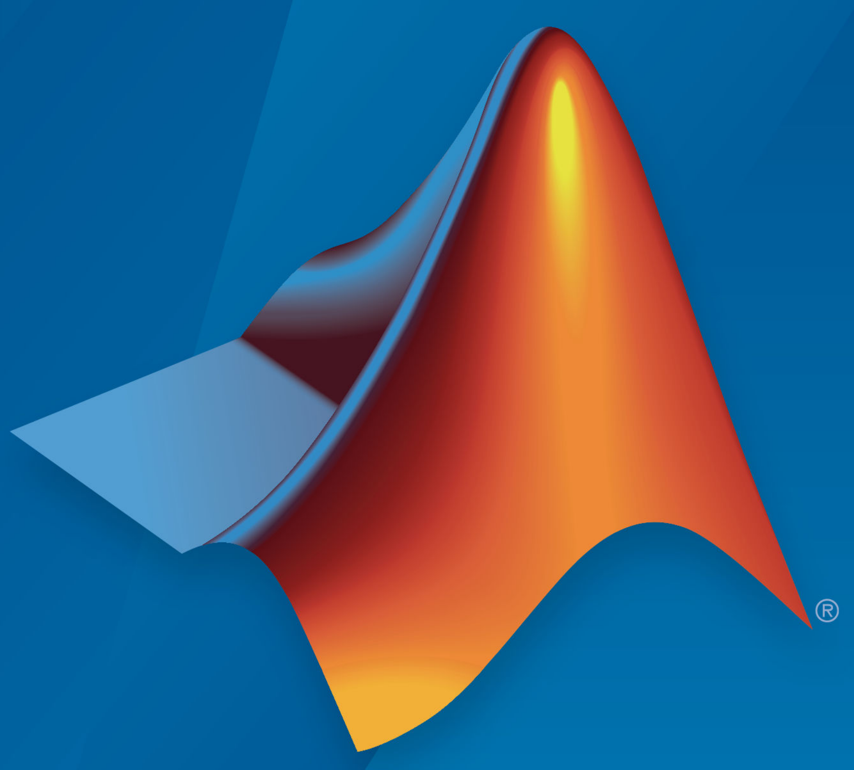
Polyspace® Code Prover™

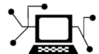
Getting Started Guide



MATLAB® & SIMULINK®

R2017b

How to Contact MathWorks



Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

Polyspace® Code Prover™ Getting Started Guide

© COPYRIGHT 2013–2017 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

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

Contents

1 Introduction to Polyspace Code Prover

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

Getting Help **1-3**
Access Documentation **1-3**
Access Contextual Help **1-3**

Quick Start Guide for Polyspace Code Prover **1-5**

2 Set Up a Polyspace Project

Compiler Requirements **2-2**

Set Up Polyspace Project **2-3**
Tutorial Overview **2-3**
What Is a Project? **2-3**
Prepare Project Folder **2-3**
Open Polyspace Code Prover **2-4**
Create Project **2-4**
Next steps **2-6**

3 Server Configuration for Remote Verification and Polyspace Metrics

Set Up Polyspace Metrics **3-2**
Requirements for Polyspace Metrics **3-2**
Start Polyspace Metrics Server **3-3**
Configure Polyspace Preference **3-4**
Configure Web Server for HTTPS **3-5**
Change Web Server Port Number for Metrics Server **3-7**

Set Up Server for Metrics and Remote Analysis **3-8**
Requirements for Remote Verification and Analysis **3-9**
Start Server for Remote Verification and Polyspace Metrics . . **3-9**
Configure Polyspace Preferences **3-11**

4 Run a Verification

Run Verification **4-2**
Tutorial Overview **4-2**
Before You Start the Tutorial **4-2**
Prepare for Verification **4-2**
Run Remote Verification **4-3**
Run Local Verification **4-5**
Next steps **4-6**

5 Review Verification Results

Review Results **5-2**
Tutorial Overview **5-2**
Open Results **5-2**
Review Results **5-3**
Generate Report **5-4**
Next steps **5-5**

6 Check Compliance with Coding Rules

Find Coding Rule Violations **6-2**
Tutorial Overview **6-2**
Specify MISRA C Checking **6-2**
Review MISRA C Violations **6-3**

7 Verifying Code Generated from Simulink Models

Verification of Code Generated from Simulink Models **7-2**

Verify Code from a Simple Simulink Model **7-3**
Create Simulink Model and Generate Code **7-3**
Run Polyspace Verification **7-5**
View Results in Polyspace Code Prover **7-5**
Trace Error to Simulink Model **7-6**
Specify Signal Ranges **7-7**
Verify Updated Model **7-9**

8 Code Verification in IBM Rational Rhapsody Environment

Verify Code in IBM Rational Rhapsody Environment **8-2**
Code Verification Approach **8-2**
Adding Polyspace Profile to Model **8-3**
Accessing Polyspace Features **8-3**
Configuring Verification Options **8-6**
Running a Verification **8-7**
Viewing Polyspace Results **8-7**
Locating Faulty Code in Rhapsody Model **8-8**
Template Configuration Files **8-9**

9 Using Bug Finder and Code Prover

Differences Between Polyspace Bug Finder and Polyspace Code Prover Analysis **9-2**
How Bug Finder and Code Prover Complement Each Other . . **9-2**
Workflow Using Both Bug Finder and Code Prover **9-8**

Introduction to Polyspace Code Prover

- "Polyspace Code Prover Product Description" on page 1-2
- "Getting Help" on page 1-3
- "Quick Start Guide for Polyspace Code Prover" on page 1-5

Polyspace Code Prover Product Description

Prove the absence of run-time errors in software

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

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

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

Key Features

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

Getting Help

In this section...
"Access Documentation" on page 1-3
"Access Contextual Help" on page 1-3

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

Access Documentation

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

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

Access Contextual Help

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

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

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

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

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

See Also

Related Examples

- "Configure Advanced Polyspace Analysis Options"
- "Configure Verification"

Quick Start Guide for Polyspace Code Prover

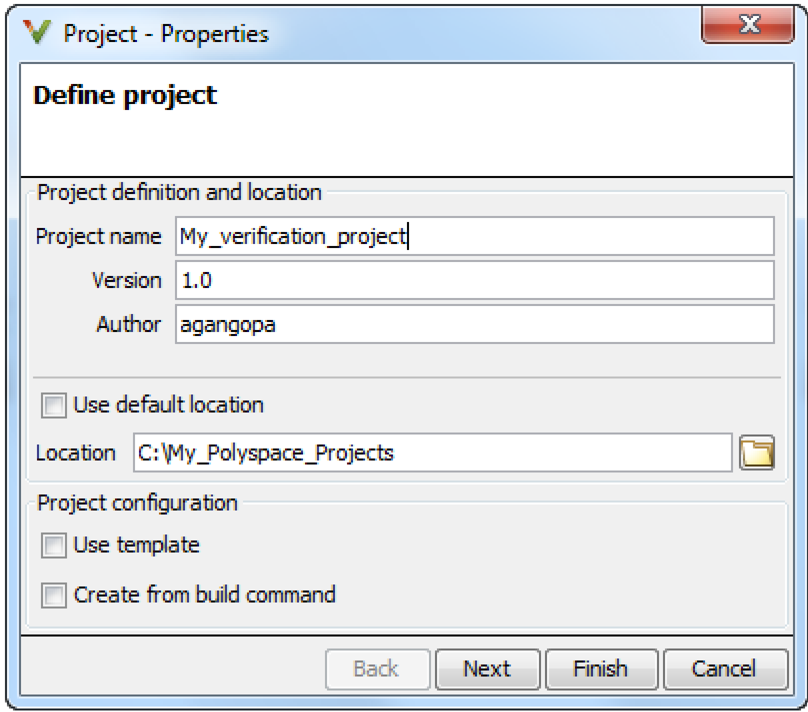
Polyspace® Code Prover™ is a sound static analysis tool that proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in C and C++ source code. It produces results without requiring program execution, code instrumentation, or test cases. Polyspace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

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

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

Step 1: Set Up Project

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



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

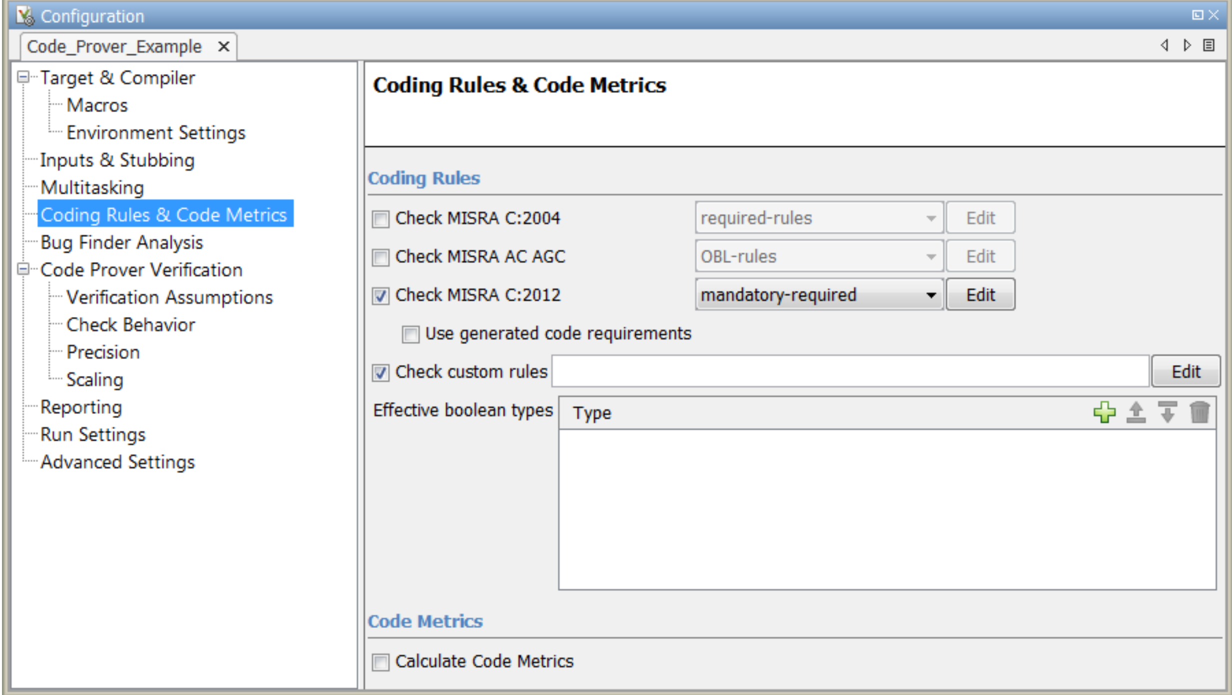
- Copy the example files `example.c` and `include.h` from *`MATLAB_Install`*`\polyspace\examples\cxx\Code_Prover_Example\sources` to a new folder. Change the read-only status of the files. Add the folder to your project as both source and include folder. *`MATLAB_Install`* is the location of your MATLAB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

Step 2: Configure Project and Run Verification

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

The default analysis options appear on the **Configuration** pane. Retain the default options or change them to your requirements. For example, to check for coding rule

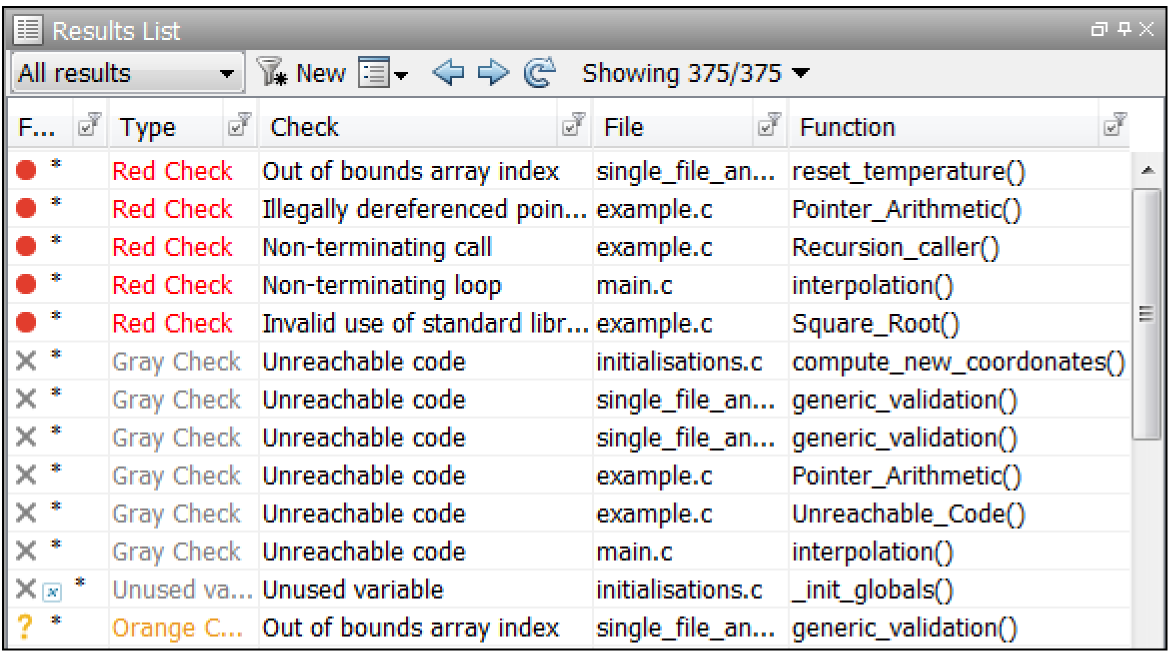
violations, select **Coding Rules & Code Metrics** and specify your options. For more information on an option, place your cursor on the option and select **More Help**.



Click the **Run Code Prover** button.

Step 3: Review Results

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



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

Set Up a Polyspace Project

- “Compiler Requirements” on page 2-2
- “Set Up Polyspace Project” on page 2-3

Compiler Requirements

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

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

Set Up Polyspace Project

In this section...

"Tutorial Overview" on page 2-3
"What Is a Project?" on page 2-3
"Prepare Project Folder" on page 2-3
"Open Polyspace Code Prover" on page 2-4
"Create Project" on page 2-4
"Next steps" on page 2-6

Tutorial Overview

In this tutorial, you create a new Polyspace Code Prover project to verify C code.

What Is a Project?

A Polyspace project consists of:

- **Source** folders and their files.
- **Include** folders.
- One or more modules. You run verification on the source files in each module. Each module has the following folders:
  - **Source** — Contains files used for verification.
  - **Configuration** — Contains analysis options used for verification.
  - **Result** — Contains results of verification.

Prepare Project Folder

In the following procedures, *`matlabroot`* is the MATLAB® installation folder, for instance, `C:\Program Files\MATLAB\R2017a`.

1. Create a folder `polyspace_project` in a particular location, for example `C:\`.
2. Open `polyspace_project` and create subfolders:

- `sources`
- `includes`

3. Copy `example.c` from *`matlabroot`*`\polyspace\examples\cxx\Code_Prover_Example\sources` to `polyspace_project\sources`.
4. Copy `include.h` from *`matlabroot`*`\polyspace\examples\cxx\Code_Prover_Example\sources` to `polyspace_project\includes`.

Open Polyspace Code Prover

- Open directly in your operating system.
  - Windows®: From the *`matlabroot`*`\polyspace\bin` folder, double-click the `polyspace` executable.
  - Linux® or Mac: Run the following command:

    ```
    /matlabroot/polyspace/bin/polyspace
    ```
- Open from MATLAB.

  From the MATLAB **Apps** gallery, click the Polyspace Code Prover app.

Create Project

- "Create New Project" on page 2-4
- "Specify Source Files and Include Folders" on page 2-5

Create New Project

1. Select **File** > **New Project**.
2. In the Project – Properties dialog box:
   - For **Project name**, enter `polyspace_project`.
   - Clear the **Use default location** check box. To specify where your `polyspace_project` folder is, click .
   - Clear the boxes under **Project Configuration**.

     For more information on the option **Use template**, see "Create Project Using Configuration Template".

For more information on the option **Create from build command**, see “Create Project Automatically”.

3. Click **Next**.

Specify Source Files and Include Folders

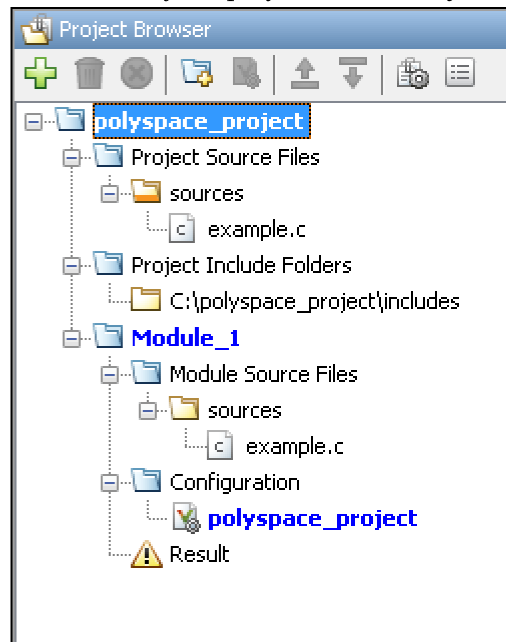
1. Use the **Browse** button to select the `sources` folder that you created.
2. Click **Add Source Folders**, then click **Next**.
3. Use the **Browse** button to select the `includes` folder that you created.
4. Click **Add Include Folders**, then click **Finish**.

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

   ```
   C:\My_Project\MySourceFiles\Includes
   ```

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

You can see your project in the **Project Browser**.



Next steps

1. "Run Verification" on page 4-2
2. "Review Results" on page 5-2
3. "Find Coding Rule Violations" on page 6-2

See Also

Related Examples

- "Create Project"

Server Configuration for Remote Verification and Polyspace Metrics

- “Set Up Polyspace Metrics” on page 3-2
- “Set Up Server for Metrics and Remote Analysis” on page 3-8

Set Up Polyspace Metrics

In this section...
“Requirements for Polyspace Metrics” on page 3-2
“Start Polyspace Metrics Server” on page 3-3
“Configure Polyspace Preference” on page 3-4
“Configure Web Server for HTTPS” on page 3-5
“Change Web Server Port Number for Metrics Server” on page 3-7

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

Requirements for Polyspace Metrics

You can use Polyspace Metrics to:

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

This table lists the requirements for Polyspace Metrics.

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

Task	Location	Requirements
Downloading *complete* results from Polyspace Metrics	Client node or a network computer	• MATLAB • Polyspace Bug Finder or Polyspace Code Prover • 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 Polyspaceon top of an older version, Polyspace Metrics automatically updates the database to the newest version.

Start Polyspace Metrics Server

This section shows you how to start the host server for Polyspace Metrics. After you complete this step, you must also configure the client-side settings on page 3-4 so that the Polyspace interface can interact with the Metrics server.

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

---

1. From the Polyspace environment, select **Metrics** > **Metrics and Remote Server Settings**.
2. Under **Polyspace Metrics Settings**, specify:
   - **User name used to start the service** — Your user name.
   - **Password** — Your password (Windows only).
   - **Communication port** — Polyspace communication port number (default 12427). This number must be the same as the communication port number specified in the Polyspace Interface preferences. See "Configure Polyspace Preference" on page 3-4.
   - **Folder where analysis data will be stored** — Results repository for Polyspace Metrics server.
3. If you have installed MATLAB Distributed Computing Server, clear the **Start the Polyspace mdce service without security level** check box.

For information about starting your remote cluster service, see "Set Up Server for Metrics and Remote Analysis" on page 3-8.

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

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

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

Configure Polyspace Preference

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

1. Select **Tools** > **Preferences**.
2. Click the **Server Configuration** tab.
3. Under the **Polyspace Metrics server configuration** section:

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

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

      You must specify the same communication port number for all clients that use the Polyspace Metrics service.
4. Under the **Polyspace Metrics web interface configuration** section:

   a. Specify a **Port used to download results**, default is 12428. If you change this port number, you must also change it in on the server side.

   b. Specify which protocol to use HTTP or HTTPS. If you select HTTPS for your web protocol, there are additional steps to set up the Metrics web server for HTTPS on page 3-5.

   c. Specify a web server port number for your chosen protocol. Default port numbers are:

- HTTP — 8080
- HTTPS — 8443

If you change the port number from the default, you must configure the same port number for the Polyspace Metrics server. See “Change Web Server Port Number for Metrics Server” on page 3-7.

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

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

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

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

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

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

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

Configure Web Server for HTTPS

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

Before you start the following procedure, you must complete “Start Polyspace Metrics Server” on page 3-3 and “Configure Polyspace Preference” on page 3-4.

To configure HTTPS access to Polyspace Metrics:

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

   ```
   MATLAB_Install\polyspace\bin\polyspace-server-settings.exe
   ```

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

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

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

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

   The `conf` folder is regenerated, including the `server.xml` file. The file now contains the text required to configure the HTTPS web server.
4. Follow the commented-out instructions in `server.xml` to create a keystore for the HTTPS certificate.
5. In the Metrics and Remote Server Settings dialog box, to restart the Polyspace Metrics service with the changes, click **Start Daemon**.

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

`https``://`*`ServerName`*`:`*`WSPN`*

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

Change Web Server Port Number for Metrics Server

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

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

   - For HTTP:

     ```
     <Connector port="8080"/>
     ```

   - For HTTPS:

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

   ```
   <Server port="8005" shutdown="SHUTDOWN">
   ```
5. In the Metrics and Remote Server Settings dialog box, select **Start Daemon** to restart the server with the new port numbers.
6. On the Polyspace toolbar, select **Tools** > **Preferences**.
7. In the **Server Configuration** tab, change the **Web server port number** to match your new value for the `port` attribute in the `Connector` element.

See Also

Related Examples

- “Generate Code Quality Metrics”

Set Up Server for Metrics and Remote Analysis

In this section...
"Requirements for Remote Verification and Analysis" on page 3-9 "Start Server for Remote Verification and Polyspace Metrics" on page 3-9 "Configure Polyspace Preferences" on page 3-11

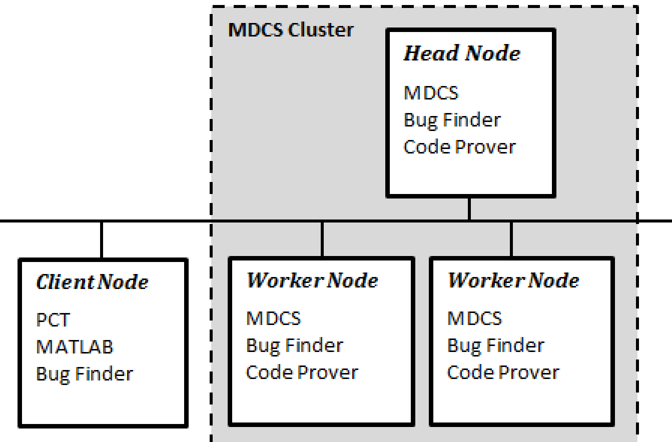
You can perform a Polyspace verification locally on your desktop or on a remote server. This topic shows how to set up Polyspace on a server for remote batch verification.

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

Type	When to Use
Remote *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.

With both local and remote verification, you can upload your results to the Polyspace Metrics web interface or view them directly on your desktop application. For more information about setting up Polyspace Metrics, see "Set Up Polyspace Metrics" on page 3-2.

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 3-9.
2. Configure the client node through the Polyspace environment preferences. See, “Configure Polyspace Preferences” on page 3-11.

Requirements for Remote Verification and Analysis

The following table lists the requirements for remote verification.

Task	Location	Requirements
Project configuration and job submission	Client node	• MATLAB • Parallel Computing Toolbox • Polyspace Bug Finder or Polyspace Code Prover
Remote analysis and verification	Head node of cluster	• MATLAB Distributed Computing Server • Polyspace Bug Finder • Polyspace Code Prover

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

Start Server for Remote Verification and Polyspace Metrics

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

1. Select **Metrics** > **Metrics and Remote Server Settings**.
2. Under **Polyspace Metrics Settings**, specify:

   - **User name used to start the service** — Your user name.
   - **Password** — Your password (Windows only).
   - **Communication port** — Polyspace communication port number (default 12427). This number must be the same as the communication port number specified on the **Polyspace Preferences** > **Server Configuration** tab.

- **Folder where analysis data will be stored** — Results repository for Polyspace Metrics server.

3 To configure the Polyspace Metrics server as the MATLAB Distributed Computing Server head node, select **Start the Polyspace mdce service without security level**.

   The `mdce` service, which is required to manage the MJS, runs on the MJS host computer with security level 0. At level 0, jobs are associated with the default user name of the user. A login or password is not required to manage and see these jobs.

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

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

   - **Mdce service port** — 27350.

     This option specifies the port on which you connect to the MJS server. If you change this number, you must change it on both the server and client side. On the client side, when you specify the job scheduler host name (**Tools** > **Preferences** and then **Server Configuration**), specify the port using the notation *`schedulerName`*`:`*`portNumber`*. For instance, `myJobScheduler:27400`. See "Verify Network Communications for Cluster Discovery" (MATLAB Distributed Computing Server).
   - **Use secure communication** – Not selected by default

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

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

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

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

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

Configure Polyspace Preferences

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

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

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

      To retrieve your IP address:

      - Windows

        i. Open **Control Panel** > **Network and Sharing Center**.

        ii. Select your active network.

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

   If required, you can configure additional options for the MJS host through the MATLAB Distributed Computing Server Admin Center. See “Configure for an MJS” (MATLAB Distributed Computing Server).
4. Under the **Polyspace Metrics server configuration** section:

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

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

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

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

   a. Specify a **Port used to download results**, default is 12428. If you change this port number, you must also change it in on the server side.

   b. Specify which protocol to use HTTP or HTTPS. If you select HTTPS for your web protocol, there are additional steps to set up the Metrics web server for HTTPS on page 3-5.

   c. Specify a web server port number for your chosen protocol. Default port numbers are:

      - HTTP — 8080
      - HTTPS — 8443

      If you change the port number from the default, you must configure the same port number for the Polyspace Metrics server. See "Change Web Server Port Number for Metrics Server" on page 3-7.

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

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

     If you want Polyspace to automatically upload your justifications to Polyspace Metrics when you save, select **Upload justifications automatically in the Polyspace Metrics repository**.

   - Download settings — In Polyspace Metrics, when you click an item to view, Polyspace downloads your results and opens them in the Polyspace environment. Select where to download your Polyspace Metrics results, either:

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

See Also

Related Examples

- “Set Up Polyspace Metrics” on page 3-2
- “Run Remote Verification”
- “Run File-by-File Remote Verification”
- “Job Manager Cannot Write to Database”

Run a Verification

Run Verification

In this section...
"Tutorial Overview" on page 4-2
"Before You Start the Tutorial" on page 4-2
"Prepare for Verification" on page 4-2
"Run Remote Verification" on page 4-3
"Run Local Verification" on page 4-5
"Next steps" on page 4-6

Tutorial Overview

In this tutorial, you run verification on your source code. Perform the steps outlined for remote verification if you want to perform verification on another machine. Otherwise, perform the steps outlined for local verification.

Before You Start the Tutorial

Before you start, you must:

- Complete "Set Up Polyspace Project" on page 2-3. You use the `polyspace_project` folder and the `polyspace_project.psprj` file in this tutorial.
- "Set Up Server for Metrics and Remote Analysis" on page 3-8 for remote verification and "Set Up Polyspace Metrics" on page 3-2 for Polyspace Metrics.

Prepare for Verification

If `polyspace_project.psprj` is not already open in the **Project Browser**, then:

1. Select **File** > **Open**.
2. In the Open File dialog box, navigate to `polyspace_project`.
3. Select the project file `polyspace_project`.
4. Click **Open**.

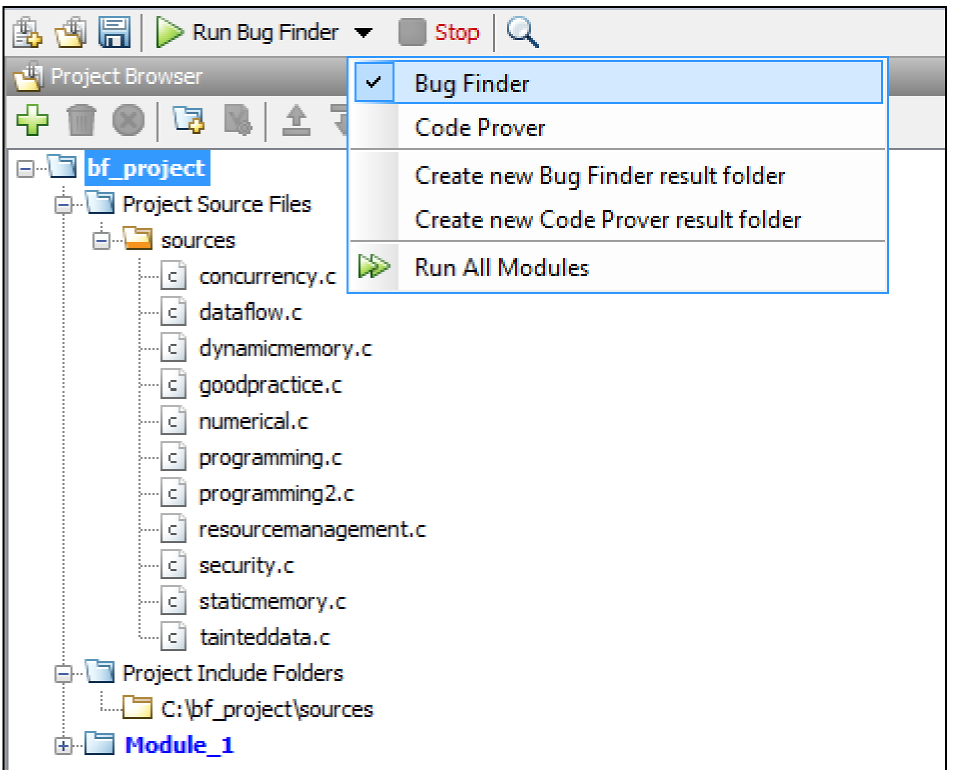
Run Remote Verification

- “Start Verification” on page 4-3
- “Monitor Progress” on page 4-5
- “Stop Verification” on page 4-5

Start Verification

Before you start remote verification, you must perform a one-time setup. See “Set Up Server for Metrics and Remote Analysis” on page 3-8.

1. On the **Project Browser** pane, select the configuration **polyspace_project** in **Module_1**.
2. On the **Configuration** pane, select **Run Settings**.
3. Select **Run Code Prover analysis on a remote cluster** and **Upload results to Polyspace Metrics**.
4. If you do not see the **Run Code Prover** button on the toolbar, select Code Prover as follows.



5 On the toolbar, click **Run Code Prover**.

The following happens:

a On the local host computer, Polyspace Code Prover compiles your code.

b The Parallel Computing Toolbox then submits the verification to the MATLAB Job Scheduler on the head node of the MATLAB Distributed Computing Server cluster.

For more information, see “Phases of Verification”.

Note If you see the message `Verification process failed`, click **OK**. For more information on troubleshooting remote verification errors, see “Polyspace Cannot Find the Server”.

Monitor Progress

To monitor the progress of a remote verification:

1. Select **Tools** > **Open Job Monitor**.
2. In the Polyspace Job Monitor, right-click your verification.
3. Select **View Log File**.

Stop Verification

To stop a remote verification:

1. Select **Tools** > **Open Job Monitor**.
2. In the Polyspace Job Monitor, right-click your verification.
3. Select **Remove From Queue**.

Run Local Verification

- “Start Verification” on page 4-5
- “Monitor Progress” on page 4-5
- “Stop Verification” on page 4-6

Start Verification

To start a verification on your local computer:

1. In the **Project Browser**, select the configuration **polyspace_project** in **Module_1**.
2. On the **Configuration** pane, select **Run Settings**. Clear **Run Code Prover analysis on a remote cluster** if it is selected.
3. On the toolbar, click **Run Code Prover**.

If the verification fails, see “Troubleshooting in Polyspace Code Prover”. For an introduction to why verification might fail in the compilation phase, see “Troubleshoot Compilation and Linking Errors”.

Monitor Progress

To monitor the progress of a local verification, on the **Output Summary** pane, use the following tabs:

- **Output Summary**
- **Run Log**

  If this window is not visible by default, select **Window** > **Show/Hide View** > **Run Log**.

When the verification is complete, you see:

- Results on the **Results List** pane.
- Statistics, such as **Code covered by verification** and **Check distribution** on the **Dashboard** pane.

Stop Verification

To stop a local verification:

1. On the toolbar, click Stop.

   A warning dialog box opens.
2. Click **Yes**.

   The verification stops. If you restart the verification, it starts from the beginning.

Next steps

1. “Review Results” on page 5-2
2. “Find Coding Rule Violations” on page 6-2

See Also

Related Examples

- “Run Analysis”

Review Verification Results

Review Results

In this section...
"Tutorial Overview" on page 5-2
"Open Results" on page 5-2
"Review Results" on page 5-3
"Generate Report" on page 5-4
"Next steps" on page 5-5

Tutorial Overview

In this tutorial, you explore the results of verifying `example.c`. Before starting this tutorial, complete "Run Verification" on page 4-2.

Open Results

- "Remote Verification" on page 5-2
- "Local Verification" on page 5-2

Remote Verification

To open results from a remote verification:

1. Select **Metrics** > **Open Metrics**.

   Alternatively, you can enter the remote address directly in a web browser. For more information, see "View Code Quality Metrics".
2. Click the **Project** cell of your verification.

   You can see a summary of your project.
3. On the **Summary** tab, click the **1.0** cell in the **Verification** column.

   Your results are downloaded into the user interface.

Local Verification

After verification, the results open automatically.

Review Results

Polyspace performs checks on each operation in your code. The software reports whether a check is green, red, orange or gray.

Check color	Indicates
Red	The code operation fails the check on every execution path.
Green	The code operation passes the check on every execution path.
Orange	The code operation fails the check on some execution paths.
Gray	The code operation is unreachable from entry-point functions.

1 On the **Results List** pane, from the list, select **File**.

   The checks are grouped by file. Within each file, the checks are grouped by function.

2 Expand the following function names and select a check in the function. The corresponding line of code on the **Source** pane appears highlighted. Further information about the check appears on the **Result Details** pane.

Function	Check	Source Code Appearance	Reason
`Unreachable_Code`	Gray **Unreachable code**	The code within braces starting from line 197 is gray.	`x` is greater than 0. So the `if` statement branch cannot be reached.
`Square_Root`	Red **Invalid use of standard library routine**	The function `sqrt` on line 182 is red.	`beta` is less than 0.75. So the argument to `sqrt` is always negative.
`Non_Infinite_Loop`	First green **Overflow**	The + sign on line 76 is green.	When `y` is too large, the `while` loop terminates. So the operation `x=x+2` never overflows.

Function	Check	Source Code Appearance	Reason
`Recursion`	Orange **Division by Zero**	The / sign on line 135 is orange.	`*depth` can be less than zero. Therefore, at some level in the recursion, the denominator can be zero.

3 To find further information about a check, do one of the following:

- Place your cursor on the check in the **Source** pane. View the tooltip.

  Use the variable range information in the tooltips to trace the data flow.

- Click the button on the **Result Details** pane. You can see a brief description of the check type, code examples and additional guidance on how to review that check type.

4 Filter **Illegally dereferenced pointer** checks. To do this, on the **Results List** pane:

a Click on the **Check** column header.

b From the drop-down list, clear **All** and select **Illegally dereferenced pointer**.

The **Results List** pane displays only the **Illegally dereferenced pointer** checks.

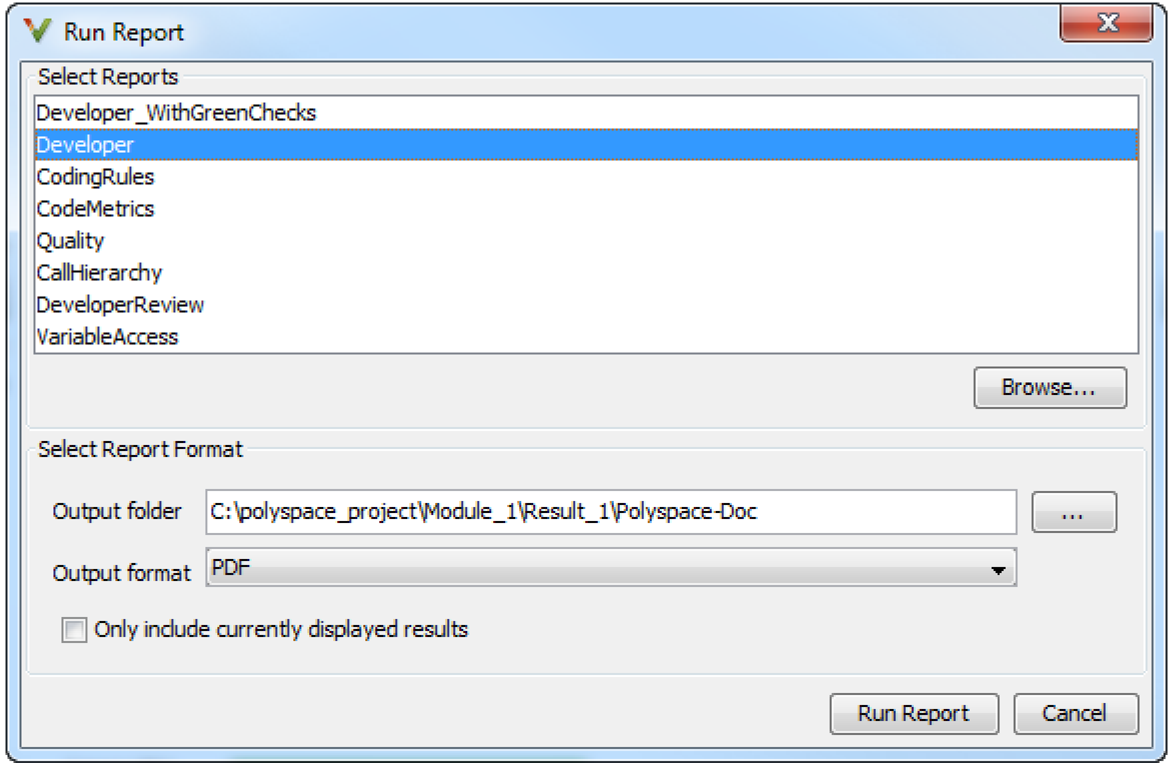
5 On the **Results List** pane, select the red **Illegally dereferenced pointer** check in the function `Pointer_Arithmetic`. Enter the following review information in the rightmost columns.

Column	Action
Severity	`High`
Status	`To fix`
Comment	`p points outside array`

Generate Report

To generate a verification report:

1. If your verification results are not already open, open them.
2. Select **Reporting** > **Run Report**.



3. In the **Select Reports** section, select **Developer**.
4. For **Output folder**, select `C:\polyspace_project\Module_1\Result_1\Polyspace-Doc`.
5. For **Output format**, select `PDF` .
6. Click **Run Report**.

   The software creates the specified report and opens it.

Next steps

"Find Coding Rule Violations" on page 6-2

See Also

Related Examples

- “Review Analysis Results”

6

Check Compliance with Coding Rules

Find Coding Rule Violations

In this section...
"Tutorial Overview" on page 6-2
"Specify MISRA C Checking" on page 6-2
"Review MISRA C Violations" on page 6-3

Tutorial Overview

In this tutorial, you analyze code to demonstrate compliance with established coding standards such as MISRA C 2004.

Using these rules during coding:

- Helps reduce amount of unproven code in your verification results.
- Improves the quality of your code.

Before you start, you must "Set Up Polyspace Project" on page 2-3.

Specify MISRA C Checking

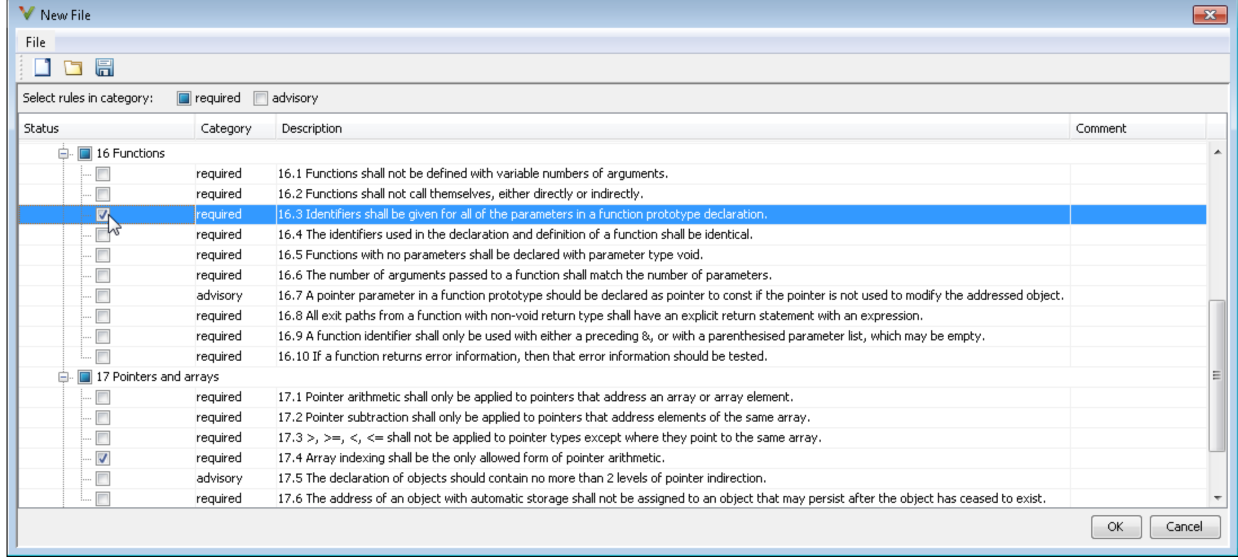
To set the MISRA C checking option:

1. On the **Project Browser**, select the configuration **polyspace_project** in **Module_1**.
2. On the **Configuration** pane, select **Coding Rules & Code Metrics**. Select **Check MISRA C:2004**.
3. From the corresponding drop-down list, select `custom`.
4. Click **Edit**. The New File dialog box opens, displaying a table of rules.
5. In the New File dialog box, specify the rules to check.

   a. Clear the **MISRA C:2004 rules** check box.

   b. Select the check boxes for the following rules.

Rule Number	Rule description
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.
17.4	Array indexing shall be the only allowed form of pointer arithmetic.



Click **OK**. Enter a name and save the file.

6 On the toolbar, click **Run Code Prover**.

After verification and coding rules checking, the results open automatically. If you have previous results on the **Results List** pane, you are prompted whether you want to open your new results. Click **OK**.

You can open your previous results from the **Project Browser** pane.

Review MISRA C Violations

To examine the MISRA C violations:

1 On the **Results List** pane, from the list, select **Family**.

The **MISRA C:2004** violations appear as a separate group.

2 Expand the nodes and select a coding-rule violation. You see the following.

Pane	Result
Source	The line containing the rule violation is highlighted.
Result Details	The following information is displayed: • Description of violated rule. • File and function where the rule violation appears. Click the [?] button. You can see a rationale for the rule. For certain rules, you can see additional code examples displaying violations of the rule.

3 On the **Source** pane, right-click the highlighted code. Select **Open Editor**.

The `example.c` file opens on the **Code Editor** tab. You can also use an external text editor. Select **Tools** > **Preferences** and specify an external editor on the **Editors** tab.

4 Fix the MISRA® violation and rerun the verification. The coding rule violation no longer appears in the results.

Verifying Code Generated from Simulink Models

- "Verification of Code Generated from Simulink Models" on page 7-2
- "Verify Code from a Simple Simulink Model" on page 7-3

Verification of Code Generated from Simulink Models

With Embedded Coder® or dSPACE® TargetLink® software, you can generate code from Simulink models. From Simulink, you can use Polyspace Code Prover to verify the generated code. The software detects run-time errors in the generated code and helps you to locate and fix model faults.

Use the following approach:

1. Configure your Simulink model and generate code. See .
2. Configure Polyspace verification options. See “Polyspace Configuration for Generated Code”

   Note After generating code, you can run a verification without manual configuration. By default, Polyspace Code Prover automatically creates a project and extracts required information from your model. However, you can also customize your verification. See “Configure Advanced Polyspace Analysis Options”.

   ---

3. Run Polyspace verification. See:

   - “Run Analysis for Embedded Coder”
   - “Run Analysis for TargetLink”
4. View results, analyze errors, locate and fix model faults. See “View Results in Polyspace Code Prover”.

   The software allows direct navigation from a run-time error in the generated code to the corresponding Simulink block or Stateflow® chart in the Simulink model. See “Identify Errors in Simulink Models”.

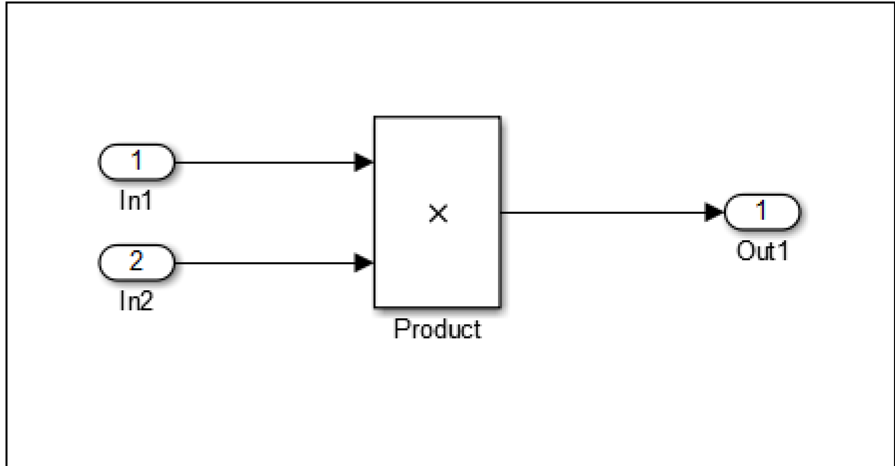
Verify Code from a Simple Simulink Model

In this section...
“Create Simulink Model and Generate Code” on page 7-3
“Run Polyspace Verification” on page 7-5
“View Results in Polyspace Code Prover” on page 7-5
“Trace Error to Simulink Model” on page 7-6
“Specify Signal Ranges” on page 7-7
“Verify Updated Model” on page 7-9

Create Simulink Model and Generate Code

To create a simple Simulink model and generate code:

1. Open MATLAB. Then start Simulink software.
2. Construct the following model.



3. Select **File** > **Save**. Then name the model `my_first_model`.
4. Open the configuration parameters for the model and specify the following configuration parameters.

Solver

Option	User Action
Type	Select `Fixed-step`.
Solver	Select `discrete (no continuous states)`.

Code Generation

Option	User Action
System target file	Enter `ert.tlc` for Embedded Coder.

Code Generation > Report

Option	User Action
Create code-generation report	Select the box.

Code Generation > Templates

Option	User Action
Generate an example main program	Clear the box.

Code Generation > Interface

Option	User Action
Remove error status in real-time model data structure	Select the box.

Optimization

Option	User Action
Remove root level I/O zero initialization	Select the box.

Optimization > Signals and Parameters

Option	User Action
Default parameter behavior	Select `Inlined`.

Also, in the **All parameters** tab, search for the parameter **Use memset to initialize floats and doubles to 0.0**, clear the box, then search for the parameter **Code-to-model**, and select the box.

For more information, see "Recommended Model Settings for Code Analysis".

5 To generate code, from the Simulink model window, select **Code** > **C/C++ Code** > **Build Model**.

Run Polyspace Verification

1 From the Simulink model window, select **Code** > **Polyspace** > **Verify Code Generated for** > **Model**.

   The verification starts, and you see messages in the MATLAB Command Window.

```
### Starting Polyspace verification for Embedded Coder
### Creating results folder results_my_first_model for system my_first_model
### Parameters used for code verification:
    System                : my_first_model
    Results Folder        : C:\results_my_first_model
    Additional Files      : 0
    Verifier settings     : PrjConfig
    DRS input mode        : DesignMinMax
    DRS parameter mode    : None
    DRS output mode       : None
    Model Reference Depth : Current model only
    Model by Model        : 0

...
```

2 Follow the progress of the verification in the MATLAB Command window.

Note Verification of this model takes about a minute. A 3,000 block model will take approximately one hour to verify, or about 15 minutes for each 2,000 lines of generated code.

View Results in Polyspace Code Prover

When the verification is complete, you can view the results using the Polyspace Code Prover interface.

1 From the Simulink model window, select **Code** > **Polyspace** > **Open Results** > **For Generated Code**.

   After a few seconds, Polyspace Code Prover opens.

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

3 Select the orange **Overflow** check.

The **Result Details** pane shows information about the orange check, and the **Source** pane shows the source code containing the orange check.

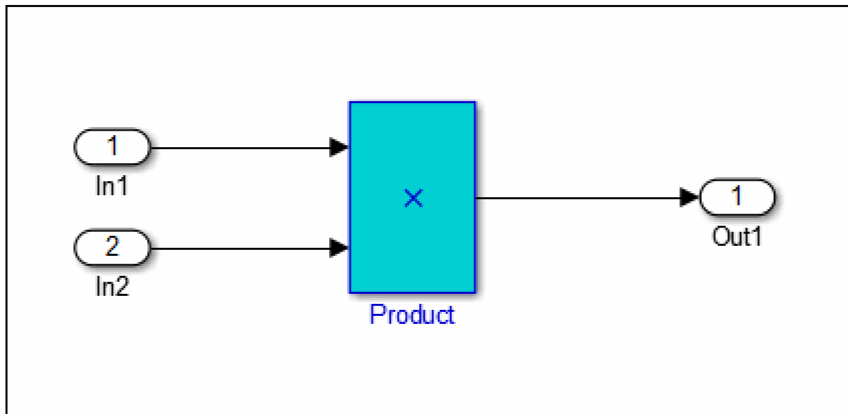
This orange check shows a potential overflow issue when multiplying the signals from the inports `In1` and `In2`. Polyspace considers that the signal values are full range. So multiplying the two signals can result in an overflow.

Trace Error to Simulink Model

To fix this overflow issue, you must return to the Simulink model.

To trace the error to your model:

1 Click the blue underlined link (`<Root>/Product`) immediately before the check in the **Source** pane. The Simulink model opens, highlighting the block with the error.



2 Examine the model. The highlighted block multiplies two full-range signals, which could result in an overflow.

3 To investigate a check, sometimes you have to trace an instance of a variable in generated code back to your model.

   Right-click an identifier and select **Go To Model**. The model shows the corresponding block highlighted in blue. If the block is in a subsystem, both the subsystem and the block are highlighted in blue.

The verification has identified a potential bug. This could be a flaw in:

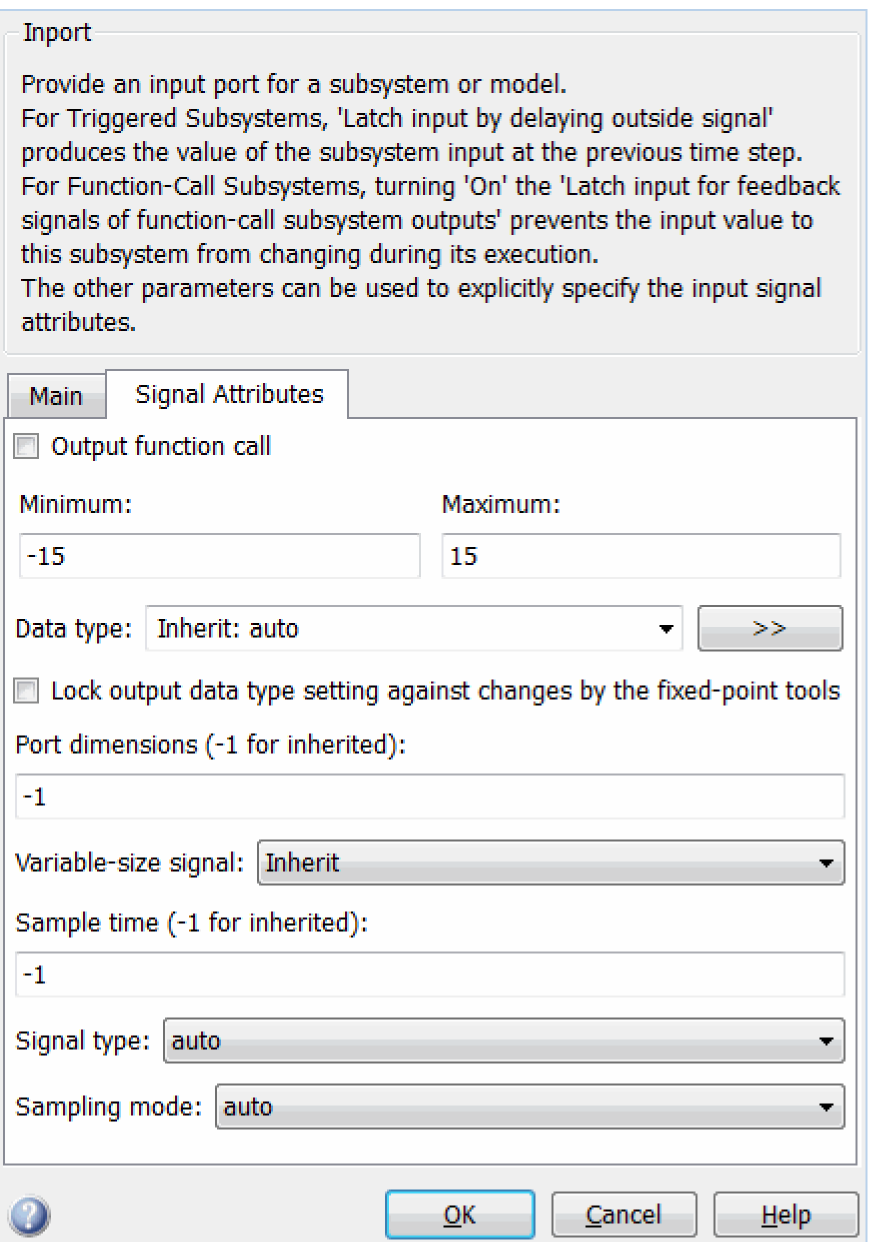
- Design — If the model should be robust for the full signal range, then the issue is a design flaw. In this case, you must change the model to accommodate the full signal range. For example, you could saturate the output of the previous block, or bound the signal with a Switch block.
- Specifications — If the model is supposed to work for specific input ranges, you can provide these ranges using block parameters or the base workspace. The next verification will read these ranges from the model, and the check will be green.

Specify Signal Ranges

If you constrain the input signals in your Simulink model, Polyspace verifies the generated code for these inputs. The **Overflow** check is green in the verification results.

To specify signal ranges using source block parameters:

1. Double-click the `In1` source block in your model. The Source Block Parameters dialog box opens.
2. Select the **Signal Attributes** tab.
3. Set the **Minimum** value for the signal to -15.
4. Set the **Maximum** value for the signal to 15.


Inport
Provide an input port for a subsystem or model.
For Triggered Subsystems, 'Latch input by delaying outside signal' produces the value of the subsystem input at the previous time step.
For Function-Call Subsystems, turning 'On' the 'Latch input for feedback signals of function-call subsystem outputs' prevents the input value to this subsystem from changing during its execution.
The other parameters can be used to explicitly specify the input signal attributes.
Main
Signal Attributes
Output function call
Minimum:
-15
Maximum:
15
Data type:
Inherit: auto
>>
Lock output data type setting against changes by the fixed-point tools
Port dimensions (-1 for inherited):
-1
Variable-size signal:
Inherit
Sample time (-1 for inherited):
-1
Signal type:
auto
Sampling mode:
auto
OK
Cancel
Help

5. Click **OK**.
6. Using above steps, set the minimum values for the `In2` block to -15 and maximum value to 15.
7. Save your model as `my_first_model_bounded`.

Verify Updated Model

After changing the model, you must regenerate code and run verification again.

To regenerate code and rerun the verification:

1. From the Simulink model, select **Code** > **C/C++ Code** > **Build Model**.

   The software generates code for the updated model.
2. Select **Code** > **Polyspace** > **Verify Code Generated for** > **Model**.

   The software verifies the generated code.
3. Select **Code** > **Polyspace** > **Open Results** > **For Generated Code**, which opens Polyspace Code Prover.

   The **Overflow** check is now green. Polyspace verification shows that the generated code does not have run-time errors.

8

Code Verification in IBM Rational Rhapsody Environment

Verify Code in IBM Rational Rhapsody Environment

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

Code Verification Approach

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

With Polyspace Code Prover, you can verify C, C++ and Ada code that you generate from your IBM Rational Rhapsody model (up to version 8.0 supported). As a result, you can detect run-time errors and automatically identify model flaws quickly and early during the design process.

For information about installing and using IBM Rational Rhapsody, go to www-01.ibm.com/software/awdtools/rhapsody/.

The approach for using Polyspace Code Prover within the IBM Rational Rhapsody MDD environment is:

- Integrate the Polyspace add-in with your Rhapsody project. See "Adding Polyspace Profile to Model" on page 8-3.
- If required, specify Polyspace configuration options in the Polyspace verification environment. See "Configuring Verification Options" on page 8-6.
- Specify the `include` path to your operating system (environment) header files and run verification. See "Running a Verification" on page 8-7.

- View results, analyze errors, and locate faulty code within model. See “Viewing Polyspace Results” on page 8-7 and “Locating Faulty Code in Rhapsody Model” on page 8-8.

Adding Polyspace Profile to Model

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

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

---

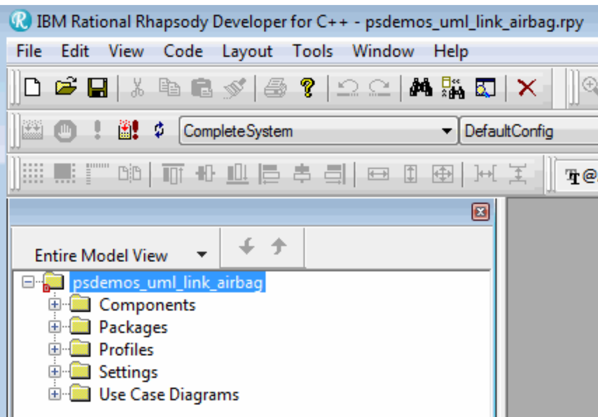
1. In the Rhapsody editor, select **File** > **Add Profile to Model**. The Add Profile to Model dialog box opens.
2. Navigate to the folder *MATLAB_Install*`\polyspace\plugin\rhapsody\profiles\Polyspace`.
3. Select the file `Polyspace.sbs`. Then click **Open**.

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

Accessing Polyspace Features

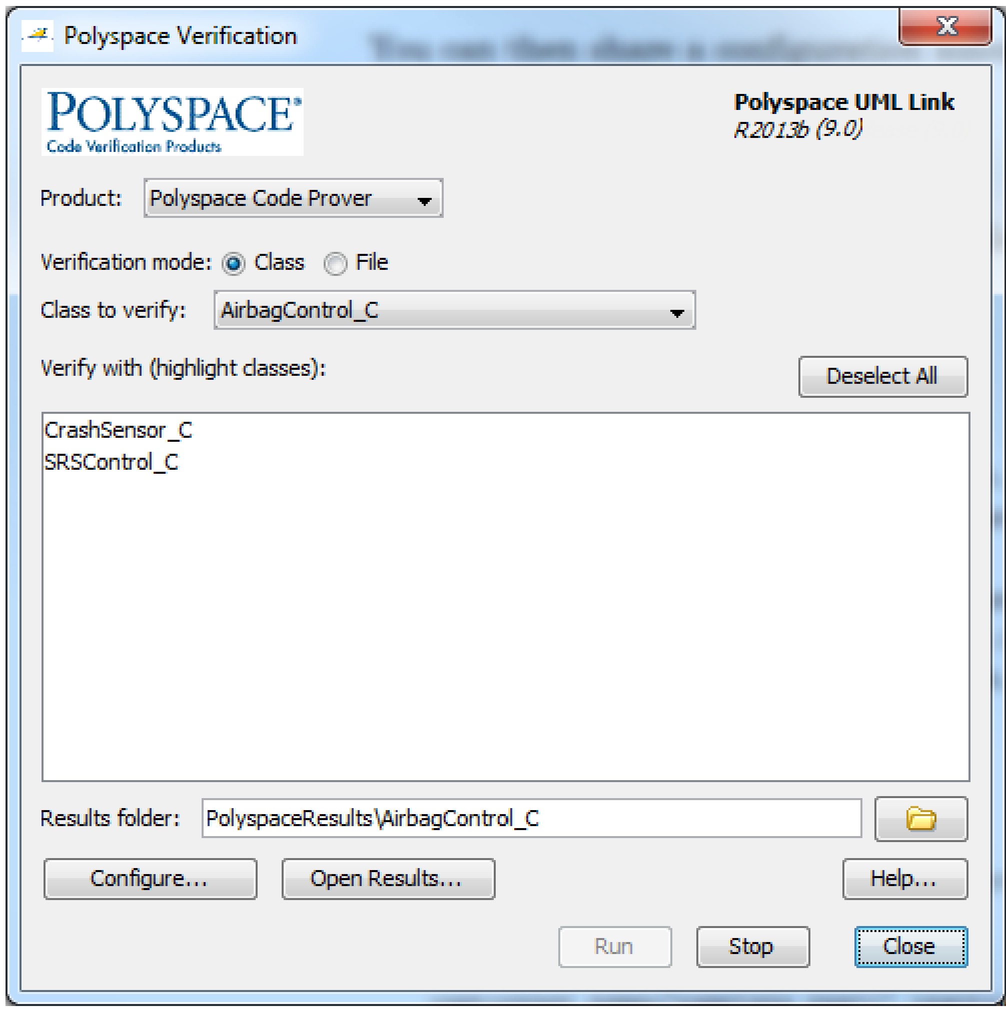
To access Polyspace features in the Rhapsody editor:

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



2. In the **Entire Model View**, expand the `Packages` node.
3. Right-click a package, for example, **AirBagFiles**.
4. From the context menu, select **Polyspace**.

   The Polyspace Verification dialog box opens.



Through the Polyspace Verification dialog box, you can:

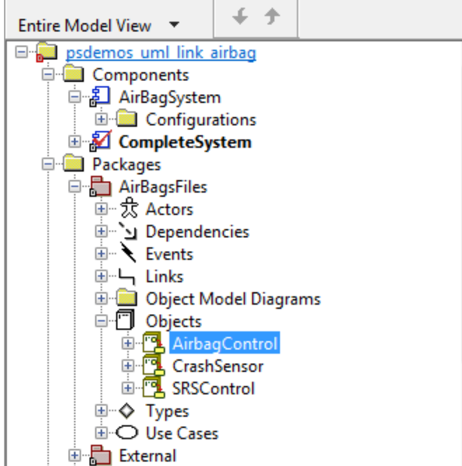
- Specify verification options. See “Configuring Verification Options” on page 8-6.
- Start a verification. See “Running a Verification” on page 8-7.
- Stop a local verification. See “Running a Verification” on page 8-7.
- View verification results. See “Viewing Polyspace Results” on page 8-7.

- Open help.
- Open the Polyspace Job Monitor. See “Running a Verification” on page 8-7.

Configuring Verification Options

To specify options for your verification:

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



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

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

Running a Verification

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

To start a verification:

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

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

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

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

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

Viewing Polyspace Results

To view results from the last local verification:

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

   The software displays results in the Polyspace user interface.

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

For more information, see “Review Analysis Results”.

Declarations for C Functions Without Arguments

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

```
void my_function()
```

rather than:

```
void my_function(void)
```

This can result in the following Polyspace compilation error:

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

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

Locating Faulty Code in Rhapsody Model

To identify the faulty code within your Rhapsody model using Polyspace verification results:

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

> **Tip** For the **Back To Model** command to work, you must have your Rhapsody model open.
>
> The **Back To Model** command works best when the Polyspace check is enclosed by the tags `//#[` and `]#//`.

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

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

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

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

Template Configuration Files

- "Using Template Configuration Files" on page 8-9
- "Default Configuration Options" on page 8-9

Using Template Configuration Files

The first time you perform a verification, the software copies a template, Polyspace configuration file, from *matlabroot*/polyspace/plugin/rhapsody/etc/template_*language*.psprj to the project folder. The software also renames the copy *model*_*language*.psprj, where:

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

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

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

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

Default Configuration Options

The template_*language*.psprj XML files specify the default option values for code verification.

The file template_C.psprj is:

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

```
  </source>
  <include>
  </include>
  <module name="Verification_1" isactive="true">
    <source>
    </source>
    <optionset name="template_psprj" isactive="true">
      <option flagname="-OS-target">no-predefined-OS</option>
      <option flagname="-respect-types-in-fields">true</option>
      <option flagname="-respect-types-in-globals">true</option>
    </optionset>
  </module>
</polyspace_project>
```

The file `template_C++.psprj` is:

```
<?xml version="1.0" encoding="UTF-8"?>
<polyspace_project name="template_psprj" language="C++" author="polyspace"
version="1.0" date="08/04/2011" path="file:/C:/Polyspace/Polyspace_Common
/Rhapsody/PolyspaceUMLLink/etc/template_C++.psprj">
  <source>
  </source>
  <include>
  </include>
  <module name="Verification_1" isactive="true">
    <source>
    </source>
    <optionset name="template_psprj" isactive="true">
      <option flagname="-D">[OM_NO_FRAMEWORK_MEMORY_MANAGER]</option>
      <option flagname="-OS-target">no-predefined-OS</option>
      <option flagname="-dialect">gnu</option>
      <option flagname="-respect-types-in-fields">true</option>
      <option flagname="-respect-types-in-globals">true</option>
      <option flagname="-target">i386</option>
    </optionset>
  </module>
</polyspace_project>
```

9

Using Bug Finder and Code Prover

Differences Between Polyspace Bug Finder and Polyspace Code Prover Analysis

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

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

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

How Bug Finder and Code Prover Complement Each Other

- "Overview" on page 9-3
- "Faster Analysis with Bug Finder" on page 9-3
- "More Exhaustive Verification with Code Prover" on page 9-3
- "More Specific Defect Types with Bug Finder" on page 9-4
- "Easier Setup Process with Bug Finder" on page 9-5
- "Fewer Runs for Clean Code with Bug Finder" on page 9-5
- "Results in Real Time with Bug Finder" on page 9-6
- "More Rigorous Data and Control Flow Analysis with Code Prover" on page 9-6
- "Few False Positives with Bug Finder" on page 9-8

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

- "Zero False Negatives with Code Prover" on page 9-8

Overview

Use both Bug Finder and Code Prover regularly in your development process. The products provide a unique set of capabilities and complement each other. For possible ways to use the products together, see "Workflow Using Both Bug Finder and Code Prover" on page 9-8.

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

Feature	Bug Finder	Code Prover
Number of checkers	178	31 (Critical subset)
Depth of analysis	Fast. For instance: • Faster analysis. • Easier set up and review. • Fewer runs for clean code. • Results in real time.	Exhaustive. For instance: • All operations of a type checked for certain critical errors. • More rigorous data and control flow analysis.
Reporting criteria	Probable defects	Proven findings
Bug finding criteria	Few false positives	Zero false negatives

Faster Analysis with Bug Finder

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

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

More Exhaustive Verification with Code Prover

Code Prover tries to prove the absence of:

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

and so on.

For each operation:

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

  You can use information provided in the Polyspace user interface to diagnose the checks. See "More Rigorous Data and Control Flow Analysis with Code Prover" on page 9-6. You can also provide contextual information to reduce unproven code even further, for instance, constrain input ranges externally.

Bug Finder does not aim for exhaustive analysis. It tries to detect as many bugs as possible and reduce false positives. For critical software components, running a bug finding tool is not sufficient because despite fixing all defects found in the analysis, you can still have errors during code execution (false negatives). After running Code Prover on your code and addressing the issues found, you can expect the quality of your code to be much higher. See "Zero False Negatives with Code Prover" on page 9-8.

More Specific Defect Types with Bug Finder

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

For instance, the statement `if(a=b)` is semantically correct according to the C language standard, but often indicates an unintended assignment. Bug Finder detects such

unintended operations. Although Code Prover does not detect such unintended operations, it can detect if an unintended operation causes other run-time errors.

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

For more information, see:

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

Easier Setup Process with Bug Finder

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

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

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

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

For more information, see "Troubleshoot Compilation and Linking Errors".

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

Fewer Runs for Clean Code with Bug Finder

To guarantee absence of certain run-time errors, Code Prover follows strict rules once it detects a run-time error in an operation. Once a run-time error occurs, the state of your

program is ill-defined and Code Prover cannot prove the absence of errors in subsequent code. Therefore:

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

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

For more information, see "Verification Following Red and Orange Checks".

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

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

Results in Real Time with Bug Finder

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

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

More Rigorous Data and Control Flow Analysis with Code Prover

For each operation in your code, Code Prover provides:

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

For a pointer, the tooltips show the variable that the pointer points to, along with the variable values.

- Graphical representation of the function call sequence that leads to the operation.

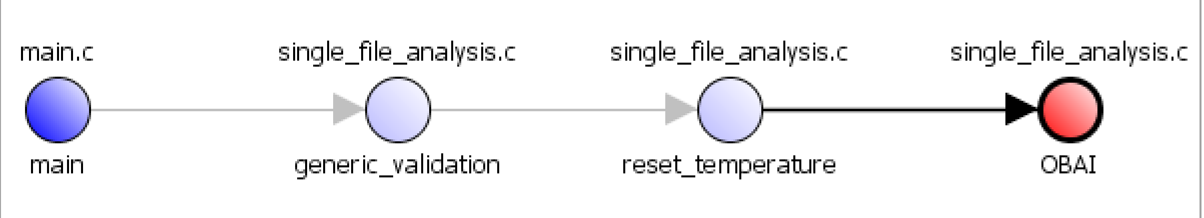
By using this range information and call graph, you can easily navigate the function call hierarchy and understand how a variable acquires values that lead to an error. For instance, for an **Out of Bounds Array Index** error, you can find where the index variable is first assigned values that lead to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

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

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

Dereference of parameter 'beta_pt' (pointer to float 32, size: 32 bits):
Pointer is not null.
Points to 4 bytes at offset 0 in buffer of 4 bytes, so is within bounds (if memory is allocated).
Pointer may point to variable or field of variable:
'beta', local to function 'Square_Root'.
Assignment to dereference of parameter 'beta_pt' (float 32): [0.1 .. 0.5]
Press 'F2' for focus

Data Flow Analysis in Code Prover



Control Flow Analysis in Code Prover

Few False Positives with Bug Finder

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

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

Zero False Negatives with Code Prover

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

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

Workflow Using Both Bug Finder and Code Prover

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

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

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

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

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

Depending on the nature of your software development workflow and available resources, there are many other ways you can incorporate the two kinds of analysis. You can run both products on your desktop during development or as part of automated testing on a remote server. Note that it is easier to interpret and fix bugs closer to development. You will benefit from using both products if you deploy them early and often in your development process.

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

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

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

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

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

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