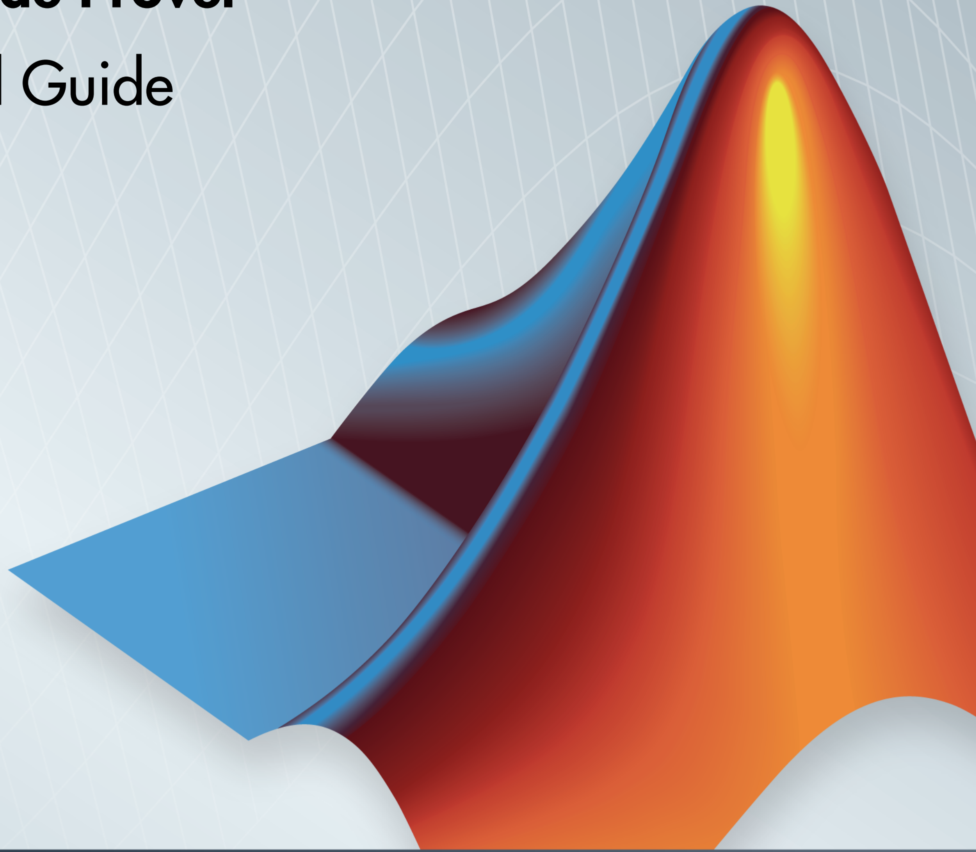
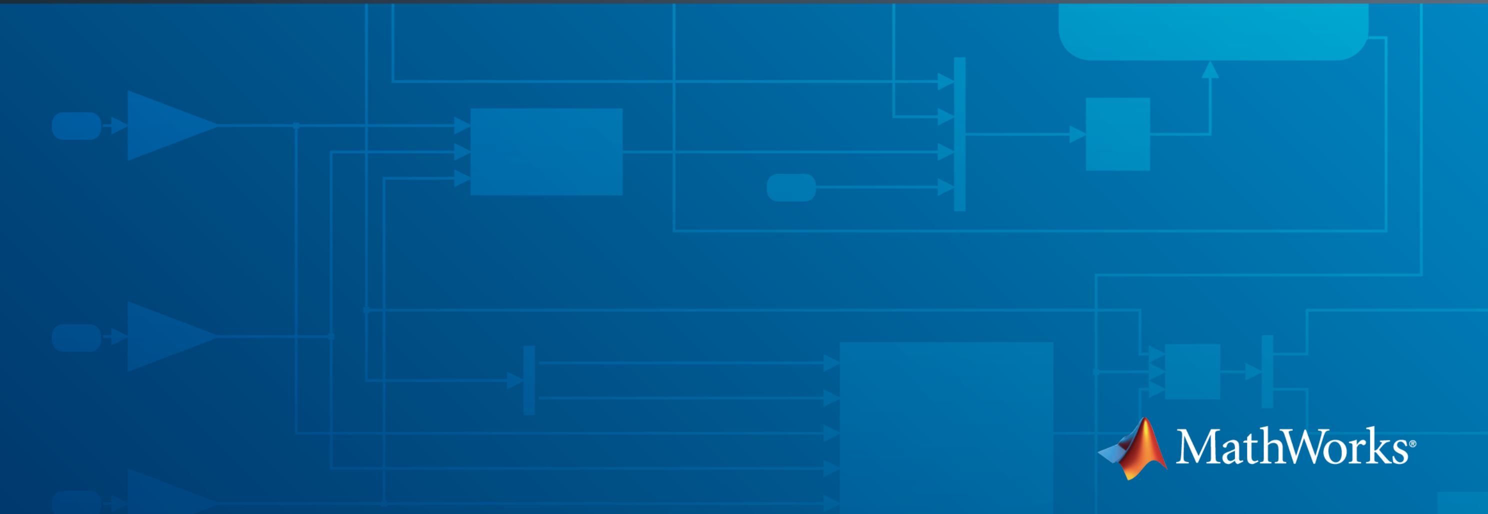
Polyspace® Code Prover™

Getting Started Guide

R2014b



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

Polyspace® Code Prover™ Getting Started Guide

Trademarks

Patents

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)

Contents

1 Introduction to Polyspace Code Prover

2 Set Up a Polyspace Project

3 Server Configuration for Remote Verification and Polyspace Metrics

4 Run a Verification

5 Review Verification Results

6 Check Compliance with Coding Rules

7 Verifying Code Generated from Simulink Models

8 Code Verification in IBM Rational Rhapsody Environment

Introduction to Polyspace Code Prover

Polyspace Code Prover Product Description

Prove the absence of run-time errors in software

Polyspace Code Prover™ 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 static analysis and abstract interpretation based on formal methods. 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...

Polyspace software provides documentation 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 Context Sensitive Help

In addition to the full documentation, you can also access contextual help for analysis options and checks.

- In the **Configuration** window, hover your cursor over an analysis option. In the tooltip, select **More Help** to open context-sensitive help about that option. (Available from the Polyspace interface and all the plug-ins.)
- In the **Check Details** window, select ? to open context-sensitive help about that check. (Only available from the Polyspace interface).

Related Examples

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

- “Configure File and Default Options in Visual Studio”

Set Up a Polyspace Project

Set Up Polyspace Project

In this section...

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** 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, `*MATLAB_Install*` is the MATLAB® installation folder, for example, `C:\Program Files\MATLAB\R2014b`.

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

   - `includes`

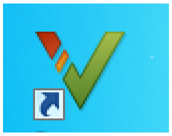
3. Copy `example.c` from `MATLAB_Install\polyspace\examples\cxx\Demo_C_Single-File\sources` to `polyspace_project\sources`.
4. Copy `include.h` from `MATLAB_Install\polyspace\examples\cxx\Demo_C_Single-File\sources` to `polyspace_project\includes`.

Open Polyspace Code Prover

- Open directly in your operating system.

  - Windows®: From the `MATLAB_Install\polyspace\bin` folder, double-click the `polyspace-code-prover` executable.

    You can create a desktop or **Start** menu shortcut to this executable with the icon

    if it does not already exist.
  - Linux® or Mac: Run the following command:

    ```
    /MATLAB_Install/polyspace/bin/polyspace-code-prover
    ```
- Open from MATLAB.

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

Create Project

- "Create New Project" on page 2-3
- "Specify Source Files and Include Folders" on page 2-4

Create New Project

1. Select **File** > **New Project**.
2. In the Project – Properties dialog box:

   - For **Project name**, enter `example_project`.

- Clear the **Use default location** check box. To specify where your `polyspace_project` folder is, click 📁.
- For **Project language**, select **C**.
- Clear the boxes under **Project Configuration**. For more information on the option **Use template**, see “What is a Project Template?”. For more information on the option **Create from build command**, see “Create Projects Automatically from Your Build System”.

3 Click **Next**.

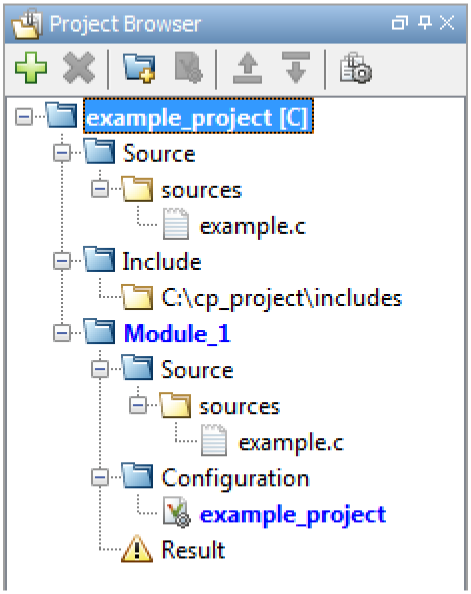
Specify Source Files and Include Folders

1 Select the `sources` folder that you created. Click **Add Source Files**.

2 Select the `includes` folder. Click **Add Include Folders**

Note: Polyspace Code Prover automatically adds standard header files to your project.

3 Click **Finish**. You can see your project in the **Project Browser**.



Next steps

1. "Run Verification"
2. "Review Results"
3. "Find Coding Rule Violations"

Server Configuration for Remote Verification and Polyspace Metrics

Set Up Polyspace Metrics

In this section...

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™ or Polyspace Code Prover
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 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 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.
   - **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 Remote Verification and 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 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 3-6 .

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 3-5.

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 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 *`%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.

Set Up Remote Verification and Analysis

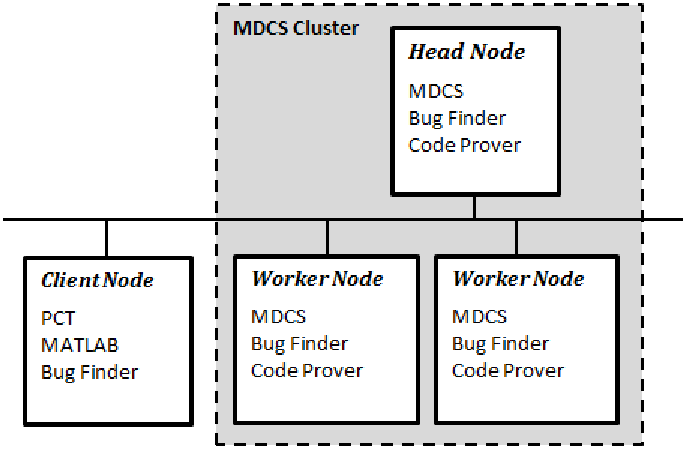
In this section...
“Requirements for Remote Verification and Analysis” on page 3-9
“Start Server for Remote Verification and Polyspace Metrics” on page 3-9
“Configure Polyspace Preferences” on page 3-10

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.
Remote *interactive*	
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 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 **Server Configuration** tab. See, “Configure Polyspace Preferences” on page 3-10.

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 MDCS 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”.

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.
   - **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.
  - **Security level in the cluster** — 0. No security.
  - **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 3-2.

Run a Verification

Run Verification

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". You use the `polyspace_project` folder and the `example_project.psprj` file in this tutorial.
- "Set Up Remote Verification and Analysis" for remote verification and "Set Up Polyspace Metrics" for Polyspace Metrics.

Prepare for Verification

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

1. Select **File** > **Open Project**.
2. In the Open Project dialog box, from the **Look in** drop-down list, navigate to `polyspace_project`.
3. Select the project file `example_project`.
4. Click **Open**.

Run Remote Verification

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

Start Verification

Before you start remote verification, you must perform a one-time setup. See “Set Up Remote Verification and Analysis”.

1. On the **Project Browser** pane, select **Module_1**.
2. On the **Configuration** pane, select **Distributed Computing**.
3. Select **Batch**. By default, this action enables the **Add to results repository** option.
4. On the Project Manager toolbar, click ▷ Run.

   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 4-4
- "Monitor Progress" on page 4-4
- "Stop Verification" on page 4-5

Start Verification

To start a verification on your local computer:

1. In the Project Manager perspective, in the **Project Browser**, select **Module_1**.
2. On the **Configuration** pane, select **Distributed Computing**. Clear **Batch** if it is selected.
3. On the Project Manager toolbar, click ▷ Run.

If the verification fails, see "Troubleshooting in Polyspace Code Prover".

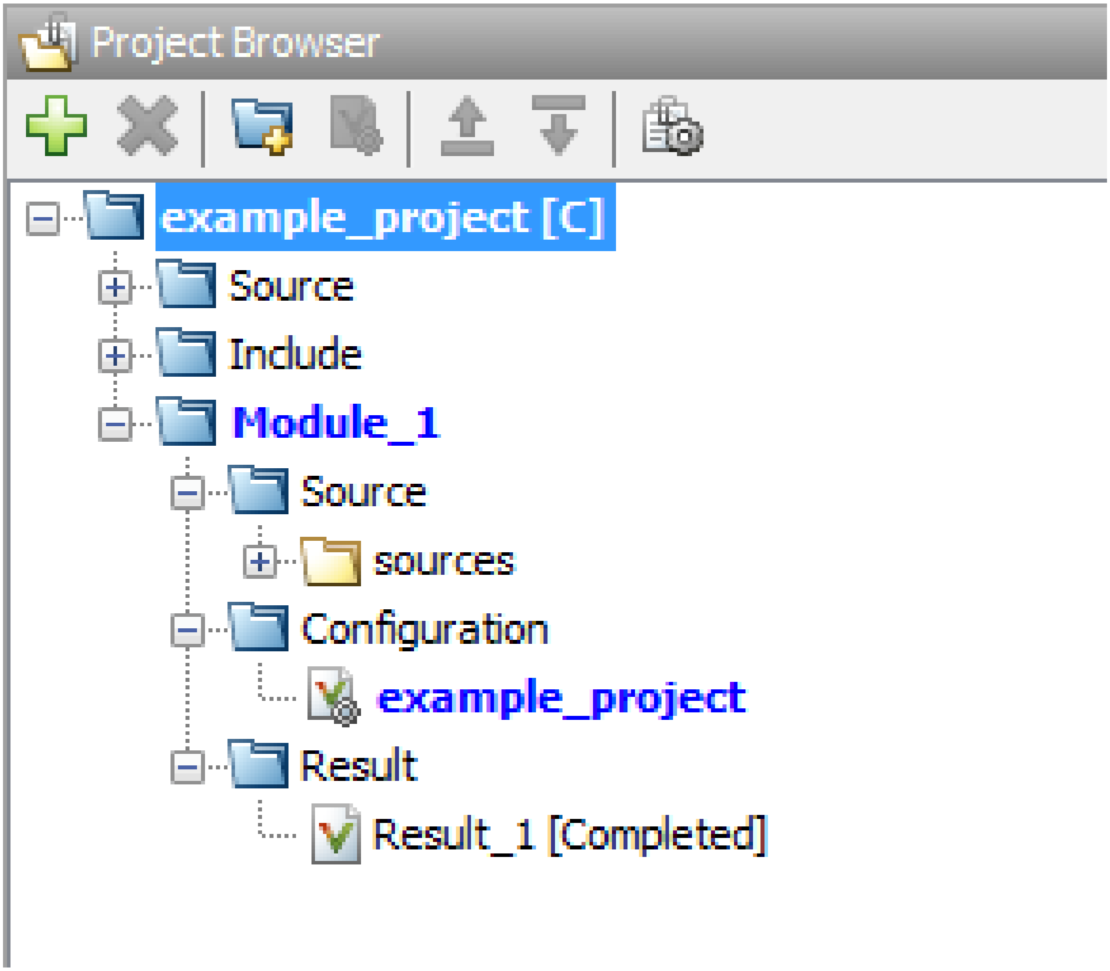
Monitor Progress

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

- **Output Summary**
- **Full Log**

When the verification is complete, you see:

- The message `Verification process completed` in the **Output Summary**.
- The results file, for example `Result_1`, in the **Project Browser**.



- Statistics, such as **Code covered by verification** and **Check Distribution** in the **Dashboard** tab.

Stop Verification

To stop a local verification:

1 On the Project Manager 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"
2. "Find Coding Rule Violations"

5

Review Verification Results

Review Results

In this section...

Tutorial Overview

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

Open Results

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 Polyspace Metrics Project Index".

2 Click the **Project** cell of your verification.

You can see a summary of your project.

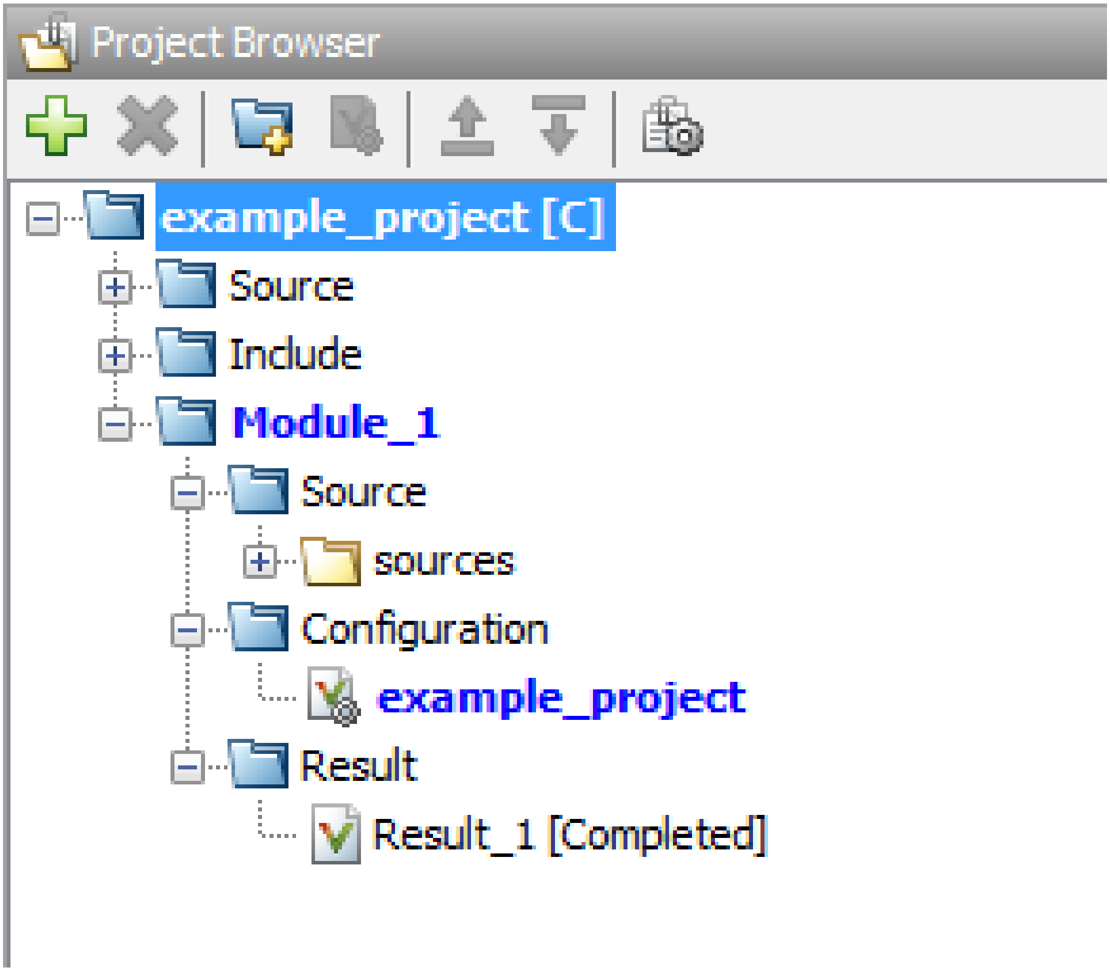
3 On the **Summary** tab, click the **1.0** cell in the **Verification** column.

Your results are downloaded into the user interface.

Local Verification

Do one of the following:

- If your project is open in the **Project Browser**, double-click the results file **Result_1**.



The software opens the results in the Results Manager perspective.

- If your project is not open in the **Project Browser**:

1 Select **File** > **Open Result**.

2 In the Open Results dialog box, navigate to the results folder:

```
polyspace_project\Module_1\Result_1\example_project
```

3 Select `example_project.pscp`.

4 Click **Open**.

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 Summary** pane, select **Group by** > **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.

Function	Check	Source Code Appearance	Reason
`Unreachable_Code`	Gray **Unreachable code**	The code within braces starting from line 193 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 178 is red.	`beta` is less than 0.75. So the argument to `sqrt` is always negative.
`Non_Infinite_Loo`	First green **Overflow**	The + sign on line 73 is green.	When `y` is too large, the `while` loop

Function	Check	Source Code Appearance	Reason
			terminates. So the operation `x=x+2` never overflows.
`Recursion`	Orange **Division by Zero**	The / sign on line 132 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:

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

4 Filter **Illegally dereferenced pointer** checks. To do this:

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

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

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

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

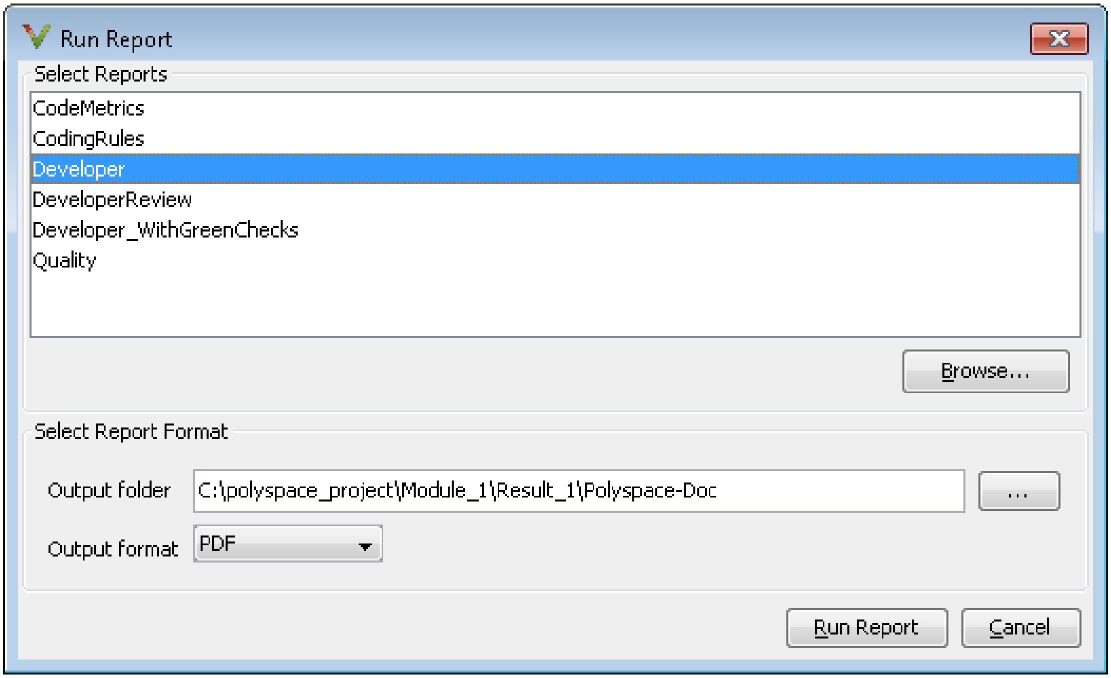
Column	Action
Classification	`High`
Status	`Fix`
Comment	`p points outside array.`

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"

Check Compliance with Coding Rules

Find Coding Rule Violations

In this section...

Tutorial Overview

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

Applying coding rules:

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

Before you start, you must “Set Up Polyspace Project”.

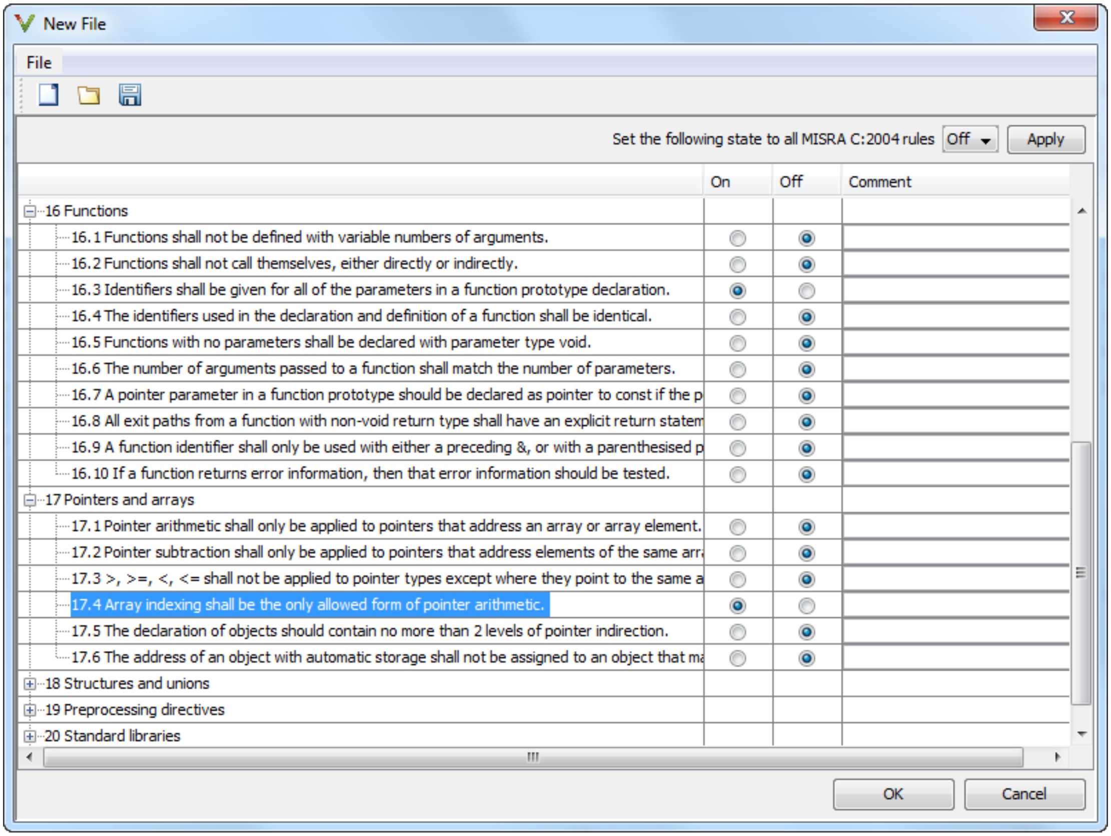
Specify MISRA C Checking

To set the MISRA C checking option:

1. On the **Project Browser**, under **Module_1** > **Configuration**, select **example_project**.
2. On the **Configuration** pane, select **Coding Rules**. Select **Check MISRA C:2004 rules**.
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, from the **Set the following state to MISRA C:2004 rules** drop-down list, select `Off`. Click **Apply**.
6. Select ● for the following rules.

Rule Number	Rule description
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.

Rule Number	Rule description
17.4	Array indexing shall be the only allowed form of pointer arithmetic.



Set the following state to all MISRA C:2004 rules Off Apply

	On	Off	Comment
16 Functions			
16.1 Functions shall not be defined with variable numbers of arguments.	○	●	
16.2 Functions shall not call themselves, either directly or indirectly.	○	●	
16.3 Identifiers shall be given for all of the parameters in a function prototype declaration.	●	○	
16.4 The identifiers used in the declaration and definition of a function shall be identical.	○	●	
16.5 Functions with no parameters shall be declared with parameter type void.	○	●	
16.6 The number of arguments passed to a function shall match the number of parameters.	○	●	
16.7 A pointer parameter in a function prototype should be declared as pointer to const if the p	○	●	
16.8 All exit paths from a function with non-void return type shall have an explicit return statem	○	●	
16.9 A function identifier shall only be used with either a preceding &, or with a parenthesised p	○	●	
16.10 If a function returns error information, then that error information should be tested.	○	●	
17 Pointers and arrays			
17.1 Pointer arithmetic shall only be applied to pointers that address an array or array element.	○	●	
17.2 Pointer subtraction shall only be applied to pointers that address elements of the same arr	○	●	
17.3 >, >=, <, <= shall not be applied to pointer types except where they point to the same a	○	●	
17.4 Array indexing shall be the only allowed form of pointer arithmetic.	●	○	
17.5 The declaration of objects should contain no more than 2 levels of pointer indirection.	○	●	
17.6 The address of an object with automatic storage shall not be assigned to an object that ma	○	●	
18 Structures and unions			
19 Preprocessing directives			
20 Standard libraries			

Click **OK** to save the file.

7 On the Project Manager toolbar, click Run .

Review MISRA C Violations

To examine the MISRA C violations:

1 In the **Project Browser** Result folder, double-click the results file.

The results open in the Results Manager perspective.

2 On the **Results Summary** pane, select **Group by** > **Family**.

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

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

Pane	Result
Source	The line containing the rule violation is highlighted.
Check Details	The following information is displayed: • Description of violated rule. • File and function where the rule violation appears.

4 On the **Source** pane, right-click the highlighted code. Select **Open Source File**.

The `example.c` file opens on the **Code Editor** tab or in an external text editor depending on your **Preferences**.

5 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

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 “Model Configuration for Code Generation and Analysis”.
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 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

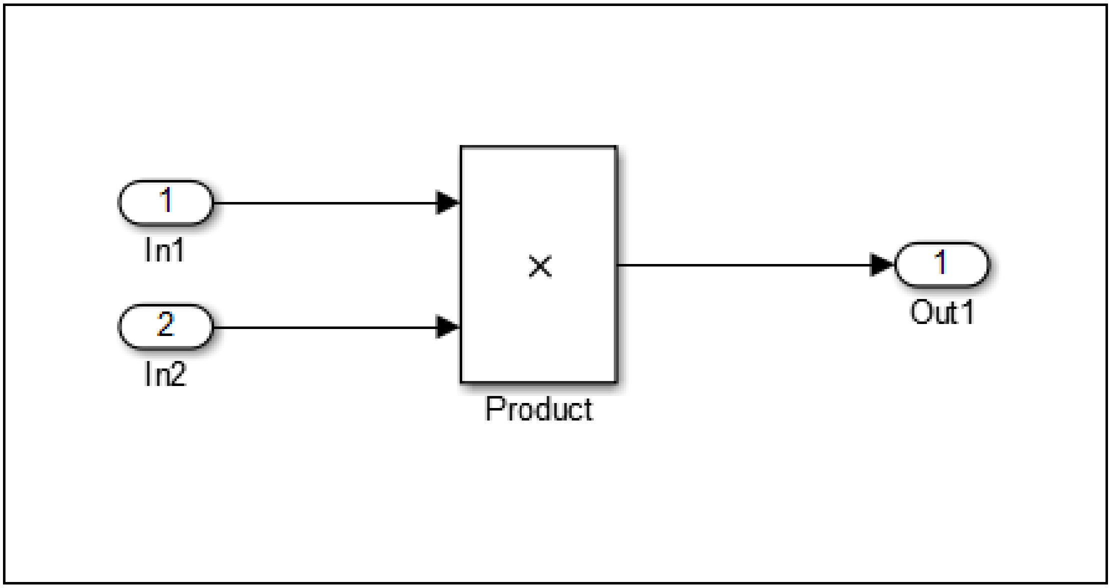
In this section...

Create Simulink Model and Generate Code

To create a simple Simulink model and generate code:

1 Open MATLAB. Then start Simulink software.

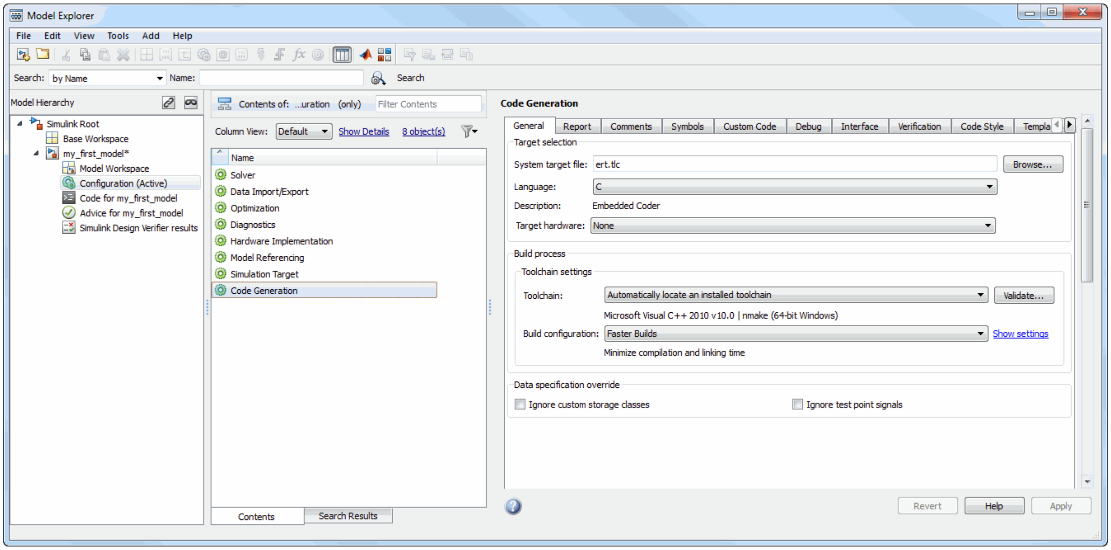
2 Construct the following model.



3 Select **File** > **Save**. Then name the model `my_first_model`.

4 Select **Tools > Model Explorer**. The Model Explorer opens.

5 From the **Model Hierarchy** tree, expand the node **my_first_model**. Select **Configuration**.



System target file: ert.tlc

Language: C

Description: Embedded Coder

Target hardware: None

Toolchain: Automatically locate an installed toolchain

Microsoft Visual C++ 2010 v10.0 | nmake (64-bit Windows)

Build configuration: Faster Builds

Minimize compilation and linking time

☐ Ignore custom storage classes ☐ Ignore test point signals

6 Select the **Configuration** for **Code Generation**. Specify the following code generation options. Click **Apply** to save your options.

Tab	Group	Option	User Action
General	**Target selection**	**System target file**	Enter `ert.tlc` for Embedded Coder.
Report		**Create code-generation report**	Select the box.
		Code-to-model	Select the box.
Templates	**Custom templates**	**Generate an example main program**	Clear the box.
Interface	**Code interface**	**Suppress error status in real-time model data structure**	Select the box.

7 Select the **Configuration** for **Solver**. Specify the following solver options. Click **Apply** to save your options.

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

8 Select the **Configuration** for **Optimization**. Specify the following optimization options. Click **Apply** to save your options.

Tab	Group	Option	User Action
General	**Data initialization**	**Remove root level I/O zero initialization**	Select the box.
		Use memset to initialize floats and doubles to 0.0	Clear the box.
Signals and Parameters	**Simulation and code generation**	**Inline parameters**	Select the box.

9 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 Results Manager perspective of the Polyspace Code Prover.

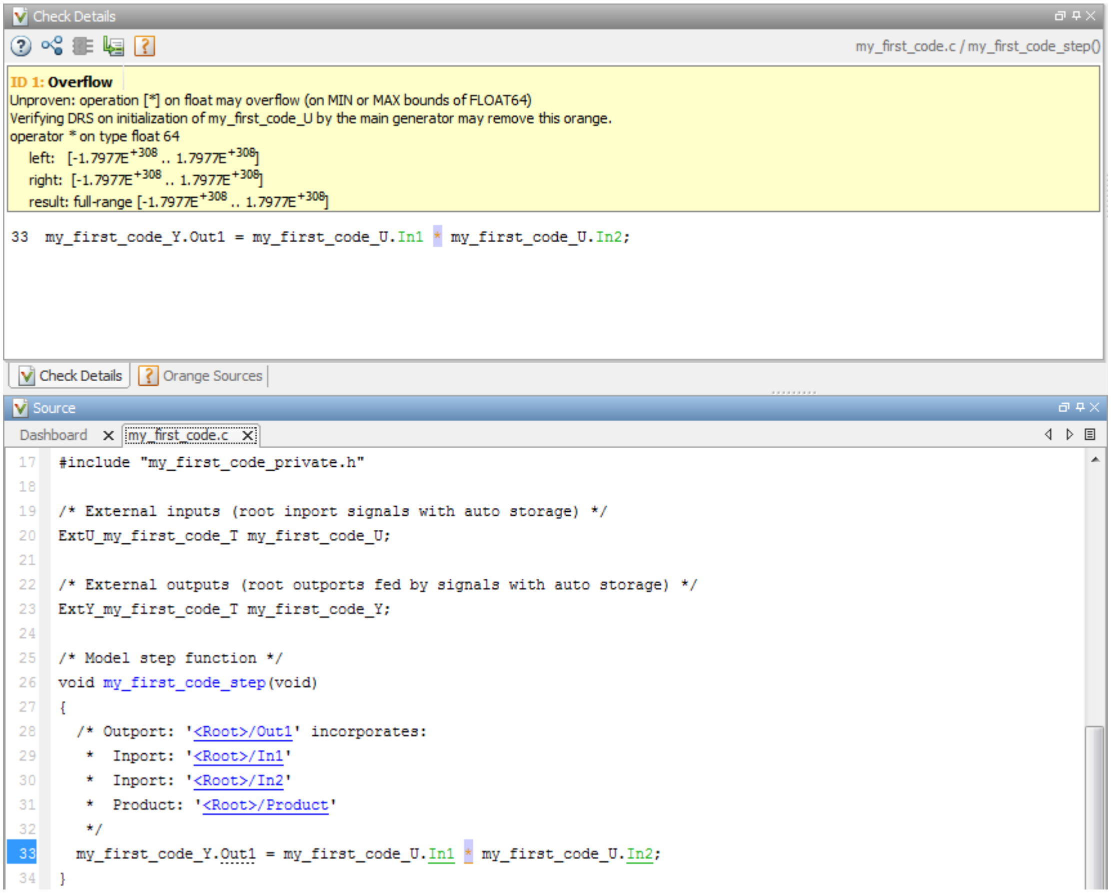
1 From the Simulink model window, select **Code** > **Polyspace** > **Open Results**.

   After a few seconds, the Results Manager perspective opens.

2 On the **Results Summary** pane, select **Group by** > **None**.

3 Select the orange **Overflow** check.

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



ID 1: Overflow
Unproven: operation [*] on float may overflow (on MIN or MAX bounds of FLOAT64)
Verifying DRS on initialization of my_first_code_U by the main generator may remove this orange.
operator * on type float 64
left: [$-1.7977E^{+308}$ .. $1.7977E^{+308}$]
right: [$-1.7977E^{+308}$ .. $1.7977E^{+308}$]
result: full-range [$-1.7977E^{+308}$ .. $1.7977E^{+308}$]

```
my_first_code_Y.Out1 = my_first_code_U.In1 * my_first_code_U.In2;
```

```
#include "my_first_code_private.h"

/* External inputs (root inport signals with auto storage) */
ExtU_my_first_code_T my_first_code_U;

/* External outputs (root outports fed by signals with auto storage) */
ExtY_my_first_code_T my_first_code_Y;

/* Model step function */
void my_first_code_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Product: '<Root>/Product'
   */
  my_first_code_Y.Out1 = my_first_code_U.In1 * my_first_code_U.In2;
}
```

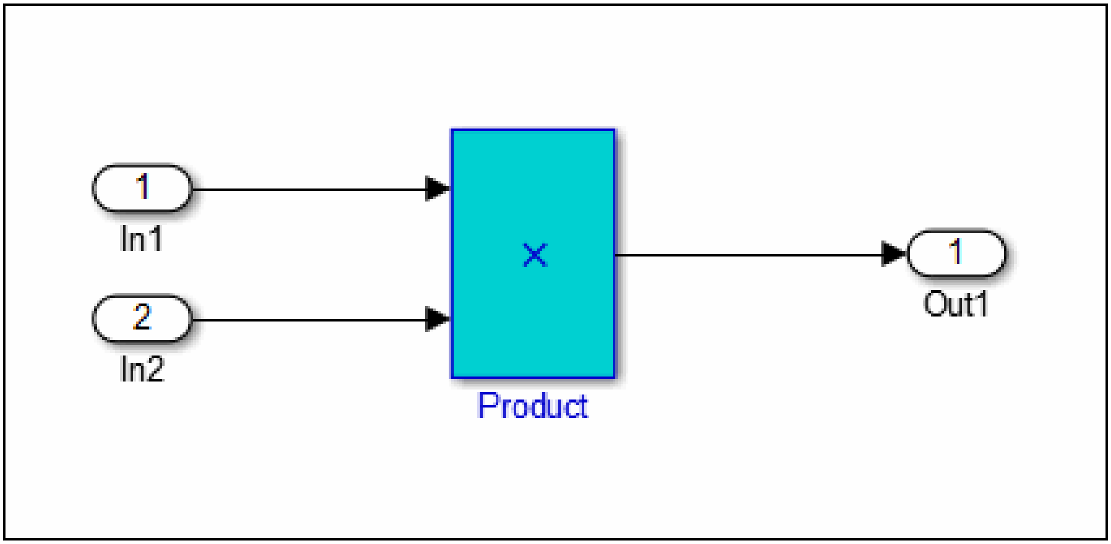
This orange check shows a potential overflow issue when multiplying the signals from the inports `In1` and `In2`. Polyspace considers that the signal values are full range. So multiplying the two signals can result in an overflow.

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.

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 inputa. The **Overflow** check is green in the verification results.

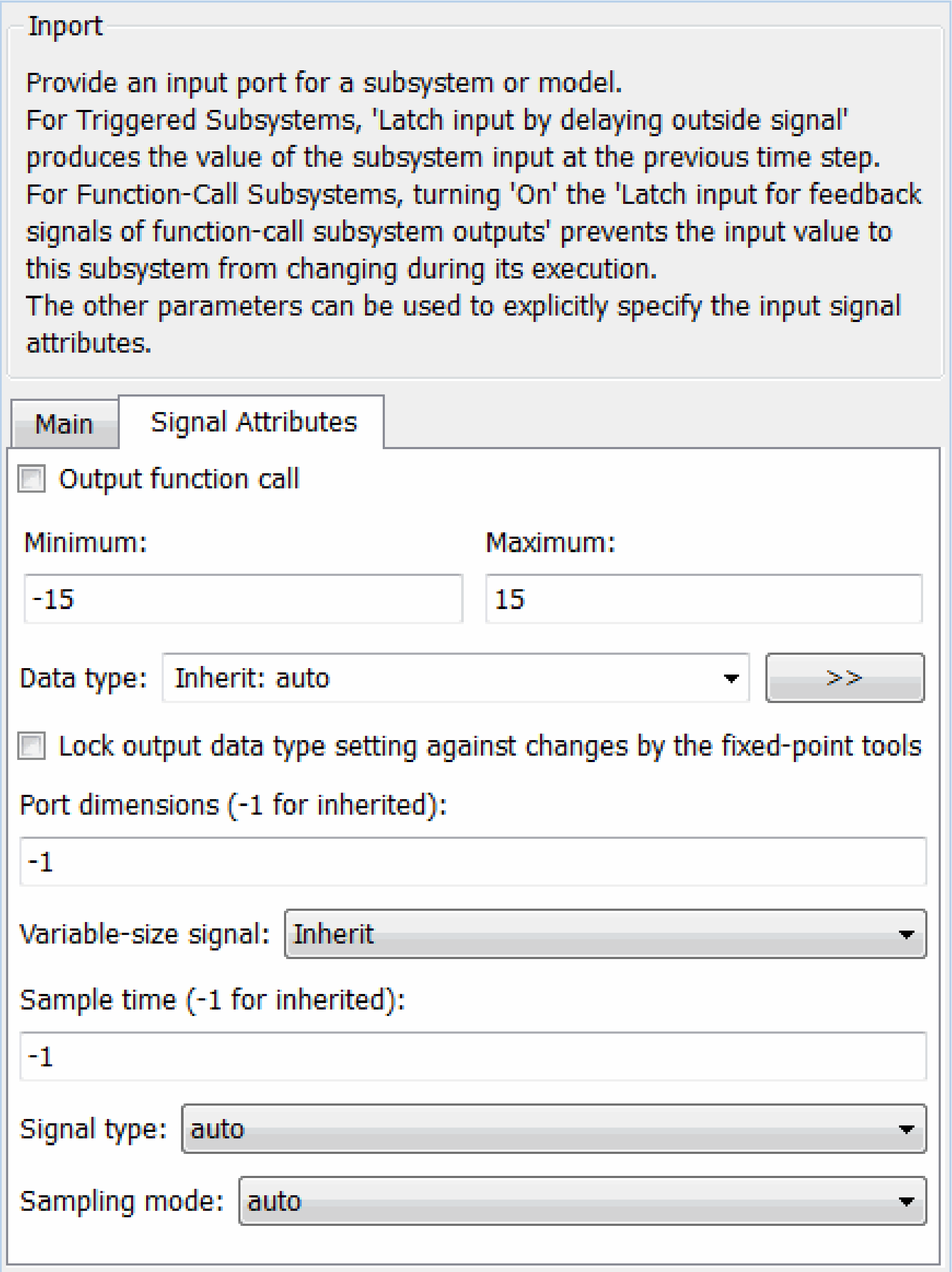
To specify signal ranges using source block parameters:

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.

□ 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

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**, 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

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. 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.

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`.

---

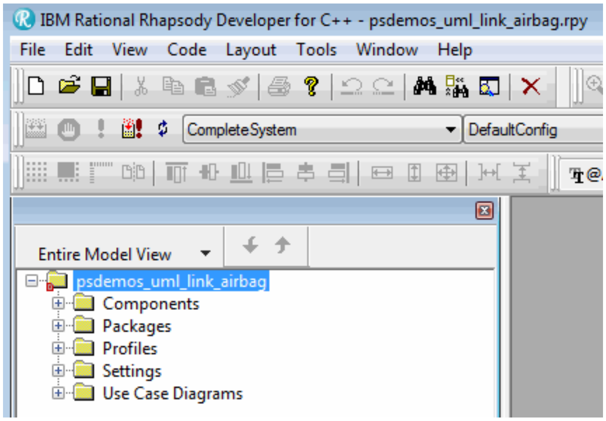
1. In the Rhapsody editor, select **File** > **Add Profile to Model**. The Add Profile to Model dialog box opens.
2. Navigate to the folder *`MATLAB_Install`*`\polyspace\plugin\rhapsody\profiles\Polyspace`.
3. Select the file `Polyspace.sbs`. Then click **Open**.

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`.

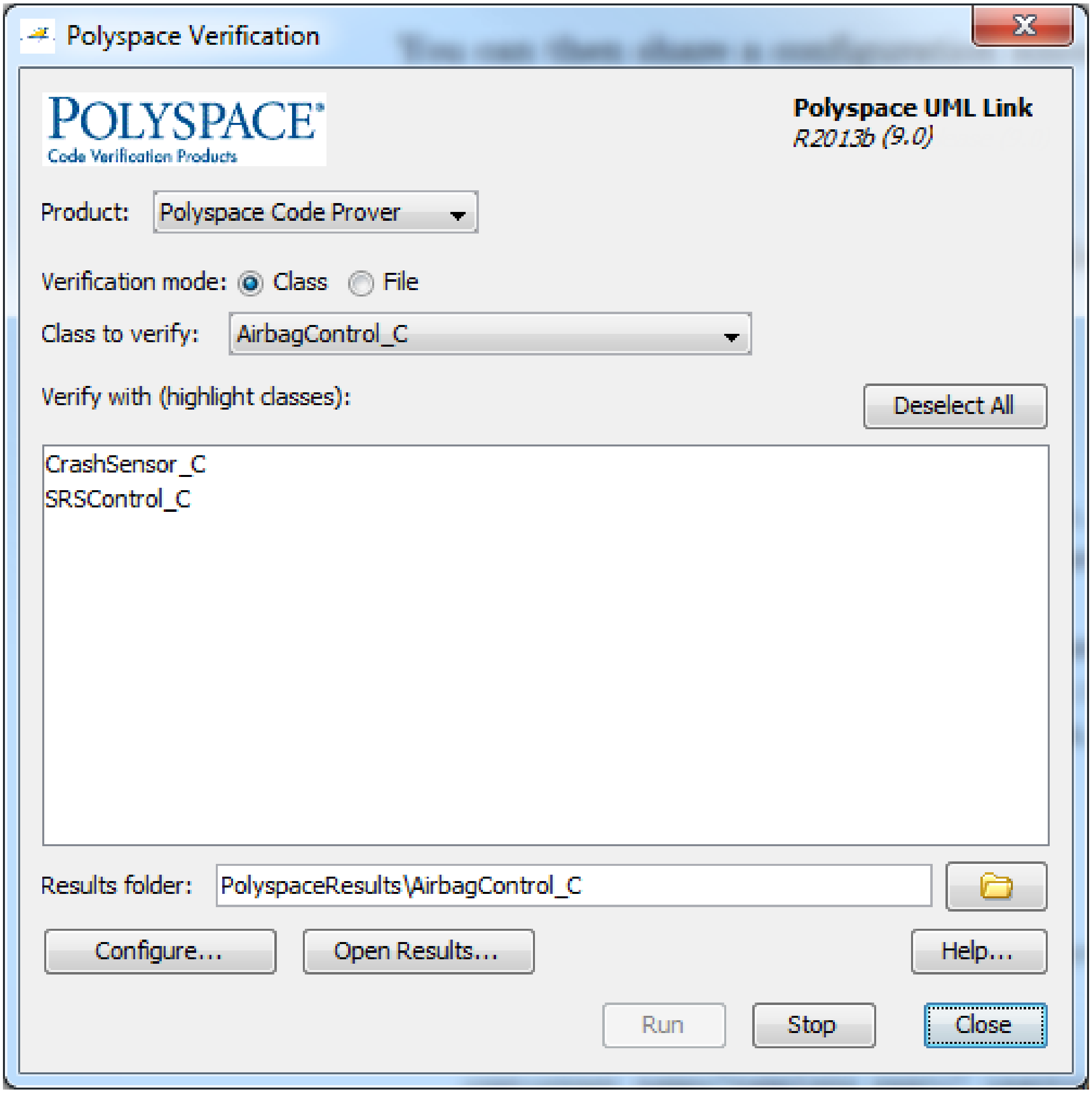


2 In the **Entire Model View**, expand the `Packages` node.

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

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

The Polyspace Verification dialog box opens.



Polyspace Verification

POLYSPACE® Code Verification Products

Polyspace UML Link
R2013b (9.0)

Product: Polyspace Code Prover

Verification mode: ● Class ○ File

Class to verify: AirbagControl_C

Verify with (highlight classes):

CrashSensor_C
SRSControl_C

Results folder: PolyspaceResults\AirbagControl_C

Through the Polyspace Verification dialog box, you can:

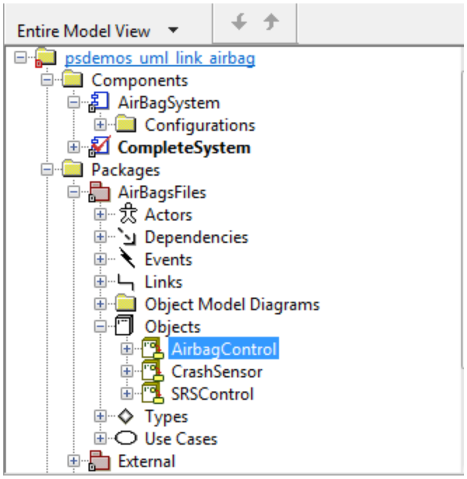
- Specify verification options. See “Configuring Verification Options” on page 8-6.

- 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** > **Target operating system** (`-OS-target`)
- **Target & Compiler** > **Dialect** (`-dialect`)
- **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 for C Code"
- "Analysis Options for C++ Code"

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 Results Manager perspective.

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

For more information, see “Run-Time Error Review”.

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 the Results Manager perspective, navigate to an error, for example.
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 8-10

Using Template Configuration Files

The first time you perform a verification, the software copies a template, Polyspace configuration file, from `*Polyspace_Install*/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="-allow-undef-variables">true</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="-allow-undef-variables">true</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>
```