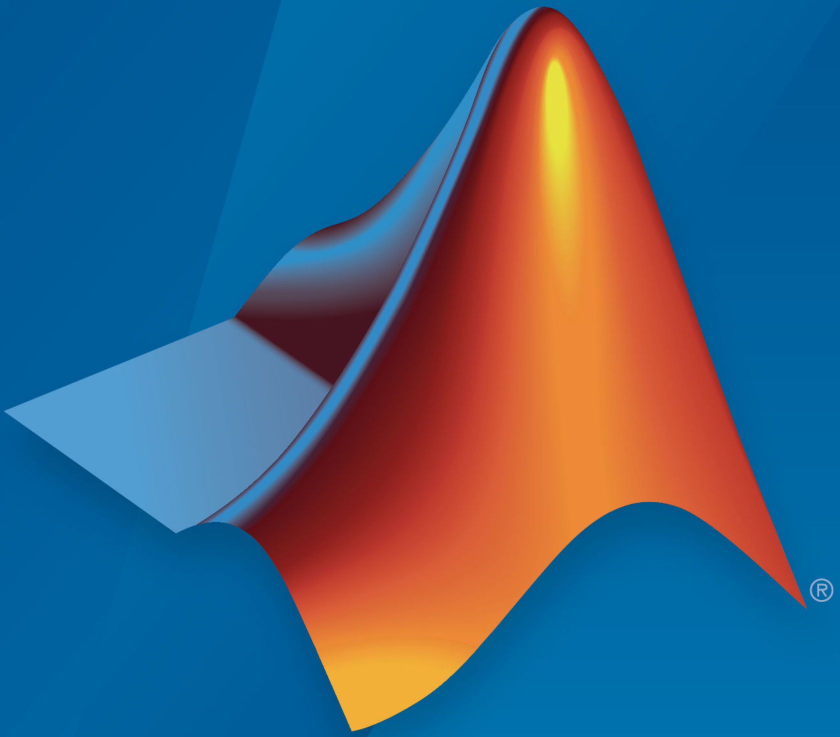
Polyspace® Bug Finder™

Getting Started Guide



MATLAB® & SIMULINK®

R2018b

MathWorks®

Revision History

September 2013	Online Only	New for Version 1.0 (Release 2013b)
March 2014	Online Only	Revised for Version 1.1 (Release 2014a)
October 2014	Online Only	Revised for Version 1.2 (Release 2014b)
March 2015	Online Only	Revised for Version 1.3 (Release 2015a)
September 2015	Online Only	Revised for Version 2.0 (Release 2015b)
October 2015	Online Only	Rereleased for Version 1.3.1 (Release 2015aSP1)
March 2016	Online Only	Revised for Version 2.1 (Release 2016a)
September 2016	Online Only	Revised for Version 2.2 (Release 2016b)
March 2017	Online Only	Revised for Version 2.3 (Release 2017a)
September 2017	Online Only	Revised for Version 2.4 (Release 2017b)
March 2018	Online Only	Revised for Version 2.5 (Release 2018a)
September 2018	Online Only	Revised for Version 2.6 (Release 2018b)

Contents

3 Analyze Code in IBM Rational Rhapsody Environment

4 Configure Server for Remote Verification and Polyspace Metrics

5 Install Polyspace Plugins

6 Polyspace Bug Finder and Polyspace Code Prover

About Polyspace Bug Finder

Polyspace Bug Finder Product Description

Identify software bugs via static analysis

Polyspace Bug Finder identifies run-time errors, concurrency issues, security vulnerabilities, and other defects in C and C++ embedded software. Using static analysis, including semantic analysis, Polyspace Bug Finder analyzes software control, data flow, and interprocedural behavior. By highlighting defects as soon as they are detected, it lets you triage and fix bugs early in the development process.

Polyspace Bug Finder checks compliance with coding rule standards such as MISRA C®, MISRA C++, JSF++, and custom naming conventions. It generates reports consisting of bugs found, code-rule violations, and code quality metrics, including cyclomatic complexity. Polyspace Bug Finder can be used with the Eclipse™ IDE and integrated into build systems.

For automatically generated code, Polyspace results can be traced back to Simulink® models and dSPACE® TargetLink® blocks.

Support for industry standards is available through IEC Certification Kit (for ISO 26262 and IEC 61508) and DO Qualification Kit (for DO-178).

Key Features

- Detection of run-time errors, concurrency issues, security vulnerabilities, and other defects
- Fast analysis of large code bases, with defects highlighted as soon as detected
- Compliance checking for MISRA C:2004, MISRA C:2012, MISRA C++:2008, JSF++, and custom naming conventions
- Cyclomatic complexity and other code metrics
- Eclipse integration
- Traceability of code verification results to Simulink models
- Bug detection with low false-positive results

Related Products

In this section...
“ Polyspace Code Prover ” on page 1-3
“Polyspace Products for Ada” on page 1-3

Polyspace Code Prover

For information about Polyspace products that verify C/C++ code, see the following:

`www.mathworks.com/products/polyspace-code-prover/`

Polyspace Products for Ada

For information about Polyspace products that verify Ada code, see the following:

`www.mathworks.com/products/polyspaceclientada/`

`www.mathworks.com/products/polyspaceserverada/`

Get Started with Polyspace Bug Finder

Compiler Requirements

Polyspace fully supports the most common compilers used to develop embedded applications. If you compile your code with one of these compilers, you can run analysis simply by specifying your compiler and target processor. See the full list of compilers on the reference page for option `Compiler (-compiler)`.

If you do not compile your code using a supported compiler, you can specify a generic compiler. If you face compilation errors from compiler-specific language extensions, you can explicitly define these extensions to work around the errors. Use the options `Preprocessor definitions (-D)` and `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

Run Polyspace Bug Finder on C/C++ Code

Polyspace Bug Finder identifies run-time errors, concurrency issues, security vulnerabilities, and other defects in C and C++ embedded software. Using static analysis, including semantic analysis, Bug Finder analyzes control flow, data flow, and interprocedural behavior. By highlighting defects as soon as they are detected, Bug Finder lets you triage and fix bugs early in the development process.

You can run Bug Finder on C/C++ code from the Polyspace user interface, in a supported development environment (IDE) such as Eclipse or using scripts. See:

- "Run Polyspace in User Interface" on page 2-3
- "Run Polyspace on Windows or Linux Command Line" on page 2-6
- "Run Polyspace in Eclipse" on page 2-7
- "Run Polyspace in MATLAB" on page 2-7

To follow the steps in this tutorial, copy the files from `*matlabroot*\polyspace\examples\cxx\Bug_Finder_Example\sources` to another folder. Here, *`matlabroot`* is the MATLAB® installation folder, for instance, `C:\Program Files\MATLAB\R2018b`.

Run Polyspace in User Interface

Open Polyspace User Interface

Double-click the `polyspace` executable in *`matlabroot`*`\polyspace\bin`. Here, *`matlabroot`* is the MATLAB installation folder, for instance, `C:\Program Files\MATLAB\R2018b`.

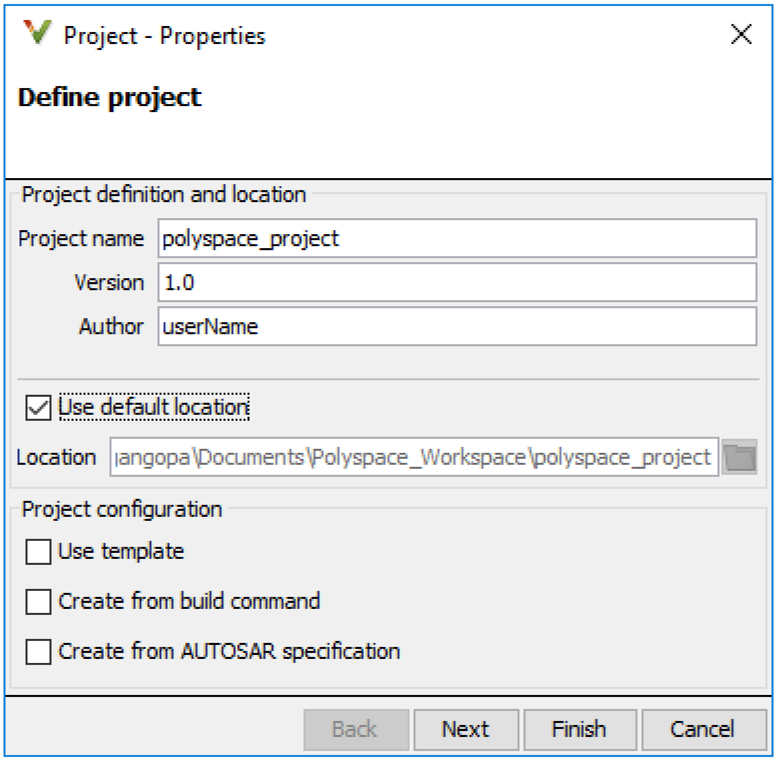
Alternatively, you can open MATLAB. In the **Apps** tab, click the Polyspace Bug Finder app.

Add Source Files

To run an analysis, you have to create a new Polyspace project. A Polyspace project points to source and include folders on your file system.

On the left of the **Start Page** pane, click **Start a new project**. Alternatively, select **File > New Project**.

After you provide a project name, on the next screens, add your source and include folders (both folders can be the same). In this tutorial, add the path to the folder in which you saved the source and include files.



Project name: polyspace_project
Version: 1.0
Author: userName
☑ Use default location
Location: jangopa\Documents\Polyspace_Workspace\polyspace_project

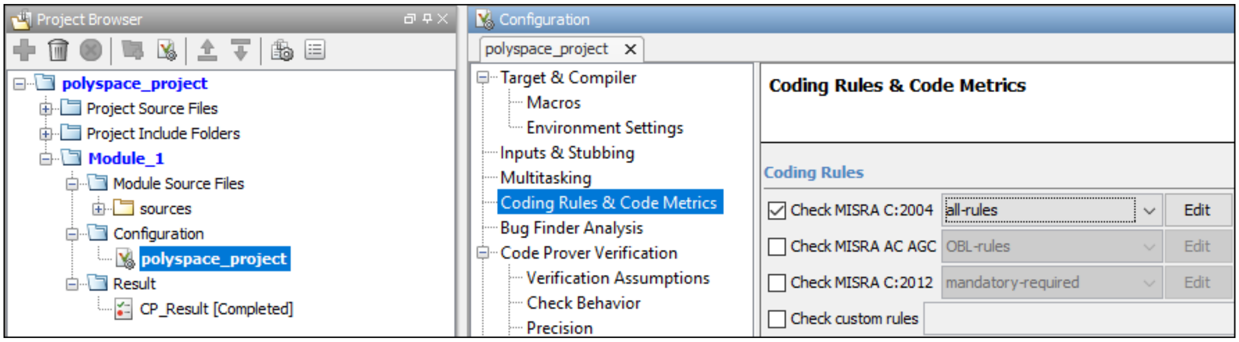
☐ Use template
☐ Create from build command
☐ Create from AUTOSAR specification

After you finish adding your source and include folders, you see a new project on the **Project Browser** pane. Your source folders are copied to the first module in the project. You can right-click a project to add more folders later. If you add folders later, you must explicitly copy them to a module.

Configure and Run Polyspace

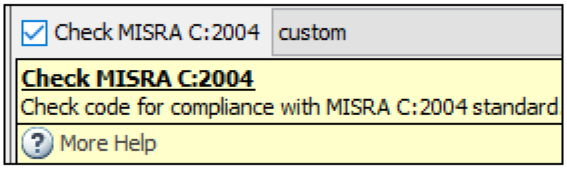
You can change the default options associated with a Polyspace analysis.

Click the **Configuration** node in your project module. On the **Configuration** pane, change options as needed. For instance, on the **Coding Rules & Code Metrics** node, select **Check MISRA C:2004**.



Check MISRA C:2004 ✓ all-rules
Check MISRA AC AGC ☐ OBL-rules
Check MISRA C:2012 ☐ mandatory-required
Check custom rules ☐

For more information, see the tooltip on each option. Click the **More help** link for context-sensitive help on the options.



Check MISRA C:2004 ✓ custom

Check MISRA C:2004
Check code for compliance with MISRA C:2004 standard

To start analysis, click **Run Bug Finder** in the top toolbar. If the button indicates Code Prover, click the arrow beside the button to switch to Bug Finder.

Follow the progress of analysis on the **Output Summary** window. After the analysis, the results open automatically.

Additional Information

See:

- “Add Source Files for Analysis in Polyspace User Interface”
- “Run Polyspace Analysis on Desktop”

Run Polyspace on Windows or Linux Command Line

You can run Bug Finder from the Windows® or Linux® command line with batch (`.bat`) files or shell (`.sh`) scripts.

To run a Bug Finder analysis, use the `polyspace-bug-finder-nodesktop` command.

To save typing the full path to the command, add the path *`matlabroot`*`\polyspace\bin` to the `Path` environment variable on your operating system. Here, *`matlabroot`* is the MATLAB installation folder, for instance, `C:\Program Files\MATLAB\R2018b`.

Navigate to the folder where you saved the files (using `cd`). Enter the following:

```
polyspace-bug-finder-nodesktop -sources numerical.c,dataflow.c -I . -results-dir .
```

Here, `.` indicates the current folder. The options used are:

- `-sources`: Specify comma-separated source files.
- `-I`: Specify path to include folder. Use the `-I` flag each time you want to add a separate include folder.
- `-results-dir`: Specify path where Polyspace Bug Finder results will be saved.

After analysis, the results are saved in the file `ps_results.psbf`. You can open this file from the Polyspace user interface. For instance, enter the following:

```
polyspace ps_results.psbf
```

Instead of specifying comma-separated sources directly on the command line, you can list the sources in a text file (one file per line). Use the option `-sources-list-file` to specify this text file.

Additional Information

See:

- “Run Polyspace Analysis from Command Line”
- `polyspace-bug-finder-nodesktop`

Run Polyspace in Eclipse

If you develop code in Eclipse or an Eclipse-based IDE, you can run Code Prover directly from your IDE.

After installing the Eclipse plugin on page 5-2, you can run Polyspace directly on the files in your Eclipse projects.

In the **Project Explorer** pane in Eclipse, select your project. To use Bug Finder for the analysis, select **Polyspace > Bug Finder**. To start the analysis, select **Polyspace > Run** (`Ctrl + R`).

After analysis, the results open automatically in Eclipse.

Additional Information

See “Run Polyspace Analysis in Eclipse”.

Run Polyspace in MATLAB

To run an analysis, use a `polyspace.Project` object. The object has two properties:

- `Configuration`: Specify the analysis options such as sources, includes, compiler and results folder using this property.
- `Results`: After analysis, read the analysis results to a MATLAB table using this property.

To run the analysis, use the `run` method of this object.

To run Polyspace on the example file `numerical.c` in *`matlabroot`*`\polyspace\examples\cxx\Bug_Finder_Examples\sources`, enter the following at the MATLAB command prompt.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(matlabroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.EnvironmentSettings.IncludeFolders = {fullfile(matlabroot, ...
'polyspace', 'examples', 'cxx', 'Bug_Finder_Example', 'sources')}
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = proj.run('bugFinder');

% Read results
bfSummary = proj.Results.getSummary('defects');
bfResults = proj.Results.getResults('readable');
```

After analysis, the results are saved in the file `ps_results.psbf`. You can open this file from the Polyspace user interface. For instance, enter the following:

```
resultsFile = fullfile(proj.Configuration.ResultsDir,'ps_results.psbf');
polyspaceBugFinder(resultsFile)
```

Additional Information

See:

- “Run Polyspace Analysis by Using MATLAB Scripts”
- `polyspace.Project`
- polyspace.Project.Configuration Properties

See Also

Related Examples

- “Review Polyspace Bug Finder Analysis Results” on page 2-10
- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Review Polyspace Bug Finder Analysis Results

Polyspace Bug Finder checks C/C++ code for defects, coding rule violations or security vulnerabilities. After you run an analysis, you open the results in the Polyspace user interface (or if you ran the analysis in Eclipse, the results open in Eclipse).

To follow the steps in this tutorial, run Polyspace using the steps in “Run Polyspace Bug Finder on C/C++ Code” on page 2-3. Alternatively, in the Polyspace user interface, open example results using **Help > Examples > Bug_Finder_Example.psprj**. If you have loaded the example results earlier and made some changes, to load a fresh copy, select **Help > Examples > Restore Default Examples**.

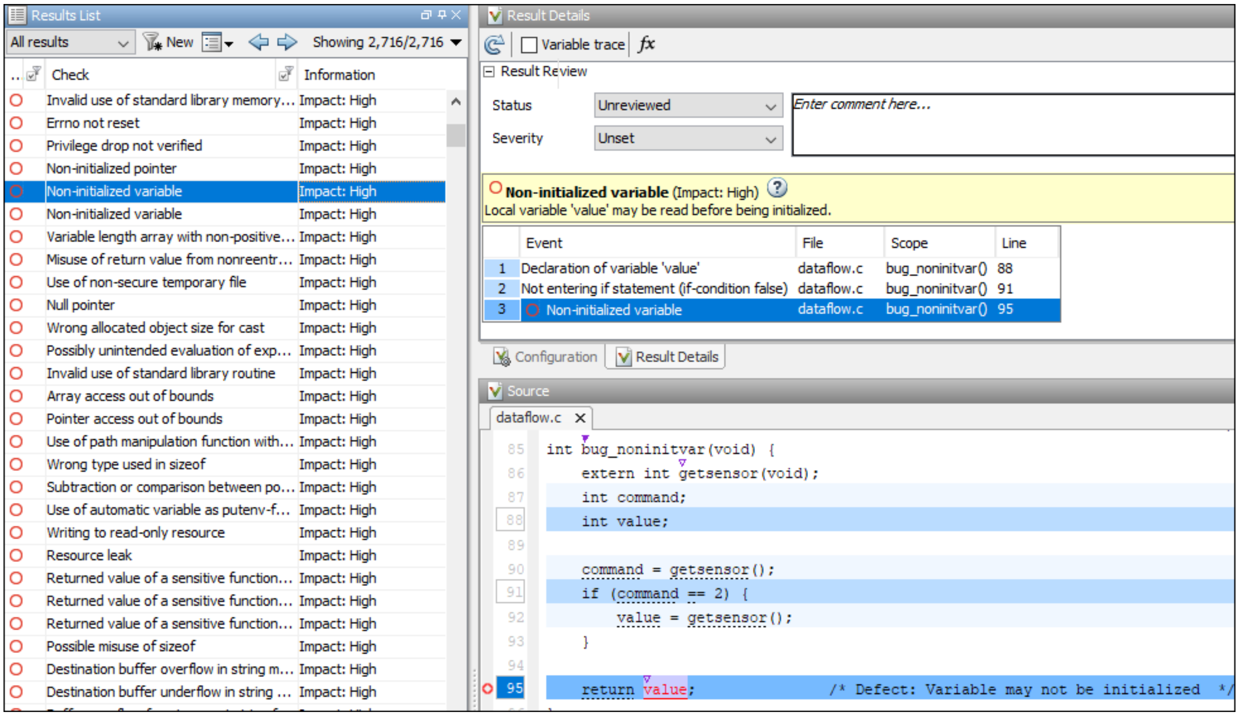
Interpret Results

Review each Polyspace result. Find the root cause of the issue.

Start from the list of results on the **Results List** pane.

- If the **Results List** pane covers the entire window, select **Window > Reset Layout > Results Review**.
- If you do not see a flat list of results, but instead see them grouped, from the ☰▾ list, select **None**.

Click the **Check** column header to sort the results alphabetically. Select one of the **Non-initialized variable** results.



Check	Information
Invalid use of standard library memory…	Impact: High
Errno not reset	Impact: High
Privilege drop not verified	Impact: High
Non-initialized pointer	Impact: High
Non-initialized variable	Impact: High
Non-initialized variable	Impact: High
Variable length array with non-positive…	Impact: High
Misuse of return value from nonreentr…	Impact: High
Use of non-secure temporary file	Impact: High
Null pointer	Impact: High
Wrong allocated object size for cast	Impact: High
Possibly unintended evaluation of exp…	Impact: High
Invalid use of standard library routine	Impact: High
Array access out of bounds	Impact: High
Pointer access out of bounds	Impact: High
Use of path manipulation function with…	Impact: High
Wrong type used in sizeof	Impact: High
Subtraction or comparison between po…	Impact: High
Use of automatic variable as putenv-f…	Impact: High
Writing to read-only resource	Impact: High
Resource leak	Impact: High
Returned value of a sensitive function…	Impact: High
Returned value of a sensitive function…	Impact: High
Returned value of a sensitive function…	Impact: High
Possible misuse of sizeof	Impact: High
Destination buffer overflow in string m…	Impact: High
Destination buffer underflow in string …	Impact: High

Result Review

Status	Unreviewed
Severity	Unset

Enter comment here...

Non-initialized variable (Impact: High)
Local variable 'value' may be read before being initialized.

	Event	File	Scope	Line
1	Declaration of variable 'value'	dataflow.c	bug_noninitvar()	88
2	Not entering if statement (if-condition false)	dataflow.c	bug_noninitvar()	91
3	Non-initialized variable	dataflow.c	bug_noninitvar()	95

```
85  int bug_noninitvar(void) {
86      extern int getsensor(void);
87      int command;
88      int value;
89
90      command = getsensor();
91      if (command == 2) {
92          value = getsensor();
93      }
94
95      return value;                 /* Defect: Variable may not be initialized  */
```

See the source code on the **Source** pane and further information about the result on the **Result Details** pane.

The **Result Details** pane also highlights a sequence of events leading to the result. For instance, for the **Non initialized variable** result, you see the following events:

- The variable `value` is declared.
- The `if` statement where `value` gets initialized is skipped.
- The variable `value` is read.

You also see these events highlighted in blue on the source code. Sometimes, these events can be located far apart in the source code. Click an event on the **Result Details** pane to navigate to the corresponding location on the source code.

Additional Information

See:

- “Interpret Polyspace Bug Finder Results”
- “Polyspace Bug Finder Results”

Address Results Through Bug Fix or Comments

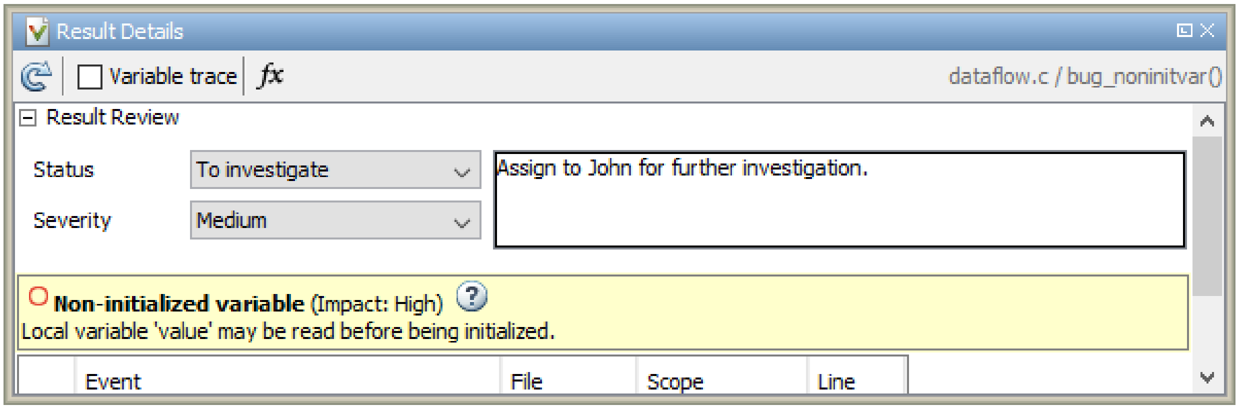
Once you understand the root cause of a Polyspace finding, you can fix your code. Otherwise, add comments to your Polyspace results to fix the code later or to justify the result. You can use the comments to keep track of your review progress.

Right-click the variable `value` on the **Source** pane. Select **Open Editor**. The code opens in a text editor. Fix the issue. For instance, you can initialize `value` during declaration.

```
int value = -1;
```

If you rerun the analysis, you do not see the **Non-initialized variable** defect.

Alternatively, if you do not want to fix the defect immediately, assign a status **To investigate** to the result. Optionally, add comments with further explanation.



Status: To investigate

Severity: Medium

Assign to John for further investigation.

Non-initialized variable (Impact: High)

Local variable 'value' may be read before being initialized.

If you assign a status **No action planned**, the result does not appear in subsequent runs on the same project.

Additional Information

See:

- “Address Polyspace Results Through Bug Fixes or Comments”
- “Annotate Code and Hide Known or Acceptable Results”

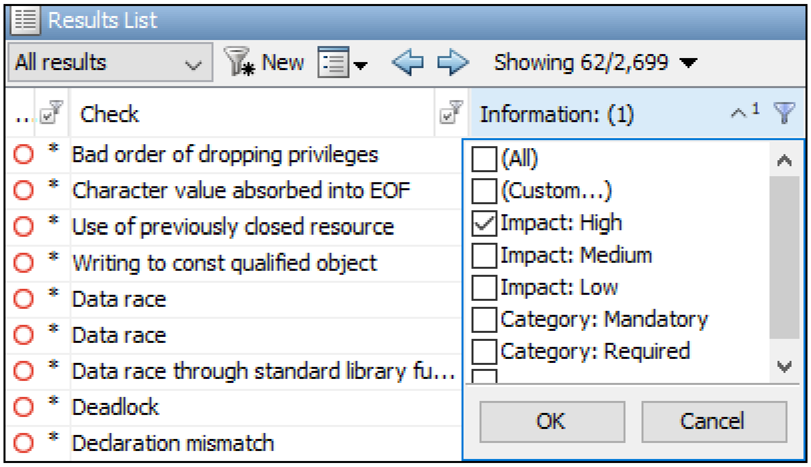
Manage Results

When you open the results of a Bug Finder analysis, you see a flat list of defects, coding rule violations or other results. To organize your review, you can narrow down the list or group results by file or result type.

For instance, you can:

- Review only high impact defects.

  Click the **Information** column header to sort defects by impact. Alternatively, you can filter out results other than high-impact defects. To begin filtering, click the icon on the column header.



	Check
○ *	Bad order of dropping privileges
○ *	Character value absorbed into EOF
○ *	Use of previously closed resource
○ *	Writing to const qualified object
○ *	Data race
○ *	Data race
○ *	Data race through standard library fu...
○ *	Deadlock
○ *	Declaration mismatch

- Review only the new results since the last analysis.

  On the **Results List** pane toolbar, click the **New** button.
- Review results in certain files or functions.

  On the **Results List** pane toolbar, from the ☰▾ list, select **File**.

Additional Information

See “Filter and Group Results”.

Analyze Code in IBM Rational Rhapsody Environment

Find Defects from IBM Rational Rhapsody

In this section...

Note The Polyspace integration with the IBM Rational Rhapsody environment will be removed after R2018b. To continue using the latest releases of Polyspace, run code analysis in the Polyspace user interface or using scripts.

Code Analysis Approach

In a collaborative Model-Driven Development (MDD) environment, software run-time errors can be produced by either design issues in the model or faulty handwritten code. You may be able to detect the flaws using code reviews and intensive testing. However, these techniques are time-consuming and expensive.

With Polyspace Bug Finder, you can analyze C/C++code that you generate from your IBM Rational Rhapsody model. As a result, you can find defects and automatically identify model flaws quickly and early during the design process.

For information about installing and using IBM Rational Rhapsody, visit the IBM website.

The approach for using Polyspace Bug Finder within the IBM Rational Rhapsody MDD environment is:

- Integrate the Polyspace add-in with your Rhapsody project. See "Adding Polyspace Profile to Model" on page 3-3.

- If required, specify Polyspace configuration options in the Polyspace environment. See “Configuring Analysis Options” on page 3-7.
- Specify the `include` path to your operating system (environment) header files and run an analysis. See “Running an Analysis” on page 3-8 and “Monitoring an Analysis” on page 3-10.
- View results, analyze errors, and locate faulty code within model. See “Viewing Polyspace Results” on page 3-11and “Locating Faulty Code in Rhapsody Model” on page 3-11.

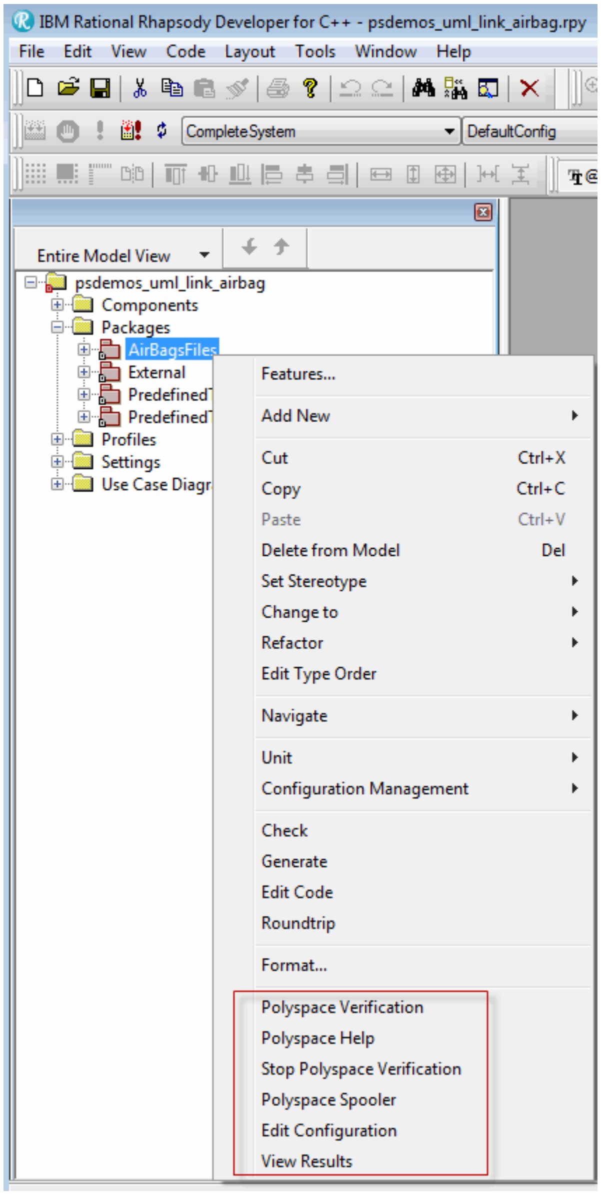
Adding Polyspace Profile to Model

Before you try to access Polyspace features, you must add the Polyspace profile to your model. Polyspace is supported for Rhapsody 7.6, 8.0, and 8.1.

Note You cannot submit local batch verifications with Polyspace for Rhapsody (for example, using local Parallel Computing Toolbox™ workers). If you want to submit local batch verifications, use the Polyspace environment or the MATLAB command, `polyspaceBugFinder`.

1. In the Rhapsody editor, select **File > Add Profile to Model**. The Add Profile to Model dialog box opens.
2. Navigate to the folder *`matlabroot`*`\polyspace\plugin\rhapsody\profiles\Polyspace`.
3. Select the file `Polyspace.sbs`. Then click **Open**.

Now, if you right-click a package or file, you see Polyspace features in the context menu.



Polyspace Verification is also available from the **Tools** menu.

Note The 64-bit version of the Polyspace product does not support the **Back to model** command with the 32-bit IBM Rational Rhapsody product.

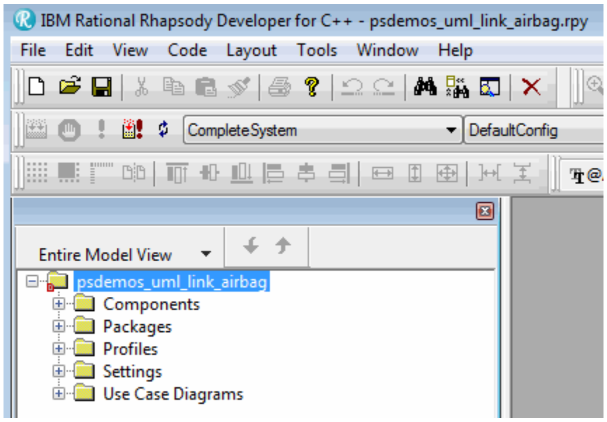
To install the 32-bit Polyspace version, from a DOS command window, run the following command:

```
DVD\Installer32bits\Windows\Disk1\InstData\VM\Polyspace.exe
```

Accessing Polyspace Features

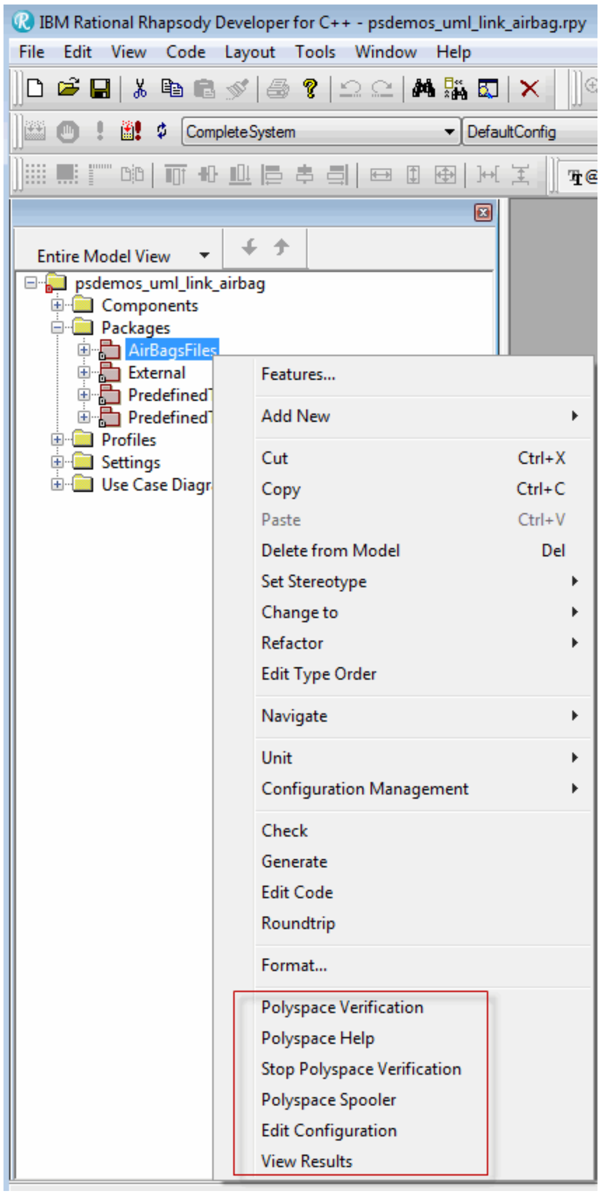
To access Polyspace features in the Rhapsody editor:

1 Open the model that you want to analyze. For example, `psdemos_uml_link_airbag.rpy` in *`matlabroot`*`/polyspace/plugin/rhapsody/psdemos`. Where *`matlabroot`* is the location of the Polyspace installation folder.



2 In the **Entire Model View**, expand the `Packages` node.

3 Right-click a package, for example, **AirBagFiles**.



You see the following Polyspace functions in the context menu:

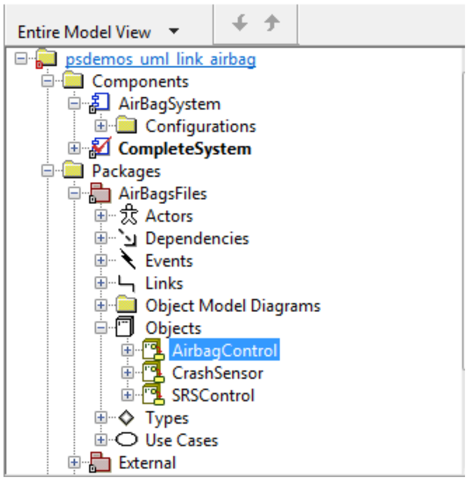
- **Polyspace Verification** — Start analysis. See “Running an Analysis” on page 3-8.
- **Polyspace Help** — Open help.
- **Stop Polyspace Verification** — Stop client-based analysis. See “Running an Analysis” on page 3-8.
- **Polyspace Job Monitor** — Open Polyspace Job Monitor. See “Monitoring an Analysis” on page 3-10.
- **Edit Configuration** — Specify analysis options. See “Configuring Analysis Options” on page 3-7.
- **View Results** — View Bug Finder results. See “Viewing Polyspace Results” on page 3-11.

Note You must add the Polyspace profile to your model before you try to access Polyspace functions. See “Adding Polyspace Profile to Model” on page 3-3.

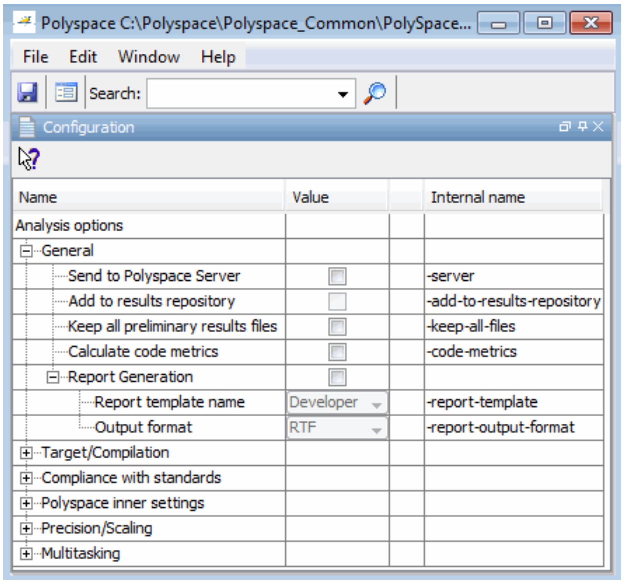
Configuring Analysis Options

To specify options for your analysis:

1 In the **Entire Model View**, right-click a package or class, for example, `AirbagControl`.



2 From the context menu, select **Edit Configuration**. The **Configuration** pane of the Polyspace environment opens.



Name	Value	Internal name
Analysis options		
General		
Send to Polyspace Server	☐	-server
Add to results repository	☐	-add-to-results-repository
Keep all preliminary results files	☐	-keep-all-files
Calculate code metrics	☐	-code-metrics
Report Generation	☐	
Report template name	Developer	-report-template
Output format	RTF	-report-output-format
Target/Compilation		
Compliance with standards		
Polyspace inner settings		
Precision/Scaling		
Multitasking		

3 Select options for your analysis. In particular, you must specify the following:

- **Compiler** (`-compiler`)
- **Include Folders** (`-I`) — Path to your operating system (environment) header files.

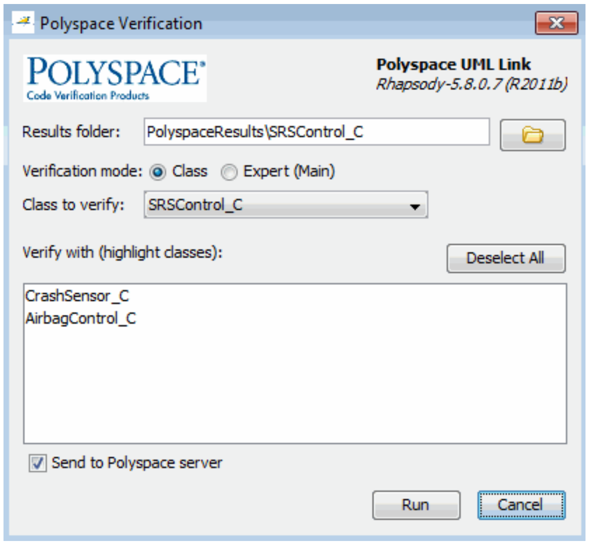
4 To save your options, in the top left corner, click the disk button.

For information on how to choose your options, see “Analysis Options”.

Running an Analysis

To start an analysis:

1 In the Rhapsody editor, select **Tools** > **Polyspace Verification**. The software opens the Polyspace Verification dialog box.



Results folder: PolyspaceResults\SRSControl_C

Verification mode: [✓ Class] Expert (Main)

Class to verify: SRSControl_C

Verify with (highlight classes):

CrashSensor_C
AirbagControl_C

[✓] Send to Polyspace server

Note Before starting an analysis, make sure that the generated code for the model is up to date.

---

2 In the Results folder field, specify a location for your analysis results.

3 Select the **Verification mode**:

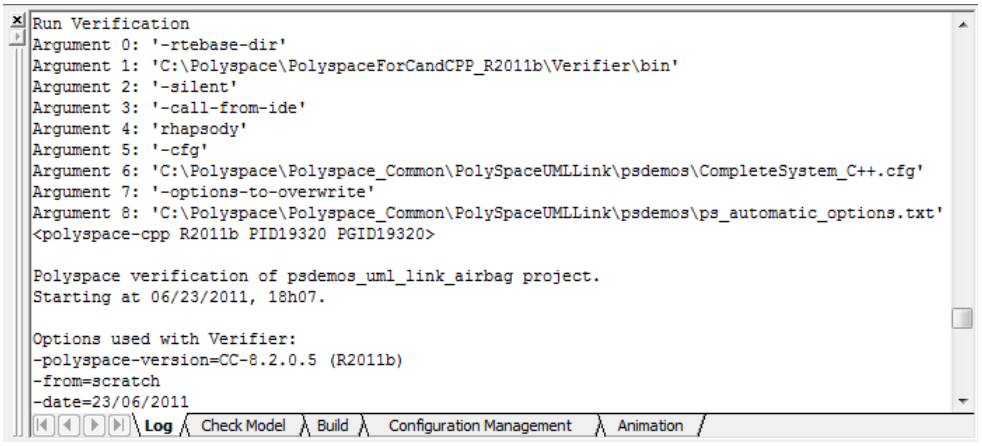
- **Class** — Select a specific class from the **Class to verify** drop-down list. In addition, under **Verify with (highlight classes)**, you can select other classes from the displayed list, for example, `CrashSensor_C`.
- **Expert** — The software analyzes code according to the **Generate a main** (`-main-generator`) options that you specify.

4 If you want to run the analysis on your Polyspace server, select **Send to Polyspace server**.

Note You cannot submit local batch analyses with Polyspace for Rhapsody (for example, using local Parallel Computing Toolbox workers). If you want to submit local batch analyses, use the Polyspace environment or the MATLAB command, `polyspaceBugFinder`.

---

5 Click **Run**. You see analysis messages on the **Log** tab of the Rhapsody editor.



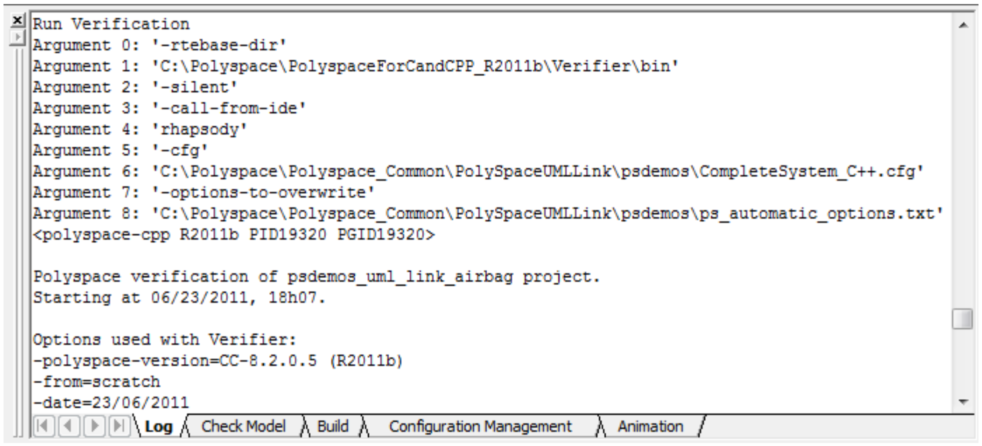
```
Run Verification
Argument 0: '-rtebase-dir'
Argument 1: 'C:\Polyspace\PolyspaceForCandCPP_R2011b\Verifier\bin'
Argument 2: '-silent'
Argument 3: '-call-from-ide'
Argument 4: 'rhapsody'
Argument 5: '-cfg'
Argument 6: 'C:\Polyspace\Polyspace_Common\PolySpaceUMLLink\psdemos\CompleteSystem_C++.cfg'
Argument 7: '-options-to-overwrite'
Argument 8: 'C:\Polyspace\Polyspace_Common\PolySpaceUMLLink\psdemos\ps_automatic_options.txt'
<polyspace-cpp R2011b PID19320 PGID19320>

Polyspace verification of psdemos_uml_link_airbag project.
Starting at 06/23/2011, 18h07.

Options used with Verifier:
-polyspace-version=CC-8.2.0.5 (R2011b)
-from=scratch
-date=23/06/2011
```

If your analysis is client-based, you can stop your analysis. In the **Entire Model View**, right-click, for example, a package or a class. From the context menu, select **Stop Polyspace Verification**.

To stop an analysis on the Polyspace Server, use the Polyspace Job Monitor. See “Monitoring an Analysis” on page 3-10.

Monitoring an Analysis

If your analysis is client-based, you can observe progress on the **Log** tab of the Rhapsody editor.



```
Run Verification
Argument 0: '-rtebase-dir'
Argument 1: 'C:\Polyspace\PolyspaceForCandCPP_R2011b\Verifier\bin'
Argument 2: '-silent'
Argument 3: '-call-from-ide'
Argument 4: 'rhapsody'
Argument 5: '-cfg'
Argument 6: 'C:\Polyspace\Polyspace_Common\PolySpaceUMLLink\psdemos\CompleteSystem_C++.cfg'
Argument 7: '-options-to-overwrite'
Argument 8: 'C:\Polyspace\Polyspace_Common\PolySpaceUMLLink\psdemos\ps_automatic_options.txt'
<polyspace-cpp R2011b PID19320 PGID19320>

Polyspace verification of psdemos_uml_link_airbag project.
Starting at 06/23/2011, 18h07.

Options used with Verifier:
-polyspace-version=CC-8.2.0.5 (R2011b)
-from=scratch
-date=23/06/2011
```

If your analysis is running on a Polyspace Server, in the **Entire Model View**, right-click, for example, a package or a class. From the context menu, select **Polyspace Job Monitor** to display the Polyspace Job Monitor. Use the Polyspace Job Monitor to manage jobs running on a Polyspace Server.

Viewing Polyspace Results

To view results from the last completed analysis, in the **Entire Model View**, right-click, for example, a package or a class. From the context menu, select **View Results**. The Polyspace environment opens, displaying the results.

For more information on Bug Finder results, see “Interpret Polyspace Bug Finder Results”.

Declarations for C Functions Without Arguments

By default, Rhapsody generates declarations for functions without parameters, using the form:

```
void my_function()
```

rather than:

```
void my_function(void)
```

This can result in the following Polyspace compilation error:

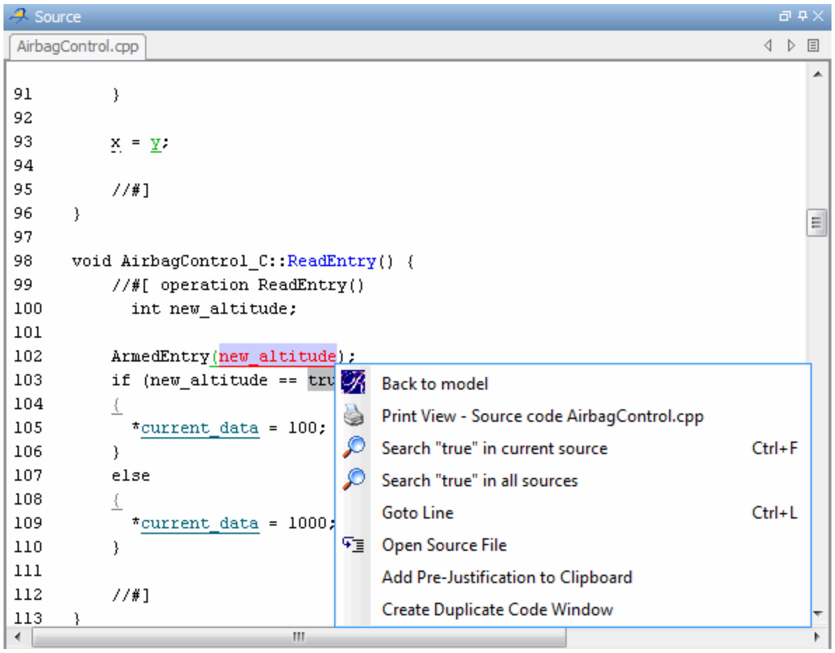
```
Fatal error: function 'my_function' has unknown prototype.
```

To avoid this problem, in Rhapsody, at the project level, set the property `C_CG::Configuration::EmptyArgumentListName` to `void`.

Locating Faulty Code in Rhapsody Model

To identify the faulty code within your Rhapsody model using Bug Finder analysis results:

1. In the Polyspace environment, navigate to an error, for example, a non-initialized variable at line 102 of `Airbag Control_C`.
2. In the Source pane, right-click the error. From the context menu, select **Back to model**.



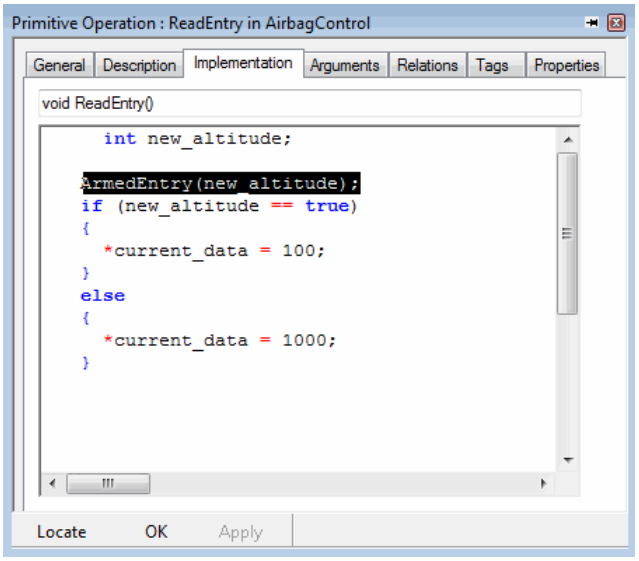
```
        }

        x = y;

        //#]
    }

    void AirbagControl_C::ReadEntry() {
        //#[ operation ReadEntry()
          int new_altitude;

        ArmedEntry(new_altitude);
        if (new_altitude == tru
        {
          *current_data = 100;
        }
        else
        {
          *current_data = 1000;
        }

        //#]
    }
```

Source
AirbagControl.cpp
Back to model
Print View - Source code AirbagControl.cpp
Search "true" in current source Ctrl+F
Search "true" in all sources
Goto Line Ctrl+L
Open Source File
Add Pre-Justification to Clipboard
Create Duplicate Code Window

Tip For the **Back to model** command to work, you must have your Rhapsody model open.

The **Back to model** command works best when the Polyspace check is enclosed by the tags `//#[` and `]#//`.

The software locates the faulty code within your Rhapsody model. Depending on the Rhapsody configuration, the faulty code appears either in a dialog box or in the code view.



void ReadEntry()

```
    int new_altitude;

  ArmedEntry(new_altitude);
  if (new_altitude == true)
  {
    *current_data = 100;
  }
  else
  {
    *current_data = 1000;
  }
```

Note The 64-bit version of the Polyspace product does not support the **Back to model** command with the 32-bit IBM Rational Rhapsody product.

To install the 32-bit Polyspace version, from a DOS command window, run the following command:

```
DVD\Installer32bits\Windows\Disk1\InstData\VM\Polyspace.exe
```

Template Configuration Files

The first time you perform an analysis, the software copies a template, Polyspace configuration file, from `matlabroot/polyspace/plugin/rhapsody/etc/template_language.psprj` to the project folder. The `template_language.psprj` files specify the default option values for code analysis. The software renames the copy to `model_language.psprj`, where:

- `model` is the name of your model
- `language` is the name of the language that the model targets, that is `C` or `C++`.

You can update the template `.psprj` file by one of the following means:

- Editing it through the Polyspace environment
- Double-clicking the file in a Windows Explorer window
- Replacing the template file with a copy of the `.psprj` file from a Rhapsody model folder

You can then share a configuration among project members and use the configuration with other projects.

Configure Server for Remote Verification and Polyspace Metrics

Set Up Polyspace Analysis on Remote Server

You can perform a Polyspace analysis locally on your desktop or on a remote server. This topic shows how to set up Polyspace on a server for remote analysis.

Choose Between Local and Remote Analysis

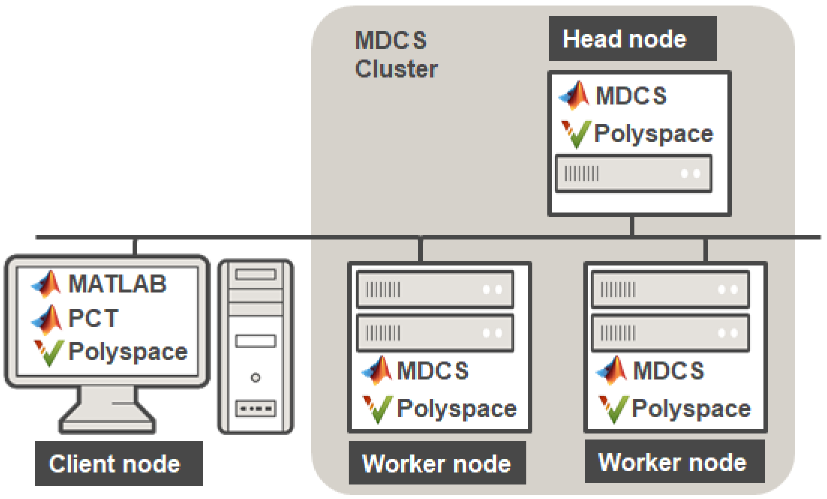
To determine when to use local or remote analysis, use the rules listed in this table.

Type	When to Use
Remote	Source files are large (more than 800 lines of code including comments) and execution time of analysis is lengthy.
Local	Source files are small and execution time of analysis is short.

Requirements for Remote Analysis

A typical distributed network for running remote analysis consists of these parts:

- **Client nodes**: On the client node, you configure your Polyspace project or scripts, and then submit a job that runs Polyspace.
- **Head node**: The head node distributes the submitted jobs to worker nodes.
- **Worker node(s)**: The Polyspace analysis runs on a worker node.



In the simplest remote analysis configuration, the same computer can serve as the head node and worker node. You can run one Polyspace analysis on one worker only. You cannot distribute the analysis over multiple workers. If you submit more than one analysis job, you can distribute the jobs over multiple workers.

This table lists the product requirements for remote analysis.

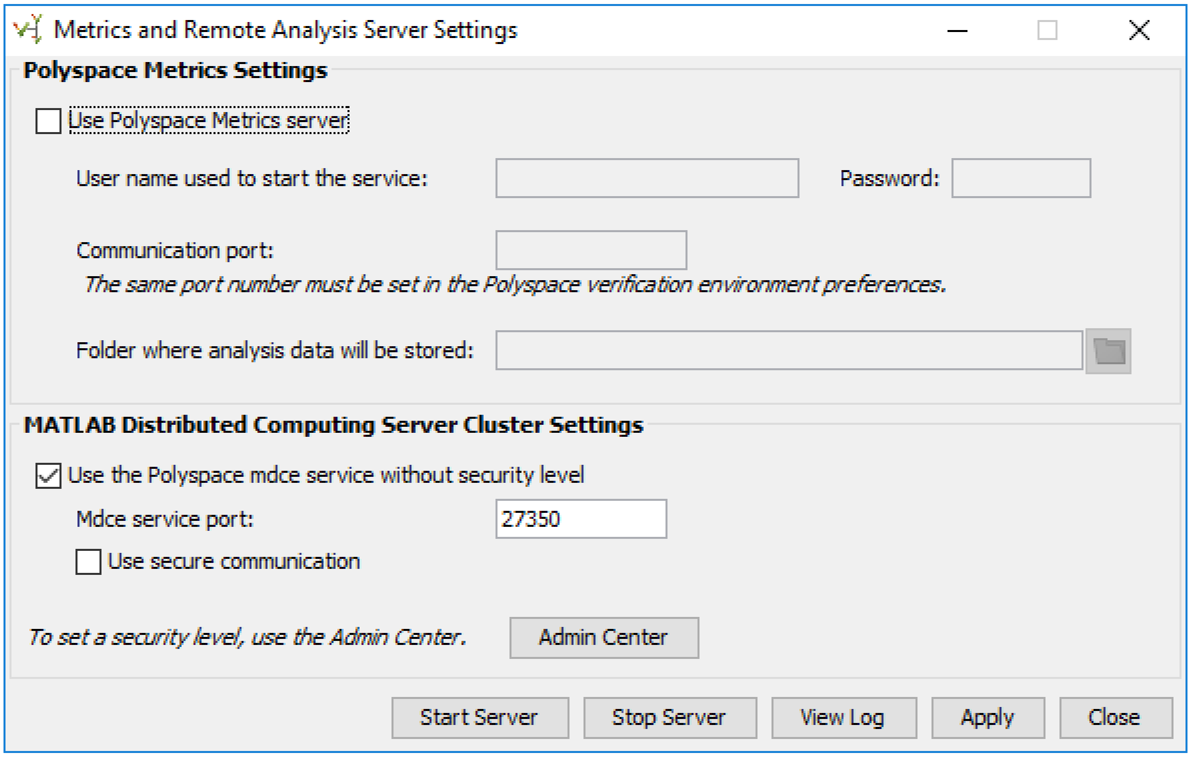
Location	Requirements
Client node	• MATLAB • Parallel Computing Toolbox • Polyspace Bug Finder or Polyspace Code Prover™ (whichever product you choose to run)
Head node and worker nodes (server side)	• MATLAB Distributed Computing Server™ • Polyspace Bug Finder • Polyspace Code Prover (if you choose to run Code Prover)

Configure and Start Server

On the computers that act as the server (head node and worker nodes), configure and start the `mdced` service through the Metrics and Remote Analysis Server Settings dialog box.

To open this dialog box, go to *`matlabroot`*`\polyspace\bin`. Here, *`matlabroot`* is the MATLAB installation folder, for instance, `C:\Program Files\MATLAB\R2018b`. Double-click the executable `polyspace-server-settings`.

Alternatively, you can open this dialog box from the Polyspace user interface. Select **Tools > Remote Analysis Server Settings**.


Metrics and Remote Analysis Server Settings

Polyspace Metrics Settings

☐ Use Polyspace Metrics server

User name used to start the service:

Password:

Communication port:

The same port number must be set in the Polyspace verification environment preferences.

Folder where analysis data will be stored:

MATLAB Distributed Computing Server Cluster Settings

☑ Use the Polyspace mdce service without security level

Mdce service port: 27350

☐ Use secure communication

To set a security level, use the Admin Center.

Configure Cluster with One Worker

You can use the same computer as the head node and worker node.

1. Select **Use the Polyspace mdce service without security level**.

   The `mdced` service runs by default with security level 0. At level 0, jobs are associated with the default user name of the user. A login or password is not required to manage and see these jobs.

   You can also use these options:

   - **Mdce service port** — The default port is 27350.

     This option specifies the port that the `mdced` service uses for server-client communication. If you change this number, you must change it on both the server and client side. On the client side, when you specify the job scheduler host name (**Tools** > **Preferences** and then **Server Configuration**), specify the port by using the notation *`hostName`*`:`*`portNumber`*. For instance, `ah-jdoe:27400`.
   - **Use secure communication** – Not selected by default.

     To encrypt communication between the job scheduler and workers, select this option.
2. To start the `mdced` service, click **Start Server**.

   The service uses the settings specified in the file `polyspace_mdce_def.bat` (Windows) or `polyspace_mdce_def.sh` (Linux) in *`matlabroot`*`\toolbox\polyspace\psdistcomp\bin`. Here, *`matlabroot`* is the MATLAB installation folder, for instance, `C:\Program Files\MATLAB\R2018b`.

The software stores the information that you specify in the Metrics and Remote Server Settings dialog box in the following file:

- On a Windows system, `%APPDATA%\PolyspaceRLDatas\polyspace.conf`
- On a Linux system, `/etc/Polyspace/polyspace.conf`

You can also set up this computer to act as the Polyspace Metrics server. Before clicking **Start Server**, select the **Use Polyspace Metrics server** box. For details, see “Set Up Polyspace Metrics” on page 4-10.

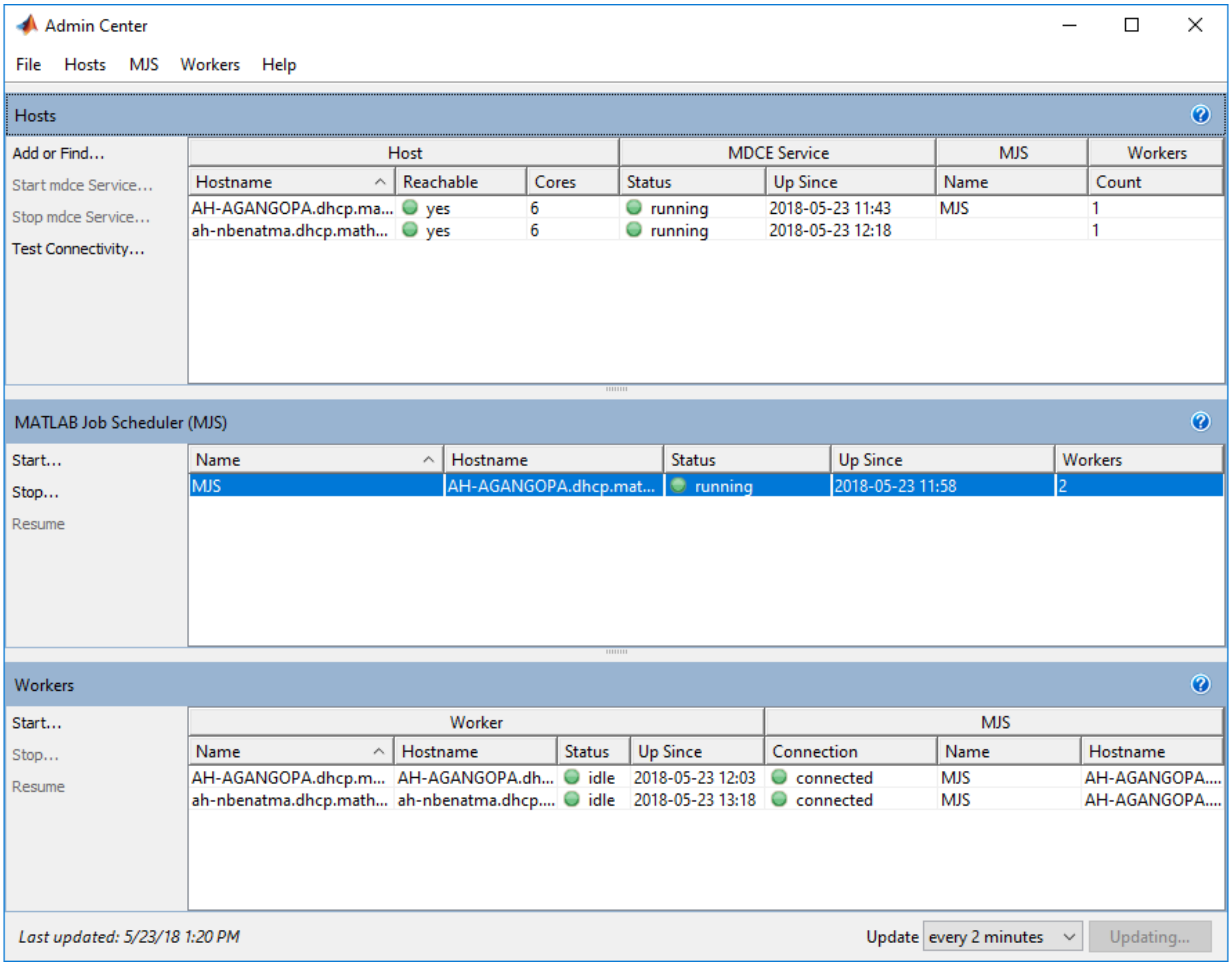
Configure Cluster with Multiple Workers

To configure a cluster with multiple workers, you must start the `mdced` service on all computers that act as worker nodes. To set up multiple workers, use the MATLAB Distributed Computing Server **Admin Center**.

You can also use this approach if you want to require authentication to use the remote server. For more information about setting up security levels for authentication, see “Set MJS Cluster Security” (MATLAB Distributed Computing Server).

To set up this configuration, on the computer that acts as the head node:

1. Open the Metrics and Remote Analysis Server Settings dialog box.
2. Click **Admin Center**.



Host			MDCE Service		MJS	Workers
Hostname	Reachable	Cores	Status	Up Since	Name	Count
AH-AGANGOPA.dhcp.ma...	● yes	6	● running	2018-05-23 11:43	MJS	1
ah-nbenatma.dhcp.math...	● yes	6	● running	2018-05-23 12:18		1

Name	Hostname	Status	Up Since	Workers
MJS	AH-AGANGOPA.dhcp.mat...	● running	2018-05-23 11:58	2

Worker				MJS		
Name	Hostname	Status	Up Since	Connection	Name	Hostname
AH-AGANGOPA.dhcp.m...	AH-AGANGOPA.dh...	● idle	2018-05-23 12:03	● connected	MJS	AH-AGANGOPA....
ah-nbenatma.dhcp.math...	ah-nbenatma.dhcp....	● idle	2018-05-23 13:18	● connected	MJS	AH-AGANGOPA....

Last updated: 5/23/18 1:20 PM

3 In the **Hosts** section, add the host names of all computers that you want to use as head and worker nodes of the cluster. Start the `mdced` service.

The service uses the settings specified in the file `*matlabroot*\toolbox\distcomp\bin\mdce_def.bat`. Here, *`matlabroot`* is the MATLAB installation folder, for instance, `C:\Program Files\MATLAB\R2018b`.

4. Right-click each host. Select either **Start MJS** (head node) or **Start Workers** (worker nodes).

   The hosts appear in the **MATLAB Job Scheduler** or **Workers** section. In each section, select the host and click **Start** to start the MATLAB Job Scheduler or the workers.

Selecting a computer as host starts the `mdced` service on that computer. You must have permission to start services on other computers in the network. For instance, on Windows, you must be in the Administrators group for other computers where you want to start the `mdced` service. Otherwise, you have to start the `mdced` services individually on each computer that acts as a worker.

For more details and command-line workflows, see:

- "Integrate MATLAB Job Scheduler (MJS)" (MATLAB Distributed Computing Server)
- `mdce`

Configure Client

Configure the client node so that it can communicate with the computer that serves as the head node of the MDCS cluster.

Configure the client node through the Polyspace environment preferences:

1. Select **Tools > Preferences**.
2. Click the **Server Configuration** tab. Under **MATLAB Distributed Computing Server cluster configuration**:

   a. In the **Job scheduler host name** field, specify the computer for the head node of the cluster. This computer hosts the MATLAB job scheduler (MJS).

      If the port used on the computer hosting the MJS is different from 27350, enter the port name explicitly with the notation `*hostName*:*portNumber*`.

   b. Due to the network setting, the job manager may be unable to connect back to your local computer. If so, enter the IP address of the client computer in the **Localhost IP address** field.

Set Up Server for Multiple Polyspace Releases

You can run jobs from multiple releases of Polyspace (for instance, R2016a and R2016b) on the same server.

- Install both releases of Polyspace and the later release of MATLAB Distributed Computing Server on the server.
- Edit the file `mdce_def.bat` or `mdce_def.sh` (located in *`matlabroot`*`toolbox\distcomp\bin\`) to refer to the earlier release. For instance, to refer to a R2016a release, find the line with `MDCS_ADDITIONAL_MATLABROOTS` and edit it like this:

  ```
  set MDCS_ADDITIONAL_MATLABROOTS=C:\Program Files\MATLAB\R2016a
  ```

Start the `mdced` service from MATLAB Distributed Computing Server **Admin Center**. See “Configure Cluster with Multiple Workers” (Polyspace Code Prover).

Once you start the Job Scheduler on the server, from your client nodes, you can submit jobs from both Polyspace releases to the same cluster.

See Also

Related Examples

- “Set Up Polyspace Metrics” on page 4-10
- “Run Polyspace Analysis on Remote Clusters”
- “Job Manager Cannot Write to Database”
- “Integrate MATLAB with Third-Party Schedulers” (MATLAB Distributed Computing Server)
- “Troubleshoot Common Problems” (MATLAB Distributed Computing Server)

Set Up Polyspace Metrics

Polyspace Metrics is a web dashboard that generates code quality metrics from your verification results. Using this dashboard, you can:

- Provide your management a high-level overview of your code quality.
- Compare your code quality against predefined standards.
- Establish a process where you review in detail only those results that fail to meet standards.
- Track improvements or regression in code quality over time.

This topic shows how to set up a Polyspace Metrics server to store Polyspace results.

Requirements for Polyspace Metrics

You can use Polyspace Metrics to:

- Store verification and analysis results.
- Evaluate and monitor software quality metrics.

This table lists the requirements for Polyspace Metrics.

Task	Location	Requirements
Project configuration and uploads to server	Client node	• MATLAB • Polyspace Bug Finder
Polyspace Metrics service	Network server or head node of MATLAB Distributed Computing Server cluster	• MATLAB • Polyspace Bug Finder Activation is not required for the Polyspace Metrics service

Task	Location	Requirements
Downloading *complete* results from Polyspace Metrics	Client node or a network computer	• MATLAB • Polyspace Bug Finder • Access to Polyspace Metrics server
Viewing results *summary* from Polyspace Metrics	A network computer	Access to Polyspace Metrics server.

You cannot merge two different Polyspace metrics databases. However, if you install a newer version of Polyspace on top of an older version, Polyspace Metrics automatically updates the database to the newest version.

Configure and Start Polyspace Metrics Server

This section shows you how to start the host server for Polyspace Metrics. After you complete this step, you must also configure the client side settings so that the Polyspace interface can interact with the Metrics server.

1 From the Polyspace environment, select **Tools > Remote Analysis Server Settings**.

Note In Linux, you need `root` privileges to start the Polyspace Metrics server.

2 Under **Polyspace Metrics Settings**, select **Use Polyspace Metrics server**.

   Specify this information:

   - **User name used to start the service** — Your user name.
   - **Password** — Your password (Windows only).
   - **Communication port** — Polyspace communication port number (default 12427). This number must be the same as the communication port number specified in the Polyspace Interface preferences. See "Configure Client Side" on page 4-12.
   - **Folder where analysis data will be stored** — Results repository for Polyspace Metrics server.

3 If you do not also want to run the analysis on a server (using a MATLAB Distributed Computing Server cluster), clear the **Start the Polyspace mdce service without security level** check box. Otherwise, when you start a server, the server doubles as a Polyspace Metrics server and an MDCS server for running analysis.

For information about starting your MDCS server, see "Set Up Polyspace Analysis on Remote Server" on page 4-2.

4 To start the Polyspace Metrics server, click **Start Server**.

Note If you are using a Mac as your Polyspace Metrics server, when you restart the machine, you must restart the Polyspace server.

The software stores the information that you specify through the Metrics and Remote Server Settings window in the following file:

- On a Windows system, `\%APPDATA%\Polyspace_RLDatas\polyspace.conf \polyspace.conf`.
- On a Linux system, `/etc/Polyspace/polyspace.conf`

To start Polyspace Metrics web server at the command line, use one of these commands:

- Windows: `perl` *`matlabroot`*`\toolbox\polyspace\psdistcomp\bin\setup-polyspace-cluster.pl`
- Linux: `./`*`matlabroot`*`/toolbox/polyspace/psdistcomp/bin/setup-polyspace-cluster`

Here, *`matlabroot`* is the MATLAB installation folder, for instance, `C:\Program Files \MATLAB\R2018b`. For more help in using the commands, use the `-h` option.

Configure Client Side

Once you have set up your Polyspace metrics server, you must set the client-side settings so that the Polyspace interface can communicate with your Metrics server.

1 Select **Tools > Preferences**.

2 Click the **Server Configuration** tab.

3 Select **Use Polyspace Metrics**.

   Specify this information:

   a If you want Polyspace to detect a server on the network that uses port 12427 (default port number), click **Automatically detect the Polyspace Metrics Server**.

   b. If you use a different port number for your Metrics server or you want to specify the server name, click **Use the following server and port**. Fill in your server name or IP address, and communication port number.

      You must specify the same communication port number for all clients that use the Polyspace Metrics service.

4. Under the **Polyspace Metrics web interface configuration** section:

   a. Specify a **Port used to download results**, default is 12428. If you change this port number, you must also change it in on the server side.
   b. Specify which protocol to use HTTP or HTTPS. If you select HTTPS for your web protocol, there are additional steps to set up the Metrics web server for HTTPS on page 4-14.
   c. Specify a web server port number for your chosen protocol. Default port numbers are:

      - HTTP — 8080
      - HTTPS — 8443

      If you change the port number from the default, you must configure the same port number for the Polyspace Metrics server. See “Configure and Start Polyspace Metrics Server” on page 4-11.

5. Under the **Upload and download settings** section:

   - Upload settings — After you review results from the Metrics repository, you can upload your comments and justifications back to the repository using **Metrics > Upload to Metrics**.

     If you want Polyspace to automatically upload your justifications to Polyspace Metrics when you save, select **Upload justifications automatically in the Polyspace Metrics repository...**.
   - Download settings — In Polyspace Metrics, when you click an item to view, Polyspace downloads your results and opens them in the Polyspace environment. Select where to download your Polyspace Metrics results, either:

     - To the project folder, or, if a project does not exist, a default folder.
     - Ask every time where to download results.

To view Polyspace Metrics, in the address bar of your web browser, enter:

```
protocol://ServerName:WSPN
```

- `protocol` is `http` or `https`.
- `ServerName` is the name or IP address of your Polyspace Metrics server.
- `WSPN` is the web server port number, the default is 8080 or 8443.

Configure Web Server for HTTPS

By default, the data transfer between Polyspace Bug Finder and the Polyspace Metrics web interface is not encrypted. You can enable HTTPS for the web protocol, which encrypts the data transfer. To set up HTTPS, you must change the server configuration and set up a keystore for the HTTPS certificate.

Before you start the following procedure, you must complete “Configure and Start Polyspace Metrics Server” on page 4-11 and “Configure Client Side” on page 4-12.

To configure HTTPS access to Polyspace Metrics:

1. Open the Metrics and Remote Server Settings dialog box. Run the following command:

   ```
   MATLAB_Install\polyspace\bin\polyspace-server-settings.exe
   ```
2. Click **Stop Daemon**. The software stops the `mdce` and Polyspace Metrics services. Now, you can make the changes required for HTTPS.
3. Open the file *`metricsRootFolder`*`\tomcat\conf\server.xml` in a text editor. Here, *`metricsRootFolder`* is the name that you specified for **Folder where analysis data will be stored**. Look for the following text:

   ```
   <!-
     <Connector port="8443" SSLEnabled="true" scheme="https"
     secure="true" clientAuth="false" sslProtocol="TLS"
     keystoreFile="<datadir>/.keystore" keystorePass="polyspace"/>
   ->
   ```

   If the text is not in your `server.xml` file:

   a. Delete the entire `..\conf\` folder.
   b. In the Metrics and Remote Server Settings dialog box, restart the daemon by clicking **Start Daemon**.
   c. Click **Stop Daemon** to stop the services again so that you can finish setting up the server for HTTPS.

The `conf` folder is regenerated, including the `server.xml` file. The file now contains the text required to configure the HTTPS web server.

4. Follow the commented-out instructions in `server.xml` to create a keystore for the HTTPS certificate.
5. In the Metrics and Remote Server Settings dialog box, to restart the Polyspace Metrics service with the changes, click **Start Daemon**.

To view Polyspace Metrics, in the address bar of your web browser, enter:

`https://ServerName:WSPN`

- *`ServerName`* is the name or IP address of the Polyspace Metrics server.
- *`WSPN`* is the web server port number.

Change Web Server Port Number for Metrics Server

If you change or specify a non-default value for the web server port number of your Polyspace Bug Finder client, you must manually configure the same value for your Polyspace Metrics server.

1. Select **Metrics** > **Metrics and Remote Server Settings**.
2. In the Metrics and Remote Server Settings dialog box, select **Stop Daemon** to stop the Polyspace Metrics server daemon.
3. In *`metricsRootFolder`*`\tomcat\conf\server.xml`, edit the `port` attribute of the `Connector` element for your web server protocol. Here, *`metricsRootFolder`* is the name that you specified for **Folder where analysis data will be stored** when setting up Polyspace Metrics.

   - For HTTP:

     ```
     <Connector port="8080"/>
     ```
   - For HTTPS:

     ```
     <Connector port="8443" SSLEnabled="true" scheme="https"
     secure="true" clientAuth="false" sslProtocol="TLS"
     keystoreFile="<datadir>/.keystore" keystorePass="polyspace"/>
     ```
4. In the same file, edit the `port` attribute of the `Server` element for your web server protocol.

   ```
   <Server port="8005" shutdown="SHUTDOWN">
   ```

5 In the Metrics and Remote Server Settings dialog box, select **Start Daemon** to restart the server with the new port numbers.
6 On the Polyspace toolbar, select **Tools** > **Preferences**.
7 In the **Server Configuration** tab, change the **Web server port number** to match your new value for the `port` attribute in the `Connector` element.

See Also

Related Examples

- "View Results List in Polyspace Metrics"

Install Polyspace Plugins

Install Polyspace Plugin for Eclipse

This topic shows how to install or uninstall the Polyspace plugin for Eclipse.

Install Polyspace Plugin for Eclipse IDE

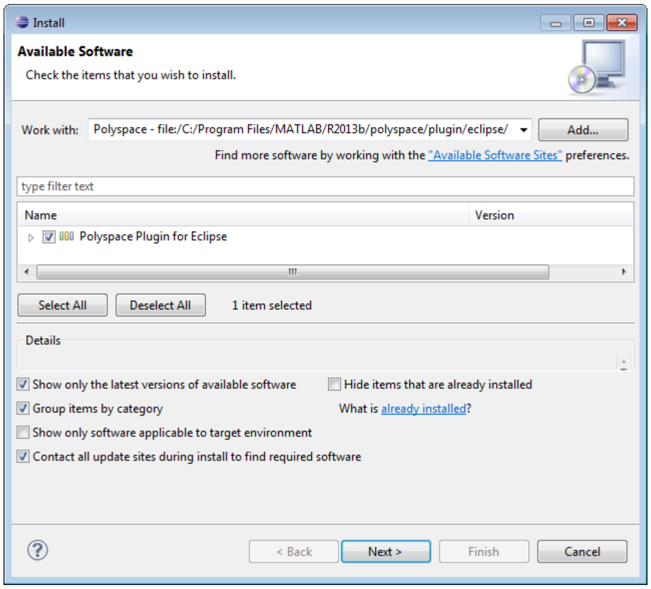
The Polyspace plugin is supported for Eclipse versions 4.3, 4.4, and 4.5. You can install the Polyspace plugin only after you:

- Install and set up Eclipse Integrated Development Environment (IDE). For more information, see the Eclipse documentation at www.eclipse.org.
- Install Java® 8 or newer. See Java documentation at www.java.com.
- Uninstall any previous Polyspace plugins. For more information, see “Uninstall Polyspace Plugin for Eclipse IDE” on page 5-4.

To install the Polyspace plugin:

1. From the Eclipse editor, select **Help > Install New Software**. The Install wizard opens, displaying the Available Software page.
2. Click **Add** to open the Add Repository dialog box.
3. In the **Name** field, specify a name for your Polyspace site, for example, `Polyspace_Eclipse_PlugIn`.
4. Click **Local**, to open the Browse for Folder dialog box.
5. Navigate to the *`MATLAB_Install`*`\polyspace\plugin\eclipse` folder. Then click **OK**.

   `MATLAB_Install` is the installation folder for the Polyspace product.
6. Click **OK** to close the Add Repository dialog box.
7. On the Available Software page, select `Polyspace Plugin for Eclipse`.



Install

Available Software

Check the items that you wish to install.

Work with: Polyspace - file:/C:/Program Files/MATLAB/R2013b/polyspace/plugin/eclipse/

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
✓ Polyspace Plugin for Eclipse	

1 item selected

Details

✓ Show only the latest versions of available software

☐ Hide items that are already installed

✓ Group items by category

What is already installed?

☐ Show only software applicable to target environment

✓ Contact all update sites during install to find required software

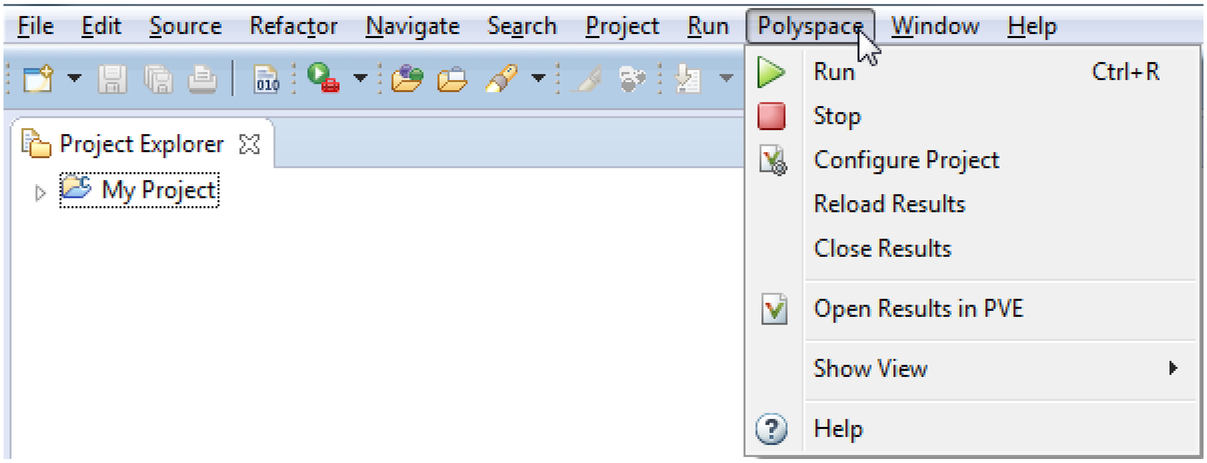
8 Click **Next**.

9 On the Install Details page, click **Next**.

10 On the Review Licenses page, review and accept the license agreement. Then click **Finish**.

Once you install the plugin, in the Eclipse editor, you’ll see:

- A **Polyspace** menu
- A **Polyspace Run - Bug Finder**, **Results List - Bug Finder**, and **Result Details** view.



Uninstall Polyspace Plugin for Eclipse IDE

Before installing a new Polyspace plugin, you must uninstall any previous Polyspace plugins:

1. In Eclipse, select **Help > About Eclipse**.
2. Select **Installation Details**.
3. Select the Polyspace plugin and select **Uninstall**.

   Follow the uninstall wizard to remove the Polyspace plugin. You must restart Eclipse for changes to take effect.

See Also

More About

- "Run Polyspace Analysis in Eclipse"

Install Polyspace Plugin for Simulink

By default, when you install Polyspace R2013b or later, the Simulink plugin is installed and connected to MATLAB.

If you model on a previous version of Simulink and MATLAB, you can also connect the Polyspace plugin on this previous version. That way you use the latest analysis software with your preferred version of Embedded Coder® or TargetLink. The Simulink plugin supports the four previous releases of MATLAB. For example, the R2017b version of the Polyspace plugin supports MATLAB versions R2015b through R2017b.

If you use a cross-version of Polyspace and MATLAB, local batch analyses can only be submitted from the Polyspace environment or using the `pslinkrun` command.

Note To install a newer version of Polyspace on MATLAB R2013b or later, you must install MATLAB without the corresponding version of Polyspace.

---

1. Using an account with read/write privileges, open the older version of MATLAB.
2. Use the `ver` command to make sure you do not have a previous version of Polyspace installed. See preceding note.
3. Change your **Current Folder** to

   `matlabroot``\toolbox\polyspace\pslink\pslink`

   `matlabroot` is the version of Polyspace you want to connect, for example, `C:\Program Files\MATLAB\R2017b`.
4. Connect the new version of Polyspace by running the command `pslinksetup('install')`.

See Also

Related Examples

- "Run Polyspace Analysis on Code Generated with Embedded Coder"

More About

- “Troubleshoot Navigation from Code to Model”

Polyspace Bug Finder and Polyspace Code Prover

Choose Between Polyspace Bug Finder and Polyspace Code Prover

Polyspace Bug Finder and Polyspace Code Prover detect run-time errors through static analysis. Though the products have a similar user interface and the mathematics underlying the analysis can sometimes be the same, the goals of the two products are different.

Bug Finder quickly analyzes your code and detects many types of defects. Code Prover checks *every* operation in your code for a set of possible run-time errors and tries to prove the absence of the error for all execution paths[1]. For instance, for *every* division in your code, a Code Prover analysis tries to prove that the denominator cannot be zero. Bug Finder does not perform such exhaustive verification. For instance, Bug Finder also checks for a division by zero error, but it might not find all operations that can cause the error.

The two products involve differences in setup, analysis and results review, because of this difference in objectives. In the following sections, we highlight the primary differences between a Bug Finder and a Code Prover analysis (also known as verification). Depending on your requirements, you can incorporate one or both kinds of analyses at appropriate points in your software development life cycle.

How Bug Finder and Code Prover Complement Each Other

1. For each operation in your code, Code Prover considers all execution paths leading to the operation that do not have a previous error. If an execution path contains an error prior to the operation, Code Prover does not consider it. See “Code Prover Analysis Following Red and Orange Checks” (Polyspace Code Prover).

- “Few False Positives with Bug Finder” on page 6-8
- “Zero False Negatives with Code Prover” on page 6-9

Overview

Use both Bug Finder and Code Prover regularly in your development process. The products provide a unique set of capabilities and complement each other. For possible ways to use the products together, see “Workflow Using Both Bug Finder and Code Prover” on page 6-9.

This table provides an overview of how the products complement each other. For details, see the sections below.

Feature	Bug Finder	Code Prover
Number of checkers	244	28 (Critical subset)
Depth of analysis	Fast. For instance: • Faster analysis. • Easier set up and review. • Fewer runs for clean code. • Results in real time.	Exhaustive. For instance: • All operations of a type checked for certain critical errors. • More rigorous data and control flow analysis.
Reporting criteria	Probable defects	Proven findings
Bug finding criteria	Few false positives	Zero false negatives

Faster Analysis with Bug Finder

How much faster the Bug Finder analysis is depends on the size of the application. The Bug Finder analysis time increases linearly with the size of the application. The Code Prover verification time increases at a rate faster than linear.

One possible workflow is to run Code Prover to analyze modules or libraries for robustness against certain errors and run Bug Finder at integration stage. Bug Finder analysis on large code bases can be completed in a much shorter time, and also find integration defects such as **Declaration mismatch** and **Data race**.

More Exhaustive Verification with Code Prover

Code Prover tries to prove the absence of:

- **Division by Zero** error on *every* division or modulus operation
- **Out of Bounds Array Index** error on *every* array access
- **Non-initialized Variable** error on *every* variable read
- **Overflow** error on *every* operation that can overflow

and so on.

For each operation:

- If Code Prover can prove the absence of the error for all execution paths, it highlights the operation in green.
- If Code Prover can prove the presence of a definite error for all execution paths, it highlights the operation in red.
- If Code Prover cannot prove the absence of an error or presence of a definite error, it highlights the operation in orange. This small percentage of orange checks indicate operations that you must review carefully, through visual inspection or testing. The orange checks often indicate hidden vulnerabilities. For instance, the operation might be safe in the current context but fail when reused in another context.

  You can use information provided in the Polyspace user interface to diagnose the checks. See “More Rigorous Data and Control Flow Analysis with Code Prover” on page 6-7. You can also provide contextual information to reduce unproven code even further, for instance, constrain input ranges externally.

Bug Finder does not aim for exhaustive analysis. It tries to detect as many bugs as possible and reduce false positives. For critical software components, running a bug finding tool is not sufficient because despite fixing all defects found in the analysis, you can still have errors during code execution (false negatives). After running Code Prover on your code and addressing the issues found, you can expect the quality of your code to be much higher. See “Zero False Negatives with Code Prover” on page 6-9.

More Specific Defect Types with Bug Finder

Code Prover checks for types of run-time errors where it is possible to mathematically prove the absence of the error. In addition to detecting errors whose absence can be mathematically proven, Bug Finder also detects other defects.

For instance, the statement `if(a=b)` is semantically correct according to the C language standard, but often indicates an unintended assignment. Bug Finder detects such unintended operations. Although Code Prover does not detect such unintended operations, it can detect if an unintended operation causes other run-time errors.

Examples of defects detected by Bug Finder but not by Code Prover include good practice defects, resource management defects, some programming defects, security defects, and defects in C++ object oriented design.

For more information, see:

- “Defects”: List of defects that Bug Finder can detect.
- “Run-Time Checks” (Polyspace Code Prover): List of run-time errors that Code Prover can detect.

Easier Setup Process with Bug Finder

Even if your code builds successfully in your compilation toolchain, it can fail in the compilation phase of a Code Prover verification. The strict compilation in Code Prover is related to its ability to prove the absence of certain run-time errors.

- Code Prover strictly follows the ANSI® C99 Standard, unless you explicitly use analysis options that emulate your compiler.

  To allow deviations from the ANSI C99 Standard, you must use the “Target and Compiler” options. If you create a Polyspace project from your build system, the options are automatically set.
- Code Prover does not allow linking errors that common compilers can permit.

  Though your compiler permits linking errors such as mismatch in function signature between compilation units, to avoid unexpected behavior at run time, you must fix the errors.

For more information, see “Troubleshoot Compilation and Linking Errors” (Polyspace Code Prover).

Bug Finder is less strict about certain compilation errors. Linking errors, such as mismatch in function signature between different compilation units, can stop a Code Prover verification but not a Bug Finder analysis. Therefore, you can run a Bug Finder analysis with less setup effort. In Bug Finder, linking errors are often reported as a defect after the analysis is complete.

Fewer Runs for Clean Code with Bug Finder

To guarantee absence of certain run-time errors, Code Prover follows strict rules once it detects a run-time error in an operation. Once a run-time error occurs, the state of your program is ill-defined and Code Prover cannot prove the absence of errors in subsequent code. Therefore:

- If Code Prover proves a definite error and displays a red check, it does not verify the remaining code in the same block.

  Exceptions include checks such as **Overflow**, where the analysis continues with the result of overflow either truncated or wrapped around.
- If Code Prover suspects the presence of an error and displays an orange check, it eliminates the path containing the error from consideration. For instance, if Code Prover detects a **Division by Zero** error in the operation `1/x`, in the subsequent operation on `x` in that block, `x` cannot be zero.
- If Code Prover detects that a code block is unreachable and displays a gray check, it does not detect errors in that block.

For more information, see “Code Prover Analysis Following Red and Orange Checks” (Polyspace Code Prover).

Therefore, once you fix red and gray checks and rerun verification, you can find more issues. You need to run verification several times and fix issues each time for completely clean code. The situation is similar to dynamic testing. In dynamic testing, once you fix a failure at a certain point in the code, you can uncover a new failure in subsequent code.

Bug Finder does not stop the entire analysis in a block after it finds a defect in that block. Even with Bug Finder, you might have to run analysis several times to obtain completely clean code. However, the number of runs required is fewer than Code Prover.

Results in Real Time with Bug Finder

Bug Finder shows some analysis results while the analysis is still running. You do not have to wait until the end of the analysis to review the results.

Code Prover shows results only after the end of the verification. Once Bug Finder finds a defect, it can display the defect. Code Prover has to prove the absence of errors on all execution paths. Therefore, it cannot display results during analysis.

More Rigorous Data and Control Flow Analysis with Code Prover

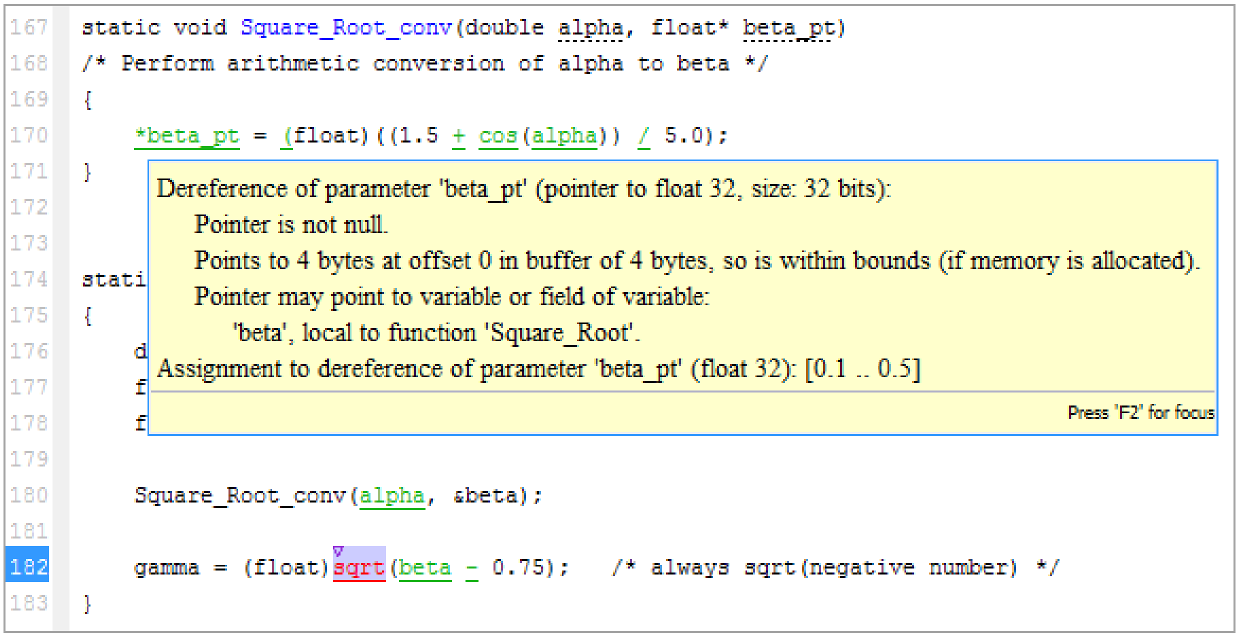
For each operation in your code, Code Prover provides:

- Tooltips showing the range of values of each variable in the operation.

  For a pointer, the tooltips show the variable that the pointer points to, along with the variable values.
- Graphical representation of the function call sequence that leads to the operation.

By using this range information and call graph, you can easily navigate the function call hierarchy and understand how a variable acquires values that lead to an error. For instance, for an **Out of Bounds Array Index** error, you can find where the index variable is first assigned values that lead to the error.

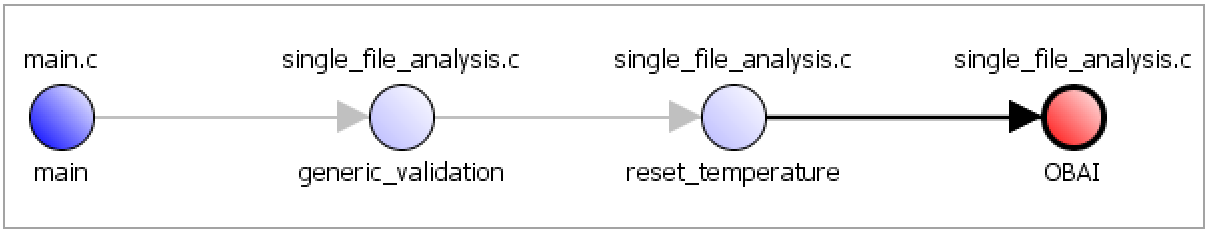
When reviewing a result in Bug Finder, you also have supporting information to understand the root cause of a defect. For instance, you have a traceback from where Bug Finder found a defect to its root cause. However, in Code Prover, you have more complete information, because the information helps you understand all execution paths in your code.



```
static void Square_Root_conv(double alpha, float* beta_pt)
/* Perform arithmetic conversion of alpha to beta */
{
    *beta_pt = (float)((1.5 + cos(alpha)) / 5.0);
}


stati
{
    d
    f
    f

    Square_Root_conv(alpha, &beta);

    gamma = (float)sqrt(beta - 0.75);   /* always sqrt(negative number) */
}
```

```
Dereference of parameter 'beta_pt' (pointer to float 32, size: 32 bits):
    Pointer is not null.
    Points to 4 bytes at offset 0 in buffer of 4 bytes, so is within bounds (if memory is allocated).
    Pointer may point to variable or field of variable:
        'beta', local to function 'Square_Root'.
Assignment to dereference of parameter 'beta_pt' (float 32): [0.1 .. 0.5]
```

Data Flow Analysis in Code Prover



Control Flow Analysis in Code Prover

Few False Positives with Bug Finder

Bug Finder aims for few false positives, that is, results that you are not likely to fix. By default, you are shown only the defects that are likely to be most meaningful for you.

Bug Finder also assigns an attribute called impact to the defect types based on the criticality of the defect and the rate of false positives. You can choose to analyze your code only for high-impact defects. You can also enable or disable a defect that you do not want to review[2].

Zero False Negatives with Code Prover

Code Prover aims for an exhaustive analysis. The software checks every operation that can trigger specific types of error. If a code operation is green, it means that the operation cannot cause those run-time errors that the software checked for[3]. In this way, the software aims for zero false negatives.

If the software cannot prove the absence of an error, it highlights the suspect operation in red or orange and requires you to review the operation.

Workflow Using Both Bug Finder and Code Prover

If you have both Bug Finder and Code Prover, based on the above differences, you can deploy the two products appropriately in your software development workflow. For instance:

- All developers in your organization can run Bug Finder on newly developed code. For maintaining standards across your organization, you can deploy a common configuration that looks only for specific defect types.

  Code Prover can be deployed as part of your unit testing suite.
- You can run Code Prover only on critical components of your project, while running Bug Finder on the entire project.
- You can run Code Prover on modules of code at the unit testing level, and run Bug Finder when integrating the modules.

  You can run Code Prover before unit testing. Code Prover exhaustively checks your code and tries to prove the presence or absence of errors. Instead of writing unit tests for your entire code, you can then write tests only for unproven code. Using Code Prover before unit testing reduces your testing efforts drastically.

Depending on the nature of your software development workflow and available resources, there are many other ways you can incorporate the two kinds of analysis. You can run both products on your desktop during development or as part of automated testing on a

2. You can also disable certain Code Prover defects related to non-initialization.
3. The Code Prover result holds only if you execute your code under the same conditions that you supplied to Code Prover through the analysis options.

remote server. Note that it is easier to interpret and fix bugs closer to development. You will benefit from using both products if you deploy them early and often in your development process.

There are two important considerations if you are running both Bug Finder and Code Prover on the same code.

- Both products can detect violations of coding rules such as MISRA C rules and JSF® C ++ rules.

  However, if you want to detect MISRA C:2012 coding rule violations alone, use Bug Finder. Bug Finder supports all the MISRA C:2012 coding rules. Code Prover does not support a few rules.
- If a result is found in both a Bug Finder and Code Prover analysis, you can comment on the Bug Finder result and import the comment to Code Prover.

  For instance, most coding rule checkers are common to Bug Finder and Code Prover. You can add comments to coding rule violations in Bug Finder and import the comments to the same violations in Code Prover. To import comments, open your result set and select **Tools > Import Comments**.
- You can use the same project for both Bug Finder and Code Prover analysis. The following set of options are common between Bug Finder and Code Prover:

  - “Target and Compiler”
  - “Macros”
  - “Environment Settings”
  - “Inputs and Stubbing”
  - “Multitasking”
  - “Coding Rules & Code Metrics”
  - “Reporting”, except `Bug Finder and Code Prover report (-report-template)`

You might have to change more of the default options when you run the Code Prover verification because Code Prover is stricter about compilation and linking errors.