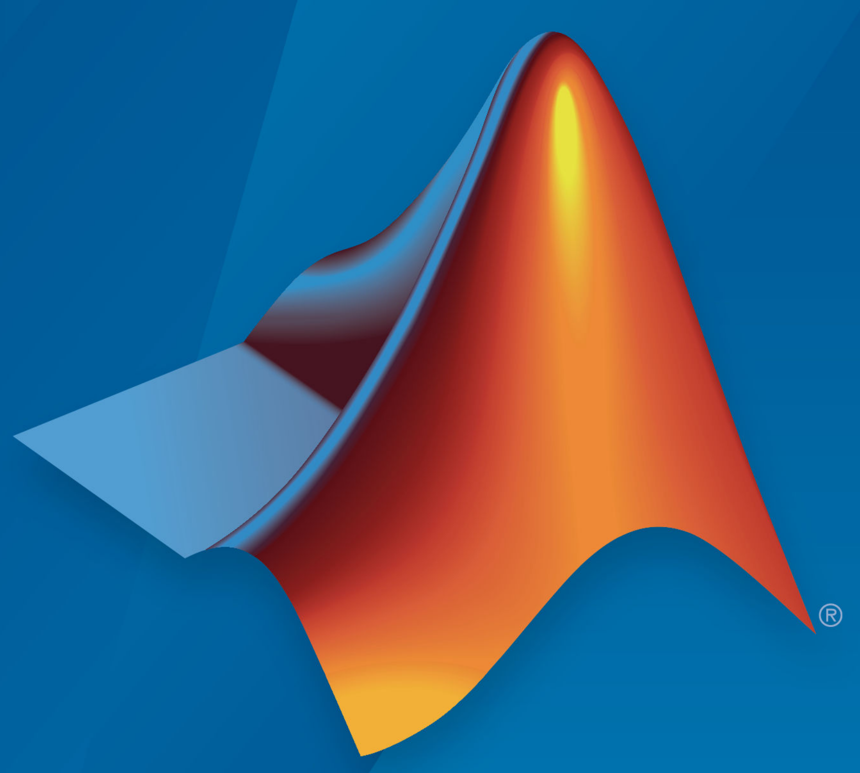
# Polyspace® Code Prover™

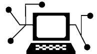
## Getting Started Guide



MATLAB® & SIMULINK®

R2018b

# 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® Code Prover™ Getting Started Guide*


© COPYRIGHT 2013–2018 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

**Trademarks**

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

**Patents**

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

**Revision History**


| | | |
|---|---|---|
| September 2013 | Online only | Revised for Version 9.0 (Release 2013b) |
| March 2014 | Online Only | Revised for Version 9.1 (Release 2014a) |
| October 2014 | Online Only | Revised for Version 9.2 (Release 2014b) |
| March 2015 | Online Only | Revised for Version 9.3 (Release 2015a) |
| September 2015 | Online Only | Revised for Version 9.4 (Release 2015b) |
| March 2016 | Online Only | Revised for Version 9.5 (Release 2016a) |
| September 2016 | Online Only | Revised for Version 9.6 (Release 2016b) |
| March 2017 | Online Only | Revised for Version 9.7 (Release 2017a) |
| September 2017 | Online Only | Revised for Version 9.8 (Release 2017b) |
| March 2018 | Online Only | Revised for Version 9.9 (Release 2018a) |
| September 2018 | Online Only | Revised for Version 9.10 (Release 2018b) |

# Contents

## 1 Introduction to Polyspace Code Prover


**Polyspace Code Prover Product Description** . . . . . . . . . . . . . . . **1-2**
Key Features . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1-2**

**Related Products** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1-3**
Polyspace Bug Finder . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1-3**
Polyspace Products for Verifying Ada Code . . . . . . . . . . . . . . . **1-3**
Tool Qualification and Certification . . . . . . . . . . . . . . . . . . . . **1-3**

**Polyspace Verification** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1-4**
Polyspace Verification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1-4**
Value of Polyspace Verification . . . . . . . . . . . . . . . . . . . . . . **1-4**
How Polyspace Verification Works . . . . . . . . . . . . . . . . . . . . . **1-6**


## 2 Get Started with Polyspace Code Prover


**Compiler Requirements** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-2**

**Run Polyspace Code Prover on C/C++ Code** . . . . . . . . . . . . . . . **2-3**
Run Polyspace in User Interface . . . . . . . . . . . . . . . . . . . . . . **2-3**
Run Polyspace on Windows or Linux Command Line . . . . . . . . **2-7**
Run Polyspace in Eclipse . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-8**
Run Polyspace in MATLAB . . . . . . . . . . . . . . . . . . . . . . . . . . **2-8**

**Review Polyspace Code Prover Analysis Results** . . . . . . . . . . . **2-11**
Interpret Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-11**
Address Results Through Bug Fix or Comments . . . . . . . . . . **2-13**
Manage Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-15**

## 3 Configure Server for Remote Verification and Polyspace Metrics

**Set Up Polyspace Analysis on Remote Server** . . . . . . . . . . . . . . **3-2**
Choose Between Local and Remote Analysis . . . . . . . . . . . . . **3-2**
Requirements for Remote Analysis . . . . . . . . . . . . . . . . . . . . **3-2**
Configure and Start Server . . . . . . . . . . . . . . . . . . . . . . . . . . **3-4**
Configure Client . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-8**
Set Up Server for Multiple Polyspace Releases . . . . . . . . . . . . **3-9**

**Set Up Polyspace Metrics** . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-10**
Requirements for Polyspace Metrics . . . . . . . . . . . . . . . . . . **3-10**
Configure and Start Polyspace Metrics Server . . . . . . . . . . . **3-11**
Configure Client Side . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-12**
Configure Web Server for HTTPS . . . . . . . . . . . . . . . . . . . . . **3-14**
Change Web Server Port Number for Metrics Server . . . . . . **3-15**

## 4 Install Polyspace Plugins

**Install Polyspace Plugin for Simulink** . . . . . . . . . . . . . . . . . . . **4-2**

**Install Polyspace Plugin for Eclipse** . . . . . . . . . . . . . . . . . . . . . **4-4**
Install Polyspace Plugin for Eclipse IDE . . . . . . . . . . . . . . . . **4-4**
Uninstall Polyspace Plugin for Eclipse IDE . . . . . . . . . . . . . . . **4-6**

## 5 Verify Code in IBM Rational Rhapsody Environment

**Verify Code in IBM Rational Rhapsody Environment** . . . . . . . . **5-2**
Code Verification Approach . . . . . . . . . . . . . . . . . . . . . . . . . . **5-2**
Adding Polyspace Profile to Model . . . . . . . . . . . . . . . . . . . . **5-3**
Accessing Polyspace Features . . . . . . . . . . . . . . . . . . . . . . . . **5-3**
Configuring Verification Options . . . . . . . . . . . . . . . . . . . . . . **5-6**
Running a Verification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-7**

Viewing Polyspace Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-7
Locating Faulty Code in Rhapsody Model . . . . . . . . . . . . . . . . 5-8
Template Configuration Files . . . . . . . . . . . . . . . . . . . . . . . . . 5-9

# 6 Polyspace Bug Finder and Polyspace Code Prover

**Choose Between Polyspace Bug Finder and Polyspace Code Prover** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **6-2**
How Bug Finder and Code Prover Complement Each Other . . **6-2**
Workflow Using Both Bug Finder and Code Prover . . . . . . . . . **6-8**

# Introduction to Polyspace Code Prover


- “Polyspace Code Prover Product Description” on page 1-2
- “Related Products” on page 1-3
- “Polyspace Verification” on page 1-4

# Polyspace Code Prover Product Description

**Prove the absence of run-time errors in software**

Polyspace Code Prover™ is a sound static analysis tool that proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in C and C++ source code. It produces results without requiring program execution, code instrumentation, or test cases. Polyspace Code Prover uses semantic analysis and abstract interpretation based on formal methods to verify software interprocedural, control, and data flow behavior. You can use it on handwritten code, generated code, or a combination of the two. Each operation is color-coded to indicate whether it is free of run-time errors, proven to fail, unreachable, or unproven.

Polyspace Code Prover also displays range information for variables and function return values, and can prove which variables exceed specified range limits. Results can be published to a dashboard to track quality metrics and ensure conformance with software quality objectives. Polyspace Code Prover can be integrated into build systems for automated verification.

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

## Key Features

- Proven absence of certain run-time errors in C and C++ code
- Color-coding of run-time errors directly in code
- Calculation of range information for variables and function return values
- Identification of variables that exceed specified range limits
- Quality metrics for tracking conformance with software quality objectives
- Web-based dashboard providing code metrics and quality status
- Guided review-checking process for classifying results and run-time error status
- Graphical display of variable reads and writes

# Related Products

| In this section... |
| --- |
| “Polyspace Bug Finder” on page 1-3 |
| “Polyspace Products for Verifying Ada Code” on page 1-3 |
| “Tool Qualification and Certification” on page 1-3 |

## Polyspace Bug Finder

For information about Polyspace Bug Finder™ , see `https://www.mathworks.com/products/polyspace-bug-finder/`.

## Polyspace Products for Verifying Ada Code

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

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

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

## Tool Qualification and Certification

You can use the DO Qualification Kit and IEC Certification Kit products to qualify Polyspace Products for C/C++ for DO and IEC Certification.

To view the artifacts available with these kits, use the Certification Artifacts Explorer. Artifacts included in the kits are not accessible from the MathWorks® web site.

For more information on the IEC Certification Kit, see IEC Certification Kit (for ISO 26262 and IEC 61508).

For more information on the DO Qualification Kit, see DO Qualification Kit (for DO-178).

# Polyspace Verification

**In this section...**

"Polyspace Verification" on page 1-4
"Value of Polyspace Verification" on page 1-4
"How Polyspace Verification Works" on page 1-6

## Polyspace Verification

Polyspace products verify C, C++, and Ada code by detecting run-time errors before code is compiled and executed.

To verify the source code, you set up verification parameters in a project, run the verification, and review the results. A graphical user interface helps you to efficiently review verification results. The software assigns a color to operations in the source code as follows:

- **Green** – Indicates that the operation is proven to not have certain kinds of error.
- **Red** – Indicates that the operation is proven to have at least one error.
- **Gray** – Indicates unreachable code.
- **Orange** – Indicates that the operation can have an error along some execution paths.

The color-coding helps you to quickly identify errors and find the exact location of an error in the source code. After you fix errors, you can easily run the verification again.

## Value of Polyspace Verification

Polyspace verification can help you to:

- "Enhance Software Reliability" on page 1-5
- "Decrease Development Time" on page 1-5
- "Improve the Development Process" on page 1-6

### Enhance Software Reliability

Polyspace software enhances the reliability of your C/C++ applications by proving code correctness and identifying run-time errors. Using advanced verification techniques, Polyspace software performs an exhaustive verification of your source code.

Because Polyspace software verifies all executions of your code, it can identify code that:

- Never has an error
- Always has an error
- Is unreachable
- Might have an error

With this information, you know how much of your code does not contain run-time errors, and you can improve the reliability of your code by fixing errors.

You can also improve the quality of your code by using Polyspace verification software to check that your code complies with established coding standards, such as the MISRA C®, MISRA® C++ or JSF® C++ standards.[1]

### Decrease Development Time

Polyspace software reduces development time by automating the verification process and helping you to efficiently review verification results. You can use it at any point in the development process. However, using it during early coding phases allows you to find errors when it is less costly to fix them.

You use Polyspace software to verify source code before compile time. To verify the source code, you set up verification parameters in a project, run the verification, and review the results. This process takes significantly less time than using manual methods or using tools that require you to modify code or run test cases.

Color-coding of results helps you to quickly identify errors. You will spend less time debugging because you can see the exact location of an error in the source code. After you fix errors, you can easily run the verification again.

Polyspace verification software helps you to use your time effectively. Because you know the parts of your code that do not have errors, you can focus on the code with proven (red code) or potential errors (orange code).

1. MISRA and MISRA C are registered trademarks of MISRA Ltd., held on behalf of the MISRA Consortium.

Reviewing code that might have errors (orange code) can be time-consuming, but Polyspace software helps you with the review process. You can use filters to focus on certain types of errors or you can allow the software to identify the code that you should review.

### Improve the Development Process

Polyspace software makes it easy to share verification parameters and results, allowing the development team to work together to improve product reliability. Once verification parameters have been set up, developers can reuse them for other files in the same application.

Polyspace verification software supports code verification throughout the development process:

- An individual developer can find and fix run-time errors during the initial coding phase.
- Quality assurance engineers can check overall reliability of an application.
- Managers can monitor application reliability by generating reports from the verification results.

## How Polyspace Verification Works

Polyspace software uses *static verification* to prove the absence of run-time errors. Static verification derives the dynamic properties of a program without actually executing it. This differs significantly from other techniques, such as run-time debugging, in that the verification it provides is not based on a given test case or set of test cases. The dynamic properties obtained in the Polyspace verification are true for all executions of the software.

### What is Static Verification

Static verification is a broad term, and is applicable to any tool that derives dynamic properties of a program without executing the program. However, most static verification tools only verify the complexity of the software, in a search for constructs that may be potentially erroneous. Polyspace verification provides deep-level verification identifying almost all run-time errors and possible access conflicts with global shared data.

Polyspace verification works by approximating the software under verification, using representative approximations of software operations and data.

For example, consider the following code:

```
for (i=0 ; i<1000 ; ++i)
{    tab[i] = foo(i);
}
```

To check that the variable i never overflows the range of tab, a traditional approach would be to enumerate each possible value of i. One thousand checks would be required.

Using the static verification approach, the variable i is modelled by its domain variation. For instance, the model of i is that it belongs to the static interval [0..999]. (Depending on the complexity of the data, convex polyhedrons, integer lattices and more elaborate models are also used for this purpose).

By definition, an approximation leads to information loss. For instance, the information that i is incremented by one every cycle in the loop is lost. However, the important fact is that this information is not required to ensure that no range error will occur; it is only necessary to prove that the domain variation of i is smaller than the range of tab. Only one check is required to establish that — and hence the gain in efficiency compared to traditional approaches.

Static code verification has an exact solution. However, this exact solution is not practical, as it would require the enumeration of all possible test cases. As a result, approximation is required for a usable tool.

### Exhaustiveness

Nothing is lost in terms of exhaustiveness. The reason is that Polyspace verification works by performing upper approximations. In other words, the computed variation domain of a program variable is a superset of its actual variation domain. As a result, Polyspace verifies run-time error items that require checking.

# Get Started with Polyspace Code Prover


- “Compiler Requirements” on page 2-2
- “Run Polyspace Code Prover on C/C++ Code” on page 2-3
- “Review Polyspace Code Prover Analysis Results” on page 2-11

# 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 Code Prover on C/C++ Code

Polyspace Code Prover is a sound static analysis tool that proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in C and C++ source code. A Code Prover analysis produces results without requiring program execution, code instrumentation, or test cases. Code Prover uses semantic analysis and abstract interpretation based on formal methods to determine control flow and data flow in the code. You can use Code Prover on handwritten code, generated code, or a combination of the two. In the analysis results, each operation is color-coded to indicate whether it is free of run-time errors, proven to fail, unreachable, or unproven.

You can run Code Prover 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 2-7
- "Run Polyspace in Eclipse" on page 2-8
- "Run Polyspace in MATLAB" on page 2-8


> To follow the steps in this tutorial, copy the files `example.c` and `include.h` from *`matlabroot`*`\polyspace\examples\cxx\Code_Prover_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 Code Prover app.

### Add Source Files

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

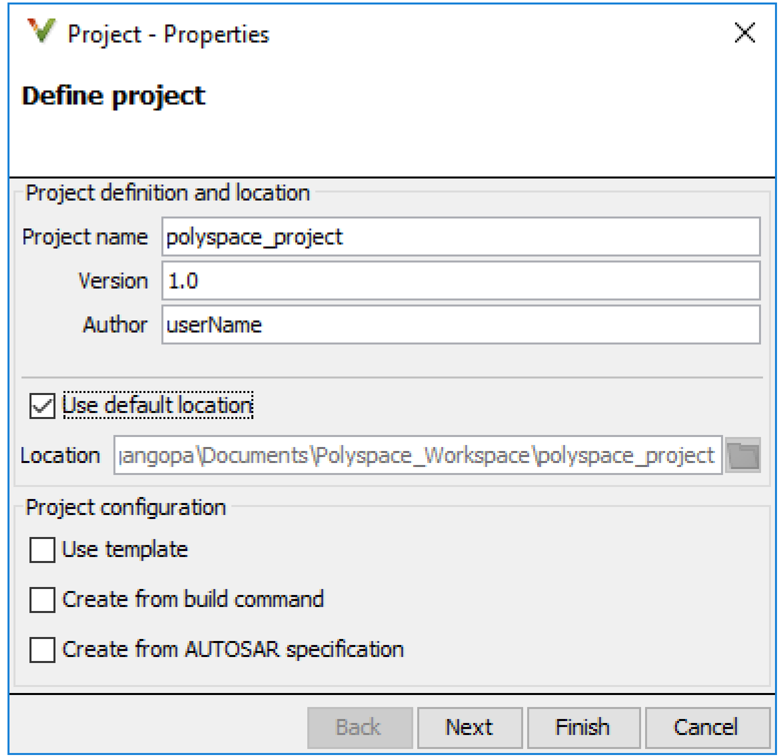
On the left of the **Start Page** pane, click **Start a new project**. Alternatively, select **File > New Project**.

After you provide a project name, on the next screens:

- Add your source folder.

  In this tutorial, add the path to the folder in which you saved the file `example.c`. Click **Next**.
- Add your include folder.

  In this tutorial, add the path to the folder in which you saved the file `include.h`. This folder can be the same as the previous folder. Click **Finish**.

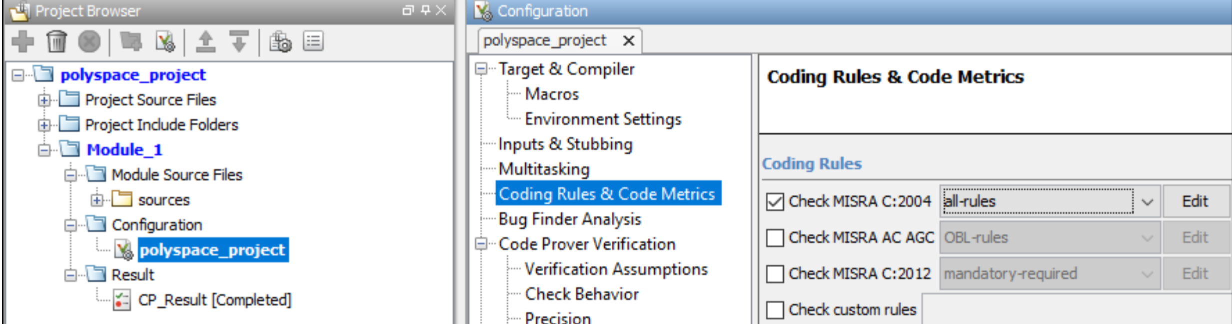



After you finish adding your source and include folders, you see a new project on the **Project Browser** pane. Your source folders are copied to the first module in the project. You can right-click a project to add more folders later. If you add folders later, you must explicitly copy them to a module.

#### 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**.




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

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

Follow the progress of verification on the **Output Summary** window. After the verification, 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 Code Prover from the Windows® or Linux® command line with batch (`.bat`) files or shell (`.sh`) scripts.

Use the `polyspace-code-prover-nodesktop` command to run a verification.

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-code-prover-nodesktop -sources example.c -I . -results-dir . -main-generator
```

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 Code Prover results will be saved.
- `Verify module or library (-main-generator)`: Specify that a `main` function must be generated if not found in the source files

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

```
polyspace ps_results.pscp
```

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-code-prover-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 4-4, you can run Polyspace directly on the files in your Eclipse projects.

In the **Project Explorer** pane in Eclipse, select your project. To use Code Prover for the analysis, select **Polyspace > Code Prover**. 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 Polyspace, 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 `example.c` in *`matlabroot`*`\polyspace\examples\cxx\Code_Prover_Examples\sources`, enter the following at the MATLAB command prompt.

```
proj = polyspace.Project

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


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

% Read results
cpSummary = proj.Results.getSummary('runtime');
cpResults = proj.Results.getResults('readable');
```

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

```
resultsFile = fullfile(proj.Configuration.ResultsDir,'ps_results.pscp');
polyspaceCodeProver(resultsFile)
```

### Additional Information

See:

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

## See Also

### Related Examples

- “Review Polyspace Code Prover Analysis Results” on page 2-11
- “Run Polyspace Analysis on Code Generated with Embedded Coder”

# Review Polyspace Code Prover Analysis Results

Polyspace Code Prover checks C/C++ code exhaustively and proves the absence of certain types of run-time errors (static analysis or verification). Whatever means you use for running the analysis, afterwards, 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 Code Prover on C/C++ Code" on page 2-3. Alternatively, in the Polyspace user interface, open example results using **Help** > **Examples** > **Code_Prover_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**.

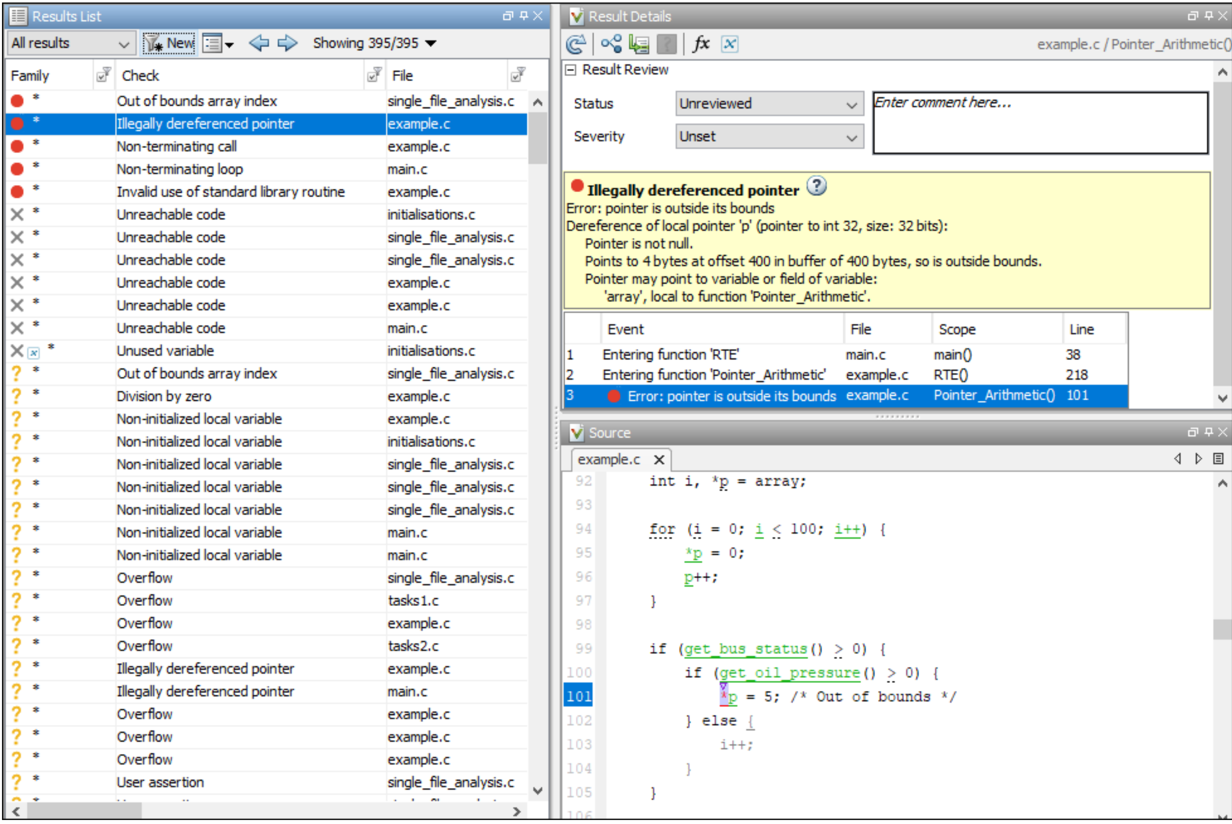
## Interpret Results

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 **Family** column header to sort the results based on how critical they are. Select the red **Illegally dereferenced pointer** check in the file `example.c`. A red check indicates that the error happens on all execution paths considered in the analysis.




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

For the **Illegally dereferenced pointer** result, the message on the **Result Details** pane indicates that the pointer p has an allowed buffer of 400 bytes. It points to a location that begins at 400 bytes from the beginning of the buffer and points to a data type of 4 bytes.

To investigate further and find the root cause of the issue, right-click the variable p on the **Source** pane and select **Search For All References**. Click each search result to navigate to the corresponding location on the source code. At each location, place your cursor on the variable p to see a tooltip that describes the variable value at that point in the code.

```
for (i = 0; i < 100; i++) {
    *p = 0;
    p+
}

if (ge
    if

    }
```

Local pointer 'p' (pointer to int 32, size: 32 bits):
Pointer is not null.
Points to 4 bytes at offset multiple of 4 in [0 .. 396] in buffer of 400 bytes, so is within bounds (if memory is allocated).
Pointer may point to variable or field of variable:
'array', local to function 'Pointer_Arithmetic'.
Press 'F2' for focus

You see that the pointer variable `p` is initialized to a 100-element `int` array. The pointer traverses the array in a `for` loop with 100 iterations and points to the end of the array. On the line with the red **Illegally dereferenced pointer** check, this pointer is dereferenced, resulting in dereference of a memory location outside the array.

#### Additional Information

See:

- “Interpret Polyspace Code Prover Results”
- “Code Prover Result and Source Code Colors”
- “Polyspace Code Prover Results”

## Address Results Through Bug Fix or Comments

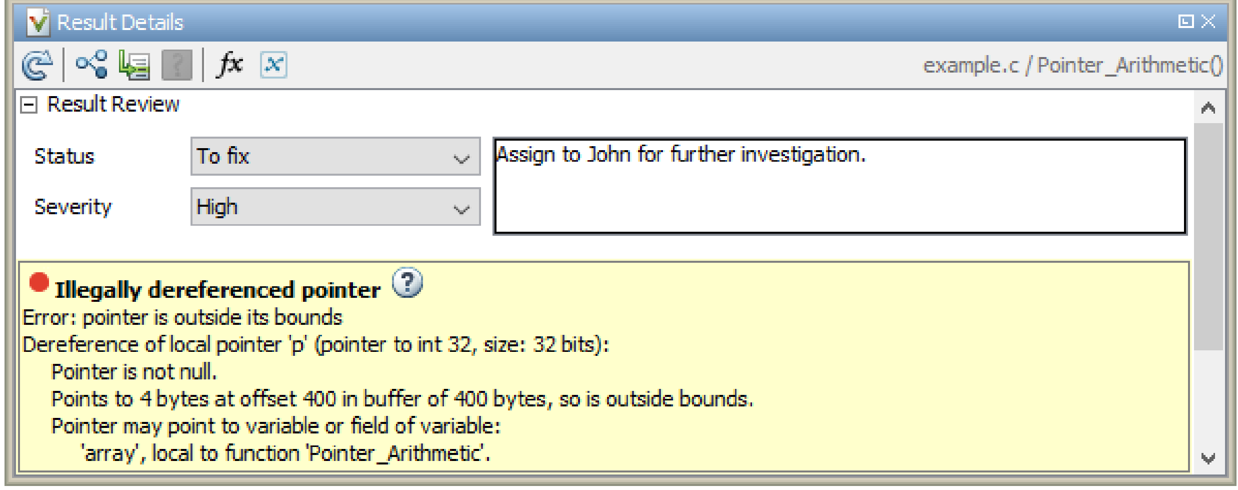
Once you understand the root cause of a Polyspace finding, you can fix your code. Otherwise, add comments to your Polyspace results to fix the code later or to justify the result. You can use the comments to keep track of your review progress.

Right-click the variable `p` on the **Source** pane. Select **Open Editor**. The code opens in a text editor. Fix the issue. For instance, you can make the pointer point to the beginning of the array after the `for` loop. Changes to the code are highlighted below.

```
...
int i, *p = array;

for (i = 0; i < 100; i++) {
    *p = 0;
     p++;
}

p = array;

if (get_bus_status() > 0)
...
```

If you rerun the analysis, you do not see the red **Illegally dereferenced pointer** check.

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.




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"

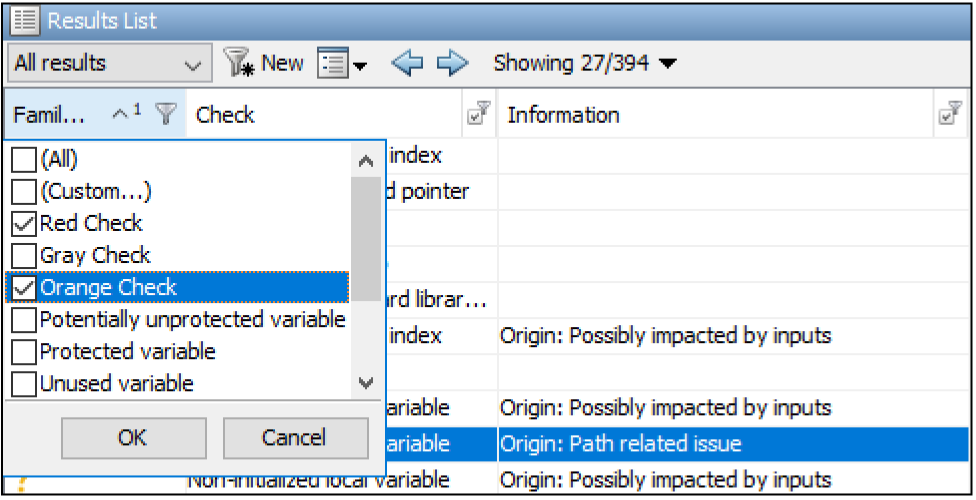
## Manage Results

When you open the results of a Code Prover analysis, you see a list of run-time checks, 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 red and critical orange checks.

  Click the **Family** column header to sort checks by color. Alternatively, you can filter out results other than red and orange checks. To begin filtering, click the icon on the column header.



You can review only the path-related orange checks because they are likely to be more critical. To filter out other checks, use the filters on the **Information** column. Clear the **All** filter and then select the filter **Origin: Path related issue**.

- 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”
- “Prioritize Check Review”

# 3

# Configure Server for Remote Verification and Polyspace Metrics


- "Set Up Polyspace Analysis on Remote Server" on page 3-2
- "Set Up Polyspace Metrics" on page 3-10

# 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

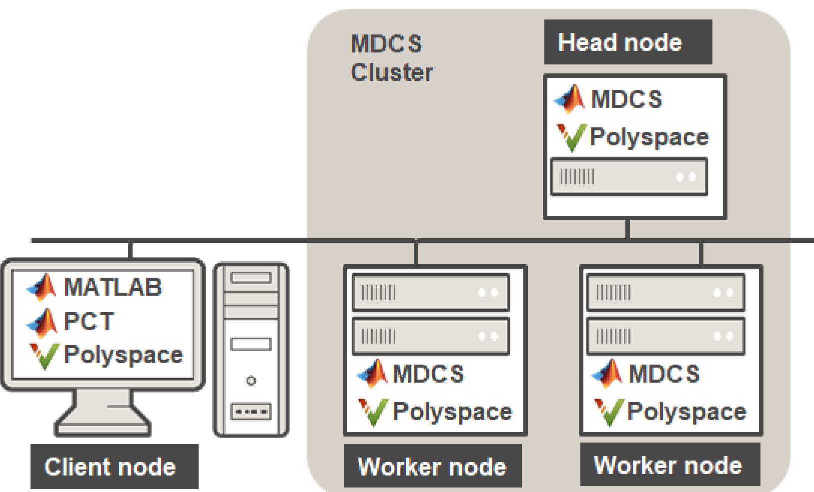
To determine when to use local or remote analysis, use the rules listed in this table.

| 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.

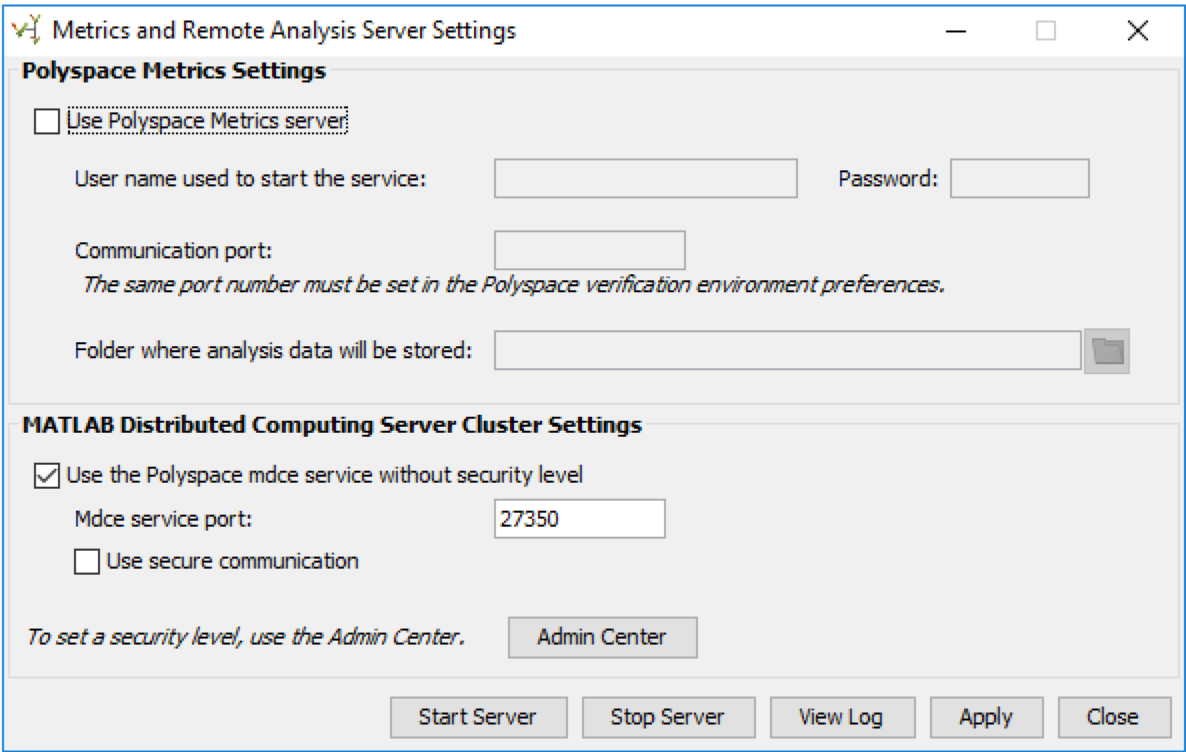
| Location | Requirements |
| --- | --- |
| Client node | • MATLAB<br>• Parallel Computing Toolbox™<br>• Polyspace Bug Finder or Polyspace Code Prover (whichever product you choose to run) |
| Head node and worker nodes (server side) | • MATLAB Distributed Computing Server™<br>• Polyspace Bug Finder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



#### 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 3-10.

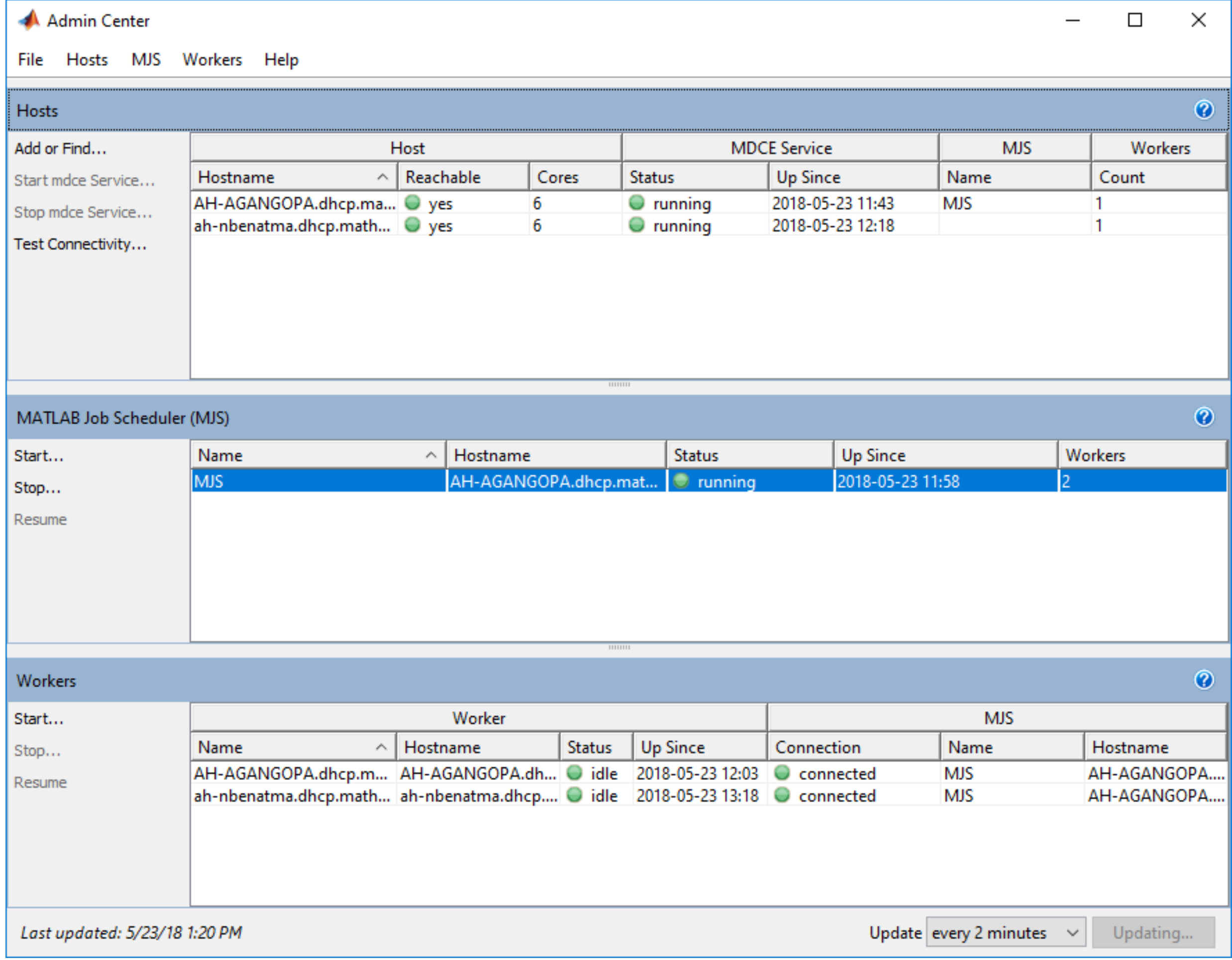
### Configure Cluster with Multiple Workers

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**.




**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” on page 3-6.

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 3-10
- “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<br>• Polyspace Bug Finder or Polyspace Code Prover |
| Polyspace Metrics service | Network server or head node of MATLAB Distributed Computing Server cluster | • MATLAB<br>• Polyspace Bug Finder or Polyspace Code Prover<br><br>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<br>• Polyspace Bug Finder or Polyspace Code Prover<br>• 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 3-12.
   - **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 3-2.

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 3-14.

   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 3-11.

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 Code Prover 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 3-11 and “Configure Client Side” on page 3-12.

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 Code Prover 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

- "Generate Code Quality Metrics"

# Install Polyspace Plugins

# 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”

# 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:

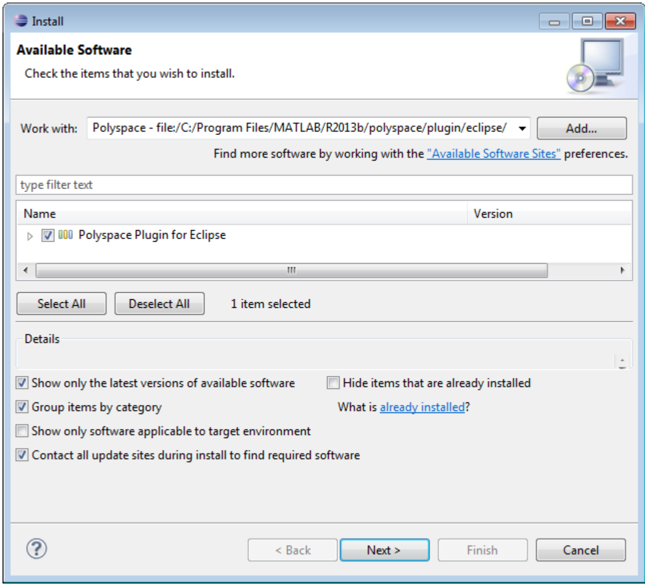
- Install and set up Eclipse Integrated Development Environment (IDE). For more information, see the Eclipse documentation at www.eclipse.org.
- Install Java® 8 or newer. See Java documentation at www.java.com.

  If you run into issues because of incompatible Java versions, see “Eclipse Java Version Incompatible with Polyspace Plug-in”.
- Uninstall any previous Polyspace plugins. For more information, see “Uninstall Polyspace Plugin for Eclipse IDE” on page 4-6.

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`.



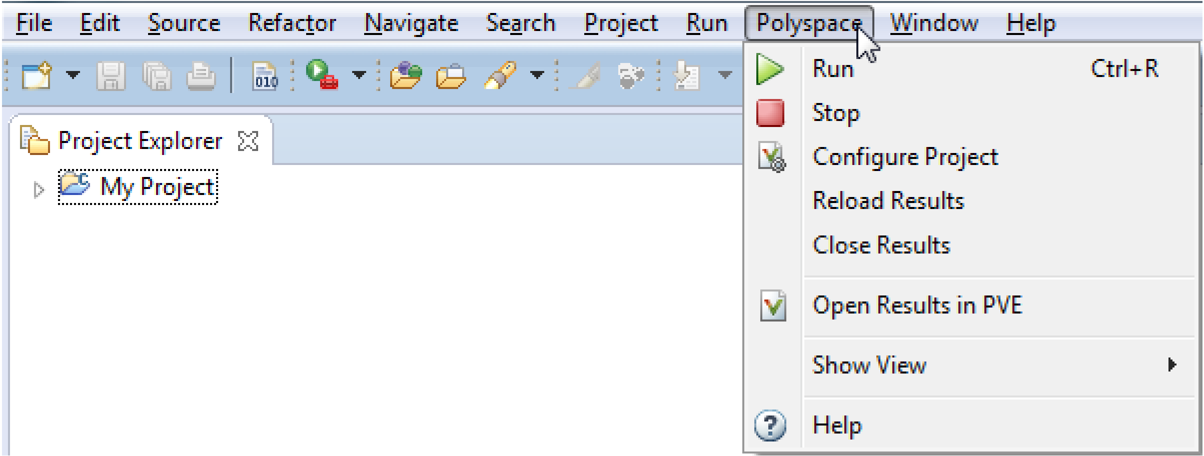

**8** Click **Next**.

**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 - Code Prover**, **Results List - Code Prover**, 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"

5

# Verify Code in IBM Rational Rhapsody Environment

# Verify Code in IBM Rational Rhapsody Environment

**In this section...**


"Code Verification Approach" on page 5-2
"Adding Polyspace Profile to Model" on page 5-3
"Accessing Polyspace Features" on page 5-3
"Configuring Verification Options" on page 5-6
"Running a Verification" on page 5-7
"Viewing Polyspace Results" on page 5-7
"Locating Faulty Code in Rhapsody Model" on page 5-8
"Template Configuration Files" on page 5-9


**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 Verification 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 Code Prover, you can verify C, C++ and Ada code that you generate from your IBM Rational® Rhapsody® model (up to version 8.0 supported). As a result, you can detect run-time errors 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 Code Prover 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 5-3.

- If required, specify Polyspace configuration options in the Polyspace verification environment. See "Configuring Verification Options" on page 5-6.
- Specify the `include` path to your operating system (environment) header files and run verification. See "Running a Verification" on page 5-7.
- View results, analyze errors, and locate faulty code within model. See "Viewing Polyspace Results" on page 5-7 and "Locating Faulty Code in Rhapsody Model" on page 5-8.

## 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, `polyspaceCodeProver`.

---

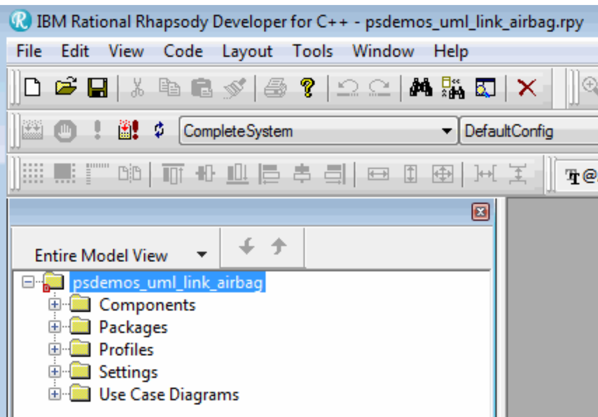
1. In the Rhapsody editor, select **File > Add Profile to Model**. The Add Profile to Model dialog box opens.
2. Navigate to the folder *MATLAB_Install*`\polyspace\plugin\rhapsody\profiles\Polyspace`.
3. Select the file `Polyspace.sbs`. Then click **Open**.

Now, if you right-click a package or file, you see the **Polyspace** item in the context menu. Selecting **Polyspace** opens the Polyspace Verification dialog box.

## Accessing Polyspace Features

To access Polyspace features in the Rhapsody editor:

1. Open the model that you want to verify. For example, `psdemos_uml_link_airbag.rpy` in *MATLAB_Install*`/polyspace/plugin/rhapsody/psdemos`.

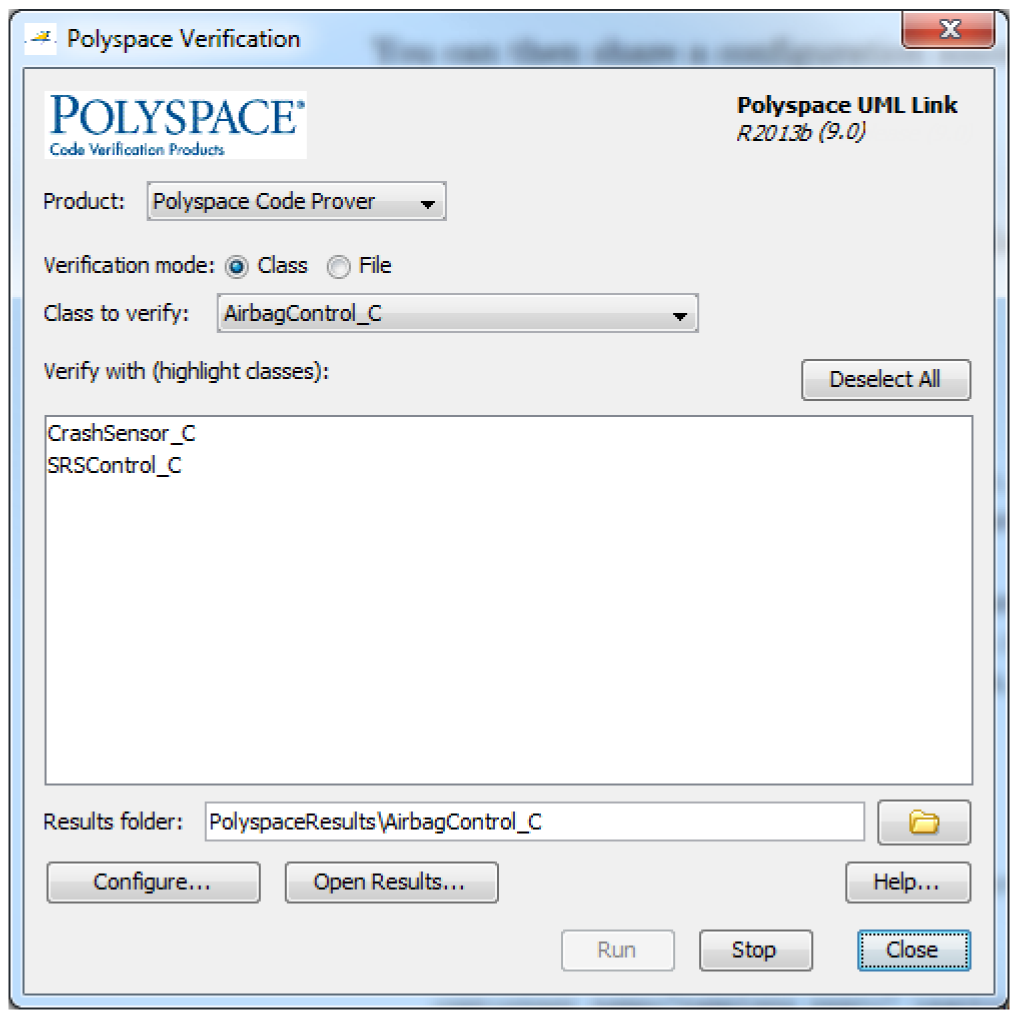



2 In the **Entire Model View**, expand the `Packages` node.

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

4 From the context menu, select **Polyspace**.

The Polyspace Verification dialog box opens.




Through the Polyspace Verification dialog box, you can:

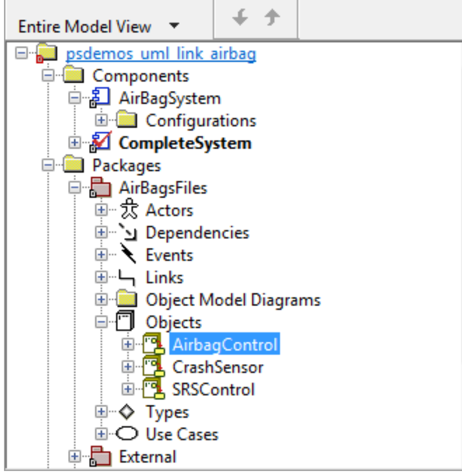
- Specify verification options. See "Configuring Verification Options" on page 5-6.
- Start a verification. See "Running a Verification" on page 5-7.
- Stop a local verification. See "Running a Verification" on page 5-7.
- View verification results. See "Viewing Polyspace Results" on page 5-7.

- Open help.
- Open the Polyspace Job Monitor. See “Running a Verification” on page 5-7.

## Configuring Verification Options

To specify options for your verification:

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




2. From the context menu, select **Polyspace**.
3. In the Polyspace Verification dialog box, click **Configure**. The **Configuration** pane of the Polyspace verification environment opens.
4. Select options for your verification. In particular, you must specify the following:
   - **Target & Compiler** > **Compiler** (`-compiler`)
   - **Target & Compiler** > **Environment Settings** > **Include** (`-include`) — Path to your operating system (environment) header files.
   - **Distributed Computing** > **Batch** (`-include`) — For local verification, clear the check box. For remote verification, select the check box.
5. To save your options, on the toolbar, click .

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

## Running a Verification

Before starting a verification, make sure that the generated code for the model is up to date.

To start a verification:

1. In the Rhapsody editor, select **Tools** > **Polyspace**. The Polyspace Verification dialog box opens.
2. In the **Results folder** field, specify a location for your verification results.
3. Select the **Verification mode**. Click **Class** or **File**. If you click **Class**, from the **Class to verify** drop-down list, select a specific class. In addition, under **Verify with (highlight classes)**, you can select other classes from the displayed list.
4. If you want to run the analysis on your Polyspace server, select **Send to Polyspace server**.

   ---
   **Note** If you are performing local batch verification with Polyspace for Rhapsody, MATLAB Distributed Computing Server, and Parallel Computing Toolbox, you can only submit local batch analyses from the Polyspace environment or using the command.

   ---
5. Click **Run**. In the **Log** view of the Rhapsody editor, you see verification messages.

If your verification is local, you can observe progress in the **Log** view of the Rhapsody editor. To stop the local verification, in the Polyspace Verification dialog box, click **Stop**.

To stop or monitor a batch verification, use the Job Monitor.

## Viewing Polyspace Results

To view results from the last local verification:

1. In the Rhapsody editor, select **Tools** > **Polyspace**.
2. In the Polyspace Verification dialog box, click **Open Results**.

   The software displays results in the Polyspace user interface.

To view results from remote verifications, use Polyspace Metrics or the Job Monitor.

For more information, see “Review Analysis 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 Polyspace verification results:

1. In your verification results, navigate to an error.
2. In the Source pane, right-click the error. From the context menu, select **Back To Model**.

   **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.

   The 64-bit version of the Polyspace product supports the **Back To Model** command only for version 8.0 of the IBM Rational Rhapsody product. For other versions, use the 32-bit Polyspace version.

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


- "Using Template Configuration Files" on page 5-9
- "Default Configuration Options" on page 5-9


### Using Template Configuration Files

The first time you perform a verification, the software copies a template, Polyspace configuration file, from `matlabroot/polyspace/plugin/rhapsody/etc/template_language.psprj` to the project folder. The software also renames the copy `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 verification 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.

### Default Configuration Options

The `template_language.psprj` XML files specify the default option values for code verification.

The file `template_C.psprj` is:

```
<?xml version="1.0" encoding="UTF-8"?>
<polyspace_project name="template_psprj" language="C" author="polyspace"
version="1.0" date="08/04/2011" path="file:/C:/Polyspace/Polyspace_Common
/Rhapsody/PolyspaceUMLLink/etc/template_C.psprj">
  <source>
  </source>
```

```
  <include>
  </include>
  <module name="Verification_1" isactive="true">
    <source>
    </source>
    <optionset name="template_psprj" isactive="true">
      <option flagname="-respect-types-in-fields">true</option>
      <option flagname="-respect-types-in-globals">true</option>
    </optionset>
  </module>
</polyspace_project>
```

The file `template_C++.psprj` is:

```
<?xml version="1.0" encoding="UTF-8"?>
<polyspace_project name="template_psprj" language="C++" author="polyspace"
version="1.0" date="08/04/2011" path="file:/C:/Polyspace/Polyspace_Common
/Rhapsody/PolyspaceUMLLink/etc/template_C++.psprj">
  <source>
  </source>
  <include>
  </include>
  <module name="Verification_1" isactive="true">
    <source>
    </source>
    <optionset name="template_psprj" isactive="true">
      <option flagname="-D">[OM_NO_FRAMEWORK_MEMORY_MANAGER]</option>
      <option flagname="-dialect">gnu</option>
      <option flagname="-respect-types-in-fields">true</option>
      <option flagname="-respect-types-in-globals">true</option>
      <option flagname="-target">i386</option>
    </optionset>
  </module>
</polyspace_project>
```

6

# 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[2]. 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


- “Overview” on page 6-3
- “Faster Analysis with Bug Finder” on page 6-3
- “More Exhaustive Verification with Code Prover” on page 6-3
- “More Specific Defect Types with Bug Finder” on page 6-4
- “Easier Setup Process with Bug Finder” on page 6-5
- “Fewer Runs for Clean Code with Bug Finder” on page 6-5
- “Results in Real Time with Bug Finder” on page 6-6
- “More Rigorous Data and Control Flow Analysis with Code Prover” on page 6-6
- “Few False Positives with Bug Finder” on page 6-8


2. 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”.

- “Zero False Negatives with Code Prover” on page 6-8

### 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 6-8.

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.<br>For instance:<br>• Faster analysis.<br>• Easier set up and review.<br>• Fewer runs for clean code.<br>• Results in real time. | Exhaustive.<br>For instance:<br>• All operations of a type checked for certain critical errors.<br>• 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 6-6. 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 6-8.

### 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 (Polyspace Bug Finder), resource management defects (Polyspace Bug Finder), some programming defects (Polyspace Bug Finder), security defects (Polyspace Bug Finder), and defects in C++ object oriented design (Polyspace Bug Finder).

For more information, see:

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

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”.

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.

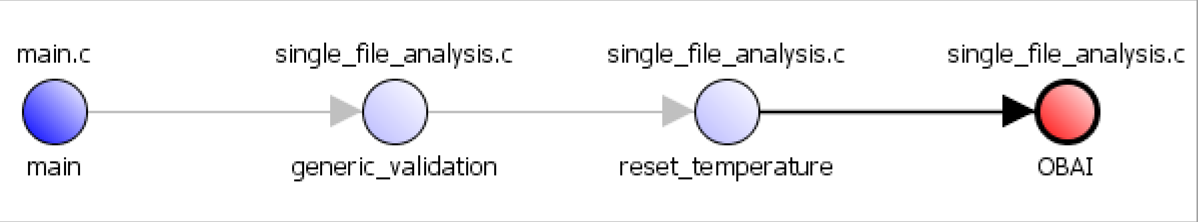
By using this range information and call graph, you can easily navigate the function call hierarchy and understand how a variable acquires values that lead to an error. For instance, for an **Out of Bounds Array Index** error, you can find where the index variable is first assigned values that lead to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

```
167  static void Square_Root_conv(double alpha, float* beta_pt)
168  /* Perform arithmetic conversion of alpha to beta */
169  {
170      *beta_pt = (float)((1.5 + cos(alpha)) / 5.0);
171  }
172
173
174  stati
175  {
176      d
177      f
178      f
179
180      Square_Root_conv(alpha, &beta);
181
182      gamma = (float)sqrt(beta - 0.75);   /* always sqrt(negative number) */
183  }
```

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]
Press 'F2' for focus

**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[3].

#### 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[4]. 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:

3. You can also disable certain Code Prover defects related to non-initialization.
4. 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.

- 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 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.