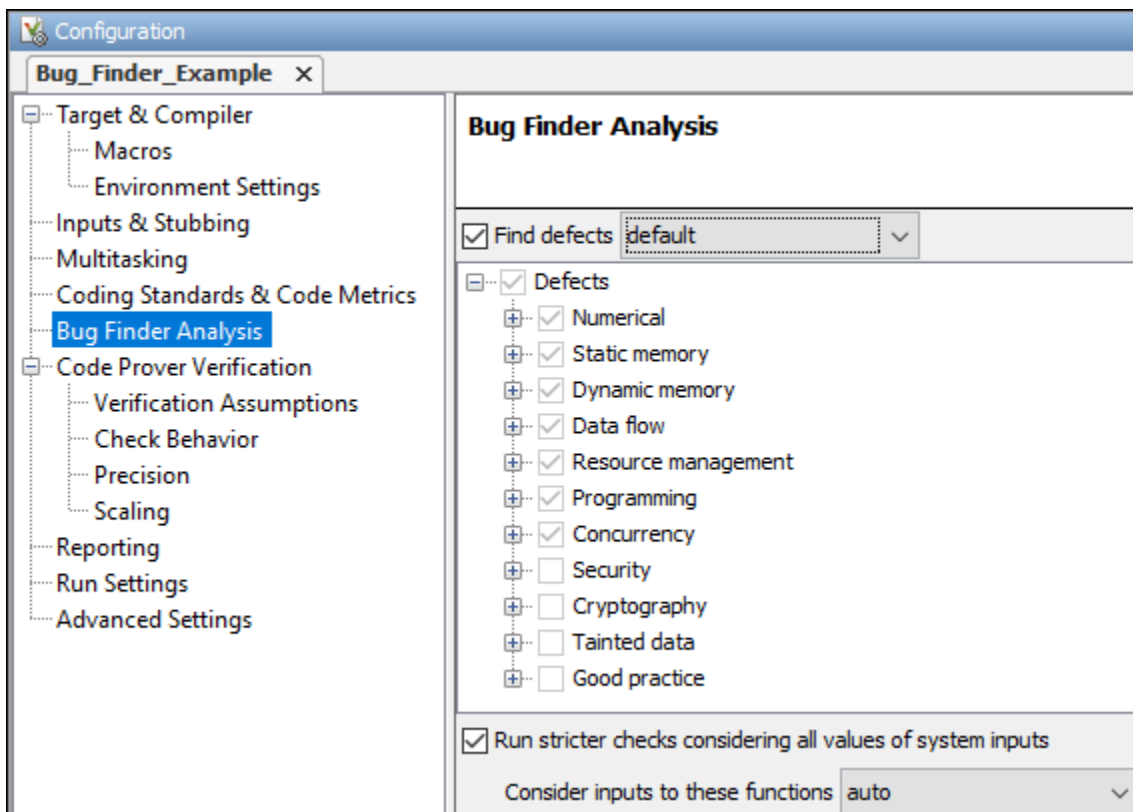
Use `polyspacesetup` instead of `pslinksetup` to integrate between Polyspace and Simulink (in the same release or across releases). See [Integrate Polyspace with MATLAB and Simulink](#).

Product: Polyspace Bug Finder (Desktop).

Analysis Results

Extending Checkers: Run stricter analysis that considers all possible values of system inputs

In R2020a, you can run a stricter Polyspace Bug Finder analysis that checks the robustness of your code against specific values of system inputs. For defects that are detected with the stricter checks, the analysis can also show an example of values that lead to the defect. Use the option Run stricter checks considering all values of system inputs (`-checks-using-system-input-values`) to enable the stricter checks.



For a subset of **Numerical** and **Static memory** defect checkers, the analysis considers all possible values of:

- Global variables
- Reads of volatile variables
- Returns of stubbed functions
- Inputs to the functions you specify with the option Consider inputs to these functions (`-system-inputs-from`)

See also Extend Bug Finder Checkers to Find Defects from Specific System Input Values.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

AUTOSAR C++14 Support: Check for 37 new rules related to lexical conventions, standard conversions, declarations, derived classes, special member functions, overloading and other groups

In R2020a, you can look for violations of these AUTOSAR C++14 rules in addition to previously supported rules.

AUTOSAR C++14 Rule	Description	Polyspace Checker
A0-1-5	There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it.	AUTOSAR C++14 Rule A0-1-5
A2-3-1	Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code.	AUTOSAR C++14 Rule A2-3-1
A2-7-1	The character \ shall not occur as a last character of a C++ comment.	AUTOSAR C++14 Rule A2-7-1
A2-10-1	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.	AUTOSAR C++14 Rule A2-10-1
A2-10-6	A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope.	AUTOSAR C++14 Rule A2-10-6
A2-13-4	String literals shall not be assigned to non-constant pointers.	AUTOSAR C++14 Rule A2-13-4
A2-13-6	Universal character names shall be used only inside character or string literals.	AUTOSAR C++14 Rule A2-13-6
A3-3-2	Static and thread-local objects shall be constant-initialized.	AUTOSAR C++14 Rule A3-3-2
A4-5-1	Expressions with type enum or enum class shall not be used as operands to built-in and overloaded operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.	AUTOSAR C++14 Rule A4-5-1

AUTOSAR C++14 Rule	Description	Polyspace Checker
A4-10-1	Only nullptr literal shall be used as the null-pointer-constraint.	AUTOSAR C++14 Rule A4-10-1
A7-1-3	CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name.	AUTOSAR C++14 Rule A7-1-3
A7-1-8	A non-type specifier shall be placed before a type specifier in a declaration.	AUTOSAR C++14 Rule A7-1-8
A7-4-1	The asm declaration shall not be used.	AUTOSAR C++14 Rule A7-4-1
A8-2-1	When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters.	AUTOSAR C++14 Rule A8-2-1
A8-5-3	A variable of type auto shall not be initialized using {} or ={} braced-initialization.	AUTOSAR C++14 Rule A8-5-3
A10-1-1	Class shall not be derived from more than one base class which is not an interface class.	AUTOSAR C++14 Rule A10-1-1
A10-3-1	Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final.	AUTOSAR C++14 Rule A10-3-1
A10-3-2	Each overriding virtual function shall be declared with the override or final specifier.	AUTOSAR C++14 Rule A10-3-2
A10-3-3	Virtual functions shall not be introduced in a final class.	AUTOSAR C++14 Rule A10-3-3
A10-3-5	A user-defined assignment operator shall not be virtual.	AUTOSAR C++14 Rule A10-3-5
A11-0-2	A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class.	AUTOSAR C++14 Rule A11-0-2

AUTOSAR C++14 Rule	Description	Polyspace Checker
A12-0-1	If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well.	AUTOSAR C++14 Rule A12-0-1
A12-4-1	Destructor of a base class shall be public virtual, public override or protected non-virtual.	AUTOSAR C++14 Rule A12-4-1
A12-8-6	Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class.	AUTOSAR C++14 Rule A12-8-6
A13-1-2	User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters.	AUTOSAR C++14 Rule A13-1-2
A13-2-3	A relational operator shall return a boolean value.	AUTOSAR C++14 Rule A13-2-3
A13-5-1	If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented.	AUTOSAR C++14 Rule A13-5-1
A13-5-2	All user-defined conversion operators shall be defined explicit.	AUTOSAR C++14 Rule A13-5-2
A14-7-2	Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared.	AUTOSAR C++14 Rule A14-7-2
A14-8-2	Explicit specializations of function templates shall not be used.	AUTOSAR C++14 Rule A14-8-2
A16-6-1	#error directive shall not be used.	AUTOSAR C++14 Rule A16-6-1
A17-6-1	Non-standard entities shall not be added to standard namespaces.	AUTOSAR C++14 Rule A17-6-1
A18-1-3	The std::auto_ptr shall not be used.	AUTOSAR C++14 Rule A18-1-3

AUTOSAR C++14 Rule	Description	Polyspace Checker
A18-1-6	All std::hash specializations for user-defined types shall have a noexcept function call operator.	AUTOSAR C++14 Rule A18-1-6
A18-5-2	Operators new and delete shall not be called explicitly.	AUTOSAR C++14 Rule A18-5-2
A18-9-3	The std::move shall not be used on objects declared const or const&.	AUTOSAR C++14 Rule A18-9-3
A23-0-1	An iterator shall not be implicitly converted to const_iterator.	AUTOSAR C++14 Rule A23-0-1

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

CERT C Support: Check for CERT C rules related to threads and hardcoded sensitive data, and recommendations related to macros and code formatting

In R2020a, you can look for violations of these CERT C rules and recommendations in addition to the previously supported ones. With these new rules, all CERT C rules can be checked with Bug Finder.

Rules

CERT C Rule	Description	Polyspace Checker
CON34-C	Declare objects shared between threads with appropriate storage durations	CERT C: Rule CON34-C
CON38-C	Preserve thread safety and liveness when using condition variables	CERT C: Rule CON38-C
MSC41-C	Never hard code sensitive information	CERT C: Rule MSC41-C
POS47-C	Do not use threads that can be canceled asynchronously	CERT C: Rule POS47-C
POS50-C	Declare objects shared between POSIX threads with appropriate storage durations	CERT C: Rule POS50-C
POS53-C	Do not use more than one mutex for concurrent waiting operations on a condition variable	CERT C: Rule POS53-C

Recommendations

CERT C Recommendation	Description	Polyspace Checker
PRE10-C	Wrap multistatement macros in a do-while loop	CERT C: Rec. PRE10-C
PRE11-C	Do not conclude macro definitions with a semicolon	CERT C: Rec. PRE11-C
EXP15-C	Do not place a semicolon on the same line as an if, for, or while statement	CERT C: Rec. EXP15-C

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

CERT C++ Support: Check for CERT C++ rule related to hard coded sensitive data, order of initialization in constructor and other issues

In R2020a, you can look for violations of these CERT C++ rules in addition to previously supported rules.

CERT C++ Rule	Description	Polyspace Checker
DCL58-CPP	Do not modify the standard namespaces	CERT C++: DCL58-CPP
MSC41-C	Never hard code sensitive information	CERT C++: MSC41-C
OOP53-CPP	Write constructor member initializers in the canonical order	CERT C++: OOP53-CPP

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

CWE Support: Check for CWE rule related to incorrect block delimitation

In R2020a, you can check for violation of this CWE rule in addition to previously supported rules.

CWE Rule	Description	Polyspace Checkers
483	Incorrect block delimitation	<ul style="list-style-type: none"> Incorrectly indented statement Semicolon on same line as if, for or while statement

For the full mapping between CWE rules and Polyspace Bug Finder defect checkers, see CWE Coding Standard and Polyspace Results.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

New Bug Finder Defect Checkers: Check for possible performance bottlenecks, hardcoded sensitive data and other issues

In R2020a, you can check for new types of defects.

A new category of C++-specific checkers checks for constructs that might cause performance issues and suggests more efficient alternatives. Other checkers include security checkers for hard coded sensitive data, good practice checkers for issues such as ill-formed macros and concurrency checkers for issues such as asynchronously cancellable threads.

Performance Checkers

Defect	Description
Const parameter values may cause unnecessary data copies	Const parameter values prevent a move operation resulting in a more performance-intensive copy operation
Const return values may cause unnecessary data copies	Const return values prevent a move operation resulting in a more performance-intensive copy operation
Empty destructors may cause unnecessary data copies	User-defined empty destructors prevent autogeneration of move constructors and move assignment operators
Inefficient string length computation	String length calculated by using string length functions on return from <code>std::basic_string::c_str()</code> instead of using <code>std::basic_string::length()</code>
<code>std::endl</code> may cause an unnecessary flush	<code>std::endl</code> is used instead of more efficient alternatives such as <code>\n</code>

Other Checkers

Defect	Description
Asynchronously cancellable thread	Calling thread might be cancelled in an unsafe state
Automatic or thread local variable escaping from a thread	Variable is passed from one thread to another without ensuring that variable stays alive for duration of both threads
Hard-coded sensitive data	Sensitive data is exposed in code, for instance as string literals
Incorrectly indented statement	Statement indentation incorrectly makes it appear as part of a block
Macro terminated with a semicolon	Macro definition ends with a semicolon
Macro with multiple statements	Macro consists of multiple semicolon-terminated statements, enclosed in braces or not
Missing final step after hashing update operation	Hash is incomplete or non-secure
Missing private key for X.509 certificate	Missing key might result in run-time error or non-secure encryption
Move operation on const object	<code>std::move</code> function is called with object declared <code>const</code> or <code>const&</code>
Multiple mutexes used with same conditional variable	Threads using different mutexes when concurrently waiting on the same condition variable is undefined behavior
Multiple threads waiting on same condition variable	Using <code>cond_signal</code> to wake up one of the threads might result in indefinite blocking
No data added into context	Performing hash operation on empty context might cause run-time errors
Possibly inappropriate data type for switch expression	Switch expression has a data type other than <code>char</code> , <code>short</code> , <code>int</code> or <code>enum</code>
Semicolon on the same line as an <code>if</code> , <code>for</code> or <code>while</code> statement	Semicolon on same line results in empty body of <code>if</code> , <code>for</code> or <code>while</code> statement
Server certificate common name not checked	Attacker might use valid certificate to impersonate trusted host
TLS/SSL connection method not set	Program cannot determine whether to call client or server routines
TLS/SSL connection method set incorrectly	Program calls functions that do not match role set by connection method
Unmodified variable not <code>const</code> -qualified	Variable is not <code>const</code> -qualified but no modification anywhere in the program
Use of a forbidden function	Use of function that appears in a blacklist of forbidden functions
Redundant expression in <code>sizeof</code> operand	<code>sizeof</code> operand contains expression that is not evaluated

Defect	Description
X.509 peer certificate not checked	Connection might be vulnerable to man-in-the-middle attacks

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Changes to coding standards checking

In R2020a, the following changes have been made in checking of previously supported rules.

Rule	Description	Change
Some MISRA C: 2012 rules that were previously specific to a C standard	<ul style="list-style-type: none"> C90-specific rules: 8.1, 17.3 C99-specific rules: 3.2, 8.10, 21.11, 21.12 	These rules are now checked irrespective of the C standard. The reason is that the constructs flagged by these rules can be found in code using either standard, possibly with language extensions.
MISRA C:2012 Rule 8.4	A compatible declaration shall be visible when an object with an external linkage is defined.	<ul style="list-style-type: none"> The checker now flags tentative definitions (variables declared without an extern specifier and not explicitly defined), for instance: <pre>uint8_t var;</pre> The checker does not raise a violation on the main function.
MISRA C++:2008 Rule 0-1-3, AUTOSAR C++14 Rule M0-1-3	A project shall not contain unused variables.	<p>The checker does not flag as unused constants used in template instantiations, such as the variable <code>size</code> here:</p> <pre>const std::uint8_t size = 2; std::array<uint8_t, size> arr = {0,1};</pre>
MISRA C++:2008 Rule 2-10-5	The identifier name of a non-member object or function with static duration should not be reused.	The checker does not flag situations where a variable defined in a header file appears to be reused because the header file is included more than once, possibly along different inclusion paths.

Rule	Description	Change
MISRA C++:2008 Rule 18-4-1	Dynamic heap memory allocation shall not be used.	The checker now flags uses of the <code>alloca</code> function. Though memory leak cannot happen with the <code>alloca</code> function, other issues associated with dynamic memory allocation, such as memory exhaustion and nondeterministic behavior, can still occur.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Compatibility Considerations

If you checked for the rules mentioned above, you might see a change in the number of violations.

Updated Bug Finder defect checkers

In R2020a, these defect checkers have been updated.

Defect	Description	Update
Copy constructor not called in initialization list	Copy constructor does not call copy constructors of some data members	The checker no longer flags copy constructors in templates. In template declarations, the member data types are not known and it is not clear which constructors need to be called.
Dead code	Code does not execute	If a <code>try</code> block contains a <code>return</code> statement, the checker no longer flags the corresponding <code>catch</code> block as dead code. A <code>return</code> statement involves a copy and copy constructors that are called might throw exceptions, resulting in the <code>catch</code> block being executed.
Missing <code>explicit</code> keyword	One-parameter constructor missing the <code>explicit</code> specifier	The checker has been updated to include user-defined conversion operators declared or defined in-class without the <code>explicit</code> keyword.

Defect	Description	Update
Missing return statement	Function does not return value though the return type is not void	The checker respects the option <code>-termination-functions</code> . If Bug Finder incorrectly flags a missing return statement on a path where a process termination function exists, you can make the analysis aware of the process termination function using this option.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Compatibility Considerations

If you check for the defects mentioned above, you can see a difference in the number of issues found.

Reviewing Results

Inputs Causing Defect: See example input values for numerical defects found with stricter analysis

In R2020a, if you enable Run stricter checks considering all values of system inputs (-checks-using-system-input-values), you can see an example of values that lead to the detected defect in the **Results Details**.

○ **Integer division by zero** (Impact: High) ? ⓘ
 Divisor is 0.
Result includes example values that lead to the defect.

	Event	File	Scope	Line
1	Function called by external code with input 's' Possible input value causing defect: {a=0, b=-2}	test.c	func()	9
2	Entering function 'func'	test.c	func()	9
3	Assignment to local variable 'j'	test.c	func()	12
4	Assignment to parameter 's'	test.c	func()	13
5	Assignment to local variable 'j'	test.c	func()	14
6	○ Integer division by zero	test.c	func()	16

Configuration
Result Details

Source

test.c x

```

1  #include <stdio.h>
2
3  typedef struct {
4      int a;
5      int b;
6
7  } S2;
8
9  int func(S2 s)
10 {
11     int i;
12     int j = 1;
13     s.a += 3;
14     j = j - s.b;
15
16     i = 1024 / (j - s.a);
17
18     return i;
19 }
```

You can use the example values to fix defects in your code that are due to specific system input values.

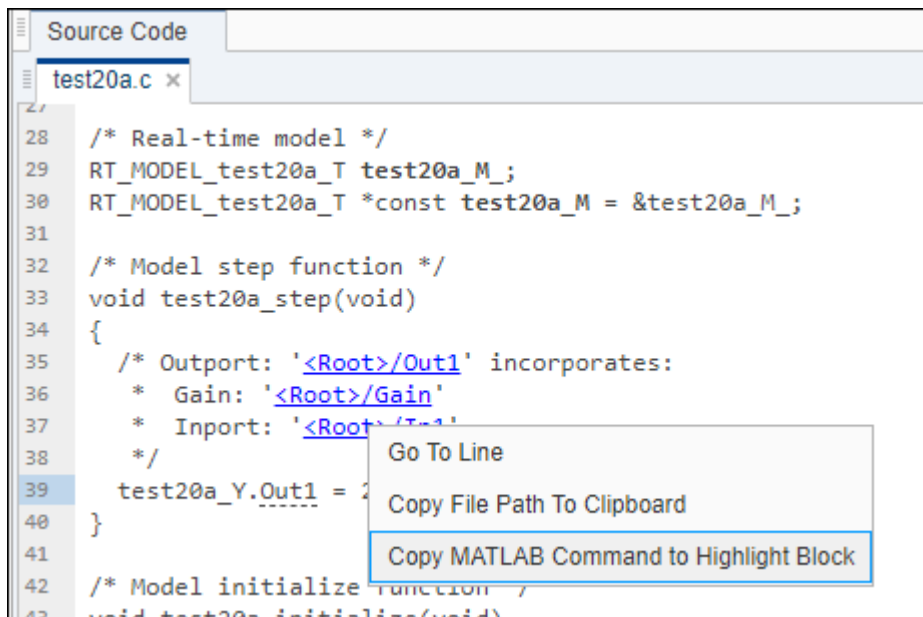
Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Access

Simulink Support: Navigate from generated code in Polyspace Access to blocks in model

In R2020a, if you run Polyspace on generated code in Simulink and upload the results to Polyspace Access, you can navigate from the source code in Polyspace Access to blocks in the model.

On the **Source Code** pane in the Polyspace Access web interface, links in code comments show blocks that generate the subsequent lines of code. To see the block in the model:

- 1 Right-click a link and select **Copy MATLAB Command to Highlight Block**.



The screenshot shows a code editor window titled 'Source Code' with a file named 'test20a.c'. The code contains several lines of C code with comments. A context menu is open over line 39, which is highlighted. The menu options are: 'Go To Line', 'Copy File Path To Clipboard', and 'Copy MATLAB Command to Highlight Block' (which is highlighted in blue). The code snippet is as follows:

```

28  /* Real-time model */
29  RT_MODEL_test20a_T test20a_M_;
30  RT_MODEL_test20a_T *const test20a_M = &test20a_M_;
31
32  /* Model step function */
33  void test20a_step(void)
34  {
35  /* Outport: '<Root>/Out1' incorporates:
36   * Gain: '<Root>/Gain'
37   * Inport: '<Root>/In1'
38   */
39  test20a_Y.Out1 = 2;
40  }
41
42  /* Model initialize function */
43  void test20a_initialize(void)

```

This action copies the MATLAB command required to highlight the block. The command uses the `Simulink.ID.hilite` function.

- 2 In MATLAB, with the model open, paste and run the copied command.

Product: Polyspace Bug Finder Access.

Bug Tracking Tool Support : Create Redmine tickets for Polyspace Access results and assign to developers

In R2020a, Polyspace Access supports integration with the Redmine bug tracking tool. If you use Redmine, after you configure Polyspace Access, you can create a Redmine ticket to track Polyspace findings. The ticket is populated with details of the finding and a link to open that finding in Polyspace Access. You can add the ticket to any existing Redmine project.

Create Redmine ticket for finding #9 (10.1 The value of an expression...)

Project*

Tracker*

Subject* 10.1 The value of an expression of integer type shall not be implicitly converted to a

Description

Implicit conversion of the expression of underlying type 'signed int' to the type 'signed char' that is not a wider integer type of the same signedness.

Found in /local/test/sources/CP_C_R2019a/single_file_analysis.c

- Go to Polyspace finding here:
[https://myAccess.company.com:9443/metrics/index.html?
a=review&p=3&r=1&fid=9](https://myAccess.company.com:9443/metrics/index.html?a=review&p=3&r=1&fid=9)

Status*

Priority*

Assignee

Estimated time

Create Cancel

Once you create a ticket, the **Result Details** pane displays a link that you can click to open the ticket in the Redmine interface. See also Track Issue in Bug Tracking Tool.

Product: Polyspace Bug Finder Access.

Bug Tracking Tool Support : Manage tickets for multiple findings

In R2020a, if you create a bug tracking tool ticket in Polyspace Access, you can select multiple findings that you associate with the ticket. If a ticket already exists, you can add that ticket to additional findings or you can detach the ticket from findings that are associated with the ticket.

Previously, you could create a ticket for only one finding at a time and you could not detach a ticket from a finding.

For more information, see Track Issue in Bug Tracking Tool.

Product: Polyspace Bug Finder Access.

Results Review : See review history of findings

In R2020a, you can open the **Review History** pane to see all the changes to the review fields of findings with a timestamp and the name of the user who made the change. On the Polyspace Access toolstrip, select **Layout > Show/Hide View**.

ENVIRONMENT		REVIEW		
Result Details Review History x				
Show All				
Date and Time	User	What Chan	Original value	New value
4/27/2020 3:35:15 PM	ps_user	Comment	Reassigning to project owner	Changing severity to low
4/27/2020 3:35:04 PM	ps_user	Severity	High	Low
4/27/2020 3:34:55 PM	ps_user	Status	To investigate	To fix
4/27/2020 3:34:22 PM	jdoe	Comment	Triage of data race defects	Reassigning to project owner
4/27/2020 3:33:16 PM	jsmith	Severity	Unset	High
4/27/2020 3:33:10 PM	jsmith	Status	Unreviewed	To investigate
4/27/2020 3:33:06 PM	jsmith	Comment		Triage of data race defects

You can use this information to better understand how and why the **Severity** or **Status** of a finding has changed, and retrieve previous comments that were overwritten.

For more information, see Review History.

Product: Polyspace Bug Finder Access.

Results Review : See the configuration options used for analysis

In R2020a, you can open the **Configuration Settings** pane to view the Polyspace configuration options that were enabled to generate the analysis results. On the Polyspace Access toolstrip, select **Layout > Show/Hide View**.

Options	Value
-author	MathWorks
-checkers	all
-code-metrics	true
-compiler	gnu4.6
-critical-section-begin	BEGIN_CRITICAL_SECTION:Cs10, acquire_sensor:Cs11, acquire_printer:Cs12, acquire_sensor2:Cs13, acquire_printer2:Cs14
-critical-section-end	END_CRITICAL_SECTION:Cs10, release_sensor:Cs11, release_printer:Cs12, release_sensor2:Cs13, release_printer2:Cs14
-custom-rules	
-date	05/05/2021
-entry-points	bug_datarace_task1, bug_datarace_task2, bug_datarace_task3, bug_datarace_task4, bug_deadlock_task1, bug_deadlock_task2, bug_doublelock_task, bug_doubleunlock_task, bug_badlock_task, bug_badunlock_task, bug_dataracestdlib_task1, bug_dataracestdlib_task2, bug_destroylocked_task, corrected_datarace_task1, corrected_datarace_task2, corrected_datarace_task3, corrected_datarace_task4, corrected_deadlock_task1, corrected_deadlock_task2, corrected_doublelock_task, corrected_doubleunlock_task, corrected_badlock_task, corrected_badunlock_task, corrected_dataracestdlib_task1, corrected_dataracestdlib_task2, corrected_destroylocked_task
-generate-integration-context-info	true
-lang	C
-misra3	all
-prog	Bug_Finder_Example-Trends
-results-dir	
-target	x86_64
-verif-version	BuildNumber-9

You can use this information to better understand your results. For instance, you might expect to see a certain coding rule violation but the checker for this rule is not enabled. Previously, you had to parse the **Run Log** to see which options and checkers were enabled.

For more information, see Configuration Settings.

Product: Polyspace Bug Finder Access.

Code Quality Objectives : Customize thresholds used to track the quality of your code

In R2020a, if you use Quality Objectives to track the quality of your code, you can customize the thresholds you use as pass/fail criteria to better align with your company or project requirements. For instance, you can define quality gates to ensure adherence to a specific external coding standard.

Project Overview | Quality Objectives x | Quality Objectives Settings x

Save | Back to default

⚠ Changes to settings apply to all projects.

Quality Objectives Criteria

- Defects 289/289
- Run-time Checks 20/30
- Global Variables 0/4
- Code Metrics 13/31
- Custom Rules 0/43
- MISRAACAGC 1/129
- MISRA C:2012 49/170
- MISRA C++:2008 73/202
- MISRA C:2004 49/131**
- JSF AV C++ 0/157
- SEI CERT C 0/203
- SEI CERT C++ 0/126
- ISO/IEC TS 17961 0/46
- AUTOSAR C++14 0/251

▼ MISRA C:2004

View by Group | View by Category

	Category	SQ01	SQ02	SQ03	SQ04	SQ05	SQ06	Exhaus
▾ MISRA C:2004 49/131	–	■	■	■	■	☑	☑	☑
▶ <input type="checkbox"/> 1 Environment 0/1	–	–	–	–	–	–	–	☑
▶ <input type="checkbox"/> 2 Language extensions 0/3	–	–	–	–	–	–	–	☑
▶ <input type="checkbox"/> 3 Documentation 0/1	–	–	–	–	–	–	–	☑
▶ <input type="checkbox"/> 4 Character sets 0/2	–	–	–	–	–	–	–	☑
▶ <input checked="" type="checkbox"/> 5 Identifiers 1/7	–	☑	☑	☑	☑	☑	☑	☑
▶ <input type="checkbox"/> 6 Types 1/5	–	☐	☐	☐	☐	☑	☑	☑
▶ <input type="checkbox"/> 7 Constants 0/1	–	–	–	–	–	–	–	☑
▶ <input checked="" type="checkbox"/> 8 Declarations and definitions 3/12	–	■	■	■	■	☑	☑	☑
▶ <input checked="" type="checkbox"/> 9 Initialization 2/3	–	☐	☐	☐	☐	☑	☑	☑
▶ <input checked="" type="checkbox"/> 10 Arithmetic type conversions 2/6	–	☐	☐	☐	☐	☑	☑	☑
▶ <input checked="" type="checkbox"/> 11 Pointer type conversions 4/5	–	■	■	■	■	☑	☑	☑
▶ <input checked="" type="checkbox"/> 12 Expressions 7/13	–	■	■	■	■	☑	☑	☑
▶ <input checked="" type="checkbox"/> 13 Control statement expressions 6/7	–	■	■	■	■	☑	☑	☑
▶ <input checked="" type="checkbox"/> 14 Control flow 4/10	–	■	■	■	■	☑	☑	☑

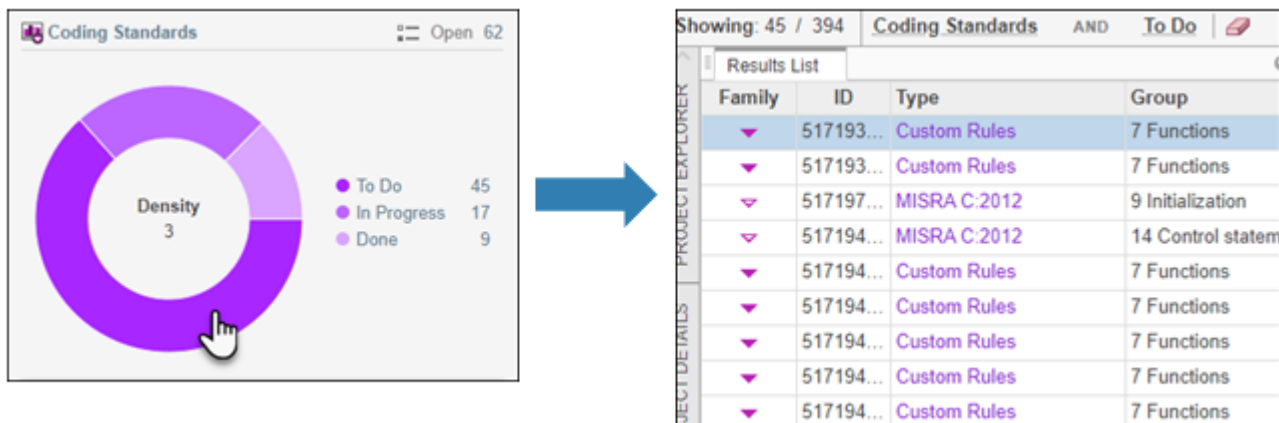
To make changes to the quality objectives settings, you must have a role of **Administrator**.

Previously, you could not see quality objective statistics for Bug Finder results. See Customize Software Quality Objectives.

Product: Polyspace Bug Finder Access.

Project Dashboard: Open results by clicking Dashboard charts

In R2020a, you can click a section of a pie chart or the legend of a pie chart to open the corresponding findings in the **Results List** and more easily narrow the scope of your review.



Product: Polyspace Bug Finder Access.

Exporting Results : Export only results that must be reviewed to satisfy software quality objectives (SQOs)

In R2020a, when exporting Polyspace results from the Polyspace Access web interface to a text file, you can export only those results that must be fixed or justified to satisfy your software quality objectives. The software quality objectives are specified through a progressively stricter set of SQO levels, numbered from 1 to 6.

See also:

- `polyspace-access`
- Send Email Notifications with Polyspace Bug Finder Results
- Bug Finder Quality Objectives (Polyspace Bug Finder Access)

You can customize the requirements of each level in the Polyspace Access web interface, and then use the option `-open-findings-for-sqo` with the level number to export only those results that must be reviewed to meet the requirements.

Product: Polyspace Bug Finder Access.

Report Generation : Configure report generator to communicate with Polyspace Access over HTTPS

In R2020a, if you generate reports for results that are stored on Polyspace Access, you can configure the `polyspace-report-generator` binary to communicate with Polyspace Access over HTTPS.

Use the `-configure-keystore` option to run this one-time configuration step. See `polyspace-report-generator`.

Previously, you needed a Polyspace Bug Finder desktop license to generate reports if Polyspace Access was configured with HTTPS.

Product: Polyspace Bug Finder Access.

Report Generation : Navigate to Polyspace Access Results List from report

In R2020a, if you generate a report for results that are stored on Polyspace Access, you can navigate from the report to the **Results List** in the Polyspace Access web interface.

ID	Guideline	Message	Function
68688	D1.1	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood. The abort function returns an implementation-defined termination status to the host environment.	File Scope
68695	21.8	The library functions abort, exit and system of <stdlib.h> shall not be used.	File Scope
68841	8.4	A compatible declaration shall be visible when an object or function with external linkage is defined. Function 'bug_datarace_task1' has no visible prototype at definition.	File Scope
68835	8.4	A compatible declaration shall be visible when an object or function with external linkage is defined. Function 'bug_datarace_task2' has no visible prototype at definition.	File Scope

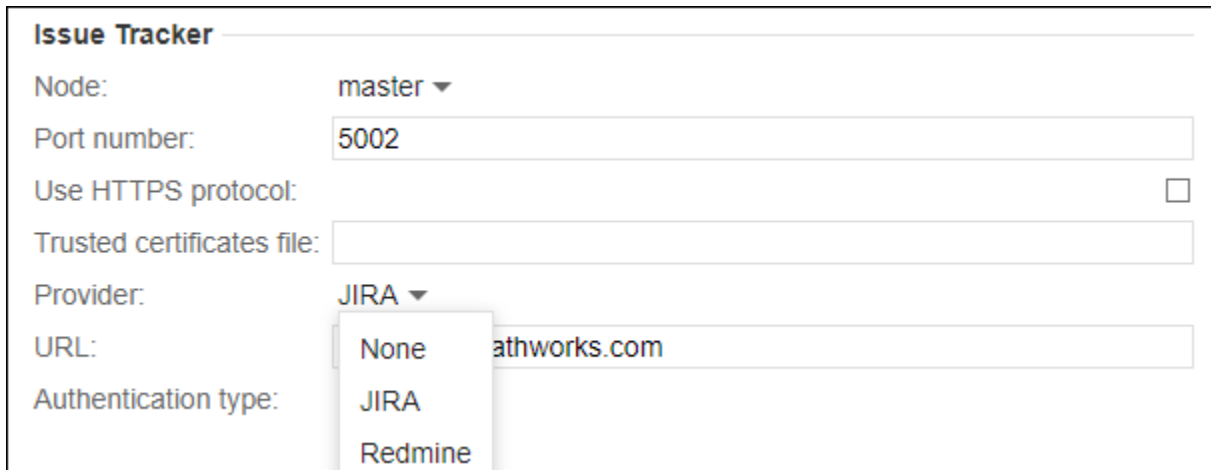
Click the link in the **ID** column to open Polyspace Access with the **Results List** filtered down to the corresponding finding.

Product: Polyspace Bug Finder Access.

Polyspace Access Installation

Installation and Configuration : New Issue Tracker service

In R2020a, use the new **Issue Tracker** service to configure Polyspace Access to integrate with the Jira software or Redmine bug tracking tools.



The screenshot shows the 'Issue Tracker' configuration form. The fields are as follows:

- Node:** master (dropdown menu)
- Port number:** 5002 (text input)
- Use HTTPS protocol:**
- Trusted certificates file:** (empty text input)
- Provider:** JIRA (dropdown menu)
- URL:** athworks.com (text input)
- Authentication type:** JIRA (dropdown menu)

The dropdown menu for 'Authentication type' is open, showing options: None, JIRA, and Redmine.

See Configure the **User Manager** and **Issue Tracker**.

Product: Polyspace Bug Finder Access.

Installation and Configuration : Change in default location of Polyspace Access data volume and working directories

In R2020a, the default location of the working directories of the Polyspace Access **Web Server** and **ETL** services and of the data volume is inside the folder where you unzipped the Polyspace Access ZIP file, under the polyspace folder.

Previously, the working directories of the **Web Server** and **ETL** were stored in the temporary files folder of your system (/tmp on Linux or %TEMP% on Windows). The data volume was stored under /var/lib/docker/volumes on Linux.

Product: Polyspace Bug Finder Access.

R2019b

Version: 3.1

New Features

Bug Fixes

Compatibility Considerations

Analysis Setup

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

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

Target Environment	
Compiler	cosmic
Target processor type	s12z

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

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.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

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

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 version before starting the Polyspace analysis.

Target Environment	
Compiler	gnu4.8
Target processor type	gnu3.4
<input type="checkbox"/> Block char16/32_t t	gnu4.6
	gnu4.7
	gnu4.8
Compiler Behavior	gnu4.9
<input type="checkbox"/> Division round down	gnu5.x
	gnu6.x
Pack alignment value	gnu7.x

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.

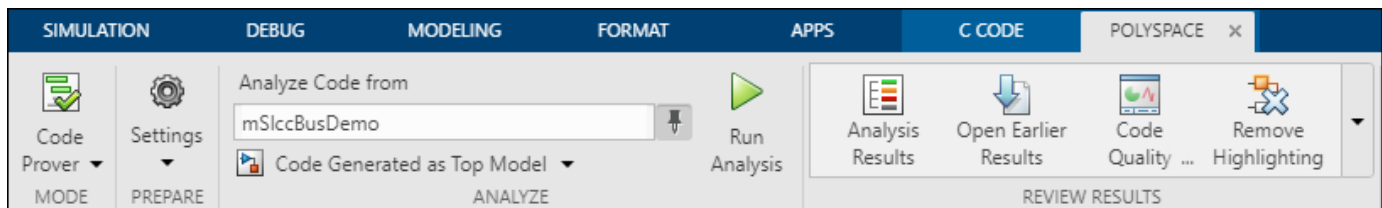
Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

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

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**.



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

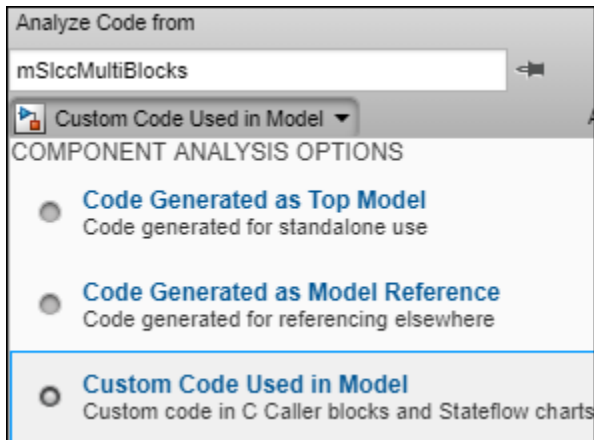
Additional Considerations

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.

Product: Polyspace Bug Finder (Desktop).

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

In R2019b, Polyspace can check functions called from C Caller blocks 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.

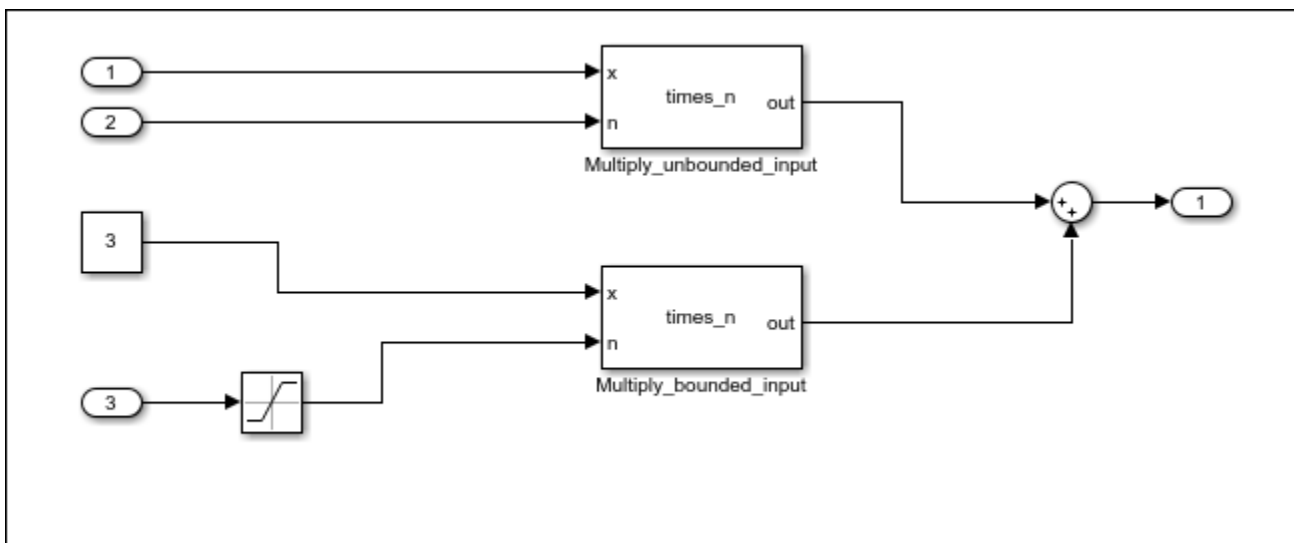
With support for custom code analysis, you can:

- 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`).



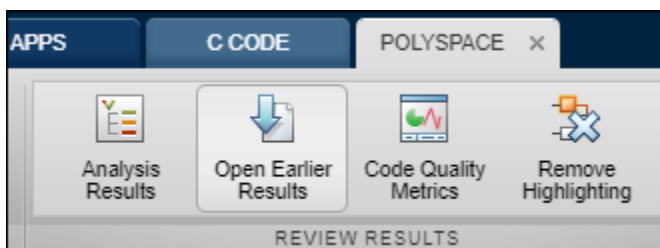
When you analyze custom code, in this case the function `times_n`, the analysis shows that an operation in the custom code can overflow. From the analysis results, you can determine that the

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.

Product: Polyspace Bug Finder (Desktop).

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

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.



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.

Product: Polyspace Bug Finder (Desktop).

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.

Product: Polyspace Bug Finder (Desktop).

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`.

Product: Polyspace Bug Finder (Desktop).

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`.

Product: Polyspace Bug Finder (Desktop).

Analysis Results

AUTOSAR C++14 Support: Check for misuse of lambda expressions, potential problems with enumerations, and other issues

In R2019b, you can look for violations of these AUTOSAR C++14 rules in addition to previously supported rules.

AUTOSAR C++14 Rule	Description	Polyspace Checker
A0-1-4	There shall be no unused named parameters in non-virtual functions.	AUTOSAR C++14 Rule A0-1-4
A3-1-2	Header files, that are defined locally in the project, shall have a file name extension of one of: .h, .hpp or .hxx.	AUTOSAR C++14 Rule A3-1-2
A5-1-2	Variables shall not be implicitly captured in a lambda expression.	AUTOSAR C++14 Rule A5-1-2
A5-1-3	Parameter list (possibly empty) shall be included in every lambda expression.	AUTOSAR C++14 Rule A5-1-3
A5-1-4	A lambda expression shall not outlive any of its reference-captured objects.	AUTOSAR C++14 Rule A5-1-4
A5-1-7	A lambda shall not be an operand to decltype or typeid.	AUTOSAR C++14 Rule A5-1-7
A5-16-1	The ternary conditional operator shall not be used as a sub-expression.	AUTOSAR C++14 Rule A5-16-1
A7-2-2	Enumeration underlying base type shall be explicitly defined.	AUTOSAR C++14 Rule A7-2-2
A7-2-3	Enumerations shall be declared as scoped enum classes.	AUTOSAR C++14 Rule A7-2-3
A16-0-1	The preprocessor shall only be used for unconditional and conditional file inclusion and include guards, and using the following directives: (1) #ifndef, (2) #ifdef, (3) #if, (4) #if defined, (5) #elif, (6) #else, (7) #define, (8) #endif, (9) #include.	AUTOSAR C++14 Rule A16-0-1
A16-7-1	The #pragma directive shall not be used.	AUTOSAR C++ 14 Rule A16-7-1

AUTOSAR C++14 Rule	Description	Polyspace Checker
A18-1-1	C-style arrays shall not be used.	AUTOSAR C++ 14 Rule A18-1-1
A18-1-2	The <code>std::vector<bool></code> specialization shall not be used.	AUTOSAR C++ 14 Rule A18-1-2
A18-5-1	Functions <code>malloc</code> , <code>calloc</code> , <code>realloc</code> and <code>free</code> shall not be used.	AUTOSAR C++ 14 Rule A18-5-1
A18-9-1	The <code>std::bind</code> shall not be used.	AUTOSAR C++ 14 Rule A18-9-1

For all supported AUTOSAR C++14 rules, see AUTOSAR C++14 Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

CERT C++ Support: Check for pointer escape via lambda expressions, exceptions caught by value, use of bitwise operations for copying objects, and other issues

In R2019b, you can look for violations of these CERT C++ rules in addition to previously supported rules.

CERT C++ Rule	Description	Polyspace Checker
DCL59-CPP	Do not define an unnamed namespace in a header file	CERT C++: DCL59-CPP
EXP61-CPP	A lambda object shall not outlive any of its reference captured objects.	CERT C++: EXP61-CPP
MEM57-CPP	Avoid using default operator new for over-aligned types	CERT C++: MEM57-CPP
ERR61-CPP	Catch exceptions by lvalue reference	CERT C++: ERR61-CPP
OOP57-CPP	Prefer special member functions and overloaded operators	CERT C++: OOP57-CPP

For all supported CERT C++ rules, see CERT C++ Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

CERT C Support: Check for undefined behavior from successive joining or detaching of the same thread

In R2019b, you can look for violations of these CERT C rules in addition to previously supported rules.

CERT C Rule	Description	Polyspace Checker
CON39-C	Do not join or detach a thread that was previously joined or detached	CERT C: Rule CON39-C

For all supported CERT C guidelines, see CERT C Rules and Recommendations.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

New Bug Finder Defect Checkers: Check for new security vulnerabilities, multithreading issues, missing C++ overloads, and other issues

In R2019b, you can check for these new types of defects.

Defect	Description
Unnamed namespace in header file	Header file contains unnamed namespace leading to multiple definitions
Lambda used as decltype or typeid operand	decltype or typeid is used on lambda expression
Operator new not overloaded for possibly overaligned class	Allocated storage might be smaller than object alignment requirement
Bytewise operations on nontrivial class object	Value representations may be improperly initialized or compared
Missing hash algorithm	Context in EVP routine is initialized without a hash algorithm
Missing salt for hashing operation	Hashed data is vulnerable to rainbow table attack
Missing X.509 certificate	Server or client cannot be authenticated
Missing certification authority list	Certificate for authentication cannot be trusted
Missing or double initialization of thread attribute	Noninitialized thread attribute used in functions that expect initialized attributes or duplicated initialization of thread attributes
Use of undefined thread ID	Thread ID from failed thread creation used in subsequent thread functions
Join or detach of a joined or detached thread	Thread that was previously joined or detached is joined or detached again

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

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

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.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Updated Bug Finder defect checkers

In R2019b, this defect checker has been updated.

Defect	Description	Update
Pointer or reference to stack variable leaving scope	Pointer to local variable leaves the variable scope	The checker now detects pointer escape via lambda expressions.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Compatibility Considerations

If you check for the defect mentioned above, you can see a difference in the number of issues found.

Reviewing Results

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

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.

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 DEFECT:FLOAT_OVFL "This operation
      cannot overflow
      because of
      external constraints" */
```

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Access

Polyspace Access Installation

User Authentication: Use LDAP search filters to restrict number of users to authenticate

In R2019b, if you use your organization's Lightweight Directory Access Protocol (LDAP) to authenticate users, you can filter for and load a subset of users from your LDAP database when you start Polyspace Bug Finder Access. Previously, you loaded all LDAP users listed under the **LDAP base** that you specified when you started Polyspace Bug Finder Access.

To filter the LDAP users, use the new **LDAP search filter** field in the Cluster Operator settings for the **User Manager** service. For more information, see Use Your Organization LDAP.

Product: Polyspace Bug Finder Access.

User Management : Update list of users from LDAP database or LDIF file

In R2019b, if you remove users from your organization's Lightweight Directory Access Protocol (LDAP) database or from the Polyspace Access embedded LDAP LDIF file, you can update the list of users stored in the Polyspace Access database. Previously, users that were removed from the LDAP database or from the LDIF file were still visible in the list of users you selected when assigning findings or managing project permissions.

To update the list of users stored in the Polyspace Access database, append `/users/list/removed` to the URL that you use to Open the Polyspace Access Web Interface. Only an **Administrator** can perform this operation. For more information, see Manage LDAP Users in Polyspace Access.

Product: Polyspace Bug Finder Access.

R2019a

Version: 3.0

New Features

Bug Fixes

Compatibility Considerations

Analysis Setup

Polyspace-only Licenses: Install Polyspace without MATLAB installation

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

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.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server , Polyspace Bug Finder Access.

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 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.

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

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.

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:

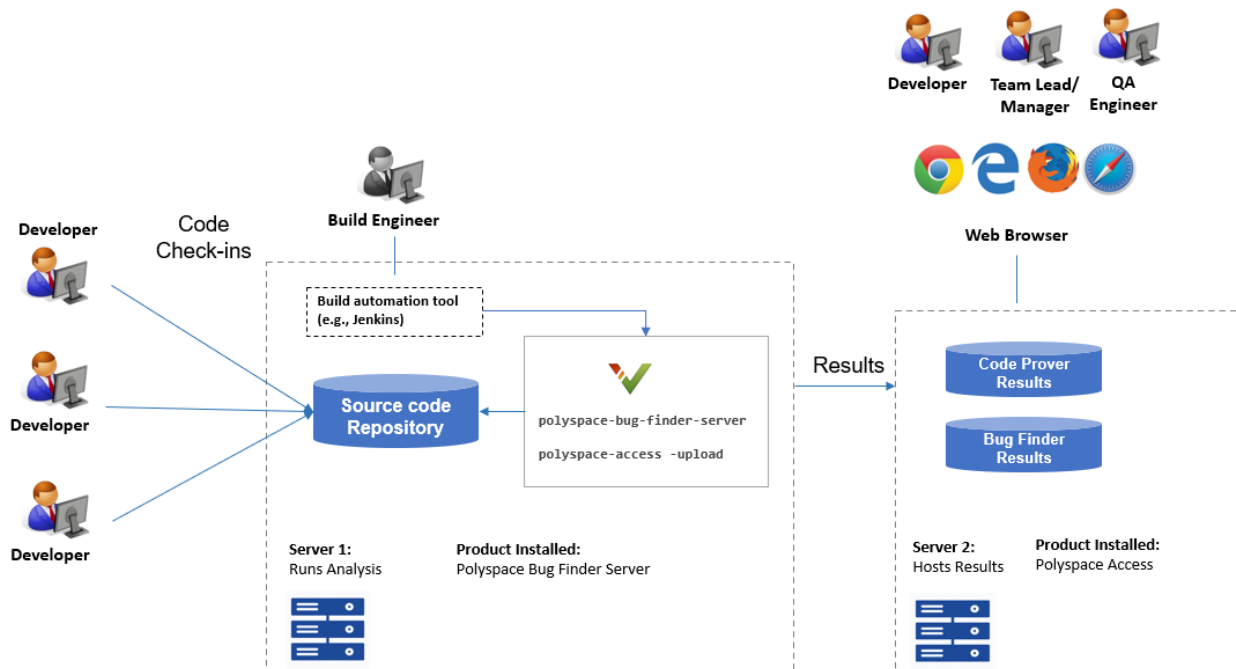
- *Prior to code submission:*

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:

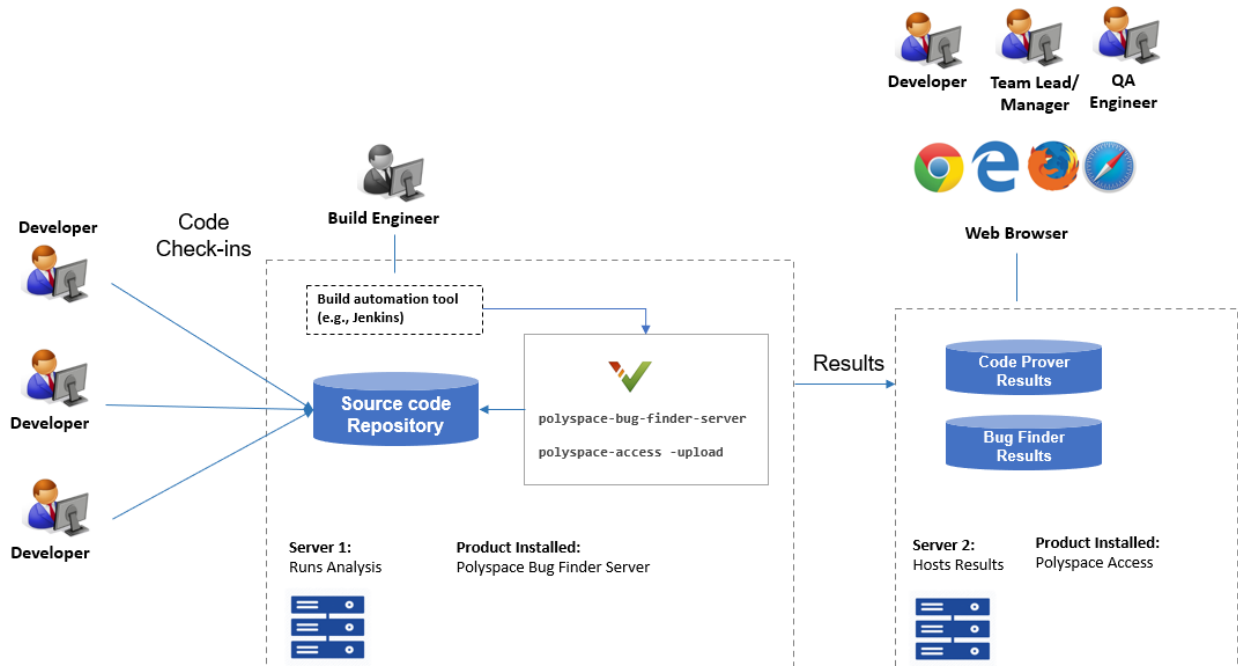
- Polyspace Bug Finder Server
- Polyspace Code Prover Server
- Polyspace Bug Finder Access
- Polyspace Code Prover Access

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server , Polyspace Bug Finder Access.

Bug Finder Analysis Engine Separated from Viewer : Run Bug Finder analysis on server and view the results from multiple client machines

In R2019a, you can run Bug Finder on a server with the new product, Polyspace Bug Finder Server. You can then host the analysis results on the same server or a second server with the product, Polyspace Bug Finder Access. Developers whose code was analyzed (and other reviewers such as quality engineers and development managers) can fetch these results from the server to their

desktops and view the results in a web browser, provided they have a Polyspace Bug Finder Access license.



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

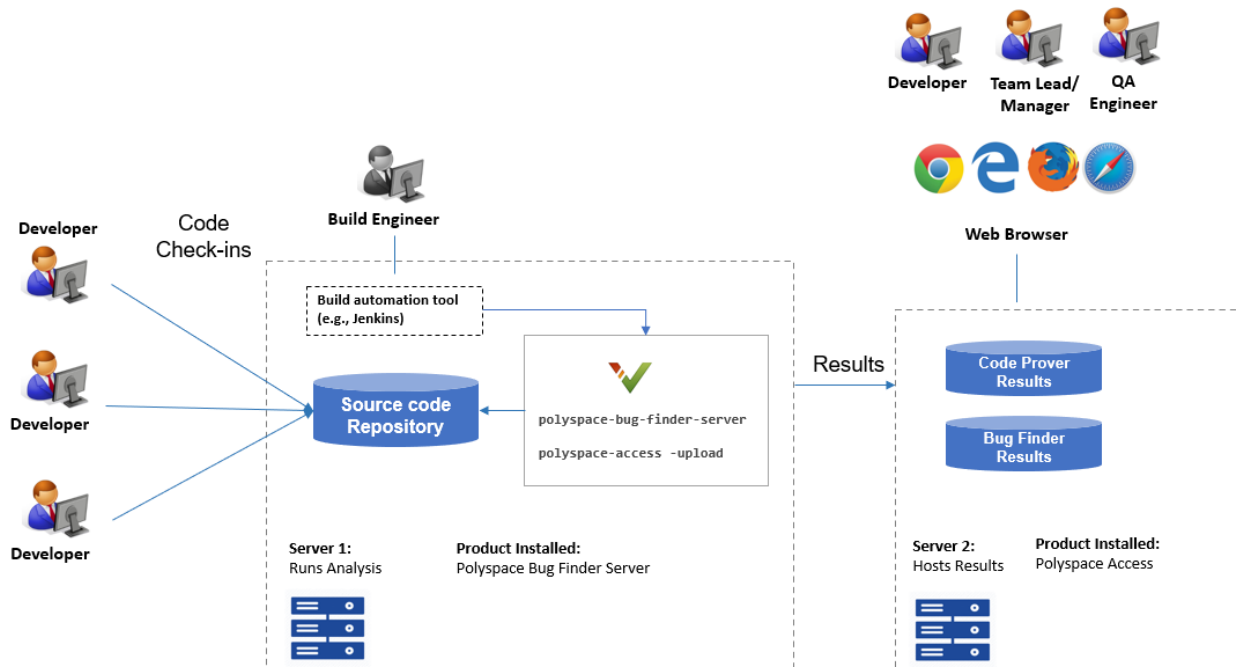
You can run the Bug Finder analysis on a few powerful server class machines but view the analysis results from many terminals.

With the desktop product, Polyspace Bug Finder, you have to run the analysis and view the results on the same machine. To view the results on a different machine, you need a second instance of a desktop product. The desktop products can now be used by individual developers on their desktops prior to code submission and the server products used after code submission. See Polyspace Products for Code Analysis and Verification.

Product: Polyspace Bug Finder Server.

Continuous Integration Support: Run Bug Finder on server class computers with continuous upload to Polyspace Access web interface

In R2019a, you can check for bugs, coding standard violations and other issues on server class machines as part of continuous integration. When developers submit code to a shared repository, a build automation tool such as Jenkins can perform the checks using the new Polyspace Bug Finder Server product. The analysis results can be uploaded to the Polyspace Access web interface for review. Each reviewer with a Polyspace Bug Finder Access license can login to the Polyspace Access web interface and review the results.



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

See:

- Install Polyspace Server and Access Products
- Run Polyspace Bug Finder on Server and Upload Results to Web Interface

The continuous integration support with the Polyspace Server and Access products enables the following:

- *Automated post-submission checks:* In a continuous integration process, build scripts run automatically on new code submissions before integration with a code base. With the new product Polyspace Bug Finder Server, a Bug Finder analysis can be included in this build process. The analysis can run a specific set of Bug Finder checkers on the new code submissions and report the results. The results can be reviewed in the Polyspace Access web interface with a Polyspace Bug Finder Access license.
- *Collaborative review:* The analysis results can be uploaded to the Polyspace Access web interface for collaborative review. For instance:
 - Each quality assurance engineer with a Polyspace Bug Finder Access license can review the Bug Finder results on a project and assign issues to developers for fixing.
 - Each development team manager with a Polyspace Bug Finder Access license can see an overview of Bug Finder results for all projects managed by the team (and also drill down to details if necessary).

For further details, see the release notes of Polyspace Bug Finder Access .

Product: Polyspace Bug Finder Server.

Continuous Integration Support : Set up testing criteria based on Bug Finder static analysis results

In R2019a, you can run Bug Finder on server class machines as part of unit and integration testing. You can define and set up testing criteria based on Bug Finder static analysis results.

For instance, you can set up the criteria that new code submissions must have zero high-impact defects before integration with a code base. Any submission with high-impact defects can cause a test failure and require code fixes.

See:

- `polyspace-bug-finder-server` for how to run Bug Finder on servers.
- `polyspace-access` for how to export Bug Finder results for comparison against predefined testing criteria.

If you use Jenkins for build automation, you can use the Polyspace plugin. The plugin provides helper functions to filter results based on predefined criteria. See *Sample Scripts for Polyspace Analysis with Jenkins*.

The continuous integration support with the Polyspace Server and Access products enables the following:

- *Automated testing*: After you define testing criteria based on Bug Finder results, you can run the tests along with regular dynamic tests. The tests can run on a periodic schedule or based on predefined triggers.
- *Prequalification with Polyspace desktop products*: Prior to code submission, to avoid test failures, developers can perform a pre-submit analysis on their code with the same criteria as the server-side analysis. Using an installation of the desktop product, Polyspace Bug Finder, developers can emulate the server-side analysis on their desktops and review the results in the user interface of the desktop product. For more information on the complete suite of Polyspace products available for deployment in a software development workflow, see *Polyspace Products for Code Analysis and Verification*.

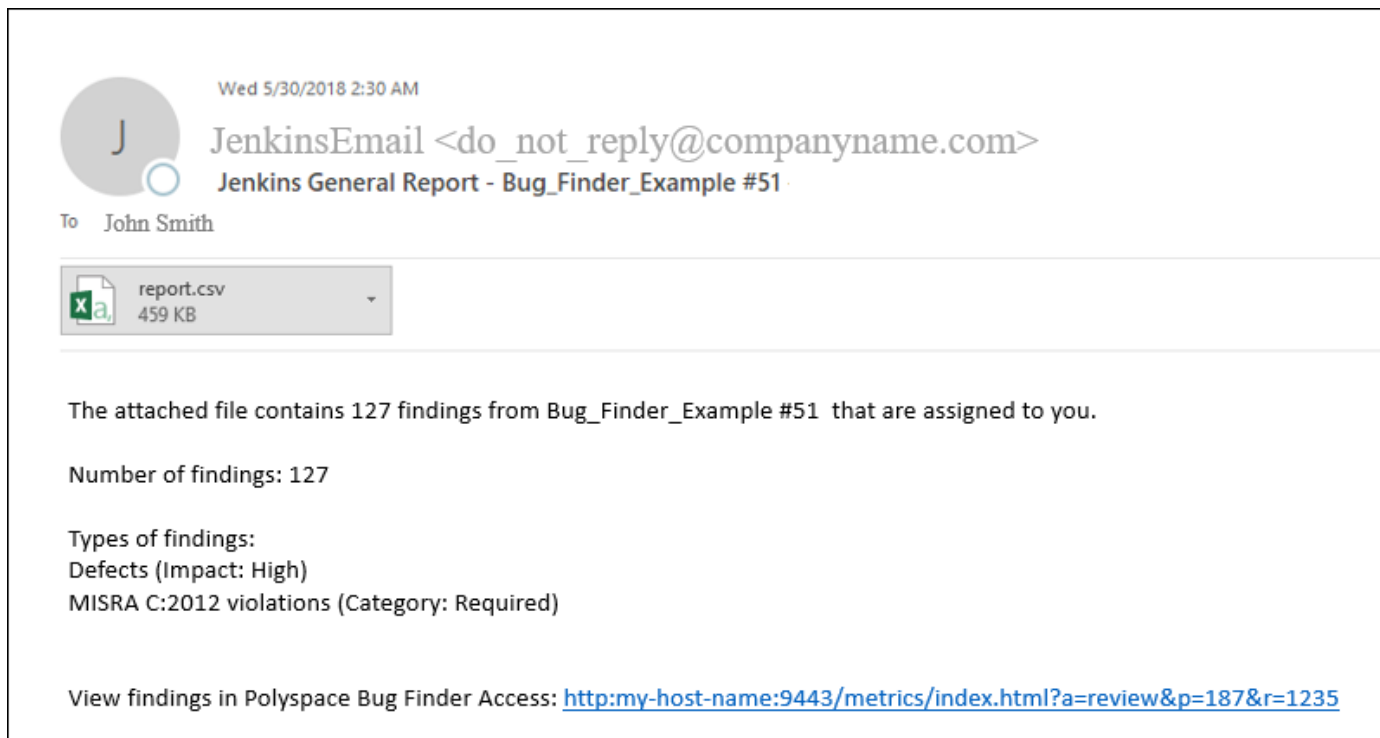
To save processing power on the desktop, the analysis can also be offloaded to a server and only the results reviewed on the desktop. See *Install Products for Submitting Polyspace Analysis from Desktops to Remote Server*.

Product: Polyspace Bug Finder Server.

Continuous Integration Support : Set up email notification with summary of Bug Finder results after analysis

In R2019a, you can set up email notification for new Bug Finder results. The email can contain:

- A summary of new results from the latest Bug Finder run only for specific files or modules.
- An attachment with a full list of the new results. Each result has an associated link to the Polyspace Access web interface for more detailed information.



See Send E-mail Notifications with Polyspace Bug Finder Results.

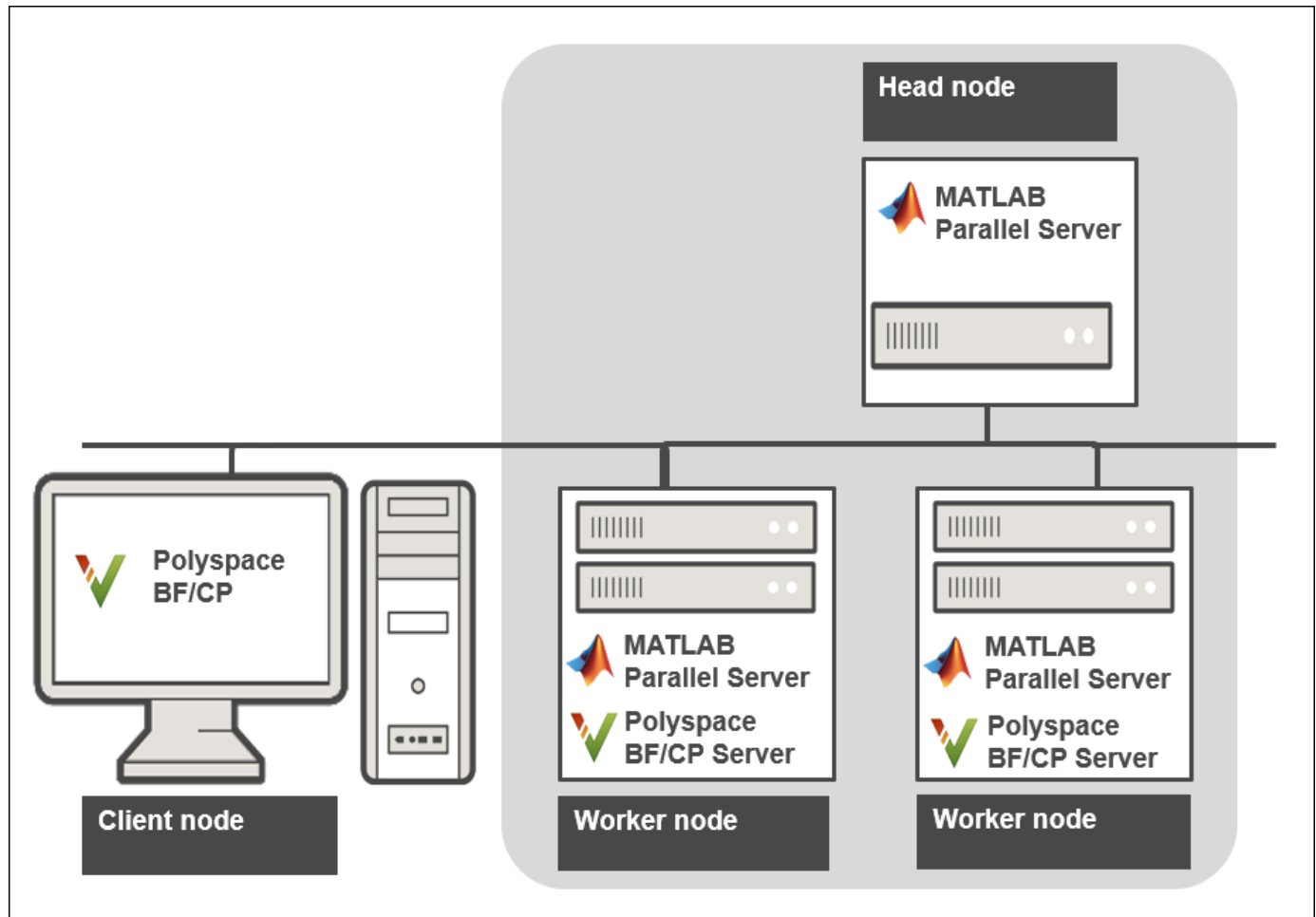
The continuous integration support with the Polyspace Server and Access products enables the following:

- *Automated notification:* Developers can get notified in their e-mail inbox about results from the last Bug Finder run on their submissions.
- *Preview of Bug Finder results:* Developers can see a preview of the new Bug Finder results. Based on their criteria for reviewing results, this preview can help them decide whether they want to see further details of the results.
- *Easy navigation from e-mail summary to Polyspace Access web interface:* Each developer with a Polyspace Bug Finder Access license can use the links in the e-mail attachments to see further details of a result in the Polyspace Access web interface.

Product: Polyspace Bug Finder Server.

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

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 Bug Finder Server license for running the analysis on dedicated servers and Polyspace Bug Finder Access licenses to review the results.

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.

Product: Polyspace Bug Finder (Desktop).

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 Bug Finder.
- On the server side, instead of the desktop product, Polyspace Bug Finder, you must install the server product, Polyspace Bug Finder 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*.

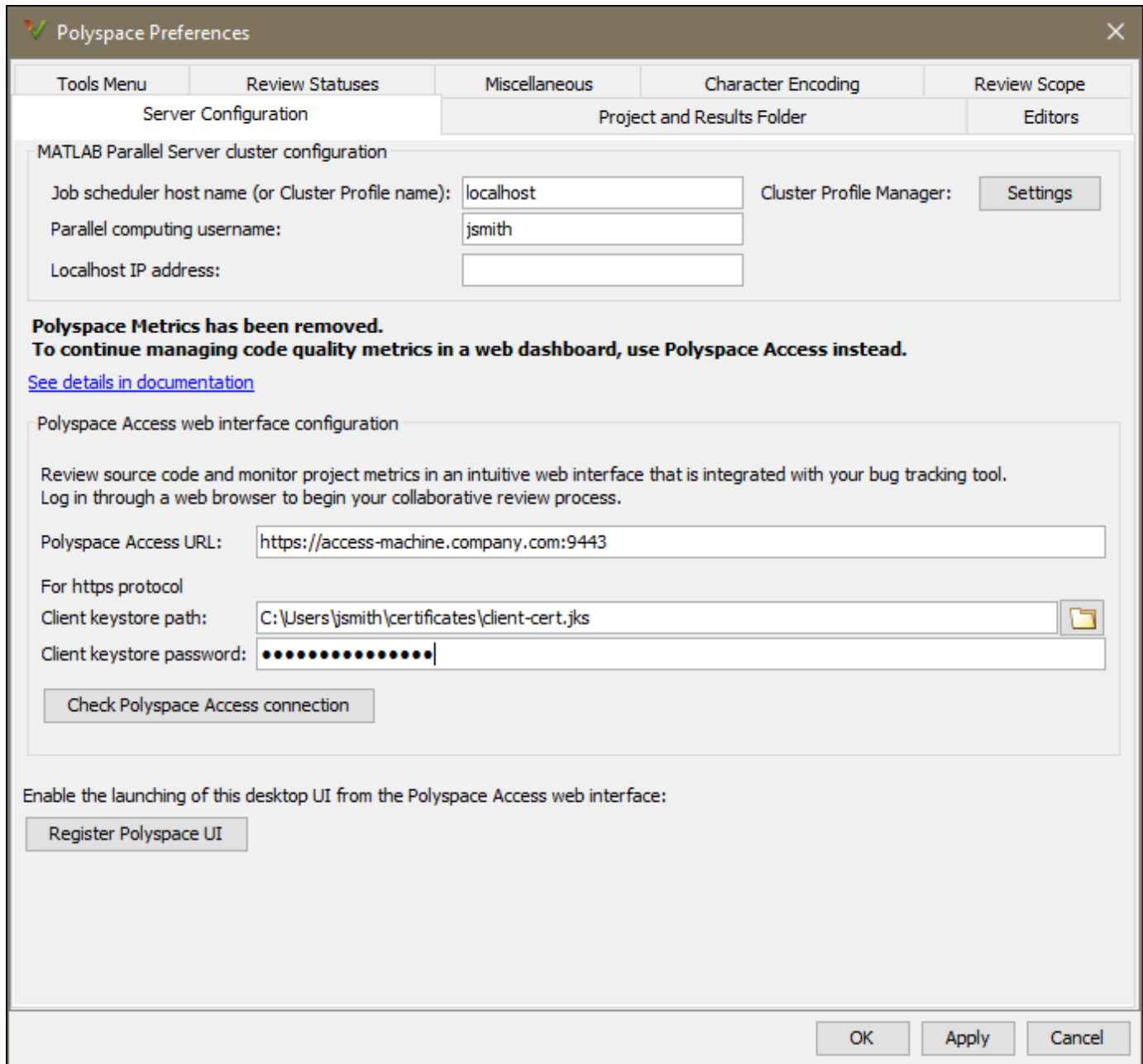
- You do not have the quick start option to start the server with one worker (the **Metrics and Remote Analysis Server Settings** interface). Instead you must use the **Admin Center** interface in MATLAB Parallel Server. In this workflow, you first start the services on all remote computers, then assign 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*.

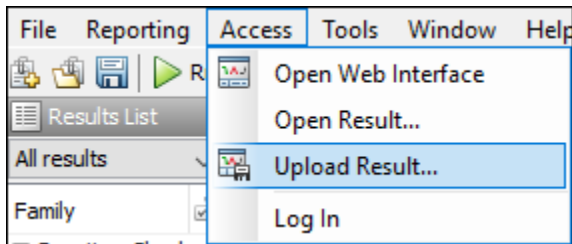
Collaborative Review Support: Upload results from Polyspace user interface to Polyspace Access web interface and share results using web links

In R2019a, you can upload Polyspace Bug Finder results to the Polyspace Access web interface. Developers with a Polyspace Bug Finder 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 Bug Finder Access documentation.

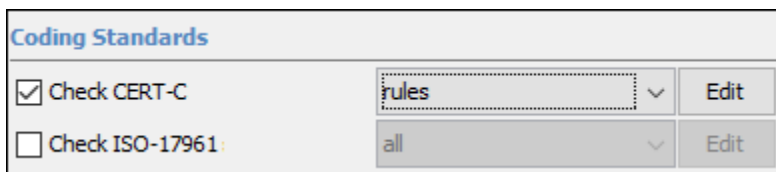
The Polyspace Access products enable the following:

- *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 Bug Finder Access license and the appropriate credentials can view the results from their web browser.

Product: Polyspace Bug Finder (Desktop).

Support for Security Standards: Check explicitly for subsets of CERT C, CERT C++ or ISO/IEC TS 17961 rules

In R2019a, you can check explicitly for violations of the CERT C, CERT C++ or ISO/IEC TS 17961 standard. You can check for all supported rules from the standard or reduce the checking to a predefined subset or your own subset of rules.



See Check for Coding Standard Violations.

The new support for security standards enable the following:

- *Direct configuration of security standards:* You can specify rules from the security standards directly in your analysis configuration. Previously, to check for a security standard, you configured Bug Finder checkers by using an external mapping between the checkers and rules from the standard.

- *More fine-grained control on checking of security standards:* Instead of checking for all supported rules, you can configure smaller subsets of the standards based on your requirements. You can check your code for up to a single rule from a standard.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

Compatibility Considerations

In previous releases, to check for a security standard, you configured Bug Finder checkers in your analysis configuration. In the list of results, you enabled a **CERT ID** or **ISO-17961 ID** column to see the CERT C or ISO/IEC TS 17961 rules corresponding to a defect. In R2019a, if you are interested in standards such as CERT C, CERT C++ or ISO/IEC TS 17961, use a workflow that is directly geared towards the standard. Enable rules from the standard that you are interested in and see rule violations explicitly in your analysis results.

SEI CERT C (289/317)			
Preprocessor (PRE)	<input type="checkbox"/>	recommend...	EXP19-C Use braces for the body of an if, for, or while state...
Declarations and Initialization (DCL)	<input type="checkbox"/>	recommend...	EXP20-C Perform explicit tests to determine success, true a...
Expressions (EXP)	<input type="checkbox"/>	rule	EXP30-C Do not depend on the order of evaluation for side ...
Integers (INT)	<input type="checkbox"/>	rule	EXP32-C Do not access a volatile object through a nonvolatil...
Floating Point (FLP)	<input checked="" type="checkbox"/>	rule	EXP33-C Do not read uninitialized memory
Arrays (ARR)	<input checked="" type="checkbox"/>	rule	EXP34-C Do not dereference null pointers
Characters and Strings (STR)	<input checked="" type="checkbox"/>	rule	EXP35-C Do not modify objects with temporary lifetime
Memory Management (MEM)	<input type="checkbox"/>	rule	EXP36-C Do not cast pointers into more strictly aligned point...
Input Output (FIO)	<input type="checkbox"/>	rule	EXP37-C Call functions with the correct number and type of ...
Environment (ENV)	<input checked="" type="checkbox"/>	rule	EXP39-C Do not access a variable through a pointer of an in...
	<input type="checkbox"/>	rule	EXP40-C Do not modify constant objects

See also Changes in Coding Standard Checking in R2019a.

Coding Standard Support: Enforce common standards across team or organization by reusing checker configuration

In R2019a, you can specify the coding standard checkers independently from the remaining analysis configuration. You can reuse this specification across multiple Polyspace projects.

Coding Rules, Coding Standards & Code Metrics	
<input checked="" type="checkbox"/> Set checkers by file	C:\Polyspace_projects\checker_specs_MISRA_C_2012
Coding Rules	
<input type="checkbox"/> Check MISRA C:2004	required-rules ▼ Edit
<input type="checkbox"/> Check MISRA AC AGC	OBL-rules ▼ Edit
<input checked="" type="checkbox"/> Check MISRA C:2012	from-file ▼ Edit
<input type="checkbox"/> Use generated code requirements	

Reusable coding standard specifications are supported for the standards MISRA C: 2004, MISRA C: 2012, MISRA C++, JSF++, CERT C, CERT C++, ISO/IEC TS 17961 and AUTOSAR C++14.

See Check for Coding Standard Violations.

The new coding standards support enables the following:

- *Project-specific settings decoupled from project-independent settings:* Analysis options such as macro definitions and entry points for multitasking can be specific to the source files in a project while the coding standard checkers can apply to multiple projects. You can now separate the checker specifications from the project-specific options and reuse the checkers across multiple projects. Previously, reusable checker specifications were not directly supported.
- *Common standard across team:* You can enforce common coding standards across a team or organization by reusing checker specifications across all projects.

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

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

- ARM® v5

Target Environment	
Compiler	armcc ▼
Target processor type	arm ▼

You can specify target arm.

See ARM v5 Compiler (-compiler armcc).

- ARM v6

Target Environment	
Compiler	armclang
Target processor type	arm

You can specify targets arm and arm64.

See ARM v6 Compiler (-compiler armclang).

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.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

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

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:

- | Target Environment | |
|-----------------------|--------|
| Compiler | gnu7.x |
| Target processor type | x86_64 |

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

- | Target Environment | |
|-----------------------|----------|
| Compiler | clang4.x |
| Target processor type | x86_64 |

clang4.x for LLVM release 4.0.0, and 4.0.1.

- | Target Environment | |
|-----------------------|----------|
| Compiler | clang5.x |
| Target processor type | x86_64 |

clang5.x for LLVM release 5.0.0, and 5.0.1.

- | Target Environment | |
|-----------------------|------------|
| Compiler | visual15.x |
| Target processor type | x86_64 |

visual15.x for Microsoft Visual C++ 2017 versions 15.0 to 15.7.

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)`.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

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

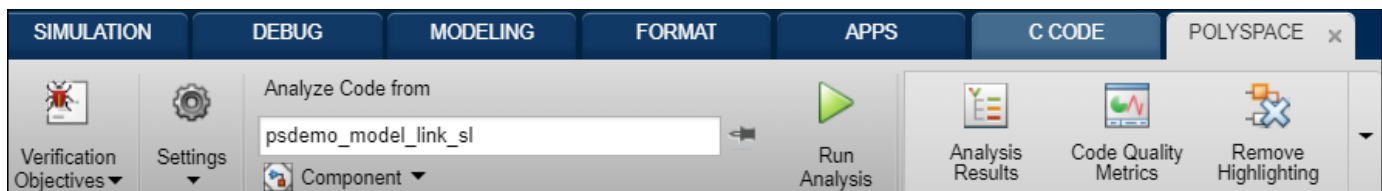
In R2019a, you have the option to turn on the Simulink Toolstrip.

- 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.

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.

Product: Polyspace Bug Finder (Desktop).

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-bug-finder-nodesktop renamed to polyspace-bug-finder

Warns

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

-report-template arguments changed for coding standard templates

Warns

A single report template `CodingStandards.rpt` is used for all coding standards (other than CWE). In particular, if you used these old templates as arguments for the option `-report-template`, switch to the new template:

- `CodingRules.rpt`
- `SecurityCERT.rpt`
- `SecurityISO_17961.rpt`

See also Changes in Coding Standard Checking in R2019a.

Find defects (-checkers) option values CERT-rules, CERT-all, and ISO-17961 are removed

Warns

Find defects (`-checkers`) option values `CERT-rules`, `CERT-all`, and `ISO-17961` are removed. Previously, you used **Find defects (-checkers)** with these options values to check your code for violations of the CERT C, CERT C++, and ISO/IEC TS 17961 coding standards. Use the new **Coding Standards & Code Metrics** analysis options `Check SEI CERT-C (-cert-c)`, `Check SEI CERT-C++ (-cert-cpp)`, and `Check ISO/IEC TS 17961 (-iso-17961)` instead.

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

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 these tables.

If the source code language is C:

Option	Use Instead
<code>-checkers CERT-rules</code>	<code>-cert-c all-rules</code>
<code>-checkers CERT-all</code>	<code>-cert-c all</code>
<code>-checkers ISO-17961</code>	<code>-iso-17961 all</code>

If the source code language is C++:

Option	Use Instead
-checkers CERT-rules	-cert-cpp all
-checkers CERT-all	

You get a warning and when you use the removed option values at the command line. The corresponding new options are applied automatically.

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 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

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
-misra3 CERT-rules	-cert-c all-rules
-misra3 CERT-all	-cert-c all
-misra3 ISO-17961	-iso-17961 all

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

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 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

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
-misra-cpp CERT-rules	-cert-cpp all
-misra-cpp CERT-all	

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

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server

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.

Product: Polyspace Bug Finder (Desktop).

polyspaceBugFinderNodesktop removed

Warns

Use `polyspaceBugFinder(projectFile, '-nodesktop')` instead of `polyspaceBugFinderNodesktop(projectFile)`.

Product: Polyspace Bug Finder (Desktop).

BugFinderReportTemplate property values changed for coding standard compliance reports

Warns

A single report template is used for all coding standards (other than CWE).

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

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

instead of these old templates:

- `CodingRules`
- `SecurityCERT`
- `SecurityISO_17961`

See also Changes in Coding Standard Checking in R2019a.

Product: Polyspace Bug Finder (Desktop).

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.

```
proj = polyspace.Project;
opts = proj.Configuration;
```

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

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

Product: Polyspace Bug Finder (Desktop).

Option values **CERT-rules**, **CERT-all**, and **ISO-17961** are removed for **BugFinderAnalysis** property **CheckersPreset**

Errors

CheckersPreset option values **CERT-rules**, **CERT-all**, and **ISO-17961** are removed. Previously, you used CheckersPreset with these options values to check your code for violations of the CERT C and ISO/IEC TS 17961 coding standards. Use 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.

To update your MATLAB code, see this table.

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

Property	Use Instead
<pre>opts.BugFinderAnalysis... .EnableCheckers=1; opts.BugFinderAnalysis... .CheckersPreset='CERT-all';</pre>	<pre>opts.CodingRulesCodeMetrics.EnableCertC=1; opts.CodingRulesCodeMetrics.CertC='all';</pre>
<pre>opts.BugFinderAnalysis... .EnableCheckers=1; opts.BugFinderAnalysis... .CheckersPreset='CERT-rules';</pre>	<pre>opts.CodingRulesCodeMetrics.EnableCertC=1; opts.CodingRulesCodeMetrics.CertC=... 'all-rules';</pre>
<pre>opts.BugFinderAnalysis... .EnableCheckers=1; opts.BugFinderAnalysis... .CheckersPreset='iso-17961';</pre>	<pre>opts.CodingRulesCodeMetrics.EnableIso17961=1; opts.CodingRulesCodeMetrics.Iso17961='all'</pre>

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

Product: Polyspace Bug Finder (Desktop).

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 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.

To update your MATLAB code, see this table.

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

Property	Use Instead
<pre>opts.CodingRulesCodeMetrics... .EnableMisraC3=1; opts.CodingRulesCodeMetrics... .MisraC3Subset='CERT-all';</pre>	<pre>opts.CodingRulesCodeMetrics.EnableCertC=1; opts.CodingRulesCodeMetrics.CertC='all';</pre>
<pre>opts.CodingRulesCodeMetrics... .EnableMisraC3=1; opts.CodingRulesCodeMetrics... .MisraC3Subset='CERT-rules';</pre>	<pre>opts.CodingRulesCodeMetrics.EnableCertC=1; opts.CodingRulesCodeMetrics.CertC... ='all-rules';</pre>
<pre>opts.CodingRulesCodeMetrics... .EnableMisraC3=1; opts.CodingRulesCodeMetrics... .MisraC3Subset='iso-17961';</pre>	<pre>opts.CodingRulesCodeMetrics.EnableIso17961 =1; opts.CodingRulesCodeMetrics.Iso17961='all';</pre>

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

Product: Polyspace Bug Finder (Desktop).

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 the new CodingRulesCodeMetrics property CertCpp instead.

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

To update your MATLAB code, see this table.

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

Property	Use Instead
<pre>opts.CodingRulesCodeMetrics... .EnableMisraCpp=1; opts.CodingRulesCodeMetrics... .MisraC3Subset='CERT-all';</pre>	<pre>opts.CodingRulesCodeMetrics.EnableCertCpp =1; opts.CodingRulesCodeMetrics.CertC='all';</pre>
<pre>opts.CodingRulesCodeMetrics... .EnableMisraCpp=1; opts.CodingRulesCodeMetrics... .MisraC3Subset='CERT-rules';</pre>	

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

Product: Polyspace Bug Finder (Desktop).

Analysis Results

AUTOSAR C++14 Support: Check for violations of rules from the AUTOSAR C++14 coding standard

In R2019a, Bug Finder can detect violations of rules from the AUTOSAR C++14 coding standard.

Coding Rules		
<input type="checkbox"/> Check MISRA C++ rules	required-rules	Edit
<input type="checkbox"/> Check JSF C++ rules	shall-rules	Edit
<input checked="" type="checkbox"/> Check AUTOSAR C++14 rules	all	Edit

Bug Finder supports a significant number of AUTOSAR C++14 rules. See Supported AUTOSAR C++14 Rules.

This feature applies to both Polyspace Bug Finder and Bug Finder Server.

The AUTOSAR C++14 standard is an improved version of the earlier MISRA C++: 2008 standard and retains only a more relevant subset of MISRA C++: 2008 rules. The AUTOSAR C++14 standard also takes into account later C++ language versions such as C++14 and incorporates elements from other coding standards such as CERT C++ and High Integrity C++ (HIC++). With Bug Finder, you can directly check for violations of rules from the AUTOSAR C++14 standard.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Improved CERT C++ Support: Check for missing overloads, ambiguous declaration syntax and other rules from CERT C++ Coding Standard

In R2019a, you can look for violations of these CERT C++ rules (in addition to previously supported rules).

CERT C++ Rule	Description	Polyspace Checker
DCL52-CPP	Never qualify a reference type with const or volatile	CERT C++: DCL52-CPP
DCL53-CPP	Do not write syntactically ambiguous declarations (most vexing parse)	CERT C++: DCL53-CPP
DCL54-CPP	Overload allocation and deallocation functions as a pair in the same scope	CERT C++: DCL54-CPP

CERT C++ Rule	Description	Polyspace Checker
EXP58-CPP	Pass an object of the correct type to va_start	CERT C++: EXP58-CPP
EXP59-CPP	Use offsetof() on valid types and members	CERT C++: EXP59-CPP

See also CERT C++ Rules.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

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

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

★ Number of Recursions (Value: 1) ?				
This metric shows the number of recursions, both direct and indirect.				
	Event	File	Scope	Line
1	Recursion cycle: operation1 => operation3 => operation4 => operation5	recursion.c	recursion.c	3

The new display of recursion cycles enables the following:

- *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`.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

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`.

New Bug Finder Defect Checkers: Check for misplaced CV qualifiers, C++ most vexing parse, ill-constructed variadic functions, and other issues

In R2019a, you can look for these new types of defects.

Defect	Description
C++ reference type qualified with <code>const</code> or <code>volatile</code>	Reference type declared with a redundant <code>const</code> or <code>volatile</code> qualifier
C++ reference to <code>const</code> -qualified type with subsequent modification	Reference to <code>const</code> -qualified type is subsequently modified
Ambiguous declaration syntax	Declaration syntax can be interpreted as object declaration or part of function declaration
Missing overload of allocation or deallocation function	Only one function in an allocation-deallocation function pair is overloaded
Incorrect type data passed to <code>va_start</code>	Data type of second argument to <code>va_start</code> macro leads to undefined behavior
Incorrect use of <code>va_start</code>	<code>va_start</code> is called in a non-variadic function or called with a second argument that is not the rightmost parameter of a variadic function
Incorrect use of <code>offsetof</code> in C++	Incorrect arguments to <code>offsetof</code> macro causes undefined behavior

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Updated code metrics specifications

In R2019a, these code metric specifications have been updated.

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

Code Metric	Update
Number of Recursions	<p>You can see a change in the number of recursions in your project.</p> <p>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.</p> <p>The metric is also supported with events showing the recursion cycles. For details, see the release note about Recursion Detection.</p>
Number of Paths	<p>You can see a high value of the metric in some cases where the metric value was previously reported as zero.</p> <p>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.</p>

Code Metric	Update
Code complexity metrics for C++ templates	<p>If you use C++ templates, you can see a difference in the value of certain metrics.</p> <p>Each instantiation of a C++ template is considered as a separate function. Code complexity metrics are reported separately for each instantiation.</p> <p>For instance, consider the function template <code>GetMax</code> instantiated twice in <code>main</code>:</p> <pre data-bbox="865 625 1209 1199"> // 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; } </pre> <p>In R2019a, the two instantiations of <code>GetMax</code> are considered as separate functions. All code metrics are reported separately for the two instantiations. Further, the number of called functions in <code>main</code> is 2.</p> <p>Previously, the two instantiations were considered as one.</p>

Code Metric	Update
Sizes of local variables	<p>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.</p> <p>For instance, consider this example:</p> <pre>class A { virtual void f(); }; class B : virtual A { };</pre> <p>Previously, the size of an object of type A was shown as 8 and B as 16. Now both sizes are calculated as 8.</p>

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Compatibility Considerations

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

Updated Bug Finder defect checkers

In R2019a, these defect checkers have been updated.

Defect	Description	Update
Data race including atomic operations	Operations on the same shared variable in two tasks can interrupt each other. All operations on shared variables including atomic operations are considered as potentially nonatomic.	<p>The checker now considers situations where the two tasks have different priorities.</p> <p>For instance, if an atomic operation in a preemptable interrupt reads or writes to the same shared variable as an operation in a nonpreemptable interrupt, the checker can detect this issue. See Define Preemptable Interrupts and Nonpreemptable Tasks.</p>
Incorrect syntax of flexible array member size	Flexible array member defined with size zero or one	The defect checker is disabled if you run a Bug Finder analysis on C90 code (using the option <code>-c-version c90</code>). See C standard version (<code>-c-version</code>).

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Server, Polyspace Bug Finder Access

Compatibility Considerations

If you check for the defects mentioned above, you can see a difference in the number of issues found.

Reviewing Results

Support for Security Standards: See CERT C, CERT C++ or ISO/IEC TS 17961 rule violations explicitly in Polyspace analysis results and reports

In R2019a, if you check for violations of the CERT C, CERT C++ or ISO/IEC TS 17961 standard, the results list and reports show the rules violated as analysis results.

The screenshot displays the Polyspace Bug Finder interface. On the left, the 'Results List' pane shows a tree view of security standards. The 'SEI CERT C' family is expanded, showing various rules with their counts. The 'EXP05-C Do not cast away a const qualification' rule is highlighted, showing it has 9 violations across multiple files, including 'programming.c'. On the right, the 'Result Details' pane shows the selected rule violation. The status is 'Unreviewed' and severity is 'Unset'. The description reads: 'SEI CERT C EXP05-C Do not cast away a const qualification. Invalid pointer cast from 'volatile float *' to 'float *': qualifier is removed.' Below this, a table lists the violation details:

Event	File	Scope	Line
1	programming.c	bug_qualifiermismatch()	194

The 'Source' pane shows the corresponding code in 'programming.c' at line 194:

```

189
190 float bug_qualifiermismatch(void) {
191     float* q;
192
193     if (randoint()) {
194         q = pvf; /* Defect: Qualifier mismatch */
195         return *q;
196     }
197     return 0.0;
198 }

```

You can directly see rules from the security standards in the Bug Finder analysis results and security-specific reports. You can explicitly filter specific rules for a more focused review. Previously, the Bug Finder analysis results contained defects mapped to rules from the standard. In the list of results, you enabled a **CERT ID** or **ISO-17961 ID** column to see the CERT C or ISO/IEC TS 17961 rules corresponding to a defect.


Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Access

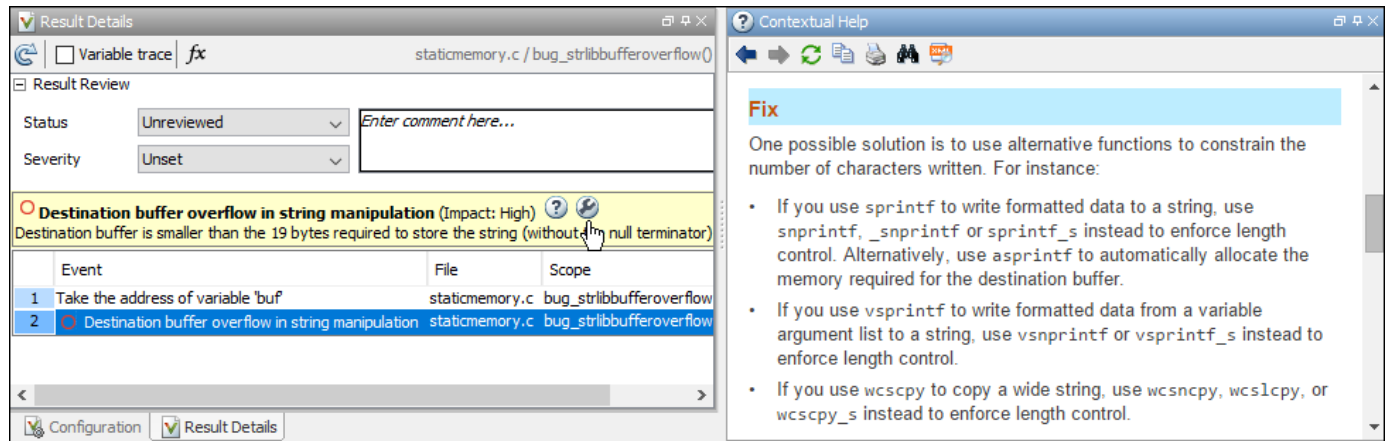
Compatibility Considerations

In previous releases, to review a CERT C or ISO/IEC TS 17961 rule violation, you reviewed defects or MISRA C: 2012 violations that are mapped to these security standards. Now, you can directly check for these standards and review the rule violations.

See also Changes in Coding Standard Checking in R2019a.

Bug Fix Suggestions: See possible fixes for types of defects found by Bug Finder

In R2019a, you can navigate from a defect found with Bug Finder to suggested fixes for the defect. To see these fix suggestions, click the  icon in the details for the defect.



You can implement one of the suggested fixes. You can also use the suggested fixes and code examples for guidance and create your own fixes.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Access

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

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 **Assertion** is selected on the **Results List** pane. The corresponding source code (line 60) 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 `=` operator in line 77), the selection in the results list and the details still show the **Assertion** result.

The screenshot displays the Polyspace Bug Finder interface. On the left, the 'Results List' pane shows a table of 30 error types, with 'Assertion' selected. The table columns are 'Check', 'Impact', and 'File'. The 'Assertion' row is highlighted in blue, indicating it is the selected result.

Check	Impact	File
Absorption of float operand	High	numerical.c
Accessing object with temporary lifetime	Low	programming.c
Alignment changed after memory reall...	Low	dynamicmemory
Alternating input and output from a st...	Low	programming.c
Array access out of bounds	High	staticmemory.c
Assertion	High	programming.c
Buffer overflow from incorrect string f...	High	staticmemory.c
Call through non-prototyped function ...	Medium	programming.c
Character value absorbed into EOF	High	programming.c
Closing previously closed resource	High	resourcemanag
Data race	High	concurrency.c
Data race through standard library fu...	High	concurrency.c
Dead code	Low	dataflow.c
Deadlock	High	concurrency.c
Deallocation of previously deallocated ...	High	dynamicmemory
Declaration mismatch	High	programming.c
Destination buffer overflow in string m...	High	staticmemory.c
Destination buffer underflow in string ...	High	staticmemory.c
Double lock	High	concurrency.c
Double unlock	High	concurrency.c
Environment pointer invalidated by pr...	Medium	programming.c
Errno not reset	High	programming.c
Float conversion overflow	High	numerical.c
Float division by zero	High	numerical.c
Format string specifiers and argument...	Low	programming.c
Improper array initialization	Medium	programming.c
Improper array initialization	Medium	programming.c
Incorrect data type passed to va_arg	Medium	programming.c
Incorrect pointer scaling	Medium	programming.c
Inline constraint not respected	Medium	programming.c
Integer conversion overflow	High	numerical.c
Integer division by zero	High	numerical.c
Invalid assumptions about memory org...	Medium	programming.c
Invalid free of pointer	High	dynamicmemory
Invalid use of = operator	Medium	programming.c

On the right, the 'Result Details' pane shows the details for the selected 'Assertion' error. It indicates 'Assertion fails.' and provides a list of events: '1 Assignment to local variable 'x' p' and '2 Assertion'. Below this, the 'Source' pane shows the corresponding code in 'programming.c'. The code snippet is as follows:

```

57  /*=====
58  int bug_assert(void
59      int x = 0;
60      assert(x == 12)
61
62      return x;
63  }
64
65  int corrected_assert
66      int x = 12;
67      assert(x == 12)
68
69      return x;
70  }
71
72
73  /*=====
74  * BAD EQUAL USE
75  *=====
76  void bug_badequalus
77      if (a == b) {
78          (void)print
79      }
80  }

```

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.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Access

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

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

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.

Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Access

Report Generation: Generate single file for HTML reports

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

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.

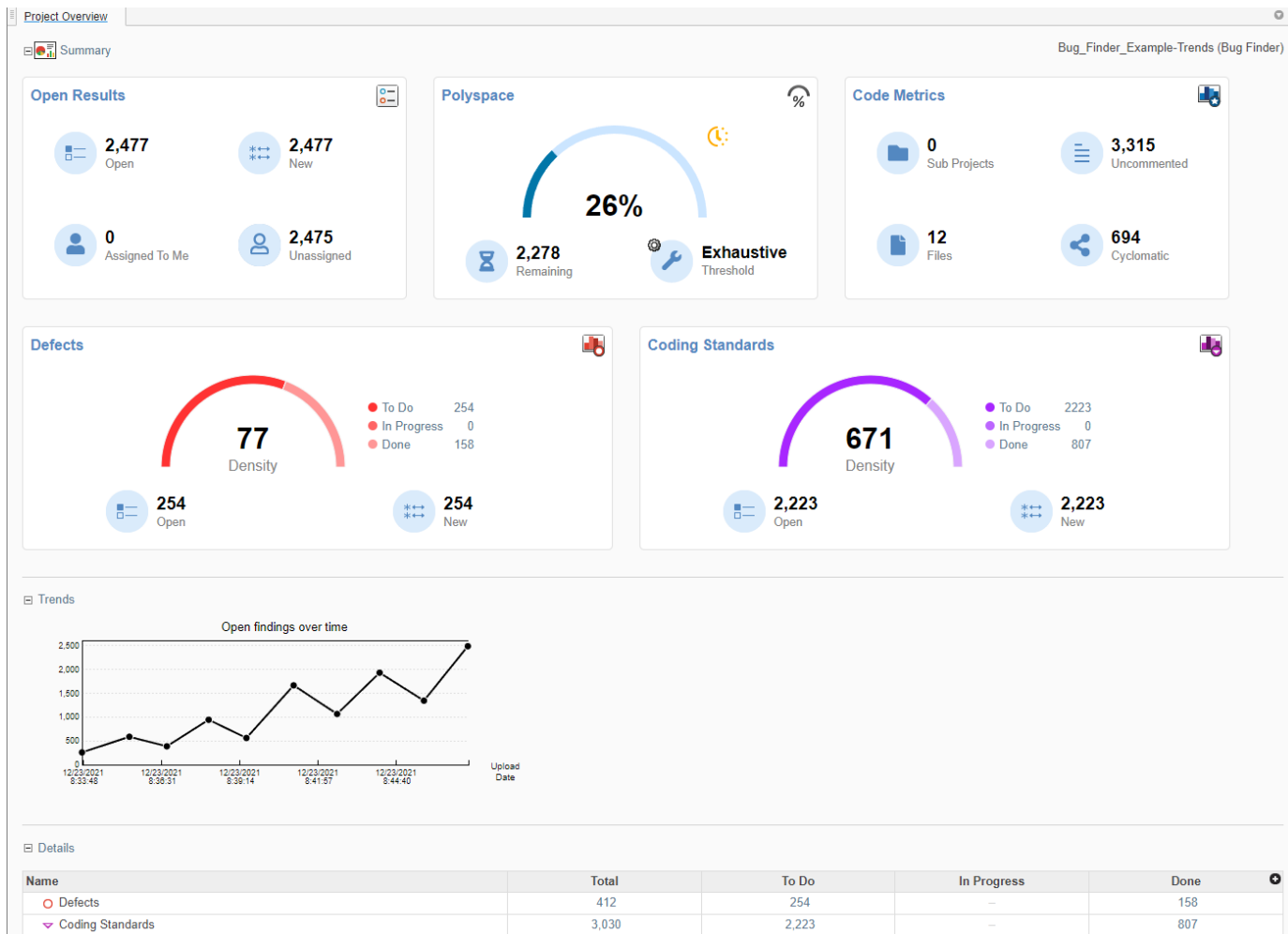
Products: Polyspace Bug Finder (Desktop), Polyspace Bug Finder Access

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.

Project Dashboard : Track progress of code quality via Polyspace results

In R2019a, you can track the progress of the code quality of your projects using the new intuitive Polyspace Bug Finder Access **DASHBOARD**. When an analysis run is uploaded to the Polyspace Access database, the dashboard updates to give a snapshot of the findings, including a progress trend for number of findings compared to previous runs.



The **DASHBOARD** perspective of the Polyspace Access web interface allows you to:

- *Prioritize reviews:* See new and open issues that have not been fixed or justified, then open a detailed results list for just those issues. You can drill down on a set of findings filtered by new, open, unassigned, by family of findings, or by file.
- *Aggregate results for multiple projects:* If your team works on multiple projects, move all the projects under an umbrella project and view a snapshot of the code quality for all your team's projects.
- *Authenticate client access:* The web interface is behind a login. Only users with a Polyspace Bug Finder Access license and the appropriate credentials can view the dashboard from their web browser.

Product: Polyspace Bug Finder Access.

Collaborative Review Support : Review Polyspace Bug Finder results and source code in web browser

In R2019a, review Polyspace analysis findings and view the findings in your source code using the new Polyspace Bug Finder Access **REVIEW** web interface. You do not need to install a Polyspace product on your machine to open and review analysis results.

The screenshot displays the Polyspace Bug Finder Access REVIEW web interface. The interface is divided into several sections:

- Top Navigation:** Includes tabs for Dashboard, Run-time Checks, Defects, Coding Standards, Code Metrics, and Global Variables. There are also filters for 'To Do', 'In Progress', and 'Done', and options to 'Show only' and 'Filter out' (both set to 'Comment, filename, etc.').
- Left Panel:** A vertical sidebar with sections for PROJECT EXPLORER, PROJECT DETAILS, FILE EXPLORER, and SUPPORT REPORT. The main area shows a 'Results List' table with columns for Family, ID, Type, Group, and Check.
- Table Data (Results List):**

Family	ID	Type	Group	Check
○	40427	Defects	Static memory	Buffer overflo
○	40461	Defects	Programming	Possibly unin
○	40464	Defects	Programming	Invalid use of
○	40482	Defects	Programming	Wrong type u
○	47905	Defects	Programming	Declaration n
○ *	47907	Defects	Programming	Typedef misn
○	47910	Defects	Concurrency	Data race
○	47912	Defects	Dynamic memory	Deallocation
○	47922	Defects	Resource management	Resource lea
○	47925	Defects	Static memory	Pointer or ref
○	47928	Defects	Data flow	Non-initialize
○	47934	Defects	Data flow	Non-initialize
○	47937	Defects	Data flow	Non-initialize
○	47959	Defects	Dynamic memory	Use of previo
○	47962	Defects	Dynamic memory	Invalid free of
○	47965	Defects	Numerical	Invalid use of
○	47968	Defects	Numerical	Invalid use of
○	47971	Defects	Numerical	Float convers
○	47974	Defects	Numerical	Integer conve
○	47977	Defects	Numerical	Absorption of
○	47986	Defects	Numerical	Invalid use of
○	48004	Defects	Programming	Character val
○	48007	Defects	Programming	Variable leng
○ *	48022	Defects	Programming	Assertion
○	48027	Defects	Programming	Erno not res
○	48030	Defects	Programming	Invalid use of
○	48032	Defects	Programming	Misuse of err
○	48035	Defects	Programming	Writing to cor
○	48040	Defects	Programming	Possible mist
○	48044	Defects	Programming	Invalid va_list
○	48047	Defects	Resource management	Use of previo
○	48050	Defects	Resource management	Closing previ
○ *	48053	Defects	Resource management	Writing to rea
○	48056	Defects	Static memory	Array access
○	48059	Defects	Static memory	Invalid use of
○	48062	Defects	Static memory	Subtraction o
○	48065	Defects	Static memory	Destination b
○	48068	Defects	Static memory	Use of autom
- Center Panel (Result Details):** Shows details for the selected defect (ID 40482). It includes a 'Status' dropdown (set to 'To Fix'), a 'Severity' dropdown (set to 'Unset'), and an 'Assigned to' field. A text area for 'Enter your comment here...' is also present. Below this, there is a yellow warning box: 'Wrong type used in sizeof (Impact: High)'. The text below the warning reads: 'The type 'char *' used for the block of memory is not a pointer to the type 'char *' used in sizeof.' There is also a table with columns 'Event', 'File', and 'Scope'.
- Right Panel (Source Code):** Displays the source code for the file 'programming.c'. The code shows a function 'bug_ptrsizeofmismatch()' and a corrected version 'corrected_ptrsizeofmismatch()'. The corrected version uses 'sizeof(char)' instead of 'sizeof(char*)' in the malloc call.

The **REVIEW** perspective of the Polyspace Access web interface:

- *Facilitates 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.
- *Authenticates client access:* The web interface is behind a login. Only users with a Polyspace Bug Finder Access license and the appropriate credentials can view the results from their web browser.

Product: Polyspace Bug Finder Access.

Collaborative Review Support : Share Polyspace Bug Finder results using web links

In R2019a, you can right-click an analysis result in the Polyspace Bug Finder Access interface to obtain a URL that you can share with other team members. The link that you provide opens the Polyspace Bug Finder Access interface and displays the finding along with the corresponding source code.

The image shows two screenshots of the Polyspace Bug Finder Access interface. The left screenshot shows a 'REVIEW' tab with a 'Results List' table. A right-click context menu is open over the row with ID 61582. The right screenshot shows the 'Result Details' view for finding ID 61582, displaying the finding description and the corresponding source code snippet.

Family	ID	Type	Group
	61576	Defect	Programming
	61581	Defect	Programming
	61582	Defect	Programming
	61594	Defect	Programming
	61597	Defect	Programming
	61598	Defect	Programming
	61599	Defect	Dynamic memory

```

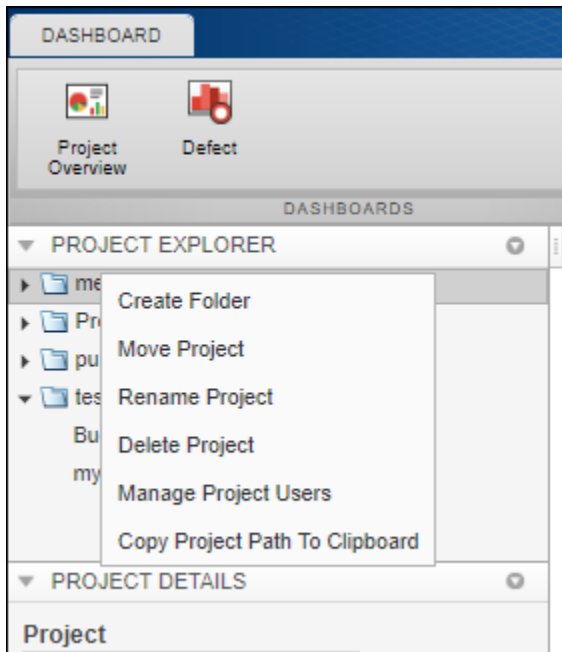
503 * OPERATOR PRECEDENCE
504 *=====
505 int bug_operatorprecedence(int
506     int res = a-b;
507     if (a < b & c)
508         res = c;
509

```

Product: Polyspace Bug Finder Access.

Project Authorization Management: Create and enforce authorization policies for access to project

In R2019a, you can manage project users in Polyspace Bug Finder Access by right-clicking a project in the **PROJECT EXPLORER** and assigning roles to member of your team. The roles authorize or forbid users from viewing projects.



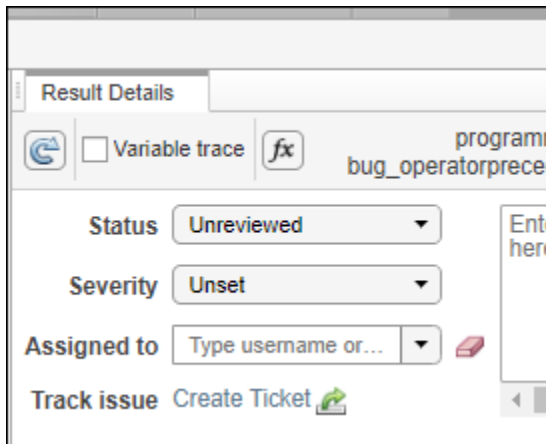
The ability to authorize access to projects allows you to:

- *Restrict access to your source code:* Use the authorization policy to restrict who can view the source code you upload with your analysis results.
- *Display relevant projects only:* When they log in to Polyspace Access, users can only see projects for which they are administrators, owners, or contributors. Use the authorization policy so that team members only see projects that they are working on.

Product: Polyspace Bug Finder Access.

Bug Tracking Tool Support : Create JIRA issues for Polyspace Bug Finder results

In R2019a, Polyspace Bug Finder Access supports integration with the JIRA software. If you have an instance of the JIRA software, after you configure Polyspace Bug Finder Access, you can create a JIRA ticket to track Polyspace findings. The ticket is populated with details of the finding and a link to open that finding in Polyspace Access. You can add the ticket to any existing JIRA project.



Once you create a ticket, the **Result Details** pane in the Polyspace Bug Finder Access web interface displays a link to the corresponding JIRA issue.

Product: Polyspace Bug Finder Access.

R2018b

Version: 2.6

New Features

Bug Fixes

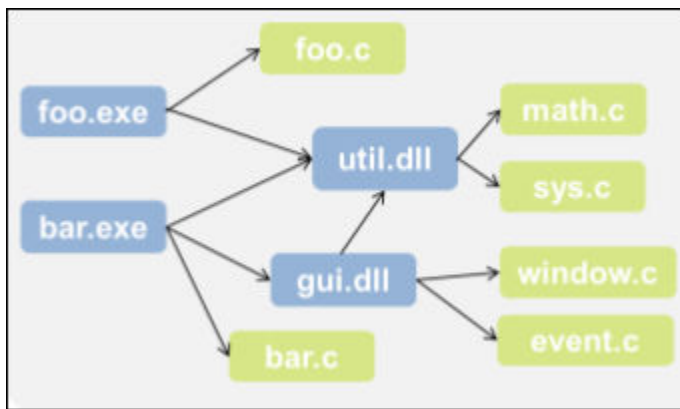
Compatibility Considerations

Analysis Setup

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

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:

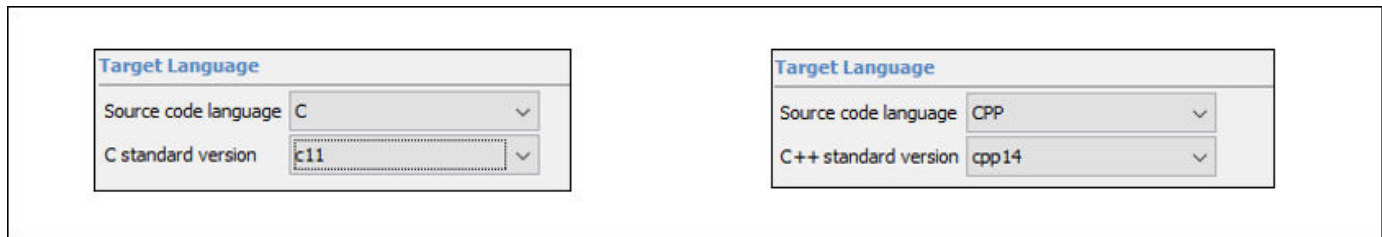
- Modularize Polyspace Analysis by Using Build Command
- `polyspace-configure`

The ability to generate configuration modules from build system has the following 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 codebase.
- *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

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



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

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

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.

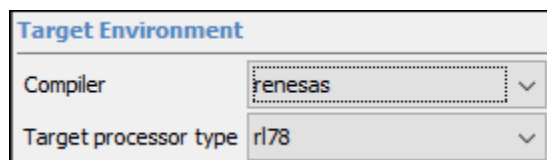
See also *Auto-Detection of Thread Creation and Critical Section in Polyspace*.

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

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: `r178`, `rh850`, or `rx`. See *Renesas Compiler (-compiler renesas)*.



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.

Changes in analysis options and binaries

Polyspace Bug Finder has new Target & Compiler options

Behavior change

Polyspace Bug Finder 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.

-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 Renesas compiler and compiler-specific language extensions such as keywords and pragmas.

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

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](#).

TargetCompiler property has a new Compiler option value renesas

Behavior change

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

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
<code>opts.Configuration.TargetCompiler... .NoLanguageExtensions = true;</code>	<code>opts.Configuration.TargetCompiler... .CVersion = 'c90';</code>
<code>opts.Configuration.TargetCompiler... .Cpp11Extension = true;</code>	<code>opts.Configuration.TargetCompiler... .CppVersion = 'cpp11';</code>

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](#).

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`.

Analysis Results

CERT C++ Support: Identify CERT C++ violations by using defect checkers and coding rules

In R2018b, you can look for violations of these CERT C++ rules and CERT C rules that apply to C++. For a list of all Polyspace results that correspond to CERT C++ violations, see CERT C++ Coding Standard and Polyspace Results.

CERT C++ Rule	Description	Polyspace Checker
CON54-CPP	Wrap functions that can spuriously wake up in a loop	Function that can spuriously wake up not wrapped in loop
EXP57-CPP	Do not cast or delete pointers to incomplete classes	Conversion or deletion of incomplete class pointer
OOP58-CPP	Copy operations must not mutate the source object	Copy operation modifying source operand
CON37-C	Do not call signal() in a multithreaded program	Signal call in multithreaded program
CON40-C	Do not refer to an atomic variable twice in an expression	Atomic load and store sequence not atomic Atomic variable accessed twice in an expression
CON41-C	Wrap functions that can fail spuriously in a loop	Function that can spuriously fail not wrapped in loop
EXP46-C	Do not use a bitwise operator with a Boolean-like operand	Possible invalid operation on boolean operand
FIO32-C	Do not perform operations on devices that are only appropriate for files	Inappropriate I/O operation on device files
FLP36-C	Preserve precision when converting integral values to floating-point type	Precision loss in integer to float conversion
INT30-C	Ensure that unsigned integer operations do not wrap	Unsigned integer constant overflow
INT32-C	Ensure that operations on signed integers do not result in overflow	Integer constant overflow

CERT C++ Rule	Description	Polyspace Checker
INT35-C	Use correct integer precisions	Integer precision exceeded Possible invalid operation on boolean operand
PRE31-C	Avoid side effects in arguments to unsafe macros	Side effect in arguments to unsafe macro
STR37-C	Arguments to character-handling functions must be representable as an unsigned char	Misuse of sign-extended character value
STR38-C	Do not confuse narrow and wide character strings and functions	Misuse of narrow or wide character string

Improved CERT C Support: Check for precision loss, blocking operations, and other rules from the CERT C Coding Standard

In R2018b, you can look for violations of these CERT C rules (in addition to previously supported rules).

CERT C Rule	Description	Polyspace Checker
CON05-C	Do not perform operations that can block while holding a lock	Blocking operation while holding lock
CON30-C	Clean up thread-specific storage	Thread-specific memory leak
CON36-C	Wrap functions that can spuriously wake up in a loop	Function that can spuriously wake up not wrapped in loop
CON37-C	Do not call signal() in a multithreaded program	Signal call in multithreaded program
CON40-C	Do not refer to an atomic variable twice in an expression	Atomic load and store sequence not atomic Atomic variable accessed twice in an expression
CON41-C	Wrap functions that can fail spuriously in a loop	Function that can spuriously fail not wrapped in loop
DCL38-C	Use the correct syntax when declaring a flexible array member	Incorrect syntax of flexible array member size
EXP46-C	Do not use a bitwise operator with a Boolean-like operand	Possible invalid operation on boolean operand

CERT C Rule	Description	Polyspace Checker
FIO32-C	Do not perform operations on devices that are only appropriate for files	Inappropriate I/O operation on device files
FLP36-C	Preserve precision when converting integral values to floating-point type	Precision loss from integer to float conversion
INT35-C	Use correct integer precisions	Integer precision exceeded Possible invalid operation on boolean operand
POS44-C	Do not use signals to terminate threads	Use of signal killing thread
POS52-C	Do not perform operations that can block while holding a POSIX lock	Blocking operation while holding lock
PRE31-C	Avoid side effects in arguments to unsafe macros	Side effect in arguments to unsafe macro
STR37-C	Arguments to character-handling functions must be representable as an unsigned char	Misuse of sign-extended character value
STR38-C	Do not confuse narrow and wide character strings and functions	Misuse of narrow or wide character string

See also Mapping Between CERT C Rules and Polyspace Results.

Constant Overflows: Check for overflows on integer constants

In R2018b, you can check for instances where a compile-time constant is assigned to a variable whose data type cannot accommodate the value.

For instance, `c` is an 8-bit signed char variable that cannot hold the value 255.

```
signed char c = 255;
```

See `Integer constant overflow` and `Unsigned integer constant overflow`.

Most compilers wrap around overflowing constants with a warning. However, if you want to check for these instances, you can enable the constant overflow checkers in Bug Finder.

Updated Bug Finder defect checkers

In R2018b, these defect checkers have been updated.

Defect	Description	Update
Write without a further read	A variable is never read after assignment	The checker now detects redundant write operations on <i>global variables</i> . For instance, you perform two write operations on a global variable without an intermediate read operation. The first write operation is redundant.
Misuse of sign-extended character value	Data type conversion with sign extension causes unexpected behavior.	The checker now detects use of sign-extended plain char variables as argument to a character-handling function.

For new Bug Finder checkers, see the release notes about CERT C and CERT C++ support.


Changes to coding standards checking

In R2018b, the following changes have been made in checking of previously supported MISRA C rules.

Rule	Description	Improvement
MISRA C:2012 Rule 2.2	There shall be no dead code.	The rule checker now flags redundant write operations on <i>global variables</i> . For instance, you perform two write operations on a global variable without an intermediate read operation. The first write operation is redundant.
MISRA C:2012 Rule 10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.	The checker now flags assignments to a boolean variable if the assigned value has a non-boolean essential type.
MISRA C++:2008 Rule 5-0-15	Array indexing shall be the only form of pointer arithmetic.	The checker does not flag array indexing on pointers that point to array variables.

Reviewing Results

Function Call Hierarchy: View call tree of functions in source code

In R2018b, you can view information about the call tree of functions in your source code by opening the **Call Hierarchy** pane. To open this pane click the  icon in the **Result Details** pane.

```

19
20 int main(void)
21 {
22     pthread_t thread_increment;
23     pthread_t thread_get;
24
25     pthread_create(&thread_increment, ((void *)0), increment_count, ((void *)0));
26     pthread_create(&thread_get, NULL, set_count, NULL);
27

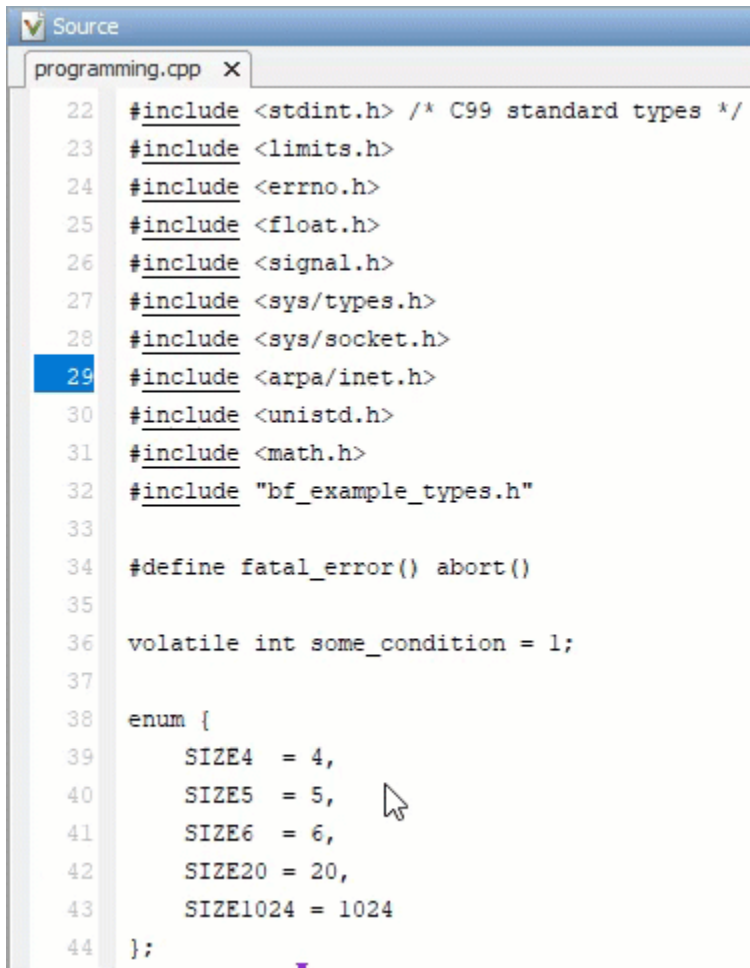
```

fx Call Hierarchy			
Calls	File	Line	Stubbed
main()	quick_test.c	20	
▶ pthread_create()	quick_test.c	25	Std library
▶ task_main:thread_increment()	quick_test.c	25	Std library
▶ increment_count()	quick_test.c	25	
▶ pthread_create()	quick_test.c	26	Std library
▶ task_main:thread_get()	quick_test.c	26	Std library
▶ set_count()	quick_test.c	26	
▶ pthread_join()	quick_test.c	28	
▶ pthread_join()	quick_test.c	29	

For a function `foo` in your source code, you can see functions and tasks that call `foo` (callers), and those called by `foo` (callees).

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

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 X
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.

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

R2018a

Version: 2.5

New Features

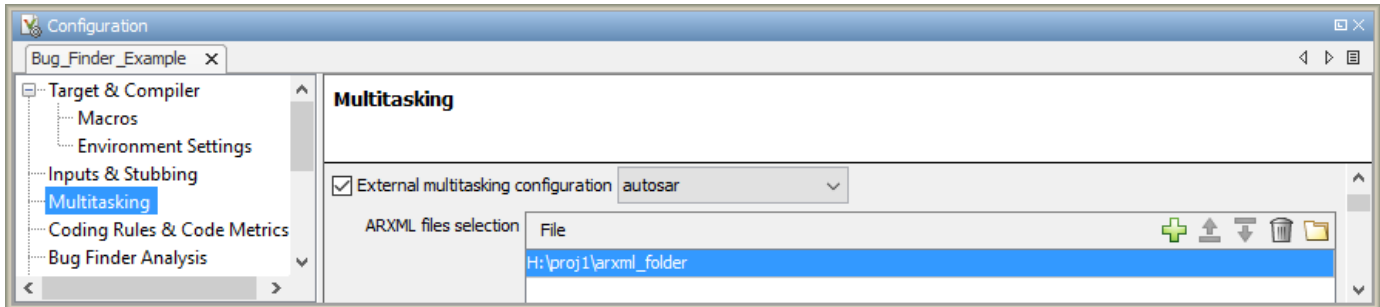
Bug Fixes

Compatibility Considerations

Analysis Setup

AUTOSAR Support: Set up Polyspace multitasking configuration automatically from an AUTOSAR description

In R2018a, Polyspace can parse your AUTOSAR specifications (.arxml files) to determine your multitasking configuration.



This feature supports AUTOSAR XML schema for releases 4.0 and later.

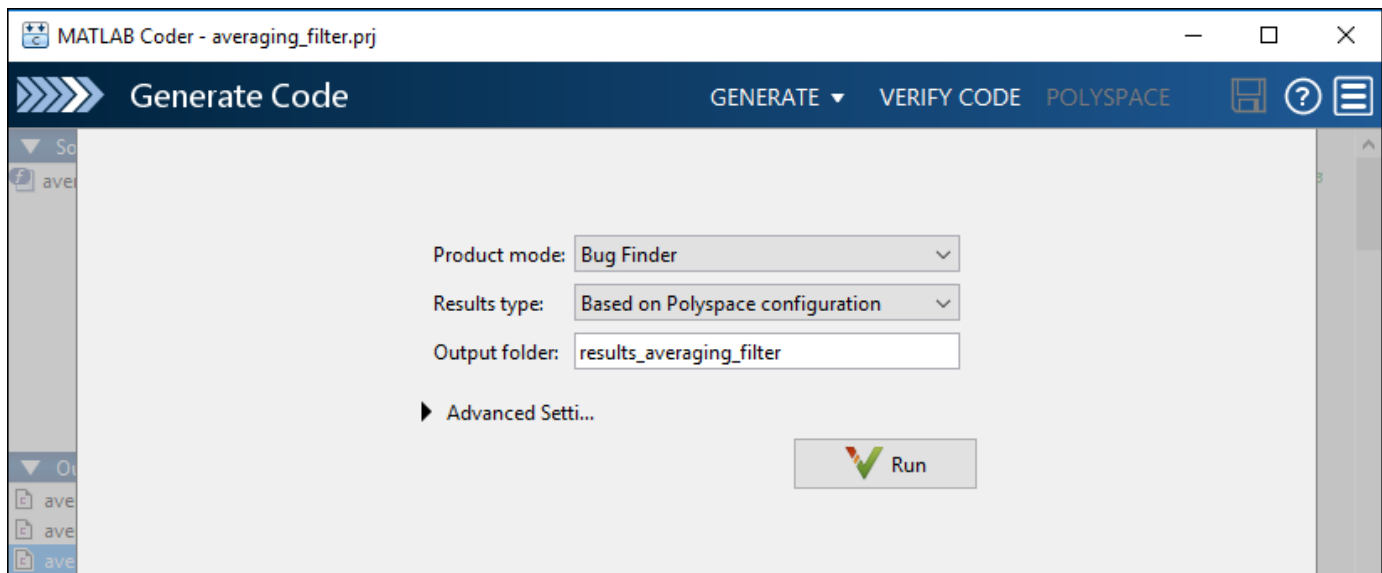
For more information, see [ARXML files selection \(-autosar-multitasking\)](#).

The creation of multitasking configuration from AUTOSAR description has the following benefits:

- *Automatic configuration:* You do not need to specify your multitasking configuration manually. Polyspace can determine the tasks, interrupts and critical sections from your AUTOSAR specifications (specifically, the ECUC-CONTAINER-VALUE element).
- *Minimal knowledge required for setup:* You do not need to know the details of the AUTOSAR specifications for configuring a Polyspace analysis. You simply provide the folder containing your .arxml files.

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

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

The ability to analyze code generated with MATLAB Coder has the following 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. 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

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.

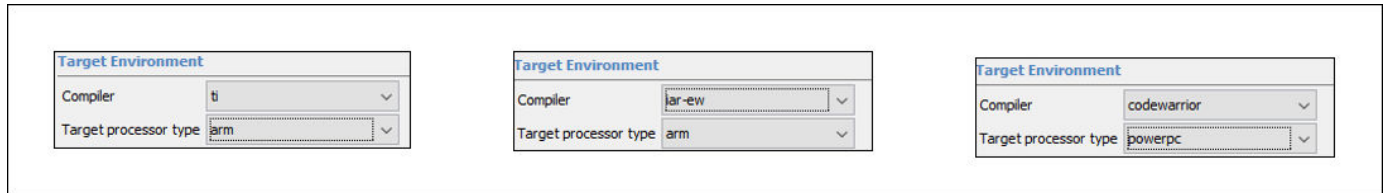
See IAR Embedded Workbench Compiler (-compiler iar-ew).

- CodeWarrior

You can specify these target processors: s12z or powerpc.

See 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.



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

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:

- The screenshot shows the 'Target Environment' dialog box with 'Compiler' set to 'gnu5.x' and 'Target processor type' set to 'i386'.

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

- The screenshot shows the 'Target Environment' dialog box with 'Compiler' set to 'gnu6.x' and 'Target processor type' set to 'i386'.

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.

- The screenshot shows the 'Target Environment' dialog box with 'Compiler' set to 'clang3.x' and 'Target processor type' set to 'i386'.

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`

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`.

The ability to include or exclude sources when using `polyspace-configure` has the following 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 Bug Finder 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 Bug Finder 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 (<code>-compiler</code>)	<ul style="list-style-type: none"> • New value <code>ti</code> added. See Compiler Support release note. • New value <code>iar-ew</code> added. See Compiler Support release note. <p>Use this value to emulate IAR compilers.</p> <p>For older Polyspace projects, you can still use option value <code>iar</code>.</p> <ul style="list-style-type: none"> • New value <code>codewarrior</code> added. See Compiler Support release note. • New value <code>gnu5.x</code> added. See Updated GCC and Clang Compiler Support release note. • New value <code>gnu6.x</code> added. See Updated GCC and Clang Compiler Support release note. • New value <code>clang3.x</code> 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
<code>-compiler clang3.5</code>	<code>-compiler clang3.x</code>

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
<code>-compiler iso</code>	<code>-compiler generic</code>
<code>-compiler none</code>	
<code>-compiler gnu</code>	<code>-compiler gnu3.4</code>
<code>-compiler visual</code>	<code>-compiler visual10.0</code>
<code>-compiler visual6</code>	
<code>-compiler visual7.0</code>	
<code>-compiler visual7.1</code>	
<code>-compiler visual8</code>	
<code>-compiler visual10</code>	

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

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.

-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
<code>-enum-type-definition defined-by-standard</code>	<code>-enum-type-definition defined-by-compiler</code>

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	<ul style="list-style-type: none"> • New value <code>ti</code> added. See Compiler Support release note. • New value <code>iar-ew</code> added. See Compiler Support release note. <p>Use this value to emulate IAR compilers.</p> <p>For older Polyspace projects, you can still use property value <code>iar</code>.</p> <ul style="list-style-type: none"> • New value <code>codewarrior</code> added. See Compiler Support release note. • New value <code>gnu5.x</code> added. See Updated GCC and Clang Compiler Support release note. • New value <code>gnu6.x</code> added. See Updated GCC and Clang Compiler Support release note. • New value <code>clang3.x</code> 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	<pre>opts.Configuration.Multitasking... .EnableExternalMultitasking=1; opts.Configuration.Multitasking... .ExternalMultitaskingType='osek';</pre>

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	<pre>opts.Configuration.TargetCompiler... .WcharTTypeIs="unsigned-long"</pre>

Property	Description
<code>opts.Configuration.TargetCompiler... .SizeTIsUnsignedLong</code>	<code>opts.Configuration.TargetCompiler... .SizeTTypeIs="unsigned-long"</code>

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
<code>opts.Configuration.TargetCompiler... .EnumTypeDefinition="defined-by-dialect"</code>	<code>opts.Configuration.TargetCompiler... .EnumTypeDefinition="defined-by-compiler"</code>

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

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

Analysis Results

CERT C Support: Check for information leakage, invalid environment pointers, and other rules from the CERT C Coding Standard

In R2018a, you can look for violations of these CERT C rules (in addition to previously supported rules).

CERT C Rule	Description	Polyspace Checker
DCL39-C	Avoid information leakage when passing a structure across a trust boundary	Information leak via structure padding
ENV31-C	Do not rely on an environment pointer after following an operation that may invalidate it	Environment pointer invalidated by previous operation
ERR32-C	Do not rely on indeterminate values of <code>errno</code>	Misuse of <code>errno</code> in a signal handler
EXP35-C	Do not modify objects with temporary lifetime	Accessing object with temporary lifetime
EXP44-C	Do not rely on side effects in operands to <code>sizeof</code> , <code>_Alignof</code> , or <code>_Generic</code>	Side effect of expression ignored
EXP47-C	Do not call <code>va_arg</code> with argument of the incorrect type	Incorrect data type passed to <code>va_arg</code> Too many <code>va_arg</code> calls for current argument list
FIO41-C	Do not call <code>getc()</code> , <code>putc()</code> , <code>getwc()</code> , or <code>putwc()</code> with a stream argument that has side effects	Stream argument with possibly unintended side effects
FLP37-C	Do not use object representations to compare floating-point values	Memory comparison of float-point values
MSC38-C	Do not treat a predefined identifier as an object if it might only be implemented as a macro	Predefined macro used as object
MSC40-C	Do not violate constraints	Inline constraint not respected
PRE30-C	Do not create a universal character name through concatenation	Universal character name from token concatenation
PRE32-C	Do not use preprocessor directives in invocations of function-like macros	Preprocessor directive in macro argument

See also Mapping Between CERT C Rules and Polyspace Results.

Cryptography Checkers: Check for security vulnerabilities such as incorrect use of public key cryptography routines

In R2018a, using Bug Finder defects, you can identify incorrect use of public key cryptography routines from the OpenSSL library.

The software detects the following issues with your use of cryptography routines.

Public key cryptography

Defect	Issue Detected
Context initialized incorrectly for cryptographic operation	Context used for cryptography operation is initialized for a different operation. For instance, you mix up encryption and decryption.
Incorrect key for cryptographic algorithm	Cryptography operation is not supported by the algorithm used in context initialization. For instance, you use the DSA algorithm for encryption.
Missing data for encryption, decryption or signing operation	Data provided for cryptography operation is NULL or data length is zero.
Missing parameters for key generation	Context used for key generation is associated with NULL parameters or not associated with parameters at all.
Missing peer key	Context used for shared secret derivation is associated with a NULL peer key or not associated with a peer key at all.
Missing private key	Context used for cryptography operation is associated with a NULL private key or not associated with a private key at all.
Missing public key	Context used for cryptography operation is associated with a NULL public key or not associated with a public key at all.
Nonsecure parameters for key generation	Context used for key generation is associated with weak parameters, for instance, insufficient parameter length.

RSA algorithm specific

Defect	Issue Detected
Incompatible padding for RSA algorithm operation	Cryptography operation is not supported by the padding type set in context.
Missing blinding for RSA algorithm	Context used in decryption or signature verification is not blinded against timing attacks.
Missing padding for RSA algorithm	Context used in encryption or signing operation is not associated with any padding.

Defect	Issue Detected
Nonsecure RSA public exponent	Context used in key generation is associated with a low exponent value.
Weak padding for RSA algorithm	Context used in encryption or signing operation is associated with an insecure padding type.

Hash functions

Defect	Issue Detected
Context initialized incorrectly for digest operation	Context used for digest operation is initialized for a different digest operation. For instance, you mix up signing and signature verification.
Nonsecure hash algorithm	Context used for message digest creation is associated with a weak algorithm.

SSL/TLS connections

Defect	Issue Detected
Nonsecure SSL/TLS protocol	Context used for handling SSL/TLS connections is not associated with a weak protocol.

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

In R2018a, you can look for violations of these MISRA C++ rules (in addition to previously supported 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.

Rule	Description
17-0-3	The names of standard library functions shall not be overridden.

See also MISRA C++ Coding Rules.

MISRA C:2012 Directive 4.8: Detect opportunities for data hiding

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, the implementation of the structure must be hidden in that unit.

See MISRA C:2012 Directive 4.8.

Using this checker, 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

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.

Improved Fast Analysis: Find some multi-file MISRA C violations in fast analysis

In R2018a, if you run fast analysis, the analysis also looks for these MISRA C violations that involve checking multiple files:

- MISRA C: 2004: Rules 8.8 and 8.9.
- MISRA C: 2012: Rules 8.5 and 8.6.

For more information, see `Use fast analysis mode for Bug Finder`.

You detect more violations in the fast analysis mode. Previously, fast analysis looked only for defects and coding rule violations that involved single files or functions.

Reviewing Results

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

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**.

Entry point	Set by
Interrupts (2)	
i1() Executes repeatedly after the main entry point completes	Manually configured
i2() Executes repeatedly after the main entry point completes	Manually configured
Preemptable interrupts (2)	
Non-preemptable tasks (4)	
Tasks (12)	
ct1() Executes repeatedly after the main entry point completes	Manually configured
ct2() Executes repeatedly after the main entry point completes	Manually configured
dt1() Starts in main at line 113	Automatically detected
dt3a() (11 instances) Starts 10 times in main at line 128 Starts in main at line 122	Automatically detected Automatically detected

The new display of concurrency modelling enables the following:

- *Easy spot-check for concurrency modelling:* 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.
- *Determination of priorities:* The entry points in this view are grouped in the order of priorities: interrupts, preemptable interrupts, non-preemptable tasks, (preemptable) tasks. To understand

why a data race does not occur between two entry points (Bug Finder), you can check if one of the entry points has lower priority than the other. See Data race.

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

Data Races: Distinguish write-write conflicts from more benign read-write conflicts

In R2018a, you can choose to review only data races that come from conflicts between two write operations.

The result details message for these data races have an additional line: Variable value may be altered by write-write concurrent access. Use the **Detail** column filters on the **Results List** pane to show only the data races that have this additional line.

Data race (Impact: High)
 Certain operations on variable 'bad_glob2' can interfere with each other and cause unpredictable values.
 Variable value may be altered by write-write concurrent accesses.

Access	Access Protections	Task	File	Scope	Line
Write (Non atomic) Operation might involve multiple machine instructions	No protection	bug_datarace_task3()	concurrency.c	bug_datarace_task3()	88
Write (Non atomic) Operation might involve multiple machine instructions	No protection	bug_datarace_task4()	concurrency.c	bug_datarace_task4()	93

Results List
 All results | New | Showing 92/92

.. Check | Detail | Information | File

- (All)
- (Custom...)
- An if (expression) construct shall be followed by a compound statement.
- Certain calls to function 'setlocale' can interfere with each other and cause unpredictable results.
- Certain operations on variable 'bad_glob1' can interfere with each other and cause unpredictable values.
- Certain operations on variable 'bad_glob2' can interfere with each other and cause unpredictable values. Variable value may be altered by write-write concurrent accesses.
- Function 'bug_badlock_task' has no visible prototype at definition.

OK Cancel

See also Data race.

Conflicts between two write operations in different threads can lead to corruption of memory and indeterminate results. You can now distinguish these conflicts from more benign conflicts between a write and read operation.

R2017b

Version: 2.4

New Features

Bug Fixes

Compatibility Considerations

Analysis Setup

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

If you build your source code with the Green Hills® MULTI 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.

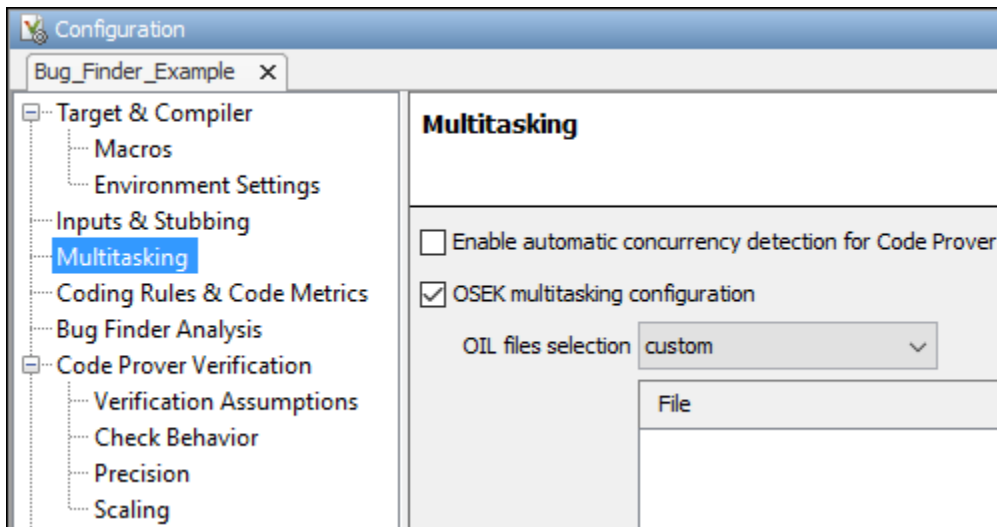
You can specify these target processors directly: `arm64`, `arm`, `i386`, `x86_64`, `powerpc`, `powerpc64`, `rh850` or `tricore`. See `Green Hills Compiler (-compiler greenhills)`.

Target Environment	
Compiler	greenhills
Target processor type	powerpc

You can now set up a Polyspace project without knowing the internal workings of your MULTI 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

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.

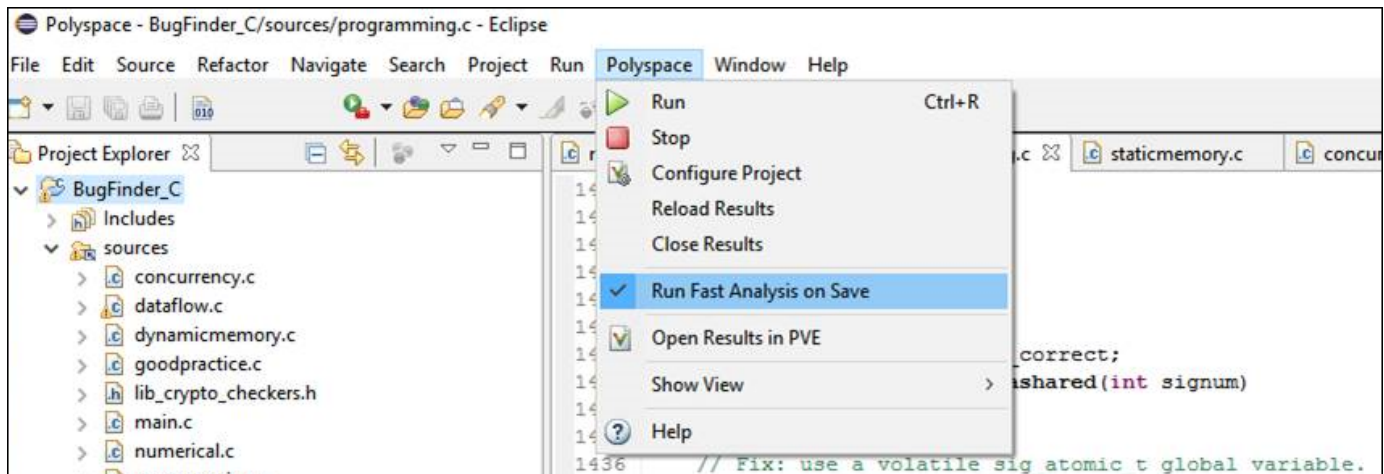


For more information, see `OSEK multitasking configuration (-osek-multitasking)`.

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

Incremental Analysis in Eclipse: Detect bugs as you type and save code in your Eclipse IDE

In R2017b, if you install the Polyspace plugin in your Eclipse IDE, the analysis runs each time you save your code.



You do not have to launch the Polyspace analysis explicitly. You can detect bugs during coding.

Additional Considerations

- *What types of bugs does the analysis look for?*

The analysis looks for the defects that can be quickly detected. You get the same results as if you had specified the option Use fast analysis mode for Bug Finder (`-fast-analysis`).

If you want to look for other kinds of defects, specify the defect checkers in your configuration and launch the analysis explicitly. See [Run Polyspace Analysis in Eclipse](#).

- *Can I disable the automatic analysis?*

You can enable or disable the automatic analysis. Select or clear **Polyspace > Run Fast Analysis on Save**.

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

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

```
obj = polyspace.Project

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

% Run analysis
bfStatus = obj.run('bugFinder');

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

For more information, see `polyspace.Project`.

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 these 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', 'Bug_Finder_Example', 'sources', 'numerical.c')};
opts.ResultsDir = resultsFolder;

polyspaceBugFinder(opts);

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

Use:

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

obj = polyspace.Project;
obj.Configuration.Sources = {fullfile(matlabroot, 'polyspace', 'examples', ...
    'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
obj.Configuration.ResultsDir = resultsFolder;

bfStatus = obj.run('bugFinder');

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

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

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.

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

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 (<code>-osek-multitasking</code>)	See OSEK Multitasking Support release note .
<code>-xml-annotations-description</code>	See Code Annotations release note .
Compiler options: <ul style="list-style-type: none"> • Management of <code>size_t</code> (<code>-size-t-type-is</code>) • Management of <code>wchar_t</code> (<code>-wchar-t-type-is</code>) 	Replaces previous options related to <code>size_t</code> and <code>wchar_t</code> .

Updated Options

Option	Change
Compiler (-compiler)	<ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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.
Invalid use of floating point operation	<p>You can detect a comparison to 0.0 when you add the option -detect-bad-float-op-on-zero.</p> <p>The defect is renamed in the user interface to : Floating point comparison with equality operators. The command-line parameter is still BAD_FLOAT_OP.</p>

Option	Change
-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 <code>visual6</code> , <code>visual7.0</code> or <code>visual7.1</code> , 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.	Management of size_t (-size-t-type-is) 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.

Analysis Results

Security Standards Support: Detect violations of all secure coding guidelines from ISO/IEC Technical Specification 17961:2013 and more guidelines from SEI CERT C Coding Standard

In R2017b, you can check your code against all the guidelines from the ISO/IEC TS 17961:2013 Standard, including guidelines for signal handlers and file manipulations. Polyspace Bug Finder also covers additional CERT C coding defects.

Signal Handler Defect Checkers

Defect	Issue Detected
Shared data access within signal handler	You use a signal handler to access a shared object that is neither of type <code>volatile sig_atomic_t</code> nor a lock-free atomic object.
Signal call from within signal handler	You call <code>signal()</code> from within an interruptible signal handler.
Return from computational exception signal handler	Your signal handler returns normally after a computational exception signal <code>SIGFPE</code> , <code>SIGILL</code> , or <code>SIGSEGV</code> .
Function called from signal handler not asynchronous-safe	You use a signal handler to call a function that is not asynchronous-safe per the POSIX standard.
Function called from signal handler not asynchronous-safe (strict)	You use a signal handler to call a function that is not asynchronous-safe per the C standard.

File and I/O manipulation Defect Checkers

Defect	Issue Detected
Misuse of a FILE object	You dereference a pointer to a FILE object or manipulate the object through its pointer.
File descriptor exposure to child process	You use the same file descriptor in multiple processes.
Invalid file position	You call <code>fsetpos()</code> with a file position that was not returned from <code>fgetpos()</code> .
Alternating input and output from a stream without flush or positioning call	You perform alternating read and write operations on a stream without a flush or positioning call.
Use of indeterminate string	You do not reset the output buffer of <code>fgets()</code> or <code>fwgets()</code> when they fail.

Memory and Pointer Manipulation Defect Checkers

Defect	Issue Detected
Alignment changed after memory reallocation	You change the memory allocation of an object to a less strict alignment.
Mismatched alloc/dealloc functions on Windows	In Windows, you deallocate memory with a function that does not match the allocation function.
Subtraction or comparison between pointers to different arrays	You subtract or compare pointers to different arrays, or null pointers.

Other Defect checkers

Defect	Issue Detected
Missing byte reordering when transferring data	You transfer data without matching the endianness of the host and network.
Unsafe call to a system function	You call <code>system()</code> , <code>popen()</code> , <code>_popen()</code> , or <code>_wopen()</code> .
Use of automatic variable as putenv-family function argument	You use an automatic duration variable as the argument of a putenv-family function.
Misuse of structure with flexible array member	You do not allocate and copy a structure with a flexible array member dynamically.
Call through non-prototyped function pointer	You declare a pointer to a function with unspecified parameters.

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

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.

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.

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 MISRAC++ 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.
MISRA C: 2012 Rule 18.2 and MISRA C++ 5-0-17	Subtraction between pointers shall only be applied to pointers that address elements of the same array.	The rule checker flags more complex cases, such as a subtraction between a pointer to a local array and a pointer to a function argument. These additional results correspond to defects flagged by the checker <code>Subtraction</code> or <code>comparison between pointers to different arrays</code> .
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: <pre>int val; int val=1;</pre>

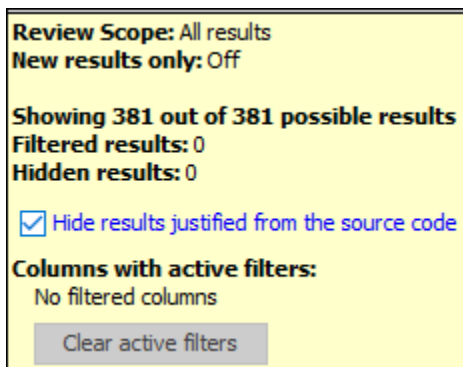
Reviewing Results

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

In R2017b, if you justify a result through source code annotations, subsequent analyses do not redisplay the result. The results do not appear in your results list or source code.

```
void bug_deadcode(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS))
        card = UNKNOWN_SUIT;
    if (card > 7) { /* polyspace DEFECT:DEAD_CODE */
        do_something_suit(card);
    }
}
```

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



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 dead 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 dead 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 [Annotate and Hide Known or Acceptable Results](#).

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 **Annotate and Hide Known or Acceptable Results**.
- *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

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.

The new annotation format has the following 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

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.

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](#).

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.

Results Review Workflow: Sort and filter results by subtype

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.

Family	Information	Detail	File	Function
-Tainted data 19				
-MISRA C:2004 1614				
-1 Environment 50				
-1.1	All code shall conform to ISO 9899:1990 'Programming languages - C', amended and corrected by ISO/IEC 9899/COR 1:1995, ISO/IEC 9899/AMD 1:1995, and ISO/IEC			
...	Category: Required	ANSI C90 forbids 'long double' type.	programming.c	bug_missingerrnoreset()
...	Category: Required	ANSI C90 forbids 'long double' type.	programming.c	corrected_missingerrnoreset()
...	Category: Required	ANSI C90 forbids 'long long int' type.	concurrency.c	corrected_datarace_task4()
...	Category: Required	ANSI C90 forbids 'long long int' type.	concurrency.c	File Scope
...	Category: Required	ANSI C90 forbids 'long long int' type.	concurrency.c	File Scope
...	Category: Required	ANSI C90 forbids 'long long int' type.	concurrency.c	bug_datarace_task4()
...	Category: Required	ANSI C90 forbids designated initializer.	numerical.c	corrected_intstdlib()
...	Category: Required	ANSI C90 forbids designated initializer.	numerical.c	corrected_intstdlib()
...	Category: Required	ANSI C90 forbids designated initializer.	programming.c	corrected_improperarrayinit()
...	Category: Required	ANSI C90 forbids designated initializer.	programming.c	corrected_improperarrayinit()
...	Category: Required	ANSI C90 forbids designated initializer.	programming.c	bug_improperarrayinit()
...	Category: Required	ANSI C90 forbids designated initializer.	programming.c	bug_improperarrayinit()
...	Category: Required	ANSI C90 forbids designated initializer.	programming.c	corrected_improperarrayinit()
...	Category: Required	ANSI C90 forbids designated initializer.	programming.c	corrected_improperarrayinit()
...	Category: Required	ANSI C90 forbids designated initializer.	programming.c	bug_improperarrayinit()
...	Category: Required	ANSI C90 forbids long long integer constants.	programming.c	corrected_unsafestrtonumeric()
...	Category: Required	ANSI C90 forbids long long integer constants.	tainteddata.c	sanitize_atoi()
...	Category: Required	ANSI C90 forbids mixed declarations and code.	goodpractice.c	corrected_hardcodedmemsize()
...	Category: Required	ANSI C90 forbids mixed declarations and code.	goodpractice.c	corrected_hardcodedloopboundary()

You can easily group 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.

Constraint Specification: Navigate easily to the constraint specification interface for Bug Finder results

In R2017b, you can open the Specified Constraints window when viewing Bug Finder results. In this window, you can specify external constraints on global variables in your code.

Name	File	Attributes	Data Type	Main Generator Called	Init Mode	Init Range	Initialize Pointer
Global Variables							
...val	file.c	const	int32		INIT	-1000..1000	
User Defined Functions							
Stubbed Functions							
Non Applicable							

To see the Specified Constraints window, with the Bug Finder results open, select **Window > Show/Hide View > Specified Constraints**.

If a global variable has a fixed value assigned in your code:

```
const int var = 1;
```


but you want to analyze the code for multiple values of the variable, you can override the assignment by using external constraints. For instance, if you see **Dead code** defects in your results from the fixed value of a variable, you can navigate to the Specified Constraints window and specify a range for the variable.

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

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

You can now 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.

R2017a

Version: 2.3

New Features

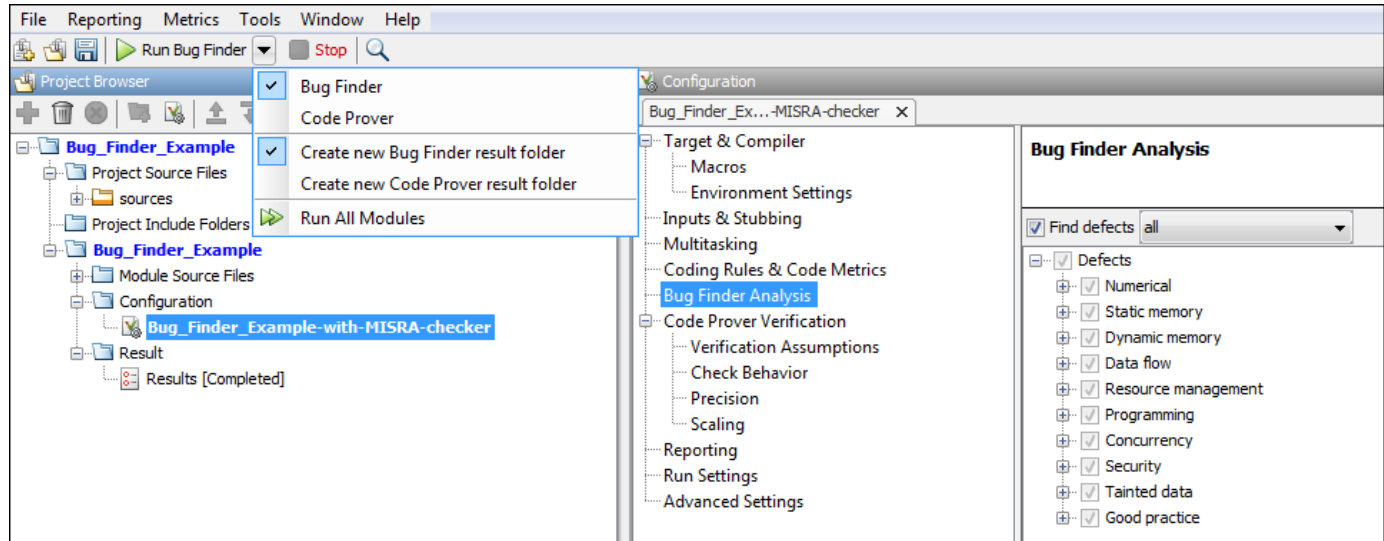
Bug Fixes

Compatibility Considerations

Analysis Setup

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

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



The unified user interface has the following 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.

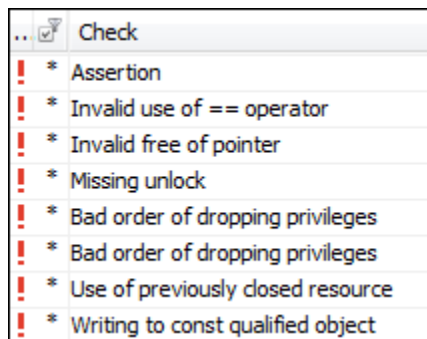
Bug Finder User:

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

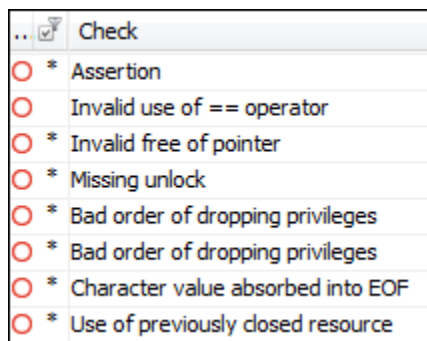
When you create a project and add your source files, they are automatically added to the first module. If you add source files later, you have to select them and using the right-click option **Copy to Module_n**, copy them to the module that you want.

- 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:

F...	<input checked="" type="checkbox"/>	Check
!	*	Out of bounds array index
!	*	Illegally dereferenced pointer
!	*	Non-terminating call
!	*	Non-terminating loop
!	*	Invalid use of standard library routine

R2017a:

F...	<input checked="" type="checkbox"/>	Check
●	*	Out of bounds array index
●	*	Illegally dereferenced pointer
●	*	Non-terminating call
●	*	Non-terminating loop
●	*	Invalid use of standard library routine

- *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
```

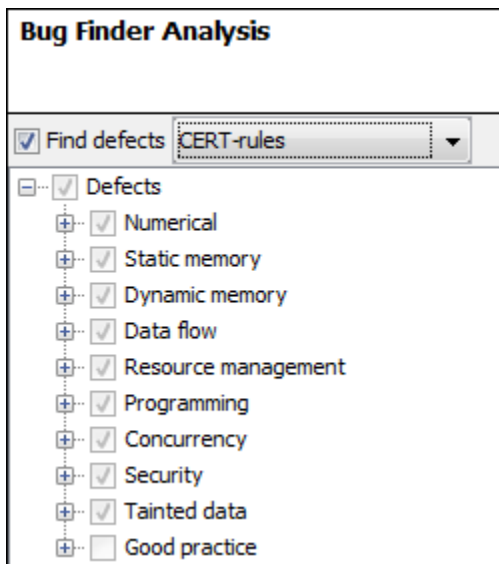
To create a Windows batch file that runs a Bug Finder analysis, run the command:

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

Easier Compliance with Security Standards: Choose CWE, CERT C99, or ISO/IEC TS 17961 coding standard and address corresponding violations through Polyspace results and security reports

In R2017a, you can provide a security standard such as CWE, CERT C99 or ISO/IEC TS 17961 for Polyspace analysis.

Analysis: The analysis runs defect and coding rule checkers that correspond to elements in the standard.



Results: After analysis, you see the security standard ID-s corresponding to each result.

Family	Check	File	Function	CERT ID
* (with red circle icon)	Incorrect pointer scaling	programming.c	bug_badptrscaling()	EXP08-C ARR39-C
* (with red circle icon)	Missing lock	concurrency.c	File Scope	CON01-C
* (with red circle icon)	Bitwise operation on negative value	numerical.c	bug_bitwiseneg()	INT13-C
* (with red circle icon)	File manipulation after chroot() without...	security.c	bug_chrootmisuse()	POS05-C
* (with red circle icon)	File manipulation after chroot() without...	security.c	bug_chrootmisuse()	POS05-C
* (with red circle icon)	Vulnerable permission assignments	security.c	bug_dangerouspermissions()	FIO06-C
* (with red circle icon)	Mismatch between data length and size	security.c	bug_datalengthmismatch()	ARR38-C

Report: When you generate a report, you can choose a template tailored for a specific security standard. The report shows the security standard ID-s corresponding to each result.

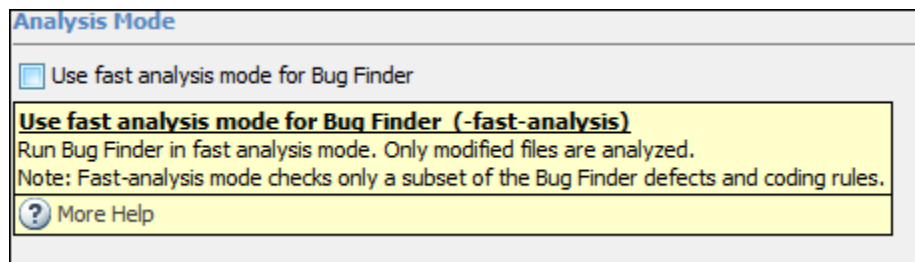
ID	Defect	Impact	Function	Detail	Severity	Status	Comment	CERT
5743	Unsigned integer conversion overflow	Low	bug_uintconvoval()	Conversion from unsigned int64 to unsigned int16 overflows. Valid range: [0 .. 65535]				INT02-C INT18-C INT31-C FLP34-C
5744	Unsigned integer conversion overflow	Low	bug_uintconvoval_wraparound()	Conversion from unsigned int32 to unsigned int8 overflows. Valid range: [0 .. 255]				INT02-C INT18-C INT31-C FLP34-C
5742	Sign change integer conversion overflow	Medium	bug_signchange()	Conversion from unsigned int32 to int32 overflows. Valid range: [-2 ³¹ .. 2 ³¹ -1]				INT31-C

You can easily adhere to a security standard using Polyspace analysis.

For details of the workflow, see Check Code for Security Standards.

Incremental Analysis of Specific Checks: Analyze only files edited since previous analysis to quickly find new defects and coding rule violations

In R2017a, you can run a fast analysis mode in Bug Finder. In this mode, if you perform an analysis and then edit some files, a later analysis considers only the files that you edited.



You wait less for analysis results from your second analysis onwards. During development, you can frequently run analysis in fast mode and quickly check for new defects.

Additional considerations:

- *Is the fast analysis mode different from a full Bug Finder analysis?*

In fast analysis mode, Bug Finder checks for a subset of defects and coding rules only. In R2017a, these defects and rules can be found within a single compilation unit, such as a single function or file. The software does not perform interprocedural or cross-functional analysis.

- *If I enable a defect checker that cannot be checked fast, what happens in the fast analysis mode?*

The defect checker is internally disabled. When you switch back to full analysis, the defect checker is enabled again. For information on:

- The defect checkers that can run fast, see Results Found by Fast Analysis.
- The option to enable fast analysis, see Use fast analysis mode for Bug Finder (-fast-analysis).

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

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`).

Target Environment	
Compiler	tasking
Target processor type	tricore

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

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.

Target Language	
Source code language	CPP
Target Environment	
Compiler	visual14.0

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

The updated Visual C++ support has the following 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, µC/OS II or C++11 multithreading functions

In R2017a, if you use the Windows, µC/OS II or C++11 functions for multitasking, the Polyspace analysis can interpret them semantically.

Polyspace interprets the following functions:

Family	Thread Created	Critical Section Begins	Critical Section Ends
Windows	CreateThread	EnterCriticalSection	LeaveCriticalSection
µC/OS II	OSTaskCreate	OSMutexPend	OSMutexPost
C++11	std::thread::thread	std::mutex::lock	std::mutex::unlock

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.

Autodetection of Concurrency Primitives: Map Unsupported Thread Creation Functions to Supported Functions

In R2017a, you can map your thread creation functions to thread-creation functions that Polyspace can detect automatically. You can also perform the mapping for functions that begin and end critical sections.

For instance, for the following code, you can map the functions `createTask`, `takeLock` and `releaseLock` to the Pthreads functions, `pthread_create`, `pthread_mutex_lock` and `pthread_mutex_unlock` respectively.

```
/* Assume global variables and functions are defined */

void* task1(void* a) {
    takeLock(&lock);
    var1++;
    var2++;
    releaseLock(&lock);
    return 0;
}

void* task2(void* a) {
    takeLock(&lock);
    var1++;
    releaseLock(&lock);
    var2++;
    return 0;
}

void main() {
    createTask(task1,&t_id1,0,0);
    createTask(task2,&t_id2,0,0);
}
```

Polyspace supports automatic concurrency detection only for certain families of concurrency primitives. You can extend the support to your family of concurrency functions by using this mapping.

If Polyspace determines your multitasking model from your code, the analysis can find possible race conditions and other defects, without additional setup efforts. Otherwise, you have to specify your multitasking model explicitly through the manual multitasking options.

Additional considerations:

- *How do I map an unsupported thread creation function to a supported function?*

You specify the mapping in an XML file. You then provide the XML file as argument of the analysis option `-function-behavior-specifications`.

For examples, see `-function-behavior-specifications`.

- *How do I know which function to map to?*

Map your function to the supported function that is most similar to your function in the number and types of parameters.

For instance, in the above example, you can map the function `createTask` to the thread creation functions `pthread_create` (POSIX®), `CreateThread` (Windows) or `OSTaskCreate` (µC/OS II). However, the arguments of `createTask` align most closely with `pthread_create`.

For the list of supported functions that you can map to, see the sample mapping file `function-behavior-specifications-sample.xml` in `matlabroot\polyspace\verifier\cxx\`. `matlabroot` is the MATLAB installation folder, such as `C:\Program Files\MATLAB\R2017a`.

Manual Multitasking Setup: Specify routines that disable and reenab all interrupts

In R2017a, when specifying your multitasking model for analysis, you can provide a routine that disables all interrupts.




For instance, in the following code, the function `disable_all_interrupts` disables all interrupts until the function `enable_all_interrupts` is called. Even if `task`, `isr1` and `isr2` run concurrently, the operations `x=0` or `x=1` cannot interrupt the operation `x++`.

```
int x;

void isr1() {
    x = 0;
}

void isr2() {
    x = 1;
}

void task() {
    disable_all_interrupts();
    x++;
    enable_all_interrupts();
}
```

Disabling all interrupts	Disabling routine	Enabling routine	  
	<code>disable_all_interrupts</code>	<code>enable_all_interrupts</code>	

If you protect operations on a shared variable by disabling interrupts, you can specify this protection for the Polyspace analysis. The analysis uses this information to give you more precise results for data race defects.

Additional considerations:

- *Does the routine disable all preemption or preemption by only a certain class of interrupts?*

The routine that you specify for the option disables preemption by all:

- Noncyclic entry points
- Cyclic tasks
- Interrupts

In other words, the analysis considers that the body of operations between the disabling routine and the enabling routine is atomic and not interruptible at all.

- *How are routines to disable interrupts different from protection via critical sections?*

In the Polyspace multitasking model, to protect two sections of code *from each other* via critical sections, you have to embed them in the same critical section. In other words, you have to place the two sections between calls to the same lock and unlock function.

For instance, suppose you use critical sections as follows:

```
void isr1() {
    begin_critical_section();
    x = 0;
    end_critical_section();
}

void isr2() {
    x = 1;
}

void task() {
    begin_critical_section();
    x++;
    end_critical_section();
}
```

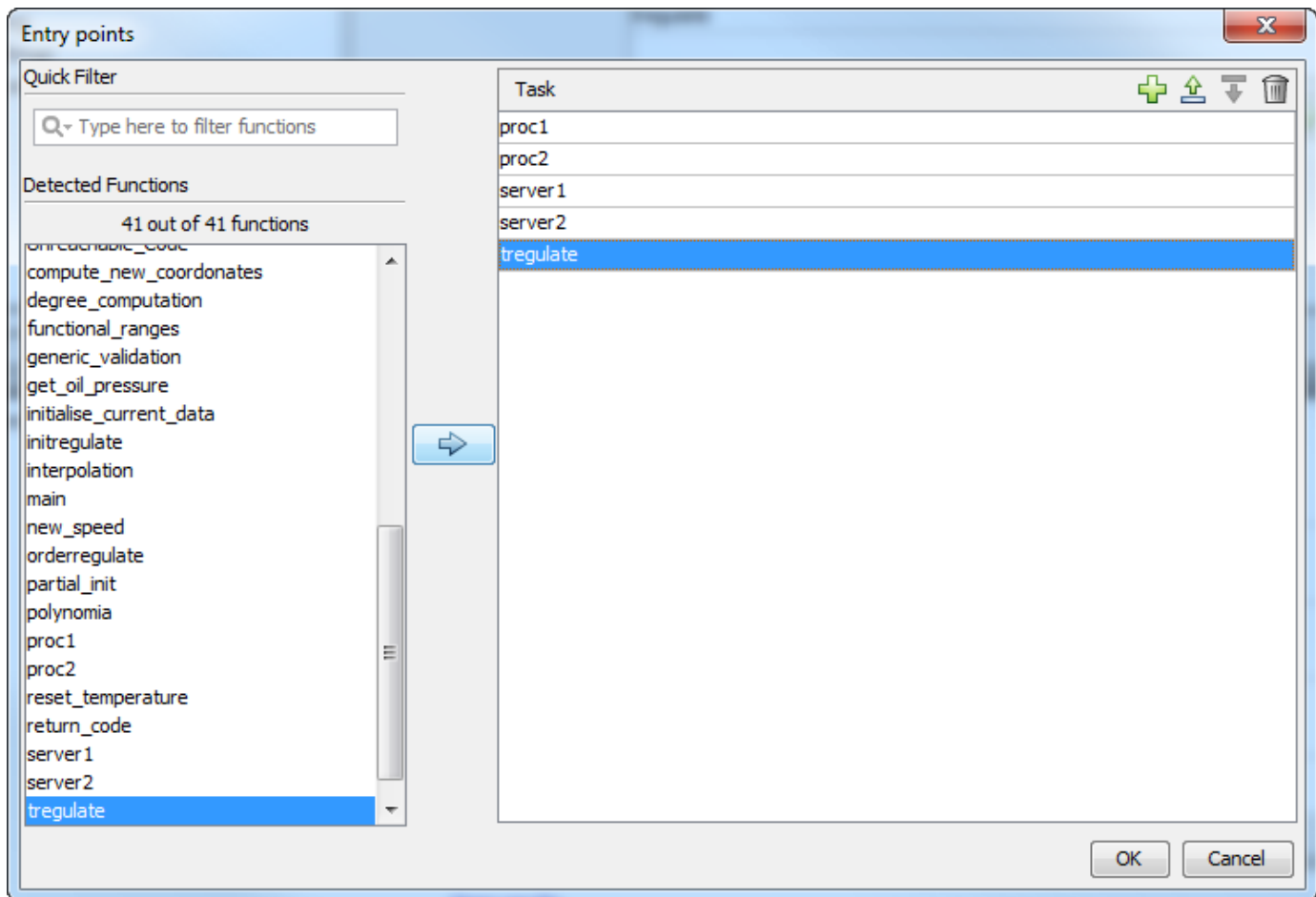
Here, the operation `x++` is protected from the operation `x=0` in `isr1`, but not from the operation `x=1` in `isr2`. If the function `begin_critical_section` disabled *all interrupts*, calling it before `x++` would have been sufficient to protect it.

In this way, critical sections are conceptually different from routines to disable all interrupts. Typically, you use one pair of routines in your code to disable and reenable interrupts, but you can have many pairs of lock and unlock functions that implement critical sections.

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

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:



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

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:

```
polyspaceBugFinder(opts);
```

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 when required, and use opts to run analysis. For the object properties, 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.

Support for 128-bit variables

In R2017a, Polyspace Bug Finder analysis supports 128-bit variables.

128-bit variables in your code do not cause compilation errors. For instance, if you use the GCC type `__int128`, you can run Polyspace Bug Finder on your code.

Improvement in automatic project creation from build systems

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.

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

New Options

Option	Description
Use fast analysis mode for Bug Finder (<code>-fast-analysis</code>)	Run analysis using faster local mode of Bug Finder. See Incremental Analysis of Select Checks on page 13-6.
Disabling all interrupts (<code>-routine-disable-interrupts</code> - <code>-routine-enable-interrupts</code>)	Specify routines that disable and reenables interrupts. See Manual Multitasking Setup on page 13-9.

Updated Options

Option	Change	More Information
Report template	Renamed in user interface	New name: Bug Finder report The command-line name is still <code>-report-template</code> .
Batch	Renamed in user interface	New name: Run Bug Finder analysis on a remote cluster The option is now in the Run Settings node in your project configuration. The command-line name is still <code>-batch</code> .
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 <code>-add-to-results-repository</code> .
Compiler (<code>-compiler</code>)	New values added	You can specify the following arguments: <ul style="list-style-type: none"> tasking See TASKING Compiler Support on page 13-7. visual14.0 See Microsoft Visual C++ Support on page 13-7.
Find defects (<code>-checkers</code>)	New value added	You can specify the following arguments: <ul style="list-style-type: none"> CWE CERT-rules CERT-all ISO-17961 See Security Standards Checking on page 13-5.
Check MISRA C:2012 (<code>-misra3</code>)	New value added	You can specify the following arguments: <ul style="list-style-type: none"> CERT-rules CERT-all ISO-17961 See Security Standards Checking on page 13-5.

Removed Options

Option	Status	Description
Disable automatic concurrency detection (-disable-concurrency-detection)	Removed	Option will be removed in a future release. Detecting concurrency primitives automatically saves time in setup and does not impact performance. The option is not required anymore.
Import Folder (-import-dir)	Warning	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 instead, <code>polyspace-comments-import</code> .
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) (Polyspace Code Prover) instead.
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: <ul style="list-style-type: none"> • Linux: If you get compilation errors, use Compiler (-compiler) (Polyspace Code Prover) <code>gnux.x</code>. Sometimes, you might also have to set Preprocessor definitions (-D) (Polyspace Code Prover) to <code>linux</code>, <code>unix</code>, or <code>__linux__</code>. • Visual: Use Compiler (-compiler) (Polyspace Code Prover) <code>visualx.x</code> • Vxworks: Use the VxWorks® configured template. For more information, see Create Project Using Configuration Template (Polyspace Code Prover). <ul style="list-style-type: none"> • Solaris: Remove -OS-target. • no-predefined-OS: Remove -OS-target.

Option	Status	Description
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.
-support-FX-option-results	Removed	

Compatibility Considerations

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

Changes in MATLAB option object properties

These classes will be removed in a future release.

- `polyspace.BugFinderOptions`: To customize Polyspace analysis of handwritten code, use `polyspace.Options` instead.
- `polyspace.ModelLinkBugFinderOptions`: 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
<code>optsOld.Reporting.EnableReportGeneration</code>	<code>optsNew.MergedReporting.EnableReportGeneration</code>
<code>optsOld.Reporting.ReportTemplate</code>	<code>optsNew.MergedReporting.BugFinderReportTemplate</code>
<code>optsOld.Reporting.ReportOutputFormat</code>	<code>optsNew.MergedReporting.ReportOutputFormat</code>

ComputingSettings

Removed	Use instead
<code>optsOld.ComputingSettings.Batch</code>	<code>optsNew.MergedComputingSettings.BatchBugFinder</code>
<code>optsOld.ComputingSettings.AddToResultsRepository</code>	<code>optsNew.MergedComputingSettings.AddToResultsRepositoryBugFinder</code>

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.

Analysis Results

Additional Defect Checkers for Security: Check for security vulnerabilities such as incorrect use of cryptographic routines

In R2017a, Polyspace Bug Finder introduces new defect checkers for preventing security vulnerabilities in your code. The most notable are the cryptography defect checkers.

Cryptography Defect Checkers

Using Polyspace Bug Finder defects, you can identify incorrect use of the EVP cipher routines from the OpenSSL library.

The following issues are detected using the cryptography defects.

Initialization Vector

Defect	Issue Detected
Constant block cipher initialization vector	You used a constant for the initialization vector.
Predictable block cipher initialization vector	You used a weak random number generator for the initialization vector.
Missing block cipher initialization vector	You forgot to associate a non-null initialization vector with the cipher context.

Key

Defect	Issue Detected
Constant cipher key	You used a constant for the encryption or decryption key.
Predictable cipher key	You used a weak random number generator for the encryption or decryption key.
Missing cipher key	You forgot to associate a non-null encryption or decryption key with the cipher context.

Wrong Order of Operations

Defect	Issue Detected
Inconsistent cipher operations	You perform a decryption on the same context as an encryption and immediately following it, or vice versa.
Missing cipher data to process	Before performing a final step, you do not perform update steps for encrypting or decrypting the data.
Missing cipher final step	You do not perform a final step after update steps for encrypting or decrypting data.

Algorithms and Modes

Defect	Issue Detected
Weak cipher algorithm	You associated a weak encryption algorithm with the cipher context.
Weak cipher mode	You associated a weak mode with the cipher context.

Defect Checkers for errno Usage

Defect	Issue Detected
Errno not checked	You call a function that sets <code>errno</code> to indicate error conditions, but do not follow the function call with a check on <code>errno</code> to see if the error occurred.
Errno not reset	You call a function that sets <code>errno</code> but do not reset <code>errno</code> prior to the call.
Misuse of <code>errno</code>	You check <code>errno</code> for error conditions following calls to functions that do not necessarily set <code>errno</code> to indicate error conditions or sets other error indicators.

Defect Checkers for Type Conversions

Defect	Issue Detected
Misuse of sign-extended character value	You perform a data type conversion with sign extension and use the resulting sign-extended character value as array index or for comparison with EOF.
Character value absorbed into EOF	You perform a data type conversion that can convert a character value that is not EOF into EOF, and then compare the result with EOF.

Defect Checkers for Memory Comparisons

Defect	Issue Detected
Memory comparison of padding data	You use <code>memcmp</code> to compare two structures and in the process, compare garbage data stored in the structure padding.
Memory comparison of strings	You use <code>memcmp</code> to compare two strings and in the process, compare garbage data stored after the null terminator.

Other Defect Checkers

Defect	Issue Detected
Misuse of return value from nonreentrant standard function	You use the pointer to a static buffer from a nonreentrant standard function despite a subsequent call to the same function.
Misuse of <code>readlink()</code>	You pass a buffer size argument to <code>readlink()</code> that does not leave space for a null terminator in the buffer.

MISRA Amendment Support: Check your code for new security guidelines in MISRA C:2012 Amendment 1

In R2017a, you can check for violations of the additional security guidelines introduced in MISRA C:2012 Amendment 1.

Rule	Description
MISRA C:2012 Directive 4.14	The validity of values received from external sources shall be checked.
MISRA C:2012 Rule 12.5	The <code>sizeof</code> operator shall not have an operand which is a function parameter declared as "array of type".
MISRA C:2012 Rule 21.13	Any value passed to a function in <code><ctype.h></code> shall be representable as an unsigned char or be the value EOF.
MISRA C:2012 Rule 21.14	The Standard Library function <code>memcmp</code> shall not be used to compare null terminated strings.
MISRA C:2012 Rule 21.15	The pointer arguments to the Standard Library functions <code>memcpy</code> , <code>memmove</code> and <code>memcmp</code> shall be pointers to qualified or unqualified versions of compatible types.
MISRA C:2012 Rule 21.16	The pointer arguments to the Standard Library function <code>memcmp</code> shall point to either a pointer type, an <i>essentially signed type</i> , an <i>essentially unsigned type</i> , an <i>essentially Boolean type</i> or an <i>essentially enum type</i> .
MISRA C:2012 Rule 21.17	Use of the string handling function from <code><string.h></code> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters.
MISRA C:2012 Rule 21.18	The <code>size_t</code> argument passed to any function in <code><string.h></code> shall have an appropriate value.
MISRA C:2012 Rule 21.19	The pointers returned by the Standard Library functions <code>localeconv</code> , <code>getenv</code> , <code>setlocale</code> or <code>strerror</code> shall only be used as if they have pointer to <code>const</code> -qualified type.
MISRA C:2012 Rule 21.20	The pointer returned by the Standard Library functions <code>asctime</code> , <code>ctime</code> , <code>gmtime</code> , <code>localtime</code> , <code>localeconv</code> , <code>getenv</code> , <code>setlocale</code> or <code>strerror</code> shall not be used following a subsequent call to the same function.
MISRA C:2012 Rule 22.7	The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF.
MISRA C:2012 Rule 22.8	The value of <code>errno</code> shall be set to zero prior to a call to an <i>errno-setting function</i> .
MISRA C:2012 Rule 22.9	The value of <code>errno</code> shall be tested against zero after calling an <i>errno-setting function</i> .
MISRA C:2012 Rule 22.10	The value of <code>errno</code> shall only be tested when the last function to be called was an <i>errno-setting function</i> .

New Code Metrics: See number of lines in header files and number of local variables per function

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 static variables for every function and method

You can determine the memory footprints of your code using these new metrics (along with other already existing metrics).

Changes to coding rule checking

New Rules Supported

In R2017a, the following new rules are supported:

- Additional security guidelines in MISRA C: 2012 Amendment 1.

See MISRA Amendment Support on page 13-19.

- MISRA C:2012 Directive 4.7 (partially supported): If a function returns error information, then that error information shall be tested.

Updated Specifications

In R2017a, the following changes have been made in checking of previously supported MISRA C rules.

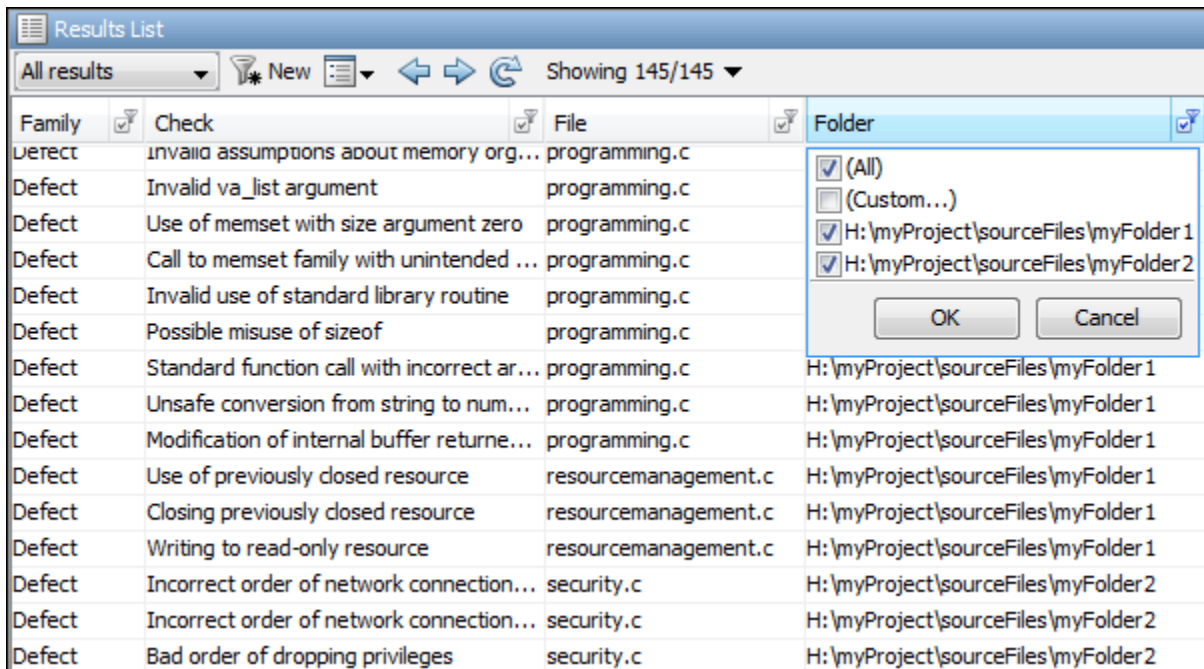
Rule	Rule	Improvement
MISRA C: 2004 Rule 5.1	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.	<p>The rule checker shows all identifiers that have the same first 31 characters as one rule violation. Previously, every pair of identifiers with same 31 characters was shown as a separate violation.</p> <p>For instance, in the following code snippet, the rule violation appears only once.</p> <pre>extern int engine_exhaust_gas_temperature_raw; static int engine_exhaust_gas_temperature_scaled; static int engine_exhaust_gas_temperature_cutoff;</pre> <p>Previously, the violation was shown three times.</p> <p>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.</p>

Rule	Rule	Improvement
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 <code>extern</code> in a non-header file violates this rule.

Reviewing Results

Folder Names in Results: Filter or organize analysis results by source folder names

In R2017a, the source folder name is shown in the list of analysis results.



You can order your results by folders or filter results belonging to specific folders. Using custom filters, you can filter out subfolders of a folder in one click.

Code to Model Traceability: Switch easily between identifiers in generated code and corresponding blocks in model

In R2017a, you can trace an instance of a variable in generated code back to your model.

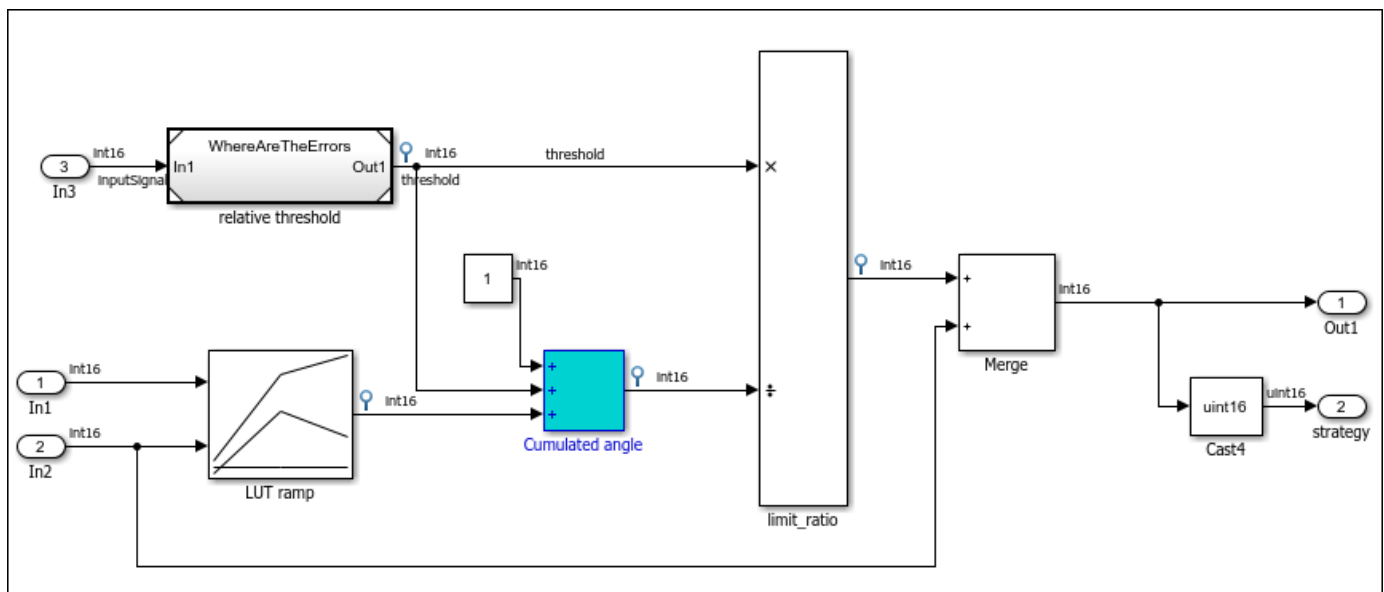

```

/* Sum: '<S4>/Cumulated angle' incorporates:
 * Constant: '<S4>/Constant'
 */
tmp = 1 + controller_B.threshold;
if (tmp > 32767) {
    tmp = 32767;
} else {
    if (tmp < -32768) {
        tmp = -32768;
    }
}

tmp += controller_B.LUTramp;
if (tmp > 32767) {
    tmp = 32767;
} else {
    if (tmp < -32768) {
        tmp = -32768;
    }
}
    
```

- Search For "threshold" in Current Source File Ctrl+F
- Search For "threshold" 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.



The new ability to trace from code to model enables the following:

- *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

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.BugFinderResults('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.BugFinderResults`.

You can use the capabilities of MATLAB to obtain graphs and statistics about your Polyspace results.

Double Lock and Other Concurrency Defects: Get help investigating the defects using detailed control flow information

In R2017a, you can see detailed control flow information for concurrency defects such as deadlock and double lock.

For instance, in the following traceback for a double lock defect, you see this information:

- Entry and exit from a function `f19`
- Entry or non-entry into `if` conditions.

○ Double lock (Impact: High) ? Task is waiting for already acquired resource.				
	Event	File	Scope	Line
1	Entering task 't19'	myFile_multitasking.c	f19()	395
2	Entering if branch (if-condition true)	myFile_multitasking.c	t19()	398
3	't19' enters critical section Lock function: 'LOCK'	myFile_multitasking.c	t19()	399
4	Entering function 'f19'	myFile_multitasking.c	t19()	405
5	Entering if branch (if-condition true)	myFile_multitasking.c	f19()	391
6	Entering function 'unlock19'	myFile_multitasking.c	f19()	392
7	Not entering if statement (if-condition false)	myFile_multitasking.c	unlock19()	385
8	Return of function 'unlock19'	myFile_multitasking.c	unlock19()	388
9	Return of function 'f19'	myFile_multitasking.c	f19()	394
10	't19' attempts to enter same critical section.	myFile_multitasking.c	t19()	406
11	○ Double lock	myFile_multitasking.c	File Scope	406

You can click each event to navigate to the corresponding location in your source code.

To fix concurrency defects, you often have to decide where to place lock and unlock functions (functions that begin and end critical sections). Using the improved traceback, you can decide the placements more easily.

Spreadsheet of Checkers: Use spreadsheet to keep track of checkers that you enable

In R2017a, the software provides a spreadsheet containing the Polyspace Bug Finder defect and coding rule checkers. The spreadsheet also maps the defects to standards such as CWE, CERT-C or ISO-17961.

The spreadsheet is in *matlabroot*\polyspace\resources. Here, *matlabroot* is the MATLAB installation folder, such as C:\Program Files\MATLAB\R2017a.

You can use this spreadsheet to keep track of the defect checkers that you enable and add notes explaining why you do not enable the other checkers.

R2016b

Version: 2.2

New Features

Bug Fixes

Compatibility Considerations

Analysis Setup

Diab Compiler Support: Set up Polyspace analysis easily for code compiled with Wind River Diab compiler

If you build your source code with 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 an analysis without facing compilation errors. See `Diab Compiler (-compiler diab)`.

The software supports version 5.9 and older versions of the Diab compiler.

Multitasking Code Analysis Setup: Specify cyclic tasks and nonpreemptable interrupts directly as analysis 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 the 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 Update 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 analysis 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 analysis from a command line.

DOS or UNIX Command Line

Use the new option `polyspace-bug-finder -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 analysis using `options_command.txt`, `source_command.txt`, and `.polyspace_conf.psprj`. 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 `.bf.psprj` file as an argument to `polyspaceBugFinder`.

The syntax `polyspaceBugFinder(PSPRJ file, '-nodesktop')` runs an analysis using the files and options from the *PSPRJ file*.

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 false Data race results.

Polyspace API in MATLAB: Configure and run Polyspace using MATLAB objects

Polyspace scripting from the MATLAB command line is now easier and more MATLAB-friendly. R2016b introduces a set of classes, methods, and function improvements to help you run Polyspace from the MATLAB command line. For more information and examples, see the linked reference pages.

Classes

Name	Description
<code>polyspace.BugFinderOptions</code>	An options object with properties that map to the Polyspace environment configuration options. Use this object to customize analysis options and run analysis.
https://www.mathworks.com/help/releases/R2016b/bugfinder/ref/polyspace.modellinkbugfinderoptions-class.html <code>polyspace.ModelLinkBugFinderOptions</code>	Another version of the <code>BugFinderOptions</code> object with properties specifically for model generated code. Use this object to customize analysis options and run analysis.
<code>polyspace.GenericTargetOptions</code>	A helper object for the <code>BugFinderOptions</code> classes. Use this object to customize a generic target.
<code>polyspace.DefectsOptions</code>	A helper object for the <code>BugFinderOptions</code> classes. Use this object to customize the list of defects checked during the analysis.
<code>polyspace.CodingRulesOptions</code>	A helper object for the <code>BugFinderOptions</code> object. Use this object to customize the list of coding rules checked during the analysis.

Methods

Name	Description
<code>polyspace.Options.copyTo</code>	Copy settings between options objects. You can use this method to copy options from a <code>BugFinderOptions</code> object to a <code>CodeProverOptions</code> object and vice versa.
<code>polyspace.Options.generateProject</code>	Generate a <code>.psprj</code> file from an options object to open in the Polyspace interface.

Functions

Name	Description
<code>polyspaceBugFinder</code>	Run an analysis using <code>BugFinderOptions</code> objects or <code>.psprj</code> 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.

Eclipse Build Support: Set up Polyspace analysis 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 for the analysis, trace the build command inside the IDE. For more information, see [Customize Analysis Options](#).

Visual Studio 2010 add-in support to be removed from installation

In a future release, the Polyspace add-in for Visual Studio 2010 will no longer 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

The Polyspace plugin for IBM Rational® Rhapsody® supports Rhapsody 8.1. For more information, see [Find Defects from IBM Rational Rhapsody](#).

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), 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, there is no ambiguity about which file to include. No warning is 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 analysis stops after compilation, for instance because of communication problems between the server and client computers, you do not have to restart the analysis from the beginning. You can reuse compilation results from the previous failed analysis.

For more information, see `-submit-job-from-previous-compilation-results`.

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 <code>-dialect</code> with <code>-compiler</code> 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 <code>Compiler (-compiler)</code> with argument <code>diab</code> .
Target processor type (-target)	Updated for the Wind River Diab compiler.	In the user interface, if you select <code>diab</code> for <code>Compiler (-compiler)</code> , 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)	<p>Removed from the user interface.</p> <p>If you use the option in your scripts, you see a warning.</p>	<p>Option will be permanently removed in a future release.</p> <p>Remove the option from your scripts. For some option arguments, you might have to perform these additional steps:</p> <ul style="list-style-type: none"> • Linux: If you get compilation errors, use a <code>gnux.x</code> argument for Compiler (-compiler). <p>Sometimes, you might have to explicitly define operating-system-specific macros such as <code>linux</code>, <code>unix</code>, or <code>__linux__</code>. See Preprocessor definitions (-D).</p> <ul style="list-style-type: none"> • Visual: Use a <code>visualx.x</code> argument for Compiler (-compiler). • Vxworks: Use the options from the VxWorks templates. <p>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 <code>-generate-launching-scripts-for</code>.</p> <ul style="list-style-type: none"> • Solaris: Just remove the option <code>-OS-target</code>. • no-predefined-OS: Just remove the option <code>-OS-target</code>.

Changes in analysis options and binaries

In R2016b, the following options have been added, changed, or removed.

For **Target & Compiler** options, see “Changes in Target & Compiler analysis options” on page 14-6. For other options, see [here](#).

New Options

Option	Description
Cyclic tasks (-cyclic-tasks)	Specify functions that represent cyclic tasks.
Interrupts (-interrupts)	Specify functions that represent nonpreemptable interrupts.
-preemptable-interrupts	Specify functions that represent preemptable interrupts.
-non-preemptable-tasks	Specify functions that represent nonpreemptable tasks.

Updated Options

Option	Change	More Information
Coding rule subsets <code>single-unit-rules</code> and <code>system-decidable-rules</code>	Subsets now available from the drop-down list.	These subsets are available for Check MISRA C:2004 (<code>-misra2</code>), Check MISRA AC AGC (<code>-misra-ac-agc</code>), and Check MISRA C:2012 (<code>-misra3</code>)

Removed Options

Option	Status	Description
Import Folder (<code>-import-dir</code>)	Warning	Option will be removed in a future release.
<code>-easy-setup-preprocess</code>	Warning	Option will be removed in a future release.
<code>polyspace-automatic-verification</code>	Warning	Binary will be removed in a future release.
<code>polyspace-verifier</code>	Warning	Binary will be removed in a future release.
<code>rte-kernel</code>	Warning	Binary will be removed in a future release.
<code>polyspace-remote</code>	Warning	Binary will be removed in a future release.
<code>gui-api</code>	Warning	Binary will be removed in a future release. Use instead, <code>polyspace-comments-import</code> .
Files and folders to ignore (<code>-includes-to-ignore</code>)	Error	Use the option <code>Do not generate results for (-do-not-generate-results-for)</code> to suppress results from headers and sources in certain files or folders.
<code>-support-FX-option-results</code>	Error	Option will be removed in a future release.
<code>polyspace-vcproj</code>	Removed	Use <code>polyspace-configure</code> or the Polyspace Add-In for Visual Studio instead.

Compatibility Considerations

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

Analysis Results

CERT C Support: Identify CERT C violations using defect checkers and coding rules

In R2016b, you can comply with more CERT C Coding Standard rules using Polyspace defects and coding rules.

For more information, see Mapping Between CERT C Standards and Polyspace Results. The new defects added in R2016b specifically for CERT C support are listed here.

Concurrency

Name	Description	CERT C Rule
Data race through standard library function call	Certain standard library functions are called from multiple tasks without protection	CON33-C: Avoid race conditions when using library functions
Destruction of locked mutex	A task is trying to destroy a locked mutex that has not yet been unlocked	CON31-C: Do not destroy a mutex while it is locked

Good Practice

Name	Description	CERT C Rule
Bitwise and arithmetic operation on the same data	Code statement with mixed bitwise and arithmetic operations	INT14-C: Avoid performing arithmetic and bitwise operations on the same data
Missing reset of a freed pointer	Pointer free not followed by a reset statement to clear leftover data	MEM01-C: Store a new value in pointers immediately after free()
Missing break of switch case	No comments at the end of switch case without a break statement	MSC17-C: Finish every set of statements associated with a case label with a break statement
Hard-coded object size used to manipulate memory	Memory manipulation uses hard-coded size instead of <code>sizeof</code>	EXP09-C: Use <code>sizeof</code> to determine the size of a type or variable

Numerical

Name	Description	CERT C Rule
Use of plain <code>char</code> type for numerical value	Plain <code>char</code> variable used in arithmetic operation without explicit signedness	INT07-C: Use only explicitly signed or unsigned <code>char</code> type for numeric values
Bitwise operation on negative value	Undefined behavior for bitwise operations on signed values	INT13-C: Use bitwise operations only on unsigned operands

Programming

Name	Description	CERT C Rule
Unsafe conversion from string to numerical value	String to number conversion without validation checks	ERR34-C: Detect errors when converting a string to a number
Abnormal termination of exit handler	Exit handler function terminates incorrectly	ENV32-C: All exit handlers must return normally
Unsafe conversion between pointer and integer	Misaligned or invalid results from conversions between pointer and integer types	INT36-C: Unsafe conversion between pointer and integer

Resources

Name	Description	CERT C Rule
Opening previously opened resource	Opening an already opened file	FIO24-C: Do not open a file that is already open

Security

Name	Description	CERT C Rule
Returned value of a sensitive function not checked	Calls to sensitive or critical functions should be checked for unexpected return values and errors	EXP12-C: Do not ignore values returned by functions ERR33-C: Detect and handle standard library errors
Bad order of dropping privileges	Dropped user or primary group privileges before dropping primary/supplementary group privileges	POS36-C: Observe correct revocation order while dropping privileges
Privilege drop not verified	Verify privilege relinquishment	POS37-C: Ensure that privilege relinquishment is successful

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 using the following 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.

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 can 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.

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 the following rules have been updated.

Standard	Rule	Change
MISRA C++:2008	5-0-3	If two types have the same size in the target configuration, Polyspace no longer raises a violation.
	5-0-6	If two types have the same size in the target configuration, Polyspace no longer raises a violation.
	5-0-8	If two types have the same size in the target configuration, Polyspace no longer raises a violation.
MISRA C:2004 and MISRA AC AGC	10.1	If two types have the same size in the target configuration, Polyspace no longer raises a violation.
	10.2	If two types have the same size in the target configuration, Polyspace no longer raises a violation.
	10.3	If two types have the same size in the target configuration, Polyspace no longer raises a violation.
	10.4	If two types have the same size in the target configuration, Polyspace no longer raises a violation.
MISRA C:2012	10.3	If two types have the same size in the target configuration, Polyspace no longer raises a violation.

Standard	Rule	Change
	10.6	If two types have the same size in the target configuration, Polyspace no longer raises a violation.
	10.7	If two types have the same size in the target configuration, Polyspace no longer raises a violation.
	10.8	If two types have the same size in the target configuration, Polyspace no longer raises a violation.

Updated Bug Finder defect checkers

For the new defects that explicitly correspond to CERT-C rules, see “CERT C Support: Identify CERT C violations using defect checkers and coding rules” on page 14-9.

Numerical

Name	Description	Update
Absorption of float operand	In an addition or subtraction, one operand is absorbed by the other and has no effect on the result	New defect

Programming

Name	Description	Update
Typedef mismatch	Mismatch between typedef statements	New defect

Static Memory

Name	Description	Update
Unreliable cast of function pointer	A function pointer is cast to another function pointer with different argument or return type	You can check C++ code for this defect.

Concurrency

Name	Description	Update
Data race	Multiple tasks perform unprotected non-atomic operations on shared variables	<p>You can see a graphical view of the call sequence leading to conflicting operations on the shared variable.</p> <p>If you have existing critical sections, this graph also shows you the critical sections. Using this information, you can easily identify how to protect the shared variable from concurrent access.</p>

Data Flow

Name	Description	Update
Write without a further read	Variable not read after assignment	The defect does not appear if the variable that is assigned the value NULL and not read again.

Reviewing Results


Data Race Graphs: Fix data race defects easily using graphical view of function call sequence

In R2016b, you can use a new graphical view to determine fixes for concurrency defects such as Data race. For each pair of conflicting operations on a shared variable, the graphical view shows:

- Two function call sequences leading to the two operations.

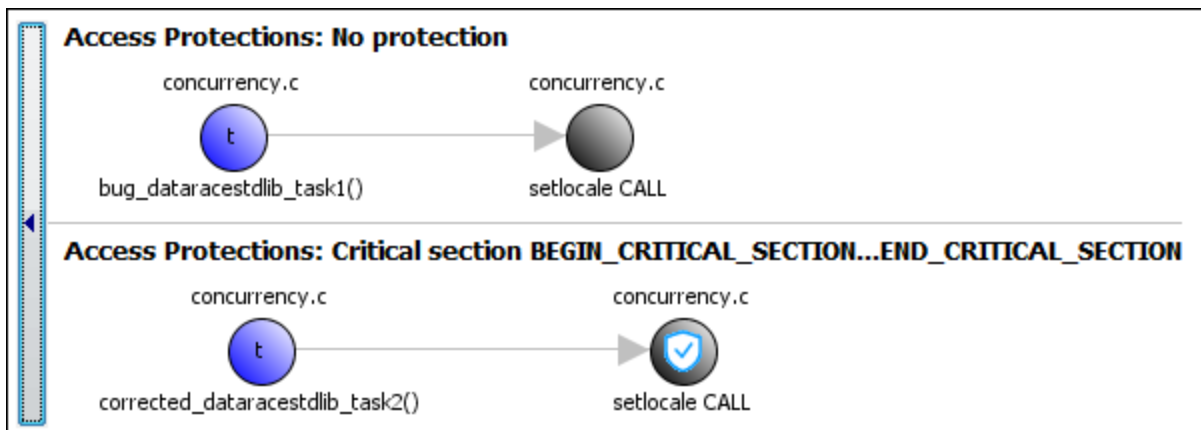
The first node in each sequence represents the entry point function. The last node represents the operation. The intermediate nodes represent functions call sequence leading from the entry point to the operations. To navigate to a function in your source code, click the corresponding node in the graph.

- Critical sections that are already active when a function is called.

If certain critical sections are active when a function is called, the corresponding node in the graph shows a  icon. To see which critical sections are active, place your cursor on the node.

Using this information, you can easily determine how to place appropriate protections and prevent two operations in different tasks/threads from conflicting with each other.

For instance, the following graph shows two tasks calling the function `setlocale`. The two calls are not protected by the same critical section even though the second call uses a critical section. To protect the two calls from interfering with each other, see the **Access Protections** entry for the critical section on the second call and reuse this critical section for the first call.



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 the results that you want to focus on. To see all results again, clear your filters in one click.

To filter results, you can use the following graphs:

- **Defect distribution by impact:** If you click a region on this pie chart that corresponds to the impact **High**, the **Results List** pane shows high-impact defects only.
- **Defect distribution by category (Top 10 only):** If you click a column corresponding to a defect, the **Results List** pane shows instances of that defect only.
- **Coding rule violations by rule (Top 10 only):** If you click a column corresponding to a coding rule, the **Results List** pane shows violations of that rule only.

For more information, see Filter and Group Results.

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.

MISRA C:2012 5.1 (Required) ? External identifiers shall be distinct. External variable engine_temperature_scaled conflicts with the external identifier engine_temperature_raw (file.c line 1).				
	Event	File	Scope	Line
1	Violation site	file.c	file.c	1
2	MISRA C:2012 5.1	file.c	File Scope	2

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.

Results in Macros Consolidated: View coding rule violations and defects 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 and defects caused by a macro are now shown on the macro definition. This change reduces the number of results with the same root cause, making your review process simpler.

Analysis 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 defects, coding rules, or code metrics you want to see. For more information, see Limit Display of Defects.

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 high-impact defects and create a report with those defects 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 using MATLAB or Excel® and generate graphs or statistics about your results that you cannot obtain readily from the user interface.

For more information, see [Export Results to Text File](#).

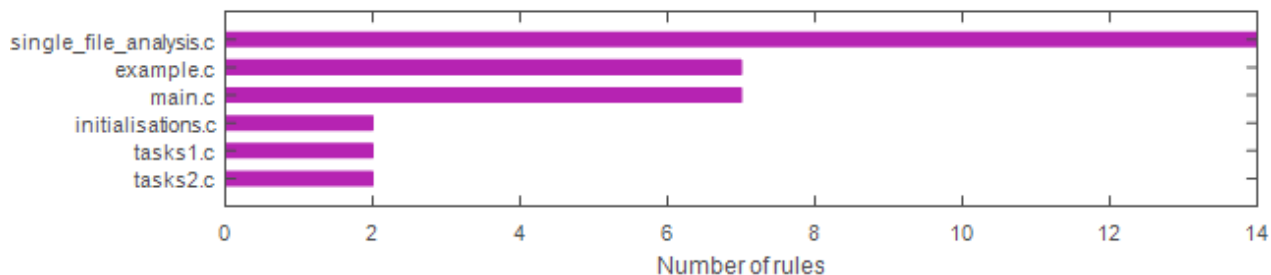
Coding Rules in Report: View improved presentation of coding rules violations in report

In R2016b, the following improvements have been made in how coding rule violations appear in the report.

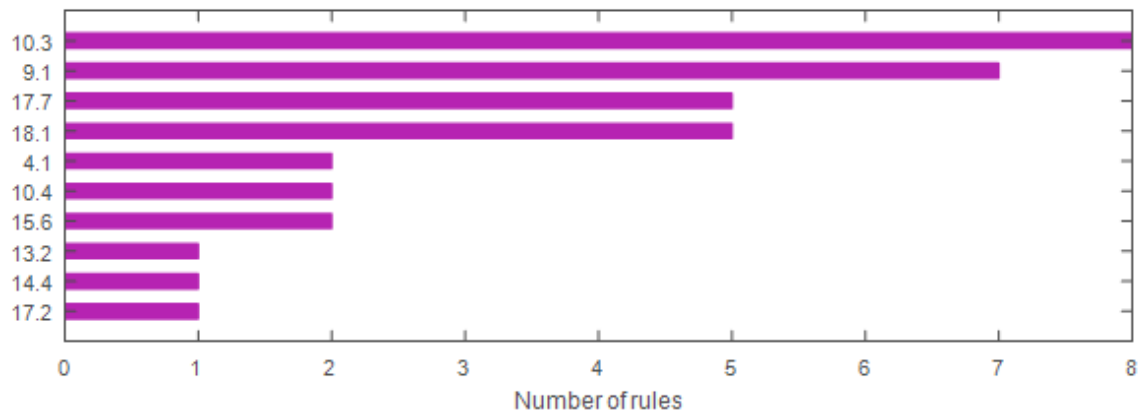
Coding Rule Graphs

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.



Coding Rule Template

You can now create a report that shows coding rules violation only. The report does not show other Polyspace Bug Finder results.

For more information, see the description of template `CodingRules` in Report template (-report-template).

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 Reports from Command Line](#).

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.


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.

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 below.



In R2016b, the same icons are arranged as below.



R2016a

Version: 2.1

New Features

Bug Fixes

Compatibility Considerations

Analysis 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 results 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, results are not generated for headers unless they are in the same location as source files. Previously, if you ran an analysis at the command line, by default, results were generated for all headers.

Due to the change in default behavior, if you rerun the analysis on a pre-R2016a project without explicitly 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 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 quick 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 analysis 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 every instance of the S-Function by analyzing with the different signal range inputs, or just a single instance of the S-Function analyzing with the specific signal ranges for that block.

Import signal ranges from model for generated code analysis

When you run a Polyspace Bug Finder analysis from Simulink, you can now include the signal range information with your analysis. The signal ranges become constraint specifications (formerly called DRS) for the variables in your analysis. For more information see, [Configure Data Range Settings and Constraints](#).

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`

Polyspace Metrics Interface Updated: View project and metrics summary and defect impact

The Polyspace Metrics web interface has been updated to include new features:

- The Bug Finder analysis uploaded to Polyspace Metrics now includes new metrics summarizing the number of defects with High, Medium, and Low impact. For more information on the impact classification, see [Classification of Defects by Impact](#).
- 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.

For more information, see [View Projects in Polyspace Metrics](#).

Source Code Search: Search huge applications more quickly

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

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

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

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

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

For more information, see [Organize Layout of Polyspace User Interface](#).

Files Not Compiled: Receive alerts about compilation errors in dashboard and reports

If some of your source files contain compilation errors, Polyspace Bug Finder analyzes those files only for code metrics and some coding rules.

In R2016a, if some of your files are analyzed only partially because of compilation errors:

- On the **Dashboard** pane, you can see that some files failed to compile. Further information about the compilation errors is available on the **Output Summary** pane. For more information, see [Dashboard](#).
- If you generate reports by using the `BugFinderSummary` or `BugFinder` template, the chapter **Polyspace Bug Finder Summary** lists the files that are partially analyzed. For more information, see [Report template \(-report-template\)](#).

Project Language Flexibility: Change your project language at any time

Projects in the Polyspace interface are no longer fixed to one language.

When you create your projects, you can add any file to the project. After you add files, select the language (C, C++, or C/C++) 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 15-5.

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 c-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 at Command Line.

- The support for IAR compilers has improved. All variations of IAR compilers are now supported for automatic project creation from build command.

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.

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.

Updated Options

Option	Change	More Information
Source code language (-lang)	New value c	Select your project language to set compilation rules and enable language specific analysis options.
Dialect (-dialect)	Unified dialects for C, C/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 C, C++, C/C++	
Respect C90 standard (-no-language-extensions)	Allowed for mixed C/C++ projects	
Pack alignment value (-pack-alignment-value)	Allowed for C, C++, C/C++	
Import folder (-import-dir)	Allowed for C, C++, C/C++	
Ignore pragma pack directives (-ignore-pragma-pack)	Allowed for C, C++, C/C++	
Division round down (-div-round-down)	Allowed for C, C++, C/C++	

Removed Options

Option	Status	Description
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.
-support-FX-option-results	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.

Analysis Results

Improvements to defect checkers

In R2016a, there are improvements in detection of certain defects. For instance, with the checkers for defects Dead code and Useless if:

- You see the code sequence leading to the defect in a greater number of situations. For more information, see Navigate to Root Cause of Defect.
- You see fewer false positives. For instance, you do not see false **Dead code** or **Useless if** defects associated with the following constructs:
 - `_setjmp`
 - Pointer parameter pointing to a global variable
- You do not see defects in templates.

Improvements in checking of previously supported MISRA C rules

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

MISRA C:2004 Rules

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

MISRA C:2012 Rules

Rule	Description	Improvement
MISRA C:2012 Rule 13.2	The value of an expression and its persistent side effects 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 <code>ans = (val+ +, val++)</code> does not violate this rule.

Standards Mapped to Defects: Observe coding standards using Polyspace Bug Finder**CERT C mapping**

In R2016a, you can now observe coding standards such as SEI CERT C Coding Standards by using Polyspace Bug Finder.

For more information, see [Mapping Between CERT C Standards and Defects](#).

CWE ID mapping

In R2016a, the following changes have been made in the mapping between CWE IDs and Polyspace Bug Finder defects.

Defect	CWE ID: Prior to R2016a	CWE ID: R2016a
Invalid use of standard library integer routine	CWE-369: Divide By Zero	<ul style="list-style-type: none"> • CWE-227: Improper fulfillment of API contract • CWE-369: Divide By Zero • CWE-682: Incorrect Calculation • CWE-872: CERT C++ Secure Coding Section 04 - Integers (INT)

For more information, see [Mapping Between CWE Identifiers and Defects](#).

Reviewing Results

More results available in real time

When you run a Bug Finder analysis, more results for blocks of code are now available while the analysis is running. For information about how to open results during the analysis, see [Open Results](#).

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

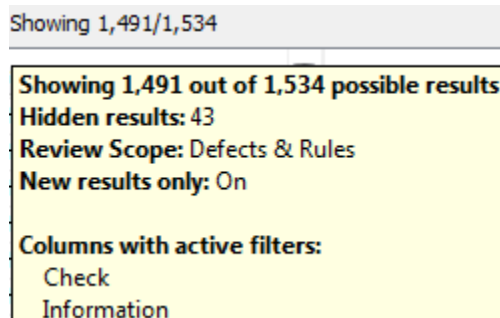
In R2016a, on the **Results Summary** or **Result Details** pane, if you start typing a review comment that you have previously entered, a drop-down list shows the previous entry. Select the previous comment from this list instead of retyping the comment.

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**.


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

In R2016a, if you apply a set of filters to your analysis results and rerun analysis on the project, 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 **Defects & Rules** filter to suppress code metrics and global variables.
- The  **New** filter to suppress results found in a previous analysis.
- Filters on the **Information** and **Check** columns.

For more information, see [Filter and Group Results](#).

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.

R2015aSP1

Version: 1.3.1

Bug Fixes

R2015b

Version: 2.0

New Features

Bug Fixes

Compatibility Considerations

Analysis Setup

Mixed C/C++ Code: Run analysis on entire project with C and C++ source files

If your coding project contains C and C++ files, you can now analyze the entire project in one Polyspace project. Use the new C/C++ setting to compile .c files with C compilation rules and compile .cpp and other files with C++ compilation rules.

To create a mixed C and C++ project:

- At the command line, use the option `-lang C-CPP`.
- In the user interface:
 - 1 Select **File > New Project**.
 - 2 In the Project properties window, select **Project Language > C++** as the main project language. Enter your other project properties as before.
 - 3 When adding source files, add your .c and .cpp files with their include files.
 - 4 In the configuration, on the **Target & Compiler** pane, set **Source code language > C-CPP**. This setting indicates to the compiler to use C compilation rules for .c files and C++ compilation rules for .cpp files. For other file extensions, Polyspace uses C++ compilation rules.
 - 5 Set your other options as required. Some limitations to consider:
 - Coding rules — You can select only one C coding rule set and one C++ coding rule set.
 - Bug Finder Defects — You can select C/C++ or C++ defects. The C++ defects are checked only on .cpp files.

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

If you use POSIX or VxWorks to perform multitasking, Polyspace can now interpret your multitasking code more easily.

Functions Polyspace can interpret:

POSIX

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

VxWorks

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

By default in R2015b, Polyspace detects thread creating and critical sections from supported multitasking functions.

For more information, see [Modeling Multitasking Code](#).

Microsoft Visual C++ 2013: 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`. Once you specify your dialect, Microsoft Visual C++ allows language extensions specific to Microsoft Visual C++ 2013. For more information, see [Dialect \(C\)](#) or [Dialect \(C++\)](#).

GNU 4.9 and Clang 3.5 Support: Analyze code compiled with GNU 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 \(C\)](#) or [Dialect \(C++\)](#).

Improvements to automatic project creation from build command

In R2015b, automatic project creation from your build command is improved:

- If you build your source code from the Cygwin environment (using either a 32-bit or 64-bit installation), Polyspace can trace your build and to 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 oriented with Polyspace Bug Finder

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

- Open Polyspace recent results and 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, the pane appears each time you open Polyspace Bug Finder. 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. You can switch between saved layouts quickly. For instance:

- You can have separate layouts for project configuration and results review.
- You can have a minimal layout with only the 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 **Coding Rules** node is renamed **Coding Rules & Code Metrics**.
The new **Coding Rules & Code Metrics** node now contains the option **Calculate Code Metrics**, which previously appeared in the **Advanced Settings** node.
- On the **Results Summary** pane, the **Category** column title is changed to **Group**. This change avoids 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 specify analysis options multiple times. This new capacity is available only at the command line or using the command-line names in the **Advanced options** pane in the user interface. You can customize pre-made configurations without having to remove options.

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`.

In the user interface, if you specify `c18` as the target on the Target and Compiler pane and in **Advanced options** enter `-target i386`, these two targets count as multiple analysis option specifications. Polyspace uses the target specified in the Advanced options dialog box, `i386`.

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.

Changes in analysis options

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

New Options

Option	Status	Description
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.
Source code language (C++) (-lang)	New in the user interface	The -lang option is now available in the Polyspace user interface. It is on the Target & compiler tab and called Source code language .
Source code language (C++) > C-CPP (-lang C-CPP)	New option setting	For C++ projects, you can choose C-CPP to analyze a mix of .c and .cpp source files.
Configure multitasking manually (C/C++)	New	A user interface option only. This option enables the previous multitasking options <ul style="list-style-type: none"> • Entry points • Critical section details • Temporally exclusive tasks
Disable automatic concurrency detection (C/C++)	New	By default, the new automatic concurrency detection is enabled. If you want to turn it off, select this option.

Updated Options

Option	Change	Description
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.
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	
Targets: <ul style="list-style-type: none"> • 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 now match for C and C++. Available values: <ul style="list-style-type: none"> • defined-by-standard (default) • auto-signed-first • auto-unsigned-first
-support-FX-option-results	No longer available in the user interface	
-pointer-is-24bits	Available in C++ projects	Available only if you use the Target setting c18.
-asm-begin -asm-end	Now available in C++ projects	
Check MISRA C:2004	Now available in C++ projects	Available only if you select Source code language > C-CPP .
Check MISRA AC AGC	Now available in C++ projects	Available only if you select Source code language > C-CPP .
Check MISRA C:2012 and Use generated code requirements (C)	Now available in C++ projects	Available only if you select Source code language > C-CPP .

Option	Change	Description
Effective boolean types (C)	Now available in C++ projects	Available only if you select Source code language > C-CPP .
Allowed pragmas (C)	Now available in C++ projects	Available only if you select Source code language > C-CPP .
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	Description
-dialect cfront2	Removed	Choose a different dialect.
-dialect cfront3	Removed	Choose a different dialect.
-passes-time	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.
-discard-asm	Removed	This option is no longer supported. Remove this option from existing configurations.
-misra2 AC-AGC-OBL-subset	Removed	Use <code>-misra-ac-agc OBL-rules</code> instead.

Compatibility Considerations

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

Binaries removed

The following binaries have been removed.

Removed binary	Use instead
polyspace-rl-manager.exe	polyspace-server-settings.exe
polyspace-spooler.exe	polyspace-job-monitor.exe
polyspace-ver.exe	polyspace-bug-finder-nodesktop -ver

The binaries to use instead 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](#).

Analysis Results

More Defect Categories: Detect security vulnerabilities, resource management issues, object oriented design issues

You can check your code against five new categories of defects:

- Resource management — Defects related to resource handling such as detection of unclosed file descriptors or use of a closed file descriptor.
- Object oriented — Defects related to C++ object-oriented programming such as detection of class design issues or issues in the inheritance hierarchy.
- Security — Defects related to security vulnerabilities such as vulnerable standard functions, use of sensitive data, and pseudo-random number generation.
- Tainted data — Defects related to using variables that someone outside your program can manipulate and externally controlled resources.
- Good practice — Defects that allow you to observe good coding practices such as detection of hard-coded memory buffer size or unused function parameters.

For information about the new defects, see “Changes to Bug Finder Defects” on page 17-12.

Complete MISRA C:2012 Support: Detect violations of all MISRA C:2012 rules

In R2015b, Polyspace Bug Finder supports the following MISRA C: 2012 coding rules.

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 Directive 4.13	Functions which are designed to provide operations on a resource should be called in an appropriate sequence.
MISRA C:2012 Rule 2.6	A function should not contain unused label declarations.
MISRA C:2012 Rule 2.7	There should be no unused parameters in functions.
MISRA C:2012 Rule 17.5	The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.
MISRA C:2012 Rule 17.8	A function parameter should not be modified.
MISRA C:2012 Rule 21.12	The exception handling features of <code><fenv.h></code> should not be used.
MISRA C:2012 Rule 22.1	All resources obtained dynamically by means of Standard Library functions shall be explicitly released.
MISRA C:2012 Rule 22.2	A block of memory shall only be freed if it was allocated by means of a Standard Library function.

Rule	Description
MISRA C:2012 Rule 22.3	The same file shall not be open for read and write access at the same time on different streams.
MISRA C:2012 Rule 22.4	There shall be no attempt to write to a stream which has been opened as read-only.
MISRA C:2012 Rule 22.5	A pointer to a FILE object shall not be dereferenced.
MISRA C:2012 Rule 22.6	The value of a pointer to a FILE shall not be used after the associated stream has been closed.

Improvements in checking of previously supported MISRA C rules

In R2015b, the following changes have been made in MISRA C checking:

MISRA C:2004

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 <code>extern</code> 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.
MISRA C:2004 Rule 19.2	Nonstandard characters should not occur in header file names in <code>#include</code> directives.	If the character <code>\</code> or <code>\\</code> occurs between the <code><</code> and <code>></code> in <code>#include <filename></code> (or between <code>"</code> and <code>"</code> in <code>#include "filename"</code>), Polyspace no longer raises violation of this rule. Therefore, you can use Windows paths to files in place of <code>filename</code> without triggering a rule violation.

MISRA C:2012

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.	<p>If a rule violation occurs because your .c file contains too many macros, Polyspace places the rule violation at the beginning of the file instead on the last macro usage.</p> <p>Therefore, you can add a comment before the first line of the .c file justifying the violation. Previously, if you placed a justification comment before the last macro usage and later added another macro usage, the comment no longer applied. For information on adding code comments to justify results, see Annotate Code for Rule Violations.</p>
MISRA C:2012 Rule 10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.	<ul style="list-style-type: none"> • If one of the operands is the constant zero, Polyspace does not raise a violation of this rule. • If one of the operands is a signed constant and the other operand is unsigned, the rule violation is not raised if the signed constant has the same representation as its unsigned equivalent. <p>For instance, the statement <code>u8b = u8a + 3;</code>, where <code>u8a</code> and <code>u8b</code> are unsigned char variables, does not violate the rule because the constants 3 and 3U have the same representation.</p>

Checking Coding Rules Using Text Files

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

For more information on checking for coding rules using text files, see Format of Custom Coding Rules File.

Changes to Bug Finder Defects

- “New Defects” on page 17-13
- “Updated Defects” on page 17-18

The following tables list updates and additions to the list of Bug Finder defect checkers.

New Defects

Tainted Data Defects

Name	Description
Array access with tainted index	Array index from unsecure source possibly outside array bounds
Command executed from externally controlled path	Path argument from an unsecure source
Execution of externally controlled command	Command argument from an unsecure source is vulnerable to OS command injection
Host change using externally controlled elements	Changing host id from an unsecure source
Library loaded from externally controlled path	Library argument from an externally controlled path
Loop bounded with tainted value	Loop controlled by a value from an unsecure source
Memory allocation with tainted size	Size argument to memory function is from an unsecure source
Pointer dereference with tainted offset	Offset is from an unsecure source and dereference may be out of bounds
Tainted division operand	Division operands from an unsecure source
Tainted modulo operand	Remainder operands from an unsecure source
Tainted NULL or non-null-terminated string	Argument is from an unsecure source and may be NULL or not NULL-terminated
Tainted sign change conversion	Value from an unsecure source changes sign
Tainted size of variable length array	Size of the variable-length array (VLA) is from an unsecure source and may be zero, negative, or too large
Tainted string format	Input format argument is from an unsecure source
Use of externally controlled environment variable	Value of environment variable from an unsecure source
Use of tainted pointer	Pointer from an unsecure source may be NULL or point to unknown memory

Good Practice Defects

Name	Description
Delete of void pointer	<code>delete</code> operates on a <code>void*</code> pointer pointing to an object
Hard coded buffer size	Size of memory buffer is a numerical value instead of symbolic constant
Hard coded loop boundary	Loop boundary is a numerical value instead of symbolic constant
Unused parameter	Function prototype has parameters not read or written in function body
Use of <code>setjmp/longjmp</code>	<code>setjmp</code> and <code>longjmp</code> cause deviation from normal control flow

Programming Defects

Name	Description
Bad file access mode or status	Access mode argument of function in <code>fopen</code> or <code>open</code> group is invalid
Call to <code>memset</code> with unintended value	<code>memset</code> or <code>wmemset</code> used with possibly incorrect arguments
Copy of overlapping memory	Source and destination arguments of a copy function have overlapping memory
Exception caught by value	<code>catch</code> statement accepts an object by value
Exception handler hidden by previous handler	<code>catch</code> statement is not reached because of an earlier <code>catch</code> statement for the same exception
Improper array initialization	Incorrect array initialization when using initializers
Incorrect pointer scaling	Implicit scaling in pointer arithmetic might be ignored
Invalid assumptions about memory organization	Address is computed by adding or subtracting from address of a variable
Invalid <code>va_list</code> argument	Variable argument list used after invalidation with <code>va_end</code> or not initialized with <code>va_start</code> or <code>va_copy</code>
Modification of internal buffer returned from nonreentrant standard function	Function attempts to modify internal buffer returned from a nonreentrant standard function
Overlapping assignment	Memory overlap between left and right sides of an assignment
Possible misuse of <code>sizeof</code>	Use of <code>sizeof</code> operator can cause unintended results
Possibly unintended evaluation of expression because of operator precedence rules	Operator precedence rules cause unexpected evaluation order in arithmetic expression
Standard function call with incorrect arguments	Argument to a standard function does not meet requirements for use in the function
Use of <code>memset</code> with size argument zero	Size argument of function in <code>memset</code> family is zero
Variable length array with nonpositive size	Size of variable-length array is zero or negative
Writing to <code>const</code> qualified object	Object declared with a <code>const</code> qualifier is modified

Resource Management Defects

Name	Description
Closing a previously closed resource	Function closes a previously closed stream
Resource leak	File stream not closed before FILE pointer scope ends or pointer is reassigned
Use of previously closed resource	Function operates on a previously closed stream
Writing to read-only resource	File opened earlier as read-only is modified

Security Defects

Name	Description
Deterministic random output from constant seed	Seeding routine uses a constant seed making the output deterministic
Execution of a binary from a relative path can be controlled by an external actor	Command with relative path is vulnerable to malicious attack
File access between time of check and use (TOCTOU)	File/directory may have changed state due to access race
File manipulation after chroot() without chdir("/")	Path-related vulnerabilities for file manipulated after call to <code>chroot</code>
Function pointer assigned with absolute address	Constant expression is used as function address is vulnerable to code injection
Incorrect order of network connection operations	Socket is not correctly established due to bad order of connection steps or missing steps
Load of library from a relative path can be controlled by an external actor	Library loaded with relative path is vulnerable to malicious attacks
Mismatch between data length and size	Data size argument is not computed from actual data length
Missing case for switch condition	Default case is missing and may be reached
Predictable random output from predictable seed	Seeding routine uses a predictable seed making the output predictable
Sensitive data printed out	Function prints out sensitive data
Sensitive heap memory not cleared before release	Sensitive data not cleared or released by memory routine
Umask used with chmod-style arguments	Unsafe argument to <code>umask</code> allows external user too much control
Uncleared sensitive data in stack	Variable in stack is not cleared and contains sensitive data
Unsafe standard encryption function	Function is not reentrant or uses a risky encryption algorithm
Unsafe standard function	Function unsafe for security-related purposes
Use of dangerous standard function	Dangerous functions cause possible buffer overflow in destination buffer
Vulnerable path manipulation	Path argument with <code>./</code> , <code>/abs/path/</code> , or other unsecure elements
Vulnerable permission assignments	Argument gives read/write/search permissions to external users
Vulnerable pseudo-random number generator	Using a cryptographically weak pseudo-random number generator

Name	Description
Use of non-secure temporary file	Temporary generated file name is unsecure
Use of obsolete standard function	Obsolete routines can cause security vulnerabilities and/or portability issues

Object-Oriented Defects

Name	Description
*this not returned in copy assignment operator	operator= method does not return a pointer to the current object
Base class assignment operator not called	Copy assignment operator does not call copy assignment operators of base subobjects
Base class destructor not virtual	Class cannot behave polymorphically for deletion of derived class objects
Copy constructor not called in initialization list	Copy constructor does not call copy constructors of some members or base classes
Incompatible types prevent overriding	Derived class method hides a <code>virtual</code> base class method instead of overriding it
Missing explicit keyword	Constructor missing the <code>explicit</code> specifier
Missing virtual inheritance	A base class is inherited both virtually and non-virtually in the same hierarchy
Member not initialized in constructor	Constructor does not initialize some members of a class
Object slicing	Derived class object passed by value to function with base class parameter
Partial override of overloaded virtual functions	Class overrides a fraction of the inherited virtual functions with a given name
Return of non const handle to encapsulated data member	Method returns pointer or reference to internal member of object
Self assignment not tested in operator	Copy assignment operator does not test for self-assignment

Updated Defects

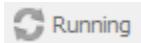
Name	Status	Additional Information
Integer conversion overflow Integer overflow Invalid use of standard library routine Shift operation overflow Sign change integer conversion overflow Shift of a negative value Unsigned integer conversion overflow Unsigned integer overflow	Updated	The defects do not appear on computations involving constants only. For instance, the assignment <code>unsigned int var = -1;</code> does not show a Sign change integer conversion overflow defect.
Format string specifiers and arguments mismatch	New category	Moved from Other to Programming
Invalid use of standard library routine	New category	Moved from Other to Programming
Assertion	New category	Moved from Other to Good practice
Large pass-by-value argument	New category	Moved from Other to Good practice
Line with more than one statement	New category	Moved from Other to Good practice

Reviewing Results

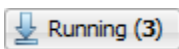
Results in Real Time: View results as they are produced

Previously, you could not review results until the analysis was complete. For local analyses in R2015b, you can start reviewing results as soon as they are available.

When you run a local analysis, a new button appears on the toolbar.

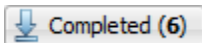


When results are available, this button becomes active.



To start reviewing available results, click this button. The button reactivates every time results are available. To load additional results, click the button again.

When the analysis is complete, to load all your results, click the button.



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


Improved Eclipse Support: View results embedded in source code and context-sensitive help

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

- Polyspace Bug Finder highlights defects in your source code in the following ways:
 - For defects, an ! mark appears before the line number on the left. For coding rule violations, a ▾ or ▼ mark appears before the line number on the left.
 - The operation containing the defect has a wavy red underlining.
 - For defects, a ■ icon appears in the overview ruler to the right of the line containing the defect. For coding rule violations, a ■ icon appears in the overview ruler to the right of the line containing the rule violation. If you place your cursor on the icon, a tooltip shows a brief description of the defect or coding rule.







In addition, a ■ icon appears at the top of the overview ruler. If you place your cursor on the icon, a tooltip states the total number of defects and coding rule violations in the file.

Using these indicators, you can track defects in your source code more easily. For more information, see [Review and Fix Results](#).

- When you select a result in the **Results Summary - Bug Finder** 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. For defects, you can sometimes see the risk associated with not fixing the defect and the most common fix for the defect.

- You can switch to a Polyspace perspective that shows only the information relevant to a Polyspace Bug Finder analysis. To open the perspective, select **Window > Open Perspective > Other**. In the Open Perspective dialog box, select **Polyspace**.

Once you switch to the Polyspace perspective, the source code shows the Polyspace Bug Finder defects only in this perspective.

- You can view results as they are produced instead of waiting till end of the analysis.
 - When you begin an analysis, a  icon appears next to the  button.
 - If results are available, the icon turns to . Click the  icon to load available results.
 - With your results open, if additional results are available, the  icon is still visible. Click the  icon to load all available results.

Defects Classified by Impact: Prioritize defect review by using the impact attribute assigned to each defect type

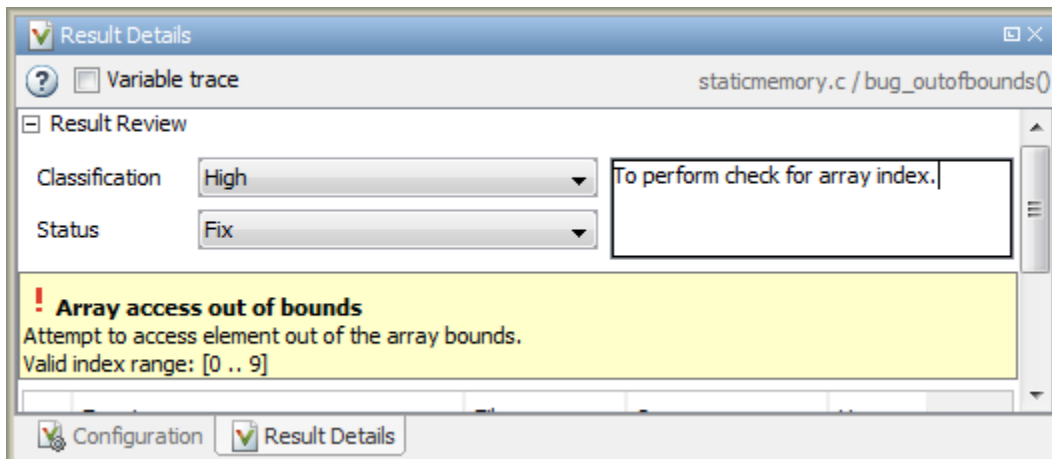
You can prioritize your result review using an **Impact** attribute assigned to the defects. The attribute is assigned based on the following considerations:

- Criticality, or whether the defect is likely to cause a code failure.
- Certainty, or the rate of false positives.

You can filter results on the **Results Summary** pane using the **Impact** attribute. Or, you can obtain a graphical visualization of the **Defect distribution by impact** on the **Dashboard** pane. For more information, see Classification of Defects by Impact.

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

In R2015b, the **Check Details** pane is renamed as **Result Details**. On this pane, you can now enter review information such as **Classification**, **Status**, and comments. For more information, see Review and Fix Results.



Previously, to enter review information while keeping the **Results Summary** pane collapsed, you used the **Check Review** pane. This pane has been removed.

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

Previously, using custom options on the **Show** menu, you suppressed only defects and code metrics (if they fell below a certain threshold). In R2015b, you can suppress a certain number or percentage of coding rule violations from the display. You use custom options in the **Show** menu on the **Results Summary** pane. You can:

- Suppress violations of coding rules that are not relevant.
- 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 and 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 information on how to create an option to suppress coding rule violations, see [Suppress Certain Rules from Display in One Click](#).

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 in the **Project Browser**, you can also right-click the **Results** node in the project and select **Open Configuration**.

Improvements in Report Templates

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

- The summary chapter in the template **BugFinder** now contains a breakup of Polyspace Bug Finder results by file, in addition to the project-wide summary.
- The summary now shows the total number of results along with the number of results reviewed.
- 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.
- The reports display the impact attribute associated with a defect.

For more information on this attribute, see [Classification of Defects by Impact](#).

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

XML and RTF report formats removed

The formats XML and RTF for report generation are not 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: 1.3

New Features

Bug Fixes

Compatibility Considerations

Analysis Setup

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

In R2015a, the Project and Results Manager perspectives have been unified. You can run the analysis and review results without switching between two perspectives.

The unification has resulted in the following major changes:

- After an analysis, the result opens automatically.

Previously, after an analysis, you had to double-click the result in the **Project Browser** to open your new results.

- You can have any of the panes open in the unified interface.

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

Project Manager	Results Manager
<ul style="list-style-type: none"> • Project Browser: Set up project. • Configuration: Specify analysis options for your project. • Output Summary: Monitor progress of analysis. • Run Log: Find information about an analysis. 	<ul style="list-style-type: none"> • Results Summary: View Polyspace results. • Source: View read-only form of source code color coded with Polyspace results. • Check Details: View details of a particular result. • 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.

Search improvements 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 previously available search options.

Option to specify program termination functions

In R2015a, you can specify functions that behave like the exit function and terminate your program.

- At the command line, use the flag `-termination-functions`.
- In the user interface, on the **Configuration** pane, select **Advanced Settings**. Enter `-termination-functions` in the **Other** field.

For more information, see `-termination-functions`.

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 Bug Finder analysis, set the **Target & Compiler > Dialect** option to `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`.

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 model and highlights the related block.

Note This feature works only with Simulink R2013b and later.

For more information about the back-to-model feature, see `Review Generated Code Results`.

Polyspace binaries being removed

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

Binary name	Use instead
polyspace-rl-manager.exe	polyspace-server-settings.exe
polyspace-spooler.exe	polyspace-job-monitor.exe
polyspace-ver.exe	polyspace-bug-finder-nodesktop -ver

Import Visual Studio project being removed

The **Tools > 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 [Create Project Automatically](#).

Analysis Results

Changes to Bug Finder defects

Defect	R2015a change
Invalid use of floating point operation	Off by default.
Line with more than one statement	Off by default.
Invalid use of = (assignment) operator	On by default for handwritten code (analyses started at the command-line or Polyspace environment). Off by default for generated code (analyses started from the Simulink plug-in).
Invalid use of == (equality) operator	On by default for handwritten code. Off by default for generated code.
Missing null in string array	On by default for handwritten code. Off by default for generated code.
Partially accessed array	On by default for handwritten code. Off by default for generated code.
Variable shadowing	On by default for handwritten code. Off by default for generated code.
Write without further read	On by default for handwritten code. Off by default for generated code.
Wrong type used in sizeof	On by default for handwritten code. Off by default for generated code.

Improvements in coding rules checking

MISRA C:2004 and MISRA AC AGC

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

MISRA C:2012

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

MISRA C++:2008

Rule Number	Effect	More Information
Rule 5-0-3	Fewer results on enumeration constants when the type of the constant is the enumeration type.	MISRA C++ Rules — Chapter 5
Rule 6-5-1	Fewer results on compliant vector variable iterators.	MISRA C++ Rules — Chapter 6
Rule 14-8-2	Fewer results for functions contained in the Files and folders to ignore (C++) option.	MISRA C++ Rules — Chapter 14
Rule 15-3-2	Fewer results for user-defined return statements after a <code>try</code> block.	MISRA C++ Rules — Chapter 15

Reviewing Results

Code complexity metrics available in user interface

In R2015a, code complexity metrics can be viewed 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 exceeds this limit, the metric appears red in **Results Summary**.
- Comment and justify the value of a metric. If a metric value exceeds specified limits and appears red, you can add a comment with the rationale.

Using Polyspace results in this way, 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 analysis.

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 metrics results, 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 latest results compared to the last run

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

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

If you make changes in your source code, you can use this feature to see only the results introduced due to those changes. You can avoid reviewing the results in your existing source code.

Simplified results infrastructure

Polyspace results folders are reorganized and simplified. Files have been removed, combined, renamed, or moved. The infrastructure changes do not change the analysis 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.psbf`. 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.

Default statuses to justify results

Polyspace Bug Finder results use certain statuses to calculate the number of justified results in Polyspace Metrics.

In R2015a, the default statuses that mark results as justified are:

- **Justified** — Previously called **Justify**, renamed in R2015a.
- **No action planned** — Existing status added to justified list in R2015a.

You can change which statuses mark results as justified from the Polyspace preferences. For more information, see Define Custom Review Status.

Filters to limit display of results

In R2015a, you can use the **Show** menu on the **Results Summary** pane to suppress certain Polyspace Bug Finder results from display.

- To suppress code complexity metrics from display, select **Show > Defects & Rules**.
- Create your own options on the **Show** menu. Select **Tools > Preferences** and create new options through the **Review Scope** tab.

For more information, see Limit Display of Defects.

R2014b

Version: 1.2

New Features

Bug Fixes

Compatibility Considerations

Analysis Setup

Parallel compilation for faster analysis

Starting in R2014b, Polyspace Bug Finder can run the compilation phase of your analysis 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 analysis process.

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

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.

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 in Polyspace interface

In R2014b, 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.

Ignore files and folders during analysis

You can now use the analysis option **Files and folders to ignore** (command line `-includes-to-ignore`) to ignore files and folders during defect checking. Previously, the **Files and folders to**

ignore option (command line `-includes-to-ignore`) ignored files and folders during coding rule checking. In R2014b, Polyspace Bug Finder uses this option to ignore specified files or folders for coding rule checking AND defect analysis.

For more information, see Files and folders to ignore (C) or Files and folders to ignore (C++).

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 analysis 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 no longer 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, Polyspace can analyze this generated code. Polyspace uses the same default analysis options and parameters as Embedded Coder.

For more information, see Embedded Coder Considerations.

Remote launcher and queue manager renamed

Polyspace renamed the remote launcher and the queue manager.

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

Compatibility Considerations

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

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 analysis.	Run > Run Report > Run Report	Reporting > Run Report
Open a generated report.	Run > Run Report > Open Report	Reporting > Open Report
Import review comments from a previous analysis.	Review > Import	Tools > Import Comments
Specify code generator for generated code.	Review > Code Generator Support	Tools > Code Generator Support
Specify settings that apply to every Polyspace project.	Options > Preferences	Tools > Preferences
Specify settings for remote analysis.	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 an analysis, you can view its progress on the **Output Summary** tab.
- In the **Project Browser**, projects appear sorted in alphabetical order instead of order of creation.
- On the **Configuration** pane, the **Interactive** option has been removed from the graphical interface. To use the interactive mode, use the `-interactive` flag at the command line, or in the **Advanced Settings > Other** text field. For more information, see `-interactive`

Polyspace binaries being removed

The following 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-rl-manager.exe	Warning	polyspace-server-settings.exe
polyspace-spooler.exe	Warning	polyspace-job-monitor.exe
polyspace-ver.exe	Warning	polyspace-bug-finder-nodesktop -ver
setup-remote-launcher.exe	Warning	<i>matlabroot</i> /toolbox/polyspace / psdistcomp/bin/setup-polyspace-cluster

Import Visual Studio project being removed

The **File > Import Visual Studio project** will be removed in a future release. Instead, use the **Create from build system** option during New Project creation. For more information, see [Create Projects Automatically from Your Build System](#).

Analysis 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.

Additional concurrency issue detection (deadlocks, double locks, and others)

Data race errors

The following defects deal with unprotected access of shared variables by multiple tasks.

Defect name	Status	More information
Race conditions	Removed	Replaced by Data race and Data race including atomic operations.
Data race	New	Checks for unprotected operations on variables shared by multiple tasks. This check applies to non-atomic operations only.
Data race including atomic operations	New	Checks for unprotected operations on variables shared by multiple tasks. This check applies to all operations, including atomic ones.

Locking errors

The following defects deal with incorrect design of critical sections. For multitasking analysis, to mark a section of code as a critical section, you must place it between two function calls. A lock function begins a critical section. An unlock function ends a critical section.

Defect name	Status	More information
Deadlock	New	Checks whether the sequence of calls to lock functions is such that two tasks block each other.
Missing lock	New	Checks whether an unlock function has a corresponding lock function.
Missing unlock	New	Checks whether a lock function has a corresponding unlock function.

Defect name	Status	More information
Double lock	New	Checks whether a lock function is called twice in a task without an unlock function being called in between.
Double unlock	New	Checks whether an unlock function is called twice in a task without a lock function being called in between.

For more information, see:

- Set Up Multitasking Analysis
- Review Concurrency Defects

New and updated defect checkers

Defect name	Status	More information
Dead code	Updated	Checks for non-executed code. No longer checks for: <ul style="list-style-type: none"> • <code>if</code> conditions that are always true without a corresponding <code>else</code>. This check is covered by the Useless if defect. • Code following control-flow statements such as <code>break</code>, <code>return</code>, or <code>goto</code> defect. This check is covered by the Unreachable code defect.
Useless if	New	Checks for if-conditions that are always true.
Unreachable code	New	Checks for code following control-flow statements such as <code>break</code> , <code>return</code> , or <code>goto</code> .
Declaration mismatch	Updated	Updated for <code>#pragma</code> packing statements.
Race conditions	Removed	Replaced by Data race and Data race including atomic operations.
Data race	New	Checks for unprotected operations on variables shared by multiple tasks. This check applies to non-atomic operations only.
Data race including atomic operations	New	Checks for unprotected operations on variables shared by multiple tasks. This check applies to all accesses, including atomic ones.
Deadlock	New	Checks whether the sequence of calls to lock functions is such that two tasks block each other.
Missing lock	New	Checks whether an unlock function has a corresponding lock function.
Missing unlock	New	Checks whether a lock function has a corresponding unlock function.
Double lock	New	Checks whether a lock function is called twice in a task without an unlock function being called in between.

Defect name	Status	More information
Double unlock	New	Checks whether an unlock function is called twice in a task without a lock function being called in between.

Reviewing Results


Context-sensitive help for analysis options and defects

Contextual help is available for analysis options in the Polyspace interface and its plug-ins. To view the contextual help for analysis options:

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

The documentation for that analysis option appears in a dockable window.

Contextual help is available for defects in the Polyspace interface. To view the contextual help:

- 1 In the Results Manager perspective, select a defect from the Results Summary.
- 2 Inside the **Check Details** pane, select .

The documentation for that Bug Finder defect appears in a dockable window.

For more information, see Getting Help.

Improved Results Manager perspective

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

- To group your defects, use the **Group by** menu on the **Results Summary** pane.
 - To leave your defects ungrouped, instead of **List of Checks**, select **Group by > None**.
 - To group defects by category, instead of **Checks by Family**, select **Group by > Family**.
 - To group defects by file and function, instead of **Checks by File/Function**, select **Group by > File**.
- On the **Source** pane:
 - If a color appears on a brace enclosing a code block, double-click the brace to highlight the block. If no color appears, click the brace once to highlight the code block.
 - If a code block is deactivated due to conditional compilation, it appears gray.

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 an analysis.

Compatibility Considerations

For existing coding rules files, coding rules that use the keyword `error` are treated in the same way as that with keyword `warning`. For more information on `warning`, see Format of Custom Coding Rules File.

R2014a

Version: 1.1

New Features

Bug Fixes

Compatibility Considerations

Analysis 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:

- At the 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 folder from which you run your build command.
 - Additional options. For more information, see [Create Project in User Interface](#).

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

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

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

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

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

For more information, see [Dialects for C](#) or [Dialects for C++](#).

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	R2013b	R2014a
New file wizard for custom coding rules.	<p>For each coding rule, you can select three results:</p> <ul style="list-style-type: none"> • Error: Analysis stops if the rule is violated. 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. 	<p>For each coding rule, you can select two results:</p> <ul style="list-style-type: none"> • On: Analysis continues even if the rule is violated. 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.
Format of the custom coding rules file.	<p>Each line in the file must have the syntax:</p> <pre>rule off error warning #comments</pre> <p>For example:</p> <pre># MISRA configuration - Proj1 10.5 off #don't check 10.5 17.2 error 17.3 warning</pre>	<p>Each line in the file must have the syntax:</p> <pre>rule off warning #comments</pre> <p>For example:</p> <pre># MISRA configuration - Proj1 10.5 off #don't check 10.5 17.2 warning 17.3 warning</pre>

Compatibility Considerations

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

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

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.

Security level support for batch analysis

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 Set MJS Cluster Security.

Interactive mode for remote analysis

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

- 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-bug-finder-nodesktop` command.
- On the MATLAB command line, add the argument `'-interactive'` to the `polyspaceBugFinder` 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.

Function replacement in Simulink plug-in

The following functions have been replaced in the Simulink plug-in by the function `pslinkfun`. These functions will 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')

For more information, see `pslinkfun`

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** does not check the configuration before an analysis.

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

For more information about **Check configuration**, see Check Simulink Model Settings.

Data range specification support

Data range specification (DRS) is available with Polyspace Bug Finder. You can add range information to global variables.

You can also use DRS information with Polyspace Code Prover. Similarly, you can use DRS information from Code Prover in Bug Finder.

For more information, see Inputs & Stubbing.

Polyspace binaries being removed

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

- `polyspace-report-generator.exe`
- `polyspace-results-repository.exe`

- `polyspace-spooler.exe`
- `polyspace-ver.exe`

Analysis Results

Classification of bugs according to the Common Weakness Enumeration (CWE) standard

In R2014a, Polyspace Bug Finder associates CWE™ IDs with many defects. For the covered defects, the IDs are listed in the **CWE ID** column on the **Results Summary** pane. To view the **CWE ID** column, right-click the **Results Summary** tab and select the **CWE ID** column.

For more information, see Common Weakness Enumeration from Bug Finder Defects.

Additional coding rules support (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.

Additional analysis checkers

Polyspace Bug Finder can now check for two additional defects in C and C++:

- **Wrong allocated object size for cast** checks for memory allocations that are not multiples of the pointer size.
- **Line with more than one statement** checks for lines that have additional statements after a semicolon.

For more information, see Wrong allocated object size for cast and Line with more than one statement.

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 analysis represents `float arr = 0.1;` as,

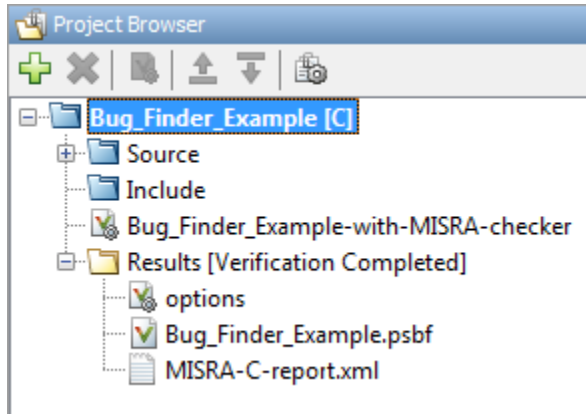
- Pre-R2013b, `arr = [9.9999E^-2, 1.0001E-1]`.
- Now, `arr = 0.1`.

Reviewing Results

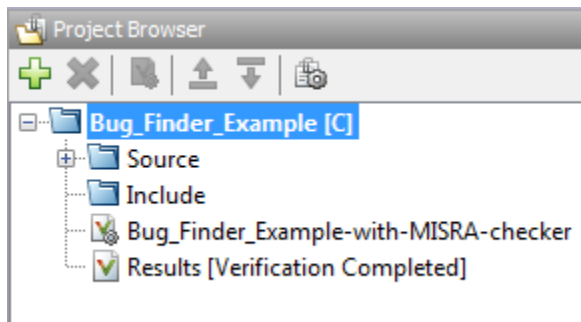
Results folder appearance in Project Browser

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



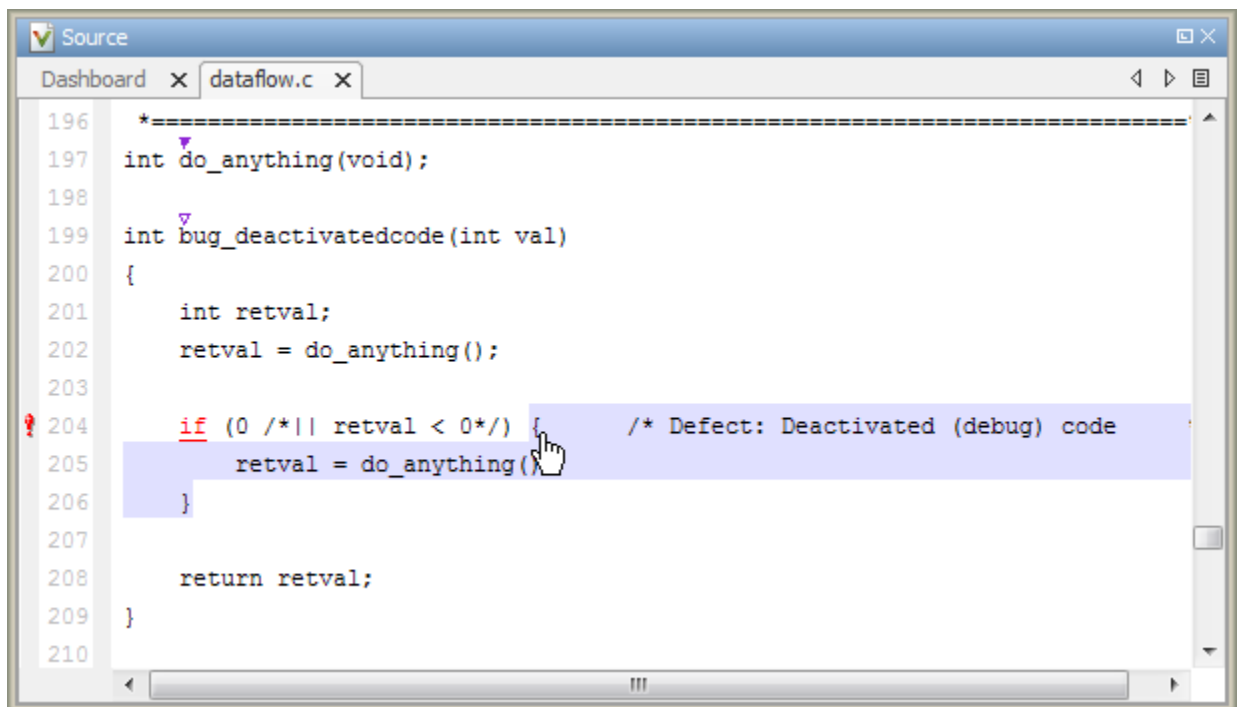
The following table lists the changes in the actions that you can perform on the results folder.

Action	R2013b	R2014a
Open results.	In the result folder, double-click result file with extension <code>.psbf</code> .	Double-click result file.
Open analysis options used for result.	In the result folder, select options .	Right-click result file and select Open Configuration .
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.

Action	R2013b	R2014a
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 example, with a **Dead code** defect for 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. You can see:

- Code covered by analysis.
- Defect distribution. You can choose to view the distribution by:
 - File**
 - Category** or defect name.
- Distribution of coding rule violations. You can choose to view the distribution by:

- **File**
- **Category** or rule number.

The **Dashboard** displays violations of different types of rules such as MISRA C, JSF C++, or custom rules on different graphs.

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 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 other instances of the variable.

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

In R2014a, 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, see [Identify Errors in Simulink Models](#).

R2013b

Version: 1.0

New Features

Analysis Setup

Introduction of Polyspace Bug Finder

Polyspace Bug Finder is a new companion product to Polyspace Code Prover. Polyspace Bug Finder analyzes C and C++ code to find possible defects and coding rule violations. Bug Finder can run fast analyses on large code bases with low false-positive results. Polyspace Bug Finder also calculates code complexity metrics with Polyspace Metrics.

Bug Finder integrates with Simulink, Eclipse, Visual Studio, and Rhapsody to help you analyze code from within your development environment.

Fast analysis of large code bases

Polyspace Bug Finder uses an efficient analysis method which produces results quickly, even from large code bases. Therefore you can fix errors and rerun the analysis without having to wait. You can find more issues early on in the development process and produce better quality code overall.

Eclipse integration

Polyspace Bug Finder comes with an Eclipse plug-in that integrates Polyspace into your development environment. You can set up options, run analyses, view results, and fix bugs in the Eclipse interface. Using the Polyspace plug-in, you can quickly find and fix bugs as you code.

For a tutorial on using the Polyspace Bug Finder plug-in, see [Find Defects from the Eclipse Plug-In](#).

Analysis Results

Detection of run-time errors, data flow problems, and other defects in C and C++ code

Polyspace Bug Finder uses static analysis to find various defects for C and C++ code with few false-positive results. The analysis does not require program execution, code instrumentation, or test cases.

Some categories of defects are:

- Numeric
- Programming
- Static memory
- Dynamic memory
- Data-flow

To see a list of defects you can find, see [Polyspace Bug Finder Defects](#).

Bug Finder analysis runs quickly, so you can fix errors and rerun analysis.

For information about running analyses, see [Find Bugs](#).

Compliance checking for MISRA-C:2004, MISRA-C++:2008, JSF++, and custom naming conventions

Polyspace Bug Finder can also check for compliance with coding rules. There are four industry-defined rules you can select:

- MISRA C
- MISRA AC-AGC
- MISRA C++
- JSF C++

In addition, you can define rules to check for naming conventions.

You can run the coding rules checker separately, or at the same time as your analysis.

For more information, see [Check Coding Rules](#).

Cyclomatic complexity and other code metrics

Using Polyspace Metrics, Polyspace Bug Finder calculates various code metrics, including cyclomatic complexity. These statistics are displayed using Polyspace Metrics, an integrated Web interface. You can use these results to track code quality over time. You can also share the code metrics, allowing others to track your project's progress.

Reviewing Results

Traceability of code analysis results to Simulink models

For generated code from Simulink models, Polyspace analysis results link directly back to your Simulink model. You can trace defects back to the block that is causing the bug.

In the Source Code view of the Results Manager, the block names appear as links. When you select a link, the corresponding block is highlighted in Simulink.

For a tutorial on using Polyspace Bug Finder with Simulink models, see [Find Defects from Simulink](#).

Access to Polyspace Code Prover results

A Polyspace Bug Finder installation also includes the Polyspace Code Prover user interface. With only a Polyspace Bug Finder license, you cannot run local Polyspace Code Prover verifications in the Polyspace Code Prover interface. However, you can use the Polyspace Code Prover interface to review results and upload comments to Polyspace Metrics.

For more information, see the [Polyspace Code Prover Documentation](#).