

Polyspace[®] Bug Finder[™]

Getting Started Guide



MATLAB[®]&SIMULINK[®]

R2017b



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Polyspace® Bug Finder™ Getting Started Guide

© COPYRIGHT 2013–2017 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	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)

About Polyspace Bug Finder

1

Polyspace Bug Finder Product Description	1-2
Key Features	1-2
Related Products	1-3
Polyspace Code Prover	1-3
Polyspace Products for Ada	1-3
Bug Finder Workflows	1-4
Polyspace and the Software Development Cycle	1-5
Software Quality and Productivity	1-5
Best Practices for Verification Workflow	1-6
Getting Help	1-7
Access Documentation	1-7
Access Contextual Help	1-7
Quick Start Guide for Polyspace Bug Finder	1-9

Tutorials

2

Compiler Requirements	2-2
Find Defects from the Polyspace Environment	2-3
Introduction	2-3
Set Up Files and Open Polyspace	2-3
Set Up Project	2-4
Configure Options	2-7

Run Analysis	2-8
Review Results	2-9
Fix Defects and Rerun Analysis	2-12
Find Defects from Simulink	2-14
Introduction	2-14
Open Model and Generate Code	2-14
Set Polyspace Options and Run Analysis	2-15
Review Results	2-15
Fix Model and Rerun Analysis	2-17
Find Defects from the Eclipse Plugin	2-19
Introduction	2-19
Run Analysis and Review Results	2-19

Polyspace UML Link RH

3

Find Defects from IBM Rational Rhapsody	3-2
Code Analysis Approach	3-2
Adding Polyspace Profile to Model	3-3
Accessing Polyspace Features	3-5
Configuring Analysis Options	3-7
Running an Analysis	3-8
Monitoring an Analysis	3-10
Viewing Polyspace Results	3-11
Locating Faulty Code in Rhapsody Model	3-11
Template Configuration Files	3-13

Installation and Configuration

4

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

Set Up Polyspace Metrics	4-7
Requirements for Polyspace Metrics	4-7
Start Polyspace Metrics Server	4-8
Configure Polyspace Preference	4-9
Configure Web Server for HTTPS	4-10
Change Web Server Port Number for Metrics Server	4-12
Set Up Server for Metrics and Remote Analysis	4-13
Requirements for Remote Verification and Analysis	4-14
Start Server for Remote Verification and Polyspace Metrics	4-14
Configure Polyspace Preferences	4-16

Using Bug Finder and Code Prover

5

Differences Between Polyspace Bug Finder and Polyspace Code Prover Analysis	5-2
How Bug Finder and Code Prover Complement Each Other ..	5-2
Workflow Using Both Bug Finder and Code Prover	5-9

About Polyspace Bug Finder

- “Polyspace Bug Finder Product Description” on page 1-2
- “Related Products” on page 1-3
- “Bug Finder Workflows” on page 1-4
- “Polyspace and the Software Development Cycle” on page 1-5
- “Getting Help” on page 1-7
- “Quick Start Guide for Polyspace Bug Finder” on page 1-9

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

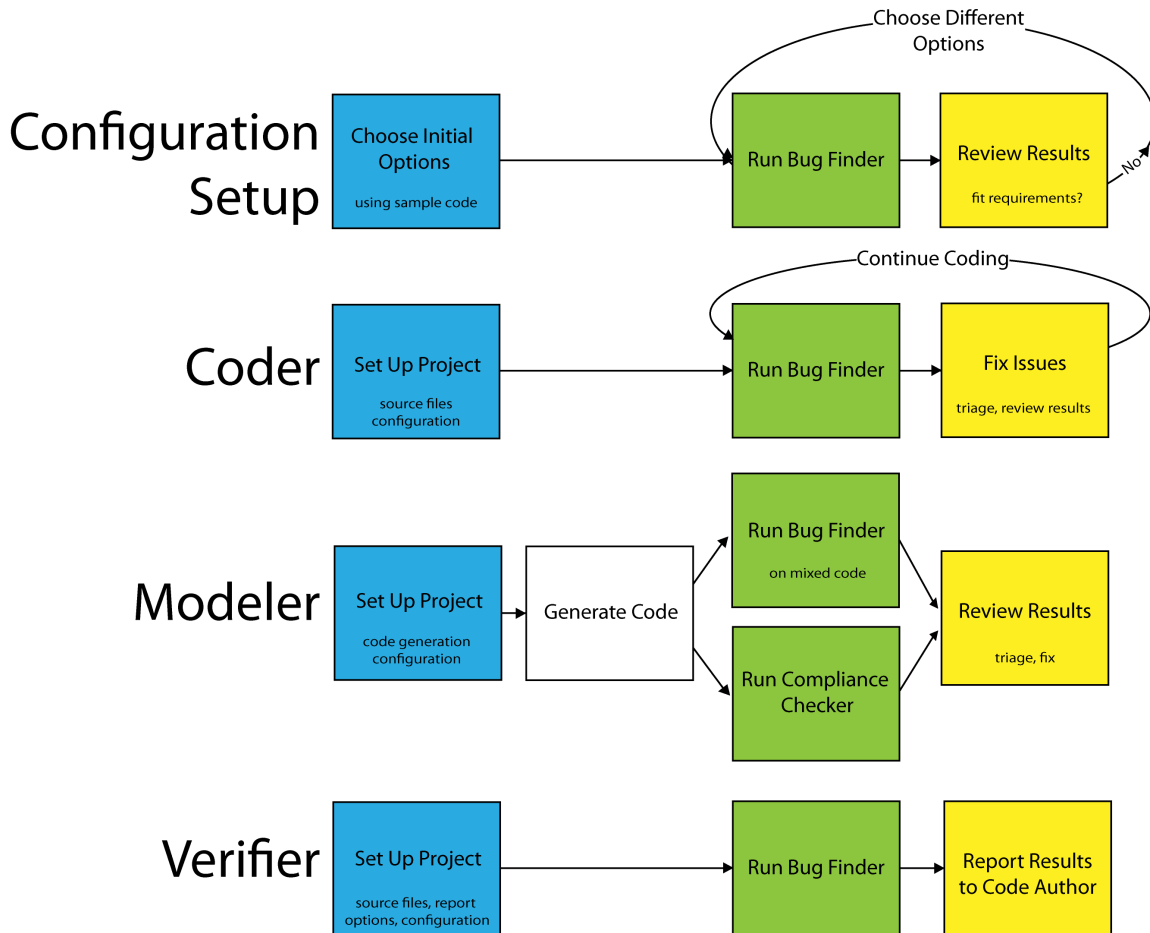
For information about Polyspace products that verify Ada code, see the following:

www.mathworks.com/products/polyspaceclientada/

www.mathworks.com/products/polyspaceserverada/

Bug Finder Workflows

This topic shows four different workflows for using the Polyspace Bug Finder product. Use Bug Finder regularly to help catch bugs and coding rule violations as you build your project.



Polyspace and the Software Development Cycle

In this section...

“Software Quality and Productivity” on page 1-5

“Best Practices for Verification Workflow” on page 1-6

Software Quality and Productivity

The goal of most software development teams is to maximize both quality and productivity. However, when developing software, there are always three related variables: cost, quality, and time.



Changing the requirements for one of these variables impacts the other two.

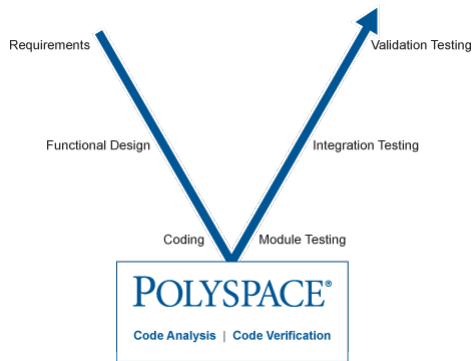
Generally, the criticality of your application determines the balance between these three variables – your quality model. With classical testing processes, development teams generally try to achieve their quality model by testing all modules in an application until each meets the required quality level. Unfortunately, this process often ends before quality requirements are met, because the available time or budget has been exhausted.

Polyspace analysis and verification allow a different process. Polyspace can support both productivity improvement and quality improvement at the same time, although there is always a balance between the aims of these activities.

To achieve maximum quality and productivity, however, you cannot simply perform code analysis or verification at the end of the development process. You must integrate both into your development process, in a way that respects time and cost restrictions.

Best Practices for Verification Workflow

Polyspace can be used throughout the software development cycle. However, to maximize both quality and productivity, the most efficient time to use it is early in the development cycle.



Polyspace Verification in the Development Cycle

Typically, verification is conducted in two stages. First, you verify code as it is written, to check coding rules and quickly identify any obvious defects. Once the code is stable, you verify it again before module/unit testing, with more stringent verification and review criteria.

Using verification early in the development cycle improves both quality and productivity, because it allows you to find and manage defects soon after the code is written. This saves time because each user is familiar with their own code, and can quickly determine why the code contains defects. In addition, defects are cheaper to fix at this stage, since they can be addressed before the code is integrated into a larger system.

Getting Help

In this section...

- “Access Documentation” on page 1-7
- “Access Contextual Help” on page 1-7

Polyspace provides documentation and contextual help in multiple locations to get you the help you need.

Access Documentation

The full documentation is available in the Polyspace interface and its plug-ins. To access the documentation:


- Polyspace interface — Select **Help > Documentation**.
- Simulink plug-in — Select **Code > Polyspace > Help**.
- Eclipse plug-in — Select **Polyspace > Help**.
- IBM® Rational® Rhapsody® plug-in — Right-click on a package. From the context menu, select **Polyspace**. In the Polyspace Verification dialog, select **Help**.

Access Contextual Help

To access contextual help for analysis options in the Polyspace interface or a Polyspace plug-in:

- 1 In the **Configuration** pane, hover your cursor over an analysis option.
- 2 In the tooltip, select **More Help**.
- 3 Look in the **Contextual Help** pane to see more help for that option.

To access contextual help for Polyspace results from the Polyspace interface:

- 1 In the **Results List** pane, select a Polyspace check.
- 2 In the **Result Details** pane, select .
- 3 Look in the **Contextual Help** pane to see more help for that check.

To access contextual help for Simulink configuration parameters, in the configuration window, right click on the parameter name and select **What's This**.

See Also

Related Examples

- “Configure Advanced Polyspace Analysis Options”
- Customize Analysis Options from Eclipse

Quick Start Guide for Polyspace Bug Finder

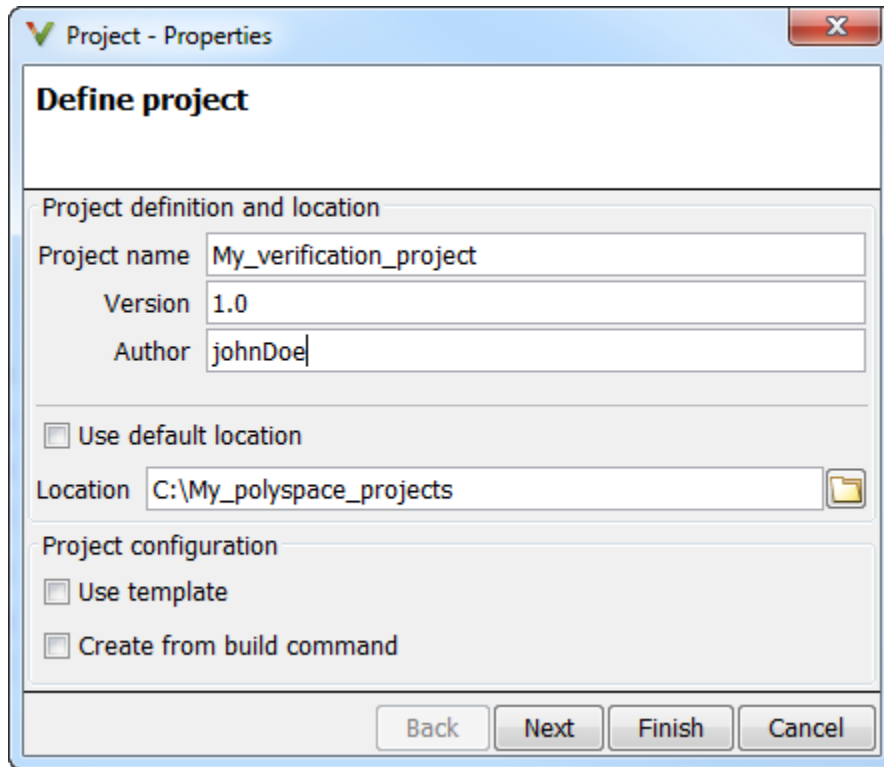
Polyspace® Bug Finder™ identifies run-time errors, concurrency issues, security vulnerabilities, and other defects in C and C++ embedded software. Polyspace Bug Finder also checks C/C++ code for compliance with coding rule standards such as MISRA C®, MISRA C++, JSF++, and custom naming conventions.

The following steps describe how to run Polyspace on your source code. If you want to skip the project setup and configuration steps:

- 1 Open the demo project. Select **Help > Examples > Bug_Finder_Example.psprj**. You see the *results* from a Polyspace run.
- 2 To see the demo *project*, select **Window > Reset Layout > Project Setup**.

Step 1: Set Up Project

In the Polyspace user interface, select **File > New Project**.



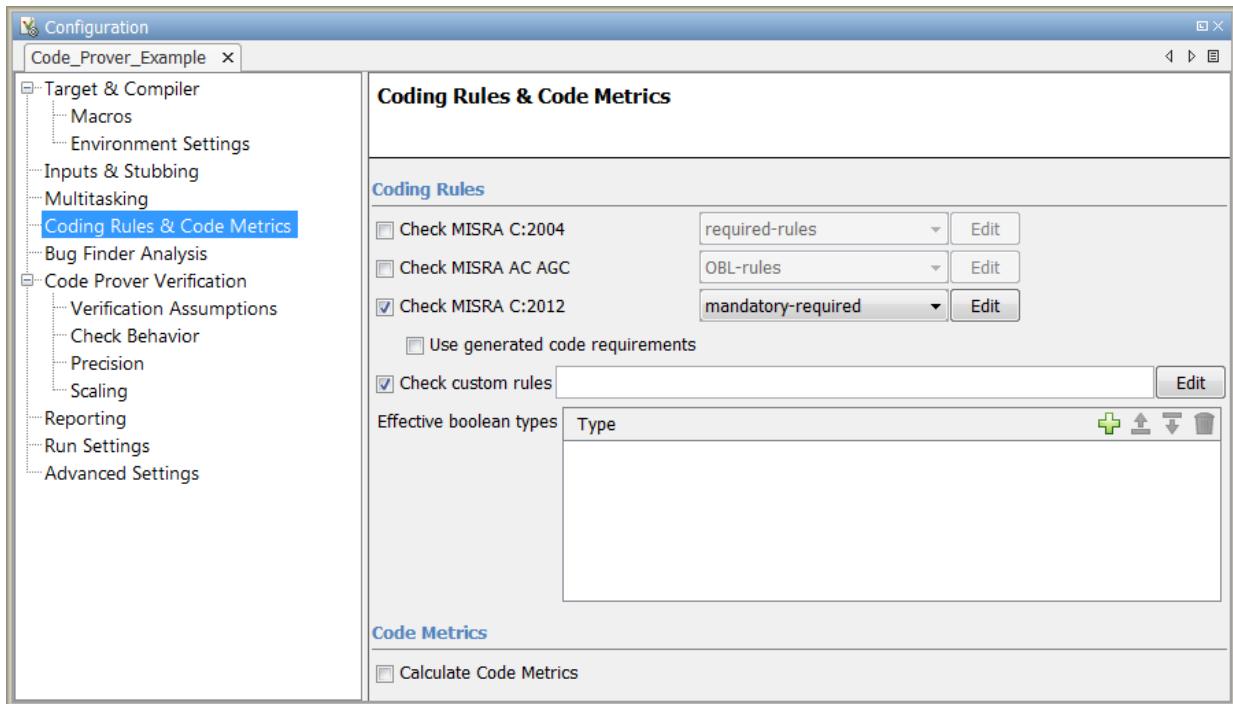
To add source code for analysis, do one of the following:

- Copy the example files from `MATLAB_Install\polyspace\examples\cxx\Bug_Finder_Example\sources` to a new folder. Change the read-only status of the files. Add the folder to your project as both source and include folder. `MATLAB_Install` is the location of your MATLAB installation such as `C:\Program Files\MATLAB\R2017a`.
- Add your own source code to the project.

Step 2: Configure Project and Run Verification

On the **Project Browser** pane, select the node below the **Configuration** node in your project.

The default analysis options appear on the **Configuration** pane. Retain the default options or change them to your requirements. For example, to check for coding rule violations, select **Coding Rules & Code Metrics** and specify your options. For more information on an option, place your cursor on the option and select **More Help**.



Click the **Run Bug Finder** button.

Step 3: Review Results

After verification, the results open on the **Results List** pane. From the grouping dropdown, select **None**. Select each result to view the source code location on the **Source** pane and further information about the result on the **Result Details** pane. For more information about a result, on the **Result Details** pane, click the question mark button.

The screenshot shows a window titled "Results List" with a toolbar containing a search icon, a "New" button, a list icon, navigation arrows, and a refresh icon. The status bar indicates "Showing 2,000/2,000". The table below lists the results:

F...	Type	Check	File	Function
○ *	Defect	Assertion	programming.c	bug_assert()
○ *	Defect	Invalid use of == operator	programming.c	bug_badequalequaluse()
○ *	Defect	Invalid free of pointer	dynamicmemor...	bug_badfree()
○ *	Defect	Missing unlock	concurrency.c	File Scope
○ *	Defect	Bad order of dropping privileges	security.c	bug_badprivilegedroporder()
○ *	Defect	Bad order of dropping privileges	security.c	bug_badprivilegedroporder()
○ *	Defect	Character value absorbed into ...	programming.c	bug_chareofconfused()
○ *	Defect	Use of previously closed resource	resourcemana...	bug_closedresourceuse_fprintf()
○ *	Defect	Writing to const qualified object	programming.c	bug_constantobjectwrite()
○ *	Defect	Data race	concurrency.c	File Scope
○ *	Defect	Data race	concurrency.c	File Scope
○ *	Defect	Data race through standard libr...	concurrency.c	File Scope
○ *	Defect	Deadlock	concurrency.c	File Scope

Review each result and determine whether you want to fix your code or add comments justifying the result.

Tutorials

- “Compiler Requirements” on page 2-2
- “Find Defects from the Polyspace Environment” on page 2-3
- “Find Defects from Simulink” on page 2-14
- “Find Defects from the Eclipse Plugin” on page 2-19

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`).

Find Defects from the Polyspace Environment

In this section...

- “Introduction” on page 2-3
- “Set Up Files and Open Polyspace” on page 2-3
- “Set Up Project” on page 2-4
- “Configure Options” on page 2-7
- “Run Analysis” on page 2-8
- “Review Results” on page 2-9
- “Fix Defects and Rerun Analysis” on page 2-12

Introduction

In this tutorial, you analyze a simple code example with Polyspace Bug Finder. The tutorial follows a common workflow for using Polyspace Bug Finder:

- 1 Set up project files.
- 2 Set configuration options.
- 3 Run analysis.
- 4 Review results.
- 5 Fix defects and rerun analysis.

Set Up Files and Open Polyspace

Before you begin an analysis, set up the source files for your Polyspace project.

In this example, *matlabroot* refers to the installation location of MATLAB®, for instance, `C:\Program Files\MATLAB\R2017b`.

- 1 In a writable location, create a folder called `bf_project`.
- 2 Copy the folder, `matlabroot\polyspace\examples\cxx\Bug_Finder_Example\sources`, to the `bf_project` folder that you created in step 1.
- 3 Make sure that the `sources` folder is writable.
- 4 Open Polyspace. You can use the Start menu, desktop shortcut, or command line:

- Start Menu: **All Programs > MATLAB > ReleaseName > Polyspace**

ReleaseName is the installed version of Polyspace.

- Desktop shortcut: if you created shortcuts during installation, from the desktop, double-click the Polyspace icon.
- DOS command line:

```
matlabroot\polyspace\bin\polyspace
```

- UNIX® command line:

```
matlabroot/polyspace/bin/polyspace
```

- MATLAB command line:

```
polyspaceBugFinder
```

Set Up Project

Set up a project for your source files and analysis options.

- 1 Select **File > New Project**.

Project - Properties

Define project


Project definition and location

Project name

Version

Author

Use default location

Location 

Project configuration

Use template

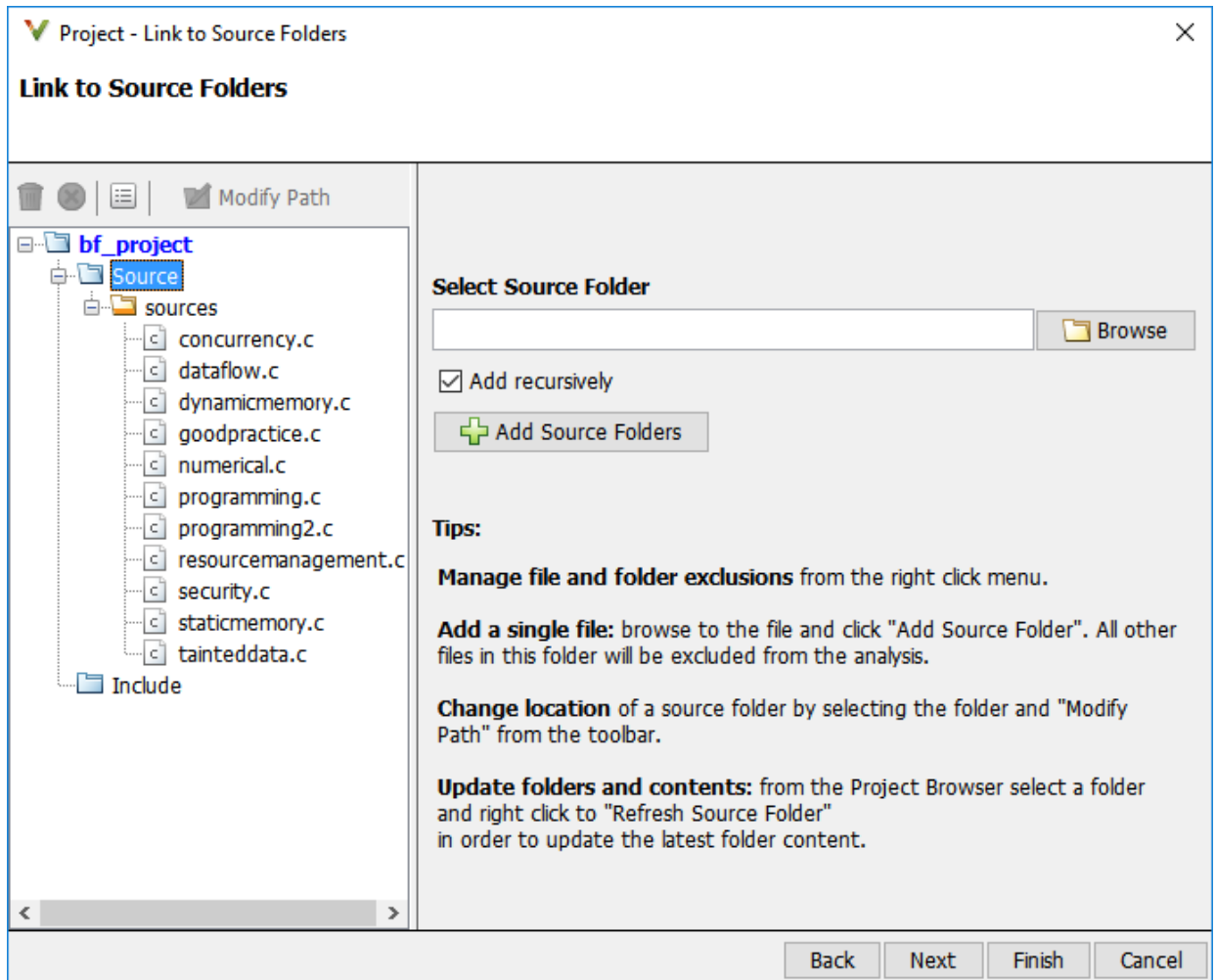
Create from build command

- 2 In **Project Name**, enter `bf_project` as your project name.
- 3 Clear **Use default location**. Enter the location of the `bf_project` folder that you created in step 1.
- 4 Select **Use template** and click **Next**.

Using a template can speed up the configuration process by presetting certain analysis options.

- 5 Select **GCC_C**, a template for C coding projects which compile with GCC. Click **Next**.
- 6 To add source files to your project, in the text box, enter the path to the `bf_project/sources` folder you created (or browse to it) and click **Add Source Folders**.

Your folder path and the source files underneath are added to the project.



Click **Next**.

- To add include folders to your project, in the text box, enter the path to the `bf_project/sources` folder you created (or browse to it) and click **Add Include Folders**.

The analysis looks for include files relative to the folder paths that you specify. For instance, if your code contains the preprocessor directive `#include<../mylib.h>` and you include the folder:

C:\My_Project\MySourceFiles\Includes

the folder C:\My_Project\MySourceFiles must contain a file mylib.h.

- 8 Click **Finish**.

Configure Options

During project setup, you selected a `GCC_C` configuration template to set the basic analysis options. You can specify additional options for checking coding rules and specific environment settings.

- 1 In the Configuration window, select the **Coding Rules & Code Metrics** node.
- 2 To add coding rules to your analysis, select **Check MISRA C:2012**.
- 3 Select the **Bug Finder Analysis** node.
- 4 To analyze all defects, select **Find Defects > all**.

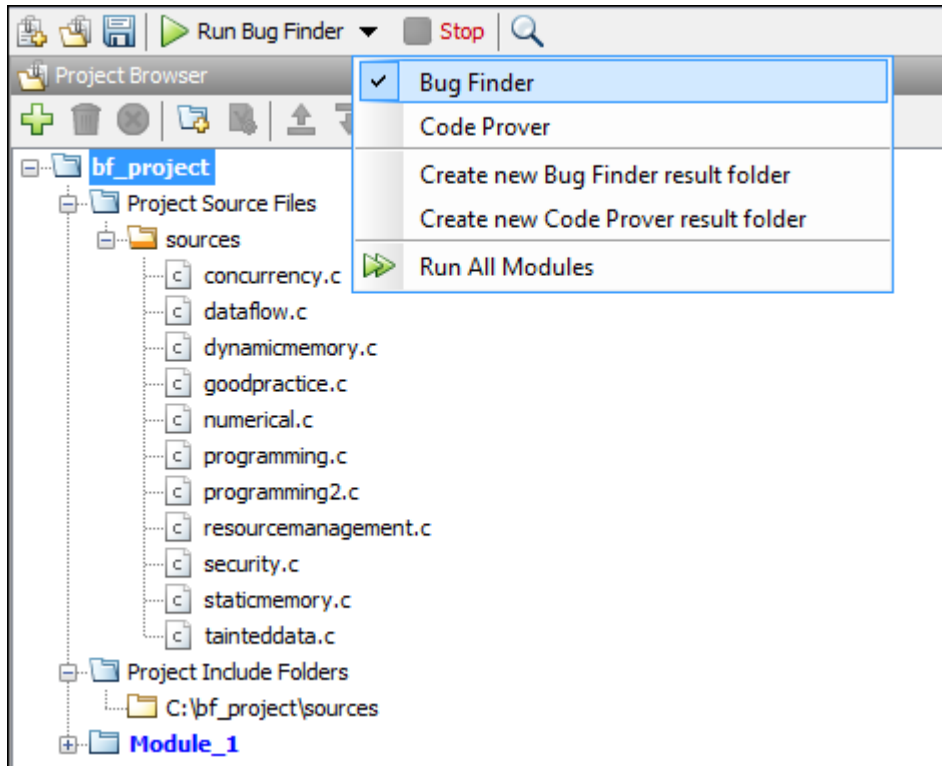
The target and compiler options are already set because this example uses a configuration template during project setup. When you analyze your own code, it is important to add all the required include files and to set the target, compiler, macro, and environment settings analysis options. If you do not, Polyspace cannot compile and analyze your code. When you run a project for the first time, it is normal to get compilation errors. For projects with your own code, look at some of the following options to improve the compilation step:

- **Target & Compiler > Compiler**: Enables different language extensions.
- **Target & Compiler > Target processor type**: Sets the size of your data types for the analysis.
- **Macros > Preprocessor definitions**: Allows you to define macros enabled by your compilation flags.
- **Multitasking**: Allows you to set up analysis of multitasking code.

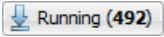
Tip You can also set up your project using a build command, such as `make`. For more information, see “Create Project Automatically”.

Run Analysis


- 1 If you do not see the **Run Bug Finder** button on the toolbar, select the Bug Finder product as follows.



- 2 Click **Run Bug Finder**.
Polyspace compiles and analyzes your project.
- 3 Follow the progress of the analysis in the **Output Summary** window.
If your project produces errors, use the troubleshooting help to diagnose the issue.


Once results are available, a button in the toolbar becomes available .

- 4 Click **Running** and start exploring your results as the analysis finishes.
Once the analysis finishes, the button on the toolbar changes to **Completed** with the number of unloaded results.

 Completed (165)

- Click **Completed** to load the remaining results.

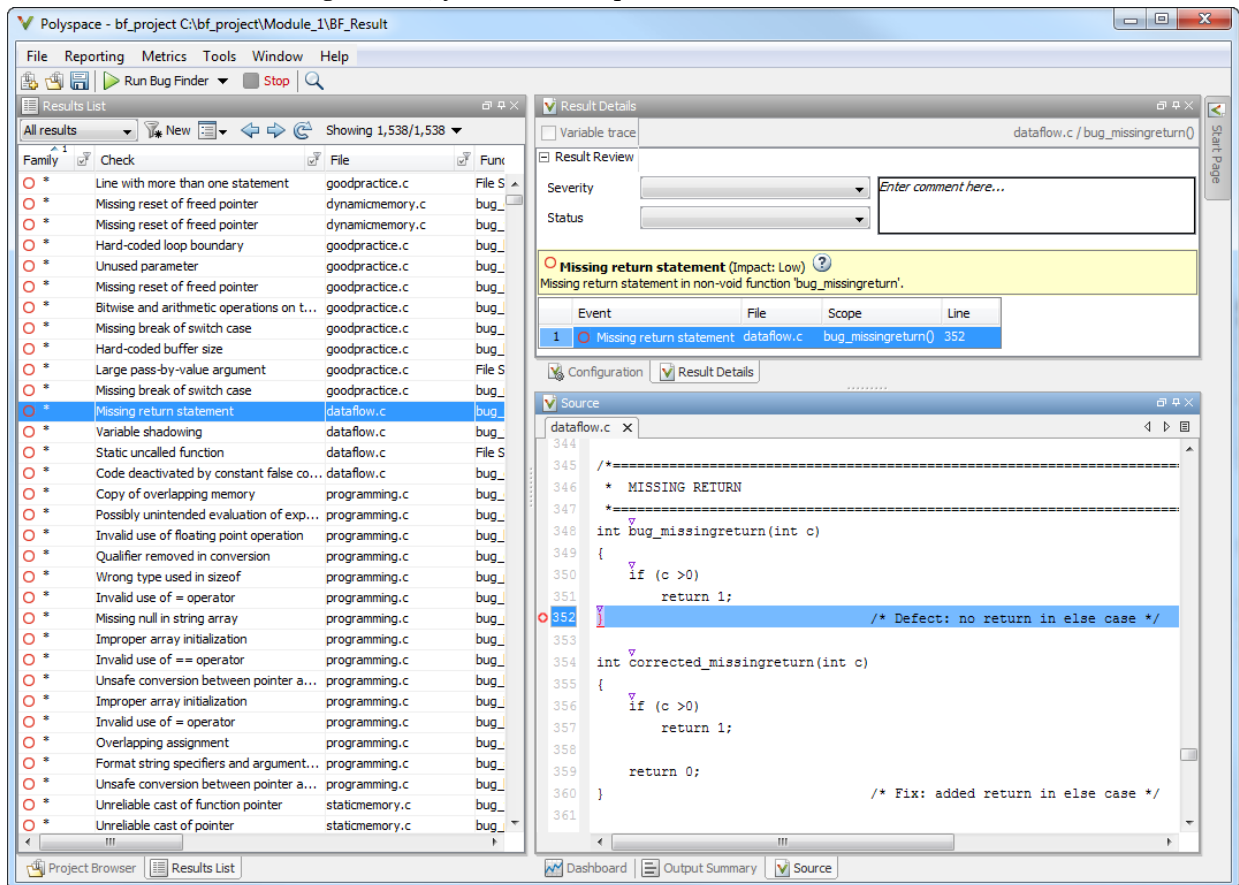
Review Results

- On the **Results List** pane, from the  list, select **None**.

You see a flat list of defects.

- Select a defect.

When reviewing results, you see three panes.



The screenshot displays the Polyspace interface with the following components:




- Results List:** A table showing a list of defects. The selected defect is:

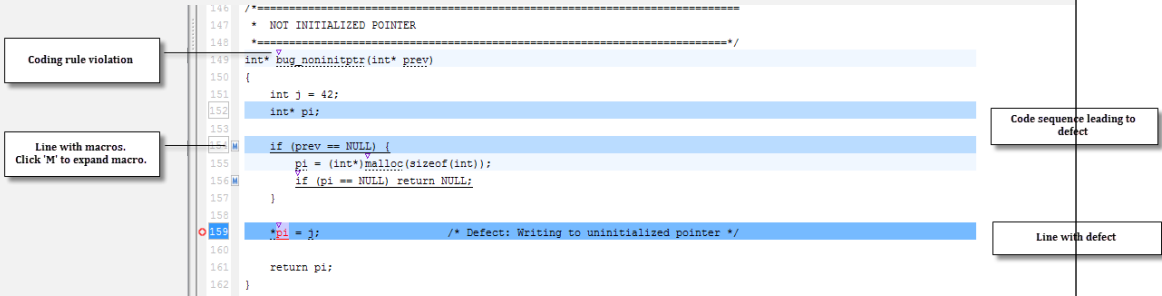
Family	Check	File	Funct
*	Missing return statement	dataflow.c	bug_...
- Result Details:** Shows the details for the selected defect:
 - Severity:
 - Status:
 - Event: Missing return statement (Impact: Low)
 - Message: Missing return statement in non-void function 'bug_missingreturn'.
 - Table:



Event	File	Scope	Line
1	dataflow.c	bug_missingreturn()	352
- Source:** Shows the source code for the selected defect:


```

344
345 /*-----
346  * MISSING RETURN
347  *-----
348 int bug_missingreturn(int c)
349 {
350     if (c > 0)
351         return 1;
352 } /* Defect: no return in else case */
353
354 int corrected_missingreturn(int c)
355 {
356     if (c > 0)
357         return 1;
358
359     return 0;
360 } /* Fix: added return in else case */
361
      
```

Pane Name	Purpose
Results List	<p>List of results</p> <p>Using the  button, you can view the results ungrouped, by family of defect types, or by file. In each of these views, you can sort or filter the results using the  icon in the column headers. Right-click the column header to add or remove columns. You can enter defect-specific comments and justifications by using the Status and Comments columns.</p>
Result Details	<p>Details about the selected result.</p> <p>The pane includes a description of the result and, if applicable, an event list to help find the root cause of the result.</p> <p>As you review your results, use the Result Review section to add justification comments and a status to the result.</p> <p>If you need more help, use the  icon to open documentation about the result.</p>

Pane Name	Purpose
Source	<p>See the selected defect in the source code.</p> <p>By default, a Dashboard tab appears showing results statistics. These statistics can provide a big picture of the defect distribution and top coding rule violations.</p> <p>As you select different checks, the source files open. Results are marked in the source code.</p>  <pre> 146 /*===== 147 * NOT INITIALIZED POINTER 148 *=====*/ 149 int* bug_noninitptr(int* prev) 150 { 151 int j = 42; 152 int* pi; 153 154 if (prev == NULL) { 155 pi = (int*)malloc(sizeof(int)); 156 if (pi == NULL) return NULL; 157 } 158 159 *pi = j; /* Defect: Writing to uninitialized pointer */ 160 161 return pi; 162 } </pre> <ul style="list-style-type: none"> • Red, underlined code with a red empty circle in the margin indicates a defect. • A purple triangle above the code indicates a coding rule violation. • Macro expansions are labeled with a blue M, which toggles between the macro and the executed code. <p>The line of code with the selected defect is highlighted in dark blue. Related lines of code are highlighted in light blue and a box outlines their line numbers.</p>

- 1 On the **Results List** pane, right-click a column header. Add the **Type** column to your results list.
- 2 On the **Type** column header, select the filter button  and filter out Defect results.
- 3 On the **Check** column header, use the filter button  to:
 - a Clear All.
 - b Select 5.3 An identifier declared in an inner scope... to view only MISRA C rule 5.3 results.

- 4 Select the coding rule violation in the `dataflow.c` file. To see the file column, you can scroll right in the Results List window.

This specific violation of MISRA C rule 5.3 must be fixed eventually, but for now, we want to add a comment to the result.

- 5 In the **Results Details** pane:
 - a Set **Severity** to `Low`.
 - b Set **Status** to `To fix`.
 - c In **Comments**, enter `Change identifier`.
- 6 Select **File > Save** to save your annotations.

Fix Defects and Rerun Analysis

- 1 Clear the filter from the **Check** column.
- 2 On the **Type** column, change the filter to see only `Defect` results.
- 3 On the **File** column, add a filter to show only results in `programming.c`.
- 4 In the `programming.c` file, find the **Invalid use of == operator** defect.


As the **Result Details** state, the error is an incorrect use of `==`. In this example, the `==` in the for-loop is supposed to be an `=`.

- 5 Right-click the red-underlined code on the **Source** pane and select **Open Editor**.

The **Code Editor** pane appears with the source code file, `programming.c`, opened to the `==` defect.

- 6 In the line where the invalid operator was found, change the `==` to `=`:

```
for (j = 5; j < (SIZE4+5); j++) {
```

- 7 Save `programming.c`  and rerun the analysis.
- 8 Once the analysis is complete, open your results and clear the filter from the **Type** and **File** columns.
- 9 Double-click the blue title bar of the Results List window. The pane maximizes to fit the Polyspace window.
- 10 Find the coding rule that you annotated earlier (5.3 An identifier declared in an inner scope...). Polyspace imports your previous comments into the new

results. You can see the severity, status, and comment that you entered are imported to the new results.

Look for the **Invalid use of == operator** defect in the `programming.c` file. The result does not appear in the **Results List** because you fixed the bug!

- 11** To return the Results List window to normal size, double-click the title bar.

Find Defects from Simulink

In this section...
“Introduction” on page 2-14
“Open Model and Generate Code” on page 2-14
“Set Polyspace Options and Run Analysis” on page 2-15
“Review Results” on page 2-15
“Fix Model and Rerun Analysis” on page 2-17

Introduction

In this tutorial, you analyze the generated code from a Simulink model using Polyspace Bug Finder. To do this analysis, the tutorial follows a common workflow for model-generated code analysis:

- 1 Create model.
- 2 Generate code.
- 3 Select Polyspace configuration options.
- 4 Run analysis.
- 5 Review results.

Open Model and Generate Code

- 1 In MATLAB, open the Polyspace example model.

```
psdemo_model_link_sl
```

- 2 Right-click the controller subsystem and select **C/C++ Code > Build This Subsystem** to generate code for the controller subsystem.
- 3 In the Build code for Subsystem: controller window, select **Build**.

Note The code generation options for this model are already set. For configuring your model, see “Recommended Model Settings for Code Analysis”.

Set Polyspace Options and Run Analysis

- 1 After the model has finished building, select **Code > Polyspace > Options**.
- 2 In the Configuration Parameters window that opens, on the **Polyspace** pane, set the following options.

Option	Value
Product mode	Bug Finder
Settings from	Project configuration and MISRA C 2012 checking for generated code

These options set the type of Polyspace analysis and configure the analysis to check for bugs and MISRA C coding rule violations.

- 3 Apply your changes and close the Configuration Parameters window.
- 4 Right-click the controller subsystem and select **Polyspace > Verify Code Generated For > Selected Subsystem**.



You can follow the progress of the analysis in the Command Window.


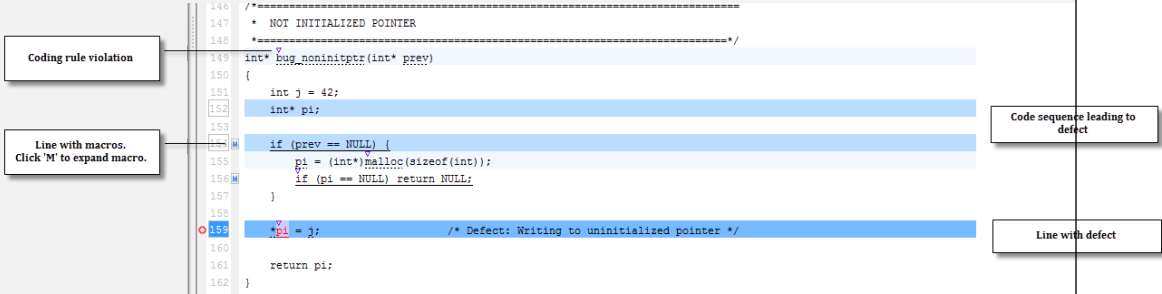
Review Results

After the analysis has finished, open and review the results in the Polyspace interface.

- 1 If the results do not open automatically, right-click the controller subsystem and select **Polyspace > Open Results**.
- 2 Select the **Integer division by zero** result.

In Bug Finder, you see three windows:

Pane Name	Purpose
Results List	<p>List of results</p> <p>Using the  button, you can view the results ungrouped, by family of defect types, or by file. In each of these views, you can sort or filter the results using the  icon in the column headers. Right-click the column header to add or remove columns. You can enter defect-specific comments and justifications by using the Status and Comments columns.</p>

Pane Name	Purpose
Result Details	<p>Details about the selected result.</p> <p>The pane includes a description of the result and, if applicable, an event list to help find the root cause of the result.</p> <p>As you review your results, use the Result Review section to add justification comments and a status to the result.</p> <p>If you need more help, use the  icon to open documentation about the result.</p>
Source	<p>See the selected defect in the source code.</p> <p>By default, a Dashboard tab appears showing results statistics. These statistics can provide a big picture of the defect distribution and top coding rule violations.</p> <p>As you select different checks, the source files open. Results are marked in the source code.</p> <div data-bbox="268 855 1426 1150" style="border: 1px solid gray; padding: 5px;">  <pre> 146 /*===== 147 * NOT INITIALIZED POINTER 148 *=====*/ 149 int* bug_noninitptr(int* prev) 150 { 151 int j = 42; 152 int* pi; 153 154 #M if (prev == NULL) { 155 pi = (int*)malloc(sizeof(int)); 156 if (pi == NULL) return NULL; 157 } 158 159 *pi = j; /* Defect: Writing to uninitialized pointer */ 160 161 return pi; 162 } </pre> <p>Coding rule violation</p> <p>Line with macros. Click 'M' to expand macro.</p> <p>Code sequence leading to defect</p> <p>Line with defect</p> </div> <ul style="list-style-type: none"> • Red, underlined code with a red empty circle in the margin indicates a defect. • A purple triangle above the code indicates a coding rule violation. • Macro expansions are labeled with a blue M, which toggles between the macro and the executed code. <p>The line of code with the selected defect is highlighted in dark blue. Related lines of code are highlighted in light blue and a box outlines their line numbers.</p>

- 3 Look at the **Result Details** pane. The description states that the divisor, `controller_B.Cumulatedangle`, may be zero.

- 4 Hover over the red division symbol, /, in the **Integer division by zero** line. The tooltip tells you the ranges of the two operands and the result. The range of the divisor (right operand) contains zero. The possibility of this value causes the **Integer division by zero** defect.
- 5 In the **Result Details** pane, starting at the bottom of the list of events, select the events to see where the value of `controller_B.Cumulatedangle` is assigned.

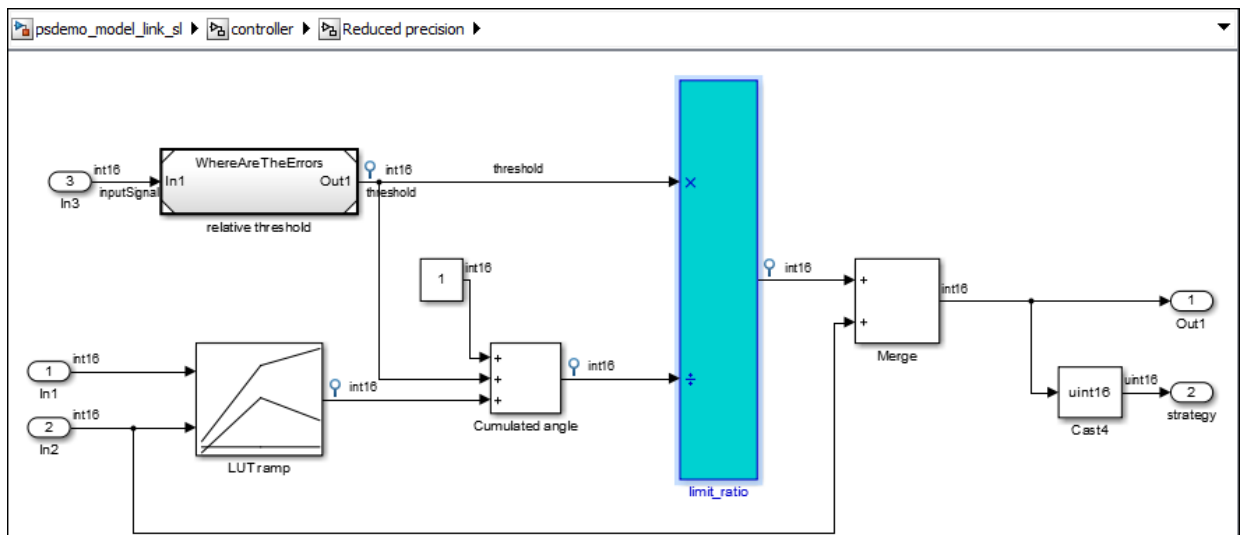
The second event shows that `controller_B.Cumulatedangle` is assigned the value of `tmp`. The statements that assign `tmp` a non-zero value are in dead if/else branches.

Fix Model and Rerun Analysis

This section shows you how to fix the **Integer division by zero** defect you examined before. As you learned from reviewing the code, the defect occurs because `controller_B.Cumulatedangle`, the divisor, can be zero.

- 1 To find where this division takes place in the model, in the **Source** pane above the **Integer division by zero** defect, click `<S4>/limit_ratio`.

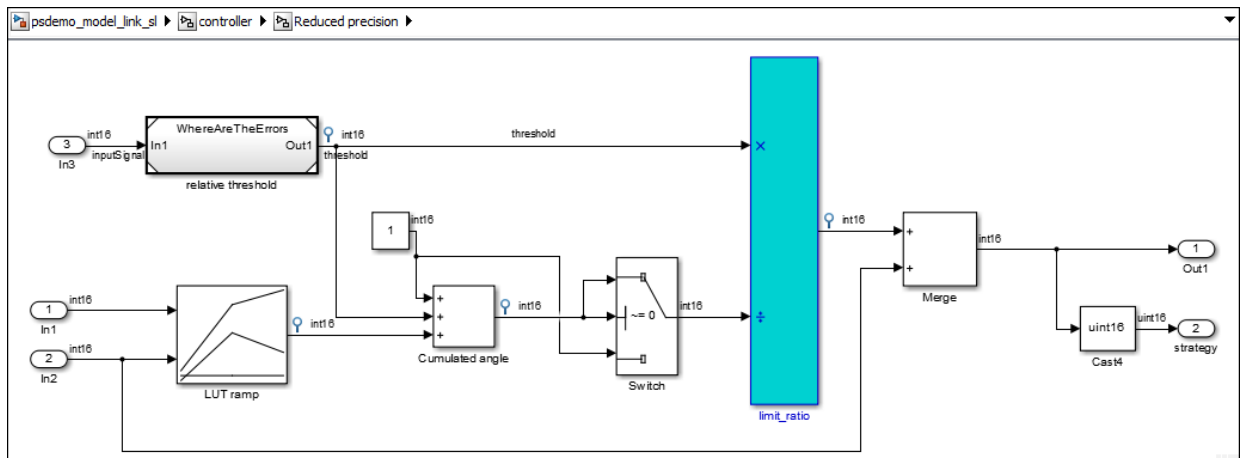
The `limit_ratio` block associated with this line of code is highlighted in the Simulink model in blue.



This link between the model and the generated code helps you identify the parts of your model that are causing defect. That way you can fix the defects in the model rather than the code.

- 2 In between the highlighted `limit_ratio` block and the Cumulated angle block, add a Switch block.
- 3 Change the criteria property of the Switch block to `u2 ~= 0`.
- 4 Connect the Switch block to the Cumulated angle signal and the constant 1 block so that the Cumulated angle is compared to zero. If the angle is not zero, the cumulated angle is output. If the signal is equal to zero, the 1 is output.

The Reduced precision subsystem should look like the following:



- 5 Regenerate the code for the controller subsystem. You might need to save a new version of the model.
- 6 Rerun the Polyspace analysis for the controller subsystem.
- 7 Open the results.

The new results no longer contain an **Integer division by zero** defect because you fixed the model and by extension the generated code.

Find Defects from the Eclipse Plugin

In this section...
“Introduction” on page 2-19
“Run Analysis and Review Results” on page 2-19

Introduction

Before starting a code analysis, you must install the Polyspace Bug Finder plugin for Eclipse. For instructions see, “Install Polyspace Plugin for Eclipse IDE” on page 4-4.

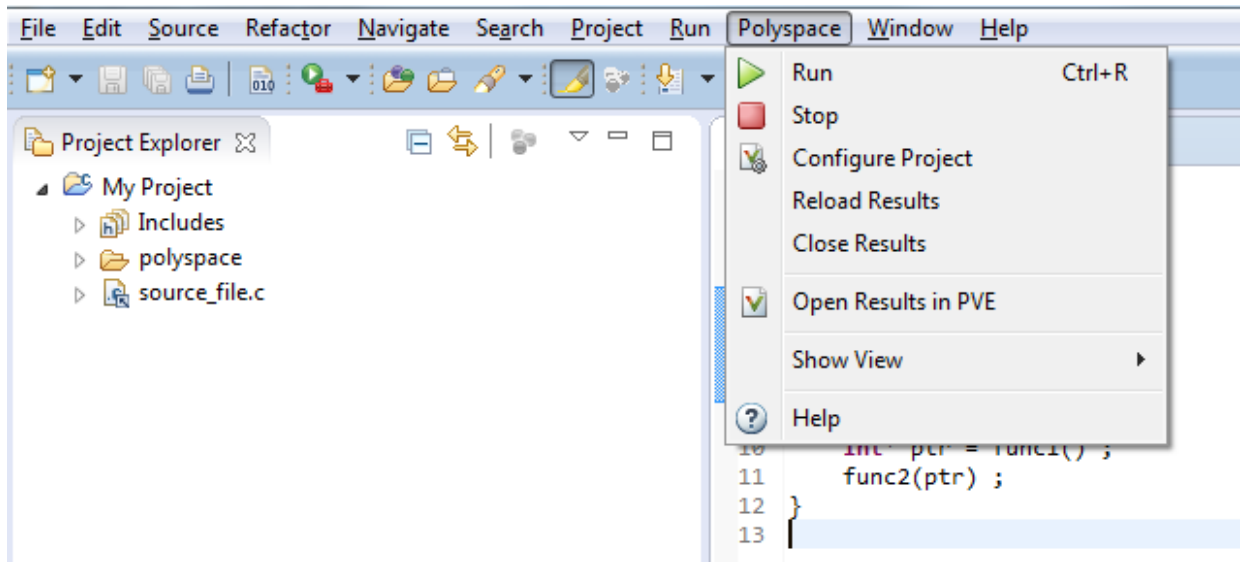
In this tutorial, you analyze a simple code example using Polyspace Bug Finder in Eclipse. A common workflow for code analysis with Polyspace Bug Finder is:

- 1 Set up project and configuration options.
- 2 Run analysis.
- 3 Review results and fix defects.
- 4 Rerun analysis.

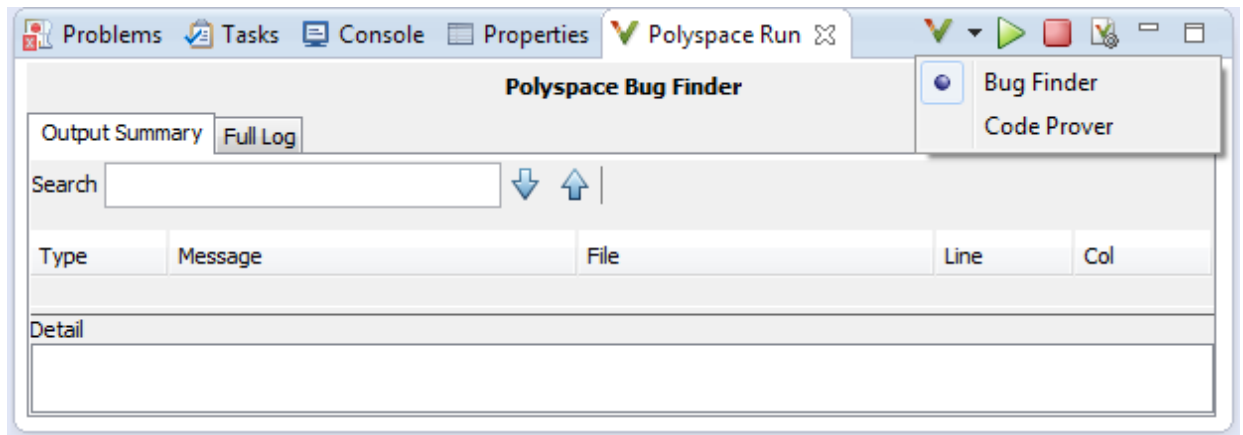
This tutorial follows a shortened version of this workflow. For details, see “Analysis in Eclipse”.

Run Analysis and Review Results

- 1 After you install the plugin, a Polyspace menu appears on the toolbar. Select **Polyspace > Show View > Show Polyspace Run view** to view the **Polyspace Run - Bug Finder** pane.



- 2 In the **Polyspace Run - Bug Finder** pane, select Bug Finder under the product icon.




By selecting Bug Finder, Polyspace uses Bug Finder configuration options and analysis to analyze your Eclipse project.

- 3 In the **Project Explorer** pane, right-click a project with C or C++ files and from the context menu select **Run Polyspace Bug Finder**.

Note You can also right-click a single source file to analyze only that file.

As the analysis runs, you can follow the progress in the **Output Summary** tab of the **Polyspace Run - Bug Finder** pane. If your code has compilation errors that prevent analysis, they appear in the same tab.

- 4 After the analysis finishes, the results appear on the **Results List - Bug Finder** pane. As you select different defects, the source code switches to that line number and details about the defect appear in the **Result Details** pane.
- 5 After fixing bugs or adding code, you can rerun the analysis from the Results List window by clicking the rerun button, .

Polyspace UML Link RH

Find Defects from IBM Rational Rhapsody

In this section...
“Code Analysis Approach” on page 3-2
“Adding Polyspace Profile to Model” on page 3-3
“Accessing Polyspace Features” on page 3-5
“Configuring Analysis Options” on page 3-7
“Running an Analysis” on page 3-8
“Monitoring an Analysis” on page 3-10
“Viewing Polyspace Results” on page 3-11
“Locating Faulty Code in Rhapsody Model” on page 3-11
“Template Configuration Files” on page 3-13

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, go to www-01.ibm.com/software/awdtools/rhapsody/.

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 3-11 and “Locating Faulty Code in Rhapsody Model” on page 3-11.

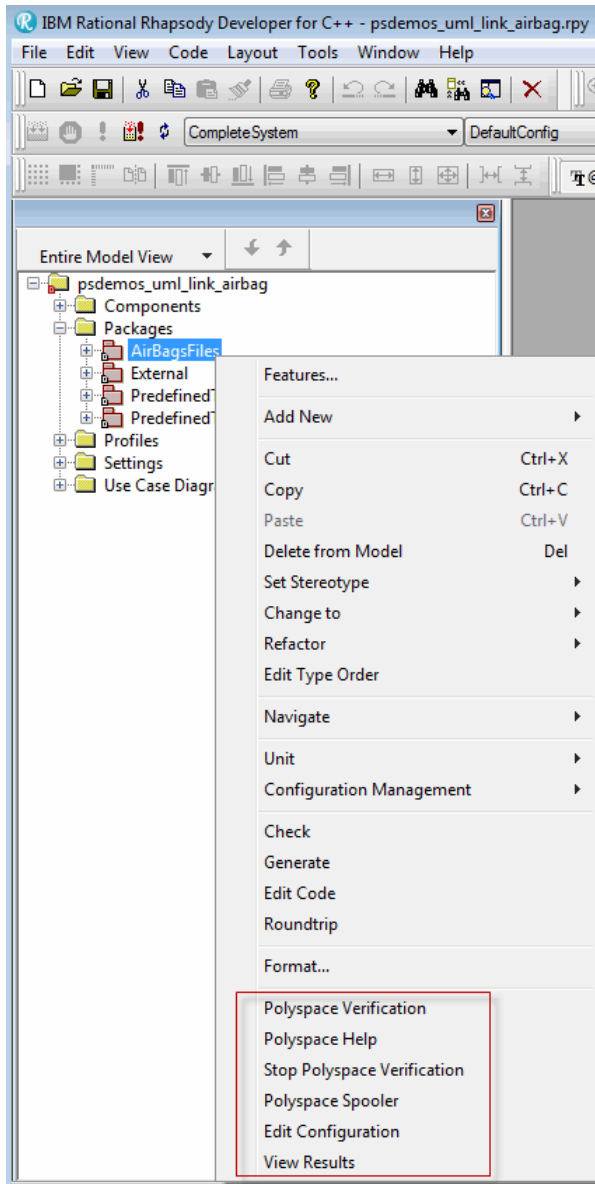
Adding Polyspace Profile to Model

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.

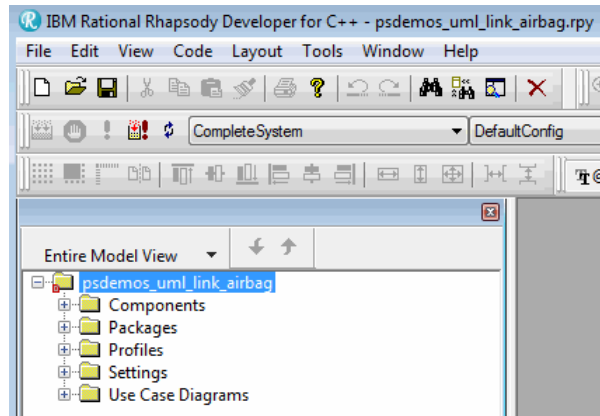
To install the 32-bit Polyspace version, from a DOS command window, run the following command:

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

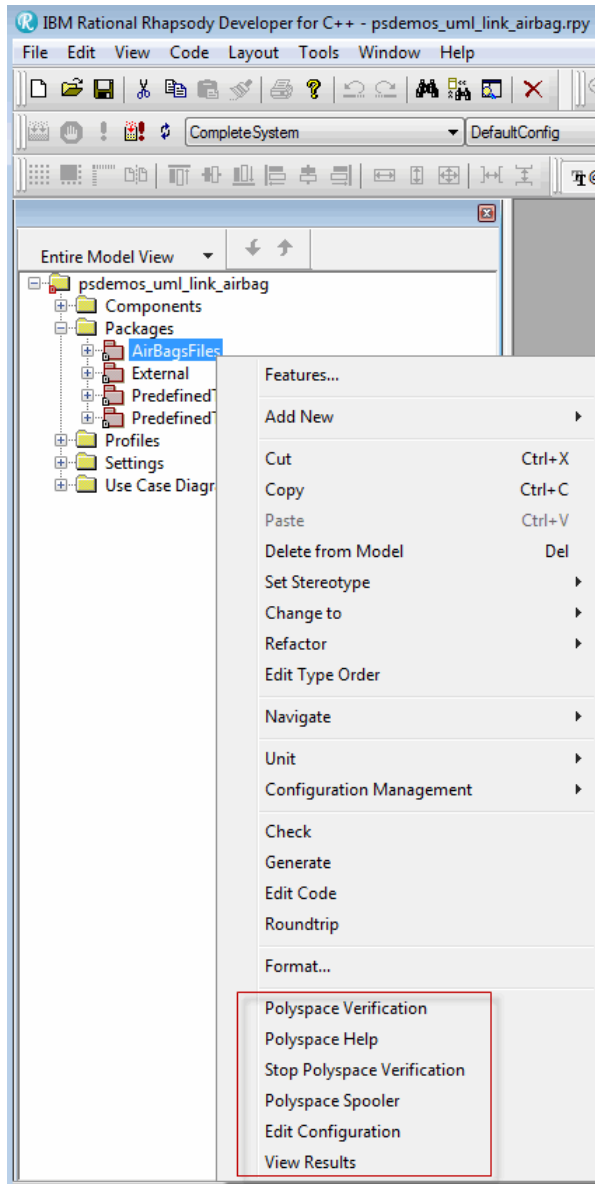
Accessing Polyspace Features

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:

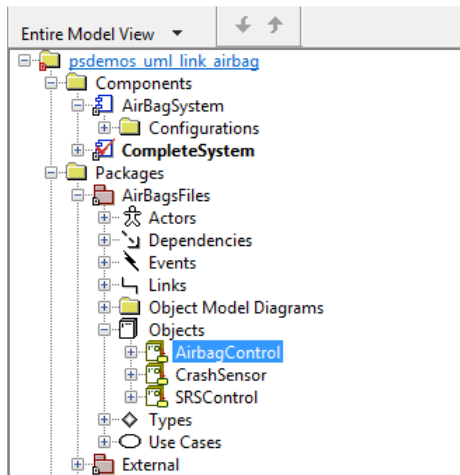
- **Polyspace Verification** — Start analysis. See “Running an Analysis” on page 3-8.
- **Polyspace Help** — Open help.
- **Stop Polyspace Verification** — Stop client-based analysis. See “Running an Analysis” on page 3-8.
- **Polyspace Job Monitor** — Open Polyspace Job Monitor. See “Monitoring an Analysis” 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.

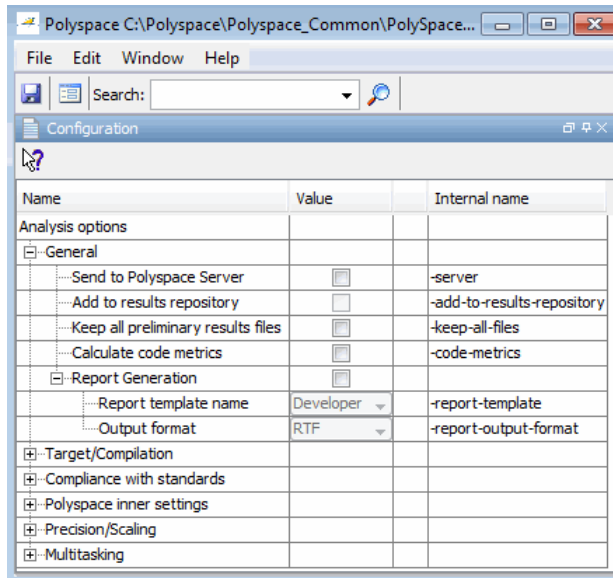
Configuring Analysis Options

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.



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.

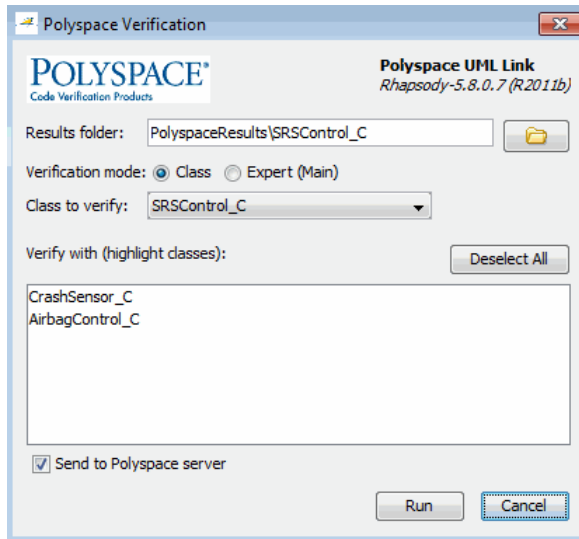
4 To save your options, in the top left corner, click the disk button.

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.

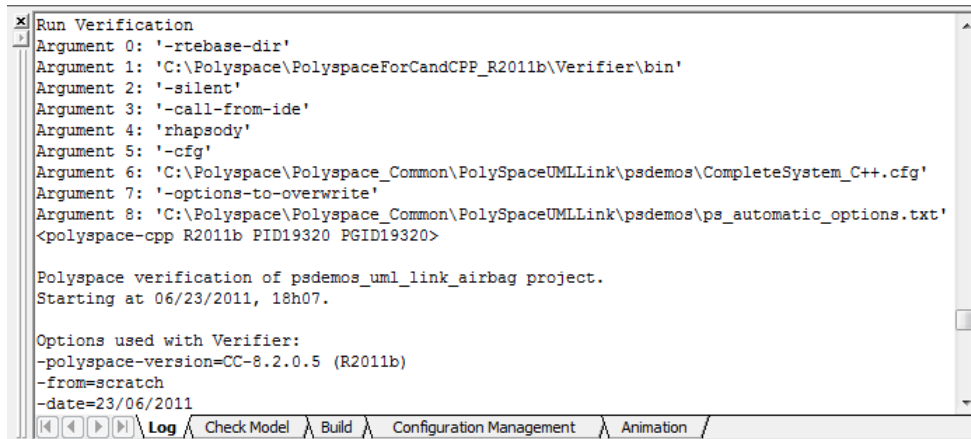


Note Before starting an analysis, make sure that the generated code for the model is up to date.

- 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.

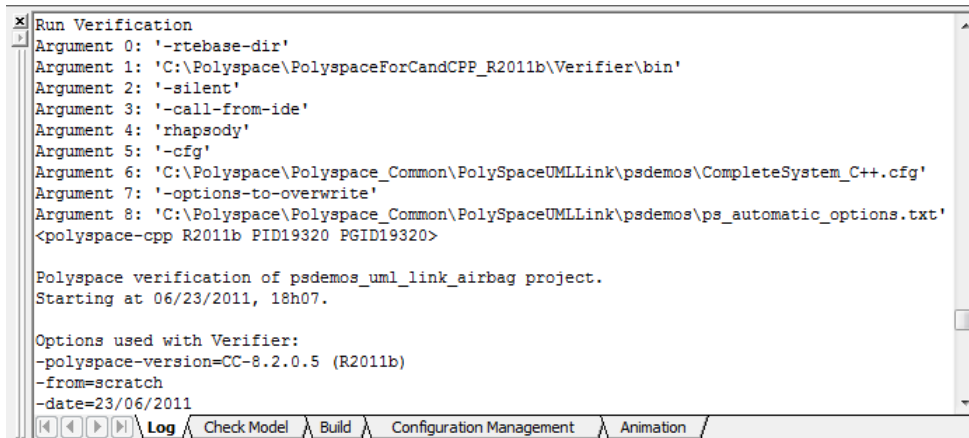
Options used with Verifier:
-polyspace-version=CC-8.2.0.5 (R2011b)
-from=scratch
-date=23/06/2011
```

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.

For more information, see “Monitor Analysis”.

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 “Result Review Process” and “Result Management”.

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.
```

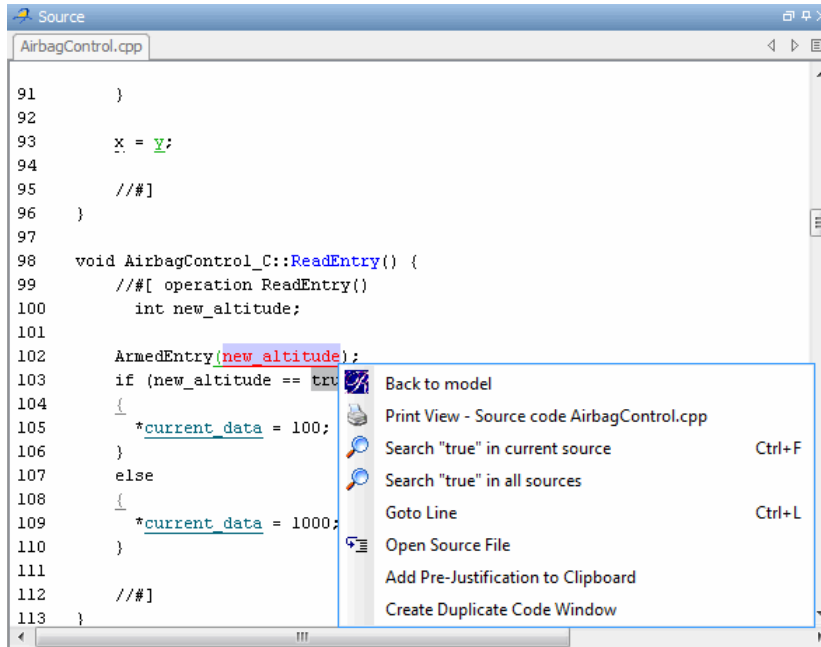
To avoid this problem, in Rhapsody, at the project level, set the property `C_CG::Configuration::EmptyArgumentListName` to `void`.

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`.

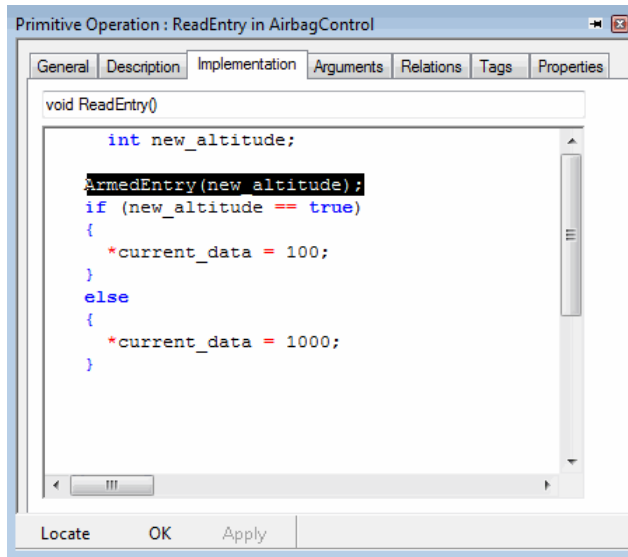
- 2 In the Source pane, right-click the error. From the context menu, select **Back to model**.



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.



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.

Installation and Configuration

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 R201b 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

- “Find Defects from Simulink” on page 2-14

More About

- “Troubleshoot Back 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

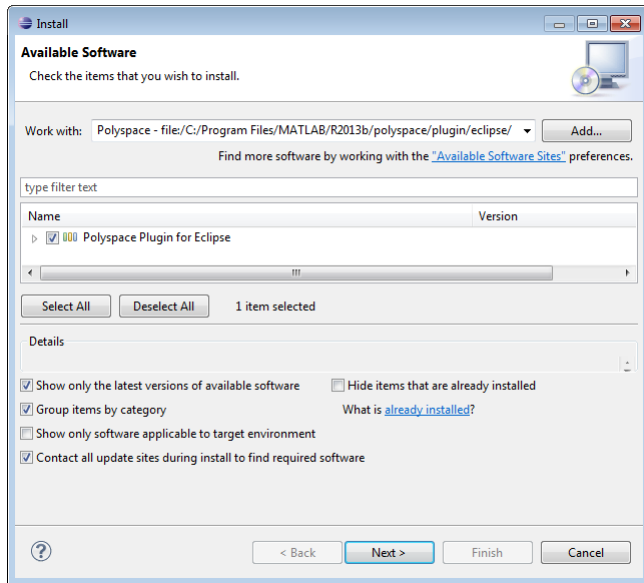
The Polyspace plugin is supported for Eclipse versions 4.3, 4.4, and 4.5. You can install the Polyspace plugin only after you:

- Install and set up Eclipse Integrated Development Environment (IDE). For more information, see the Eclipse documentation at www.eclipse.org.
- Install Java® 7. See Java documentation at www.java.com.
- 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**.

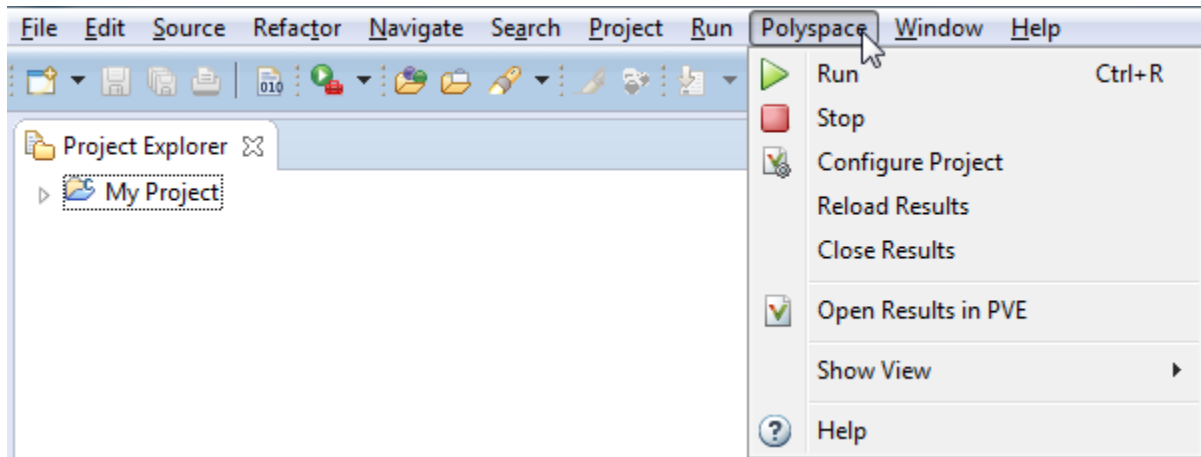
MATLAB_Install is the installation folder for the Polyspace product.
- 6 Click **OK** to close the Add Repository dialog box.
- 7 On the Available Software page, select Polyspace Plugin for Eclipse.



- 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 - 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

Related Examples

- “Find Defects from the Eclipse Plugin” on page 2-19

More About

- “Analysis in Eclipse”

Set Up Polyspace Metrics

In this section...
“Requirements for Polyspace Metrics” on page 4-7
“Start Polyspace Metrics Server” on page 4-8
“Configure Polyspace Preference” on page 4-9
“Configure Web Server for HTTPS” on page 4-10
“Change Web Server Port Number for Metrics Server” on page 4-12

This topic shows how to set up a Polyspace Web Metrics server to store results and monitor software quality.

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	<ul style="list-style-type: none"> • MATLAB • Polyspace Bug Finder
Polyspace Metrics service	Network server or head node of MATLAB Distributed Computing Server™ cluster	<ul style="list-style-type: none"> • MATLAB • Polyspace Bug Finder <p>Activation is not required for the Polyspace Metrics service</p>

Task	Location	Requirements
Downloading <i>complete</i> results from Polyspace Metrics	Client node or a network computer	<ul style="list-style-type: none"> • MATLAB • Polyspace Bug Finder • Access to Polyspace Metrics server
Viewing results <i>summary</i> 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.

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 on page 4-9 so that the Polyspace interface can interact with the Metrics server.

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

- 1 From the Polyspace environment, select **Metrics > Metrics and Remote Server Settings**.
- 2 Under **Polyspace Metrics Settings**, specify:
 - **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 Polyspace Preference” on page 4-9.
 - **Folder where analysis data will be stored** — Results repository for Polyspace Metrics server.
- 3 If you have installed MATLAB Distributed Computing Server, clear the **Start the Polyspace mdce service without security level** check box.

For information about starting your remote cluster service, see “Set Up Server for Metrics and Remote Analysis” on page 4-13.

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

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`

Configure Polyspace Preference

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 Under the **Polyspace Metrics server configuration** section:
 - 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 4-10.
 - 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 “Change Web Server Port Number for Metrics Server” on page 4-12.

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 “Start Polyspace Metrics Server” on page 4-8 and “Configure Polyspace Preference” on page 4-9.

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”

## Set Up Server for Metrics and Remote Analysis

### In this section...

“Requirements for Remote Verification and Analysis” on page 4-14

“Start Server for Remote Verification and Polyspace Metrics” on page 4-14

“Configure Polyspace Preferences” on page 4-16

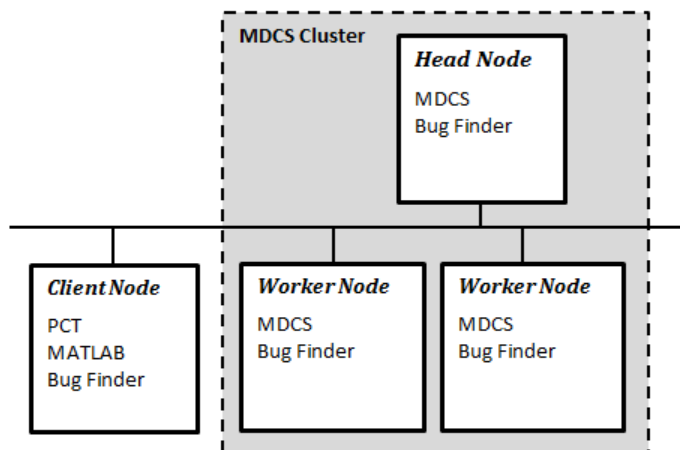
You can perform a Polyspace verification locally on your desktop or on a remote server. This topic shows how to set up Polyspace on a server for remote batch verification.

Use these rules to determine whether to opt for remote or local verification.

Type	When to Use
Remote <i>batch</i>	Source files are large (more than 800 lines of code including comments), and execution time of verification is long.
Local	Source files are small, and execution time of verification is short.

With both local and remote verification, you can upload your results to the Polyspace Metrics web interface or view them directly on your desktop application. For more information about setting up Polyspace Metrics, see “Set Up Polyspace Metrics” on page 4-7.

The following figure shows a network that consists of a MATLAB Distributed Computing Server cluster and a Parallel Computing Toolbox client. Polyspace Code Prover™ and Polyspace Bug Finder are installed on the head node and client nodes.



To set up remote verification:

- 1 Configure the head node with the Metrics and Remote Server Settings dialog box. See, “Start Server for Remote Verification and Polyspace Metrics” on page 4-14.
- 2 Configure the client node through the Polyspace environment preferences. See, “Configure Polyspace Preferences” on page 4-16.

### Requirements for Remote Verification and Analysis

The following table lists the requirements for remote analysis.

Task	Location	Requirements
Project configuration and job submission	Client node	<ul style="list-style-type: none"> <li>• MATLAB</li> <li>• Parallel Computing Toolbox</li> <li>• Polyspace Bug Finder</li> </ul>
Remote analysis	Head node of cluster	<ul style="list-style-type: none"> <li>• MATLAB Distributed Computing Server</li> <li>• Polyspace Bug Finder</li> </ul>

For information about setting up a computer cluster, see “Install Products and Choose Cluster Configuration” (MATLAB Distributed Computing Server).

### Start Server for Remote Verification and Polyspace Metrics

This procedure describes how to set up a MATLAB Distributed Computing Server head node that is also the Polyspace Metrics server. If you do not want to set up Polyspace Metrics, use the MATLAB Distributed Computing Server Admin Center to set up a server for your remote verifications. See “Install Products and Choose Cluster Configuration” (MATLAB Distributed Computing Server).

- 1 Select **Metrics > Metrics and Remote Server Settings**.
- 2 Under **Polyspace Metrics Settings**, specify:
  - **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 on the **Polyspace Preferences > Server Configuration** tab.

- **Folder where analysis data will be stored** — Results repository for Polyspace Metrics server.
- 3** To configure the Polyspace Metrics server as the MATLAB Distributed Computing Server head node, select **Start the Polyspace mdce service without security level**.

The `mdce` service, which is required to manage the MJS, runs on the MJS host computer 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.

If you want to require authentication to use the remote server, use the MATLAB Distributed Computing Server **Admin Center**. For more information about setting up security levels, see “Set MJS Cluster Security” (MATLAB Distributed Computing Server).

Under **Start the Polyspace mdce service without security level**, you see the following additional options:

- **Mdce service port** — 27350.

This option specifies the port on which you connect to the MJS server. 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 using the notation `schedulerName:portNumber`. For instance, `myJobScheduler:27400`. See “Verify Network Communications for Cluster Discovery” (MATLAB Distributed Computing Server).

- **Use secure communication** – Not selected by default

By default, communication between the job manager and workers is not encrypted. To make the connection more secure, you can select this option to encrypt communications. Alternatively, you can increase the security level of your MJS server. See “Set MJS Cluster Security” (MATLAB Distributed Computing Server).

- 4** To start the Polyspace Metrics server and `mdce` service, click **Start Daemon**.

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

- On a Windows system, `%APPDATA%\PolyspaceRLDats\polyspace.conf`

- On a Linux system, `/etc/Polyspace/polyspace.conf`

### Configure Polyspace Preferences

- 1 Select **Tools > Preferences**.
- 2 Click the **Server Configuration** tab.
- 3 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).
  - b Due to network setting, the job manager may be unable to connect back to your local computer. If this is the case, enter the IP address of the client computer in the **Localhost IP address** field.

To retrieve your IP address:

- Windows
  - i Open **Control Panel > Network and Sharing Center**.
  - ii Select your active network.
  - iii In the Status window, click **Details**. Your IP address is listed under **IPv4 address**.
- Linux — Run the `ifconfig` command and find the `inet addr` corresponding to your network connection.
- Mac — Open **System Preferences > Network**.

If required, you can configure additional options for the MJS host through the MATLAB Distributed Computing Server Admin Center. See “Configure for an MJS” (MATLAB Distributed Computing Server).

- 4 Under the **Polyspace Metrics server configuration** section:
  - 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.

- 5 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 4-10.
  - 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 “Change Web Server Port Number for Metrics Server” on page 4-12.

- 6 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.

## See Also

### Related Examples

- “Set Up Polyspace Metrics” on page 4-7
- “Run Remote Batch Analysis”
- “Job Manager Cannot Write to Database”

# Using Bug Finder and Code Prover

---

## Differences Between Polyspace Bug Finder and Polyspace Code Prover Analysis

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<sup>1</sup>. 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 5-3
- “Faster Analysis with Bug Finder” on page 5-3
- “More Exhaustive Verification with Code Prover” on page 5-3
- “More Specific Defect Types with Bug Finder” on page 5-4
- “Easier Setup Process with Bug Finder” on page 5-5
- “Fewer Runs for Clean Code with Bug Finder” on page 5-6
- “Results in Real Time with Bug Finder” on page 5-6
- “More Rigorous Data and Control Flow Analysis with Code Prover” on page 5-7
- “Few False Positives with Bug Finder” on page 5-8

---

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 “Verification Following Red and Orange Checks” (Polyspace Code Prover).

- “Zero False Negatives with Code Prover” on page 5-9

## 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 5-9.

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	178	31 (Critical subset)
Depth of analysis	Fast. For instance: <ul style="list-style-type: none"> <li>• Faster analysis.</li> <li>• Easier set up and review.</li> <li>• Fewer runs for clean code.</li> <li>• Results in real time.</li> </ul>	Exhaustive. For instance: <ul style="list-style-type: none"> <li>• All operations of a type checked for certain critical errors.</li> <li>• More rigorous data and control flow analysis.</li> </ul>
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 5-7. 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 5-9.

### 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 & 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 “Verification 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.

```

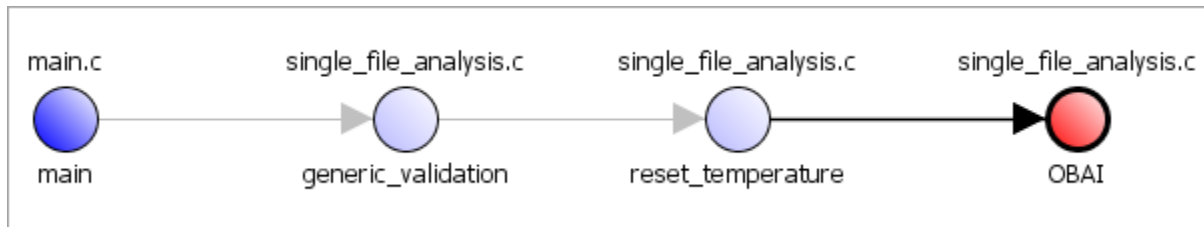
167 static void Square_Root_conv(double alpha, float* beta_pt)
168 /* Perform arithmetic conversion of alpha to beta */
169 {
170 *beta_pt = (float)((1.5 + cos(alpha)) / 5.0);
171 }
172
173
174 stati
175 {
176 d
177 f
178 f
179
180 Square_Root_conv(alpha, sbeta);
181
182 gamma = (float)sqrt(beta - 0.75); /* 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<sup>2</sup>.

## 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<sup>3</sup>. 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 the Bug Finder and Code Prover softwares, 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 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.

---

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.

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 & Compiler”
  - “Macros”
  - “Environment Settings”
  - “Inputs & 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.