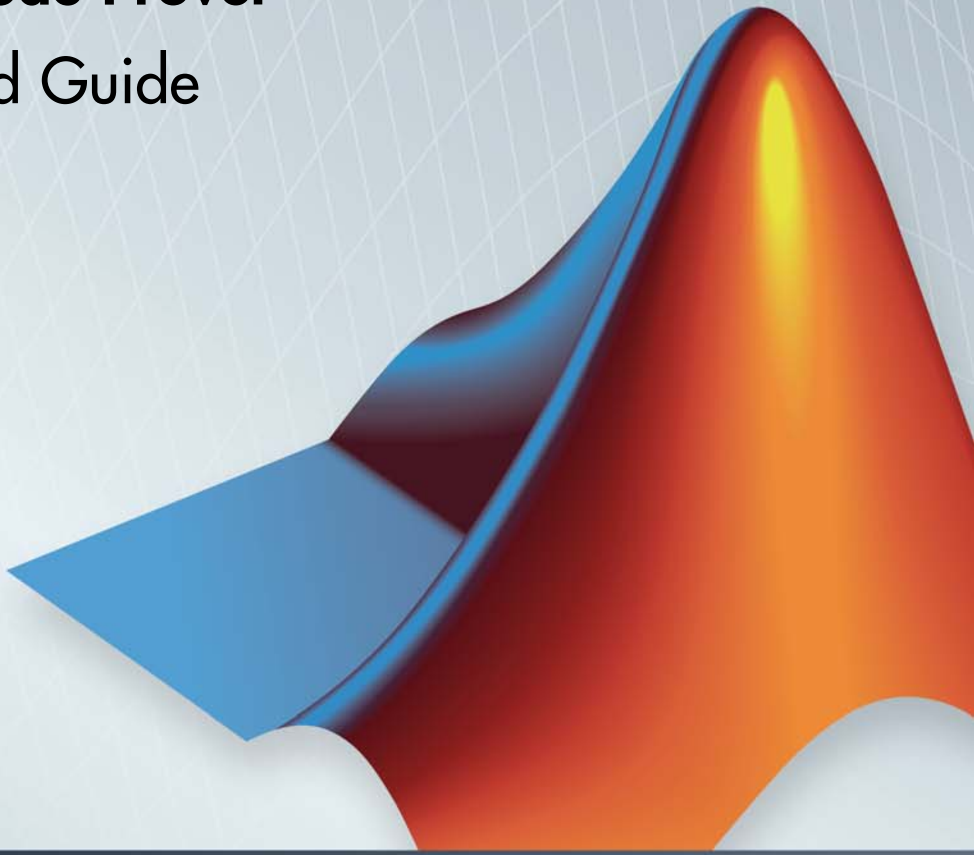
Polyspace® Code Prover™

Getting Started Guide

R2013**b**



MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Polyspace® Code Prover™ Getting Started Guide

Revision History

September 2013	Online only	Revised for Version 9.0 (Release 2013b)

Contents

4 Run a Verification

5 Review Verification Results

6 Check Compliance with Coding Rules

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 conditions under 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 conditions under which variables 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

Set Up a Polyspace Project

Set Up Polyspace Project

In this section...

Tutorial Overview

Before you can run a verification of your source code, you must have a project. In this tutorial, you create the project that you use to run verifications in later tutorials.

What Is a Project?

Polyspace Code Prover project is a set of parameters that specify the verification of your software project source files. A project includes:

- **Source** files.
- **Include** folders.
- One or more modules. Each module has the following folders:
  - **Source** — Specific versions of source files used in the verification
  - **Configuration** — One or more configurations. Each configuration specifies a set of verification options.
  - **Result** — Verification results.

Through the Project Manager perspective, you can create your own project, use an existing project, or modify a project.

In this tutorial, you create a new project and save the project (`.psprj`) file.

Prepare Project Folders

Before you verify a C source file with Polyspace Code Prover software, you must know the locations of the C source file and the include files. You must also know where you want to store the verification results.

For each project, decide where to store source files and results. For example, you can create a project folder, and then in that folder, create separate folders for the source files, include files, and results.

For this tutorial, prepare a project folder as follows:

1. Create a project folder named `polyspace_project`.
2. Open `polyspace_project`, and create the following folders:
   - `sources`
   - `includes`
3. Copy the file `example.c` and `single_file_analysis.c` from

   `MATLAB_Install``\polyspace\examples\cxx\Demo_C_Single-File\sources`

   to

   `polyspace_project\sources`

   `MATLAB_Install` is the MATLAB® installation folder, for example, `C:\Program Files\MATLAB\R2013b`.
4. Copy the files `include.h`, `math.h`, `single_file_analysis.h` and `single_file_private.h` from

   `MATLAB_Install``\polyspace\examples\cxx\Demo_C_Single-File\sources`

   to

   `polyspace_project\includes`.

Open Polyspace Code Prover

In Windows®, do one of the following:

- From the folder *`MATLAB_Install`*`\polyspace\bin`, double-click the Polyspace Code Prover icon.
- Double-click a desktop Polyspace Code Prover shortcut. To create a shortcut, in the folder *`MATLAB_Install`*`\polyspace\bin`, right-click `polyspace-code-prover`. Then, from the context menu, select **Create shortcut**.
- In a DOS command window, enter:

  ```
  MATLAB_Install\polyspace\bin\polyspace-code-prover
  ```

  `MATLAB_Install` is your MATLAB installation folder, for example:

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

- From the MATLAB apps gallery, click the Polyspace Code Prover app.

In Linux®, do one of the following:

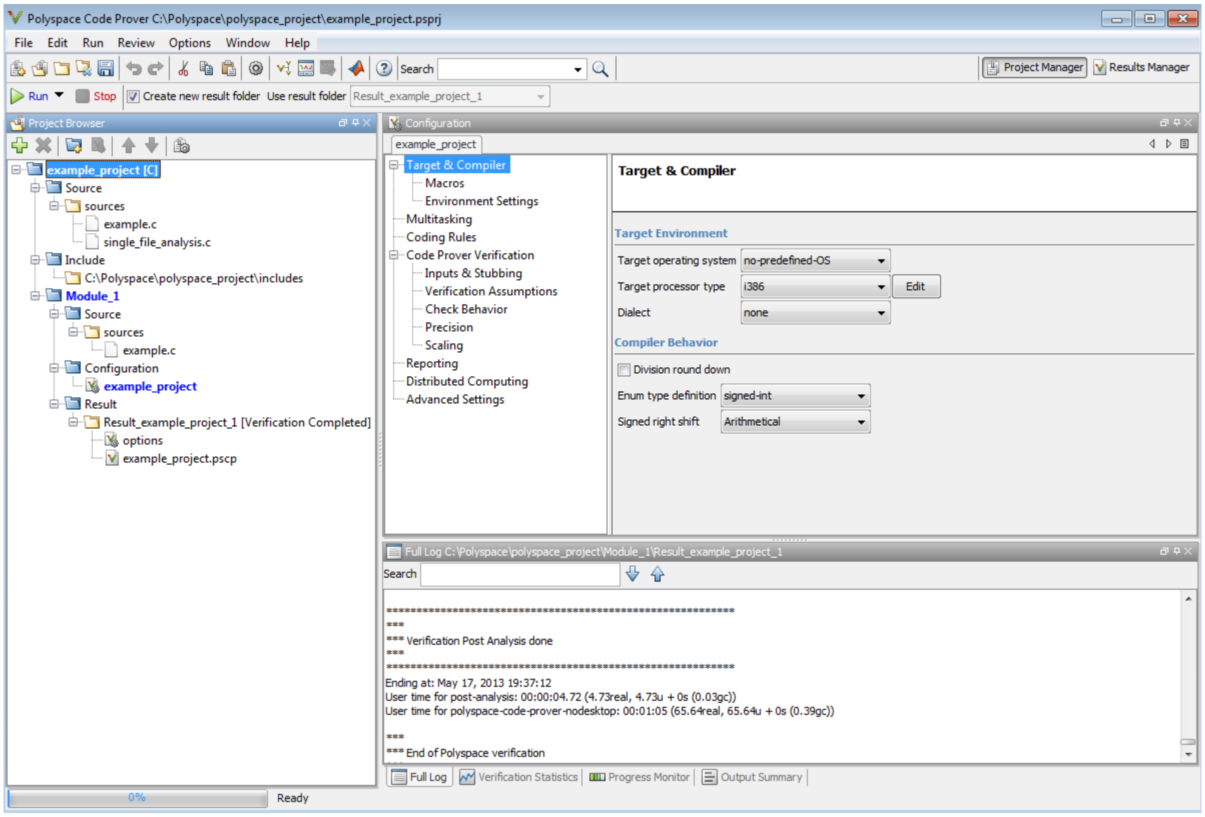
- Run the following command:

  ```
  /MATLAB_Install/polyspace/bin/polyspace-code-prover
  ```

- From the MATLAB apps gallery, click the Polyspace Code Prover app.

Note If MATLAB is not open, you can open MATLAB from Polyspace Code Prover. On the toolbar, click the icon.

By default, the Polyspace Code Prover displays the Project Manager perspective. The Project Manager perspective has three main display sections.



example_project [C]
- Source
  - sources
    - example.c
    - single_file_analysis.c
- Include
  - C:\Polyspace\polyspace_project\includes
- Module_1
  - Source
    - sources
      - example.c
  - Configuration
    - example_project
  - Result
    - Result_example_project_1 [Verification Completed]
      - options
      - example_project.pscp

- Target & Compiler
  - Macros
  - Environment Settings
- Multitasking
- Coding Rules
- Code Prover Verification
  - Inputs & Stubbing
  - Verification Assumptions
  - Check Behavior
  - Precision
  - Scaling
- Reporting
- Distributed Computing
- Advanced Settings

Target & Compiler

Target Environment

Target operating system: no-predefined-OS
Target processor type: i386 (Edit)
Dialect: none

Compiler Behavior

Division round down: ☐
Enum type definition: signed-int
Signed right shift: Arithmetical

```
****************************************************************
***
*** Verification Post Analysis done
***
****************************************************************
Ending at: May 17, 2013 19:37:12
User time for post-analysis: 00:00:04.72 (4.73real, 4.73u + 0s (0.03gc))
User time for polyspace-code-prover-nodesktop: 00:01:05 (65.64real, 65.64u + 0s (0.39gc))

***
*** End of Polyspace verification
```

Display section	For
Project Browser	Specifying: • Source files • Include folders • Results folder
Configuration	Specifying verification options
Output	Monitoring the progress of a verification, viewing status, log messages, and general verification statistics.

You can resize or hide any of these sections. You learn more about the Project Manager perspective later in this tutorial.

Create a New Project to Verify the Example C File

To run a verification, you must have a project, saved with file type `psprj`, . In this part of the tutorial, you create a new project for verifying `example.c` and `single_file_analysis.c`.

To create a new project, you:

- "Open a New Project" on page 2-6
- "Specify Source Files and Include Folders" on page 2-8
- "Specify Target Environment" on page 2-9
- "Specify Analysis Options" on page 2-10
- "Save the Project" on page 2-10

Open a New Project

To open a new project for verifying `example.c` and `single_file_analysis.c`:

1 Select **File > New Project**. The Polyspace Project – Properties dialog box opens.

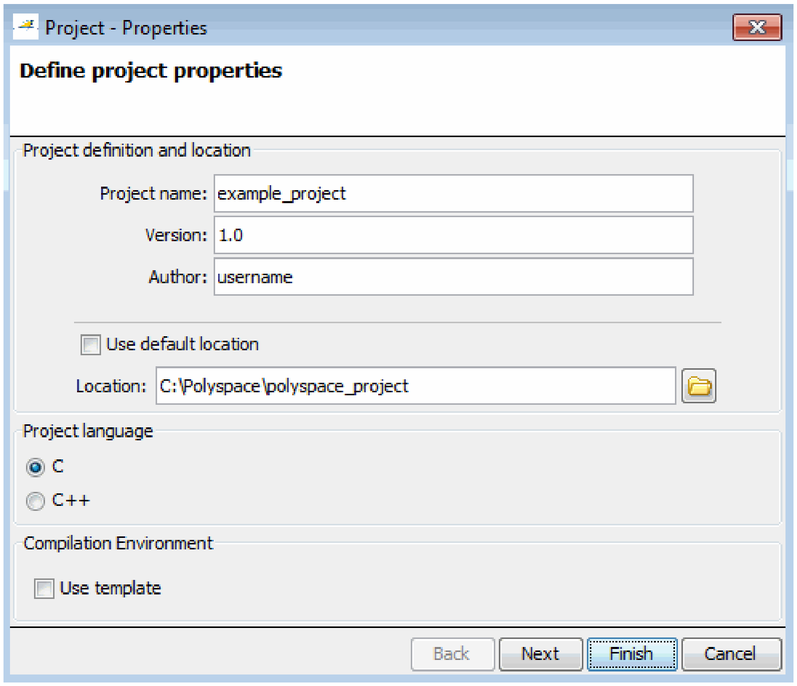
2 In the **Project name** field, enter `example_project`.

3 Clear the **Use default location** check box. In this tutorial, you change the location to the project folder that you created in "Prepare Project Folders" on page 2-3.

Note You can update the default project location. Select **Options > Preferences**, which opens the Polyspace Preferences dialog box. On the **Project and Results Folder** tab, in the **Default project location** field, specify the new default location.

4 In the **Location** field, enter or navigate to the project folder that you created earlier, that is, `C:\Polyspace\polyspace_project`.

5 In the Project language section, click **C**.



Project definition and location

Project name: example_project

Version: 1.0

Author: username

[ ] Use default location

Location: C:\Polyspace\polyspace_project

Project language

[✓ C] C++

Compilation Environment

[ ] Use template

6 Click **Finish**.

Specify Source Files and Include Folders

To specify the source files and include folders for the verification of `example.c` and `single_file_analysis.c`:

1 In the Project Browser, select the Source folder.

2 On the Project Browser toolbar, click the **Add source** icon ➕. The Project – Add Source Files and Include Folders dialog box opens.

3 The project folder `polyspace_project` must be visible in the **Look in** field. Otherwise, navigate to this folder.

4 Select the `sources` folder. Then click **Add Source Files**.

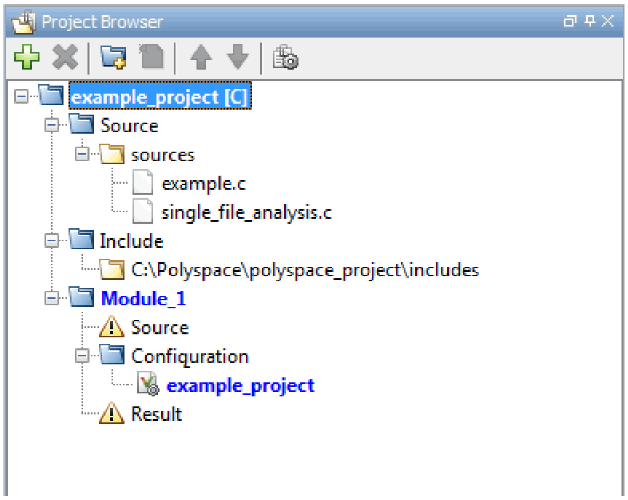
   The software adds `example.c` and `single_file_analysis.c` to the **Source** folder of `example_project`.

5 Select the `includes` folder. Then click **Add Include Folders**.

   The software adds `includes` to the **Include** folder of `example_project`.

   Note In addition to the include folders that you specify, Polyspace Code Prover automatically adds the standard includes to your project.

6 Click **Finish** to apply the changes and close the dialog box.



Specify Target Environment

Many applications are designed to run on specific target CPUs and operating systems. Since some run-time errors are dependent on the target, before running you must specify the type of CPU and operating system used in the target environment.

In the Project Manager perspective, the **Configuration > Target & Compiler** pane allows you to specify the target operating system and processor type for your application.

To specify the target environment for this tutorial:

1. Select the **Configuration > Target & Compiler** pane.

2. From the **Target operating system** drop-down list, selectno_predefined_OS.

3. From the **Target processor type** drop-down list, select i386.

For more information about emulating your target environment, see "Set Up a Target".

Specify Analysis Options

In the Project Manager perspective, the **Configuration** pane allows you to specify analysis options that the software uses during verification. For this tutorial, use the default values for all options.

For more information, see "Analysis Options for C Code" or "Analysis Options for C++ Code".

Save the Project

To save the project, select **File > Save** or enter **Ctrl+S**.

The software saves your project using the **Project name** and **Location** that you specified when creating the project.

Server Configuration for Remote Verification and Polyspace Metrics

Set Up Remote Verification and Polyspace Metrics

In this section...

With Polyspace Code Prover, you can run the following types of verifications.

Verification type	Run when
Remote	Source files are large (more than 800 lines of code including comments), and execution time of verification is long.
Local	Source files are small, and execution time of verification is short.

You require Polyspace Metrics to:

- Manage remote verifications, for example, to monitor progress, stop a verification, or view results.
- Evaluate and monitor software quality metrics.

To set up a configuration for remote verification and Polyspace Metrics, see:

- "Requirements for Remote Verification and Polyspace Metrics" on page 3-3
- "Configure Server for Remote Verification and Polyspace Metrics" on page 3-4
- "Configure Web Server for HTTPS" on page 3-8
- "Change Web Server Port Number for Polyspace Metrics Server" on page 3-9

Requirements for Remote Verification and Polyspace Metrics

The following table lists the requirements for remote verification and Polyspace Metrics.

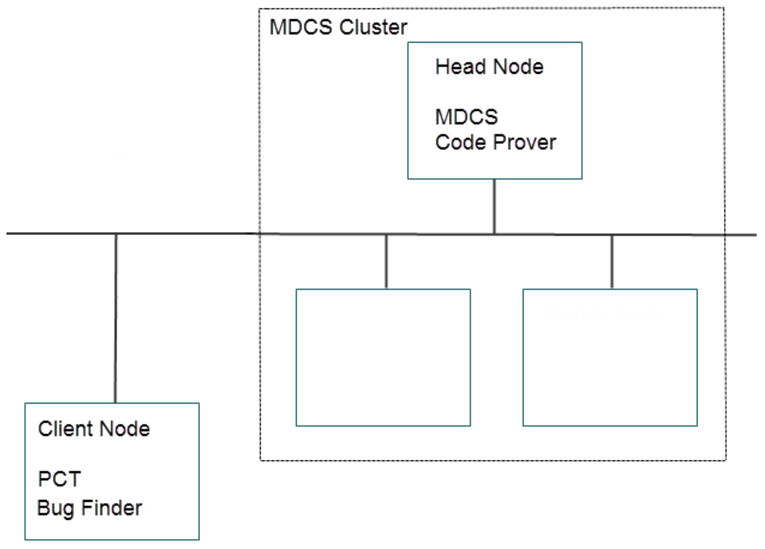
Task	Location	Requirements
Project configuration and verification submission	Client node	• MATLAB • Polyspace Bug Finder™ or Polyspace Code Prover • Parallel Computing Toolbox™
Remote verification	Head node of MDCS cluster	• MATLAB • Polyspace Code Prover • MATLAB Distributed Computing Server™
Polyspace Metrics service	Head node of MDCS cluster or any network server	• MATLAB • Polyspace Bug Finder or Polyspace Code Prover
Downloading of *complete* verification results from Polyspace Metrics	Client node or any network computer	• MATLAB • Polyspace Bug Finder or Polyspace Code Prover • Access to Polyspace Metrics server
Viewing of verification results *summary* from Polyspace Metrics	Any network computer	Access to Polyspace Metrics server.

For configuration details, see "Configure Server for Remote Verification and Polyspace Metrics" on page 3-4.

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

Configure Server for Remote Verification and Polyspace Metrics

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



To set up remote verification and Polyspace Metrics, configure the head node through the Metrics and Remote Server Settings dialog box and the client node through the **Server Configuration** tab:

Metrics and Remote Server Settings

1 Select **Options > 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 Under **Polyspace MDCS Cluster Security Settings**, you see the following options with default values:

   - **Start the Polyspace MDCE service** — Selected. The `mdce` service, which is required to manage the MJS, will run on the MJS host computer.
   - **MDCE service port** — 27350.
   - **Use secure communication** – Not selected. Communication is not encrypted. For encrypted communication, select this check box.

   For information about MATLAB Distributed Computing Server cluster security, see “Cluster Security”.

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

Use the Metrics and Remote Server Settings dialog box to start and stop `mdce` services only if you configure the MDCS head node as the Polyspace Metrics server. Otherwise, clear the **Start the Polyspace MDCE service** check box, and use the MDCS Admin Center. To open the MDCS Admin Center, run:

```
MATLAB_Install/toolbox/distcomp/bin/admincenter
```

For information about the MDCS Admin Center, see “Cluster Processes and Profiles”.

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`

Server Configuration

1. Select **Options > 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**:
   - 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 Polyspace Code Prover, the software downloads results to the verification launch folder. However, 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 Polyspace Code Prover, the folder and its contents are not deleted.

- In the **Port number** field, specify the port number for communication between Polyspace Code Prover 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 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 3-8.

  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 Polyspace Metrics Server" on page 3-9 .

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

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

- *`protocol`* is either `http` or `https.`
- *`ServerName`* is the name or IP address of your Polyspace Metrics server.
- *`WSPN`* is the Web server port number.

Note To access Polyspace Metrics when the Polyspace Metrics server and MJS are not hosted by the same computer, you must add the following line to the `polyspace.conf` file :

```
job_scheduler=Release:HeadNodeHostName:JobSchedulerName
```

For information about required products, see "Requirements for Remote Verification and Polyspace Metrics" on page 3-3.

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 "Configure Server for Remote Verification and Polyspace Metrics" 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:

   ```
   Polyspace_Install\polyspace\bin\polyspace-rl-manager.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 *`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 necessary text 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 the following URL:

```
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 Polyspace 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.

In *`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"/>
  ```

Run a Verification

Run Verification

In this section...

Tutorial Overview

After creating the project `example_project.psprj`, which is described in "Set Up Polyspace Project" on page 2-2, you can run a local or remote verification.

Verification type	Use when
Remote	Source files are large (more than 800 lines of code including comments), and execution time of verification is long. You want to generate Polyspace Metrics. Through Polyspace Metrics, you can manage verifications and monitor quality over a project lifecycle.
Local	Source files are small, and execution time of verification is short.

In this tutorial, you learn how to start local and remote verifications using the Project Manager. You also verify the file `example.c` and `single_file_analysis.c`.

The local and remote verifications generate the same results for your project. In the tutorial "Review Verification Results" on page 5-2, you review these results.

Before You Start the Tutorial

Before you start this tutorial, you must:

- "Set Up Remote Verification and Polyspace Metrics" on page 3-2
- Complete "Set Up Polyspace Project" on page 2-2 as the folders and project file (`example_project.psprj`) from that tutorial are required.

Prepare for Verification

Open the Project

To run a verification, you must have an open project file. For this tutorial, use the project file `example_project.psprj` that you created in "Set Up Polyspace Project" on page 2-2. If `example_project.psprj` is not already open, then:

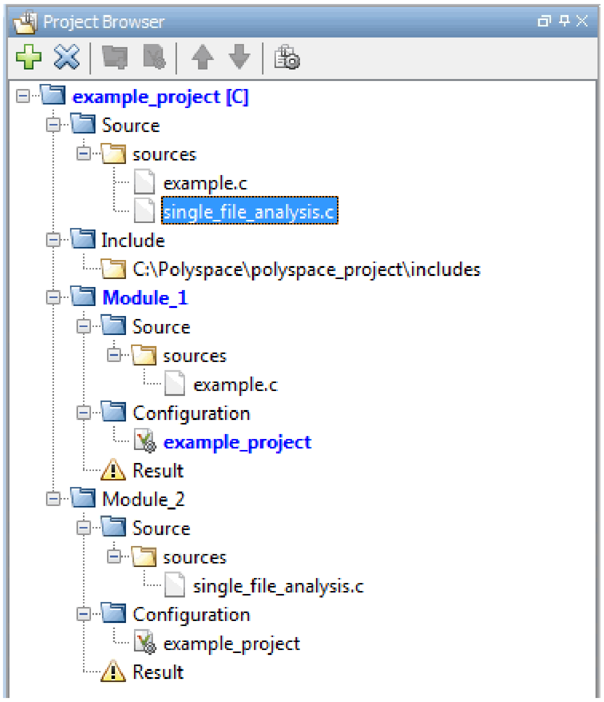
1. Select **File > Open Project**.
2. In the Open Project dialog box, from the **Look in** drop-down list, navigate to `polyspace_project`.
3. Select `example_project.psprj`.
4. Click **Open**.

Specify Source Files to Verify

Each Polyspace Code Prover project can contain multiple modules. With each module, you verify a specific set of source files using a specific set of analysis options.

Before you start a verification, you must specify the files that you want to verify by copying them into a module. In this tutorial, **example_project[C]** has two files that require verification. To verify the files separately, copy the files into individual modules:

1. In the Project Browser, right-click `example.c`.
2. From the context menu, select **Copy Source File to > Module_1**.
3. In the Project Browser, right-click **example_project[C]**.
4. Select **Create New Module**.
5. Right-click `single_file_analysis.c`.
6. Select **Copy Source File to > Module_2**.



Check for Compilation Problems

During a verification, if the Compilation Assistant detects compilation errors, the verification stops. The software displays errors and possible solutions on the **Output Summary** tab.

Note The Compilation Assistant does not support the verification option `-unit-by-unit`. For more information, see, "Check for Compilation Problems".

To check your project for compilation problems:

1 Select **Options > Preferences**.

2 In the Polyspace Preferences dialog box, click the **Project and Results Folder** tab.

3 Select the **Use Compilation Assistant** check box. Then click **OK**.

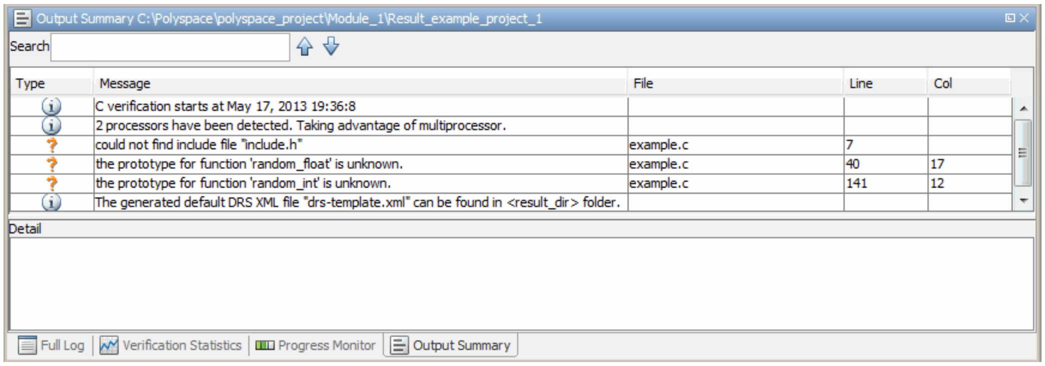
4 From the Project Manager perspective, in the **Configuration > Distributed Computing** pane, clear the **Batch** check box.

5 In the Project Browser, right-click the **Include** folder (`C:\Polyspace\polyspace_project\includes`), and then select **Remove**. The missing include files will cause compilation problems.

6 Select **Module_1**.

7 On the Project Manager toolbar, click Run.

The software compiles your code and checks for errors. It reports the results on the **Output Summary** tab.



Output Summary C:\Polyspace\polyspace_project\Module_1\Result_example_project_1

Search

Type	Message	File	Line	Col
ⓘ	C verification starts at May 17, 2013 19:36:8			
ⓘ	2 processors have been detected. Taking advantage of multiprocessor.			
?	could not find include file "include.h"	example.c	7	
?	the prototype for function 'random_float' is unknown.	example.c	40	17
?	the prototype for function 'random_int' is unknown.	example.c	141	12
ⓘ	The generated default DRS XML file "drs-template.xml" can be found in <result_dir> folder.			

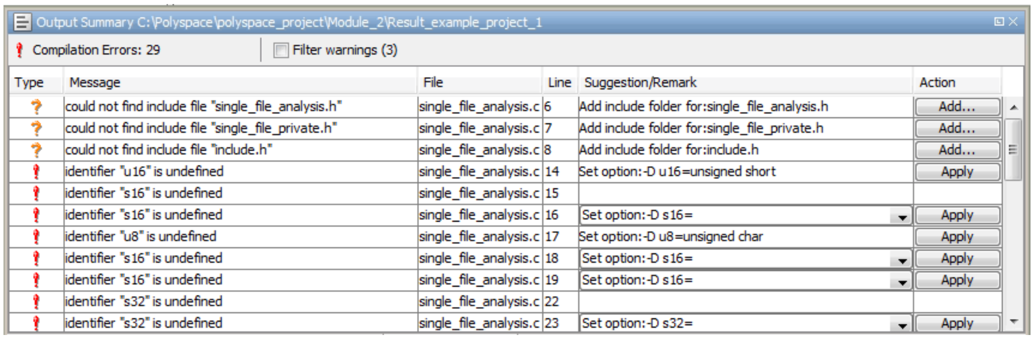
Detail

In this case, the software generates only warnings (in orange), not errors, for the missing include files. Because there are no errors, the verification is still completed. It is a good practice to resolve warnings for missing include files first, because they might be the root cause of other warnings or errors.

8 In the Project Browser, select **Module_2**.

9 Click the Run button again. The verification software runs with `single_file_analysis.c` as the source file.

Again, the software compiles your code and checks for errors. It reports the results on the **Output Summary** tab.



Output Summary C:\Polyspace\polyspace_project\Module_2\Result_example_project_1

Compilation Errors: 29 | Filter warnings (3)

Type	Message	File	Line	Suggestion/Remark	Action
?	could not find include file "single_file_analysis.h"	single_file_analysis.c	6	Add include folder for:single_file_analysis.h	Add...
?	could not find include file "single_file_private.h"	single_file_analysis.c	7	Add include folder for:single_file_private.h	Add...
?	could not find include file "include.h"	single_file_analysis.c	8	Add include folder for:include.h	Add...
!	identifier "u16" is undefined	single_file_analysis.c	14	Set option:-D u16=unsigned short	Apply
!	identifier "s16" is undefined	single_file_analysis.c	15		
!	identifier "s16" is undefined	single_file_analysis.c	16	Set option:-D s16=	Apply
!	identifier "u8" is undefined	single_file_analysis.c	17	Set option:-D u8=unsigned char	Apply
!	identifier "s16" is undefined	single_file_analysis.c	18	Set option:-D s16=	Apply
!	identifier "s16" is undefined	single_file_analysis.c	19	Set option:-D s16=	Apply
!	identifier "s32" is undefined	single_file_analysis.c	22		
!	identifier "s32" is undefined	single_file_analysis.c	23	Set option:-D s32=	Apply

In this case, the software generates:

- Warnings for the missing include files.
- Errors for undefined identifiers, which stop the verification. The **Suggestion/Remark** column indicates that definitions for data types are required. This information is present in the include files that were removed.

10 In the **Project Browser** tree, right-click the **Include** folder. From the context menu, select **Add Source**.

The Project - Add Source Files and Include Folders dialog box opens.

11 If you are not in the `polyspace_project` folder, navigate to this folder.

12 Select the `includes` folder. Then click **Add Include Folders**.

The software adds the `includes` folder to the **Include** folder for `example_project`.

13 Click **Finish**.

14 In the Project Browser, select `Module_1`. On the Project Manager toolbar, click ▷ Run.

   The verification should start and run to completion without any include file warnings.

15 In the Project Browser, select `Module_2`. On the Project Manager toolbar, click ▷ Run.

   The verification should start and run to completion without any warnings or errors.

Start Remote Verification from Project Manager

1 In the Project Manager perspective, from the **Project Browser** pane, select the module you want to verify.

2 Select the **Configuration > Distributed Computing** pane.

3 Select the **Batch** check box. By default, the software also selects the **Add results to repository** check box, which enables the generation of Polyspace Metrics.

4 On the Project Manager toolbar, click ▷ Run.

On the local host computer, the Polyspace Code Prover software performs code compilation . Then the Parallel Computing Toolbox software submits the verification to the MATLAB job scheduler (MJS) 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** and go to “Verification Process Failed Errors”.

To monitor progress, use Polyspace Metrics. See “Monitor Verification Progress” on page 4-10.

Start Local Verification from Project Manager

To start a verification on your local computer:

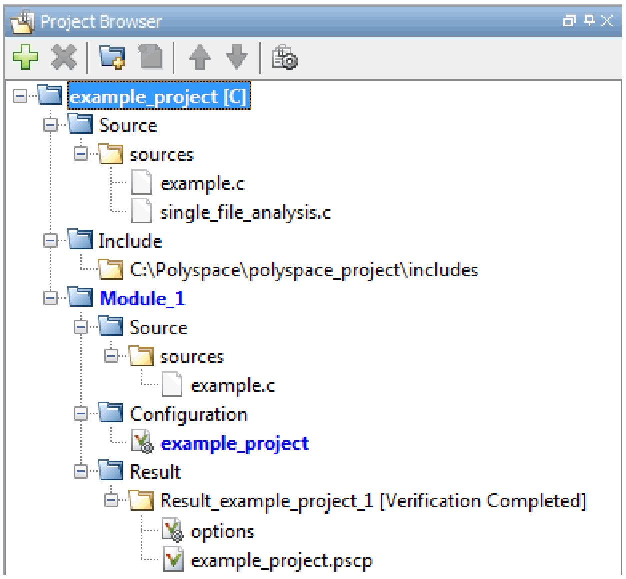
1. In the Project Manager perspective, from the **Project Browser** view, select the module you want to verify.
2. Select the **Configuration > Distributed Computing** pane.
3. By default, the **Batch** check box is not selected. However, if this check box is selected, you must clear the check box.
4. On the Project Manager toolbar, click Run.

You can monitor the progress of the verification through the **Progress Monitor**, **Full Log**, and **Output Summary** tabs. See “Monitor Verification Progress” on page 4-10.

If the verification fails, go to “Verification Process Failed Errors”.

When the verification is complete, you see:

- In the **Full Log**, the message `End of Polyspace verification`.
- In the **Project Browser**, the results file, for example, `example_project.pscp`.



In the tutorial "Review Verification Results" on page 5-2, you open the Results Manager perspective and review the verification results.

Monitor Verification Progress

To monitor the progress of a *remote* verification, open the verification log:

1. Open the **Runs** view of Polyspace Metrics.

2. Right-click the verification.

3. From the context menu, select **View Log**.

For more information, see "Manage Previous Verifications With Polyspace Metrics".

To monitor the progress of a *local* or *remote interactive* verification, from the Project Manager perspective, use the following tabs :

- **Progress Monitor** — A blue progress bar indicates the current phase of the verification. The tab also displays the time and percentage completed for each phase.
- **Full Log** — This tab displays messages, errors, and statistics for all phases of the verification. To search for a term, in the **Search** field, enter the required term. Click the up arrow or down arrow to move sequentially through occurrences of this term.
- **Output Summary** — Displays compile phase messages and errors. To search for a term, in the **Search** field, enter the required term. Click the up or down arrow to move sequentially through occurrences of the term.

At the end of a local or remote interactive verification, the **Verification Statistics** tab displays statistics, for example, code coverage and check distribution.

Stop Verification

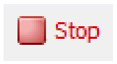
To stop a remote verification:

1. Open the **Runs** view of Polyspace Metrics.
2. Right-click the verification.
3. From the context menu, select **Stop and Remove Job**.

For more information, see "Manage Previous Verifications With Polyspace Metrics".

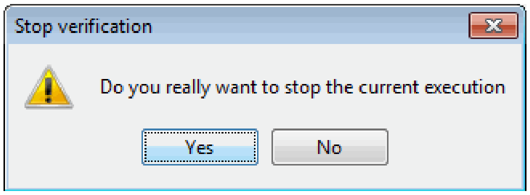
To stop a local verification:

1. On the Project Manager toolbar, click

   .

   A warning dialog box opens.


Stop verification

Do you really want to stop the current execution

2 Click **Yes**. The verification stops, and the results are incomplete. If you start another verification, the verification starts from the beginning.

5

Review Verification Results

Review Verification Results

In this section...

Tutorial Overview

In the previous tutorial, "Run Verification" on page 4-2, you completed a verification of `example.c`. In this tutorial, you explore the verification results.

Polyspace Code Prover provides a Results Manager perspective, which you use to review results. In this tutorial, you learn:

1. How to use the Results Manager perspective, including how to:
   - Open the Results Manager perspective and view verification results.
   - Review results.
   - Generate reports.
2. How to interpret the color coding that the software uses to indicate the severity of an error.
3. How to find the location of an error in the source code.

Before You Start

Before starting this tutorial, complete the tutorial "Run Verification" on page 4-2.

Open Remote Verification Results

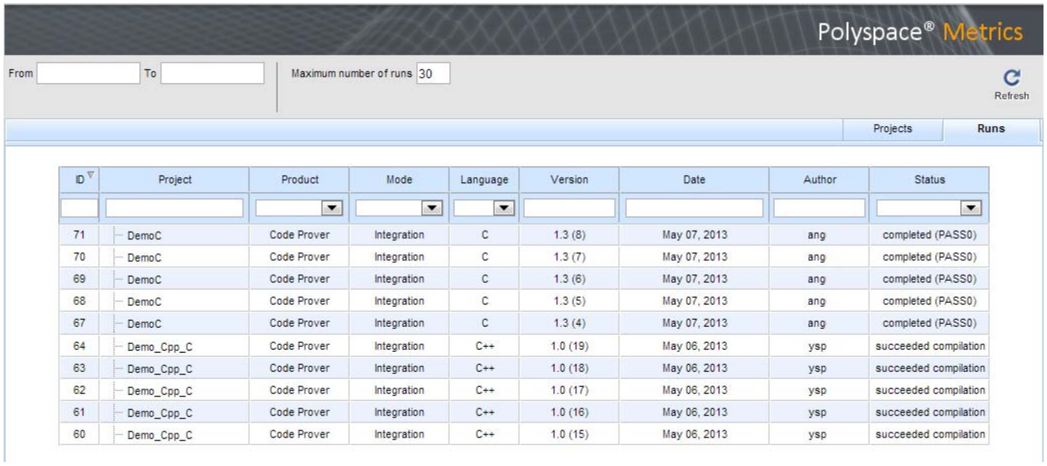
Use Polyspace Metrics to open results from a remote verification.

1 In the address bar of your Web browser, enter the following URL:

   `protocol://ServerName:PortNumber`

   - `protocol` is either `http` (default) or `https`.
   - `ServerName` is the name or IP address of your Polyspace Metrics server.
   - `PortNumber` is the Web server port number (default 8080).

   For reference, save the Polyspace Metrics Web page as a bookmark.



ID	Project	Product	Mode	Language	Version	Date	Author	Status
71	DemoC	Code Prover	Integration	C	1.3 (8)	May 07, 2013	ang	completed (PASS0)
70	DemoC	Code Prover	Integration	C	1.3 (7)	May 07, 2013	ang	completed (PASS0)
69	DemoC	Code Prover	Integration	C	1.3 (6)	May 07, 2013	ang	completed (PASS0)
68	DemoC	Code Prover	Integration	C	1.3 (5)	May 07, 2013	ang	completed (PASS0)
67	DemoC	Code Prover	Integration	C	1.3 (4)	May 07, 2013	ang	completed (PASS0)
64	Demo_Cpp_C	Code Prover	Integration	C++	1.0 (19)	May 06, 2013	ysp	succeeded compilation
63	Demo_Cpp_C	Code Prover	Integration	C++	1.0 (18)	May 06, 2013	ysp	succeeded compilation
62	Demo_Cpp_C	Code Prover	Integration	C++	1.0 (17)	May 06, 2013	ysp	succeeded compilation
61	Demo_Cpp_C	Code Prover	Integration	C++	1.0 (16)	May 06, 2013	ysp	succeeded compilation
60	Demo_Cpp_C	Code Prover	Integration	C++	1.0 (15)	May 06, 2013	ysp	succeeded compilation

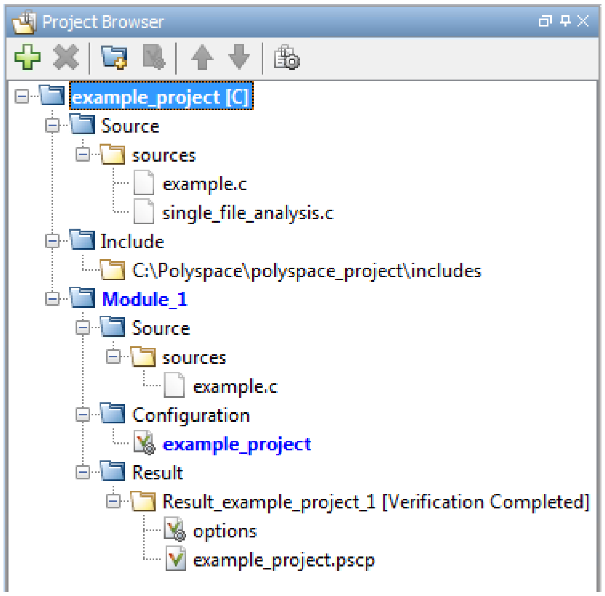
2 Click the **Project** or **Version** cell of your verification.

   The software downloads and opens the results in the Results Manager perspective of Polyspace Code Prover. See "Explore Results Manager perspective" on page 5-5.

Note If the verification is unit-by-unit, then clicking the **Project** or **Version** cell opens the Select the results set to review dialog box. From the **Results Set** drop-down list, select the results set that you want to review. Alternatively, select the **Download all results sets** check box. Then click **OK**.

Open Local Verification Results

If your project is open in the Project Manager perspective, from the Project Browser, double-click the results file `example_project.pscp`.



The software opens the results in the Results Manager perspective. See "Explore Results Manager perspective" on page 5-5.

Alternatively:

1 On the Polyspace Code Prover toolbar, select **File > Open Result**.

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

```
polyspace_project\Module_1\Result_example_project_1
```

3 Select the file `example_project.pscp`.

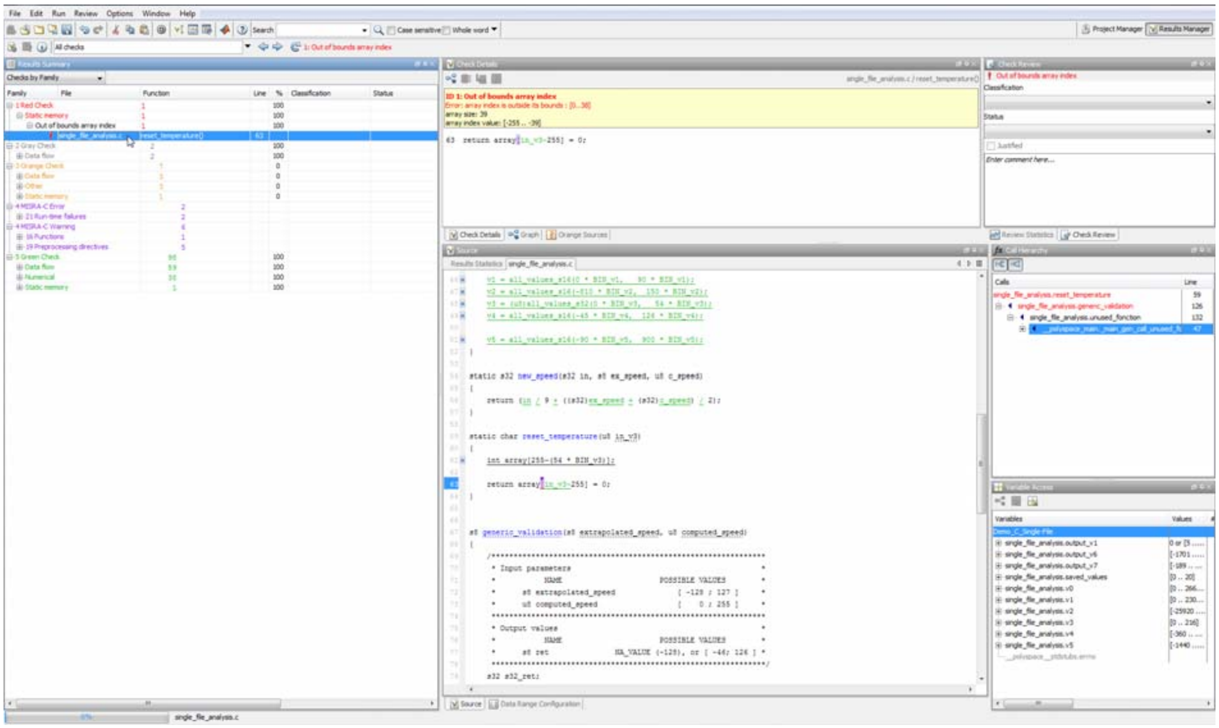
4 Click **Open**.

Explore Results Manager perspective

- "Overview" on page 5-5
- "Results Summary" on page 5-6

Overview

The Results Manager perspective looks like the following figure.



Family	File	Function	Line	%	Classification	Status
1 Red Check		1		100		
Static memory		1		100		
Out of bounds array index		1		100		
	single_file_analysis.c	reset_temperature()	63			
2 Gray Check		2		100		
Data flow		2		100		
3 Orange Check		7		0		
Data flow		3		0		
Other		3		0		
Static memory		1		0		
4 MISRA-C Error		2				
21 Run-time failures		2				
4 MISRA-C Warning		6				
16 Functions		1				
19 Preprocessing directives		5				
5 Green Check		95		100		
Data flow		59		100		
Numerical		35		100		
Static memory		1		100		

single_file_analysis.c / reset_temperature()

ID 1: Out of bounds array index
Error: array index is outside its bounds : [0..38]
array size: 39
array index value: [-255 .. -39]

```
63  return array[in_v3-255] = 0;
```

Check Review: Out of bounds array index
Classification
Status
Justified
Enter comment here...

```
    v1 = all_values_s16(0 * BIN_v1,   90 * BIN_v1);
    v2 = all_values_s16(-810 * BIN_v2,  150 * BIN_v2);
    v3 = (u8)all_values_s32(0 * BIN_v3,   54 * BIN_v3);
    v4 = all_values_s16(-45 * BIN_v4,  126 * BIN_v4);

    v5 = all_values_s16(-90 * BIN_v5,  900 * BIN_v5);
}

static s32 new_speed(s32 in, s8 ex_speed, u8 c_speed)
{
    return (in / 9 + ((s32)ex_speed + (s32)c_speed) / 2);
}

static char reset_temperature(u8 in_v3)
{
    int array[255-(54 * BIN_v3)];

    return array[in_v3-255] = 0;
}


s8 generic_validation(s8 extrapolated_speed, u8 computed_speed)
{
    /**************************************************************
     * Input parameters                                           *
     *          NAME                          POSSIBLE VALUES     *
     *      s8 extrapolated_speed                [ -128 ; 127 ]   *
     *      u8 computed_speed                    [    0 ; 255 ]   *
     **************************************************************
     * Output values                                              *
     *          NAME                          POSSIBLE VALUES     *
     *      s8 ret              NA_VALUE (-128), or [ -46; 126 ]  *
     **************************************************************/
    s32 s32_ret;
```

Calls	Line
single_file_analysis.reset_temperature	59
single_file_analysis.generic_validation	126
single_file_analysis.unused_fonction	132
__polyspace_main._main_gen_call_unused_fc	47

Variables	Values
Demo_C_Single-File	
single_file_analysis.output_v1	0 or [3 .....
single_file_analysis.output_v6	[-1701 .....
single_file_analysis.output_v7	[-189 ... ...
single_file_analysis.saved_values	[0 .. 20]
single_file_analysis.v0	[0 .. 266...
single_file_analysis.v1	[0 .. 230...
single_file_analysis.v2	[-25920 ....
single_file_analysis.v3	[0 .. 216]
single_file_analysis.v4	[-360 ... ...
single_file_analysis.v5	[-1440 .....
__polyspace__stdstubs.errno	

The Results Manager perspective has six sections below the toolbar. Each section provides a different view of the results. The following table describes these views.

This pane or view ...	Displays ...
Results Summary	List of checks (diagnostics) for each file and function in the project
Source	Source code for a selected check in the **Results Summary** view
Check Details	Details about the selected check
Check Review	Review information about selected check
Variable Access	Information about global variables declared in the source code
Call Hierarchy	Tree structure of function calls

You can resize or hide any of these sections. You learn more about the Results Manager perspective later in this tutorial.

Results Summary

The **Results Summary** pane lists all checks along with their attributes. To organize your check review, from the drop-down list on this pane, select one of the following options:

- `List of Checks`: Lists all checks without any grouping. The checks are sorted in the following order:
  1. **Red**: Indicates code that is proven to contain an error. The check indicates that the code will fail every time it is executed.
  2. **Gray** — Indicates unreachable code.
  3. **Orange** — Indicates unproven code that might contain an error.
  4. **Purple.** — Indicates coding rule violation.
  5. **Green** — Indicates code that is proven to not contain an error.
- `Checks by Family`: Lists all checks grouped by color. Within each color, the checks are grouped by category. For more information on the checks covered by a category, see the check reference pages.

- `Checks by Class`: Lists all checks grouped by class. Within each class, the checks are grouped by method. The first group, **Global Scope**, lists all checks not occurring in a class definition.

  This option is available for `C++` code only.

- `Checks by File/Function`: Lists all checks grouped by file. Within each file, the checks are grouped by function.

For each check, the **Results Summary** pane contains the check attributes, listed in columns:

Attribute	Description
Family	Group to which the check belongs. For instance, if you choose the grouping `Checks by File/Function`, this column contains the name of the file and function containing the check.
Check	Description of the error
Information	For run-time errors, this attribute indicates whether the check is related to path or bounded input values. For coding rule violations, this attribute indicates whether the rule is `Required`.
File	File containing the instruction where the check occurs
Function	Function containing the instruction where the check occurs. If the function is a method of a class, it appears in the format *`class_name`*`::`*`function_name`*.
Line	Line number of the instruction where the check occurs.

Attribute	Description
Classification	Level of severity you have assigned to the check. The possible levels are: • `Unset` • `High` • `Medium` • `Low` • `Not a defect`
Status	Review status you have assigned to the check. The possible statuses are: • `Fix` • `Improve` • `Investigate` • `Justify with annotations` • `No action planned` • `Other` • `Restart with different options`
Justified	Check boxes showing whether you have justified the checks
Comments	Comments you have entered about the check

To show or hide any of the columns, right-click anywhere on the column titles. From the context menu, select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through all the checks. For more information, see “Review and Comment Checks”.

- Organize your check review using filters on the appropriate columns. For more information, see “Organize Check Review Using Filters and Groups”.

Review Results

- “View Examples of Checks” on page 5-9
- “Review Checks” on page 5-13
- “Review Checks Using Predefined Methodologies” on page 5-17
- “Organize Check Review Using Filters and Groups” on page 5-18

View Examples of Checks

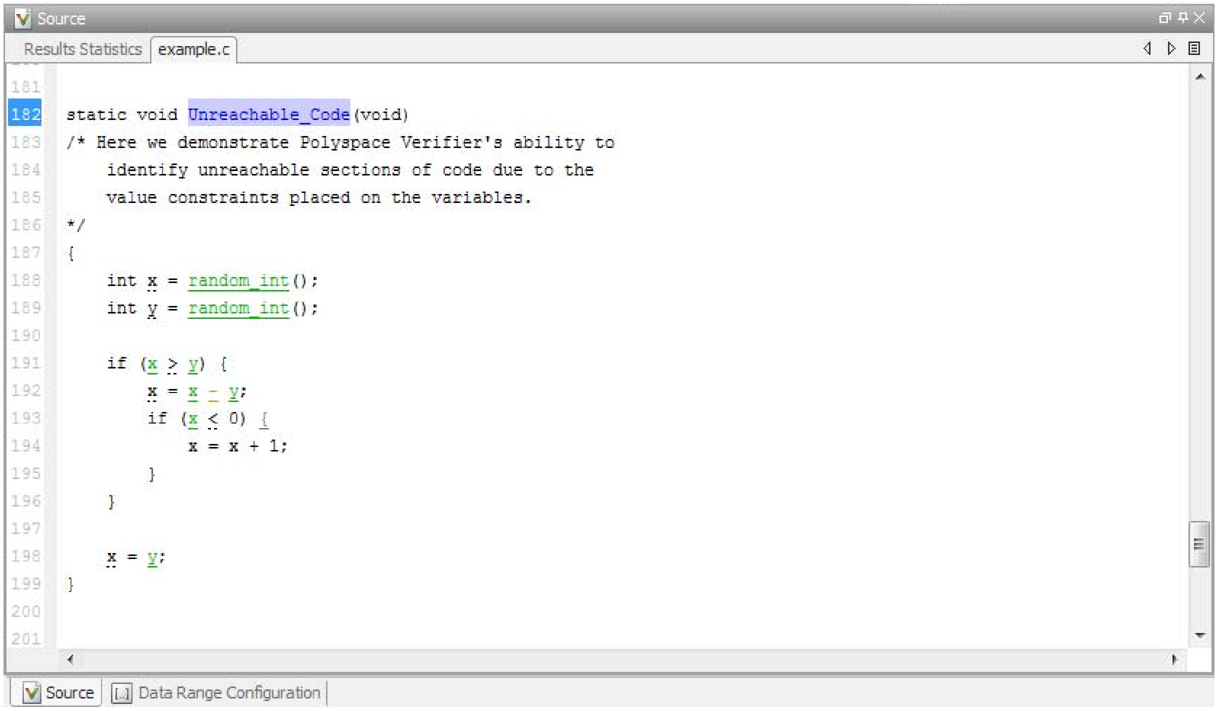
In this part of the tutorial, you learn about categories of errors by reviewing the following examples in `example.c`:

- “Example: Unreachable Code” on page 5-10
- “Example: Arithmetic Error” on page 5-11
- “Example: A Function with No Errors” on page 5-12
- “Example: Division by Zero” on page 5-12

Example: Unreachable Code. Unreachable code is code that never executes. The code verification software displays unreachable code in gray. In the following example, you look at an example of unreachable code.

1 On the **Results Summary** pane, click `Unreachable_Code()`.

   The source code view displays the source code for this function.



```
181
182 static void Unreachable_Code(void)
183 /* Here we demonstrate Polyspace Verifier's ability to
184     identify unreachable sections of code due to the
185     value constraints placed on the variables.
186 */
187 {
188     int x = random_int();
189     int y = random_int();
190
191     if (x > y) {
192         x = x - y;
193         if (x < 0) {
194             x = x + 1;
195         }
196     }
197
198     x = y;
199 }
200
201
```

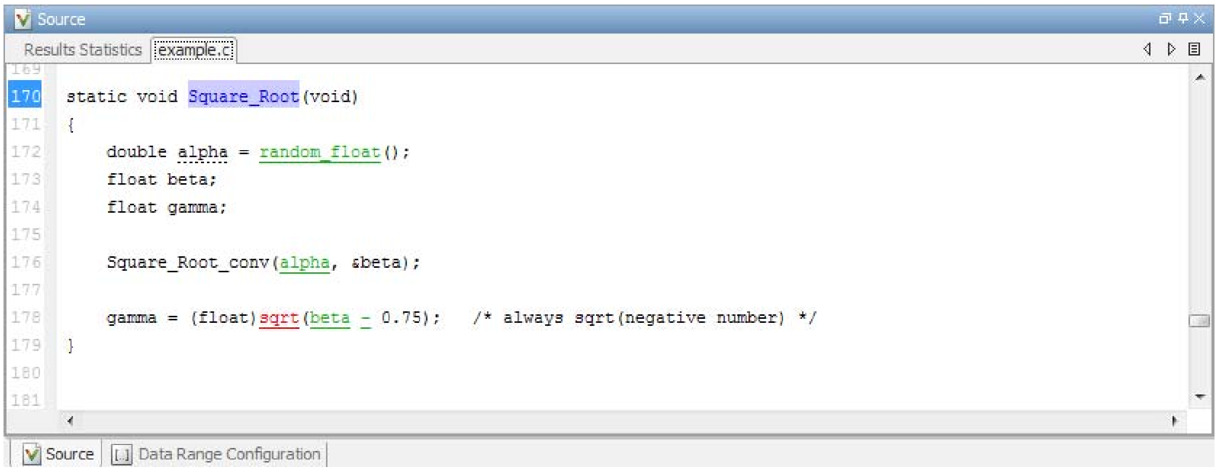
2 Examine the source code.

   At line 193, the condition `x < 0` is always false. The curly bracket { is gray because the branch is never executed.

Example: Arithmetic Error. In the following example, the code verification software detects a memory corruption error:

1. On the **Results Summary** pane, select the red `Square_Root()` function.

   The source code view displays the source code for this function.



```
170  static void Square_Root(void)
171  {
172      double alpha = random_float();
173      float beta;
174      float gamma;
175
176      Square_Root_conv(alpha, &beta);
177
178      gamma = (float)sqrt(beta - 0.75);   /* always sqrt(negative number) */
179  }
180
181
```

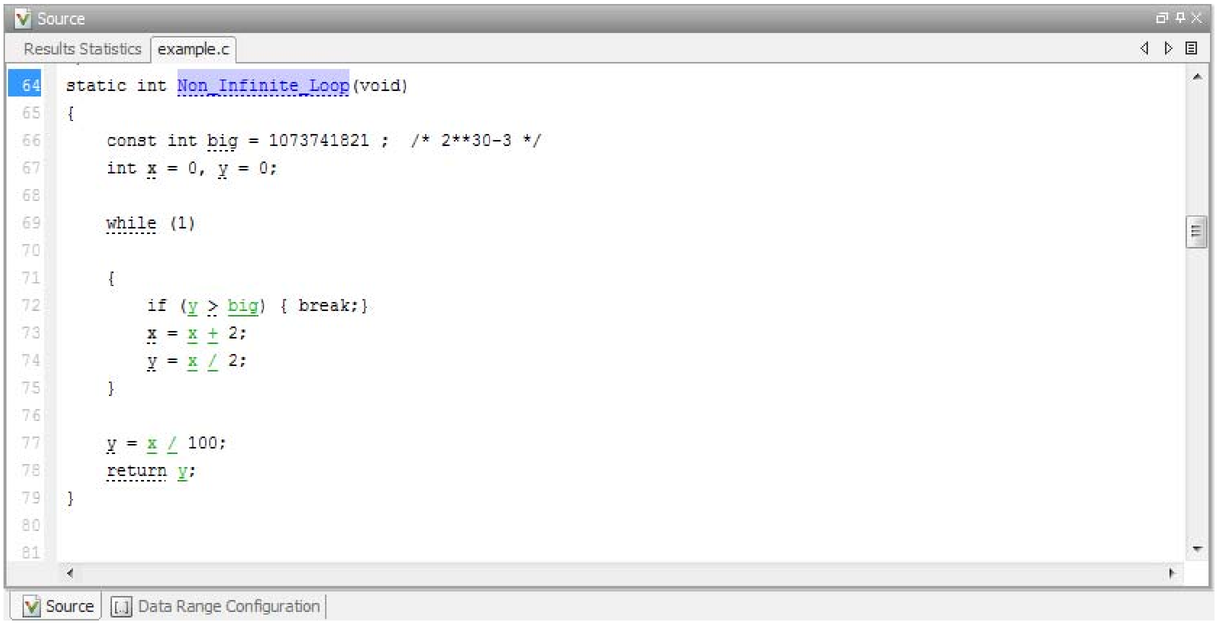
2. Examine the source code.

   Because `beta` is always less than 0.75, the argument to the `sqrt()` function at line 178 is always negative.

Example: A Function with No Errors. In the following example, the code verification software verifies code with a large number of iterations, and determines that the loop terminates and a variable does not overflow:

1 On the **Results Summary** pane, click the green `Non_Infinite_Loop()` function.

   The source code view displays the source code for this function.



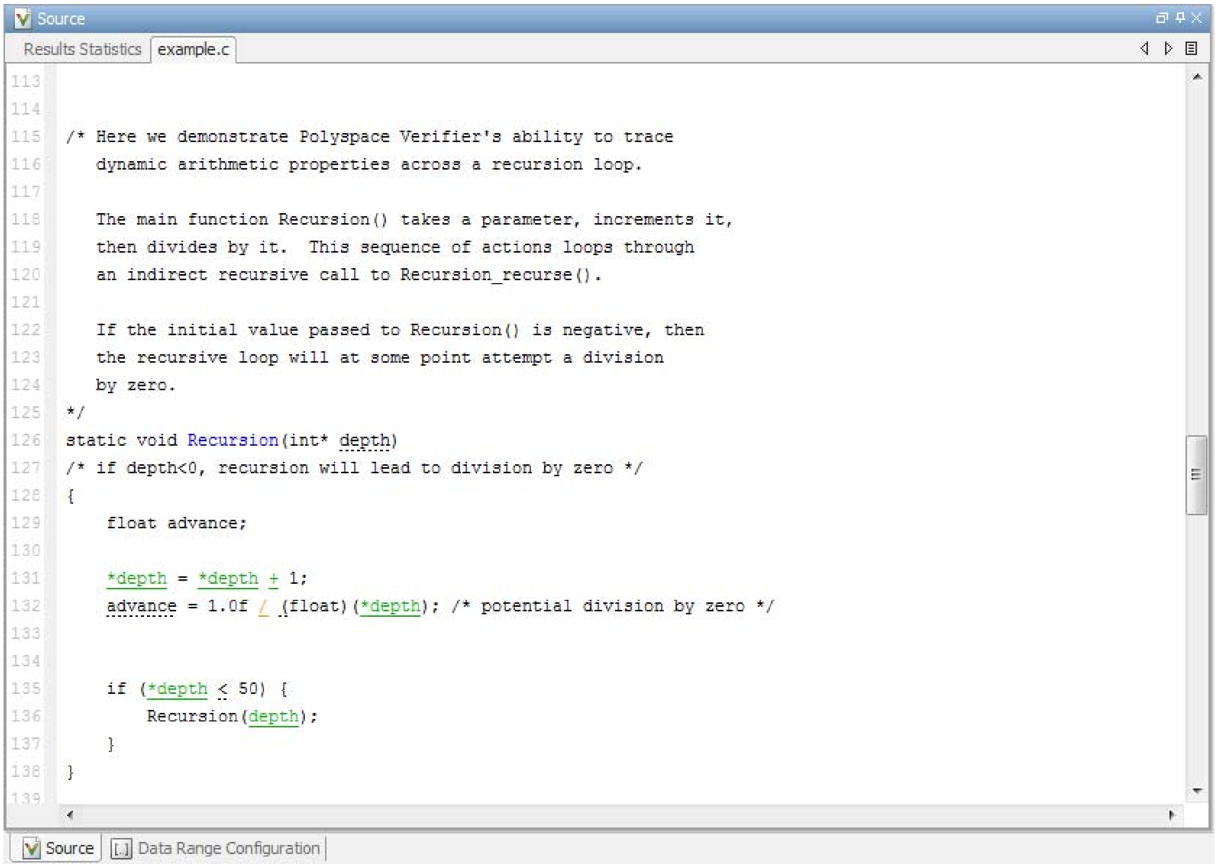
```
64  static int Non_Infinite_Loop(void)
65  {
66      const int big = 1073741821 ;  /* 2**30-3 */
67      int x = 0, y = 0;
68
69      while (1)
70
71      {
72          if (y > big) { break;}
73          x = x + 2;
74          y = x / 2;
75      }
76
77      y = x / 100;
78      return y;
79  }
80
81
```

2 Examine the source code. The variable `x` never overflows because the `while` loop at line `69` terminates before x can overflow.

Example: Division by Zero. In the following example, the code verification software detects division by zero:

1 On the **Results Summary** pane, select `Recursion()`.

   The source code view displays the source code for this function.



```
113
114
115  /* Here we demonstrate Polyspace Verifier's ability to trace
116     dynamic arithmetic properties across a recursion loop.
117
118     The main function Recursion() takes a parameter, increments it,
119     then divides by it.  This sequence of actions loops through
120     an indirect recursive call to Recursion_recurse().
121
122     If the initial value passed to Recursion() is negative, then
123     the recursive loop will at some point attempt a division
124     by zero.
125  */
126  static void Recursion(int* depth)
127  /* if depth<0, recursion will lead to division by zero */
128  {
129      float advance;
130
131      *depth = *depth + 1;
132      advance = 1.0f / (float)(*depth); /* potential division by zero */
133
134
135      if (*depth < 50) {
136          Recursion(depth);
137      }
138  }
139
```

2 Examine the `Recursion()` function.

When `Recursion()` is called with `depth` less than zero, the code at line `132` results in division by zero. The orange color indicates that this operation is a potential error (depending on the value of `depth`).

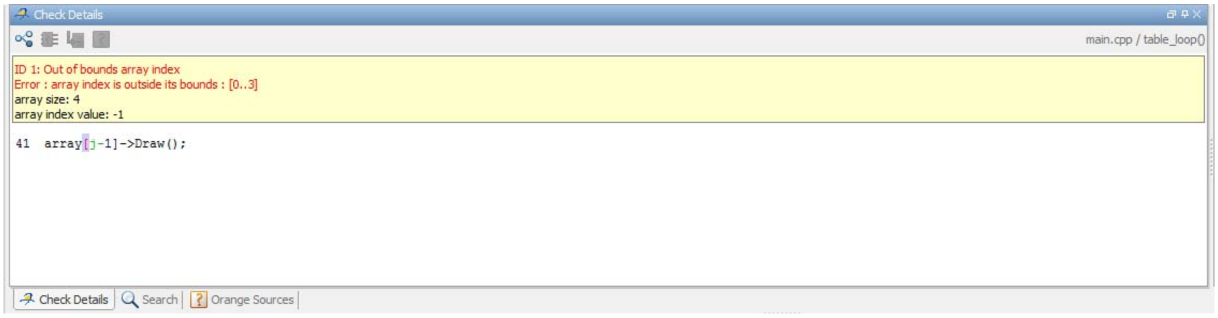
Review Checks

This example shows how to review and comment checks using the Results Manager perspective. When reviewing checks, you can assign a status to checks, and enter comments to describe the results of your review. These actions help you to track the progress of your review and avoid reviewing the same check twice.

Review and Comment Individual Check

1 On the **Results Summary** pane, select the check that you want to review.

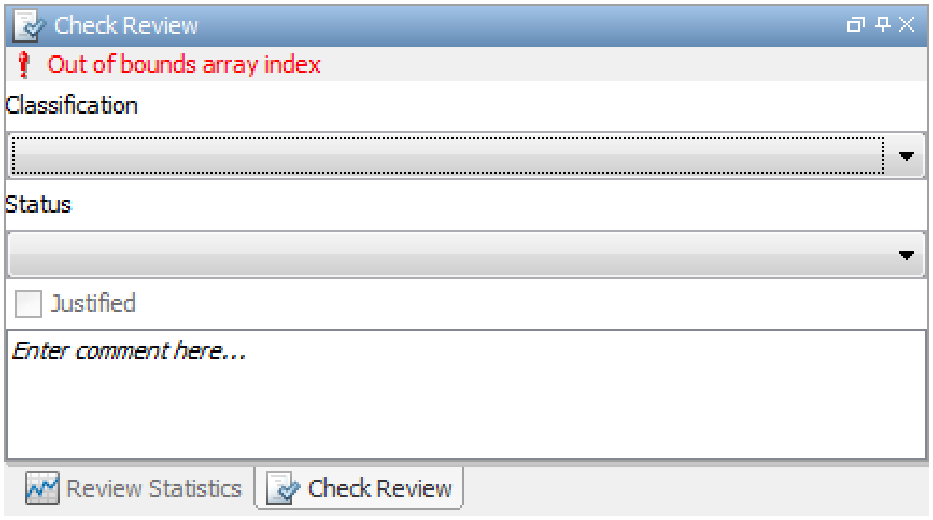
The **Check Details** pane displays information about the current check.



ID 1: Out of bounds array index
Error : array index is outside its bounds : [0..3]
array size: 4
array index value: -1

```
41  array[j-1]->Draw();
```

The **Check Review** tab displays fields where you can enter review information.



2 Select a **Classification** to describe the severity of the issue:

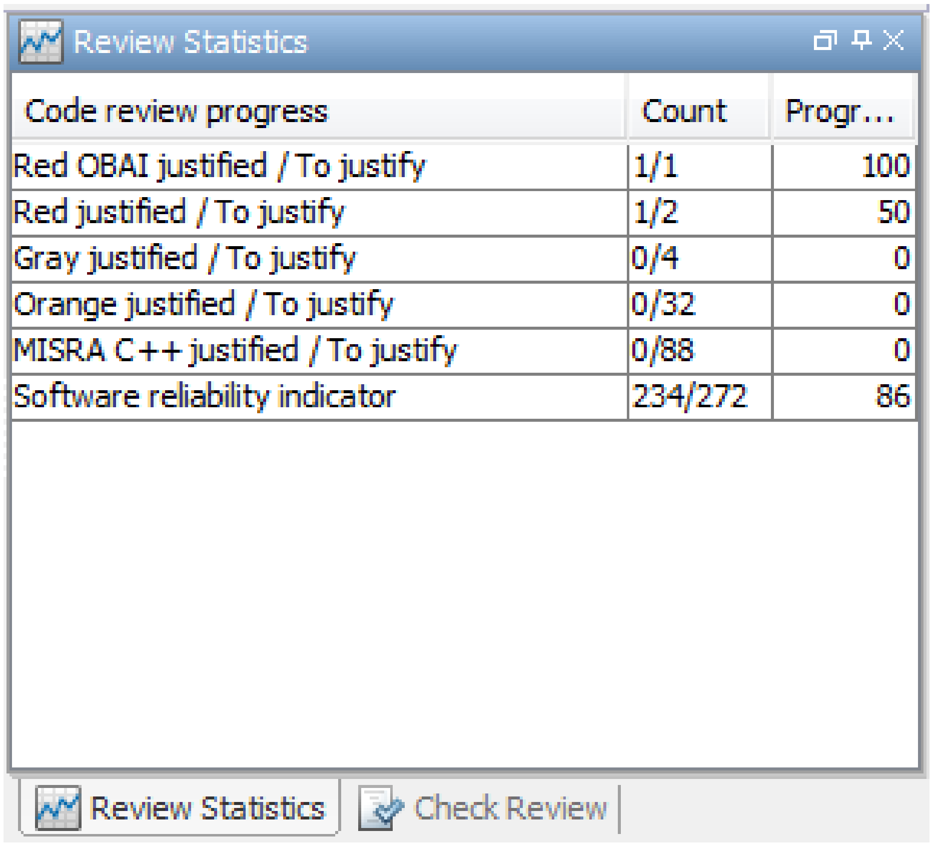
- `Unset`
- `High`
- `Medium`
- `Low`
- `Not a defect`

3 Select a **Status** to describe how you intend to address the issue:

- `Fix`
- `Improve`
- `Investigate`
- `Justify with annotations`
- `No action planned`
- `Other`
- `Restart with different options`
- `Undecided`

4 To justify the check, select one of the **Status** options, `Justify with annotations` or `No action planned`.

On the **Review Statistics** pane, the software updates the ratios of errors justified to total errors.



Code review progress	Count	Progr...
Red OBAI justified / To justify	1/1	100
Red justified / To justify	1/2	50
Gray justified / To justify	0/4	0
Orange justified / To justify	0/32	0
MISRA C++ justified / To justify	0/88	0
Software reliability indicator	234/272	86

5 In the **Comment** field, enter remarks, for example, defect or justification information.

Note You can also enter the review information through the **Classification**, **Status**, and **Comment** fields on the **Results Summary** pane.

Save Review Comments

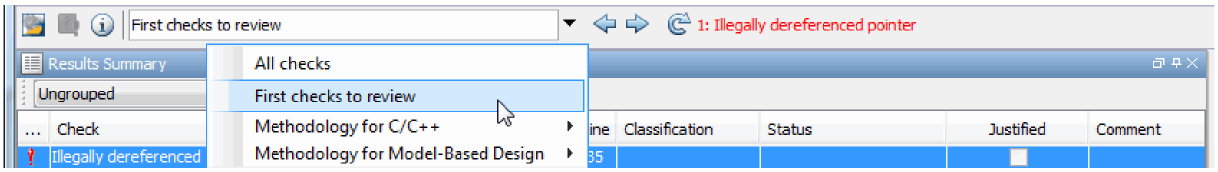
After you have reviewed your results, save your comments with the verification results. Saving your comments makes them available the next time that you open the results file, allowing you to avoid reviewing the same check twice.

To save your review comments, select **File > Save**. Your comments are saved with the verification results.

Review Checks Using Predefined Methodologies

This example shows how to incrementally review checks using predefined review methodologies provided by Polyspace Code Prover.

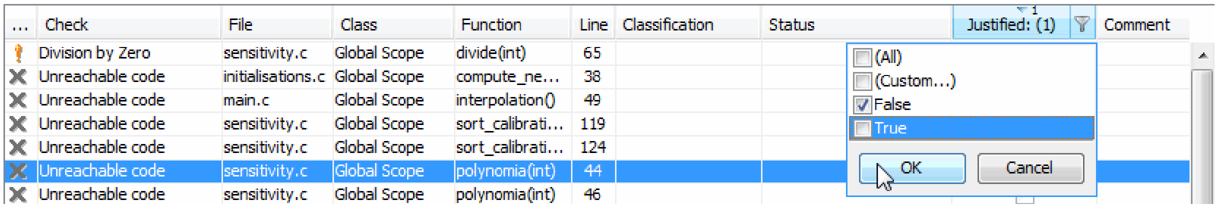
1. In the Results manager perspective, from the drop-down list above the **Results Summary** pane, select the methodology, **First checks to review**.



   The **Results Summary** pane displays all red and gray checks, as well as orange checks most likely to be run-time errors. Investigate and fix the errors, set the status of the checks, and justify them.

2. Once all the checks have been justified, select the methodology, **Methodology for C/C++ > Light**.

   In the **Results Summary** pane, you can view all red, gray, and purple checks, as well as a subset of orange checks. To filter the unjustified checks, from the drop-down list beside the **Justified** column header, clear all boxes except **False** and select **OK**.



...	Check	File	Class	Function	Line	Classification	Status	Justified: (1)	Comment
	Division by Zero	sensitivity.c	Global Scope	divide(int)	65				
	Unreachable code	initialisations.c	Global Scope	compute_ne...	38				
	Unreachable code	main.c	Global Scope	interpolation()	49				
	Unreachable code	sensitivity.c	Global Scope	sort_calibrati...	119				
	Unreachable code	sensitivity.c	Global Scope	sort_calibrati...	124				
	Unreachable code	sensitivity.c	Global Scope	polynomia(int)	44				
	Unreachable code	sensitivity.c	Global Scope	polynomia(int)	46				

   Investigate and fix the errors, set the status of the checks, and justify them.

3 Once all the checks have been justified, to see a larger subset of orange checks, select **Methodology for C/C++ > Moderate**. The number of orange checks in the **Results Summary** pane increases.

   To refresh the list to show unjustified checks only, reopen the drop-down list beside the **Justified** column header and select **OK**.

   Investigate and fix the errors, set the status of the checks and justify them.

4 To exhaustively review all of the checks, select **All checks**. In addition to all orange checks, this methodology also reveals all the green checks in the **Results Summary** pane.

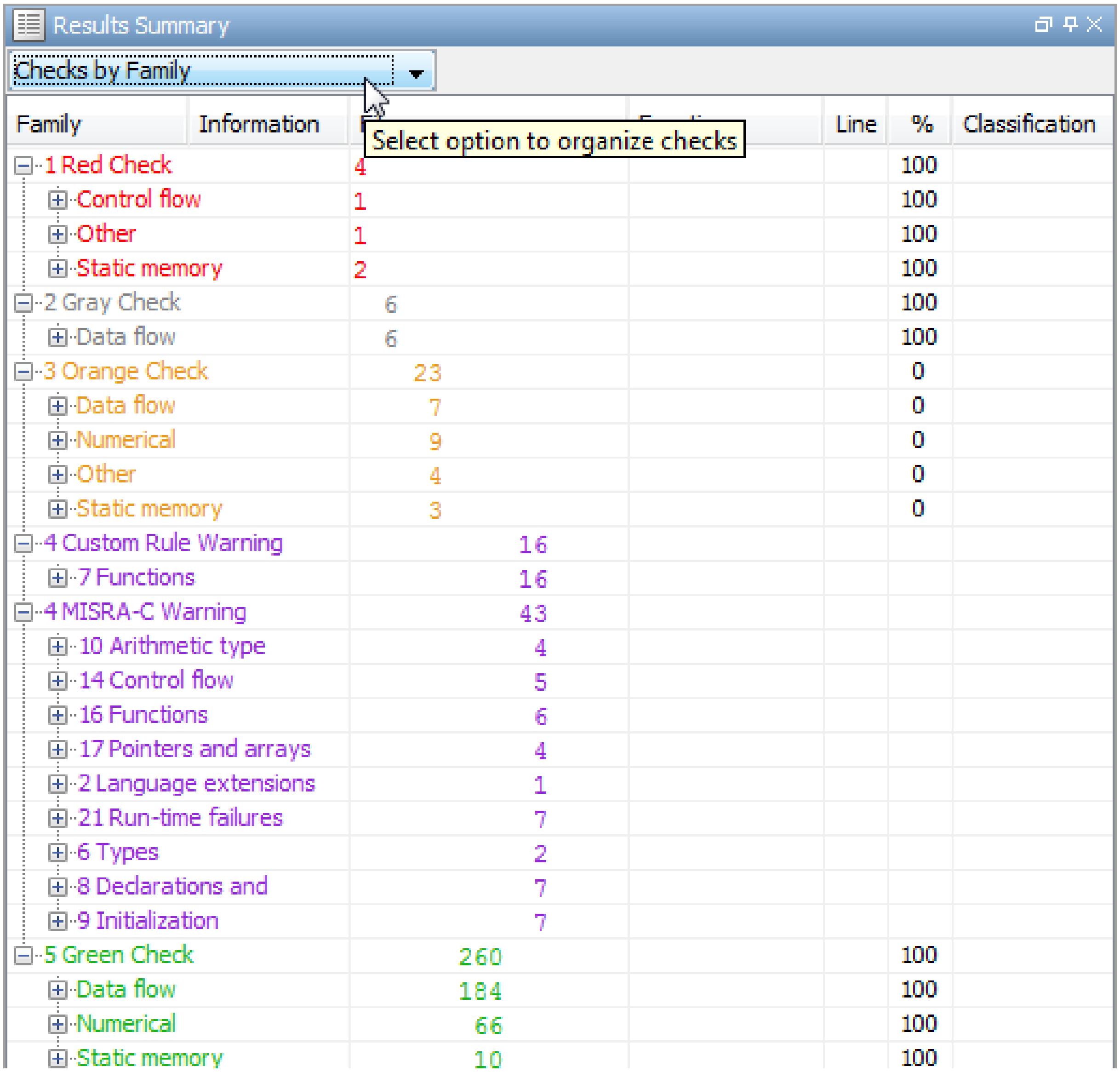
Organize Check Review Using Filters and Groups

To review all checks resulting from `Illegally dereferenced pointer`:

1 Open the results file, with extension, `.pscp`.

2 On the **Results Summary** pane, from the drop-down list, select `Checks by Family`.

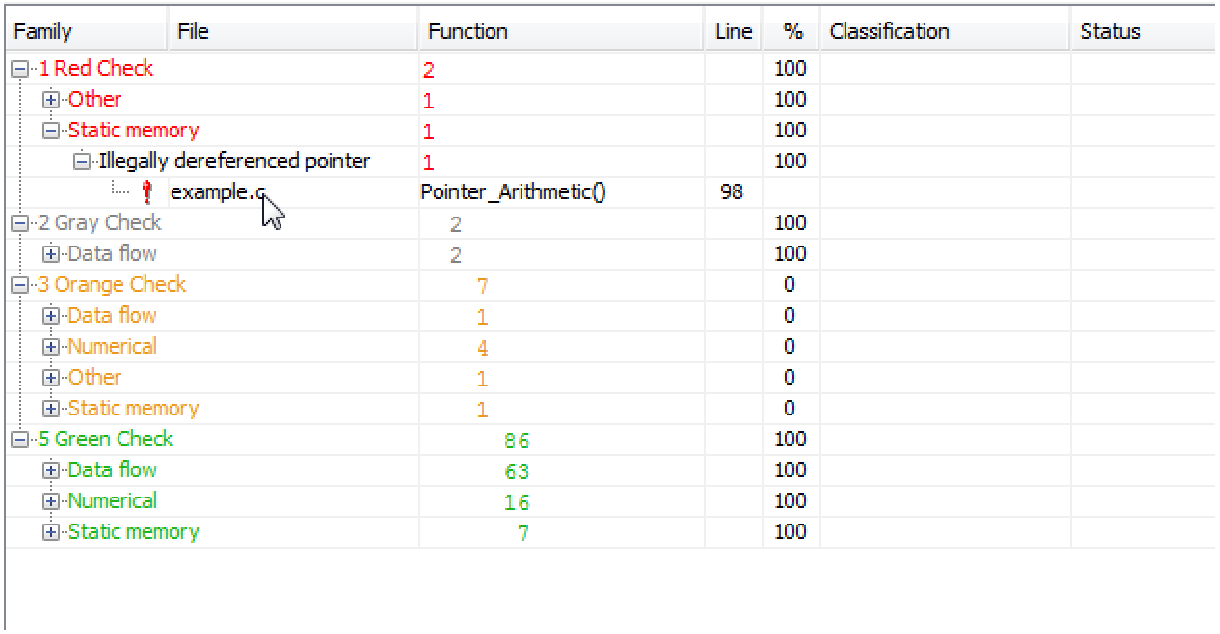
   The checks are grouped by type of check.



Family	Information	F	[illegible]	Line	%	Classification
⊟ 1 Red Check		4			100	
⊞ Control flow		1			100	
⊞ Other		1			100	
⊞ Static memory		2			100	
⊟ 2 Gray Check		6			100	
⊞ Data flow		6			100	
⊟ 3 Orange Check		23			0	
⊞ Data flow		7			0	
⊞ Numerical		9			0	
⊞ Other		4			0	
⊞ Static memory		3			0	
⊟ 4 Custom Rule Warning		16				
⊞ 7 Functions		16				
⊟ 4 MISRA-C Warning		43				
⊞ 10 Arithmetic type		4				
⊞ 14 Control flow		5				
⊞ 16 Functions		6				
⊞ 17 Pointers and arrays		4				
⊞ 2 Language extensions		1				
⊞ 21 Run-time failures		7				
⊞ 6 Types		2				
⊞ 8 Declarations and		7				
⊞ 9 Initialization		7				
⊟ 5 Green Check		260			100	
⊞ Data flow		184			100	
⊞ Numerical		66			100	
⊞ Static memory		10			100	

3 Under the category **1 Red Check**, expand the subcategory **Static memory**.

You see the subcategory **Illegally dereferenced pointer**.



Family	File	Function	Line	%	Classification	Status
1 Red Check		2		100		
Other		1		100		
Static memory		1		100		
Illegally dereferenced pointer		1		100		
	example.c	Pointer_Arithmetic()	98			
2 Gray Check		2		100		
Data flow		2		100		
3 Orange Check		7		0		
Data flow		1		0		
Numerical		4		0		
Other		1		0		
Static memory		1		0		
5 Green Check		86		100		
Data flow		63		100		
Numerical		16		100		
Static memory		7		100		

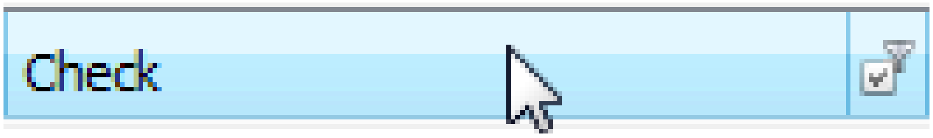
Expand **Illegally dereferenced pointer** to view all red checks resulting from this error.

To see further information about a check, select it. The information appears on the **Check Details** pane.

4 To view all orange checks resulting from this error, repeat step 3 for the subcategory **Static memory** under the category **3 Orange Check**.

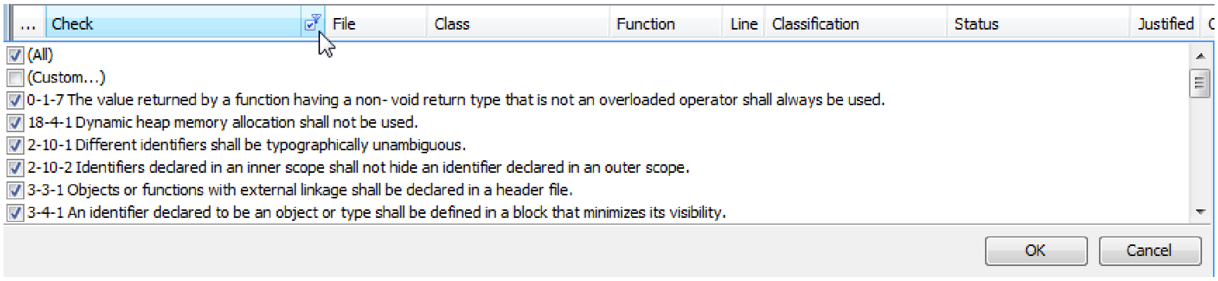
5 To view only the checks resulting from the error, `Illegally dereferenced pointer`, on the **Results Summary** pane, from the drop-down list, select `List of Checks`.

6 Place your cursor on the **Check** column head.



7 Click the filter icon.

A context menu lists all the filter options available.



☑ (All)
☐ (Custom...)
☑ 0-1-7 The value returned by a function having a non- void return type that is not an overloaded operator shall always be used.
☑ 18-4-1 Dynamic heap memory allocation shall not be used.
☑ 2-10-1 Different identifiers shall be typographically unambiguous.
☑ 2-10-2 Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
☑ 3-3-1 Objects or functions with external linkage shall be declared in a header file.
☑ 3-4-1 An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

8 Clear the **All** check box.

9 Scroll down to the **Illegally dereferenced pointer** check box and select it. Click **OK**.

The **Results Summary** pane displays only the checks resulting from the `Illegally dereferenced pointer` error.

Automatically Test Unproven Code

Reviewing orange code to find true errors is a time-consuming task. You can use the Automatic Orange Tester to automatically create and run test cases to identify errors in the orange code. The workflow for using the Automatic Orange Tester is:

1 Set an option to indicate that you want the software to run the Automatic Orange Tester at the end of the verification.

2 Run the verification. The software uses results from the Automatic Orange Tester to identify potential run-time errors.

3 If you want to perform further dynamic tests on the code, run the Automatic Orange Tester manually.

4 Review the results.

To learn how to use the Automatic Orange Tester, see "Test Orange Checks Automatically".

Generate Reports of Verification Results

Polyspace Report Generator Overview

The Polyspace Report Generator allows you to generate reports about your verification results, using predefined report templates.

Report Templates. The Polyspace Report Generator provides the following report templates:

- **Coding Rules Report** – Provides information about compliance with MISRA C®, MISRA® AC AGC, MISRA C++, JSF C++, and custom coding rules, as well as Polyspace configuration settings for the verification.
- **Developer Report** – Provides information useful to developers, including summary results, coding rule violations, detailed lists of red, orange, and gray checks, and Polyspace configuration settings for the verification. Detailed results are sorted by type of check (Proven Run-Time Violations, Proven Unreachable Code Branches, Unreachable Functions, and Unproven Run-Time Checks).
- **Developer Review Report** – Provides the same information as the Developer Report, but reviewed results are sorted by review classification (`High`, `Medium`, `Low`, `Not a defect`) and status, and untagged checks are sorted by file location.
- **Developer with Green Checks Report** – Provides the same content as the Developer Report, but also includes a detailed list of green checks.
- **Quality Report** – Provides information useful to quality engineers, including summary results, statistics about the code, graphs showing distributions of checks per file, and Polyspace configuration settings for the verification.
- **Software Quality Objectives Report** – Provides comprehensive information on software quality objectives (SQO), including code metrics, code analysis (coding-rules checker results), code verification (run-time checks), and the configuration settings for the verification. The code metrics section provides the same information displayed by Polyspace Metrics.

Generate Report for `example.c`

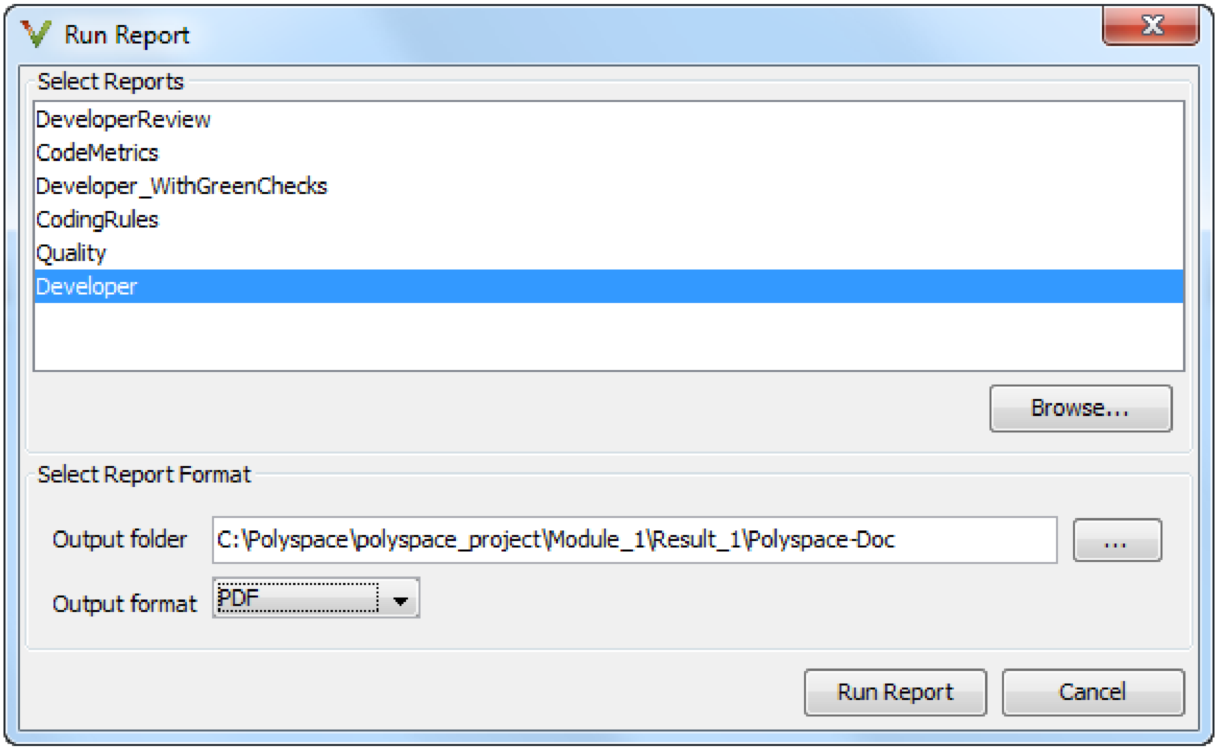
You can generate reports for any verification results using the Polyspace Report Generator.

To generate a verification report:

1. If your verification results are not already open, open them.

2. Select **Run > Run Report > Run Report...**.

   The Run Report dialog box opens.



Output folder C:\Polyspace\polyspace_project\Module_1\Result_1\Polyspace-Doc

Output format PDF

3. In the Select Report Template section, select **Developer**.

4. In the Output folder section, select the folder `polyspace_project\Module_1\Result_1\Polyspace-Doc`.

5. Select `PDF` Output format.

6. Click **Run Report**.

   The software creates the specified report. When report generation is complete, the report opens.

6

Check Compliance with Coding Rules

Check Compliance with Coding Rules

In this section...

Tutorial Overview

Polyspace software allows you to analyze code to demonstrate compliance with established C or C++ coding standards (MISRA C 2004, MISRA C++:2008, or JSF++:2005).[1]

Applying coding rules can both reduce the number of orange checks in your verification results and improve the quality of your code. Coding rules are the most efficient way to reduce orange checks.

To check compliance with coding rules, you set an option in your project and then run a verification. Polyspace software finds the violations during the compile phase of a verification. When you have addressed all coding rule violations, you run the verification again.

For more information on the coding rules checker, see "Overview of Polyspace Code Analysis".

In this tutorial, you learn how to:

1 Create a module within your project.

1. MISRA and MISRA C are registered trademarks of MISRA Ltd., held on behalf of the MISRA Consortium.

2 Set an option for checking MISRA C compliance.

3 Select MISRA C rules to check.

4 Run a verification with MISRA C checking.

5 View coding rules violations using the Coding Rules perspective.

Before You Start

Before you start this tutorial, you must complete "Set Up Polyspace Project" on page 2-2. For this tutorial, you use the folders from that tutorial.

Create New Module for Coding Rules Checking

- "Open Your Example Project" on page 6-3
- "Create New Module for MISRA C Checking" on page 6-4
- "Configure Text Editor" on page 6-7

Open Your Example Project

For this tutorial, you modify `example_project.psprj` to include MISRA C checking.

To open `example_project.psprj`:

1 Select **File > Open Project**.

2 In the Open Project dialog box, navigate to `polyspace_project`.

3 Select `example_project.psprj`.

4 Click **Open**.

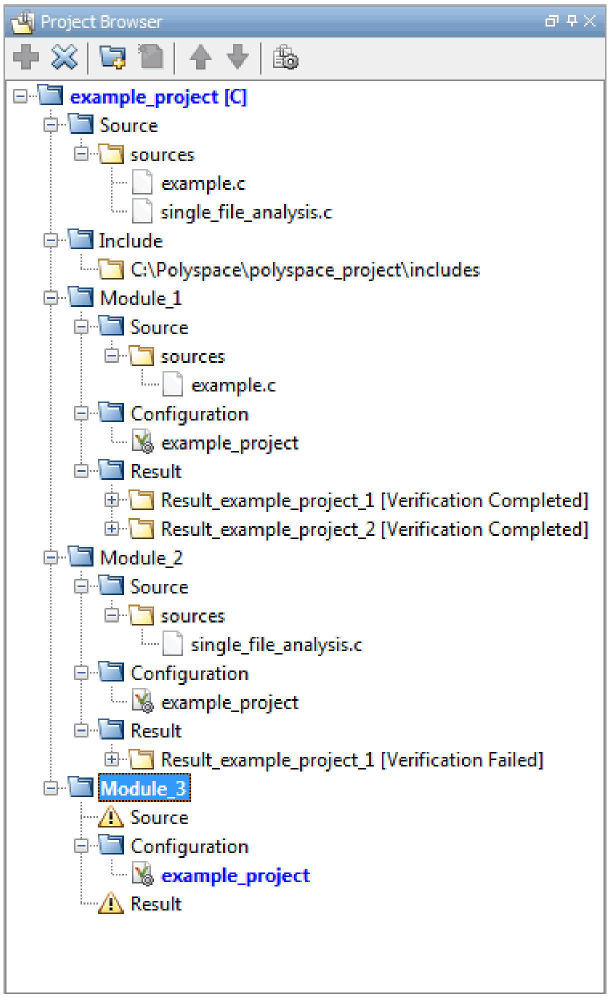
Create New Module for MISRA C Checking

A Polyspace Code Prover project can contain multiple modules. Each of these modules can verify a specific set of source files using a specific set of analysis options. In this section, you create a third module to check coding rules compliance for the `example.c` file and a new configuration, `example_project_1`.

1. From the Project Manager perspective, in the Project Browser, select **example_project [C]**.

2. On the Project Browser toolbar, click the **Create new module** icon.

   You see a new module , `Module_3`.



```
example_project [C]
    Source
        sources
            example.c
            single_file_analysis.c
    Include
        C:\Polyspace\polyspace_project\includes
    Module_1
        Source
            sources
                example.c
        Configuration
            example_project
        Result
            Result_example_project_1 [Verification Completed]
            Result_example_project_2 [Verification Completed]
    Module_2
        Source
            sources
                single_file_analysis.c
        Configuration
            example_project
        Result
            Result_example_project_1 [Verification Failed]
    Module_3
        Source
        Configuration
            example_project
        Result
```

3 Under **example_project [C] > Source**, right-click `example.c`. From the context menu, select **Copy Source File to > Module_3**.

4 Under **Module_3**, right-click the **Configuration** folder. From the context menu, select **Create New Configuration**.

5 Right-click the `example_project_1` configuration. From the context menu, select **Set as Active Configuration**.

Configure Text Editor

Before you check MISRA rules, configure your text editor in the Polyspace Preferences dialog box, which allows you to view source files directly from the Results Manager perspective.

To configure your text editor:

1. From the Polyspace Code Prover toolbar, select **Options > Preferences**.
2. In the Polyspace Preferences dialog box, click the **Editors** tab.
3. In the **Text Editor** field, specify an editor for viewing source files from the Project Manager logs. For example:

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

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

   - `Emacs`
   - `Notepad++` — Windows only
   - `UltraEdit`
   - `VisualStudio`
   - `WordPad` — Windows only
   - `gVim`

   If you are using another text editor, select `Custom` from the drop-down list, and specify the command-line arguments for the text editor.

5. Click **OK**.

Set MISRA C Checking Option

You set up MISRA C checking by setting an analysis option and then selecting the rules to check. To set the MISRA C checking option:

1. In the Project Browser, select the `example_project_1` configuration.
2. Select the **Configuration > Coding Rules** pane.

3 Select the **Check MISRA C rules** check box.

4 Use the corresponding drop-down list to specify the rules. For example, select `required-rules`.

5 You can also specify the following options:

- **Files and folders to ignore** — Files, if any, to exclude from the checking.
- **Effective boolean types** — Data types that you want Polyspace to consider as Boolean.
- **Allowed pragmas** — Undocumented pragma directives for which rule MISRA C 3.4 must not be applied.

Select Coding Rules to Check

You must have a rules file to run a verification with MISRA C checking. You can use an existing file or create a new one. You create a new rules file for this tutorial by:

- "Creating a MISRA C Rules File" on page 6-8
- "Set All Rules to Off" on page 6-10
- "Selecting Rules to Check" on page 6-10

Creating a MISRA C Rules File

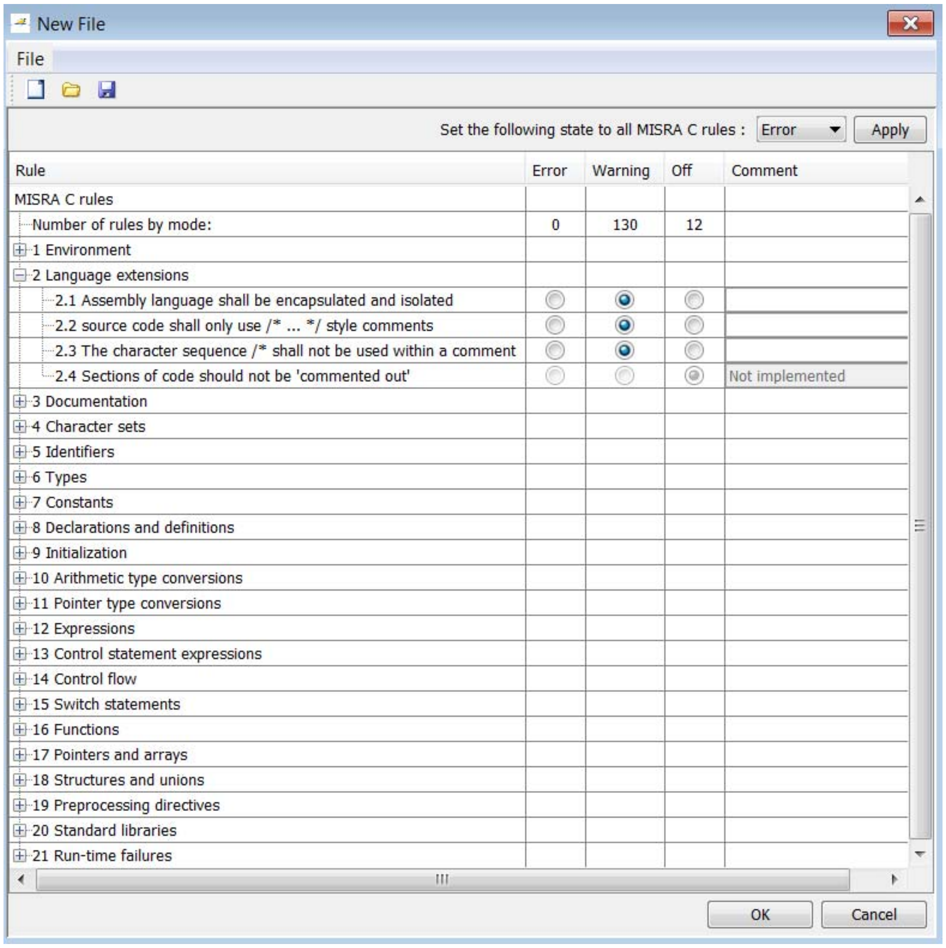
To open a new rules file:

1 In the Project Manager perspective, select the **Configuration > Coding Rules** pane.

2 Select the **Check MISRA C rules** check box.

3 From the corresponding drop-down list, select `custom`.

4 Click the **Edit** button. The New File dialog box opens, displaying a table of rules.



Set the following state to all MISRA C rules : Error Apply

Rule	Error	Warning	Off	Comment
MISRA C rules				
Number of rules by mode:	0	130	12	
1 Environment				
2 Language extensions				
2.1 Assembly language shall be encapsulated and isolated	○	●	○	
2.2 source code shall only use /* ... */ style comments	○	●	○	
2.3 The character sequence /* shall not be used within a comment	○	●	○	
2.4 Sections of code should not be 'commented out'	○	○	●	Not implemented
3 Documentation				
4 Character sets				
5 Identifiers				
6 Types				
7 Constants				
8 Declarations and definitions				
9 Initialization				
10 Arithmetic type conversions				
11 Pointer type conversions				
12 Expressions				
13 Control statement expressions				
14 Control flow				
15 Switch statements				
16 Functions				
17 Pointers and arrays				
18 Structures and unions				
19 Preprocessing directives				
20 Standard libraries				
21 Run-time failures				

5 For each rule, specify one of the following states.

State	Causes the verification to ...
Error	End after the compile phase when this rule is violated.
Warning	Display warning message and continue verification when this rule is violated.
Off	Skip checking of this rule.

The default state for most rules is `Warning`. The state for rules that have not yet been implemented is `Off`. Some rules have a fixed state of `Error`, which you cannot change.

6 Click **OK**.

7 Use the Save as dialog box to save your rules file.

Set All Rules to Off

In this tutorial, you select only a few rules. Therefore, first set the state of all rules to `Off`. Later, you can select the specific rules that you want to check.

To set the state of all rules to `Off`:

1 In the New File dialog box, from the **Set the following state to all MISRA C rules** drop-down list, select `Off`.

2 Click **Apply**.

Selecting Rules to Check

To select the rules to check for this tutorial:

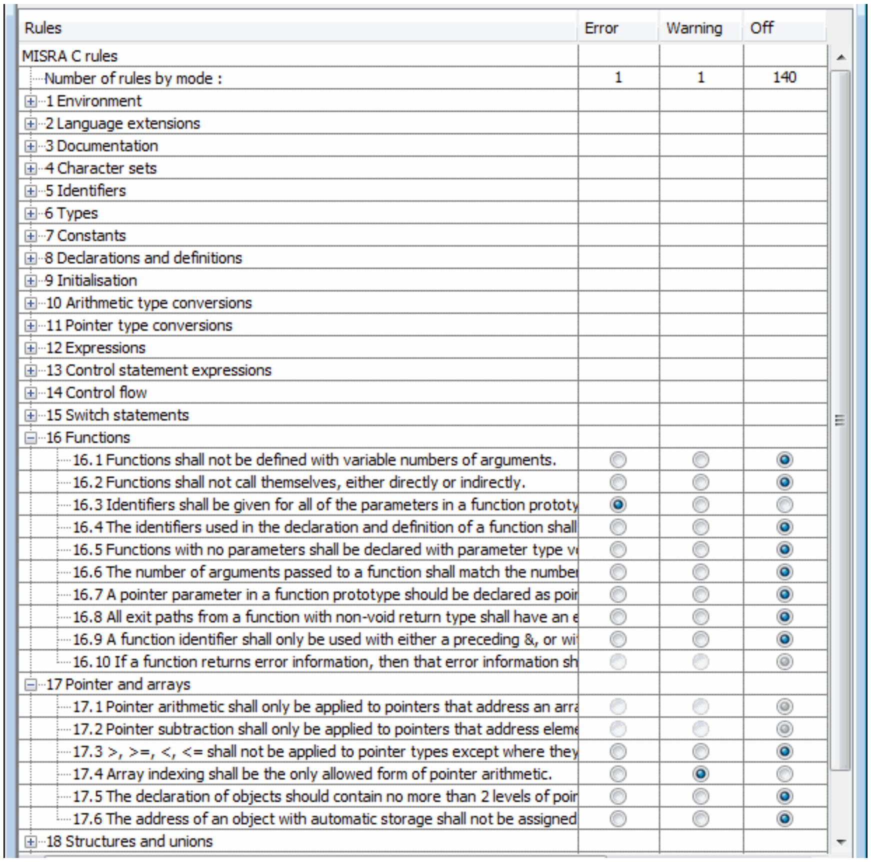
1 Expand the set of rules named `16 Functions`.

2 Select the **Error** column for `16.3`.

3 Expand the set of rules named `17 Pointers and arrays`.

4 Select the **Warning** column for `17.4`.

The completed rules table looks like the following figure.



Rules	Error	Warning	Off
MISRA C rules			
Number of rules by mode :	1	1	140
1 Environment			
2 Language extensions			
3 Documentation			
4 Character sets			
5 Identifiers			
6 Types			
7 Constants			
8 Declarations and definitions			
9 Initialisation			
10 Arithmetic type conversions			
11 Pointer type conversions			
12 Expressions			
13 Control statement expressions			
14 Control flow			
15 Switch statements			
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 prototy	●	○	○
16.4 The identifiers used in the declaration and definition of a function shall	○	○	●
16.5 Functions with no parameters shall be declared with parameter type v	○	○	●
16.6 The number of arguments passed to a function shall match the numbe	○	○	●
16.7 A pointer parameter in a function prototype should be declared as poi	○	○	●
16.8 All exit paths from a function with non-void return type shall have an e	○	○	●
16.9 A function identifier shall only be used with either a preceding &, or wi	○	○	●
16.10 If a function returns error information, then that error information sh	○	○	●
17 Pointer and arrays			
17.1 Pointer arithmetic shall only be applied to pointers that address an arra	○	○	●
17.2 Pointer subtraction shall only be applied to pointers that address eleme	○	○	●
17.3 >, >=, <, <= shall not be applied to pointer types except where they	○	○	●
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 poi	○	○	●
17.6 The address of an object with automatic storage shall not be assigned	○	○	●
18 Structures and unions			

5 Click **OK** to save the rules and close the window.

6 In the Save as dialog box, in the **File**, field, enter `misrac.txt`

7 Click **OK** to save the file and close the dialog box.

Exclude Files from MISRA C Checking

You can exclude files from MISRA C checking. For example, you might want to exclude some include files. To exclude `math.h` from the MISRA C checking of the project `example_project.psprj`:

1 In the Project Manager perspective, select the **Configuration > Coding Rules** pane.

2 Select the **Files and folders to ignore** check box.

3 From the corresponding drop-down list, select `custom`.

4 In the **File/Folder** view, click .

5 Use the Open File dialog box to navigate to the folder `polyspace_project\includes`.

6 Select the file `math.h`.

7 Click **Open**.

   You see the file `math.h` in the **File/Folder** view.

Run a Verification with Coding Rules Checking

When you run a verification with the MISRA C option selected, the software checks most of the MISRA C rules during the compile phase.[2]

To start the verification:

1 In the Project Browser, select your project configuration, for example, `example_project_1`.

2 On the Project Manager toolbar, click the **Run** button .

   The verification fails because of MISRA C violations. You see messages in the **Full Log**, and the **Output Summary** indicates that the verification has detected MISRA errors. If a rule with state `Error` is violated, the verification stops.

Examine MISRA C Violations

To examine the MISRA C violations:

2. MISRA and MISRA C are registered trademarks of MISRA Ltd., held on behalf of the MISRA Consortium.

1 In the Project Browser Result folder, double-click **MISRA-C-report.xml**, which opens the Results Manager perspective.

2 On the **Results Summary** pane, right-click on any column header. Select **Type**. Place your cursor on the **Type** column and select the filter icon . Clear the **All** check box and select the **4 MISRA-C Warning** check box.

   Only coding-rule violations remain on the **Results Summary** pane.

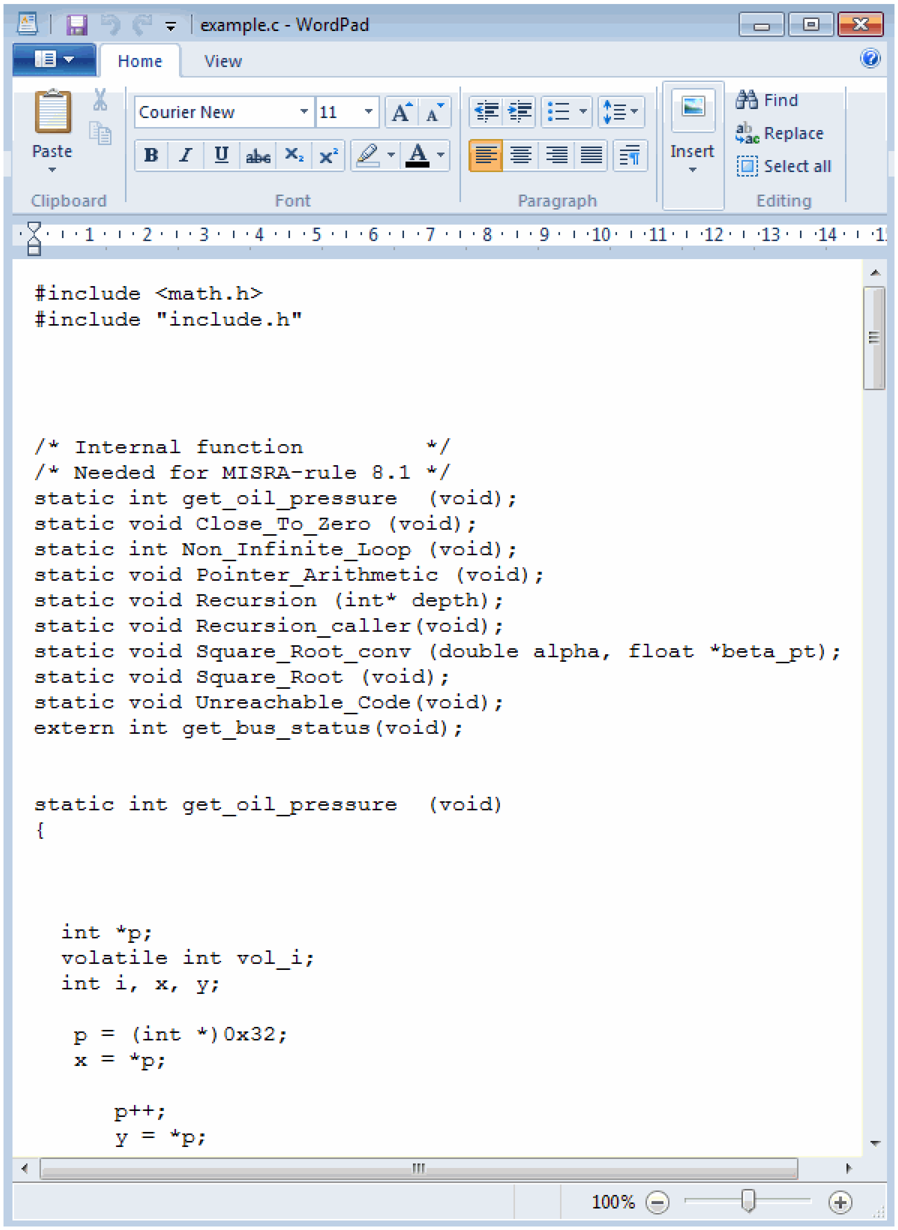
3 Click any violation.

   In the **Check Details** pane, you see a description of the violated rule and the name of the file in which the violation was found. In the **Source** pane, you see the source code that contains the violation.

   The code uses a form of pointer arithmetic that is not allowed, a violation of rule 17.4.

4 In the **Source** pane, right-click the highlighted code containing the violation of rule 17.4. From the context menu, select **Open Source File**. Before you can open source files, you must configure a text editor. See “Configure Text Editor” on page 6-7.

   The `example.c` file opens in your text editor.



```
#include <math.h>
#include "include.h"



/* Internal function         */
/* Needed for MISRA-rule 8.1 */
static int get_oil_pressure  (void);
static void Close_To_Zero (void);
static int Non_Infinite_Loop (void);
static void Pointer_Arithmetic (void);
static void Recursion (int* depth);
static void Recursion_caller(void);
static void Square_Root_conv (double alpha, float *beta_pt);
static void Square_Root (void);
static void Unreachable_Code(void);
extern int get_bus_status(void);


static int get_oil_pressure  (void)
{



  int *p;
  volatile int vol_i;
  int i, x, y;

   p = (int *)0x32;
   x = *p;

      p++;
      y = *p;
```

5 Fix the MISRA violation and run the verification again. The results are the same as those from the tutorial in “Run Verification” on page 4-2.

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 Options from Simulink”.

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”.

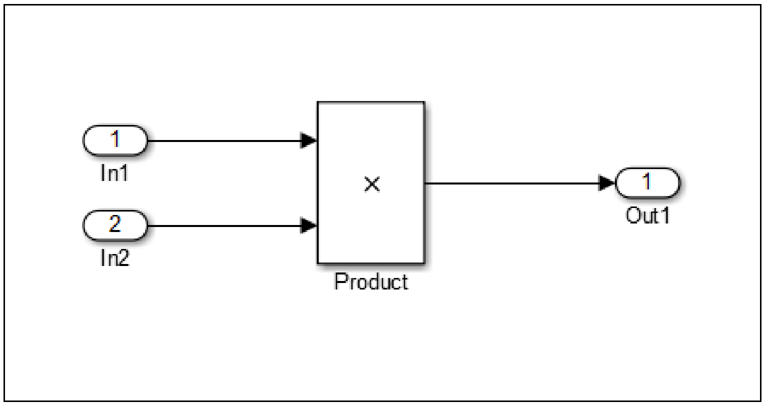
Verify Code from a Simple Simulink Model

In this section...

Create Simulink Model and Generate Code

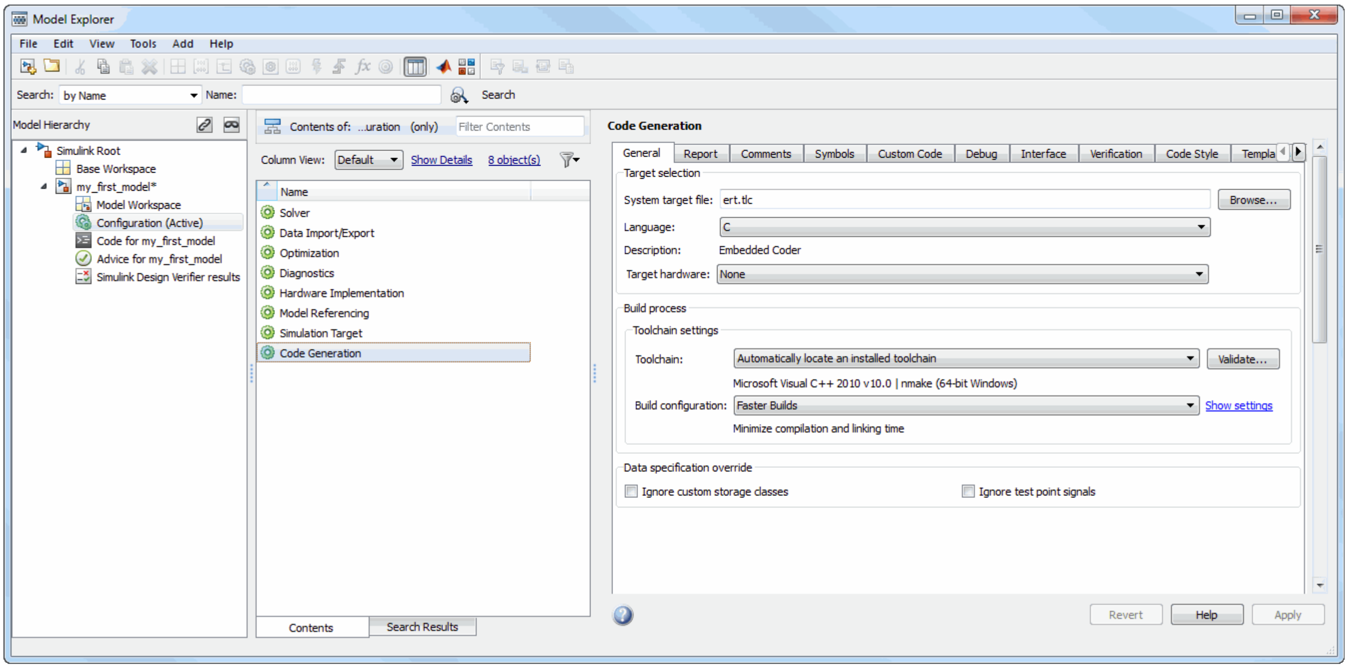
To create a simple Simulink model and generate code:

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



3. Select **File > Save**. Then name the model `my_first_code`.
4. Select **Tools > Model Explorer**. The Model Explorer opens.
5. From the **Model Hierarchy** tree, expand the node **my_first_code**.

6 Select **Configuration > Code Generation**, which displays Code Generation configuration parameters.



Code Generation

Target selection

System target file: ert.tlc

Language: C

Description: Embedded Coder

Target hardware: None

Build process

Toolchain settings

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

Data specification override

☐ Ignore custom storage classes ☐ Ignore test point signals

7 Select the **General** tab, and then set the **System target file** to `ert.tlc (Embedded Coder)`.

8 Select the **Report** tab.

9 Select **Create code-generation report**, and then select **Code-to-model** navigation.

10 Select the **Templates** tab.

11 In the **Custom templates** section, clear the check box **Generate an example main program**.

12 Select the **Interface** tab.

13 In the **Code interface** section, select the **Suppress error status in real-time model data structure** check box.

14 Click **Apply** in the lower-right corner of the window.

15 Select **Configuration > Solver**, which displays Solver configuration parameters.

16 In the **Solver options** section, set the solver **Type** to `Fixed-step`. Then, set the **Solver** to `discrete (no continuous states)`.

17 Click **Apply**.

18 Select **Configuration > Optimization**, which displays Optimization configuration parameters. Then:

- On the **General** tab, in the **Data initialization** section, select the **Remove root level I/O zero initialization** check box.
- On the **General** tab, clear the **Use memset to initialize floats and doubles to 0.0** check box
- On the **Signals and Parameters** tab, in the **Simulation and code generation** section, select the **Inline parameters** check box.

19 Click **Apply**.

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

21 Save your Simulink model.

Run Polyspace Verification

To start the 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.

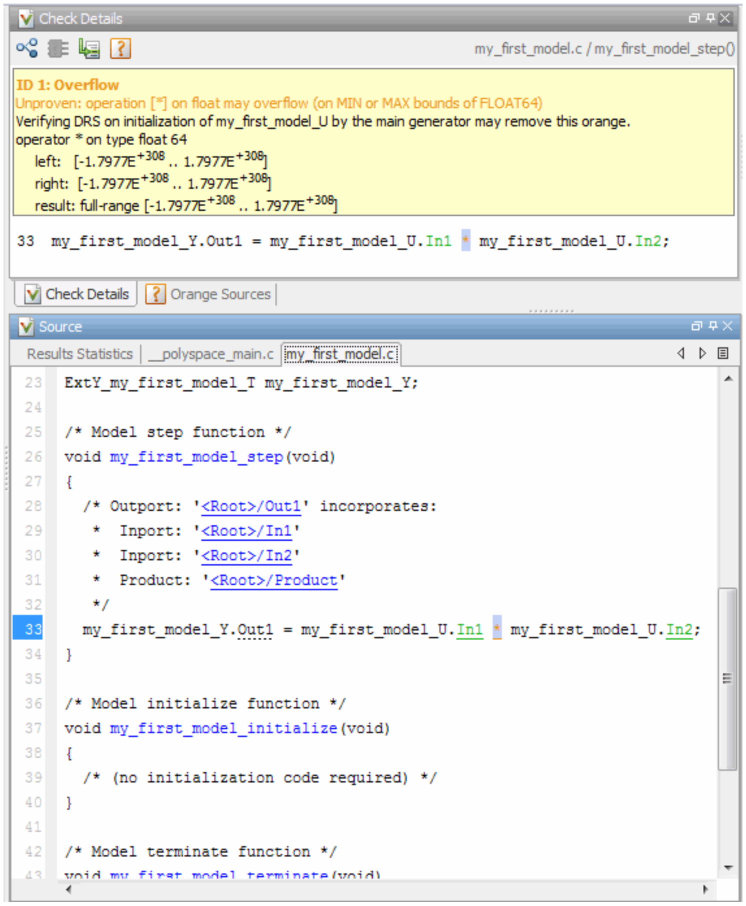
1 From the Simulink model window, select **Code > Polyspace > Open Results > For Generated Code**.

After a few seconds, the Results Manager perspective opens.

2 In the **Results Summary** view, select the drop-down menu and change the results organization to `List of Checks`.

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_model_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_model_Y.Out1 = my_first_model_U.In1 * my_first_model_U.In2;
```

```
ExtY_my_first_model_T my_first_model_Y;

/* Model step function */
void my_first_model_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Product: '<Root>/Product'
   */
  my_first_model_Y.Out1 = my_first_model_U.In1 * my_first_model_U.In2;
}

/* Model initialize function */
void my_first_model_initialize(void)
{
  /* (no initialization code required) */
}

/* Model terminate function */
void my_first_model_terminate(void)
```

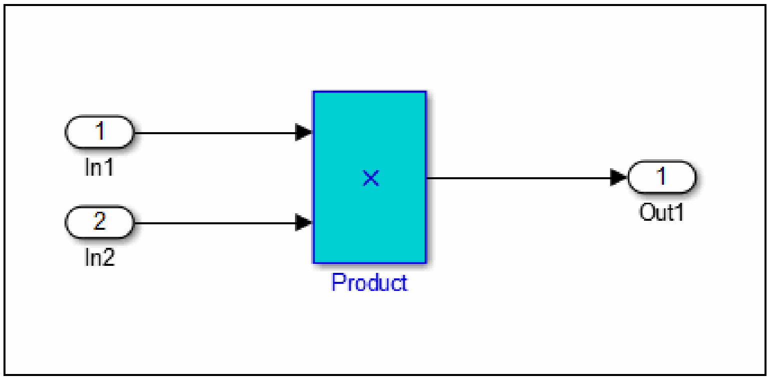
This orange check shows a potential overflow issue when multiplying the signals from the inports `In1` and `In2`. Polyspace software assumes that the signal values are full range, and the multiplication of the two signals may result in an overflow.

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:

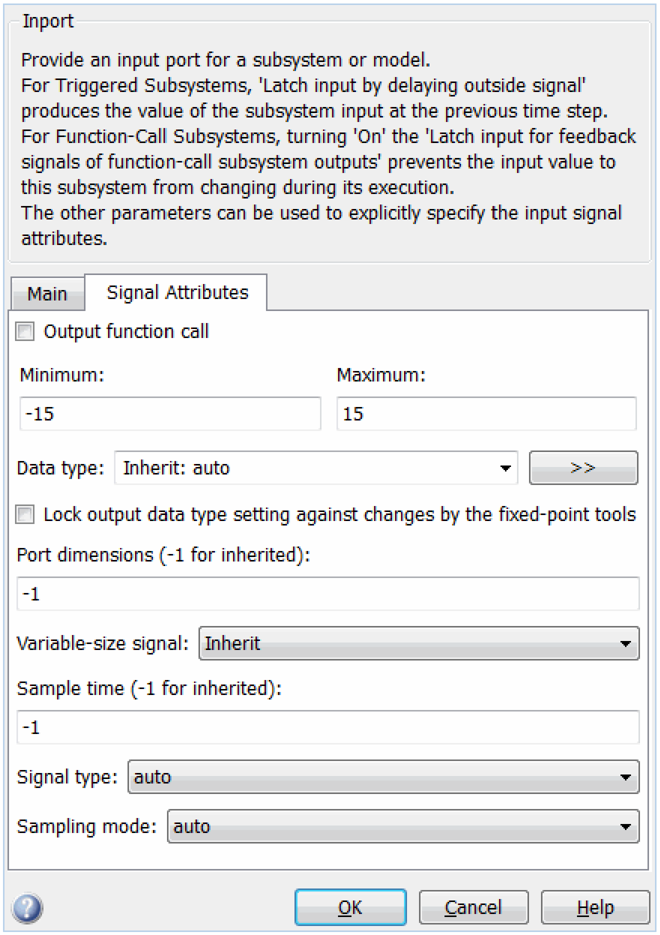
- **Design** — If the model should be robust for the full signal range, then the issue is a design flaw. In this case, you must change the model to accommodate the full signal range. For example, you could saturate the output of the previous block, or bound the signal with a Switch block.
- **Specifications** — 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 signals in your Simulink model to specified ranges, Polyspace software automatically applies these constraints during verification of the generated code. The Overflow check will then be 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 Repeat steps 1–6 for the `In2` block.

7 Save your model as `my_first_code_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.

4. In the Results Manager perspective, select the **Results Explorer** tab.

   The Overflow check is now green. Polyspace verification shows that no run-time errors are present in the model.

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 and "Monitoring a Verification" on page 8-8.
- View results, analyze errors, and locate faulty code within model. See "Viewing Polyspace Results" on page 8-8 and "Locating Faulty Code in Rhapsody Model" on page 8-9.

Adding Polyspace Profile to Model

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

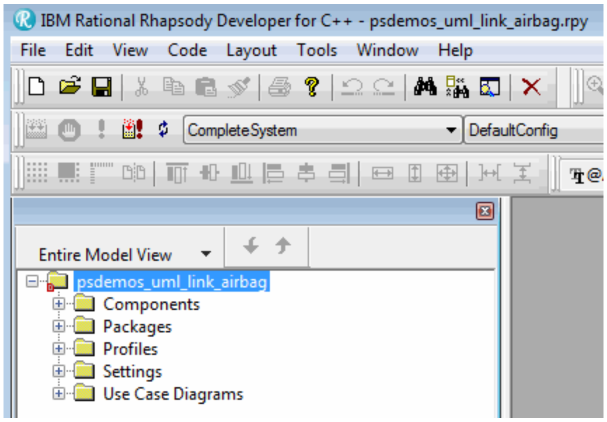
1. In the Rhapsody editor, select **File > Add Profile to Model**. The Add Profile to Model dialog box opens.
2. Navigate to the folder *`MATLAB_Install`*`\polyspace\plugin\rhapsody\profiles\Polyspace`.
3. Select the file `Polyspace.sbs`. Then click **Open**.

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`.

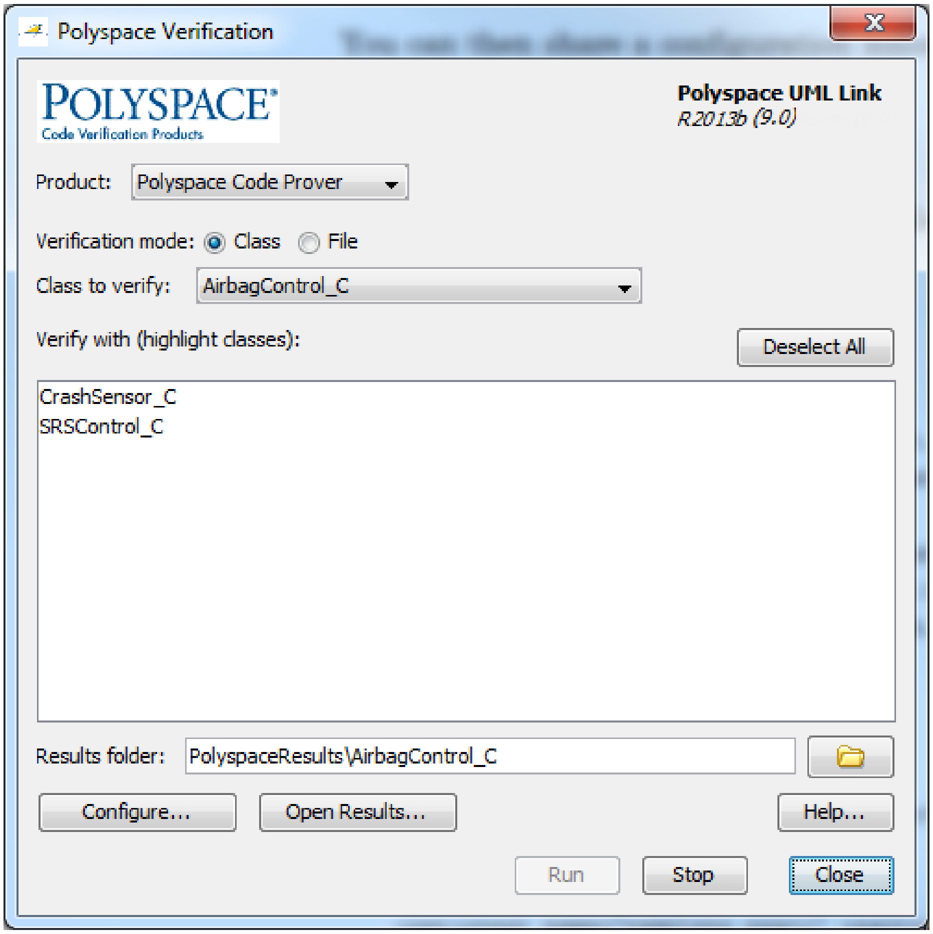


2 In the **Entire Model View**, expand the `Packages` node.

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

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

The Polyspace Verification dialog box opens.



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:

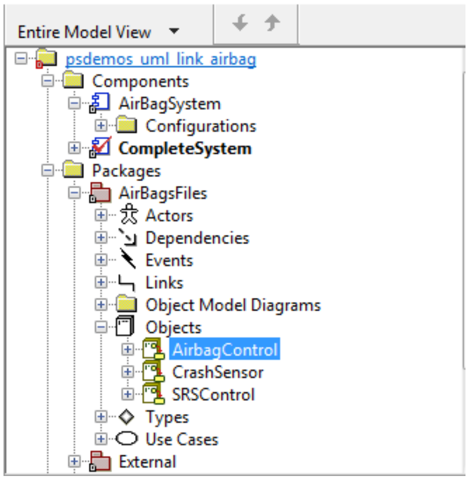
- Specify verification options. See “Configuring Verification Options” on page 8-6.
- Start a verification. See “Running a Verification” on page 8-7.
- Stop a local verification. See “Running a Verification” on page 8-7.
- View verification results. See “Viewing Polyspace Results” on page 8-8.
- Open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

- Open the Polyspace Queue Manager. See “Monitoring a Verification” on page 8-8.

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

To stop a local verification, in the Polyspace Verification dialog box, click **Stop**.

To stop a remote verification, use Polyspace Metrics or the Queue Manager. See:

- "Manage Previous Verifications With Polyspace Metrics"
- "Manage Remote Verifications"

.

Monitoring a Verification

If your verification is local, you can observe progress in the **Log** view of the Rhapsody editor.

If your verification is remote, use Polyspace Metrics or the Queue Manager.

For more information, see:

- "Manage Previous Verifications With Polyspace Metrics"
- "Manage Remote Verifications"

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 Queue Manager.

For more information, see "Run-Time Error Review".

Declarations for C Functions Without Arguments

By default, Rhapsody generates declarations for functions without any 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 8-10
- "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>
```

Index