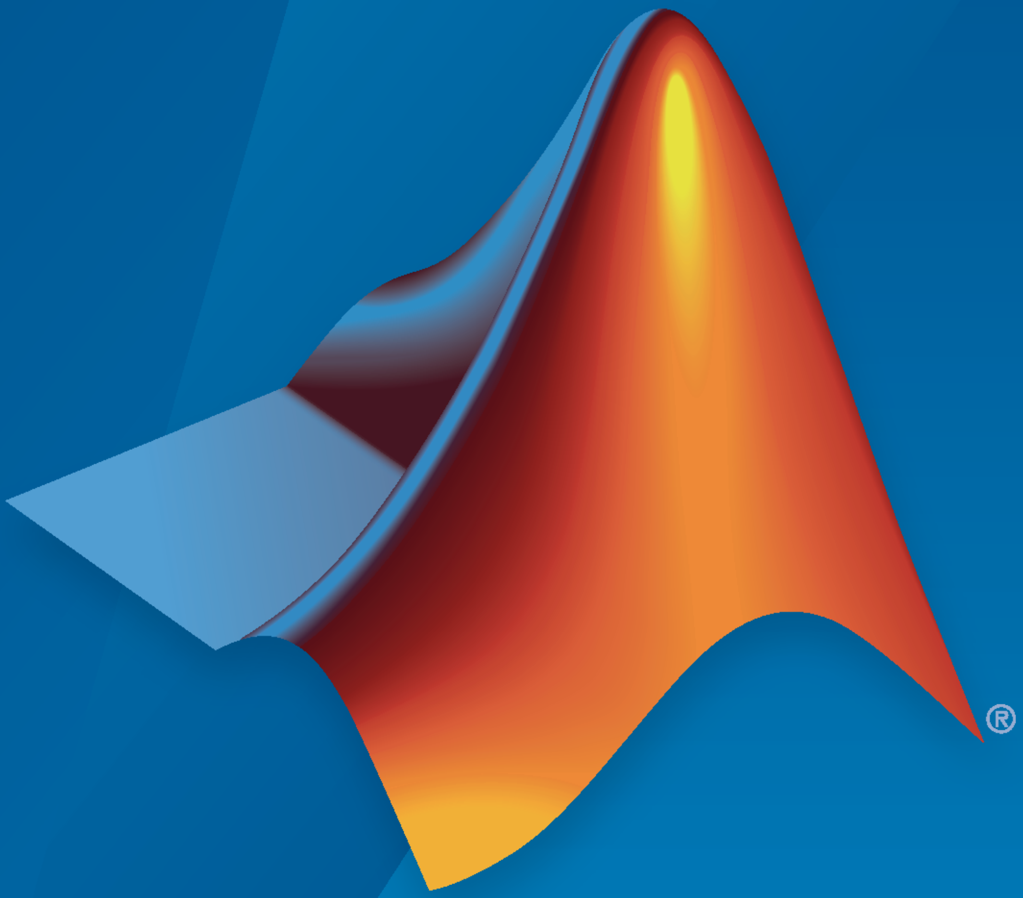
Polyspace® Code Prover™ Server™ Release Notes



MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

Polyspace® Code Prover™ Server™ Release Notes

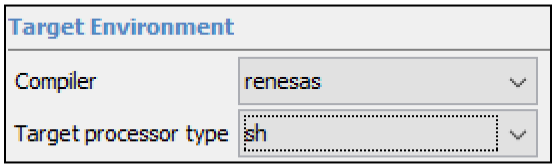
Contents

R2020b

R2020b

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

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


Target Environment

Compiler: renesas

Target processor type: sh

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

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

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

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

See also "Check if Polyspace Supports Build Scripts".

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

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

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

See also:

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

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

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

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

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

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

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

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

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

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

See also `polyspace-configure`.

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

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

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

See also `polyspace-configure`.

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

Offloading Analysis: Submit Polyspace analysis jobs from CI server to a dedicated analysis cluster

Summary: 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 a one or several servers. In the latter case, a head node distributes the jobs to several worker nodes which perform the analysis. MATLAB® Parallel Server™ is required on all servers involved in distributing jobs or running the analysis.

See “Offload Polyspace Analysis from Continuous Integration Server to Another Server”.

Benefits: 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.

Offloading Analysis: Server-side errors reported back to client side

Summary: 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.

Benefits: Starting R2020b, you can troubleshoot server-side errors more easily by using the log reported on the client side.

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

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

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

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

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

See also `polyspace-results-export`.

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

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

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

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

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

Or as a `-api-key` entry:

```
-api-key keyValue123
```

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

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

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

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

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

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

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

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

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

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

See also `polyspace-comments-import`.

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

AUTOSAR Support: Analysis more resilient to ARXML errors

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

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

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

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

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

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

For more information, see:

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

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

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

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

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

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

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

See also:

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

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

Changes in analysis options and binaries

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

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

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

Changes in run-time checks

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

Check	Change

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

Compatibility Considerations

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

Updated code metrics specifications

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

Code Metric	**Update**

`Number of Called Functions`	These metrics now accounts for function calls in a C++ constructor initializer list.
	
	For instance, in this code snippet, the number of called functions of `Derived::Derived()` is one. Previously, the number was computed as zero.
	
	`class  Base`
	`{`
	`  int b;`
	`  public:`
	`      Base() {`
	`          b = 0;`
	`      };`
	`};`
	`class Derived : public Base`
	`{`
	`  int d;`
	`  public:`
	`      Derived() : Base() {`
	`          d = 0;`
	`      };`
	`};`

Compatibility Considerations

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

R2020a

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

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

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

```
#include <limits.h>

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

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

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

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

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

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

You specify a compiler using the option `Compiler (-compiler)`.

```
polyspace-code_prover-server -compiler microchip -target pic -sources file.c ....
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

For more information, see:

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

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

Exporting Results: Export only results that must be reviewed to satisfy software quality objectives (SQOs)

Summary: 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 Code Prover Results
- Software Quality Objectives (Polyspace Code Prover Access)

Benefits: 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.

Jenkins Support: Use sample Jenkins Pipeline script to run Polyspace as part of continuous delivery pipeline

Summary: 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.

Benefits: 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.

Changes in analysis options and binaries

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

Warns

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

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

See also `-code-behavior-specifications`.

Changes in run-time checks

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

Check	**Change**
`Uncaught exception`	The check no longer flags the case where a function throws an exception whose data type is not in the list of exception types that the function is declared to throw. For instance, the function `foo` is declared to throw exceptions of type `int` and `std::exception`: `void foo2() throw(std::exception, int);` Code Prover used to check if the function can throw exceptions outside the specified types. The check is not performed from R2020a onwards. Dynamic exception specification is deprecated in C++11 and removed in the later standard C++17. See also Dynamic exception specification in the C++ standard.

Compatibility Considerations

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

Report Generation: Configure report generator to communicate with Polyspace Access over HTTPS

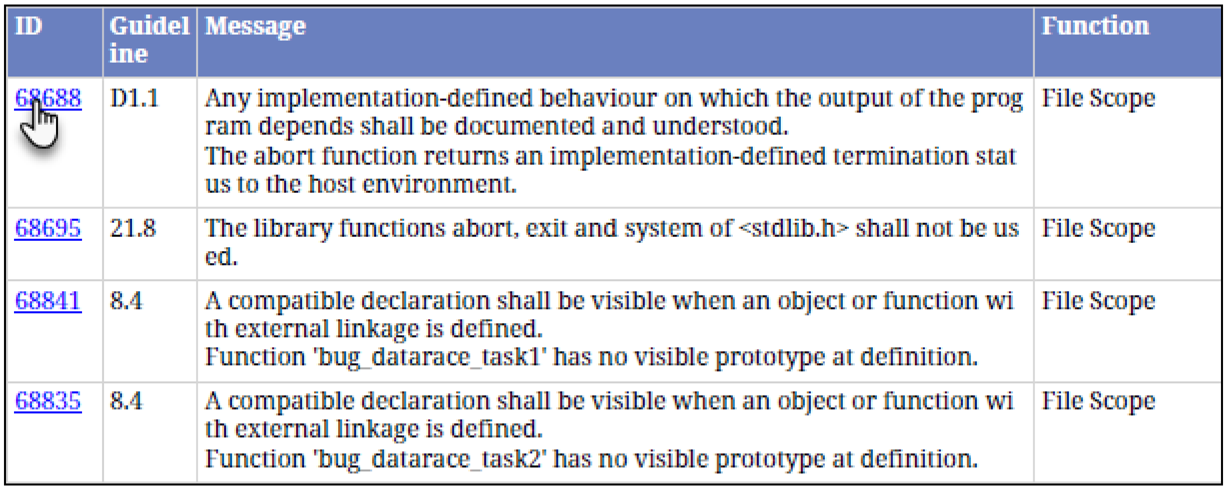
In R2020a, if you generate reports for results that are stored on Polyspace Access, you can configure the `polyspace-report-generator` binary to communicate with Polyspace Access over HTTPS.

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.

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.

R2019b

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

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

You specify this Code Prover mode using the option `-shared-variables-mode`:

```
polyspace-code-prover-server -shared-variables-mode -sources-list-file sourceList.txt ....
```

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

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

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

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

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

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

You specify a compiler using the option `Compiler (-compiler)`.

```
polyspace-code-prover-server -compiler cosmic -target s12z -sources-list-file sourceList.txt ....
```

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

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

Summary: In R2019b, if you create a Polyspace analysis configuration from your build system using the `polyspace-configure` command, 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.

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

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

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

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

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

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

Compatibility Considerations

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

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

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

See `MISRA C:2012 Dir 4.12`.

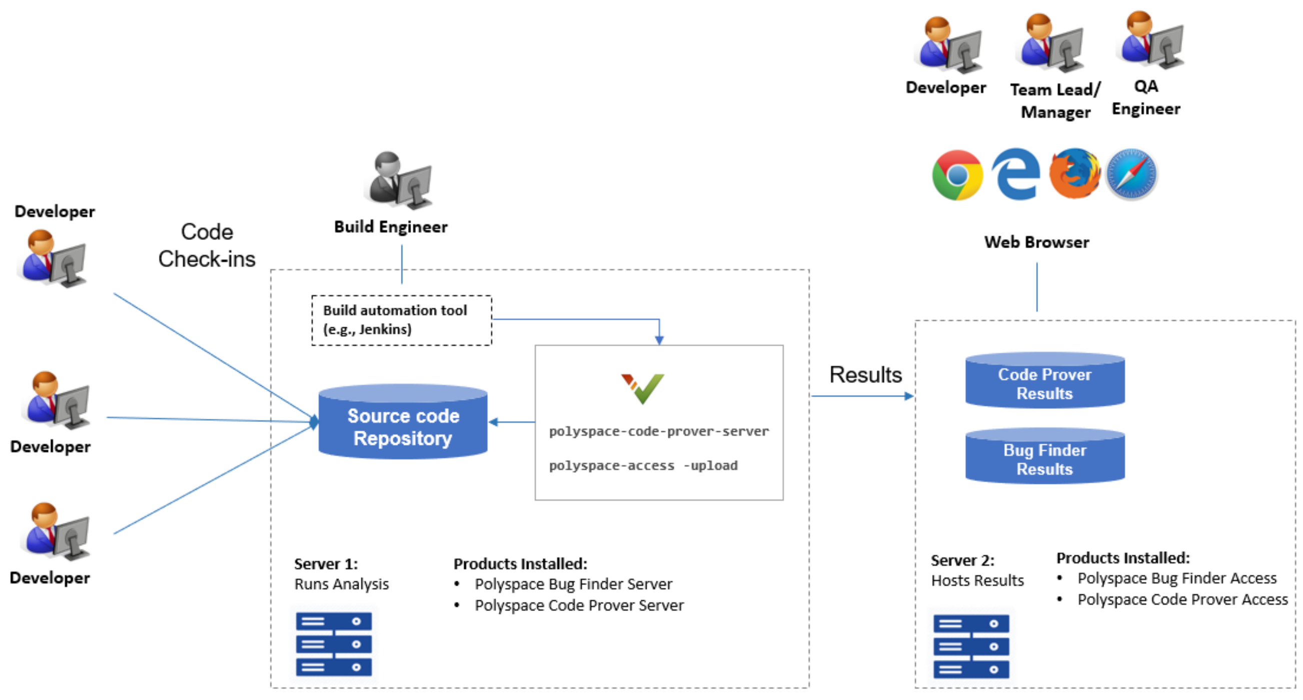
R2019a

Version: 10.0

New Features

Code Prover Analysis Engine Separated from Viewer: Run Code Prover analysis on server and view the results from multiple client machines

Summary: In R2019a, you can run Code Prover on a server with the new product, Polyspace Code Prover™ Server™. You can then host the analysis results on the same server or a second server with the product, Polyspace Code Prover 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 Code Prover Access license.



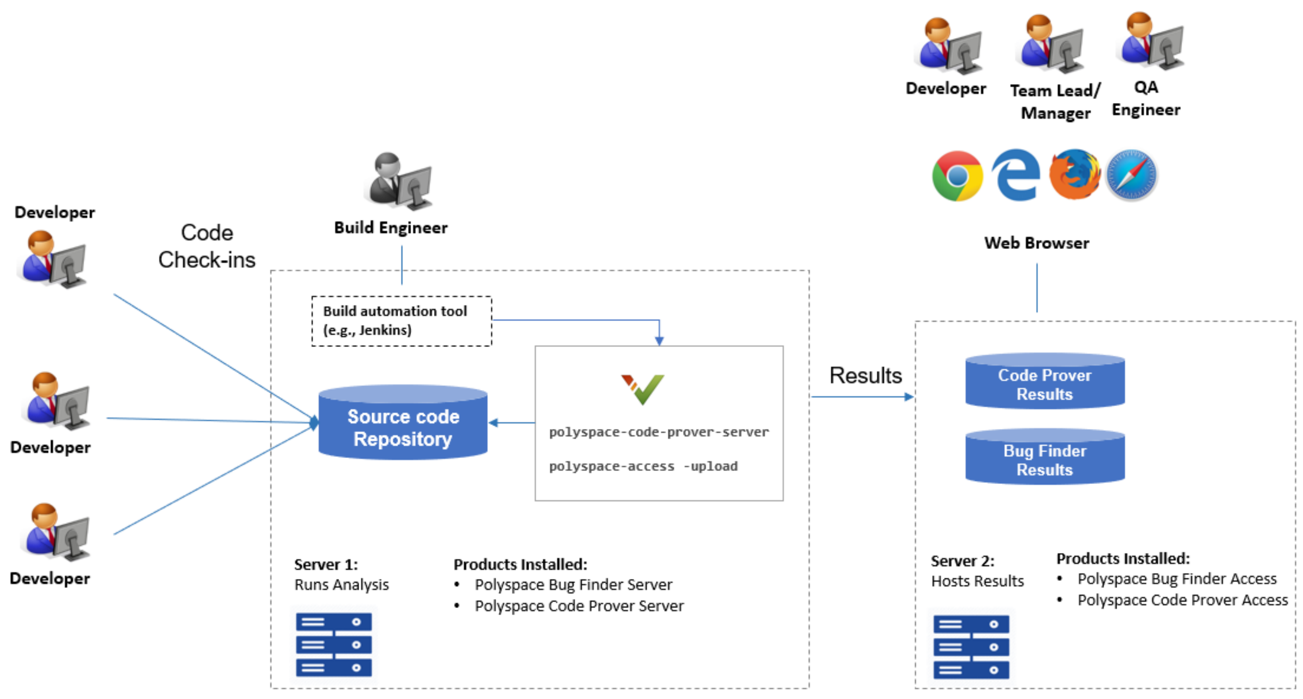
Note: Depending on the specifications, the same computer can serve as both Server 1 and Server 2.

Benefits: You can run the Code Prover analysis on a few powerful server class machines but view the analysis results from many terminals.

With the desktop product, Polyspace Code Prover, 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 (Polyspace Bug Finder Server).

Continuous Integration Support: Run Code Prover on server class computers with continuous upload to Polyspace Access web interface

Summary: In R2019a, you can check exhaustively for run-time errors 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 Code Prover Server product. The analysis results can be uploaded to the Polyspace Access web interface for review. Each reviewer with a Polyspace Code Prover 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 Code Prover on Server and Upload Results to Web Interface

Benefits:

- *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 Code Prover Server, a Code Prover analysis can be included in this build process. The analysis can run Code Prover checks on the new code submissions and report the results. The results can be reviewed in the Polyspace Access web interface with a Polyspace Code Prover 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 Code Prover Access license can review the Code Prover results for a project and assign issues to developers for fixing.
  - Each development team manager with a Polyspace Code Prover Access license can see an overview of Code Prover 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 Code Prover Access .

Continuous Integration Support: Set up testing criteria based on Code Prover static analysis results

Summary: In R2019a, you can run Code Prover on server class machines as part of unit and integration testing. You can define and set up testing criteria based on Code Prover static analysis results.

For instance, you can set up the criteria that new code submissions must have zero red checks (definite run-time errors) before integration with a code base. Any submission with red checks can cause a test failure and require code fixes.

See:

- `polyspace-code-prover-server` for how to run Code Prover on servers.
- `polyspace-access` for how to export Code Prover 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.

Benefits:

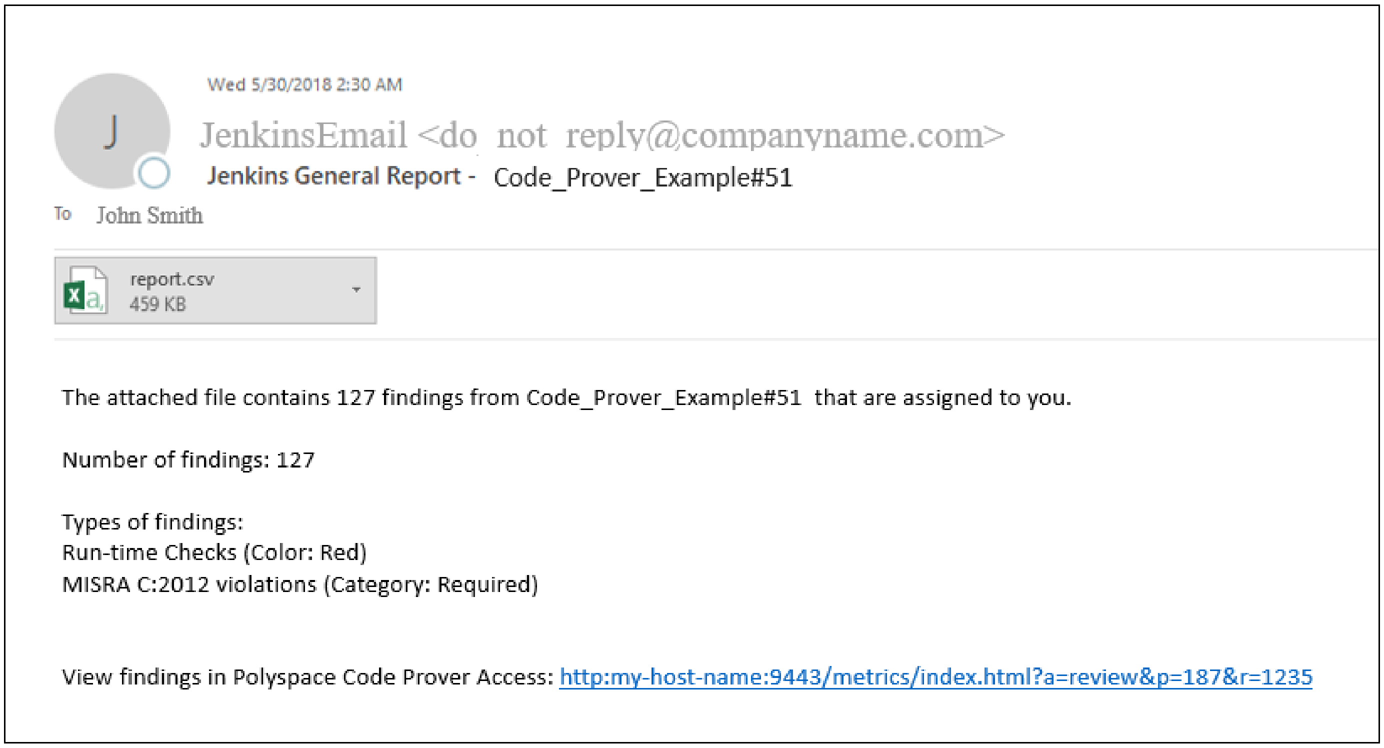
- *Automated testing*: After you define testing criteria based on Code Prover results, you can run the tests along with regular dynamic tests. The tests can run on a periodic schedule or based on predefined triggers.
- *Prequalification with Polyspace desktop products*: Prior to code submission, to avoid test failures, developers can check their submission with the same criteria as the server-side analysis. Using an installation of the desktop product, Polyspace Code Prover, 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 (Polyspace Bug Finder Server).

  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 (Polyspace Bug Finder Server).

Continuous Integration Support: Set up email notification with summary of Code Prover results after analysis

Summary: In R2019a, you can set up email notification for new Code Prover results. The email can contain:

- A summary of new results from the latest Code Prover 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.



Wed 5/30/2018 2:30 AM

JenkinsEmail <do not reply@companyname.com>

Jenkins General Report - Code_Prover_Example#51

To John Smith

report.csv
459 KB

The attached file contains 127 findings from Code_Prover_Example#51 that are assigned to you.

Number of findings: 127

Types of findings:
Run-time Checks (Color: Red)
MISRA C:2012 violations (Category: Required)

View findings in Polyspace Code Prover Access: http:my-host-name:9443/metrics/index.html?a=review&p=187&r=1235

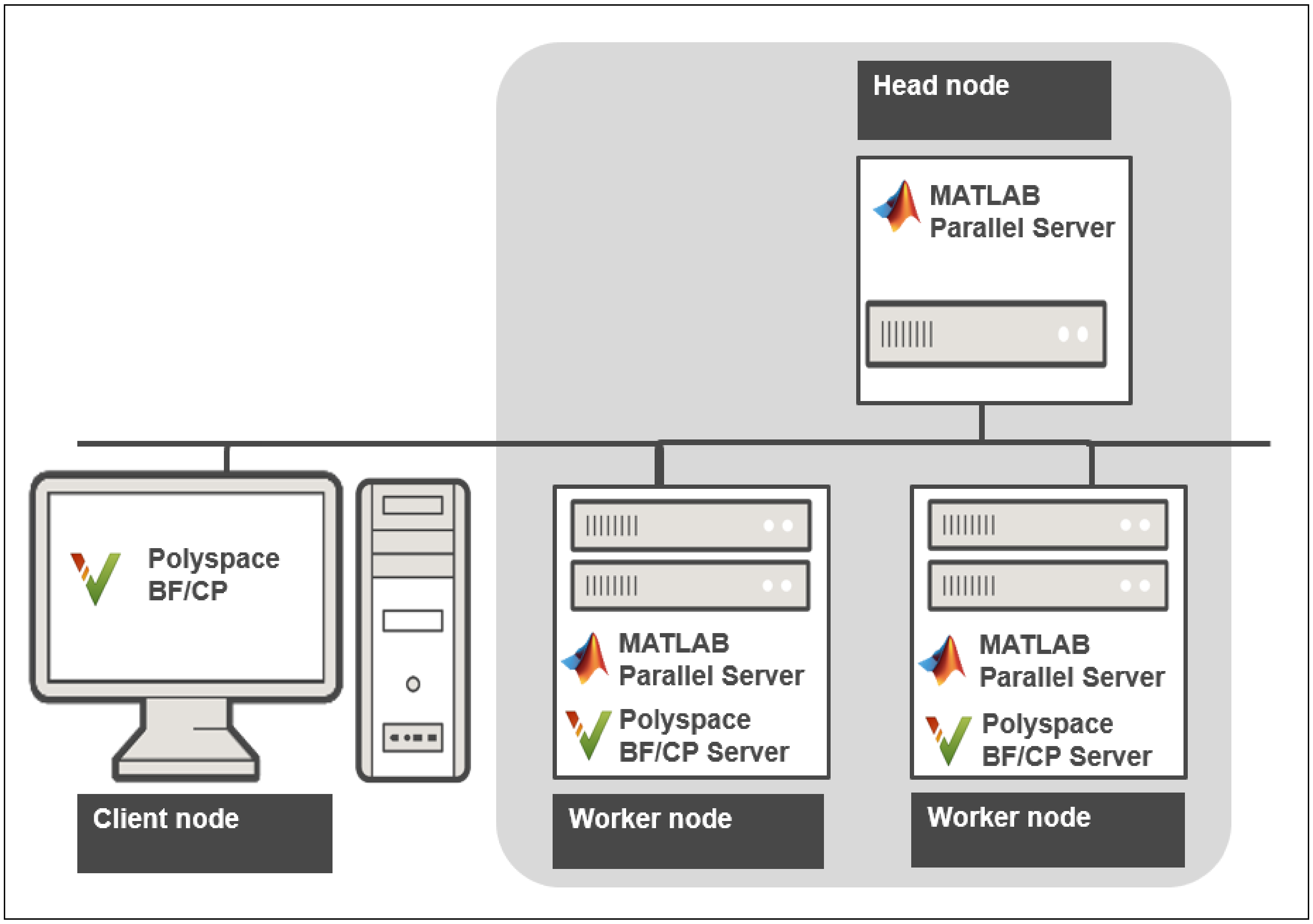
See Send E-mail Notifications with Polyspace Code Prover Results.

Benefits:

- *Automated notification*: Developers can get notified in their e-mail inbox about results from the last Code Prover run on their submissions.
- *Preview of Code Prover results*: Developers can see a preview of the new Code Prover 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 Code Prover Access license can use the links in the e-mail attachments to see further details of a result in the Polyspace Access web interface.

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

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



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

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