Polyspace® Bug Finder™ Release Notes

MATLAB® & SIMULINK®

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.
3 Apple Hill Drive
Natick, MA 01760-2098

Polyspace® Bug Finder ™ Release Notes

Trademarks

Patents

Contents

R2018a

R2017b

R2017a

R2016b

R2016a

R2015aSP1

R2015b

R2015a

R2014b

R2014a

R2013b

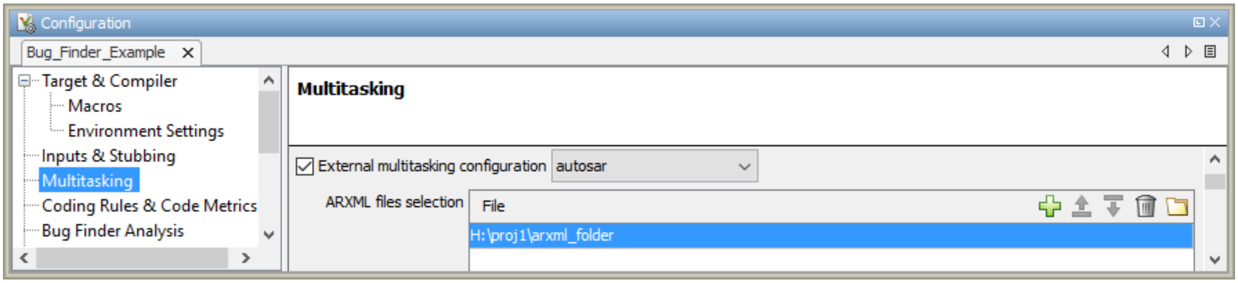
1

R2018a

Analysis Setup

AUTOSAR Support: Set up Polyspace multitasking configuration automatically from an AUTOSAR description

Summary: In R2018a, Polyspace can parse your AUTOSAR specifications (`.arxml` files) to determine your multitasking configuration.



☑ External multitasking configuration: autosar

ARXML files selection

File
H:\proj1\arxml_folder

This feature supports AUTOSAR XML schema for releases 4.0 and later.

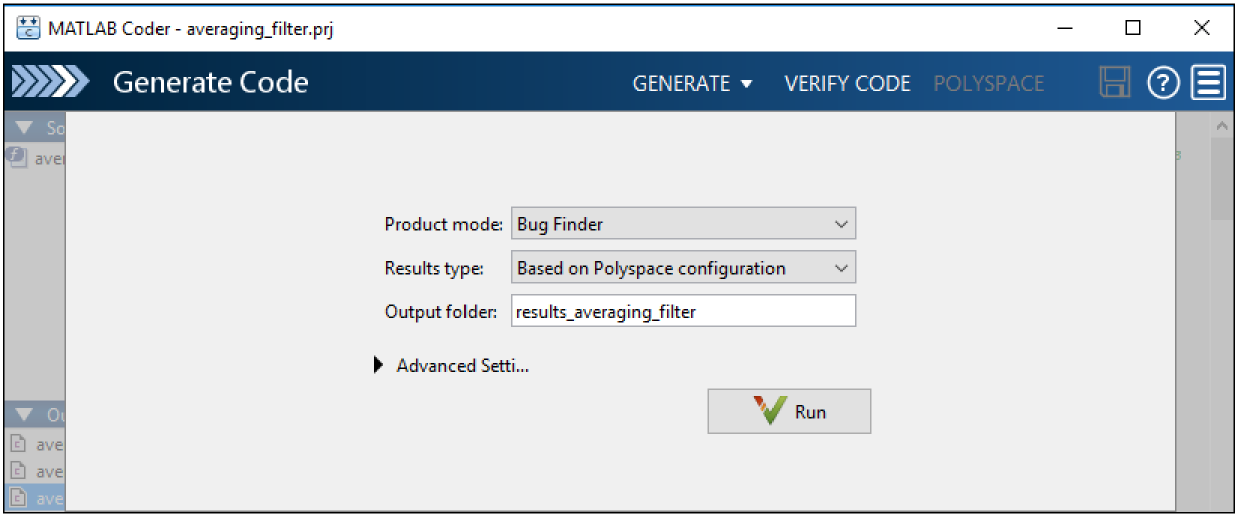
For more information, see `ARXML files selection (-autosar-multitasking)`.

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

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



Product mode: Bug Finder

Results type: Based on Polyspace configuration

Output folder: results_averaging_filter

For details, see:

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

Benefits:

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

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

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

- Texas Instruments™

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

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

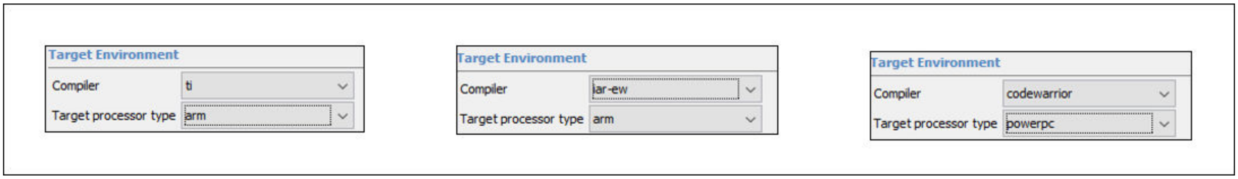
  You can specify these target processors: `arm`, `avr`, `msp430`, `rh850` and `rl78`.

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

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

  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.



Target Environment
Compiler: ti
Target processor type: arm

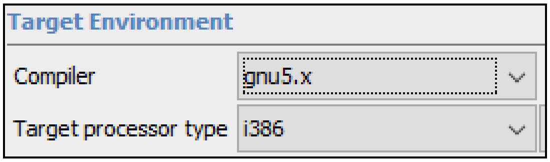
Target Environment
Compiler: iar-ew
Target processor type: arm

Target Environment
Compiler: codewarrior
Target processor type: powerpc

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

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

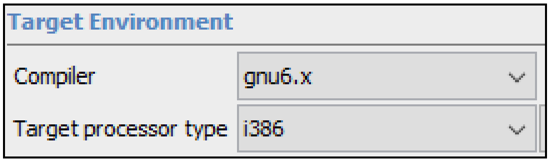
Summary: In R2018a, if you build your source code using these versions of GCC or Clang compilers, you can specify the following compiler option values to setup your Polyspace analysis:

- 

  Target Environment

  Compiler gnu5.x

  Target processor type i386

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

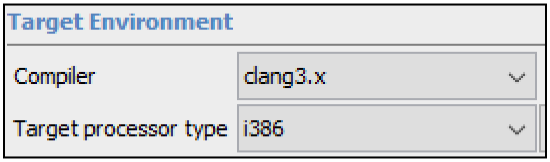
  Target Environment

  Compiler gnu6.x

  Target processor type 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.

- 

  Target Environment

  Compiler clang3.x

  Target processor type 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

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

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

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

```
polyspace-configure -no-project make -B \
  -build-trace trace.txt -cache-path /tmp/cache
```

2. Create a Polyspace project using the build trace and cache. Include or exclude files as needed using shell GLOB patterns.

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

   The preceding example includes sources in folder paths containing `src` and excludes `.c` files ending with `_test`.
3. Delete the build trace and cache.

For more information, see `polyspace-configure`.

Benefits:

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

Support for IBM Rational Rhapsody to be removed

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

Compatibility Considerations

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

Changes in analysis options and binaries

In R2018a, the following options have been added, changed, or removed.

New Options

Option	Description
`ARXML files selection (-autosar-multitasking)`	See AUTOSAR Support release note.

Updated Options

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

Option	Change
	See Updated GCC and Clang Compiler Support release note. • Option value `clang3.5` is deprecated. Use `clang3.x` instead.

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.

Changes in MATLAB option object properties

In R2018a, the following MATLAB object option properties have been added, changed, or removed.

In the tables, `opts = polyspace.Project`.

New Properties

Property	Description

`opts.Configuration.Multitasking...`
`.EnableExternalMultitasking`
 	
`opts.Configuration.Multitasking...`
`.ExternalMultitaskingType`
 	
`opts.Configuration.Multitasking...`
`.ArxmlMultitasking`
	For more information, see Properties.

Updated Properties

Property	Description

`opts.Configuration...`
`.TargetCompiler.Compiler`
	• New value `ti` added. See Compiler Support release note.
	• New value `iar-ew` added. See Compiler Support release note.
	Use this value to emulate IAR compilers.
	For older Polyspace projects, you can still use property value `iar`.
	• New value `codewarrior` added. See Compiler Support release note.
	• New value `gnu5.x` added. See Updated GCC and Clang Compiler Support release note.
	• New value `gnu6.x` added. See Updated GCC and Clang Compiler Support release note.
	• New value `clang3.x` added. See Updated GCC and Clang Compiler Support release note.
	• Property value `clang3.5` is removed. Use `clang3.x` instead.

Removed Properties

Property	Use Instead

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

Compatibility Considerations

Substitute instances of the removed or updated properties in your MATLAB code with the appropriate replacement.

If you use a removed property or property value, you get an error message.

Analysis Results

CERT C Support: Check for information leakage, invalid environment pointers, and other rules from the CERT C Coding Standard

Summary: 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 errno	`Misuse of errno 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 sizeof, _Alignof, or _Generic	`Side effect of expression ignored`

EXP47-C	Do not call va_arg with argument of the incorrect type	`Incorrect data type passed to va_arg`
		`Too many va_arg calls for current argument list`

FIO41-C	Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects	`Stream argument with possibly unintended side effects`

CERT C Rule	Description	Polyspace Checker
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

Summary: 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.

Defect	Issue Detected
`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

Summary: 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.

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

See also “MISRA C++ Coding Rules”.

MISRA C:2012 Directive 4.8: Detect opportunities for data hiding

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

See `MISRA C:2012 Directive 4.8`.

Benefits: 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

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

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

Improved Fast Analysis: Find some multi-file MISRA C violations in fast analysis

Summary: 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`.

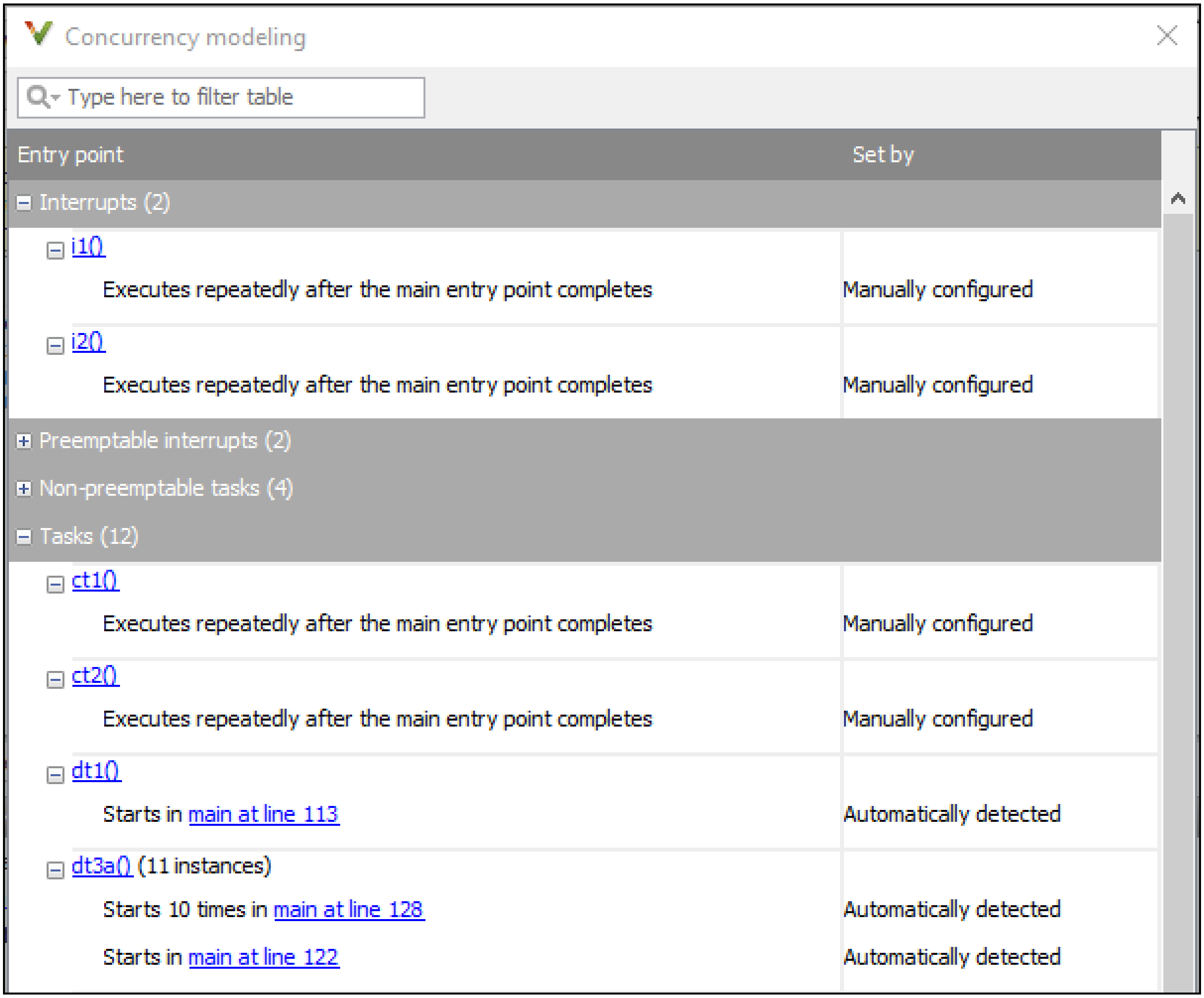
Benefits: You detect more violations in the fast analysis mode. Previously, fast analysis looked only for defects and coding rule violations that involved single files or functions.

Reviewing Results

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

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

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



Concurrency modeling

Type here to filter table

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	Automatically detected
Starts in main at line 122	Automatically detected

Benefits:

- *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)

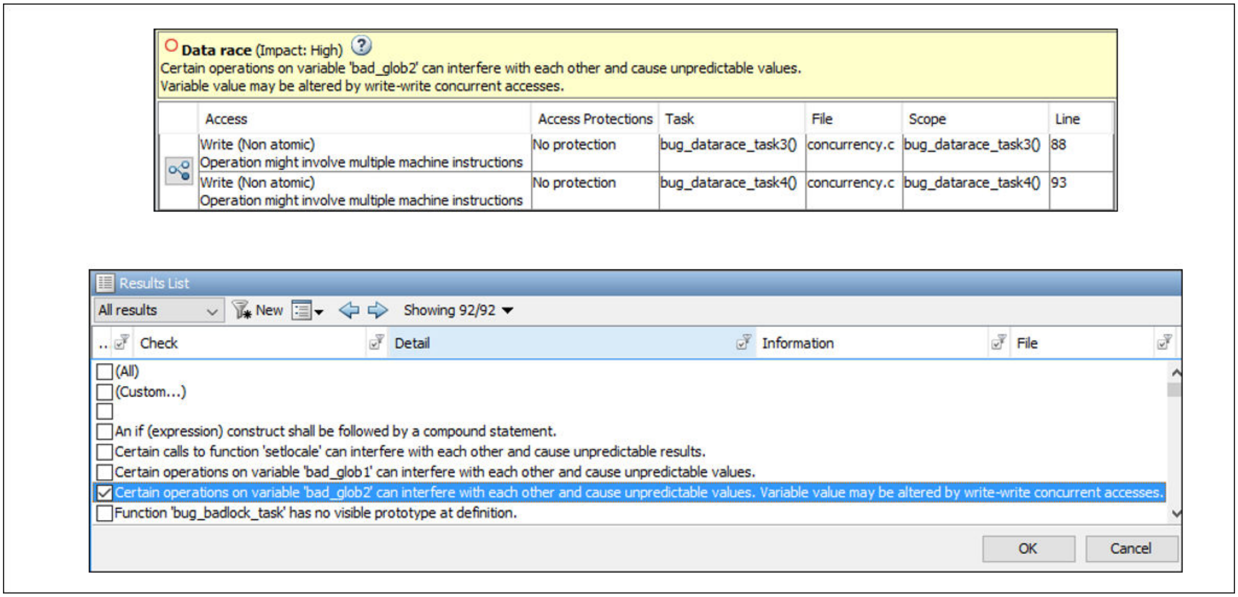
tasks. To understand why a data race does not occur between two entry points (Bug Finder), you can check if one of the entry points has lower priority than the other. See `Data race`.

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

Summary: 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

□ (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.

See also `Data race`.

Benefits: 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.

2

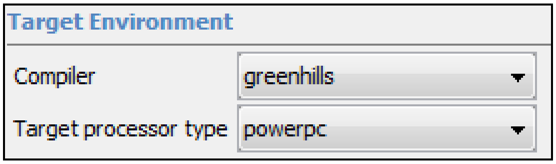
R2017b

Analysis Setup

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

Summary: 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

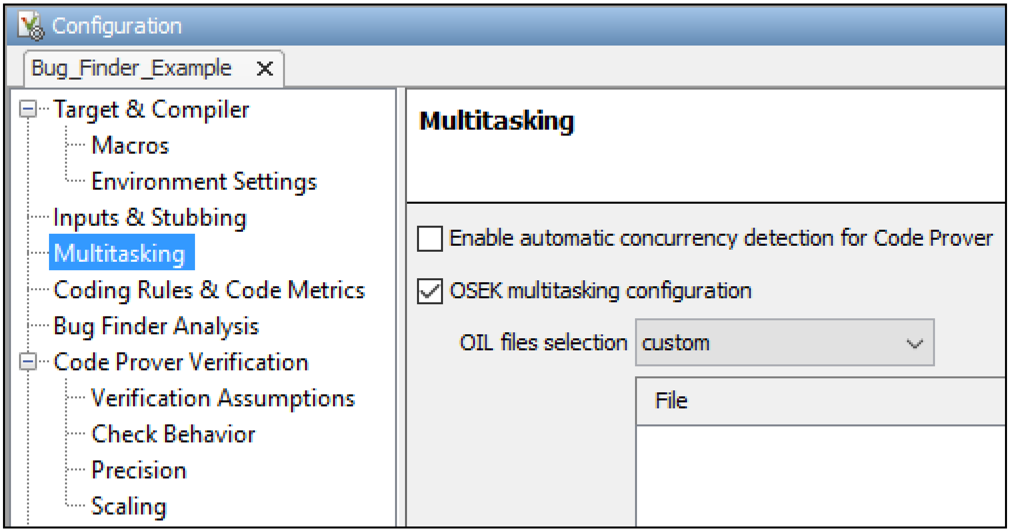
Compiler: greenhills

Target processor type: powerpc

Benefits: 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

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



Multitasking

□ Enable automatic concurrency detection for Code Prover

☑ OSEK multitasking configuration

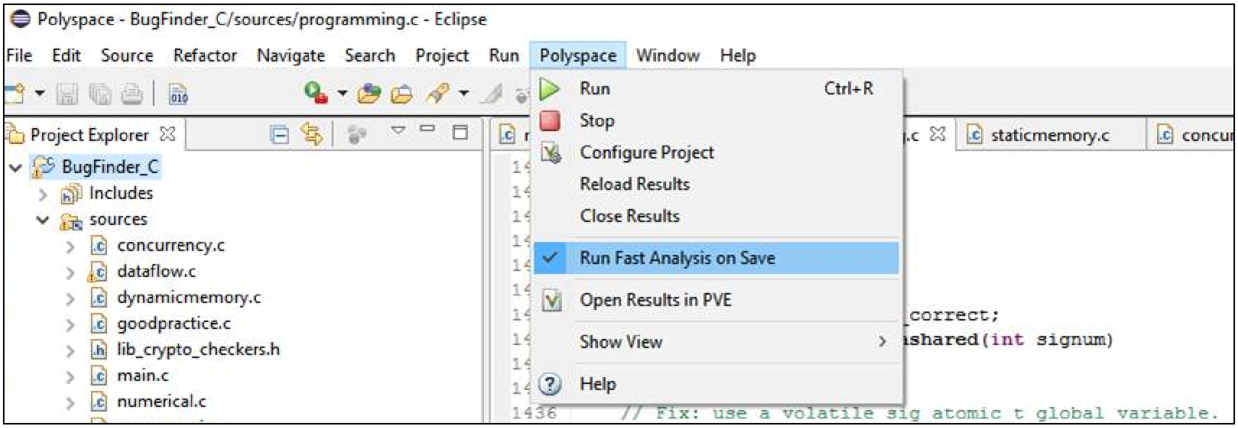
OIL files selection: custom

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

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

Incremental Analysis in Eclipse: Detect bugs as you type and save code in your Eclipse IDE

Summary: In R2017b, if you install the Polyspace plugin in your Eclipse™ IDE, the analysis runs each time you save your code.



Benefits: 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

Summary: 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`.

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

Additional Considerations

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

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

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

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

Compatibility Considerations

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

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

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

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

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

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

Changes in analysis options and binaries

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

New Options

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

Updated Options

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

Option	Change
Enum type definition (`-enum-type-definition`)	Option value `defined-by-standard` changed to `defined-by-compiler`.
Invalid use of floating point operation	You can detect a comparison to `0.0` when you add the option `-detect-bad-float-op-on-zero`. The defect is renamed in the user interface to : `Floating point comparison with equality operators`. The command-line parameter is still `BAD_FLOAT_OP`.
`-asm-begin` and `-asm-end`	Available for all compilers.

Removed Options

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

Compatibility Considerations

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

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

Summary: 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 `volatile sig_atomic_t` nor a lock-free atomic object.
`Signal call from within signal handler`	You call `signal()` from within an interruptible signal handler.
`Return from computational exception signal handler`	Your signal handler returns normally after a computational exception signal `SIGFPE`, `SIGILL`, or `SIGSEGV`.
`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 `fsetpos()` with a file position that was not returned from `fgetpos()`.
`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 `fgets()` or `fwgets()` 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 transfering data`	You transfer data without matching the endianness of the host and network.
`Unsafe call to a system function`	You call `system()`, `popen()`, `_popen()`, or `_wopen()`.
`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

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

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

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.

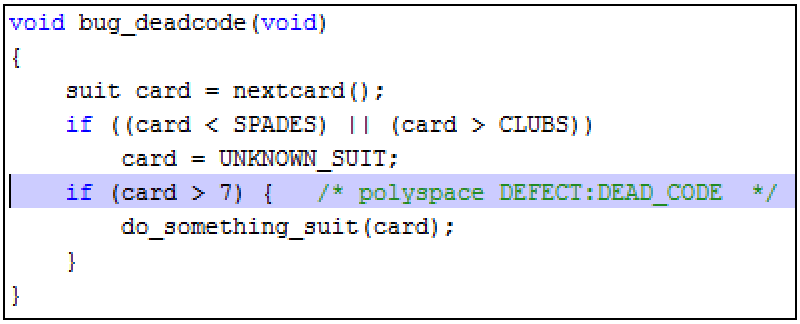
Rule	Description	Improvement
MISRA C: 2004 Rule 17.4 and MISRAC++ Rule 5-0-15	Array indexing shall be the only allowed form of pointer arithmetic.	The rule checker flags array indexing on nonarray pointers. Previously, the checker flagged only explicit pointer arithmetic on pointers.
MISRA C: 2012 Rule 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 `Subtraction or comparison between pointers to different arrays`.

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

Reviewing Results

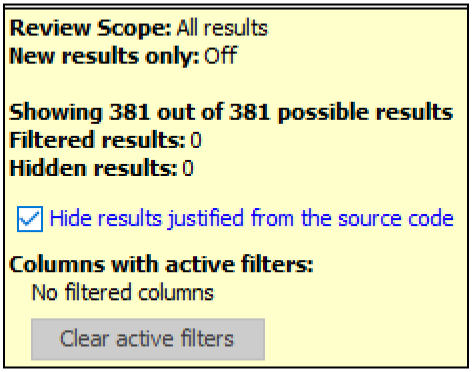
Result Review Workflow: Hide results that you reviewed once and justified through source code annotations

Summary: In R2017b, if you justify a result through source code annotations, subsequent analyses do not redisplay 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.



Review Scope: All results
New results only: Off

Showing 381 out of 381 possible results
Filtered results: 0
Hidden results: 0

☑ Hide results justified from the source code

Columns with active filters:
No filtered columns

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

Suppose the analysis flags an error-handling statement as 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

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

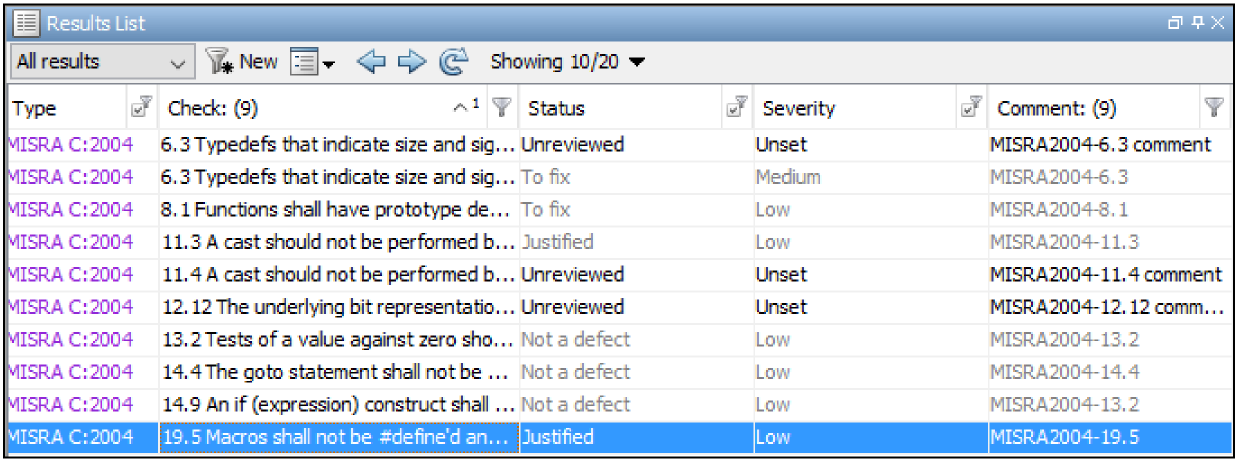
Benefits:

- *Easier results review:* With the new annotation format, you can provide a justification for multiple types of results on the same line. Previously, you had to enter the justification for different types of results, such as defects and coding rules violations, on different lines.
- *Custom annotation format:* You can use an XML file to define any annotation format and map it to the Polyspace syntax. When you analyze your code, Polyspace can interpret the annotations regardless of the format.

Polyspace still supports annotations that use the old syntax.

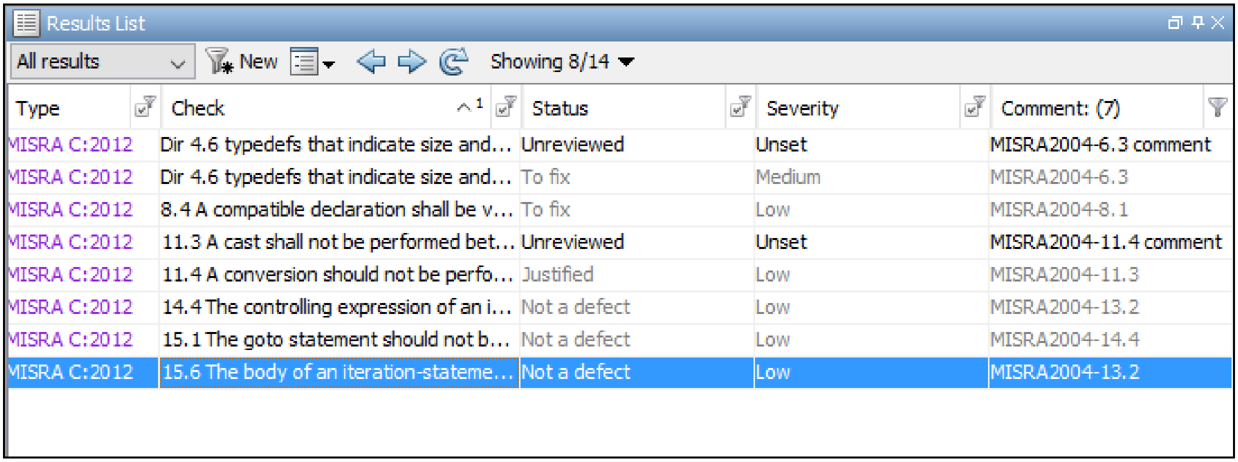
MISRA Comments and Code Annotations: Import your existing MISRA C:2004 justifications to MISRA C:2012 results

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



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

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



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

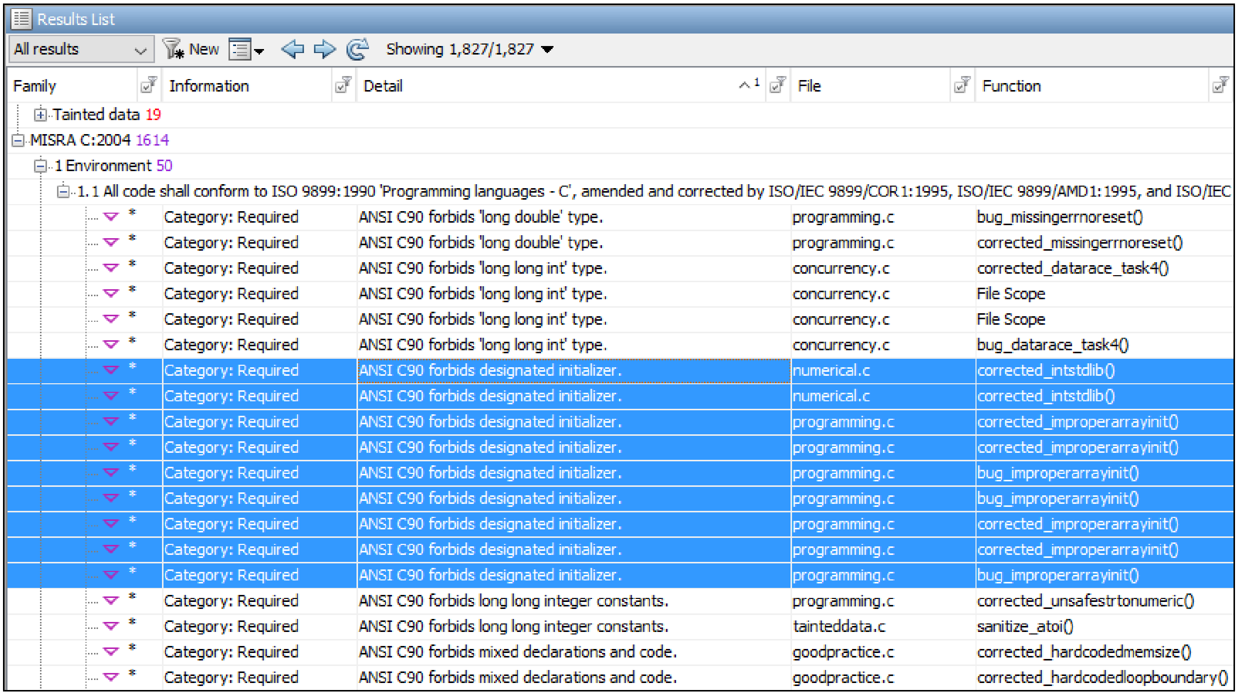
For more information, see Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results.

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

Results Review Workflow: Sort and filter results by subtype

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

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

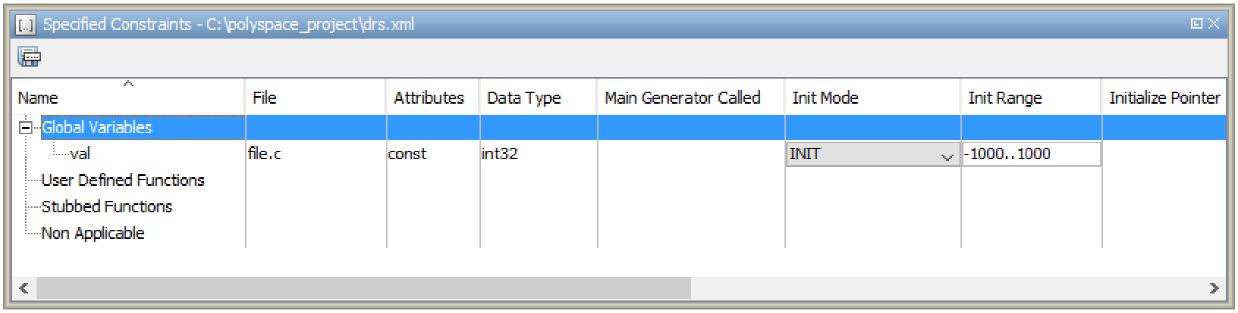


Family	Information	Detail	File	Function
⊞ Tainted data 19				
⊟ MISRA C:2004 1614				
⊟ 1 Environment 50				
⊟ 1.1 All code shall conform to ISO 9899:1990 'Programming languages - C', amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC				
▽ *	Category: Required	ANSI 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()

Benefits: 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

Summary: 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.



Specified Constraints - C:\polyspace_project\drs.xml

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**.

Benefits: 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

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

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

Benefits: You can follow your review progress more easily.

Additional Considerations

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

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

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

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

Compatibility Considerations

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

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

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

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

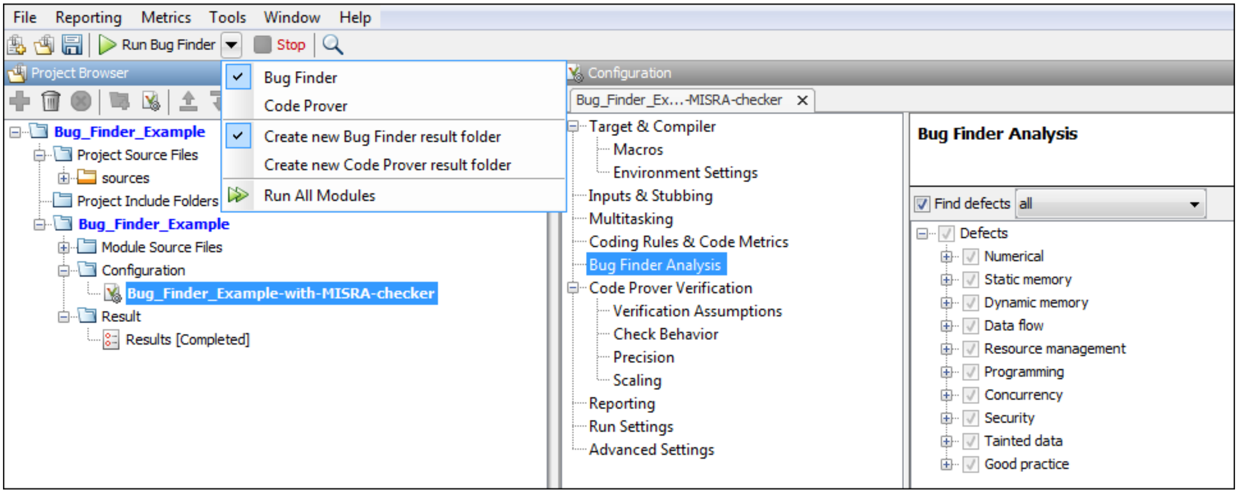
3

R2017a

Analysis Setup

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

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



☑ Find defects: all

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.

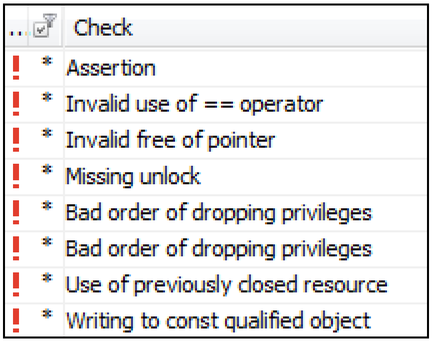
  Bug Finder User:

  - 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*.

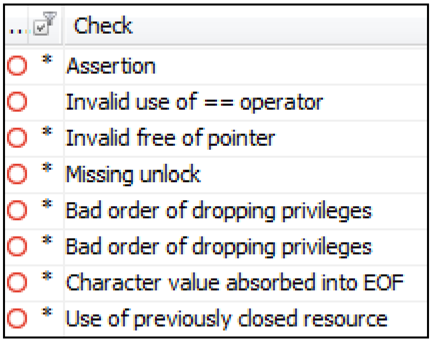
- A new icon is used to denote defects.

  Before R2017a:



..		Check
!	*	Assertion
!	*	Invalid use of == operator
!	*	Invalid free of pointer
!	*	Missing unlock
!	*	Bad order of dropping privileges
!	*	Bad order of dropping privileges
!	*	Use of previously closed resource
!	*	Writing to const qualified object

  R2017a:



..		Check
○	*	Assertion
○		Invalid use of == operator
○	*	Invalid free of pointer
○	*	Missing unlock
○	*	Bad order of dropping privileges
○	*	Bad order of dropping privileges
○	*	Character value absorbed into EOF
○	*	Use of previously closed resource

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.

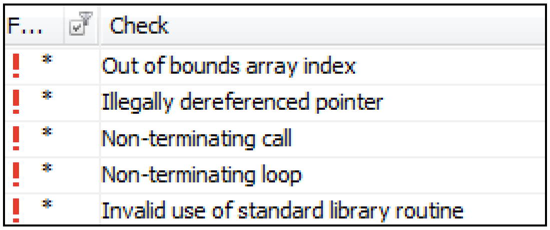
  You can change this default behavior and create a new result folder for the second analysis. Use the option **Create new Code Prover result folder** from the **Run** button dropdown.
- If some of your files do not compile, the analysis continues with the remaining files. If a file with compilation errors 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`).

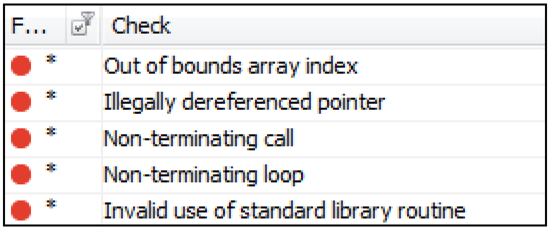
- A new icon is used to denote definite run-time errors or red checks.

  Before R2017a:



F…	☑	Check
!	*	Out of bounds array index
!	*	Illegally dereferenced pointer
!	*	Non-terminating call
!	*	Non-terminating loop
!	*	Invalid use of standard library routine

  R2017a:



F…	☑	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
```

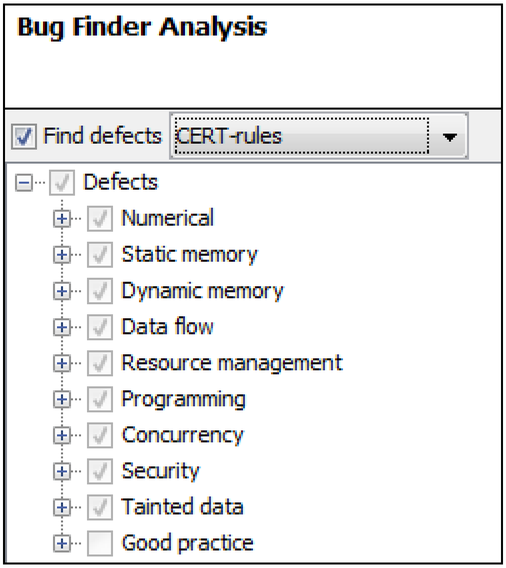
To create a Windows batch file that runs a Bug Finder analysis, run the command:

```
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

Summary: 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.

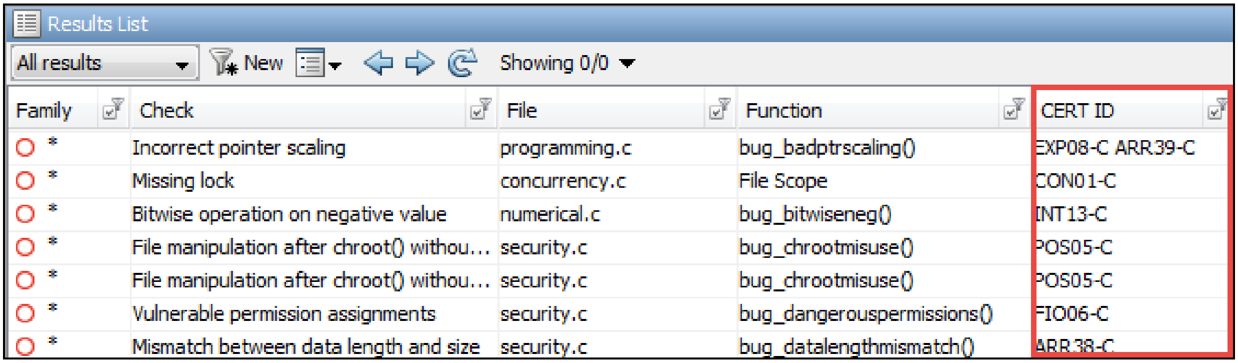


Bug Finder Analysis

☑ Find defects: CERT-rules

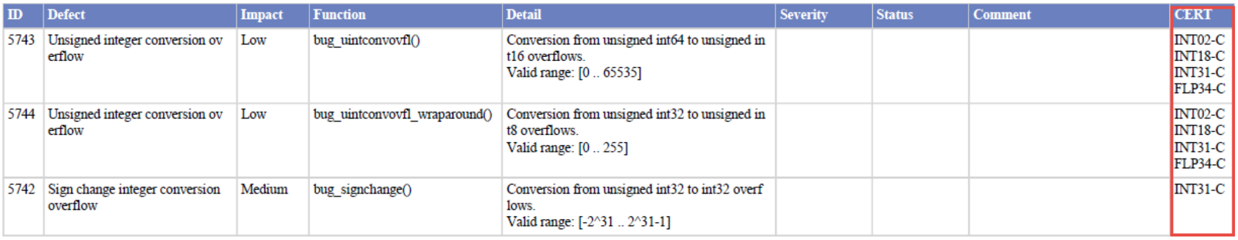
☑ Defects
- ☑ Numerical
- ☑ Static memory
- ☑ Dynamic memory
- ☑ Data flow
- ☑ Resource management
- ☑ Programming
- ☑ Concurrency
- ☑ Security
- ☑ Tainted data
- □ Good practice

Results: After analysis, you see the security standard ID-s corresponding to each result.



Family		Check	File	Function	CERT ID
○	*	Incorrect pointer scaling	programming.c	bug_badptrscaling()	EXP08-C ARR39-C
○	*	Missing lock	concurrency.c	File Scope	CON01-C
○	*	Bitwise operation on negative value	numerical.c	bug_bitwiseneg()	INT13-C
○	*	File manipulation after chroot() withou...	security.c	bug_chrootmisuse()	POS05-C
○	*	File manipulation after chroot() withou...	security.c	bug_chrootmisuse()	POS05-C
○	*	Vulnerable permission assignments	security.c	bug_dangerouspermissions()	FIO06-C
○	*	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.



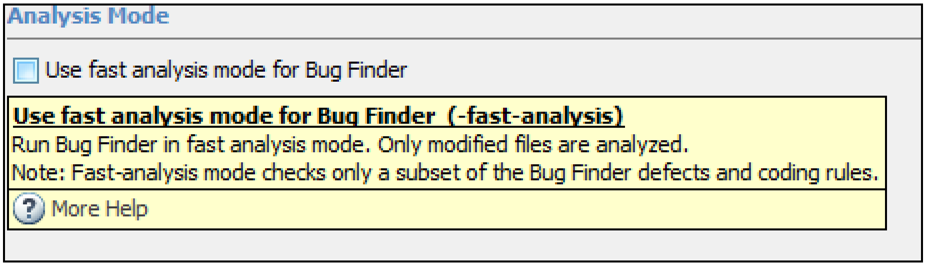
ID	Defect	Impact	Function	Detail	Severity	Status	Comment	CERT
5743	Unsigned integer conversion overflow	Low	bug_uintconvovfl()	Conversion from unsigned int64 to unsigned int16 overflows. Valid range: [0 .. 65535]				INT02-C INT18-C INT31-C FLP34-C
5744	Unsigned integer conversion overflow	Low	bug_uintconvovfl_wraparound()	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

Benefits: 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

Summary: 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.



Analysis Mode

□ Use fast analysis mode for Bug Finder

Use fast analysis mode for Bug Finder (-fast-analysis)
Run Bug Finder in fast analysis mode. Only modified files are analyzed.
Note: Fast-analysis mode checks only a subset of the Bug Finder defects and coding rules.

Benefits: 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?*

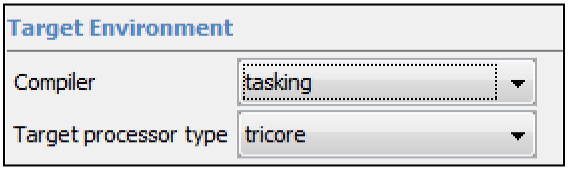
  The defect checker is internally disabled. When you switch back to full analysis, the defect checker is enabled again. For information on:

  - 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

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

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



Target Environment

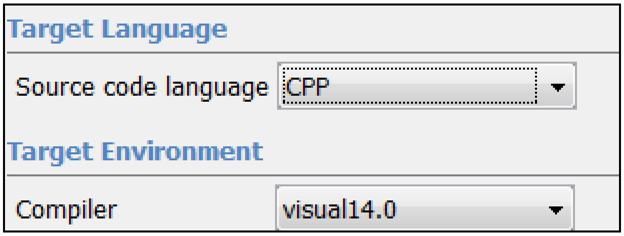
Compiler: tasking

Target processor type: tricore

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

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

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



Target Language

Source code language: CPP

Target Environment

Compiler: visual14.0

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

Benefits:

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

Autodetection of Concurrency Primitives: Multitasking model detected from Windows, µC/OS II or C++11 multithreading functions

Summary: 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`

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

Autodetection of Concurrency Primitives: Map Unsupported Thread Creation Functions to Supported Functions

Summary: 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);
}
```

Benefits: 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 reenable all interrupts

Summary: In R2017a, when specifying your multitasking model for analysis, you can provide a routine that disables all interrupts.

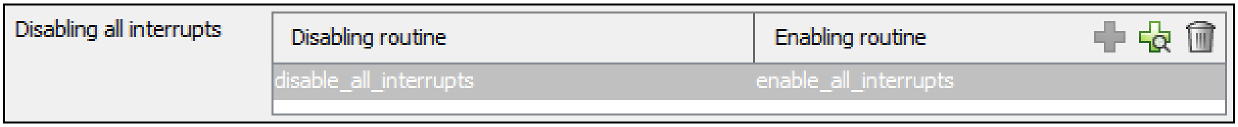
For instance, in the following code, the function `disable_all_interrupts` disables all interrupts until the function `enable_all_interrupts` is called. Even if `task`, `isr1` and `isr2` run concurrently, the operations `x=0` or `x=1` cannot interrupt the operation `x++`.

```
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
disable_all_interrupts	enable_all_interrupts

Benefits: 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();
  }
  ```

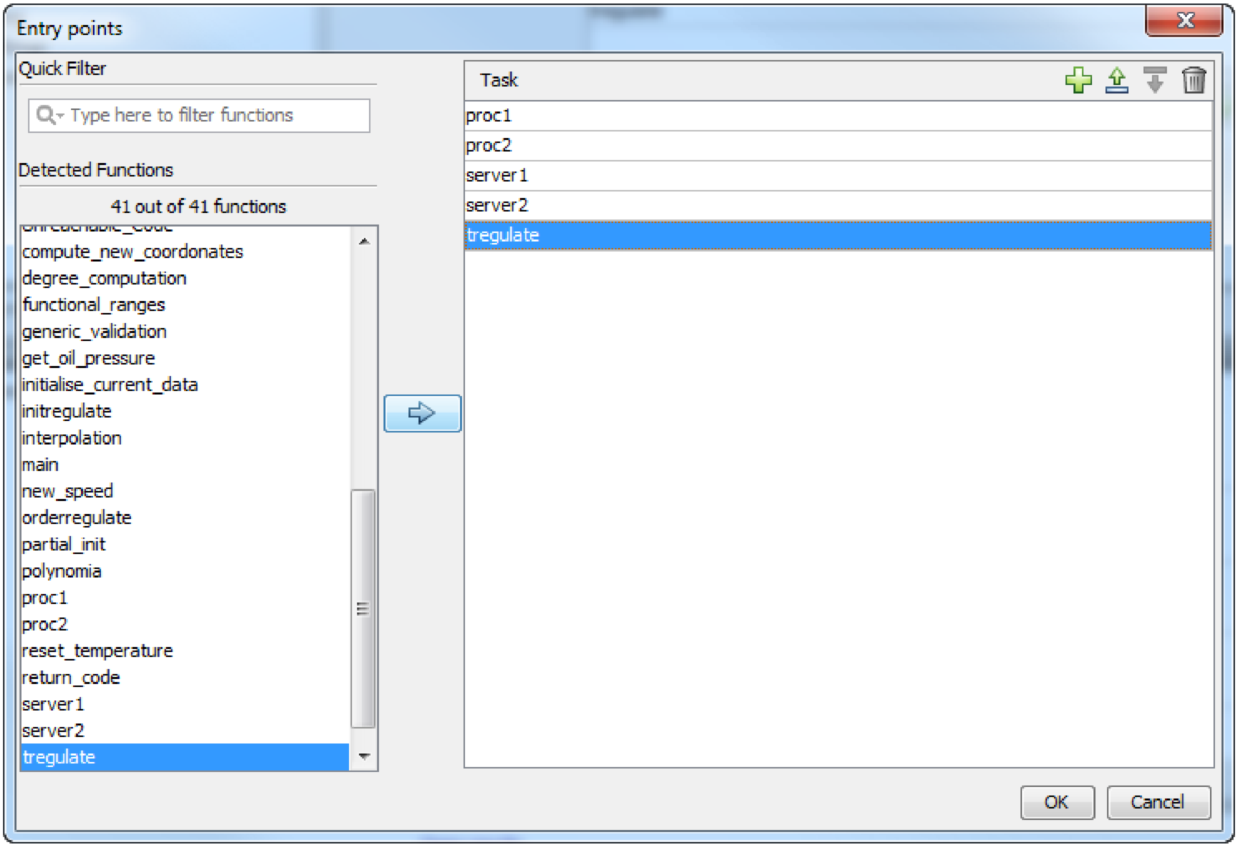
  Here, the operation `x++` is protected from the operation `x=0` in `isr1`, but not from the operation `x=1` in `isr2`. If the function `begin_critical_section` disabled *all interrupts*, calling it before `x++` would have been sufficient to protect it.

  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

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

For instance, to specify which functions act as entry points to your multitasking application, you can choose the names from a list as follows:



unreachable_code
compute_new_coordonates
degree_computation
functional_ranges
generic_validation
get_oil_pressure
initialise_current_data
initregulate
interpolation
main
new_speed
orderregulate
partial_init
polynomia
proc1
proc2
reset_temperature
return_code
server1
server2
tregulate

proc1
proc2
server1
server2
tregulate

Benefits: You do not have to enter the names manually. If the functions list is long, you can start typing the function name to reduce the list.

Polyspace API in MATLAB: Create MATLAB objects from Polyspace projects to run analysis

Summary: In R2017a, you can create a MATLAB object from a Polyspace project (`.psrpj` file). For instance, if you have a file `myProject.psprj` in the current working folder, enter:

```
opts = polyspace.loadProject('myProject.psprj')
```

Use the object `opts` in MATLAB scripts to run a Polyspace analysis:

```
polyspaceBugFinder(opts);
```

Benefits:

You can now consider the following workflows:

- *Set options in GUI and script analysis*: Use the Polyspace user interface to specify options in your Polyspace project, or adjust options based on results from a trial run. After the options are stable, create a MATLAB object `opts` from the project and store it in a MAT-file. As you move along in your development cycle, simply load `opts` from your MAT-file, update `opts.Sources` to add new source files, update other properties 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

Summary: In R2017a, Polyspace Bug Finder analysis supports 128-bit variables.

Benefits: 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

Summary: In R2017a, by default, automatic project creation will throw an error if a project with the same name exists in the output folder.

If you encounter an error, avoid the name conflict: change the project name, output folder, or remove your older project.

Benefits: You cannot overwrite existing projects by accident. If you use scripts that are intended to overwrite existing projects, use the additional option `-allow-overwrite`.

Changes in analysis options and binaries

In R2017a, the following options have been added, changed, or removed.

New Options

Option	Description
Use fast analysis mode for Bug Finder (`-fast-analysis`)	Run analysis using faster local mode of Bug Finder. See Incremental Analysis of Select Checks on page 3-7.
Disabling all interrupts (`-routine-disable-interrupts -routine-enable-interrupts`)	Specify routines that disable and reenable interrupts. See Manual Multitasking Setup on page 3-12.

Updated Options

Option	Change	More Information
Report template	Renamed in user interface	New name: **Bug Finder report** The command-line name is still `-report-template`.
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 `-batch`.
Add to results repository	Renamed in user interface	New name: **Upload results to Polyspace Metrics** The option is now in the **Run Settings** node in your project configuration. The command-line name is still `-add-to-results-repository`.
Compiler (`-compiler`)	New values added	You can specify the following arguments: • `tasking` See TASKING Compiler Support on page 3-8. • `visual14.0` See Microsoft Visual C++ Support on page 3-9.

Option	Change	More Information
Find defects (`-checkers`)	New value added	You can specify the following arguments: • `CWE` • `CERT-rules` • `CERT-all` • `ISO-17961` See Security Standards Checking on page 3-6.
Check MISRA C:2012 (`-misra3`)	New value added	You can specify the following arguments: • `CERT-rules` • `CERT-all` • `ISO-17961` See Security Standards Checking on page 3-6.

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, `polyspace-comments-import`.
`polyspace-automatic-verification`	Error	Binary will be removed in a future release.
`polyspace-remote`	Error	Binary will be removed in a future release.
`polyspace-verifier`	Error	Binary will be removed in a future release.
`rte-kernel`	Error	Binary will be removed in a future release.
Dialect (`-dialect`)	Error	Option will be removed in a future release. Use Compiler (`-compiler`) instead.

Option	Status	Description
Target operating system (`-OS-target`)	Error	Option will be removed in a future release. If you use this option in scripts, see the list below for replacements: • `Linux`: If you get compilation errors, use Compiler (`-compiler`) `gnu`*`x.x`*. Sometimes, you might also have to set Preprocessor definitions (`-D`) to `linux`, `unix`, or `__linux__`. • `Visual`: Use Compiler (`-compiler`) `visual`*`x.x`* • `Vxworks`: Use the VxWorks® configured template. For more information, see Create Project Using Configuration Template. • `Solaris`: Remove `-OS-target`. • `no-predefined-OS`: Remove `-OS-target`.
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 options object

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
`optsOld.Reporting.EnableReportGeneration`	`optsNew.MergedReporting.EnableReportGeneration`
`optsOld.Reporting.ReportTemplate`	`optsNew.MergedReporting.BugFinderReportTemplate`
`optsOld.Reporting.ReportOutputFormat`	`optsNew.MergedReporting.ReportOutputFormat`

ComputingSettings

Removed	Use instead
`optsOld.ComputingSettings.Batch`	`optsNew.MergedComputingSettings.BatchBugFinder`
`optsOld.ComputingSettings.AddToResultsRepository`	`optsNew.MergedComputingSettings.AddToResultsRepositoryBugFinder`

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

Summary: 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 `errno` to indicate error conditions, but do not follow the function call with a check on `errno` to see if the error occurred.
`Errno not reset`	You call a function that sets `errno` but do not reset `errno` prior to the call.
`Misuse of errno`	You check `errno` for error conditions following calls to functions that do not necessarily set `errno` 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 memcmp to compare two structures and in the process, compare garbage data stored in the structure padding.
Memory comparison of strings	You use memcmp 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 readlink()	You pass a buffer size argument to readlink() 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

Summary: 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 sizeof 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 <ctype.h> shall be representable as an unsigned char or be the value EOF.
MISRA C:2012 Rule 21.14	The Standard Library function memcmp shall not be used to compare null terminated strings.

Rule	Description
MISRA C:2012 Rule 21.15	The pointer arguments to the Standard Library functions memcpy, memmove and memcmp 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 memcmp shall point to either a pointer type, an *essentially signed type*, an *essentially unsigned type*, an *essentially Boolean type* or an *essentially enum type*.
MISRA C:2012 Rule 21.17	Use of the string handling function from <string.h> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters.
MISRA C:2012 Rule 21.18	The size_t argument passed to any function in <string.h> shall have an appropriate value.
MISRA C:2012 Rule 21.19	The pointers returned by the Standard Library functions localeconv, getenv, setlocale or strerror shall only be used as if they have pointer to const-qualified type.
MISRA C:2012 Rule 21.20	The pointer returned by the Standard Library functions asctime, ctime, gmtime, localtime, localeconv, getenv, setlocale or strerror 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 errno shall be set to zero prior to a call to an *errno-setting function*.
MISRA C:2012 Rule 22.9	The value of errno shall be tested against zero after calling an *errno-setting function*.
MISRA C:2012 Rule 22.10	The value of errno shall only be tested when the last function to be called was an *errno-setting function*.

New Code Metrics: See number of lines in header files and number of local variables per function

Summary: In R2017a, Polyspace can provide the following new code complexity metrics:

- Number of lines and number of lines without comments in header files
- Number of local non-static variables for every function and method
- Number of static variables for every function and method

Benefits: You can determine the memory footprints of your code using these new metrics (along with other already existing metrics).

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 3-25.
- `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

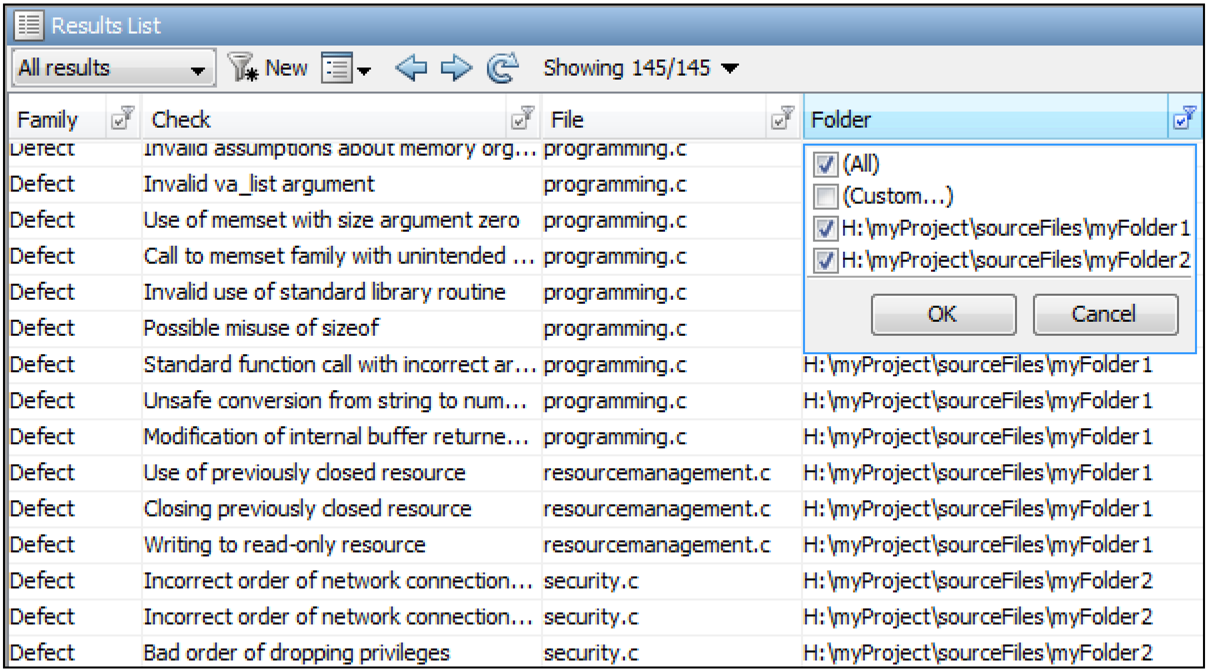
MISRA C: 2004 Rule 5.1	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.	The rule checker shows all identifiers that have the same first 31 characters as one rule violation. Previously, every pair of identifiers with same 31 characters was shown as a separate violation.
		For instance, in the following code snippet, the rule violation appears only once.
		`extern int`
		` engine_exhaust_gas_temperature_raw;`
		`static int`
		` engine_exhaust_gas_temperature_scaled;`
		`static int`
		` engine_exhaust_gas_temperature_cutoff;`
		Previously, the violation was shown three times.
		You have to review only one rule violation for every group of identifiers with the same 31 characters. You can still see all instances of conflicting identifier names in the event history of that rule violation.

`MISRA C:2012 Rule 8.5`	An external object or function shall be declared once in one and only one file.	The rule checker considers that variables or functions declared `extern` in a non-header file violates this rule.

Reviewing Results

Folder Names in Results: Filter or organize analysis results by source folder names

Summary: In R2017a, the source folder name is shown in the list of analysis results.



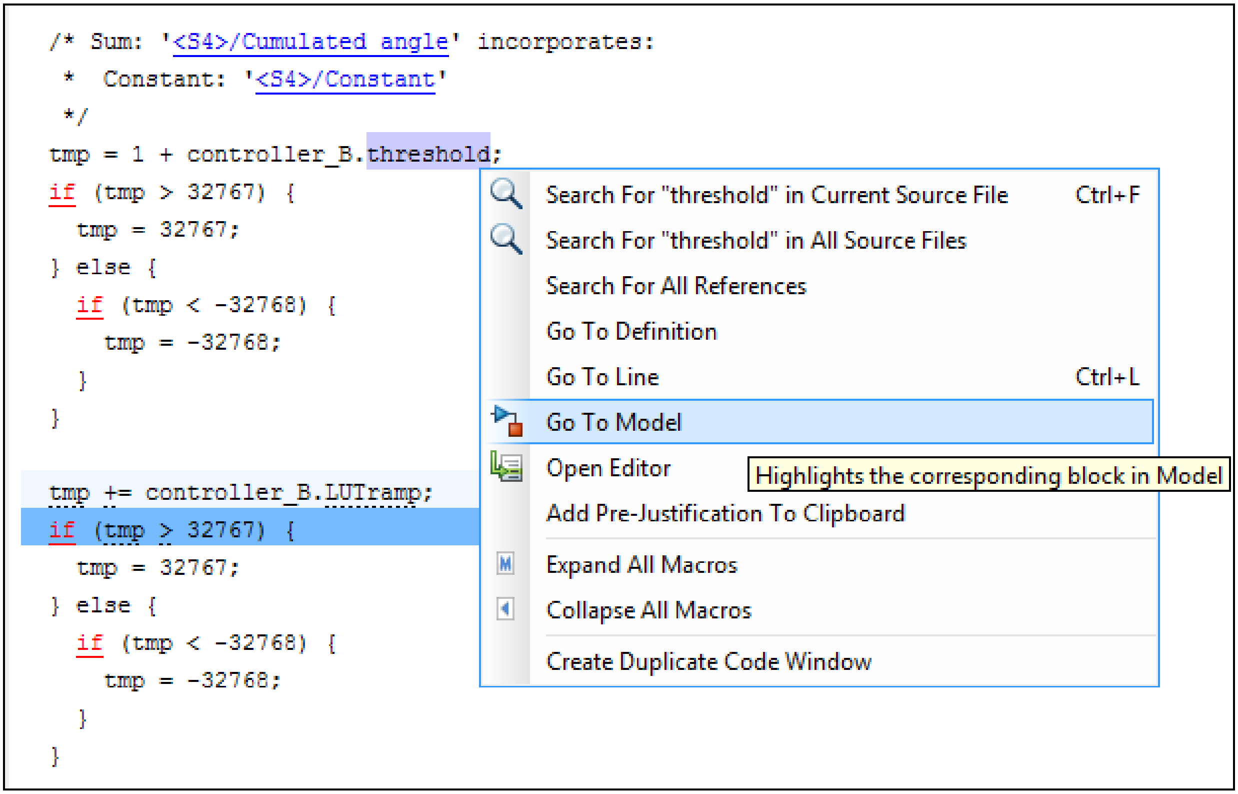
Family	Check	File	Folder
Defect	Invalid assumptions about memory org…	programming.c	
Defect	Invalid va_list argument	programming.c	
Defect	Use of memset with size argument zero	programming.c	
Defect	Call to memset family with unintended …	programming.c	
Defect	Invalid use of standard library routine	programming.c	
Defect	Possible misuse of sizeof	programming.c	
Defect	Standard function call with incorrect ar…	programming.c	H:\myProject\sourceFiles\myFolder1
Defect	Unsafe conversion from string to num…	programming.c	H:\myProject\sourceFiles\myFolder1
Defect	Modification of internal buffer returne…	programming.c	H:\myProject\sourceFiles\myFolder1
Defect	Use of previously closed resource	resourcemanagement.c	H:\myProject\sourceFiles\myFolder1
Defect	Closing previously closed resource	resourcemanagement.c	H:\myProject\sourceFiles\myFolder1
Defect	Writing to read-only resource	resourcemanagement.c	H:\myProject\sourceFiles\myFolder1
Defect	Incorrect order of network connection…	security.c	H:\myProject\sourceFiles\myFolder2
Defect	Incorrect order of network connection…	security.c	H:\myProject\sourceFiles\myFolder2
Defect	Bad order of dropping privileges	security.c	H:\myProject\sourceFiles\myFolder2

☑ (All)
□ (Custom…)
☑ H:\myProject\sourceFiles\myFolder1
☑ H:\myProject\sourceFiles\myFolder2

Benefits: 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

Summary: In R2017a, you can trace an instance of a variable in generated code back to your model.

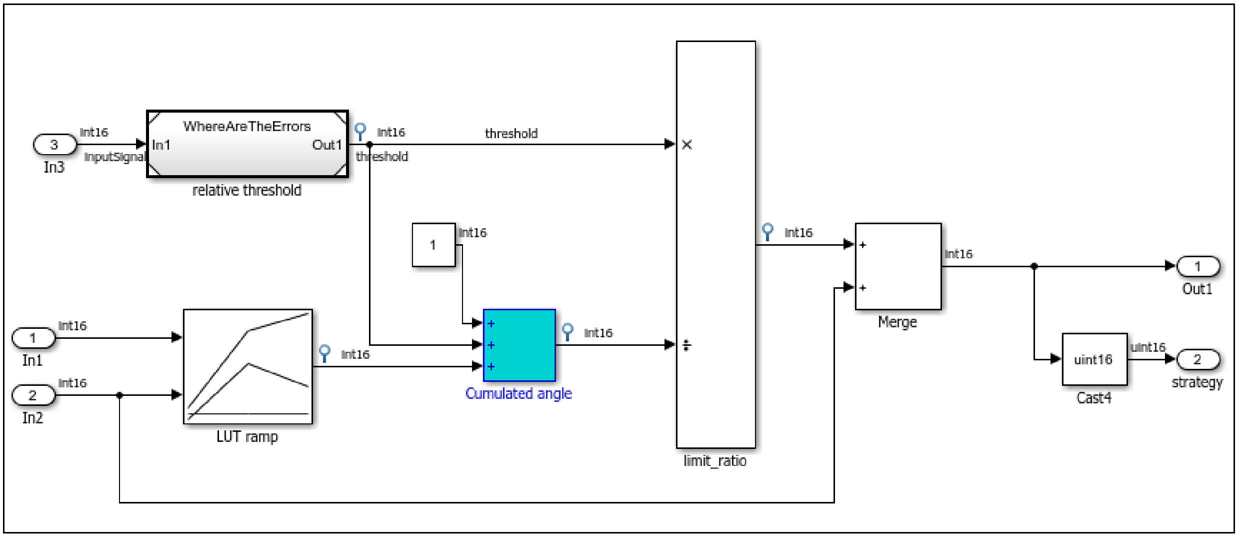


```
/* Sum: '<S4>/Cumulated angle' incorporates:
 *  Constant: '<S4>/Constant'
 */
tmp = 1 + controller_B.threshold;
if (tmp > 32767) {
  tmp = 32767;
} else {
  if (tmp < -32768) {
    tmp = -32768;
  }
}

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.



Benefits:

- *More convenient navigation*: Previously, you traced back from code to model via links in code comments. You can now navigate from the code operations themselves.
- *More fine-grained navigation*: You can easily identify which block in your model leads to which operation in the generated code.

Polyspace API in MATLAB: Read Polyspace analysis results from MATLAB

Summary: You can read your Polyspace analysis results into a MATLAB table. For instance, if the folder `C:\MyResults` contains results of a Polyspace analysis, enter the following:

```
resObj = polyspace.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`.

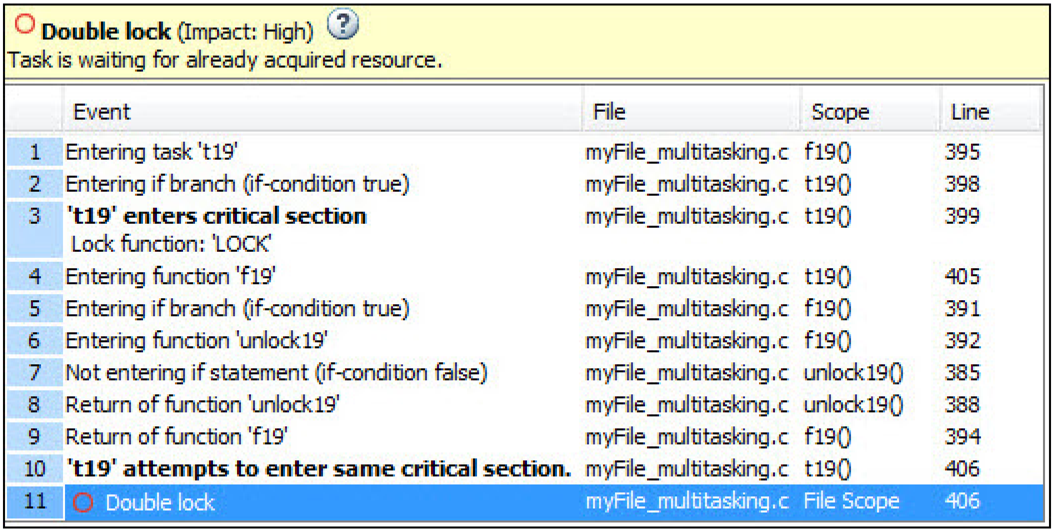
Benefits: You can use the capabilities of MATLAB to obtain graphs and statistics about your Polyspace results.

Double Lock and Other Concurrency Defects: Get help investigating the defects using detailed control flow information

Summary: 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.

Benefits: 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

Summary: In R2017a, the software provides a spreadsheet containing the Polyspace Bug Finder defect and coding rule checkers. The spreadsheet also maps the defects to Polyspace Code Prover checks (where a check exists) and 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`.

Benefits: 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.

4

R2016b

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 the 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
`polyspace.BugFinderOptions`	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.htmlpolyspace.ModelLinkBugFinderOptions	Another version of the `BugFinderOptions` object with properties specifically for model generated code. Use this object to customize analysis options and run analysis.
`polyspace.GenericTargetOptions`	A helper object for the `BugFinderOptions` classes. Use this object to customize a generic target.
`polyspace.DefectsOptions`	A helper object for the `BugFinderOptions` classes. Use this object to customize the list of defects checked during the analysis.
`polyspace.CodingRulesOptions`	A helper object for the `BugFinderOptions` object. Use this object to customize the list of coding rules checked during the analysis.

Methods

Name	Description
`polyspace.Options.copyTo`	Copy settings between options objects. You can use this method to copy options from a `BugFinderOptions` object to a `CodeProverOptions` object and vice versa.
`polyspace.Options.generateProject`	Generate a `.psprj` file from an options object to open in the Polyspace interface.

Functions

Name	Description
`polyspaceBugFinder`	Run an analysis using `BugFinderOptions` objects or `.psprj` files.

Configuration Parameters Help: View descriptions of Polyspace options in Simulink configuration parameters

When you use the Simulink® plugin, you must set Simulink configuration parameters to run your analysis. If you need help setting the configuration parameters, you can now right-click a configuration parameter and get `What’s This` help. When you select `What’s This`, a help window opens with details about the different settings and limitations of the parameter.

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 `-dialect` with `-compiler` while retaining the option argument. In the user interface, this replacement is done automatically for existing projects. If you use the Wind River Diab compiler to build your source code, use the option Compiler (-compiler) with argument `diab`.
Target processor type (-target)	Updated for the Wind River Diab compiler.	In the user interface, if you select `diab` for Compiler (-compiler), you see target processors that are tailored to the Diab compiler. For the processor specifications, see the contextual help.

Option	Change	More Information
Target operating system (`-OS-target`)	Removed from the user interface. If you use the option in your scripts, you see a warning.	Option will be permanently removed in a future release. Remove the option from your scripts. For some option arguments, you might have to perform these additional steps: • `Linux`: If you get compilation errors, use a `gnux.x` argument for Compiler (-compiler). Sometimes, you might have to explicitly define operating-system-specific macros such as `linux`, `unix`, or `__linux__`. See Preprocessor definitions (-D). • `Visual`: Use a `visualx.x` argument for Compiler (-compiler). • `Vxworks`: Use the options from the VxWorks templates. Create a Polyspace project using one of the VxWorks templates and generate a script from your project. Copy the options related to the VxWorks template from this script. For more information, see Create Project Using Configuration Templateand the reference page for `-generate-launching-scripts-for`. • `Solaris`: Just remove the option `-OS-target`. • `no-predefined-OS`: Just remove the option `-OS-target`.

Changes in analysis options and binaries

In R2016b, the following options have been added, changed, or removed.

For **Target & Compiler** options, see “Changes in Target & Compiler analysis options” on page 4-8. 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 `single-unit-rules` and `system-decidable-rules`	Subsets now available from the drop-down list.	These subsets are available for Check MISRA C:2004 (-misra2), Check MISRA AC AGC (-misra-ac-agc), and Check MISRA C:2012 (-misra3)

Removed Options

Option	Status	Description
Import Folder (`-import-dir`)	Warning	Option will be removed in a future release.
`-easy-setup-preprocess`	Warning	Option will be removed in a future release.
`polyspace-automatic-verification`	Warning	Binary will be removed in a future release.
`polyspace-verifier`	Warning	Binary will be removed in a future release.
`rte-kernel`	Warning	Binary will be removed in a future release.
`polyspace-remote`	Warning	Binary will be removed in a future release.
`gui-api`	Warning	Binary will be removed in a future release. Use instead, `polyspace-comments-import`.
Files and folders to ignore (`-includes-to-ignore`)	Error	Use the option Do not generate results for (`-do-not-generate-results-for`) to suppress results from headers and sources in certain files or folders.
`-support-FX-option-results`	Error	Option will be removed in a future release.
`polyspace-vcproj`	Removed	Use `polyspace-configure` or the Polyspace Add-In for Visual Studio instead.

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 `sizeof`	EXP09-C: Use sizeof to determine the size of a type or variable

Numerical

Name	Description	CERT C Rule
Use of plain char type for numerical value	Plain `char` variable used in arithmetic operation without explicit signedness	INT07-C: Use only explicitly signed or unsigned char 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.
	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 defects

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 4-12.

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	You can see a graphical view of the call sequence leading to conflicting operations on the shared variable. 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.

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:

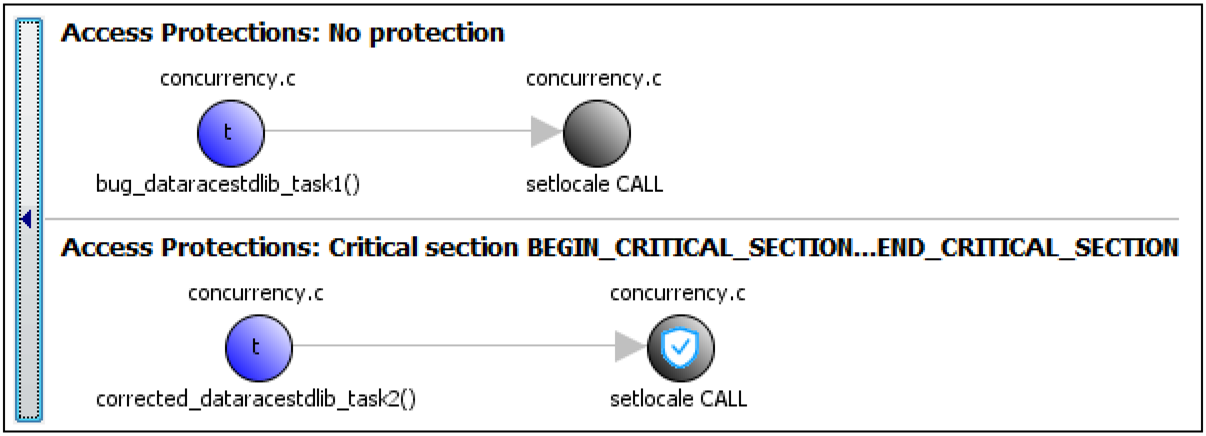
- Two function call sequences leading to the two operations.

  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.


Access Protections: No protection
concurrency.c
t
bug_dataracestdlib_task1()
concurrency.c
setlocale CALL
Access Protections: Critical section BEGIN_CRITICAL_SECTION...END_CRITICAL_SECTION
concurrency.c
t
corrected_dataracestdlib_task2()
concurrency.c
setlocale 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.

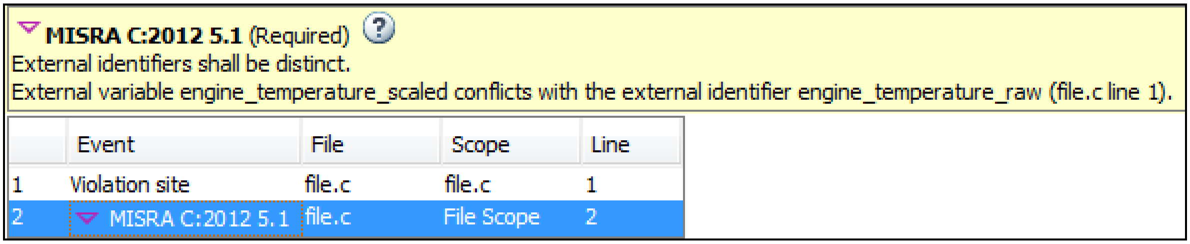
To filter results, you can use the following graphs:

- **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.

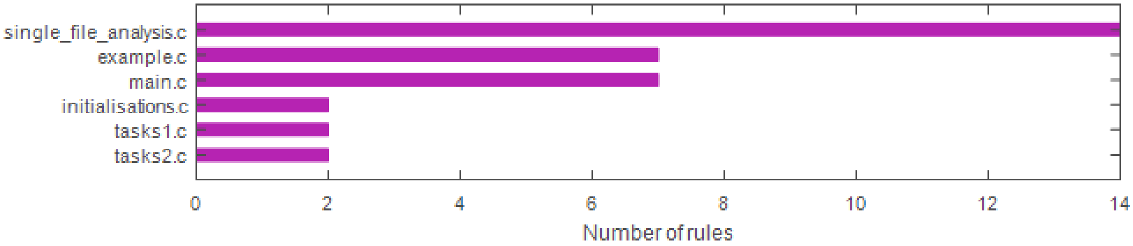
Coding Rules in Report: View improved presentation of coding rules violations in report

In R2016b, the following improvements have been made in how coding rule violations appear in the report.

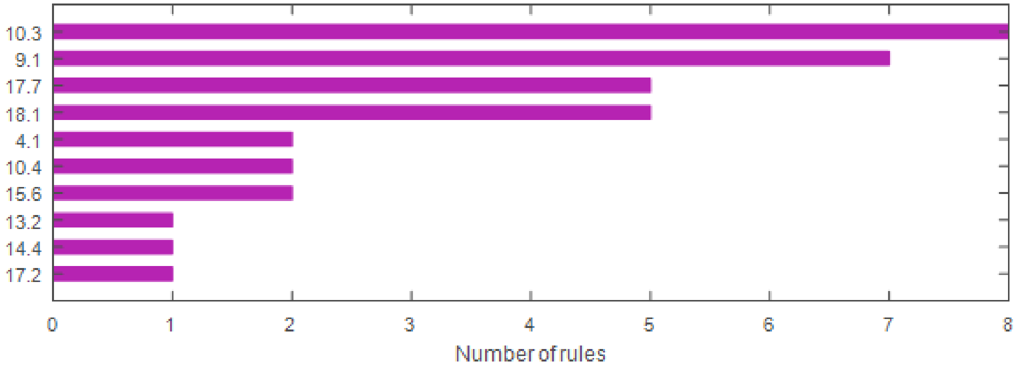
Coding Rule Graphs

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.

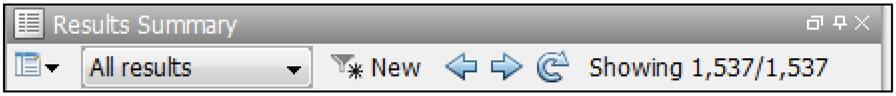
- The report generation is faster. For large results, the report generation is much less likely to cause out-of-memory errors.
- The reports use an improved visual display.

Changes in Polyspace User Interface

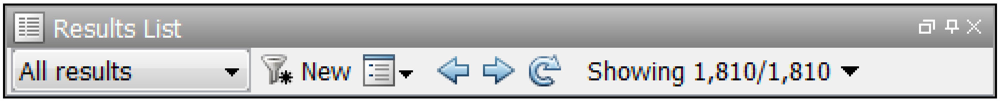
The following table lists minor changes to the user interface including new pane names and new icons.

- **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.



5

R2016a

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 5-7.

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

MISRA C:2012 Rules

Rule	Description	Improvement
`MISRA C:2012 Rule 13.2`	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.	The rule checker no longer flags expressions with the comma operator that can be evaluated in only one order. For instance, the statement `ans = (val++, val++)` does not violate this rule.

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	• 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

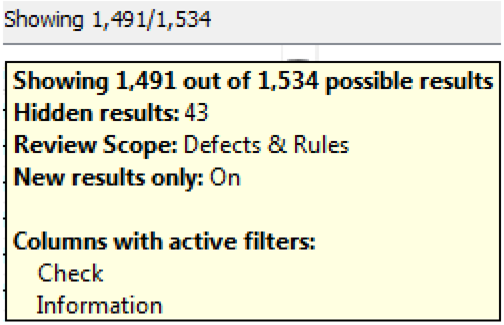
In R2016a, on the **Results Summary** or **Result Details** pane, if you start typing a review comment that you have previously entered, a drop-down list shows the previous entry. Select the previous comment from this list instead of retyping the comment.

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.



Showing 1,491/1,534

Showing 1,491 out of 1,534 possible results
Hidden results: 43
Review Scope: Defects & Rules
New results only: On

Columns with active filters:
Check
Information

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

7

R2015b

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 • **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: • `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: • `defined-by-standard (default)` • `auto-signed-first` • `auto-unsigned-first`
`-support-FX-option-results`	No longer available in the user interface	

Option	Change	Description
`-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**.
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 `-misra-ac-agc OBL-rules` 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 7-17.

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.

Rule	Description
MISRA C:2012 Rule 17.5	The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.
MISRA C:2012 Rule 17.8	A function parameter should not be modified.
MISRA C:2012 Rule 21.12	The exception handling features of `<fenv.h>` should not be used.
MISRA C:2012 Rule 22.1	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.
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 `extern` in a non-header file violate this rule.

Rule	Description	Improvement
`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 `#include` directives.	If the character `\` or `\\` occurs between the `<` and `>` in `#include <filename>` (or between `"` and `"` in `#include "filename"`), Polyspace no longer raises violation of this rule. Therefore, you can use Windows paths to files in place of `filename` 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.

Rule	Description	Improvement
`MISRA C:2012 Rule 1.1`	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.	If a rule violation occurs because your `.c` file contains too many macros, Polyspace places the rule violation at the beginning of the file instead on the last macro usage. 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.
`MISRA C:2012 Rule 10.4`	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.	• If one of the operands is the constant zero, Polyspace does not raise a violation of this rule. • If one of the operands is a signed constant and the other operand is unsigned, the rule violation is not raised if the signed constant has the same representation as its unsigned equivalent. For instance, the statement `u8b = u8a + 3;`, where `u8a` and `u8b` are `unsigned char` variables, does not violate the rule because the constants `3` and `3U` have the same representation.

Checking Coding Rules Using Text Files

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

For more information on checking for coding rules using text files, see Format of Custom Coding Rules File.

Changes to Bug Finder Defects

- “New Defects” on page 7-18
- “Updated Defects” on page 7-25

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

Name	Description
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	`delete` operates on a `void*` 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 setjmp/longjmp	`setjmp` and `longjmp` cause deviation from normal control flow

Programming Defects

Name	Description
Bad file access mode or status	Access mode argument of function in fopen or open group is invalid
Call to memset with unintended value	memset or wmemset used with possibly incorrect arguments
Copy of overlapping memory	Source and destination arguments of a copy function have overlapping memory
Exception caught by value	catch statement accepts an object by value
Exception handler hidden by previous handler	catch statement is not reached because of an earlier catch 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 va_list argument	Variable argument list used after invalidation with va_end or not initialized with va_start or va_copy
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 sizeof	Use of sizeof 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 memset with size argument zero	Size argument of function in memset family is zero

Name	Description
Variable length array with nonpositive size	Size of variable-length array is zero or negative
Writing to const qualified object	Object declared with a `const` 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 `chroot`
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 `umask` allows external user too much control
Uncleared sensitive data in stack	Variable in stack is not cleared and contains sensitive data

Name	Description
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 `/../`, `/abs/path/`, 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
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 `virtual` base class method instead of overriding it
Missing explicit keyword	Constructor missing the `explicit` 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

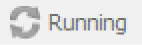
Name	Status	Additional Information
Integer conversion overflow Integer overflow Invalid use of standard library routine Shift operation overflow Sign change integer conversion overflow Shift of a negative value Unsigned integer conversion overflow Unsigned integer overflow	Updated	The defects do not appear on computations involving constants only. For instance, the assignment `unsigned int var = -1;` 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

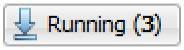
Results in Real Time: View results as they are produced

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.

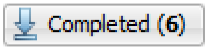


When results are available, this button becomes active.



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:

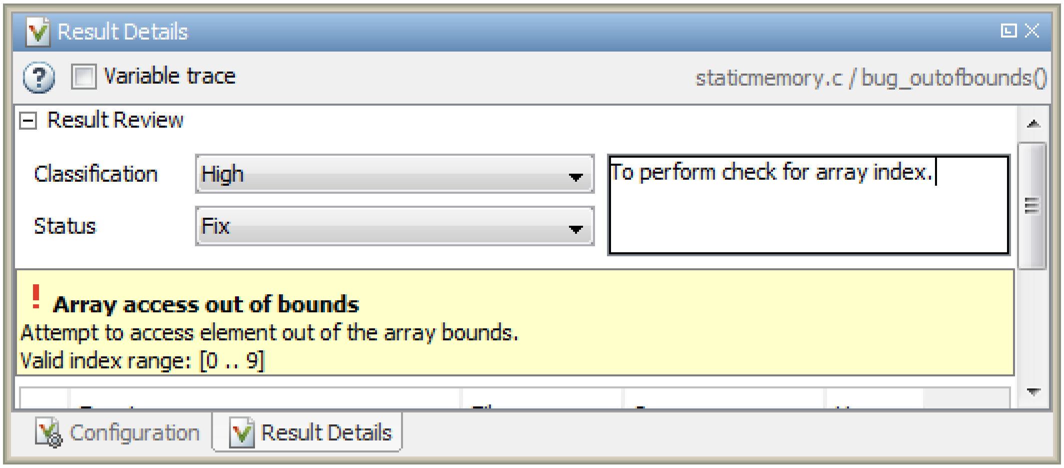
- Criticality, or whether the defect is likely to cause a code failure.

- 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.



Classification: High

Status: Fix

To perform check for array index.

Array access out of bounds
Attempt to access element out of the array bounds.
Valid index range: [0 .. 9]

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++).

8

R2015a

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
- **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.	- **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 `#if` preprocessor directives. Fewer results for variables cast to effective Boolean types.	MISRA C:2004 Rules — Chapter 12: Expressions
Rule 12.12	Fewer results when converting to an array of `float`	MISRA C:2004 Rules — Chapter 12: Expressions

MISRA C:2012

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

MISRA C++:2008

Rule Number	Effect	More Information
Rule 5-0-3	Fewer results on enumeration constants when the type of the constant is the enumeration type.	MISRA C++ Rules — Chapter 5
Rule 6-5-1	Fewer results on compliant vector variable iterators.	MISRA C++ Rules — Chapter 6
Rule 14-8-2	Fewer results for functions contained in the Files and folders to ignore (C++) option.	MISRA C++ Rules — Chapter 14
Rule 15-3-2	Fewer results for user-defined return statements after a `try` block.	MISRA C++ Rules — Chapter 15

Reviewing Results

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.

9

R2014b

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

Compatibility Considerations

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

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	*matlabroot*/toolbox/polyspace / psdistcomp/bin/setup-polyspace-cluster

Import Visual Studio project being removed

The **File > Import Visual Studio project** will be removed in a future release. Instead, use the **Create from build system** option during New Project creation. For more information, see 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.
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: • `if` conditions that are always true without a corresponding `else`. This check is covered by the Useless if defect. • Code following control-flow statements such as `break`, `return`, or `goto` 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 `break`, `return`, or `goto`.
Declaration mismatch	Updated	Updated for `#pragma` 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 ℹ.

   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.

10

R2014a

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 ▷ Run. In the **Project Browser**, you see your new Polyspace project with the required source files, include folders, and **Target & Compiler** options.
- On the MATLAB command line: Use the `polyspaceConfigure` function. For more information, see Create Project from MATLAB Command Line.

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

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

- `gnu4.7` for GNU 4.7

- `visual11.0` for Microsoft Visual Studio C++ 2012

For more information, 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.	For each coding rule, you can select three results: • **Error**: Analysis stops if the rule is violated. The rule violation is displayed on the **Output Summary** tab in the Project Manager perspective. • **Warning**: Analysis continues even if the rule is violated. The rule violation is displayed on the **Results Summary** pane in the Result Manager perspective. • **Off**: Polyspace does not check for violation of the rule.	For each coding rule, you can select two results: • **On**: Analysis continues even if the rule is violated. The rule violation is displayed on the **Results Summary** pane in the Result Manager perspective. • **Off**: Polyspace does not check for violation of the rule.

Coding Rules Feature	R2013b	R2014a

Format of the custom coding rules file.	Each line in the file must have the syntax:
	rule `off|error|warning` *#comments*
	For example:
	`# MISRA configuration - Proj1`
	`10.5 off #don't check 10.5`
	`17.2 error`
	`17.3 warning`
		Each line in the file must have the syntax:
		rule `off|warning` *#comments*
		For example:
		`# MISRA configuration - Proj1`
		`10.5 off #don't check 10.5`
		`17.2 warning`
		`17.3 warning`

Compatibility Considerations

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

- If you run analysis from the user interface, it will be treated in the same way as the keyword `warning` The 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 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 Distributed Computing 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,

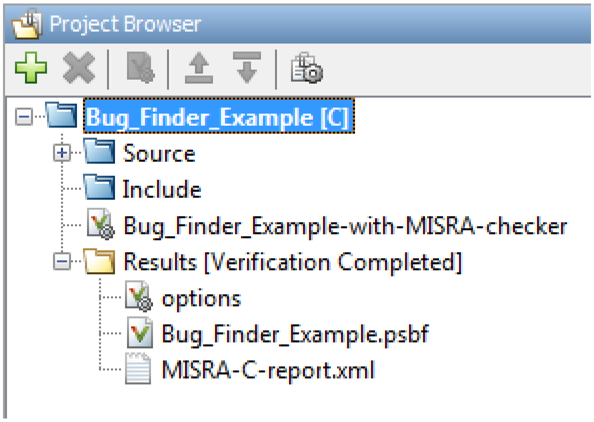
- Pre-R2013b, `arr = [9.9999E^-2,1.0001E-1]`.
- Now, `arr = 0.1`.

Reviewing Results

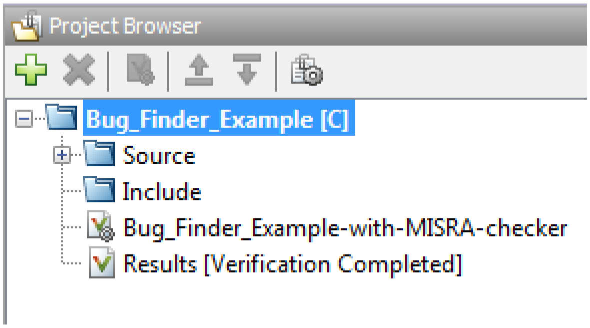
Results folder appearance in Project Browser

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

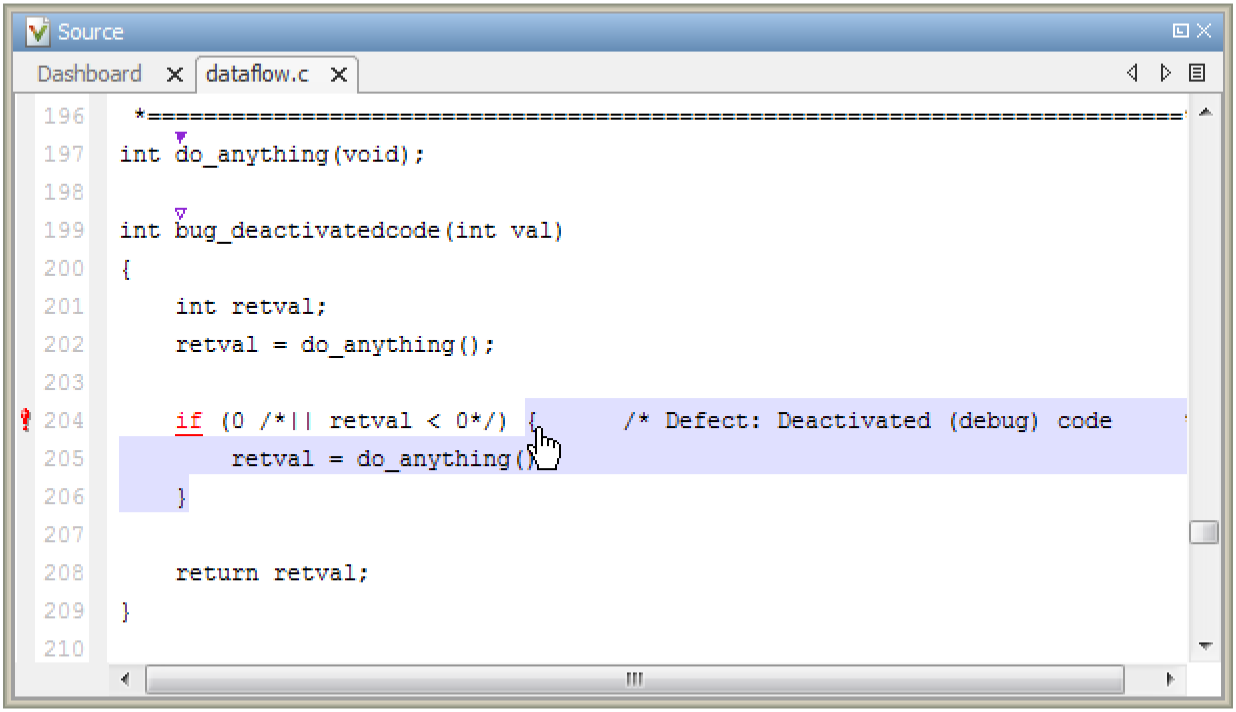

Project Browser
Bug_Finder_Example [C]
Source
Include
Bug_Finder_Example-with-MISRA-checker
Results [Verification Completed]

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 `.psbf`.	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.
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.



```
 *=======================================================================
int do_anything(void);

int bug_deactivatedcode(int val)
{
    int retval;
    retval = do_anything();

    if (0 /*|| retval < 0*/) {      /* Defect: Deactivated (debug) code
        retval = do_anything();
    }

    return retval;
}
```

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.

11

R2013b

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.