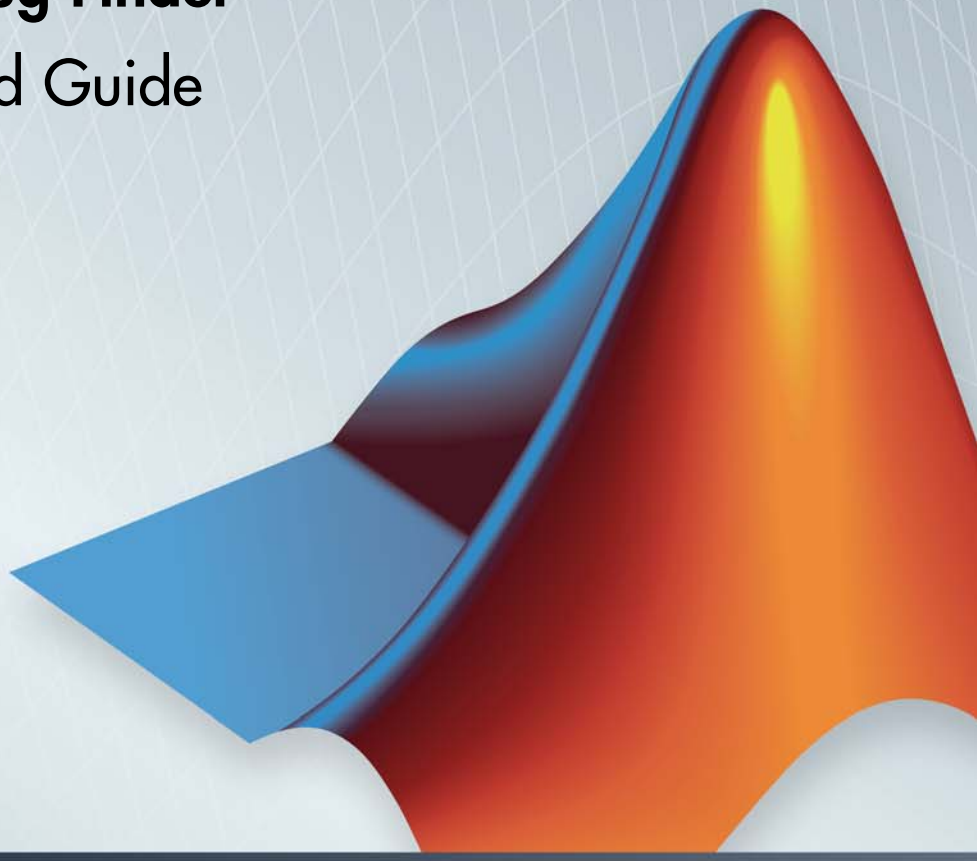
Polyspace® Bug Finder™

Getting Started Guide

R2014**a**



MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Polyspace® Bug Finder™ Getting Started Guide

© COPYRIGHT 2013–2014 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)

Contents

1 About Polyspace Bug Finder

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

2 Tutorials

Find Defects from the Polyspace Environment 2-2
Introduction 2-2
Set Up Project 2-2
Configure Text Editor 2-5
Configure Coding Rules and Run Analysis 2-6
Review Results 2-6
Fix Defects and Rerun Analysis 2-10

Find Defects from Simulink 2-11
Introduction 2-11
Create Simulink Model and Generate Code 2-11
Run Bug Finder Analysis 2-13
Review Results 2-13

Find Defects from the Eclipse Plug-In **2-17**
Introduction **2-17**
Run Analysis and Review Results **2-17**

Find Defects from Visual Studio **2-20**
Introduction **2-20**
Run Analysis in Visual Studio **2-20**
Review Results **2-24**

Install Polyspace Plug-In for Simulink **2-25**

Install Polyspace Plug-In for Eclipse **2-26**
Install Polyspace Plug-In for Eclipse IDE **2-26**
Uninstall Polyspace Plug-In for Eclipse IDE **2-28**

Install Polyspace Add-In for Visual Studio **2-29**
Install Polyspace Add-In for Visual Studio **2-29**
Uninstall Polyspace Add-In for Visual Studio **2-30**

3 Polyspace UML Link RH

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 **3-9**
Monitoring an Analysis **3-11**
Viewing Polyspace Results **3-11**
Locating Faulty Code in Rhapsody Model **3-12**
Template Configuration Files **3-14**

1

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

Polyspace Bug Finder Product Description

Identify software defects via static analysis

Polyspace® Bug Finder™ identifies run-time errors, data flow problems, and other defects in C and C++ embedded software. Using static analysis, Polyspace Bug Finder analyzes software control, data flow, and interprocedural behavior. 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 such as 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, dSPACE® TargetLink® blocks, and IBM® Rational® Rhapsody® diagrams.

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

- Detection of run-time errors, data flow problems, and other defects in C and C++ code
- Fast analysis of large code bases
- Compliance checking for MISRA C:2004, MISRA C++:2008, JSF++, and custom naming conventions
- Cyclomatic complexity and other code metrics
- Eclipse integration
- Traceability of code verification results to Simulink models
- Access to Polyspace Code Prover™ results
- 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

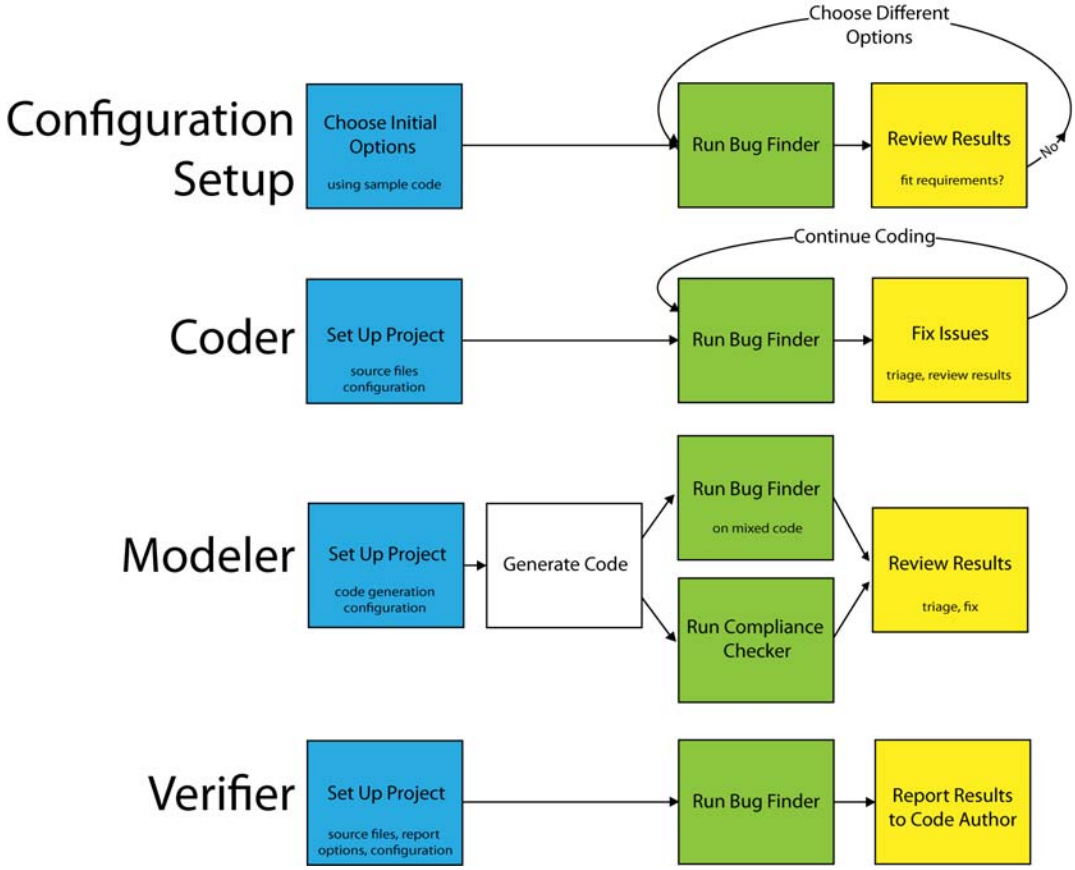
For information about Polyspace products that verify Ada code, see the following:

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

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

Bug Finder Workflows

Below are 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.

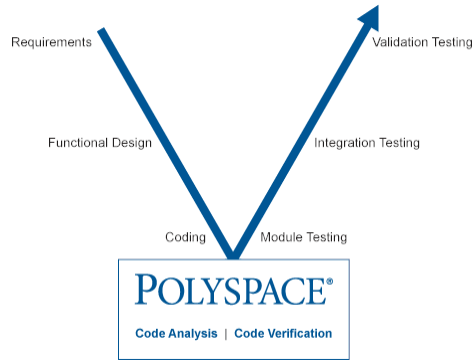
Generally, the criticality of your application determines the balance between these three variables – your quality model. With classical testing processes, development teams generally try to achieve their quality model by testing all modules in an application until each meets the required quality level. 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.

2

Tutorials

- "Find Defects from the Polyspace Environment" on page 2-2
- "Find Defects from Simulink" on page 2-11
- "Find Defects from the Eclipse Plug-In" on page 2-17
- "Find Defects from Visual Studio" on page 2-20
- "Install Polyspace Plug-In for Simulink" on page 2-25
- "Install Polyspace Plug-In for Eclipse" on page 2-26
- "Install Polyspace Add-In for Visual Studio" on page 2-29

Find Defects from the Polyspace Environment

In this section...
"Introduction" on page 2-2
"Set Up Project" on page 2-2
"Configure Text Editor" on page 2-5
"Configure Coding Rules and Run Analysis" on page 2-6
"Review Results" on page 2-6
"Fix Defects and Rerun Analysis" on page 2-10

Introduction

In this tutorial, you analyze a simple code example using Polyspace Bug Finder. To do this analysis, the tutorial follows a common workflow for using Polyspace Bug Finder:

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

Set Up Project

Set up the source files and create a new Polyspace project to analyze these files.

Note In this example, *MATLAB_ROOT* refers to the installation location of MATLAB®. For example:

```
C:\Program Files\MATLAB\R2013b.
```

---

1. In a writable folder, create a new folder called `bf_project`.

2. Copy the folder, *MATLAB_ROOT*\polyspace\examples\cxx\Bug_Finder_Example\sources, to the bf_project folder that you created in step 1.

3. Open Bug Finder. To open Bug Finder, you can use the Start menu, desktop shortcut, or command line:

   - Start Menu: **All Programs > MATLAB > R2013b > Polyspace Bug Finder R2013b**
   - Desktop shortcut: if you created shortcuts during installation, select the Polyspace Bug Finder icon from the desktop
   - DOS command-line: enter:

     ```
     MATLAB_ROOT\polyspace\bin\polyspace-bug-finder
     ```

   - UNIX command-line: enter:

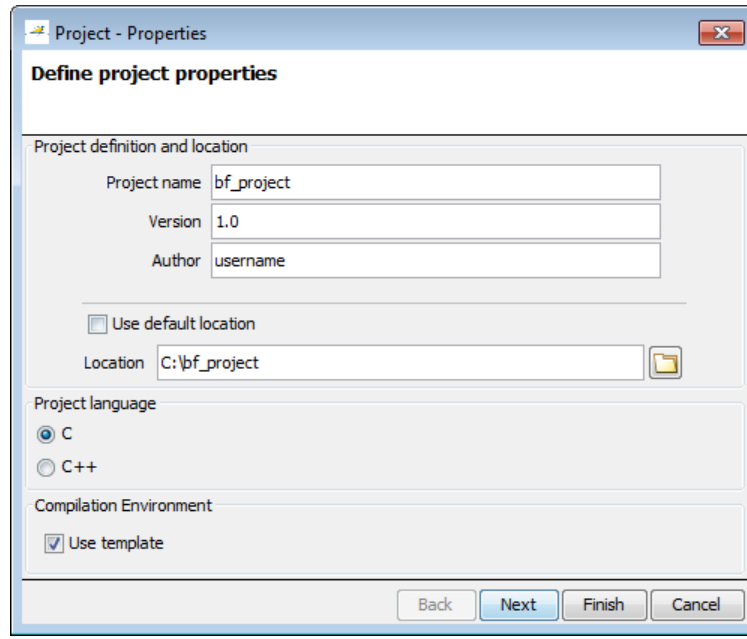
     ```
     MATLAB_ROOT/polyspace/bin/polyspace-bug-finder
     ```

4. In Polyspace Bug Finder, select **File > New Project**. The Project Wizard opens to help you create a new Polyspace project.

5. In the **Project Name** field, enter bf_project as your project name.

6. Clear the **Use default location** check box. Enter the location of the bf_project folder that you created in step 1.

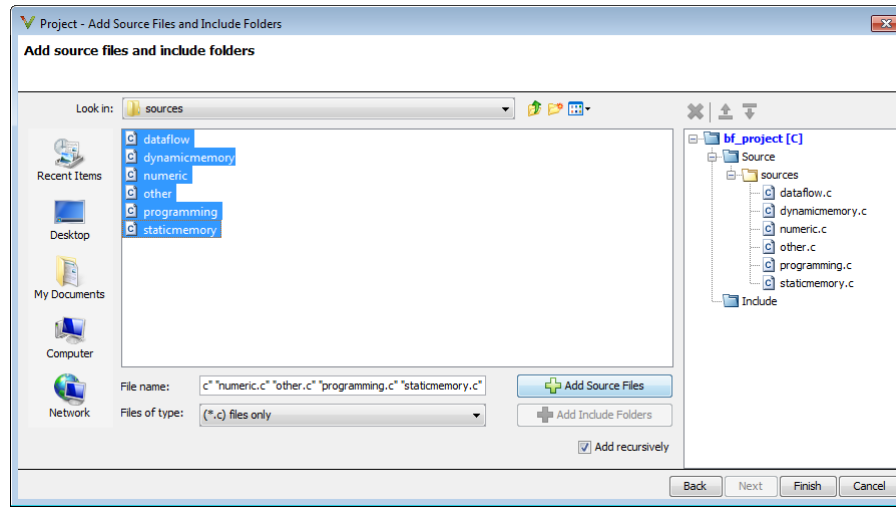
7. Select the **Use Template** check box to speed up the configuration process by using a preconfigured template.



8 Click **Next**.

9 Select **Baseline_C**, a generic template for C coding projects. Click **Next**.

10 Select the `sources` folder and click **Add Source Files** to add the source files to your project.



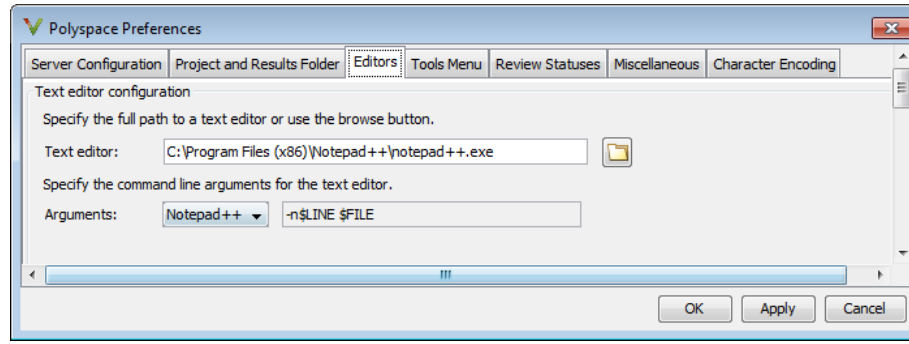
11 Click **Finish**.

Configure Text Editor

Before you start an analysis, configure your text editors through the **Polyspace Preferences** dialog box. You can then view source files directly from the Polyspace user interface.

1 Select **Options > Preferences**.

2 In the Preferences dialog box, select the **Editors** tab.



3 Specify a **Text Editor** to use to view source files from the Project Manager logs. For example:

```
C:\Program Files\Windows NT\Accessories\wordpad.exe
```

4 From the **Arguments** drop-down list, select your text editor to specify the command line arguments for that editor.

If you are using a text editor that is not in the drop-down list, select `Custom`. In the text field to the right of `Custom`, specify the command line arguments for the text editor.

5 Click **OK**.

Configure Coding Rules and Run Analysis

During the project setup, you selected a configuration template. The template already set most analysis options. Now, before running analysis, you specify one additional option for checking coding rules.

1 In the **Coding Rules** pane of the Configuration window, select **Check MISRA C rules** to add coding rules checking to your analysis.

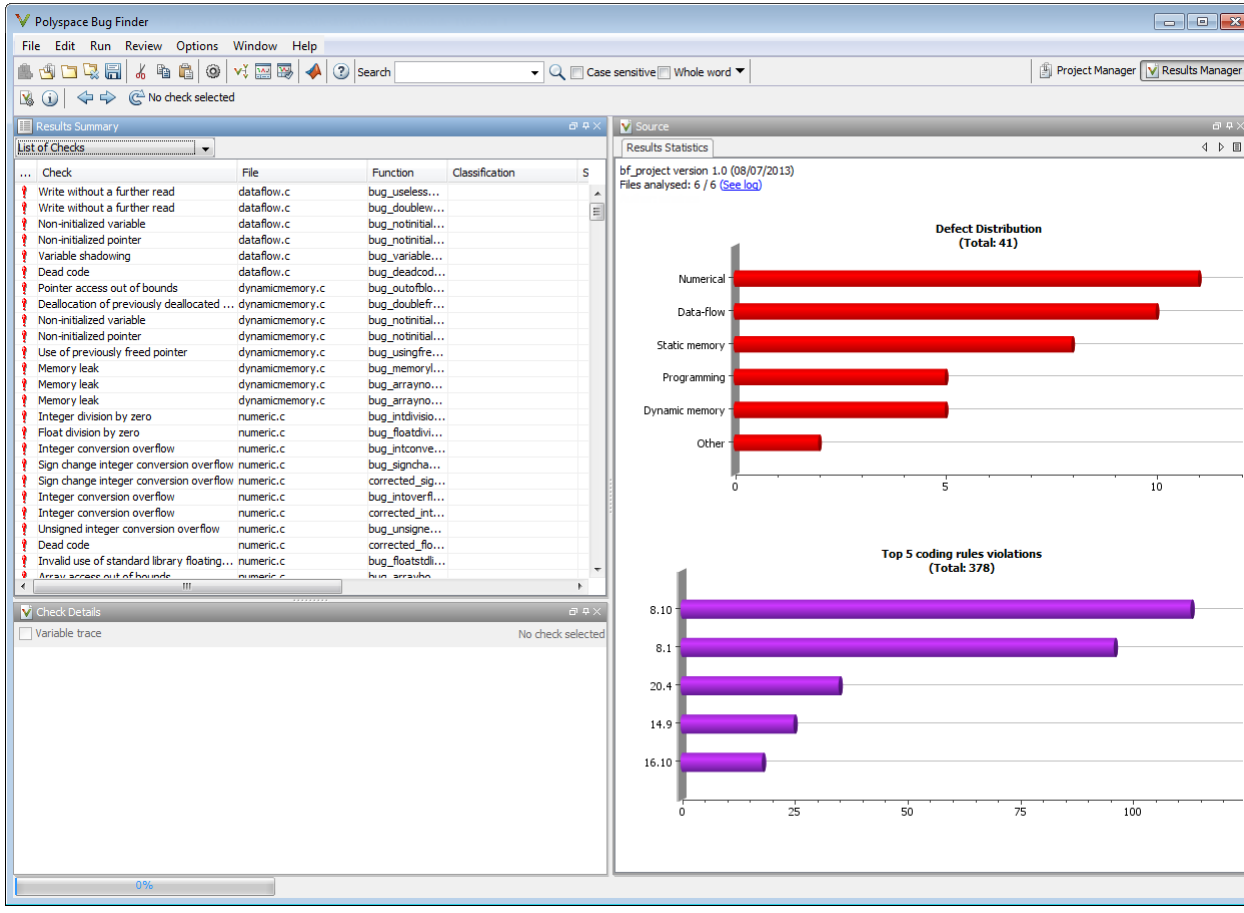
2 From the Polyspace toolbar, select **Run**. You can follow the analysis in the Progress Monitor window.

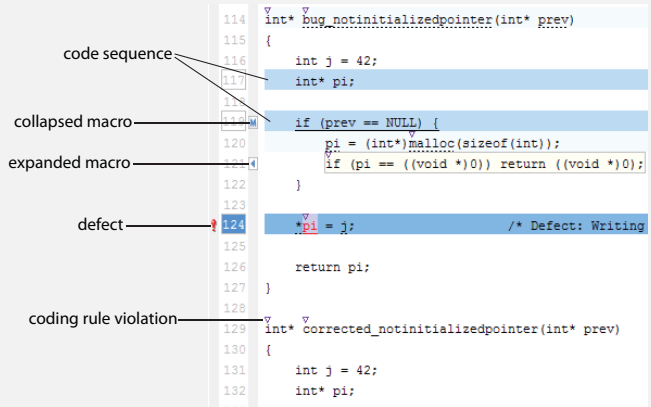
Review Results

1 After the analysis is complete, from the **Project Browser**, double-click `Results > bf_project.psbf`.

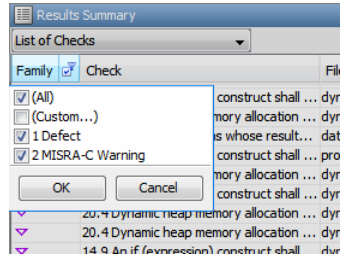
Polyspace switches to the Result Manager perspective and displays the analysis results.

In the Bug Finder Results Manager, you see three windows:


Polyspace Bug Finder
File Edit Run Review Options Window Help
Search
Case sensitive
Whole word
Project Manager
Results Manager
No check selected
Results Summary
List of Checks
... Check File Function Classification S
Write without a further read dataflow.c bug_useless...
Write without a further read dataflow.c bug_doublew...
Non-initialized variable dataflow.c bug_notinitial...
Non-initialized pointer dataflow.c bug_notinitial...
Variable shadowing dataflow.c bug_variable...
Dead code dataflow.c bug_deadcod...
Pointer access out of bounds dynamicmemory.c bug_outofblo...
Deallocation of previously deallocated ... dynamicmemory.c bug_doublefr...
Non-initialized variable dynamicmemory.c bug_notinitial...
Non-initialized pointer dynamicmemory.c bug_notinitial...
Use of previously freed pointer dynamicmemory.c bug_usingfre...
Memory leak dynamicmemory.c bug_memoryl...
Memory leak dynamicmemory.c bug_arrayno...
Memory leak dynamicmemory.c bug_arrayno...
Integer division by zero numeric.c bug_intdivisio...
Float division by zero numeric.c bug_floatdivi...
Integer conversion overflow numeric.c bug_intconve...
Sign change integer conversion overflow numeric.c bug_signcha...
Sign change integer conversion overflow numeric.c corrected_sig...
Integer conversion overflow numeric.c bug_intoverfl...
Integer conversion overflow numeric.c corrected_int...
Unsigned integer conversion overflow numeric.c bug_unsigne...
Dead code numeric.c corrected_flo...
Invalid use of standard library floating... numeric.c bug_floatstdli...
Check Details
Variable trace
No check selected
0%
Source
Results Statistics
bf_project version 1.0 (08/07/2013)
Files analysed: 6 / 6 (See log)
Defect Distribution
(Total: 41)
Numerical
Data-flow
Static memory
Programming
Dynamic memory
Other
0
5
10
Top 5 coding rules violations
(Total: 378)
8.10
8.1
20.4
14.9
16.10
0
25
50
75
100

Window Name	Purpose
Results Summary	See defects found. You can view the results in an ungrouped list, by file/function, or by defect type. In any of these views, you can sort or filter the results using the column headers. Right-click the column header to add or remove columns. You can enter defect-specific comments and justifications using the Status and Comments columns.
Check Details	See more details about a selected defect. The window includes a description of the defect, the offending line of code, and, if applicable, a code sequence to help find the root cause of the defect.
Source	See the defect in the source code. By default, a separate tab appears showing global results statistics. These statistics can provide a big picture of the defect distribution and top coding rule violations. As you select different checks, the source files open. Results are marked in the source code. Red underlined code with a red exclamation point 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. The line of code with the selected defect is highlighted in dark blue. If a code sequence is available, those sequential lines of code are highlighted in light blue and a box outlines their line numbers. 

1. In the Results Summary window, make sure that you select the **List of Checks** setting.
2. Right-click on the column headers and add the **ID** column to the Results Summary window.
3. Hover your cursor over the **Check** column header and select the filter button . Clear `All` and select `5.2 Identifiers in an inner scope shall not...` to view only MISRA C rule 5.2 results.
4. Select the coding rule violation in the `dataflow.c` file. This violation of MISRA C rule 5.2 must be fixed eventually, but can be saved for later. The Results Summary window has a classification, a status, and a comments columns insert annotations that you can refer to later.
5. For the selected result, change the **Status** to `Fix`.
6. In the **Comments** field enter, `Change identifier`.
7. Select **File > Save** to save your annotations.
8. Clear the filter from the **Check** column.
9. Hover your cursor over the **Family** column header (the column of icons) and select the filter button . Clear `MISRA-C Warning` to filter out the coding rule violations. Click **OK**.



   The only results left are the defects.
10. Select the **File** column to sort by file
11. Locate the `Invalid use of == operator` in the `programming.c` file.

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

12 In the next line of code, select the red bracket to highlight the `Array access out of bounds` result.

This defect is an out-of-bounds array access. As the check details state, the array index, `i`, exceeds the array bounds. `i` reaches a value of 8 because of the incorrect equals operator from the previous line. The previous error causes the for-loop to loop nine times instead of four.

Fix Defects and Rerun Analysis

1 Select the `Invalid use of == operator` defect.

2 In the Source Code window, right-click the red code and select **Open Source Code**.

In a text editor, the source code file, `programming.c`, opens to the incorrect line.

3 At line 45 where the invalid operator was found, change the == to =:

```
for (j = 5; j < 9; j++) {
```

4 Save `programming.c`.

5 In the Polyspace environment, return to the Project Manager.

6 Rerun the analysis.

7 Open the new results.

Polyspace automatically imports your previous comments into the new results. In the Results Summary window, find the coding rule that you annotated earlier. You can see the `Fix` status and comment that you entered now imported to the new results.

Find Defects from Simulink

In this section...
"Introduction" on page 2-11
"Create Simulink Model and Generate Code" on page 2-11
"Run Bug Finder Analysis" on page 2-13
"Review Results" on page 2-13

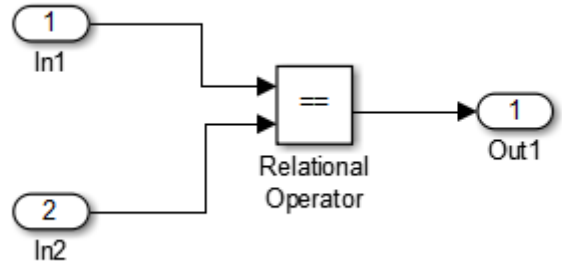
Introduction

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. Generate code.
2. Set up Polyspace configuration options.
3. Run analysis.
4. Review results.

Create Simulink Model and Generate Code

1. Open MATLAB and open Simulink.
2. Create the model below.



3 Select **Code > C/C++ Code > Code Generation Options** to open the Model Configuration window.

4 Set the following options:

Pane	Option	Value
Code Generation	**System target file**	`ert.tlc`
	Generate code only	Selected
Solver	**Type**	`Fixed-step`
	Solver	`discrete (no continuous states)`
	Fixed-step size	`0.1`
Optimization	**Remove root level I/O zero initialization**	Selected
	'Use memset to initialize floats and doubles to 0.0	Not selected
Optimization > Signals and Parameters	**Inline parameters**	Selected

5 Save your model as `bf_model`.

6 From the Simulink menu, select **Code > C/C++ Code > Build Model** to generate code.

Run Bug Finder 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 rule checking`
Open results automatically after verification	On

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

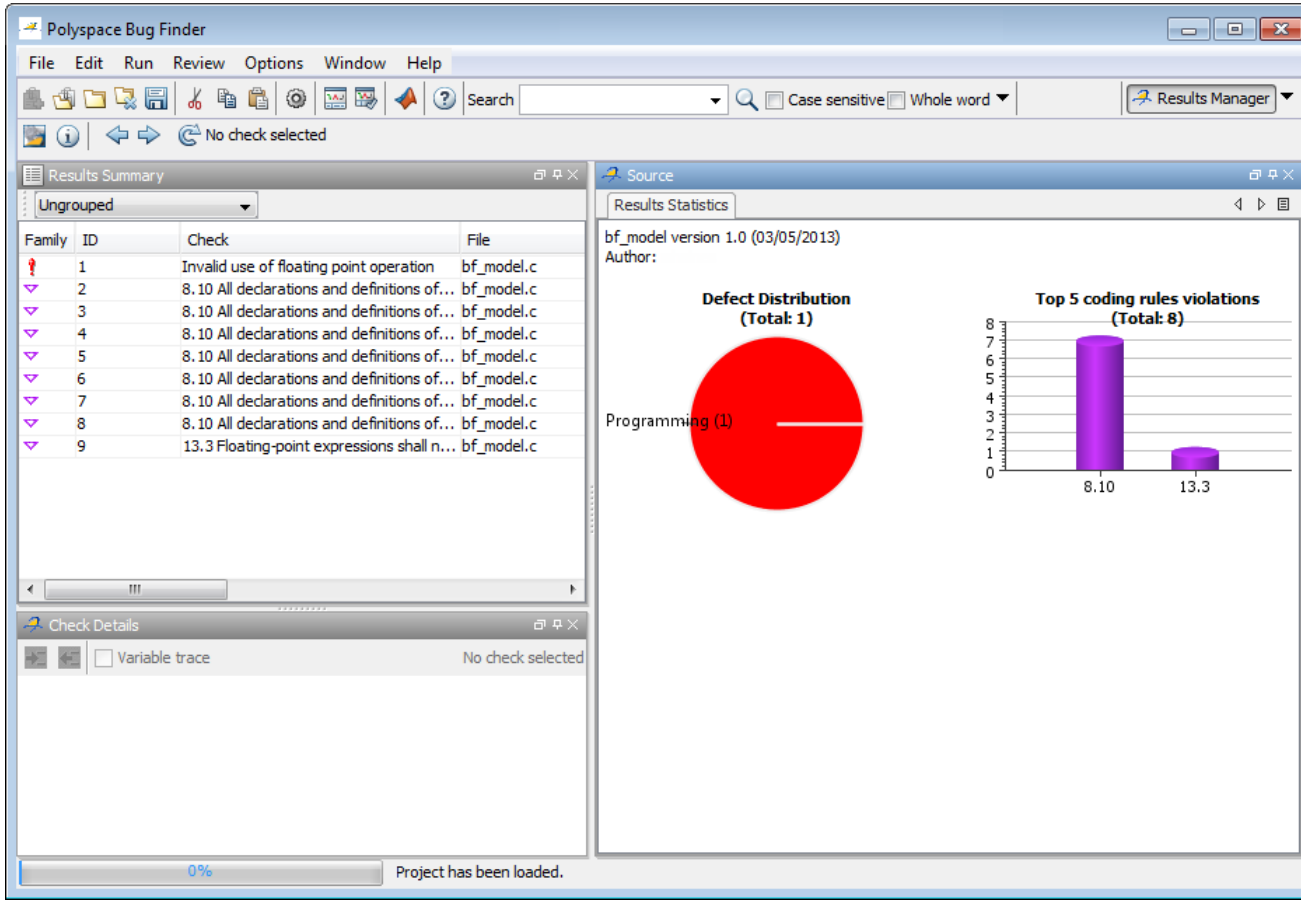
3 From the same pane, apply your changes and click **Run verification** to start the Bug Finder.

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

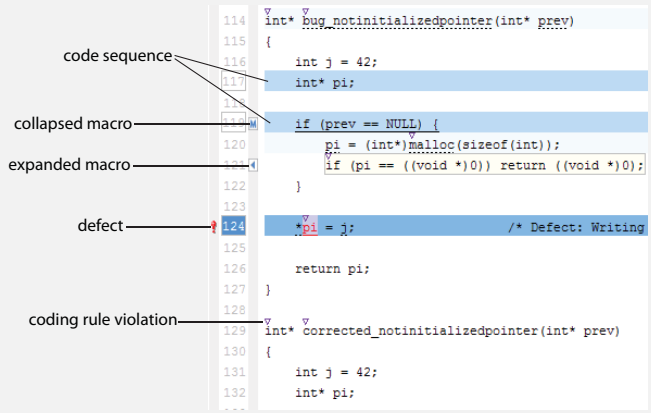
Review Results

1 After the analysis has finished, the results open in the Results Manager window of the Polyspace environment.

In the Bug Finder Results Manager, you see three windows:



Window Name	Purpose
Results Summary	See defects found. You can view the results in an ungrouped list, by file/function, or by defect type. In any of these views, you can sort or filter the results using the column

Window Name	Purpose
	headers. Right-click the column header to add or remove columns. You can enter defect-specific comments and justifications using the Status and Comments columns.
Check Details	See more details about a selected defect. The window includes a description of the defect, the offending line of code, and, if applicable, a code sequence to help find the root cause of the defect.
Source	See the defect in the source code. By default, a separate tab appears showing global results statistics. These statistics can provide a big picture of the defect distribution and top coding rule violations. As you select different checks, the source files open. Results are marked in the source code. Red underlined code with a red exclamation point 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. The line of code with the selected defect is highlighted in dark blue. If a code sequence is available, those sequential lines of code are highlighted in light blue and a box outlines their line numbers. 

For this simple model, Bug Finder found one defect and eight coding rule violations.

2 Select the `Invalid use of floating point operation` defect.

The Source Code pane highlights the faulty generated code. If you look in the Check Details pane, the equality operator between the two inputs is imprecise because both input types are floating point.

3 In the source code above the defect, click `Root\In1`.

In the Simulink model, the associated In1 block is highlighted in blue. This link between the model and the generated code helps you identify the parts of your model that are causing defect.

4 Double-click the highlighted In1 block.

5 In the **Signal Attributes** tab, change the `Data type` to `Int8`. Select **Okay**.

6 Regenerate the code by selecting **Code > C/C++ Code > Build Model**.

7 Rerun the analysis by selecting **Code > Polyspace > Verify Generated Code for > Model**.

The results open automatically in Polyspace. In the new results, the previous defect and an associated MISRA® coding violation have disappeared.

Find Defects from the Eclipse Plug-In

In this section...

"Introduction" on page 2-17

"Run Analysis and Review Results" on page 2-17

Introduction

Before starting a code analysis, you must install the Polyspace Bug Finder plug-in for Eclipse. For instructions see, "Install Polyspace Plug-In for Eclipse IDE" on page 2-26.

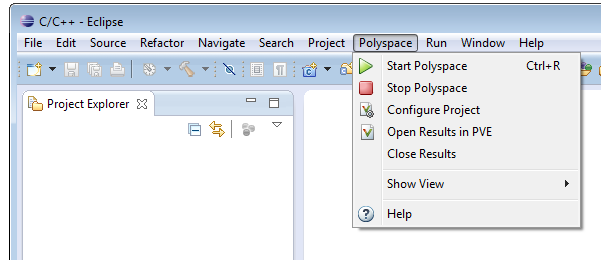
In this tutorial, you analyze a simple code example using Polyspace Bug Finder in Eclipse. A common workflow for code analysis with Polyspace Bug Finder is:

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

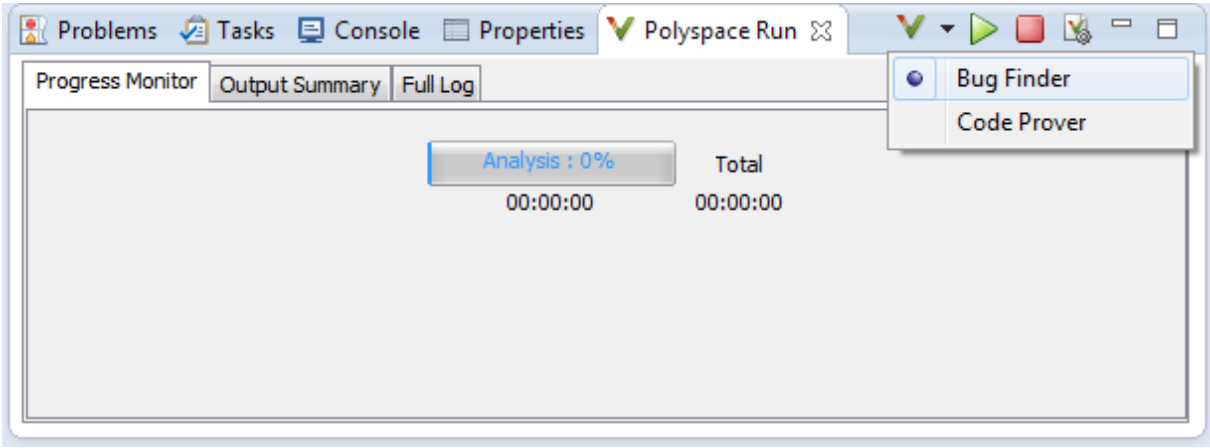
This tutorial follows a shortened version of this workflow.

Run Analysis and Review Results

1. After you install the plug-in, a Polyspace menu appears on the toolbar. Select **Polyspace > Show View > Show Polyspace Run view** to view the Polyspace Run window.



2 In the Polyspace Run window, select the product you want to use.



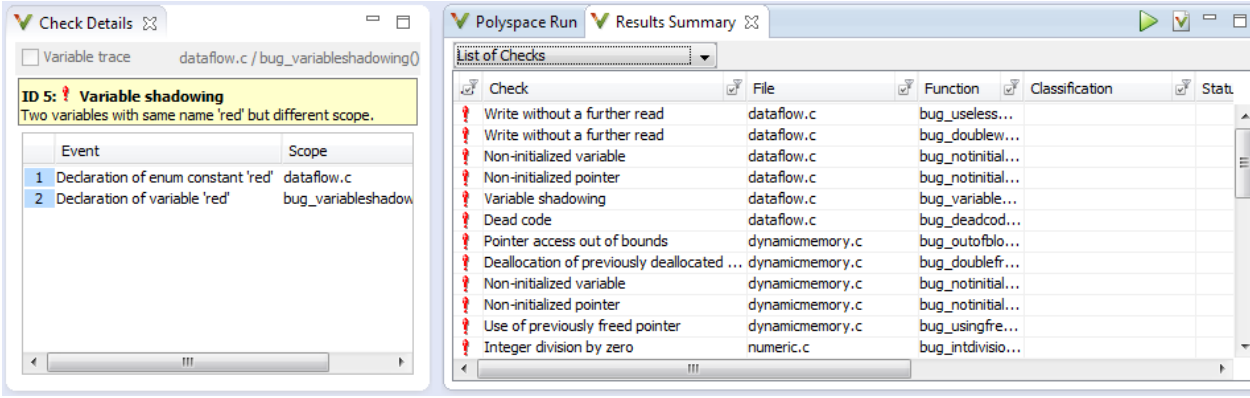
Polyspace will use Bug Finder configuration options and analysis to analyze your Eclipse project.

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

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

As the analysis runs, you can follow the progress in the **Progress Monitor** tab of the Polyspace Run window. If your code has compilation errors that prevent analysis, they appear in the **Output Summary** tab.

4 After the analysis finishes, the Results Summary window appears. As you select different defects, the source code switches to that line number. Details about the error appear in the Check Details window.



5 After fixing bugs or adding code, you can rerun the analysis from the Results Summary window by clicking the rerun button, .

Find Defects from Visual Studio

In this section...
"Introduction" on page 2-20
"Run Analysis in Visual Studio" on page 2-20
"Review Results" on page 2-24

Introduction

Before starting a code analysis, you must install the Polyspace Bug Finder plug-in for Visual Studio®. For instructions see, "Install Polyspace Add-In for Visual Studio" on page 2-29.

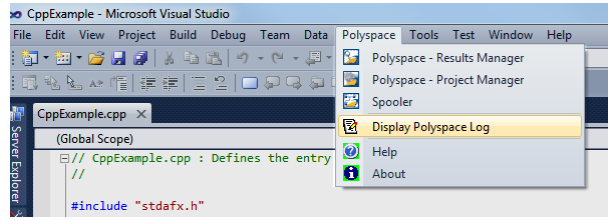
In this tutorial, you analyze a simple code example using Polyspace Bug Finder in Visual Studio. A common workflow for code analysis with Polyspace Bug Finder is:

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.

Run Analysis in Visual Studio

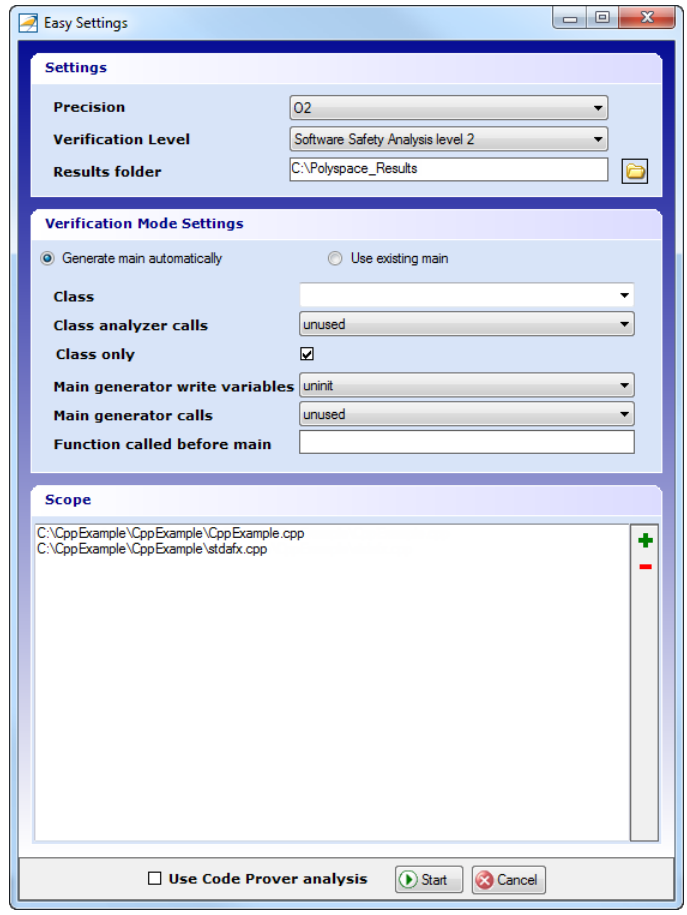
1. After you install the plug-in, a Polyspace menu appears on the toolbar. Select **Polyspace > Display Polyspace Log** to view the Polyspace Log window.



2 In the Visual Studio **Solution Explorer** view, select one or more files that you want to analyze.

3 Right-click the selection, and select **Polyspace Verification**.

The Easy Settings dialog box opens.



4 In the Easy Settings dialog box, you can specify the following options for your analyses:

- Under **Settings**, configure the following:
  - **Precision** — Precision of analysis (`-O`)
  - **Passes** — Level of analysis (`-to`)
  - **Results folder** – Location where software stores analysis results (`-results-dir`)
- Under **Verification Mode Settings**, configure the following:

- **Generate main** or **Use existing** — Whether Polyspace generates a `main` subprogram (`-main-generator`) or uses an existing subprogram (`-main`)
- **Class** — Name of class to analyze (`-class-analyzer`)
- **Class analyzer calls** — Functions called by generated `main` subprogram (`-class-analyzer-calls`)
- **Class only** — Analysis of class contents only (`-class-only`)
- **Main generator write** — Type of initialization for global variables (`-main-generator-writes-variables`)
- **Main generator calls** — Functions (not in a class) called by generated `main` subprogram (`-main-generator-calls`)
- **Function called before main** — Function called before the generated `main` subprogram (`-function-call-before-main`)

- Under **Scope**, you can modify the list of files and classes to analyze.

For information on *how* to choose your options, see "Analysis Options for C++".

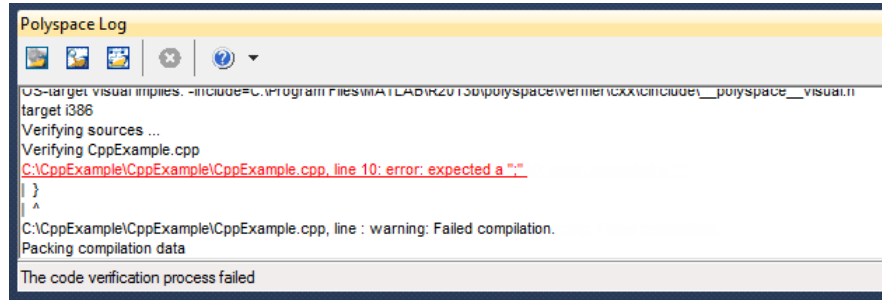
Note In the Configuration window of the Polyspace interface, you configure options that you cannot set in the Easy Settings dialog box. See "Customize Polyspace Options".

---

5. If required, clear the **Use Code Prover analysis** check box.
6. Click **Start** to start the analysis.

   Once you start the software, you can follow its progress in the **Polyspace Log** view.

   Compilation errors are highlighted as links. Click a link to display the file and line that produced the error.



If the analysis is being carried out on a server, use the Polyspace Spooler or Polyspace Metrics to follow the analysis progress. Select **Polyspace > Spooler**, which opens the Polyspace Queue Manager Interface dialog box.

Review Results

Select to open the Results Manager perspective of the Polyspace interface with the last available results. If the analysis has been carried out on a server, download the results before opening the Results Manager perspective.

For information on reviewing and understanding Polyspace Bug Finder results, see “View Results”.

Install Polyspace Plug-In for Simulink

The Simulink plug-in supports the four previous releases of MATLAB. For example, the R2014a version of the Polyspace plug-in supports MATLAB R2012a through R2014a.

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

You can also install the Polyspace plug-in for Simulink on a previous version of MATLAB. You can then use the latest verification software with an older version of Embedded Coder® or TargetLink. However, if you use a cross-version installation of Polyspace and MATLAB, local batch analyses can only be submitted from the Polyspace environment or `polyspaceBugFinder`.

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

To install the Polyspace plug-in on a previous version of MATLAB:

1. Using an account with read/write privileges, open the older version of MATLAB.
2. Change your **Current Folder** to *`matlabroot`*`\polyspace\toolbox\pslink\pslink`. *`matlabroot`* is the Simulink plug-in that you want to install, for example, `C:\Program Files\MATLAB\R2014a`.
3. If you have a previous version of Polyspace installed, execute the `pslinksetup('uninstall')` command to uninstall it. This command does not work with MATLAB R2013b or later (see the preceding Note).
4. Execute the `pslinksetup('install')` command to install the new version of Polyspace.

Install Polyspace Plug-In for Eclipse

In this section...

"Install Polyspace Plug-In for Eclipse IDE" on page 2-26

"Uninstall Polyspace Plug-In for Eclipse IDE" on page 2-28

Install Polyspace Plug-In for Eclipse IDE

You can install the Polyspace plug-in only after you:

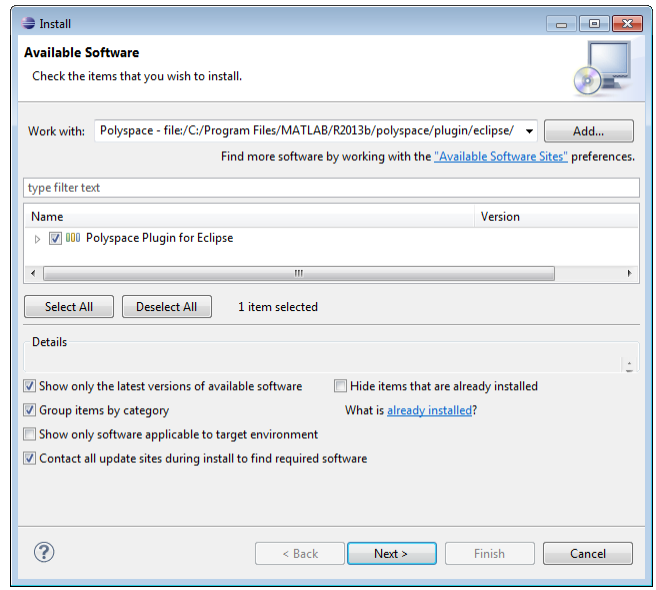
- Install and set up Eclipse Integrated Development Environment (IDE). For more information, see the Eclipse documentation at www.eclipse.org.
- Install Java® 7. See Java documentation at www.java.com.
- Uninstall any previous Polyspace plug-ins. For more information, see "Uninstall Polyspace Plug-In for Eclipse IDE" on page 2-28.

To install the Polyspace plug-in:

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`*`\matlab\polyspace\plugin\eclipse` folder. Then click **OK**.

   `MATLAB_Install` is the installation folder for the Polyspace product, for example:

   ```
   C:\Program Files\MATLAB\R2013b
   ```
6. Click **OK** to close the Add Repository dialog box.
7. On the Available Software page, select `Polyspace Plugin for Eclipse`.



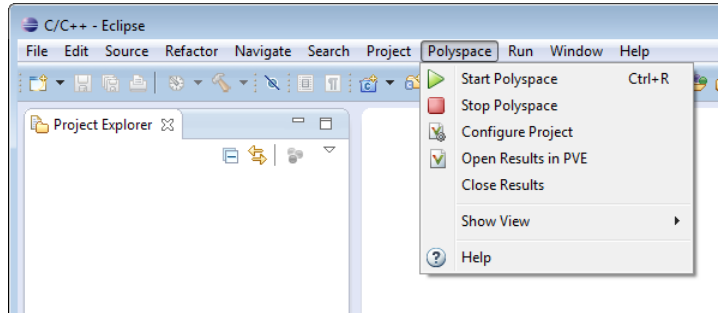
8 Click **Next**.

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

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

Once you install the plug-in, in the Eclipse editor, you'll see:

- A **Polyspace** menu
- A **Polyspace Run** view



Uninstall Polyspace Plug-In for Eclipse IDE

Before installing a new Polyspace plug-in, you must uninstall any previous Polyspace plug-ins:

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

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

Install Polyspace Add-In for Visual Studio

Install Polyspace Add-In for Visual Studio

You can install the Polyspace add-in only after you:

- Install Visual Studio.
- Uninstall any previous Polyspace add-ins. For more information see "Uninstall Polyspace Add-In for Visual Studio" on page 2-30.

To install the Polyspace add-in:

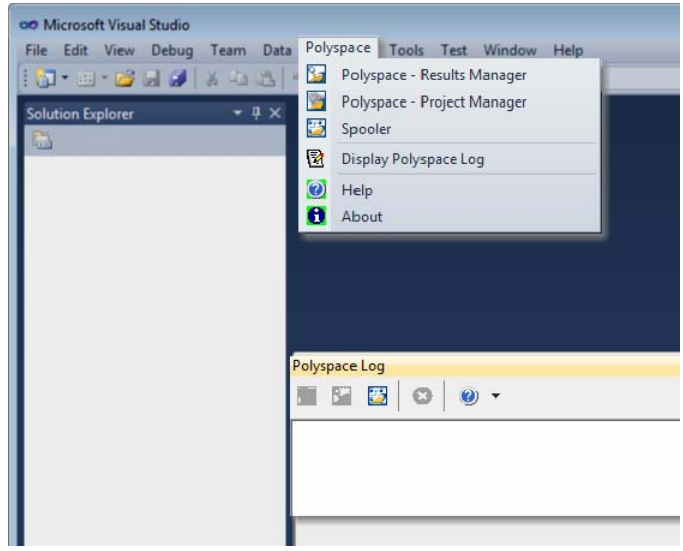
1. In the Visual Studio editor, select **Tools > Options** to open the Options dialog box.
2. Select the **Environment > Add-in/Macros Security** pane to display the list of Visual Studio add-in folders.
3. Select the following check boxes:
   - **Allow macros to run**
   - **Allow Add-in components to load**
4. Click **Add** to open the Browse For Folder dialog box.
5. Navigate to
   MATLAB_Install`\matlab\polyspace\plugin\msvc\`*VS_version*
   - *MATLAB_Install* is the installation folder for the Polyspace product, for example:

     ```
     C:\Program Files\MATLAB\R2013b
     ```

   - *VS_version* corresponds to the version of Visual Studio that you have installed, for example, `2010`.
6. Click **OK** to close the Browse for Folder dialog box.
7. To close the Options dialog box, click **OK**.

You must restart Visual Studio for the changes to take effect. After you install the add-in, the Visual Studio editor has:

- A **Polyspace** menu
- A **Polyspace Log** view



Uninstall Polyspace Add-In for Visual Studio

Before installing a new Polyspace add-in, you must uninstall any previous Polyspace add-ins.

1. In the Visual Studio editor, select **Tools > Options** to open the Options dialog box.
2. Select the **Environment > Add-in/Macros Security** pane to display the list of Visual Studio add-in folders.
3. Select the Polyspace add-in and select **Remove**.
4. To close the Options dialog box, click **OK**.

   You must restart Visual Studio for the changes to take effect.

3

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 3-9
"Monitoring an Analysis" on page 3-11
"Viewing Polyspace Results" on page 3-11
"Locating Faulty Code in Rhapsody Model" on page 3-12
"Template Configuration Files" on page 3-14

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.

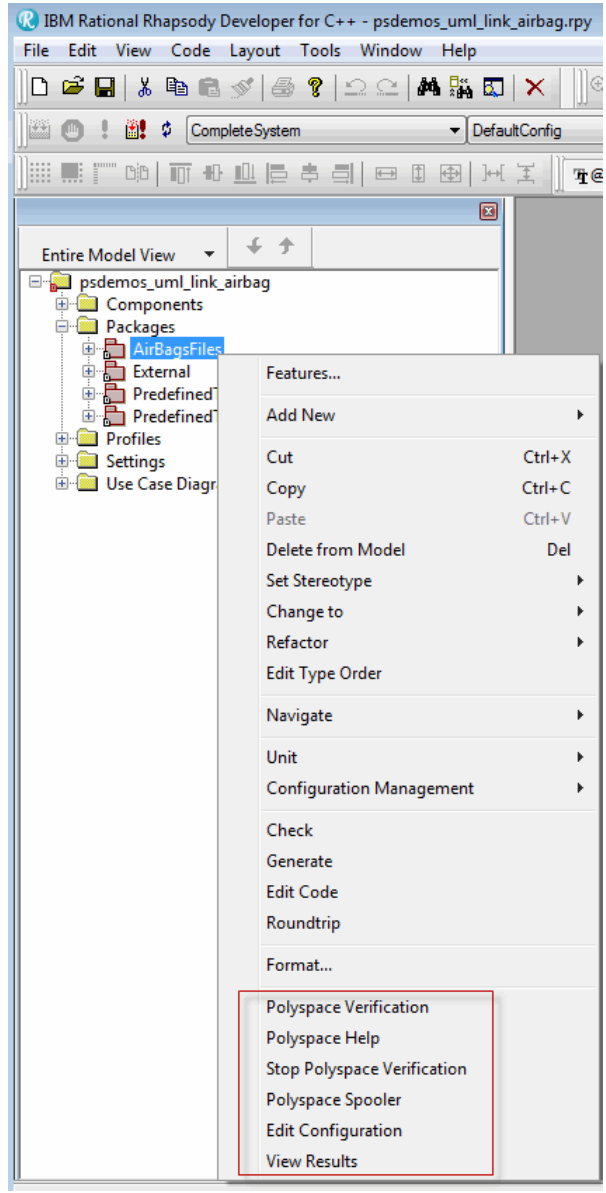
- If required, specify Polyspace configuration options in the Polyspace environment. See “Configuring Analysis Options” on page 3-7.
- Specify the `include` path to your operating system (environment) header files and run an analysis. See “Running an Analysis” on page 3-9 and “Monitoring an Analysis” on page 3-11.
- View results, analyze errors, and locate faulty code within model. See “Viewing Polyspace Results” on page 3-11and “Locating Faulty Code in Rhapsody Model” on page 3-12.

Adding Polyspace Profile to Model

Before you try to access Polyspace features, you must add the Polyspace profile to your model :

1. In the Rhapsody editor, select **File > Add Profile to Model**. The Add Profile to Model dialog box opens.
2. Navigate to the folder *`Polyspace_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 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.

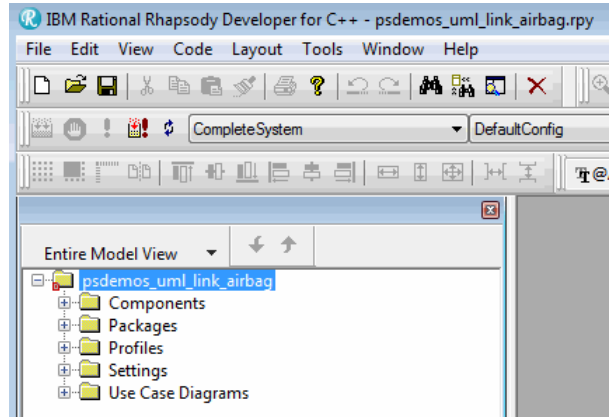
To install the 32-bit Polyspace version, from a DOS command window, run the following command:

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

Accessing Polyspace Features

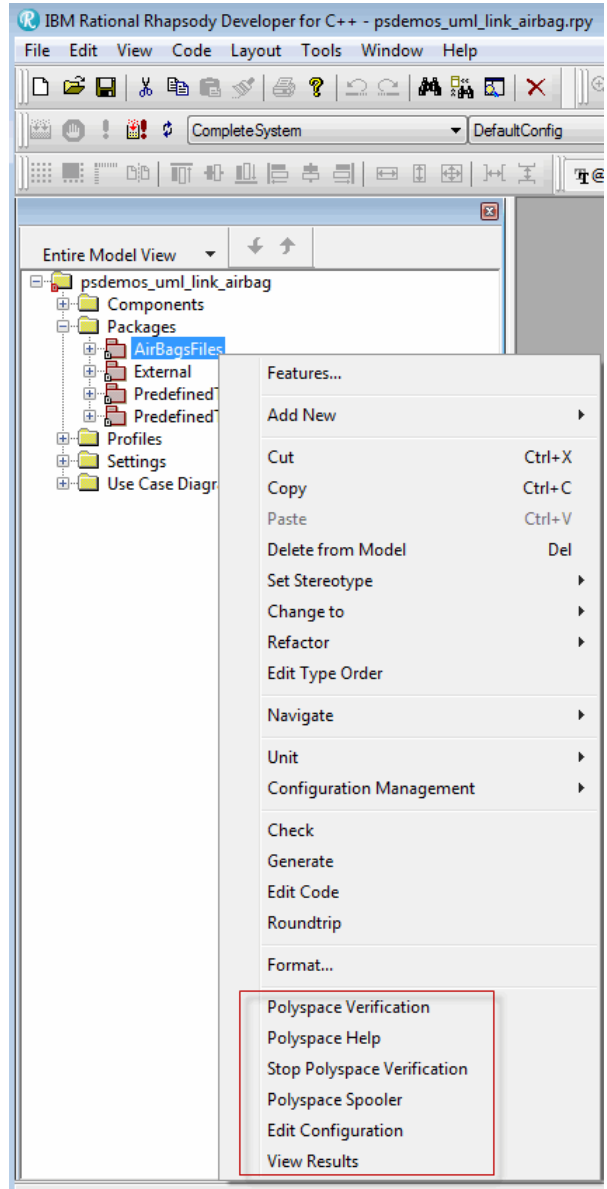
To access Polyspace features in the Rhapsody editor:

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:

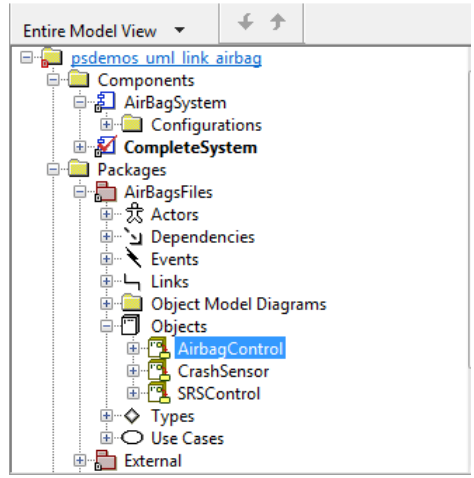
- **Polyspace Verification** — Start analysis. See "Running an Analysis" on page 3-9.
- **Polyspace Help** — Open help.
- **Stop Polyspace Verification** — Stop client-based analysis. See "Running an Analysis" on page 3-9.
- **Polyspace Spooler** — Open Polyspace Queue Manager. See "Monitoring an Analysis" on page 3-11.
- **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.

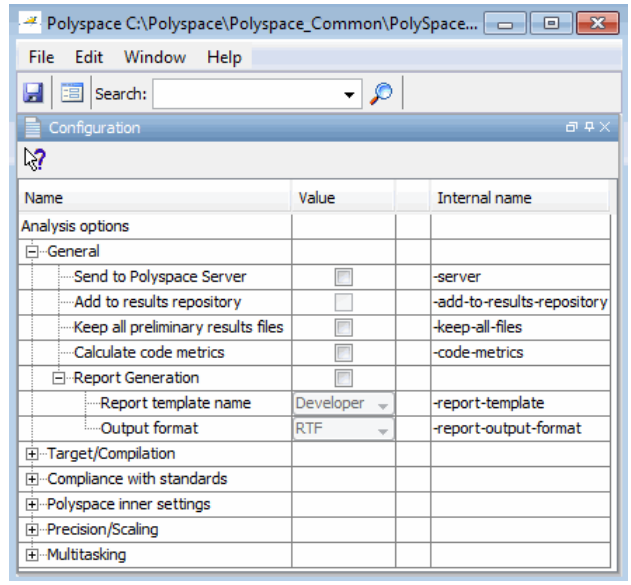
Configuring Analysis Options

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:

- **Target operating system** (`-OS-target`)
- **Dialect** (`-dialect`)
- **Include Folders** (`-I`) — Path to your operating system (environment) header files.

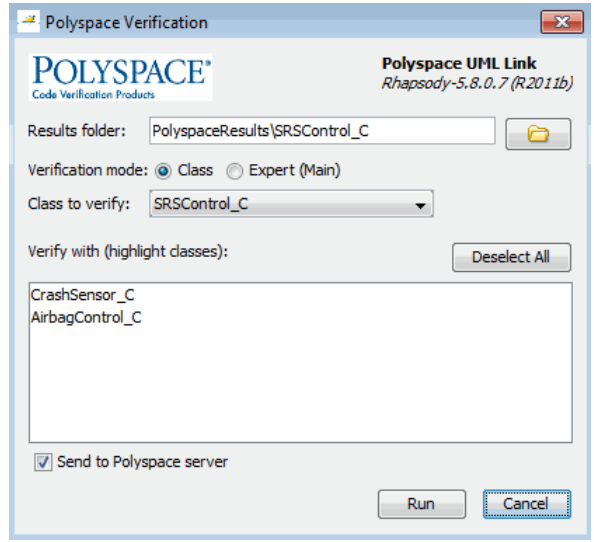
4 To save your options, in the top left corner, click the disk button.

For information on how to choose your options, see "Analysis Options for C" or "Analysis Options for C++".

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.

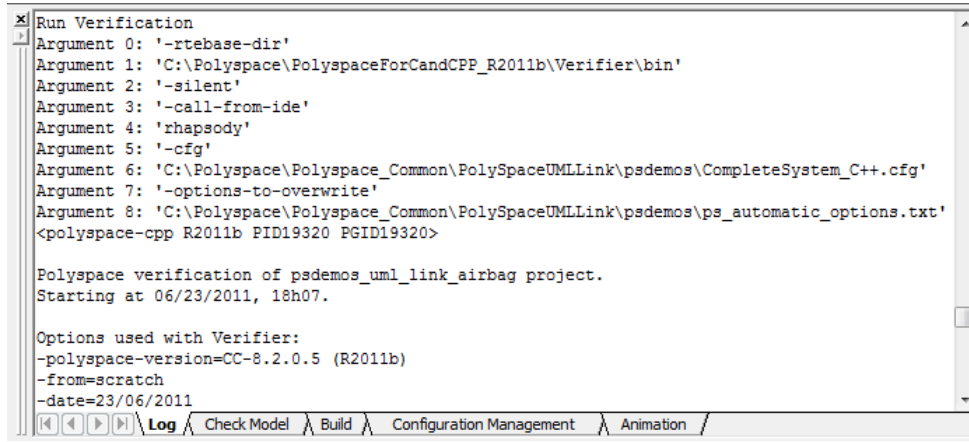
2 In the Results folder field, specify a location for your analysis results.

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**.

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

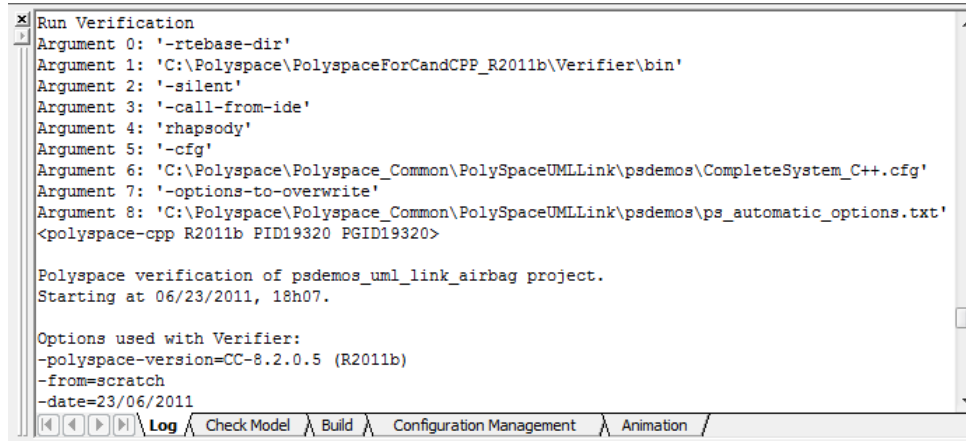


If your analysis is client-based, you can stop your analysis. In the **Entire Model View**, right-click, for example, a package or a class. From the context menu, select **Stop Polyspace Verification**.

To stop an analysis on the Polyspace Server, use the Polyspace Queue Manager. See “Monitoring an Analysis” on page 3-11.

Monitoring an Analysis

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



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 Spooler** to display the Polyspace Queue Manager (or Spooler). Use the Polyspace Queue Manager to manage jobs running on a Polyspace Server.

For more information, see “Verification Management”.

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 results in the Results Manager perspective.

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

Declarations for C Functions Without Arguments

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

```
void my_function()
```

rather than:

```
void my_function(void)
```

This can result in the following Polyspace compilation error:

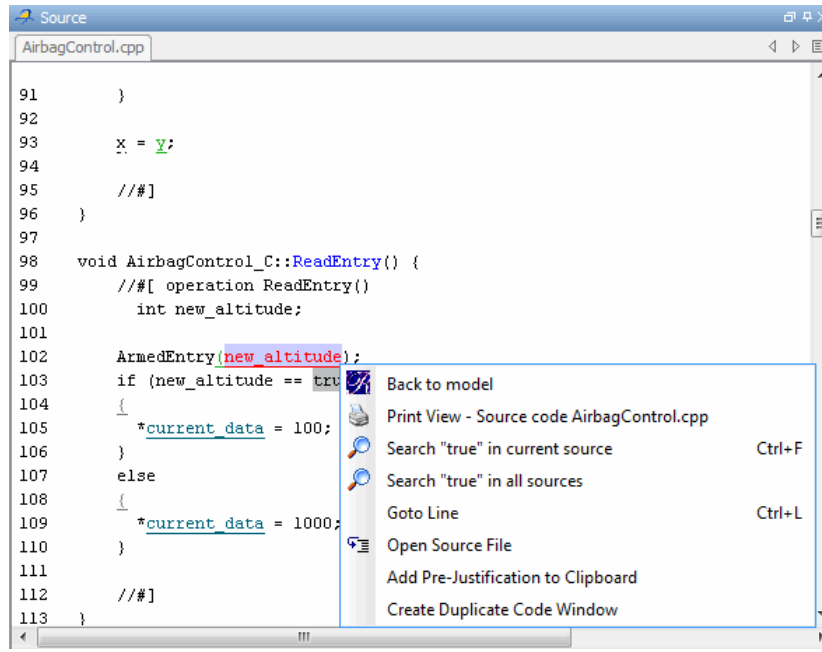
```
Fatal error: function 'my_function' has unknown prototype.
```

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

Locating Faulty Code in Rhapsody Model

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

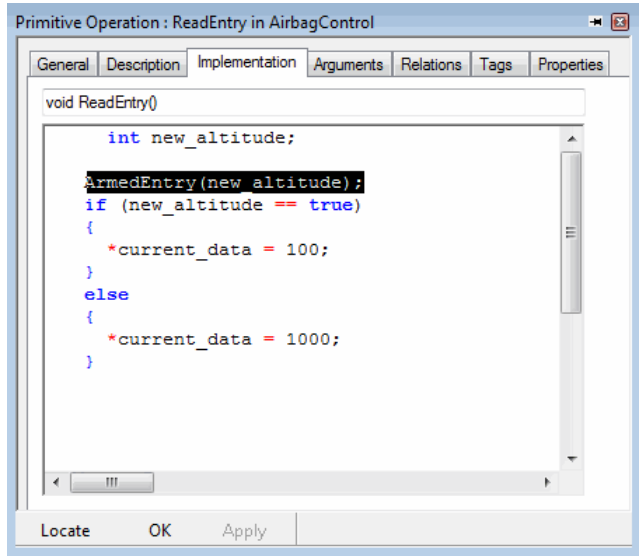
1. In the Results Manager perspective of the Polyspace environment, navigate to an error, for example, a non-initialized variable at line 102 of `Airbag Control_C`.
2. In the Source pane, right-click the error. From the context menu, select **Back to model**.



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.