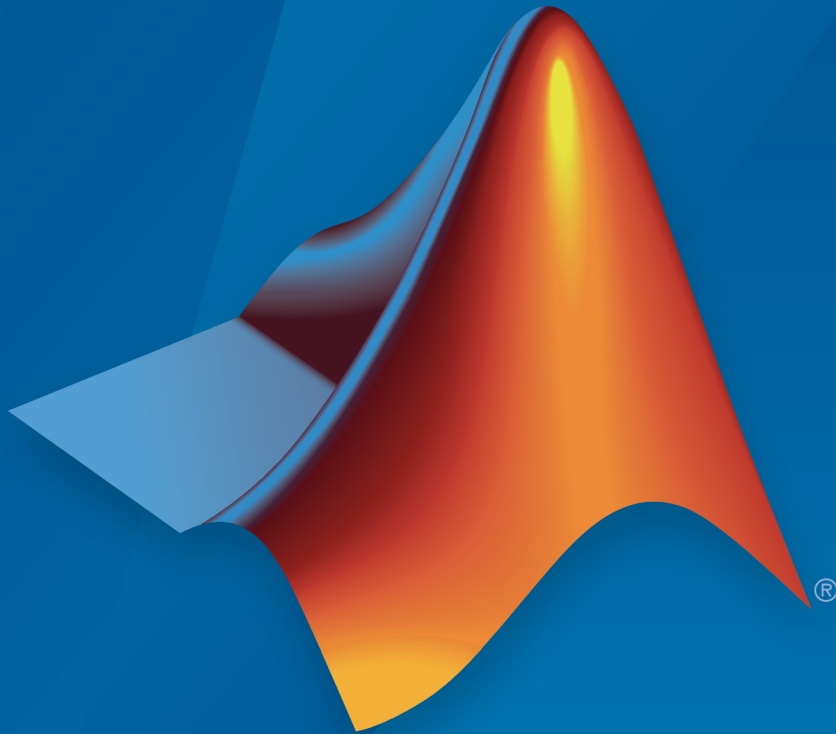# Polyspace® Bug Finder™

Getting Started Guide

MATLAB®&SIMULINK®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

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

Contents

Polyspace UML Link RH

3

Installation and Configuration

4

About Polyspace Bug Finder

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: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
- 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.

Choose Different Options

Configuration Setup

Choose Initial Options
using sample code

Run Bug Finder

Review Results
fit requirements?

No

Continue Coding

Coder

Set Up Project
source files configuration

Run Bug Finder

Fix Issues
triage, review results

Modeler

Set Up Project
code generation configuration

Generate Code

Run Bug Finder
on mixed code

Run Compliance Checker

Review Results
triage, fix

Verifier

Set Up Project
source files, report options, configuration

Run Bug Finder

Report Results to Code Author

Polyspace and the Software Development Cycle

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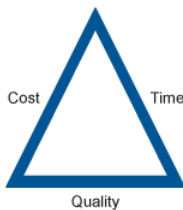
Changing the requirements for one of these variables impacts the other two.

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.
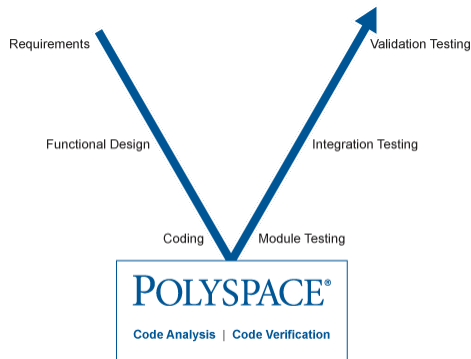
Polyspace Verification in the Development Cycle

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 describing workflows, tasks, concepts, analysis options, checks, and functions.

Access Documentation

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

- Polyspace interface — Select **Help** > **Help**.
- Simulink plug-in — Select **Code** > **Polyspace** > **Help**.
- Eclipse plug-in — Select **Polyspace** > **Help**.
- Visual Studio® add-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 Summary** pane, select a Polyspace check.
2 In the **Check Details** pane, select .
3 Look in the **Contextual Help** pane to see more help for that check.

Related Examples

- "Configure Polyspace Analysis Options and Properties"
- Customize Analysis Options from Eclipse
- Customize Analysis Options from Visual Studio

Tutorials

Find Defects from the Polyspace Environment

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

Set Up Files and Open Polyspace Bug Finder

Before you begin an analysis, set up the source files and create a new Polyspace project.

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

1 In a writable location, 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 Check and make sure the sources folder is writable.

4 Open Polyspace Bug Finder.

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

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

    Where *ReleaseName* is the installed version of Polyspace.

- Desktop shortcut: if you created shortcuts during installation, select the Polyspace Bug Finder icon from the desktop.

- DOS command-line:

    MATLAB_ROOT\polyspace\bin\polyspace-bug-finder

- UNIX® command-line:

    MATLAB_ROOT/polyspace/bin/polyspace-bug-finder

- MATLAB command-line:

    polyspaceBugFinder

Set Up Project

This section sets up a project and for your source files and analysis options.

1 Select **File** > **New Project**. The Project Wizard opens to help you create a new Polyspace project.

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.
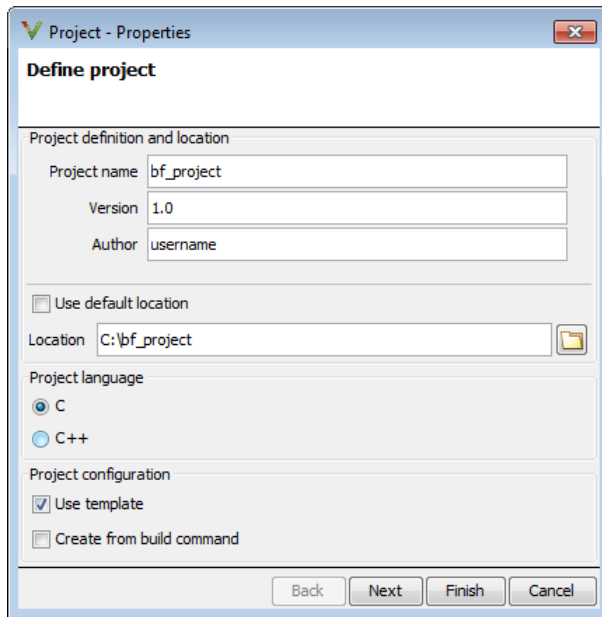
5 Select **GCC_C**, a template for C coding projects which compile with GCC. Click **Next**.

6 Select the `sources` folder and click **Add Source Files** to add the source files to your project.
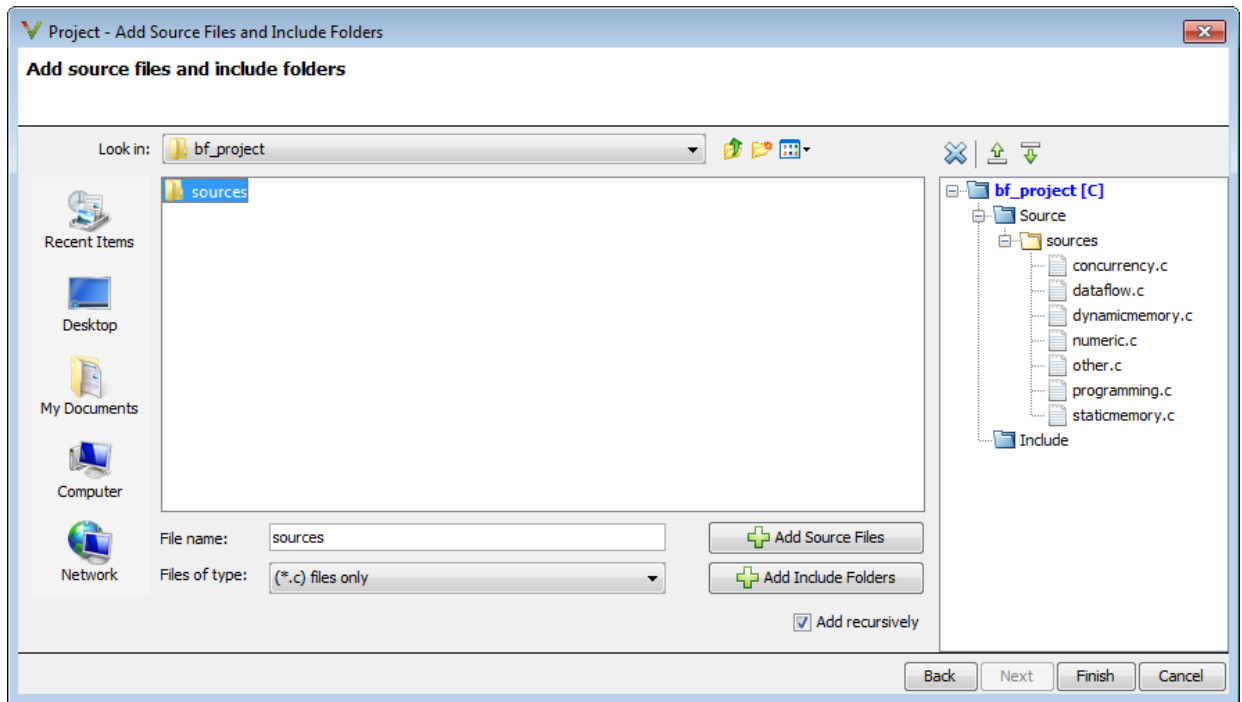
You can also select folders containing header files or other include files. However, this code example does not require include files.

7 Click **Finish**.

Configure Options

During the project setup, you selected a configuration template. The template sets the basic analysis options. Before running the analysis, you can specify additional options for checking coding rules and defining your environment.

1 In the Configuration window, select the **Coding Rules** pane.

2 Select **Check MISRA C:2012** to add coding rules checking to your analysis.

3 Select the **Target & Compiler** node.

The Target and Compiler options provide Polyspace with more information about your code and how it compiles.

When you analyze your own code, it is important to include all the required files during project set up and set your analysis options correctly. If you do not, Polyspace will be unable to compile and analyze your code. Options to consider are:

- **Target & Compiler** > **Target operating system**, which provides Polyspace with the `#define`.
- **Target & Compiler** > **Target processor type**, which sets the size of your data types for the analysis.
- **Target & Compiler** > **Dialect**, which enables different language extensions.
- **Macros** > **Preprocessor definitions**, which is where you enter your compilation flags.
- **Multitasking** options, which are used when analyzing multitasking code.

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

Run Analysis

1   From the Polyspace toolbar, select Run.

    Polyspace compiles and analyzes your project.

2   Follow the progress of the analysis in the **Output Summary** window.

    If your project runs into errors, use the troubleshooting help for Polyspace Bug Finder and Polyspace Code Prover to diagnose the problem.

After the analysis is complete, Polyspace switches to the results panes and displays the analysis results.

Review Results

1   On the **Results Summary** pane, select **Group by** > **None**.
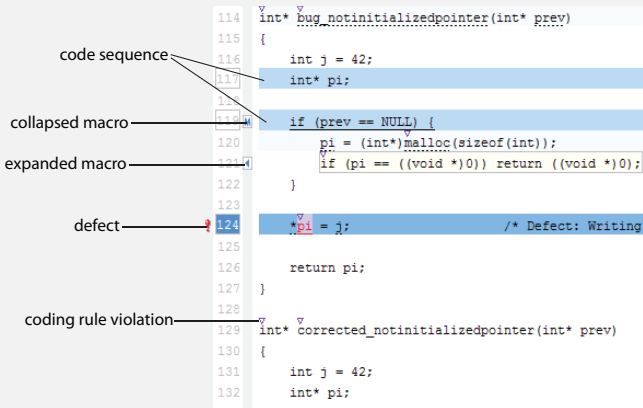
    You see a flat list of defects.

2   Select a defect.

    When reviewing results, you see three panes:

Window Name	Purpose
Results Summary	List of results 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 using the **Status** and **Comments** columns.
Check Details	Details about a selected defect. The window includes a description of the defect and, if applicable, a code sequence to help find the root cause of the defect.

Window Name	Purpose
Source	See the defect in the source code. 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. 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 On the **Results Summary** pane, right-click a column header. Select **ID**.

2 On the **Check** column header, select the filter button ⬚.

    a Clear All.

b Select `5.3 An identifier declared in an inner scope...` to view only MISRA C rule 5.3 results.

3 Select the coding rule violation in the `dataflow.c` file. This violation of MISRA C rule 5.3 must be fixed eventually, but can be saved for later.

4 In the **Results Summary** pane, scroll to the right to view the **Classification**, **Status**, and **Comment** column.

5 For the selected coding rule violation:

a Set **Classification** to `Low`.

b Set **Status** to `Fix`.

c In **Comments**, enter `Change identifier`.

6 Select **File** > **Save** to save your annotations.

7 Clear the filter from the **Check** column.

8 On the **Family** column header, select the filter button ![filter]. The column lists a **!** for defects and purple inverted triangles for coding rule violations.

9 Clear `MISRA C:2012` to filter out the coding rule violations. Click **OK**.

The only results listed are the defects.

Fix Defects and Rerun Analysis

1   Locate the **Invalid use of == operator** defect 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 `=`.

2   On the **Source** pane, select the red bracket in the next line of code to highlight the **Array access out of bounds** defect.

    This defect is an out-of-bounds array access. As the **Check Details** state, the array index, `i`, exceeds the array bounds. If you hover over the red `i`, the tooltip shows that `i` reaches a value of 8 because of the incorrect equals operator from the previous line. The previous error causes the `for` to loop from 0 to 8 instead of 0 to 3.

3   Right-click the bad code in the **Source** pane and select **Open Editor**.

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

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

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

5   Save `programming.c`.

6   Re run ▷ Run the analysis.

    Polyspace automatically imports your previous comments into the new results. In the Results Summary window, find the coding rule that you annotated earlier (`5.3 An identifier declared in an inner scope...`). You can see the classification, status, and comment that you entered now imported to the new results.

    Additionally, the **Invalid use of == operator** defect and the corresponding **Array access out of bounds** defect does not appear in the **Results Summary** because of your fix.

Find Defects from Simulink

In this section...

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
   ```
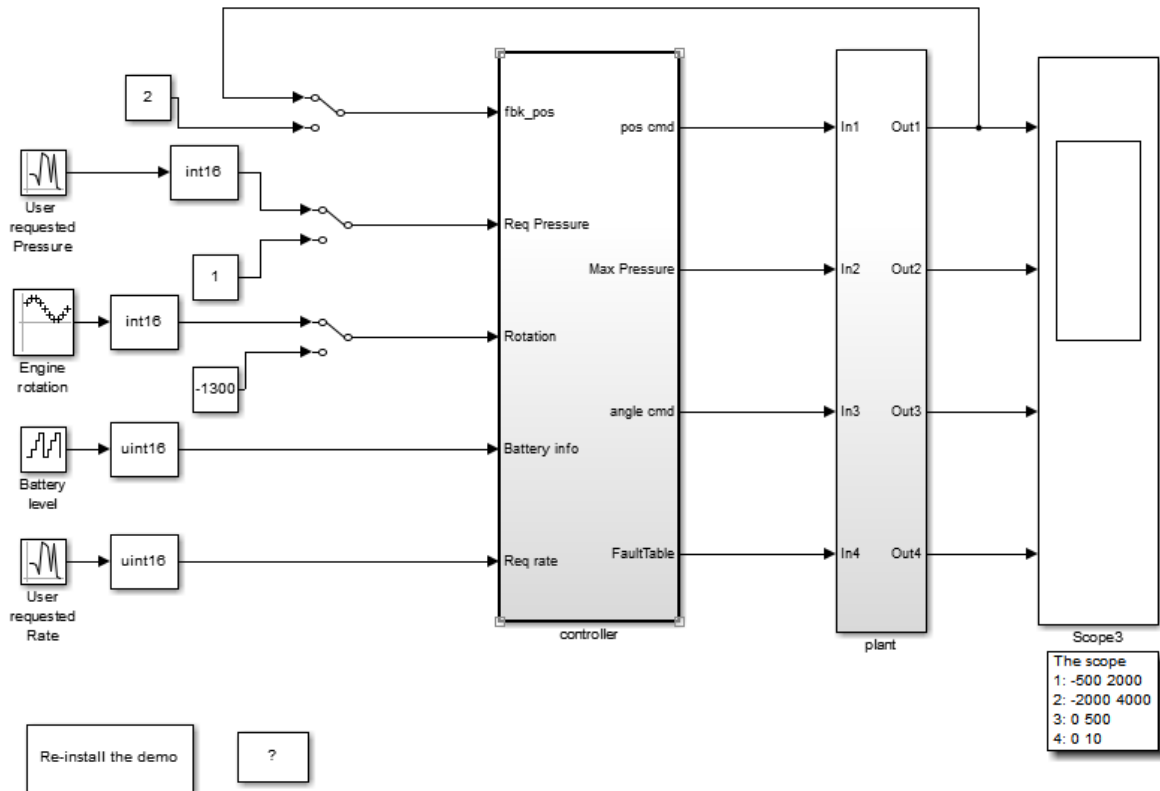
Copyright 2010-2012 The MathWorks, Inc.

2 Double-click the Re-install the demo block to install an S-Function.

3 Right-click on the controller subsystem and select **C/C++ Code** > **Build This Subsystem** to generate code for the controller subsystem.

4 In the Build code for Subsystem: controller window, select **Build**.

Note: The code generation options for this model are already set. For 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 AGC checking`

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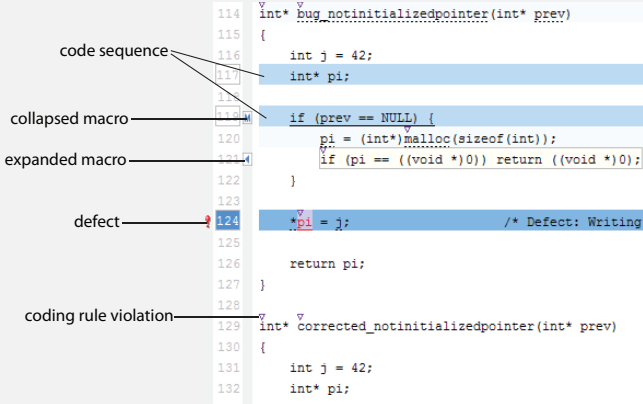
After the analysis has finished, open and review the results in the Polyspace interface.

1   Right-click on the controller subsystem and select **Polyspace** > **Open Results**.

2   Select the `Integer division by zero` result.

In Bug Finder, you see three windows:

Window Name	Purpose
Results Summary	List of results 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 using the **Status** and **Comments** columns.
Check Details	Details about a selected defect. The window includes a description of the defect and, if applicable, a code sequence to help find the root cause of the defect.

Window Name	Purpose
Source	See the defect in the source code. 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. 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.

3 Look at the Check Details pane. The descriptions 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 operator) contains zero. The possibility of this value causes the `Integer division by zero` defect.

5 In the Check Details, select the traceback lines starting at the bottom to see where the value of `controller_B.Cumulatedangle` is assigned.

In event 2 of the traceback, `controller_B.Cumulatedangle` is assigned the value of `tmp`. The if/else statements that assign `tmp` a value does not check if `tmp` is zero.

Fix Model and Rerun Analysis

This section shows you how to fix the `Integer division by zero` defect you examined above. 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.
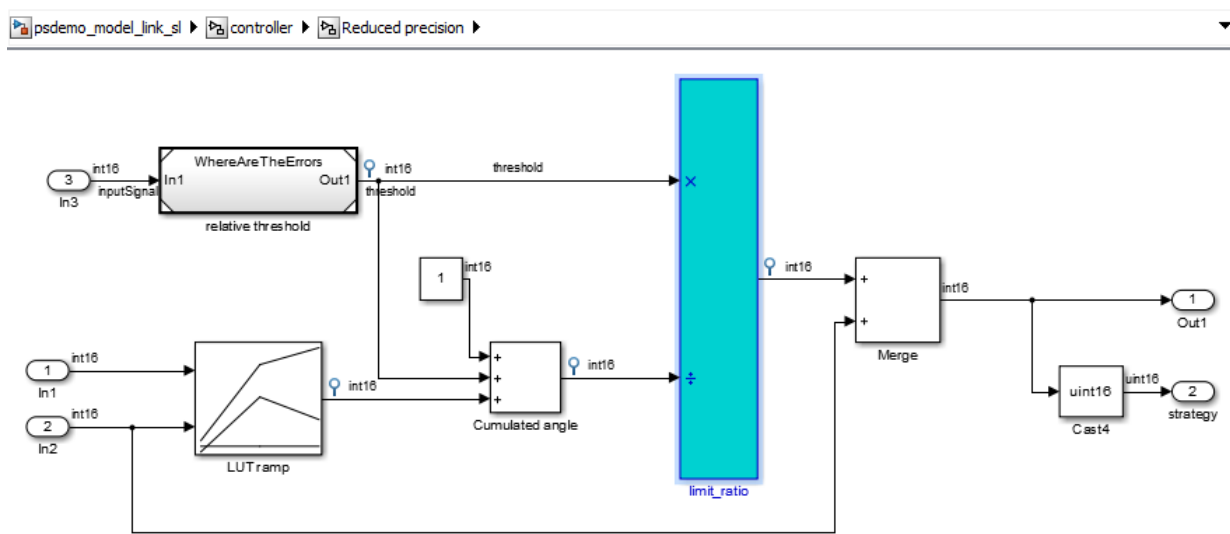
This link between the model and the generated code helps you identify the parts of your model that are causing defect. That way you can fix the defects in the model rather than the code.

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 to u2 ~= 0.

4 Connect the Switch block to the Cumulated angle signal and the constant 1 block such that if the Cumulated angle is not equal to zero, the Cumulated angle is output. If the signal is equal to zero, the 1 is output.

The Reduced precision subsystem should look like this:
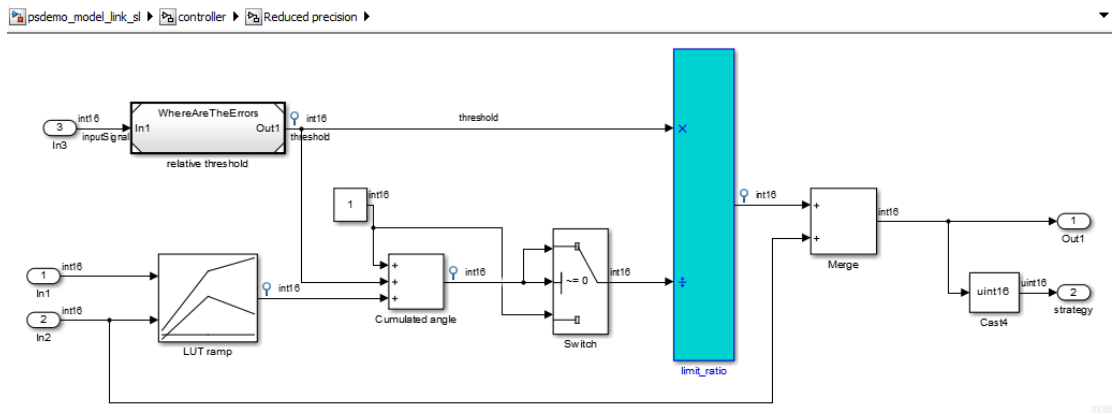
5 Regenerate the code for the controller subsystem. You might need to save a new version of the model.

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 Plug-In

In this section...
"Introduction" on page 2-18
"Run Analysis and Review Results" on page 2-18

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 4-3.

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.
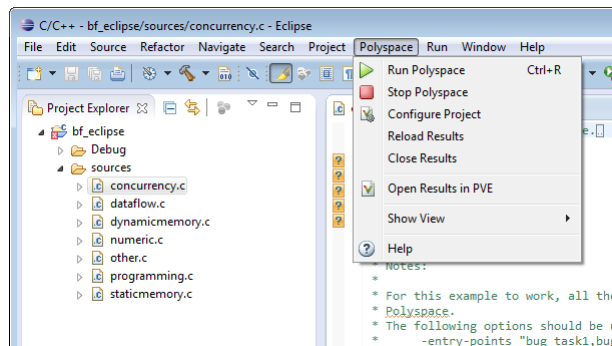
2   In the Polyspace Run window, select `Bug Finder` under the product icon.
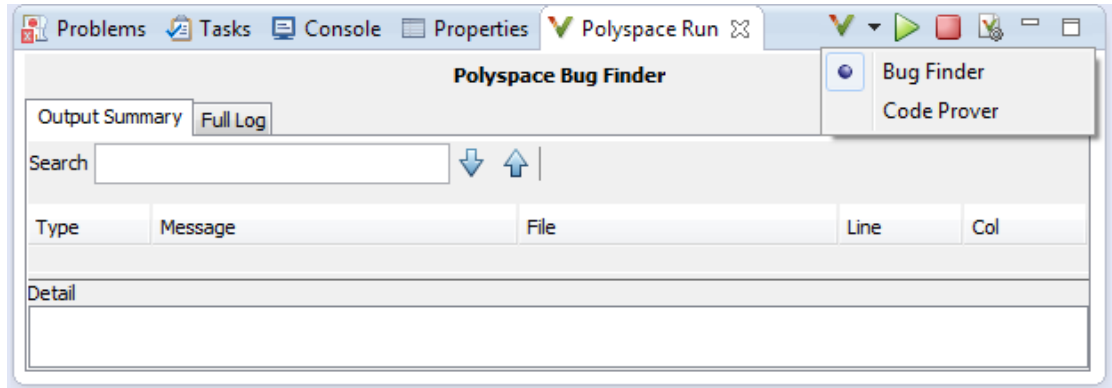
By selecting `Bug Finder`, Polyspace uses Bug Finder configuration options and analysis to analyze your Eclipse project.

3   In the **Eclipse 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 only analyze that file.

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

4   After the analysis finishes, the **Results Summary** and **Check Details** windows appear. Move the Check Details window so you can view both windows at the same time.
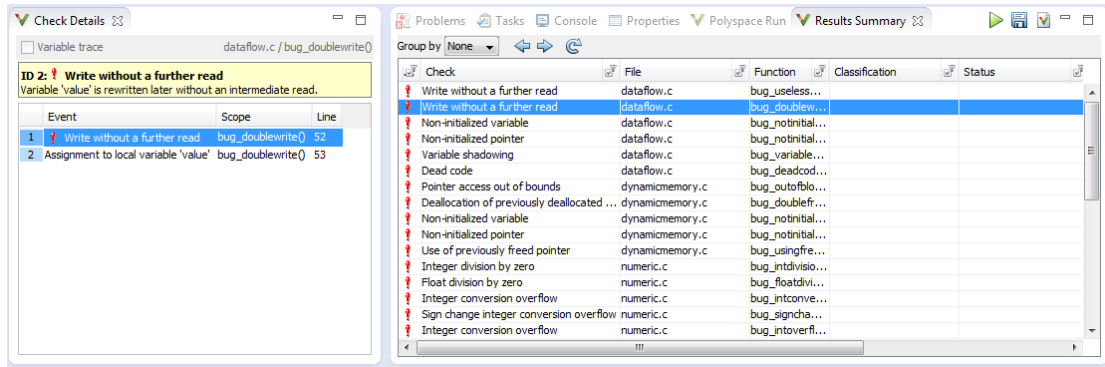
As you select different defects, the source code switches to that line number and details about the defect appear in the Check Details window.

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 2-21
"Run Analysis in Visual Studio" on page 2-21
"Open and 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 4-6.

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.
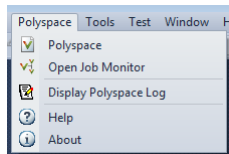
2 In the Visual Studio **Solution Explorer** view, select one or more files that you want to analyze.
3 Right-click the selection, and select **Polyspace Verification**.

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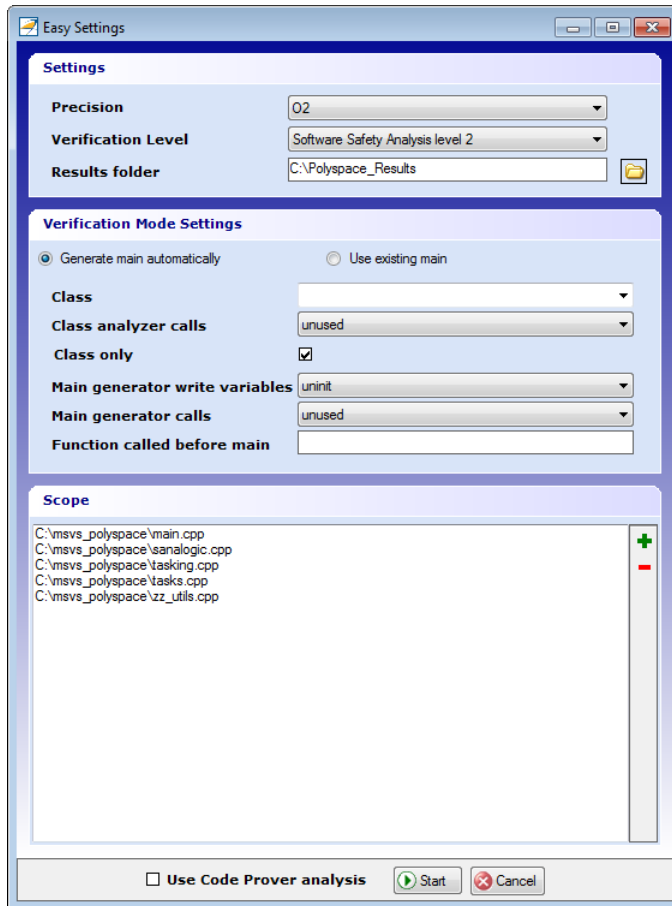
4 In the Easy Settings dialog box, you can specify the following options for your analyses:

- Under **Settings**, configure the following:

  - **Precision** — Precision of analysis
  - **Passes** — Level of analysis
  - **Results folder** – Location where software stores analysis results

- Under **Verification Mode Settings**, configure the following:

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

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

    a Select ➕. The Select Files and Classes dialog box opens.
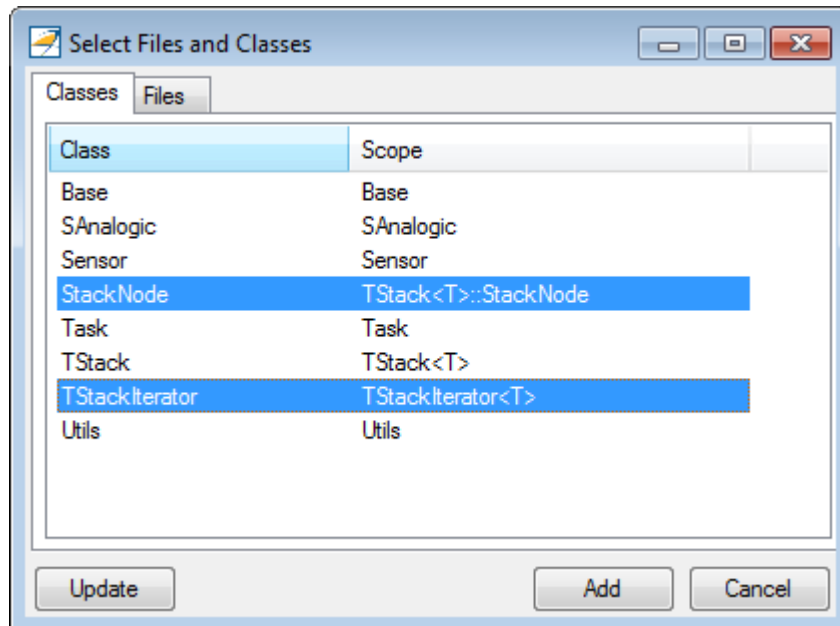
    b Select the classes that you want to analyze, then click **Add**.

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".

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

5 Make sure the **Use Code Prover analysis** check box is not selected.

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.
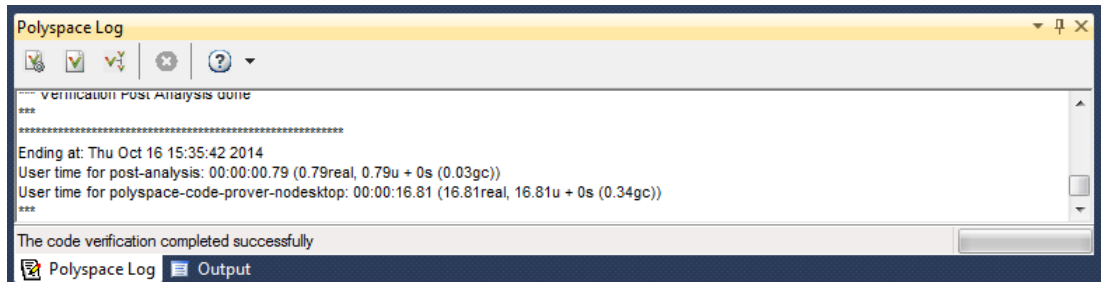
If the analysis is being carried out on a server, use the Polyspace Job Monitor to follow the progress of the analysis.

Open and Review Results

1
After your analysis completes, select [icon].

2 In Polyspace, review your results.

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

Polyspace UML Link RH

Find Defects from IBM Rational Rhapsody

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 3-11and "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.

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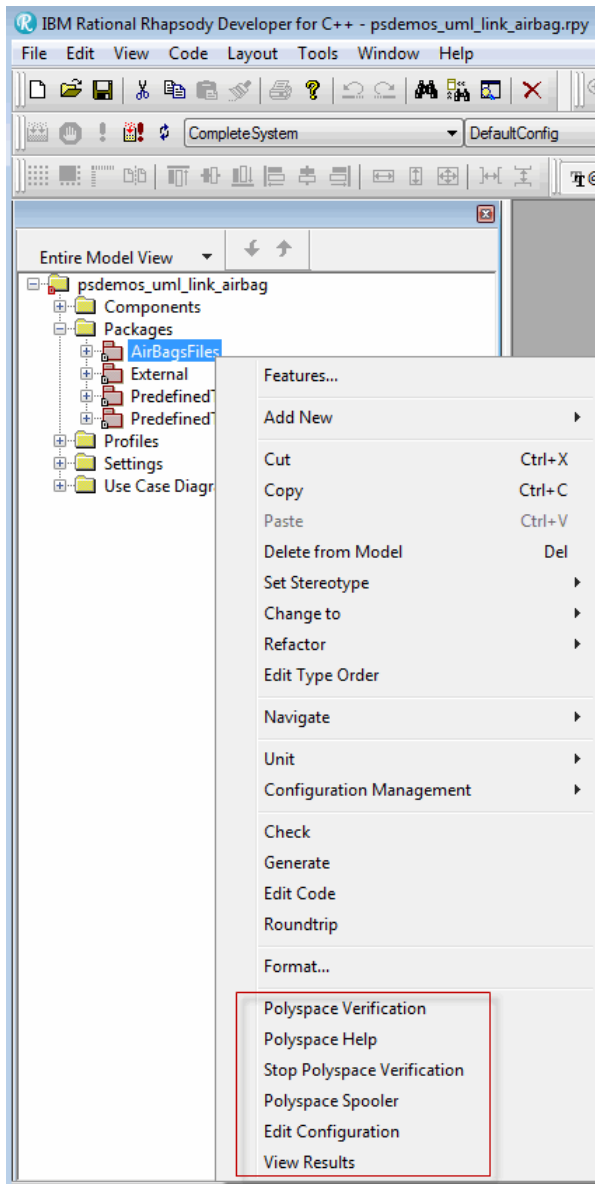
Polyspace Verification is also available from the **Tools** menu.

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.
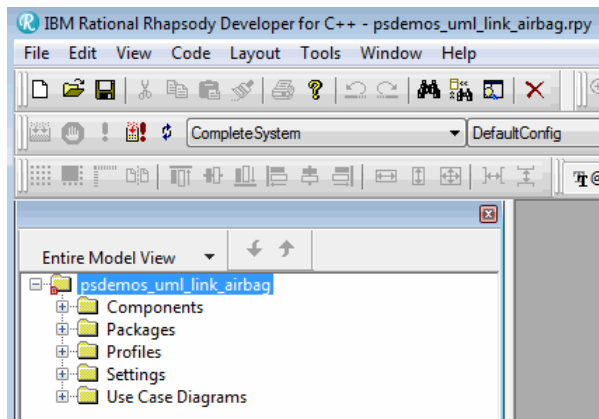
2 In the **Entire Model View**, expand the Packages node.
3 Right-click a package, for example, **AirBagFiles**.
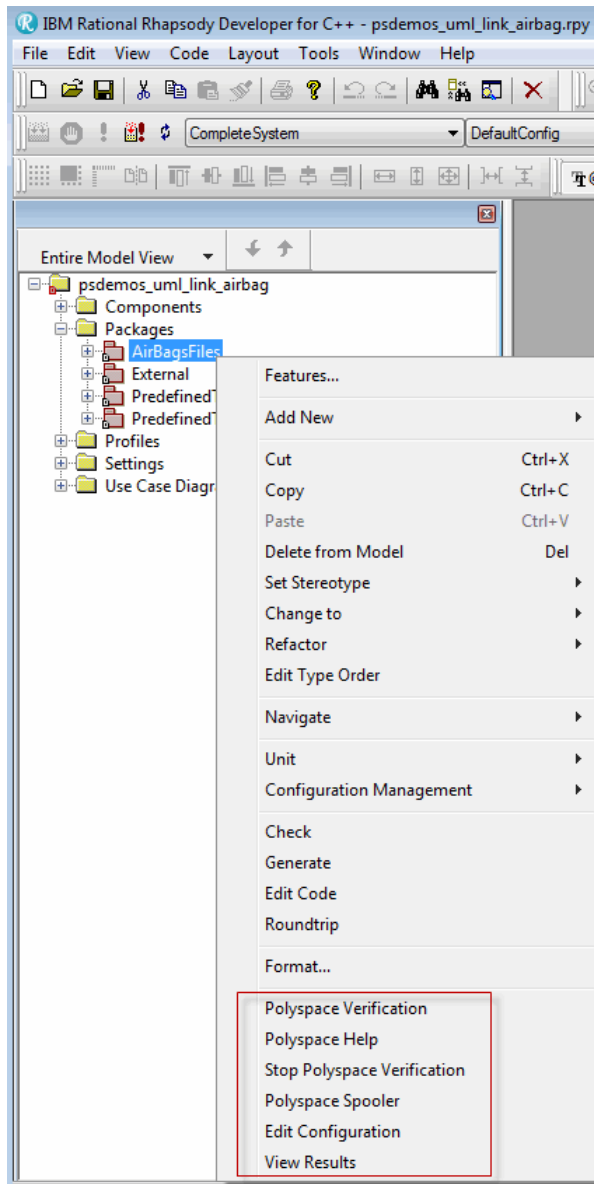
You see the following Polyspace functions in the context menu:

- **Polyspace Verification** — Start analysis. See "Running an Analysis" on page 3-8.
- **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`.
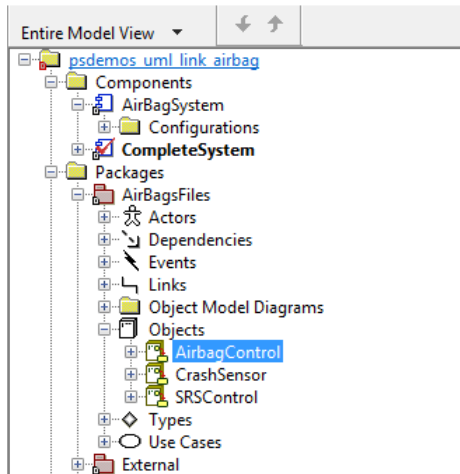
2 From the context menu, select **Edit Configuration**. The **Configuration** pane of the Polyspace environment opens.
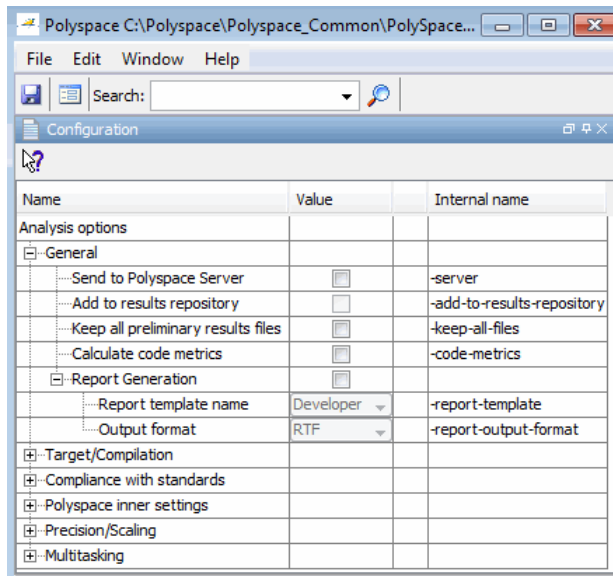
3 Select options for your analysis. In particular, you must specify the following:

- **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.
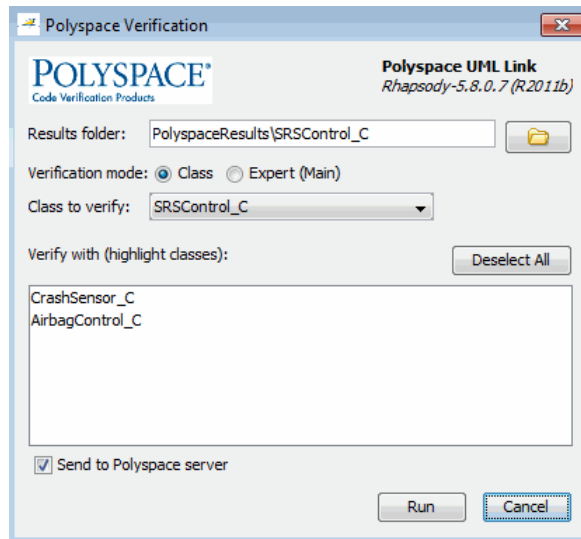
Note: Before starting an analysis, make sure that the generated code for the model is up to date.
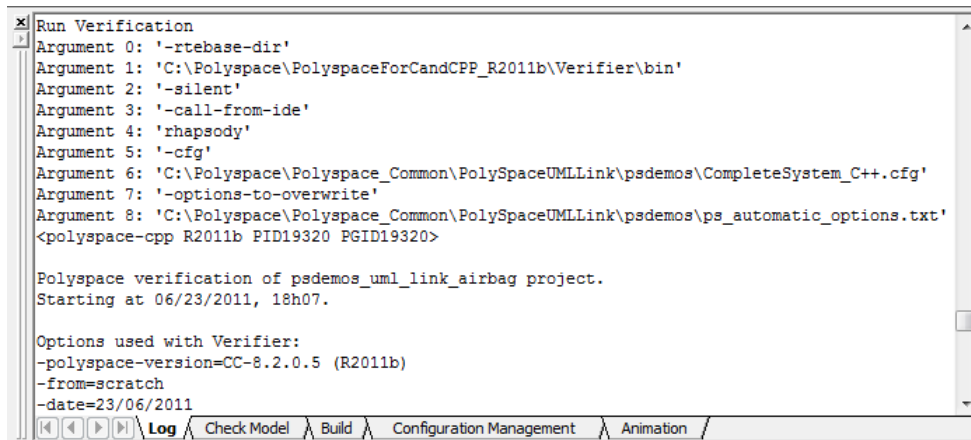
2   In the Results folder field, specify a location for your analysis results.

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**.

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
```

Log / Check Model / Build / Configuration Management / Animation /

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
```

Log / Check Model / Build / Configuration Management / Animation /

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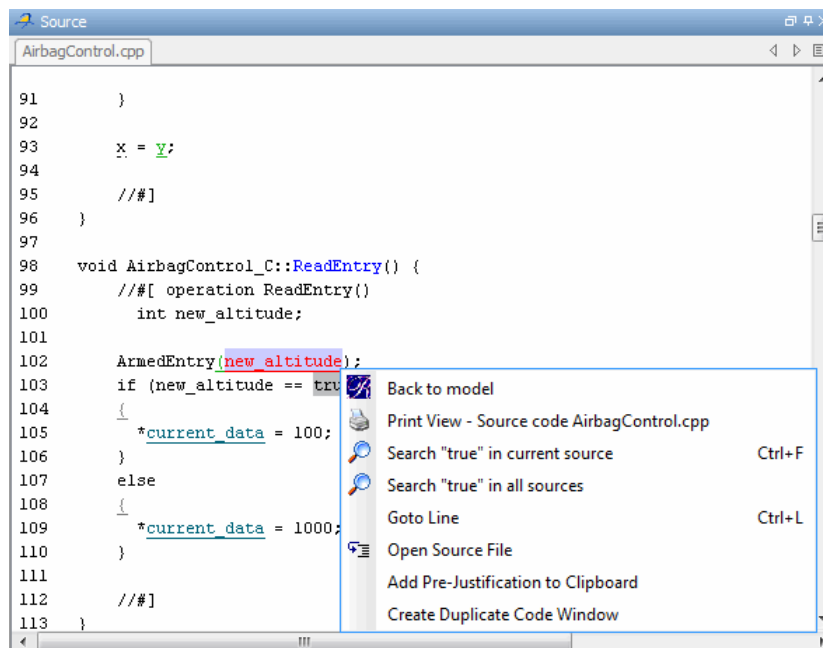
Tip For the **Back to model** command to work, you must have your Rhapsody model
open.

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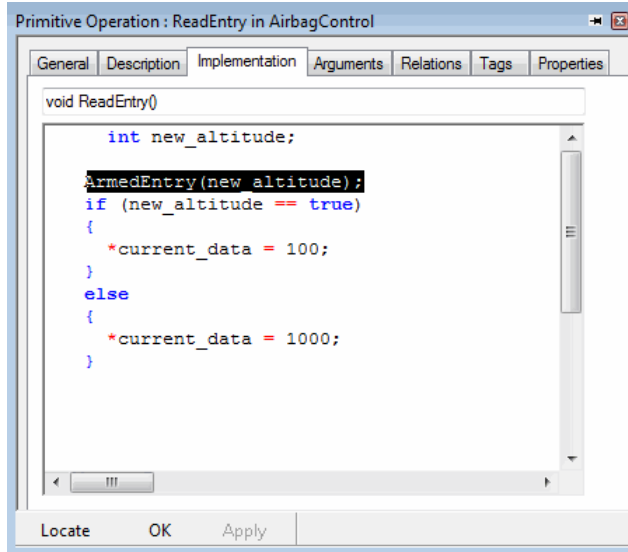
Note: The 64-bit version of the Polyspace product does not support the **Back to model** command with the 32-bit IBM Rational Rhapsody product.

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 Plug-In for Simulink

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

If you model on a previous version of Simulink and MATLAB, you can also connect the Polyspace plug-in on this previous version. That way you use the latest verification software with your preferred version of Embedded Coder® or TargetLink. The Simulink plug-in supports the four previous releases of MATLAB. For example, the R2015a version of the Polyspace plug-in supports MATLAB R2013a, R2013b, R2014a,R2014b, and R2015a.

However, 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 If you have a previous version of Polyspace connected, execute the `pslinksetup('uninstall')` command to disconnect it. This command does not work with MATLAB R2013b or later (see preceding Note).
3 Restart MATLAB.
4 Change your **Current Folder** to *matlabroot*`\polyspace\toolbox\pslink` `\pslink`. *matlabroot* is the Simulink plug-in that you want to connect, for example, `C:\Program Files\MATLAB\R2015a`.
5 Execute the `pslinksetup('install')` command to connect the new version of Polyspace.

Related Examples

· "Find Defects from Simulink"

More About

· "Troubleshoot Back to Model"

Install Polyspace Plug-In for Eclipse

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 4-5.

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*\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`.
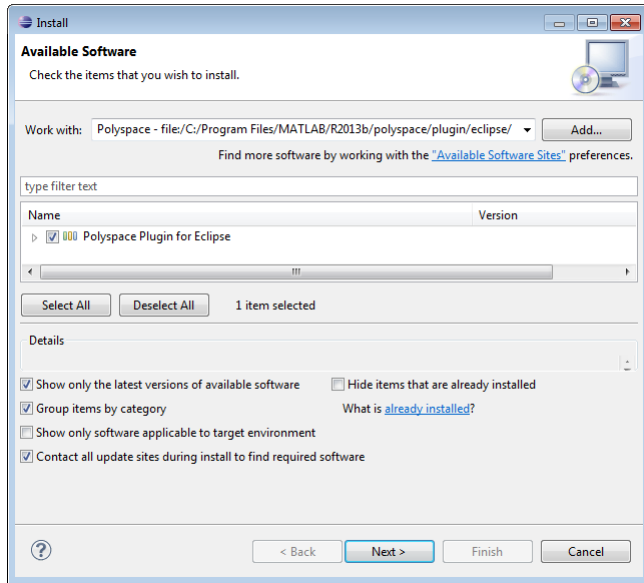
8   Click **Next**.

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

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

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

- A **Polyspace** menu

- A **Polyspace Run**, **Results Summary**, and **Check Details** view.
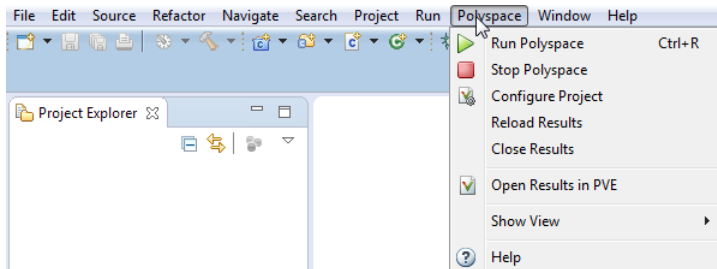
Uninstall Polyspace Plug-In for Eclipse IDE

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.

Related Examples
- "Find Defects from the Eclipse Plug-In"

More About
- "Eclipse Environment"

Install Polyspace Add-In for Visual Studio

Install Polyspace Add-In

The Polyspace Add-in is supported for Visual Studio 2008, 2010. 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 4-7.

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*\polyspace\plugin\msvc*VS_version*

- *MATLAB_Install* is the installation folder for the Polyspace product.
- *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
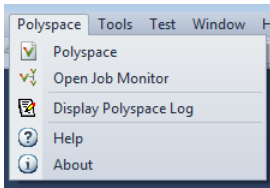
- A **Polyspace Log** view

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.

Related Examples

- "Find Defects from Visual Studio"

More About

- "Visual Studio Environment"

Set Up Polyspace Metrics

In this section...
"Requirements for Polyspace Metrics" on page 4-8
"Start Polyspace Metrics Server" on page 4-9
"Configure Polyspace Preference" on page 4-9
"Configure Web Server for HTTPS" on page 4-11
"Change Web Server Port Number for Metrics Server" on page 4-12

Requirements for Polyspace Metrics

You can use Polyspace Metrics to:

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

The following table lists the requirements for Polyspace Metrics.

Task	Location	Requirements
Project configuration and uploads to server	Client node	• MATLAB • Polyspace Bug Finder
Polyspace Metrics service	Network server or head node of MDCS cluster	• MATLAB • Polyspace Bug Finder Activation is not required for the Polyspace Metrics service
Downloading *complete* results from Polyspace Metrics	Client node or a network computer	• MATLAB • Polyspace Bug Finder • Access to Polyspace Metrics server
Viewing results *summary* from Polyspace Metrics	A network computer	Access to Polyspace Metrics server.

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

Start Polyspace Metrics Server

This task shows you how to start the host server for Polyspace Metrics.

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 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.
3 If you want to configure your MDCS head node (for remote verifications and analyses) as the Polyspace Metrics server, select **Start the Polyspace mdce service without security level**. Otherwise, clear this check box. For more information about starting your remote cluster service, see "Set Up Server for Remote Verification and Analysis" on page 4-14.
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 dialog box in the following file:

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

Configure Polyspace Preference

1 Select **Tools** > **Preferences**.

2   Click the **Polyspace Preferences** > **Server Configuration** tab.

3   Under **Metrics configuration**:

- If you want the software to detect a server on the network that uses port 12427, click **Automatically detect the Polyspace Metrics Server**.

  Otherwise, to specify the host computer for your Polyspace Metrics server, click **Use the following server and port**. Enter an IP address (or server name) and the Polyspace communication port number (default 12427). You must specify the same port number for all clients that use the Polyspace Metrics service.

- By default, the software selects the **Download results automatically** check box.

  In the **Folder** field, specify a local folder for downloading result files from Polyspace Metrics.

  In Polyspace Metrics, when you click an item to view it within the Polyspace environment, the software downloads results to the analysis launch folder. If this folder does not exist, the software downloads results to the folder specified in the **Folder** field. The default is `C:\Temp`.

  If you clear the **Download results automatically** check box, when you click an item in Polyspace Metrics, a dialog box opens. In this dialog box, you can specify your locally accessible folder. When you exit the Polyspace environment, the folder and its contents are not deleted.

- In the **Port number** field, specify the port number for communication between the Polyspace environment and the Polyspace Metrics Web interface. The default is `12428`.

- In the **Web server port number** field, specify the port number for the Web server. For HTTP, the default is `8080`.

  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 .

  If you use HTTPS for your Web protocol, select **Use secure HTTPS protocol instead of HTTP protocol to access Metrics results**. Specify your port number in the corresponding field. For HTTPS, the default is `8443`.

There are additional steps to set up the Web server for HTTPS. See "Configure Web Server for HTTPS" on page 4-11.

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.

Configure Web Server for HTTPS

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

Before you start the following procedure, you must complete "Start Polyspace Metrics Server" on page 4-9 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 `%APPDATA%\Polyspace_RLDatas\tomcat\conf\server.xml` file in a text editor. Look for the following text:

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

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

    a   Delete the entire `..\conf\` folder.

    b   In the Metrics and Remote Server Settings dialog box, restart the daemon by clicking **Start Daemon**.

    c   Click **Stop Daemon** to stop the services again so that you can finish setting up the server for HTTPS.

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

4   Follow the commented-out instructions in `server.xml` to create a keystore for the HTTPS certificate.

5   In the Metrics and Remote Server Settings dialog box, to restart the Polyspace Metrics service with the changes, click **Start Daemon**.

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

https://*ServerName*:*WSPN*

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

Change Web Server Port Number for Metrics Server

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

1   Select **Metrics** > **Metrics and Remote Server Settings**.

2   In the Metrics and Remote Server Settings dialog box, select **Stop Daemon** to stop the Polyspace Metrics server daemon.

3   In *AppData*`\Polyspace_RLDatas\tomcat\conf\server.xml`, edit the `port` attribute of the `Connector` element for your Web server protocol.

- 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 Metrics and Remote Server Settings dialog box, select **Start Daemon** to restart the server with the new port number.

5 On the Polyspace toolbar, select **Tools** > **Preferences**.

6 In the **Server Configuration** tab, change the **Web server port number** to match your new value.

Related Examples

• "View Results Summary in Polyspace Metrics"

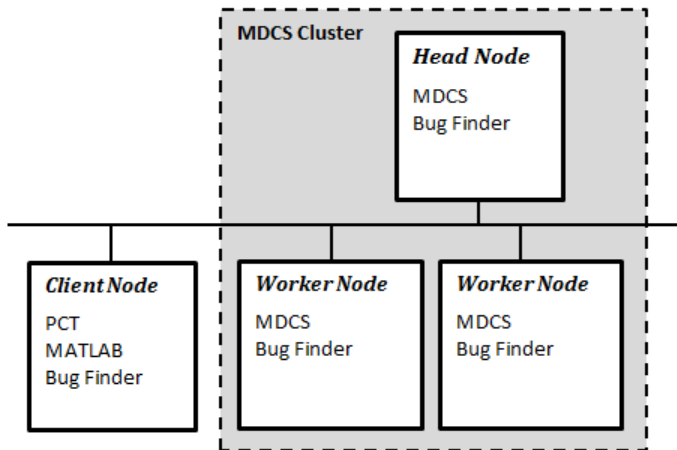# Set Up Server for Remote Verification and Analysis

In this section...
"Requirements for Remote Verification and Analysis" on page 4-15
"Start Server for Remote Verification and Polyspace Metrics" on page 4-15
"Configure Polyspace Preferences" on page 4-16

You can run the following types of verification and analyses.

Analysis type	Run when
Remote *batch*	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.

You can also use Polyspace Metrics with your remote verifications, but it is not required. For more information about setting up Polyspace Metrics, see "Set Up Polyspace Metrics" on page 4-8.

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 4-15.

2 Configure the client node through the **Server Configuration** tab. 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	• MATLAB • Parallel Computing Toolbox • Polyspace Bug Finder
Remote analysis	Head node of MDCS cluster	• MATLAB Distributed Computing Server • Polyspace Bug Finder

For information about setting up a computer cluster, see "Install Products and Choose Cluster Configuration".

Start Server for Remote Verification and Polyspace Metrics

This procedure describes how to set up an MDCS head node that is also the Polyspace Metrics server. If you do not want to set up Polyspace Metrics, use the MDCS Admin Center to set up a server for your remote verifications. See "Install Products and Choose Cluster Configuration".

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.

3 If you want to configure the MDCS head node as the Polyspace Metrics server, under **Polyspace MDCS Cluster Security Settings**, you see the following options with default values:

- **Start the Polyspace mdce service without security** — Selected. The `mdce` service, which is required to manage the MJS, runs on the MJS host computer with security level 0. If you want to require authentication to use the remote server, use the MDCS **Admin Center**. For more information about setting up security levels, see "Set MJS Cluster Security".

- **MDCE service port** — 27350.

- **Use secure communication** – Not selected. Communication is not encrypted. You can, for example, increase the security level and use secure communication.

4 To start the Polyspace Metrics and `mdce` services, 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%\PolyspaceRLDatas\polyspace.conf`

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

Configure Polyspace Preferences

1 Select **Tools** > **Preferences**.

2 Click the **Polyspace Preferences** > **Server Configuration** tab.

3 Under **MDCS cluster configuration**, 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).

You can configure the MJS host through the MATLAB Distributed Computing Server Admin Center. See "Configure for an MJS".

4 Under **Metrics configuration**, specify the host computer for your Polyspace Metrics server or let Polyspace detect the server. For more information, see "Set Up Polyspace Metrics" on page 4-8.

Related Examples

- "Run Remote Batch Analysis"