Polyspace® Bug Finder™
User's Guide

R2014**a**

MATLAB® & SIMULINK®

MathWorks®

Polyspace® Bug Finder™ User's Guide

Revision History

September 2013	Online only	New for Version 1.0 (Release 2013b)
March 2014	Online Only	Revised for Version 1.1 (Release 2014a)

Contents

1 Project Configuration

What Is a Project? **1-3**

What is a Project Template? **1-4**

Open Polyspace Bug Finder **1-5**

Set Up Polyspace Metrics **1-7**
Requirements for Polyspace Metrics **1-7**
Start Polyspace Metrics Server **1-8**
Configure Polyspace Preference **1-9**
Configure Web Server for HTTPS **1-10**
Change Web Server Port Number for Polyspace Metrics Server **1-12**

Set Up Remote Verification and Analysis **1-13**
Requirements for Remote Analysis **1-14**
Start Server for Remote Analysis and Polyspace Metrics . . **1-15**
Configure Polyspace Preferences **1-16**

Create New Project **1-17**

Add Source Files and Include Folders **1-19**

Specify Analysis Options **1-21**
Specify Options in User Interface **1-21**
Specify Options from DOS and UNIX Command Line **1-22**
Specify Options from MATLAB Command Line **1-22**

Save Analysis Options as Project Template **1-24**

Specify Text Editor **1-27**

Define Custom Review Status **1-29**

Compilation Errors **1-33**

Model Synchronous Tasks **1-34**

Prepare Multitasking Code **1-37**
Model Interruptions and Asynchronous Events and Tasks **1-37**
Are Interruptions Maskable or Preemptive? **1-39**
Model Shared Variables **1-41**
Model Mailbox Messaging **1-45**
Atomicity and Interrupted Instructions **1-48**

Priorities **1-50**

Annotate Code for Known Defects **1-51**
How to Add Annotations **1-51**
Syntax for Annotations **1-51**

Annotate Code for Rule Violations **1-54**
How to Add Annotations **1-54**
Syntax for Annotations **1-55**

Copy and Paste Annotations **1-58**

Predefined Target Processor Specifications **1-59**

Modify Predefined Target Processor Attributes **1-62**

Specify Generic Target Processors **1-64**
Define Generic Target **1-64**
Common Generic Targets **1-65**
View or Modify Existing Generic Targets **1-66**
Delete Generic Target **1-67**

Compile Operating System-Dependent Code **1-69**
Predefined Compilation Flags for C Code **1-69**
Predefined Compilation Flags for C++ Code **1-71**

My Target Application Runs on Linux 1-73
My Target Application Runs on Solaris 1-74
My Target Application Runs on Vxworks 1-74
My Target Application Does Not Run on Linux, vxworks nor Solaris 1-75

Address Alignment **1-76**

Ignore or Replace Keywords Before Compilation **1-77**
Content of myTpl.pl file 1-77
Perl Regular Expression Summary 1-78

Analyze Keil or IAR Dialects **1-80**

Gather Compilation Options Efficiently **1-88**

Specify Data Ranges for Global Variables **1-90**
Overview of Data Range Specifications (DRS) 1-90
Specify Data Ranges Using DRS Template 1-90
Remove Non Applicable Entries from DRS File 1-92
DRS Configuration Settings 1-93
Specify Data Ranges Using Existing DRS Configuration . . 1-96
Edit Existing DRS Configuration 1-97
XML Format of DRS File 1-98
Specify Data Ranges Using Text Files 1-104

2 **Setting Up Project: Additional Information**

Create Projects Using Visual Studio Information **2-2**
Use Visual Studio Project 2-2
Trace Visual Studio Build 2-3

Cannot create project from Visual Studio build **2-6**

Storage of Polyspace Preferences **2-7**

3 Coding Rule Sets and Concepts

Rule Checking **3-2**

Custom Naming Convention Rules **3-3**

Polyspace MISRA C and MISRA AC AGC Checkers **3-10**

Software Quality Objective Subsets (C) **3-11**
Rules in `SQO-Subset1` **3-11**
Rules in `SQO-Subset2` **3-13**

Software Quality Objective Subsets (AC AGC) **3-16**
Rules in `SQO-Subset1` **3-16**
Rules in `SQO-Subset2` **3-16**

MISRA C:2004 Coding Rules **3-18**
Supported MISRA C:2004 Rules **3-18**
MISRA C:2004 Rules Not Checked **3-56**

Polyspace MISRA C++ Checker **3-59**

Software Quality Objective Subsets (C++) **3-60**
SQO Subset 1 – Direct Impact on Selectivity **3-60**
SQO Subset 2 – Indirect Impact on Selectivity **3-63**

MISRA C++ Coding Rules **3-69**
Supported MISRA C++ Coding Rules **3-69**
MISRA C++ Rules Not Checked **3-89**

Polyspace JSF C++ Checker **3-95**

JSF C++ Coding Rules **3-96**
Supported JSF C++ Coding Rules **3-96**
JSF++ Rules Not Checked **3-121**

4 Check Coding Rules from the Polyspace Environment

Activate Coding Rules Checker 4-2

Select Specific MISRA or JSF Coding Rules 4-6

Create Custom Coding Rules 4-8

Format of Custom Coding Rules File 4-10

Exclude Files From Rule Checking 4-12

Allow Custom Pragma Directives 4-13

Specify Boolean Types 4-14

Find Coding Rule Violations 4-15

Review Coding Rule Violations 4-16

Apply Coding Rule Violation Filters 4-18

5 Find Bugs From the Polyspace Environment

Choose Specific Defects 5-2

Run Local Analysis 5-3

Run Remote Batch Analysis 5-4

Monitor Analysis 5-5

Specify Results Folder **5-6**

6 View Results in the Polyspace Environment

Open Results **6-2**

View Results Summary in Polyspace Metrics **6-3**

Download Results From Polyspace Metrics **6-5**

Filter and Group Results **6-8**

Generate Reports **6-15**

Review and Comment Results **6-16**

Import Comments from Previous Analyses **6-20**

Code Metrics **6-21**

View Code Sequence Causing Defect **6-28**

Results Folder Contents **6-31**
Files in the Results Folder **6-31**
Files in the `ALL` Subfolder **6-31**
Files in the `Polyspace-Doc` Subfolder **6-32**

Windows in the Results Manager Perspective **6-33**
Dashboard **6-33**
Results Summary **6-37**
Source **6-39**
Check Details **6-45**

Bug Finder Defect Categories **6-47**

Numerical **6-47**
Static Memory **6-47**
Dynamic Memory **6-48**
Programming **6-48**
Data-flow **6-48**
Other **6-48**

Common Weakness Enumeration from Bug Finder Defects **6-49**

7 Command-Line Analysis

Run Analysis from the Command Line **7-2**
Usage of Bug Finder at the Command Line **7-2**
Complete Workflow Examples **7-2**

Manage Remote Analyses at the Command Line **7-4**

Create Projects Automatically from Your Build System **7-6**
Create Project in User Interface **7-6**
Create Project from DOS and UNIX Command Line **7-7**
Create Project from MATLAB Command Line **7-8**

Requirements for Project Creation from Build Systems **7-10**

8 Polyspace Bug Finder Analysis in Simulink

Embedded Coder Considerations **8-2**
Subsystems **8-2**
Default Options **8-2**

Recommended Polyspace Bug Finder Options for Analyzing Generated Code 8-3
Hardware Mapping Between Simulink and Polyspace 8-5

TargetLink Considerations 8-6
TargetLink Support 8-6
Subsystems 8-6
Default Options 8-6
Lookup Tables 8-7
Code Generation Options 8-7

Run Analysis on Generated Code 8-8

View Results in the Polyspace Environment 8-9

Identify Errors in Simulink Models 8-10

9 Configure Model for Code Analysis

Model Configuration for Code Generation and Analysis 9-2

Configure Simulink Model 9-3

Recommended Model Settings for Code Analysis 9-5

Check Simulink Model Settings 9-7

Check Simulink Model Settings Before Code Generation 9-8

Check Simulink Model Settings Before Analysis 9-10

Annotate Blocks for Known Errors or Coding-Rule Violations 9-12

10 Configure Code Analysis Options

Polyspace Configuration for Generated Code 10-2

Include Handwritten Code 10-3

Specify Remote Analysis 10-5

Configure Analysis Depth for Referenced Models 10-6

Specify Location of Results 10-7

Check Coding Rules Compliance 10-8

Configure Polyspace Options from Simulink 10-10

Configure Polyspace Project Properties 10-11

Create a Polyspace Configuration File Template 10-12

Specify Header Files for Target Compiler 10-15

Open Polyspace Results Automatically 10-16

Remove Polyspace Options From Simulink Model 10-17

11 Run Polyspace on Generated Code

Specify Type of Analysis to Perform 11-2

Run Analysis for Embedded Coder 11-5

Run Analysis for TargetLink **11-7**

Monitor Progress **11-8**
Local Analyses **11-8**
Remote Batch Analyses **11-8**

12 Check Coding Rules from Eclipse

Activate Coding Rules Checker **12-2**

Select Specific MISRA or JSF Coding Rules **12-6**

Create Custom Coding Rules File **12-8**

Contents of Custom Coding Rules File **12-10**

Exclude Files from Rules Checking **12-12**

Allow Custom Pragma Directives **12-13**

Specify Boolean Types **12-14**

Find Coding Rule Violations **12-15**

Review Coding Rule Violations **12-16**

Apply Coding Rule Violation Filters **12-18**

13 Find Bugs from Eclipse

Run Analysis 13-2

Customize Analysis Options 13-3

14 View Results in Eclipse

Filter and Group Results 14-2

View Results 14-8

Review and Fix Results 14-9

Understanding the Results Views 14-12
  Results Summary 14-12
  Check Details 14-14

15 Check Coding Rules from Microsoft Visual Studio

Activate C++ Coding Rules Checker 15-2

16 Find Bugs from Microsoft Visual Studio

Run Analysis 16-2

Monitor Analysis 16-5

Customize Polyspace Options 16-6

17 Open Results from Microsoft Visual Studio

Open Results in Polyspace Environment 17-2

Project Configuration

- "What Is a Project?" on page 1-3
- "What is a Project Template?" on page 1-4
- "Open Polyspace® Bug Finder™" on page 1-5
- "Set Up Polyspace Metrics" on page 1-7
- "Set Up Remote Verification and Analysis" on page 1-13
- "Create New Project" on page 1-17
- "Add Source Files and Include Folders" on page 1-19
- "Specify Analysis Options" on page 1-21
- "Save Analysis Options as Project Template" on page 1-24
- "Specify Text Editor" on page 1-27
- "Define Custom Review Status" on page 1-29
- "Compilation Errors" on page 1-33
- "Model Synchronous Tasks" on page 1-34
- "Prepare Multitasking Code" on page 1-37
- "Priorities" on page 1-50
- "Annotate Code for Known Defects" on page 1-51
- "Annotate Code for Rule Violations" on page 1-54
- "Copy and Paste Annotations" on page 1-58
- "Predefined Target Processor Specifications" on page 1-59
- "Modify Predefined Target Processor Attributes" on page 1-62
- "Specify Generic Target Processors" on page 1-64

- “Compile Operating System-Dependent Code” on page 1-69
- “Address Alignment” on page 1-76
- “Ignore or Replace Keywords Before Compilation” on page 1-77
- “Analyze Keil or IAR Dialects” on page 1-80
- “Gather Compilation Options Efficiently” on page 1-88
- “Specify Data Ranges for Global Variables” on page 1-90

What Is a Project?

In Polyspace® software, a project is a named set of parameters for your software project's source files. A project includes:

- Source files
- Include folders
- A configuration, specifying a set of analysis options

Use the Project Manager perspective in the Polyspace interface to create and modify a project.

What is a Project Template?

A **Project Template** is a predefined set of analysis options for a specific compilation environment. When creating a new project, you have the option to:

- Use an existing template to automatically set analysis options for your compiler.

  Polyspace software provides predefined templates for common compilers such as `IAR`, `Kiel`, and `VxWorks Aonix`, `Rational`, and `Greenhills`. For additional templates, see Polyspace Compiler Templates .

- Set analysis options manually. You can save your options to a custom template and reuse them later. For more information, see "Save Analysis Options as Project Template" on page 1-24.

Open Polyspace Bug Finder

In MATLAB®, do one of the following:

- In the apps gallery, click Polyspace Bug Finder™.
- In the Command Window, enter:

  ```
  polyspaceBugFinder
  ```

  There are additional options for this command. For help, enter:

  ```
  help polyspaceBugFinder
  ```

In Windows®, do one of the following:

- From the folder *`matlabroot`*`\polyspace\bin`, double-click the Polyspace Bug Finder icon.
- Double-click a desktop Polyspace Bug Finder shortcut.

  To create this shortcut, in the folder *`matlabroot`*`\polyspace\bin`, right-click `polyspace-bug-finder`. Then, from the context menu, select **Create shortcut**.
- At a DOS command prompt, enter:

  ```
  matlabroot\polyspace\bin\polyspace-bug-finder
  ```

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

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

In Linux®:

- Run the following command:

  ```
  matlabroot/polyspace/bin/polyspace-bug-finder
  ```

Polyspace Bug Finder can be opened simultaneously with Polyspace Code Prover™ or a second instance of Polyspace Bug Finder. However, only one code analysis can be run at a time.

If you try to run Polyspace processes from multiple windows, you will get a `License Error  4,0`. To avoid this error, close any additional Polyspace windows before running an analysis.

Set Up Polyspace Metrics

In this section...

"Requirements for Polyspace Metrics" on page 1-7

"Start Polyspace Metrics Server" on page 1-8

"Configure Polyspace Preference" on page 1-9

"Configure Web Server for HTTPS" on page 1-10

"Change Web Server Port Number for Polyspace Metrics Server" on page 1-12

Requirements for Polyspace Metrics

You can use Polyspace Metrics to:

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

The following table lists the requirements for Polyspace Metrics.

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

Task	Location	Requirements
Downloading *complete* results from Polyspace Metrics	Client node or a network computer	• MATLAB • Polyspace Bug Finder • Access to Polyspace Metrics server
Viewing results *summary* from Polyspace Metrics	A network computer	Access to Polyspace Metrics server.

Start Polyspace Metrics Server

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 If you want to configure your MDCS head node (for remote verifications and analyses) as the Polyspace Metrics server, select **Start the Polyspace mdce service without security level**. Otherwise, clear this check box. For more information about starting your remote cluster service, see "Set Up Remote Verification and Analysis" on page 1-13.

.

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

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

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

Configure Polyspace Preference

1. Select **Options > Preferences**.

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

3. Under **Metrics configuration**:

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

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

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

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

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

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

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

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

  If you change the port number from the default, you must configure the same port number for the Polyspace Metrics server. See "Change Web Server Port Number for Polyspace Metrics Server" on page 1-12 .

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

  There are additional steps to set up the Web server for HTTPS. See "Configure Web Server for HTTPS" on page 1-10.

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

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

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

Configure Web Server for HTTPS

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

Before you start the following procedure, you must complete "Start Polyspace Metrics Server" on page 1-8 and "Configure Polyspace Preference" on page 1-9.

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 *`AppData`*`Polyspace_RLDatas\tomcat\conf\server.xml` file in a text editor. Look for the following text:

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

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

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

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

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

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

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

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

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

`https://ServerName:WSPN`

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

Change Web Server Port Number for 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.

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

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

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

   - For HTTP:

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

   - For HTTPS:

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

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

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

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

Set Up Remote Verification and Analysis

In this section...

"Requirements for Remote Analysis" on page 1-14

"Start Server for Remote Analysis and Polyspace Metrics" on page 1-15

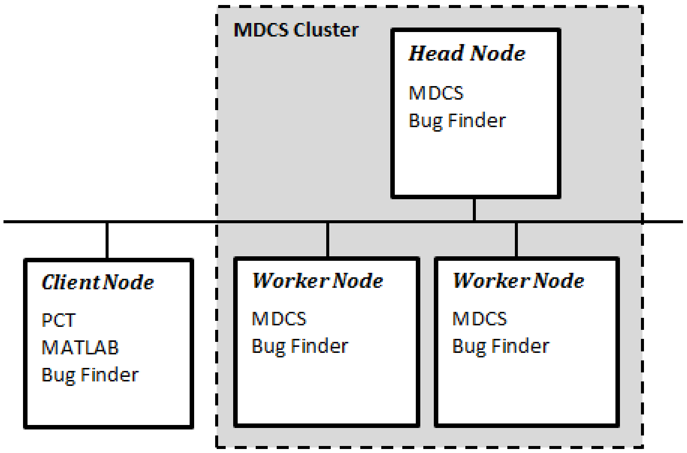
"Configure Polyspace Preferences" on page 1-16

You can run the following types of verification and analyses.

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

You can also use Polyspace Metrics with your remote verifications, but it is not required. For more information about setting up Polyspace Metrics, see "Set Up Polyspace Metrics" on page 1-7.

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



To set up remote verification:

1. Configure the head node with the Metrics and Remote Server Settings dialog box. See, "Start Server for Remote Analysis and Polyspace Metrics" on page 1-15.

2. Configure the client node through the **Server Configuration** tab. See, "Configure Polyspace Preferences" on page 1-16.

Requirements for Remote Analysis

The following table lists the requirements for remote analysis.

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

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

Start Server for Remote Analysis and Polyspace Metrics

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

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

   - **Start the Polyspace mdce service without security** — Selected. The `mdce` service, which is required to manage the MJS, runs on the MJS host computer with security level 0. If you want to require authentication to use the remote server, use the MDCS **Admin Center**. For more information about setting up security levels, see "Set MJS Cluster Security".
   - **MDCE service port** — 27350.
   - **Security level in the cluster** — 0. No security.

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

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

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

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

Configure Polyspace Preferences

1 Select **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**, specify the host computer for your Polyspace Metrics server or let Polyspace detect the server. For more information, see “Set Up Polyspace Metrics” on page 1-7.

Create New Project

This example shows how to create a new project in Polyspace Bug Finder. Before you create a project, you must know:

- Location of source files
- Location of include files
- Location where analysis results will be stored

For the three locations, you will find it convenient to create three subfolders under a common project folder. For instance, under the folder `polyspace_project`, you can create three subfolders `sources`,`includes` and `results`.

1. Select **File > New Project...**.

2. In the Project – Properties dialog box, enter the following information:

   - **Project name**
   - **Location**: Folder where you will store the project file with extension `.psprj`. You can use this file to open an existing project.

     The software assigns a default location to your project. You can change this default on the **Project and Results Folder** tab in the Polyspace Preferences dialog box.
   - **Project language**

   If you want to use a template, select the **Use template** check box. Then, click **Next**.

3. Select the template for your compiler. If your compiler does not appear in the list of predefined templates, select **Baseline**. You can then start with a generic template. Click **Next**.

4. Add source files and include folders to your project.

   - Navigate to the location where you stored your source files. Select the source files for your project. Click **Add Source Files**.

- The software automatically adds the standard include files to your project. To use custom include files, navigate to the folder containing your include files. Click **Add Include Folders**.

5 Click **Finish**.

   The new project opens in the **Project Browser**.

6 Save the project. Select **File > Save** or enter **Ctrl+S**.

Add Source Files and Include Folders

This example shows how to add source files and include folders to an existing project.

Add Sources and Includes

1 In the **Project Browser**, select your project.

2 Click the **Add source** icon ➕.

3 Add source files and include folders to your project.

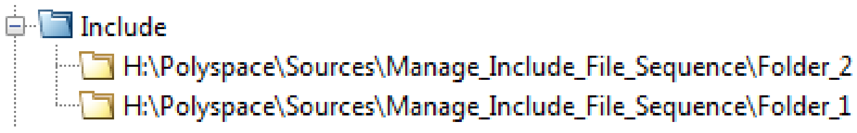
- Navigate to the location where you stored your source files. Select the source files for your project. Click **Add Source Files**.
- The software adds standard include files to your project. To use custom include files, navigate to the folder containing your include files. Click **Add Include Folders**.

4 Click **Finish**.

Manage Include File Sequence

You can change the order of include folders to manage the sequence in which include files are compiled. When multiple include files by the same name exist in different folders, it is convenient to change the order of include folders instead of reorganizing the contents of your folders. For a particular include file name, the software includes the file in the first include folder under ***`Project_Name`*** > **Include**.

In the following figure, `Folder_1` and `Folder_2` contain the same include file `include.h`. If your source code includes this header file, during compilation, `Folder_2/include.h` is included in preference to `Folder_1/include.h`.



Include
H:\Polyspace\Sources\Manage_Include_File_Sequence\Folder_2
H:\Polyspace\Sources\Manage_Include_File_Sequence\Folder_1

To change the order of include folders:

1 In the **Project Browser**, expand the **Include** folder.

2 Select the include folder that you want to move.

3 To move the folder, click either ⇧ or ⇩ on the Project Browser toolbar.

Related Examples

- “Specify Results Folder” on page 5-6
- “Create New Project” on page 1-17

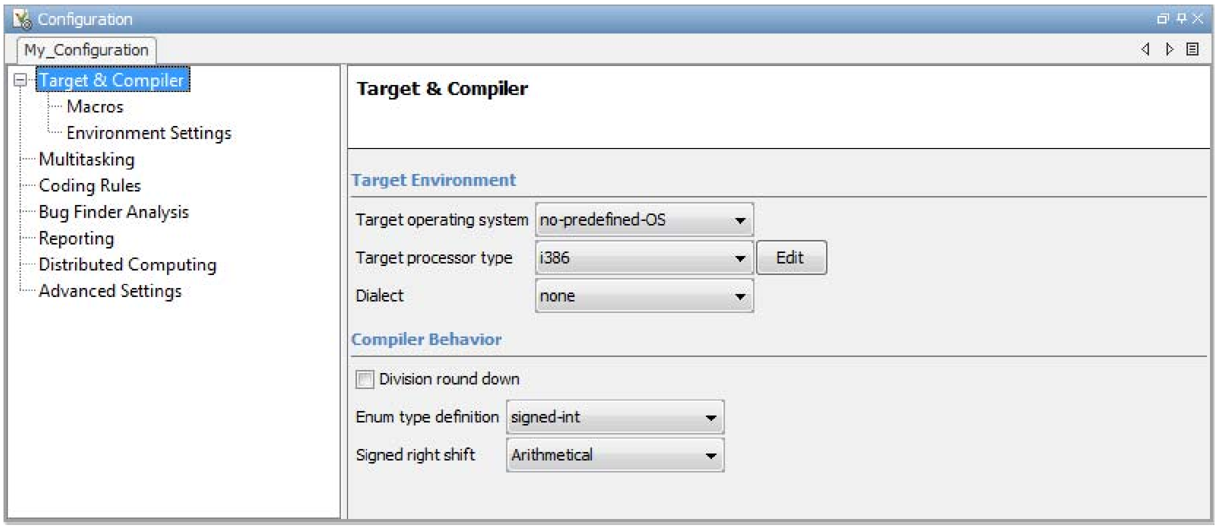
Specify Analysis Options

You can either retain the default analysis options used by the software or change them to your requirements. To specify analysis options:

In this section...
"Specify Options in User Interface" on page 1-21
"Specify Options from DOS and UNIX Command Line" on page 1-22
"Specify Options from MATLAB Command Line" on page 1-22

Specify Options in User Interface

In the Polyspace Project Manager perspective, use the **Configuration** pane.



Target operating system: no-predefined-OS
Target processor type: i386
Dialect: none
Division round down: ☐
Enum type definition: signed-int
Signed right shift: Arithmetical

For instance:

- To specify the target processor, select **Target & Compiler** in the **Configuration** tree view. Select your processor from the **Target processor type** drop-down list.

- To check for violation of MISRA C® rules, select **Coding Rules**. Check the **Check MISRA C Rules** box. To check for a subset of rules, select an option from the drop-down list.

Specify Options from DOS and UNIX Command Line

At the DOS or UNIX® command-line, append analysis options to the `polyspace-bug-finder-nodesktop` command. For instance:

- To specify the target processor, use the `-target` option. For instance, to specify the `m68k` processor for your source file `file.c`, use the command:

  ```
  polyspace-bug-finder-nodesktop -sources "file.c" -lang c -target m68k
  ```

- To check for violation of MISRA C rules, use the `-misra2` option. For instance, to check for only the required MISRA C rules on your source file `file.c`, use the command:

  ```
  polyspace-bug-finder-nodesktop -sources "file.c" -misra2 required-rules
  ```

Specify Options from MATLAB Command Line

At the MATLAB command-line, enter analysis options and their values as string arguments to the `polyspaceBugFinder` function. For instance:

- To specify the target processor, use the `-target` option. For instance, to specify the `m68k` processor for your source file `file.c`, enter:

  ```
  polyspaceBugFinder('-sources','file.c','-lang','c','-target','m68k')
  ```

- To check for violation of MISRA C rules, use the `-misra2` option. For instance, to check for only the required MISRA C rules on your source file `file.c`, enter:

  ```
  polyspaceBugFinder('-sources','file.c','-misra2','required-rules')
  ```

See Also

polyspaceBugFinder

Related Examples

- "Save Analysis Options as Project Template" on page 1-24

Concepts

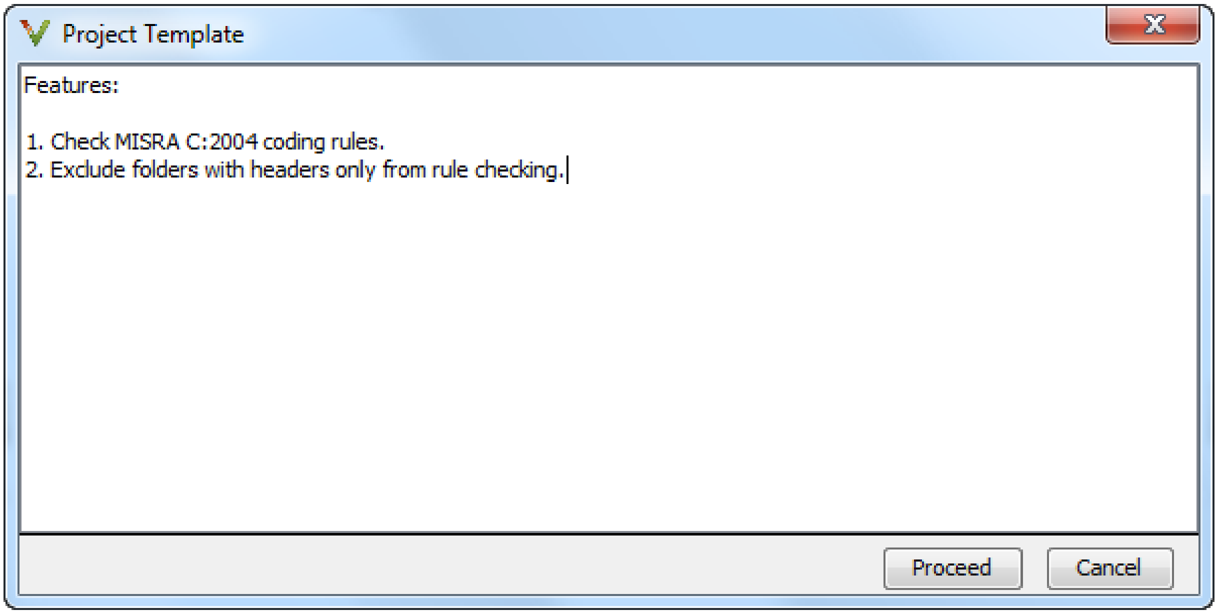
- “Analysis Options for C”
- “Analysis Options for C++”

Save Analysis Options as Project Template

This example shows how to save your analysis options for use in other projects. Once you have configured analysis options for a project, you can save the configuration as a **Project Template**. You can use this saved configuration to automatically set up analysis options for other projects.

- To create a **Project Template** from an open project:

  1. Right-click the configuration that you want to use, and then select **Save As Template**.
  2. Enter a description for the template, then click **Proceed**. Save your Template file.

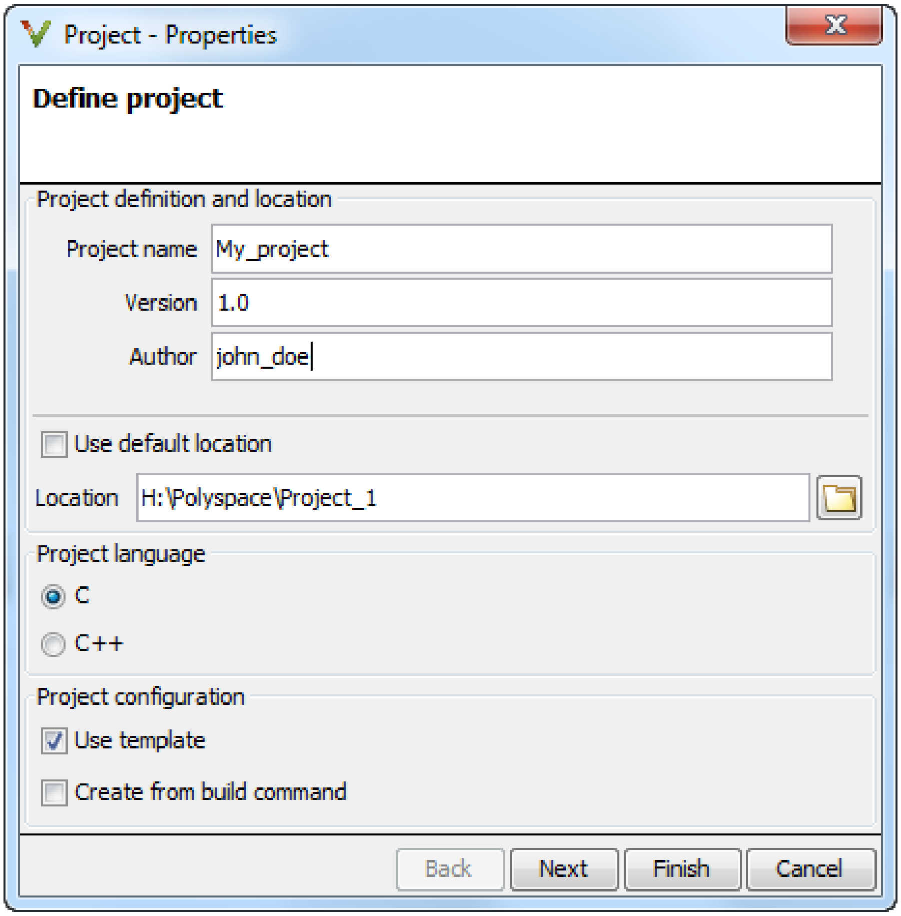


Features:

1. Check MISRA C:2004 coding rules.
2. Exclude folders with headers only from rule checking.

- When you create a new project, to use a saved template:

  1. Under **Project configuration**, check the **Use template** box. Click **Next**.


Project - Properties

Define project

Project definition and location

Project name: My_project

Version: 1.0

Author: john_doe

[ ] Use default location

Location: H:\Polyspace\Project_1

Project language

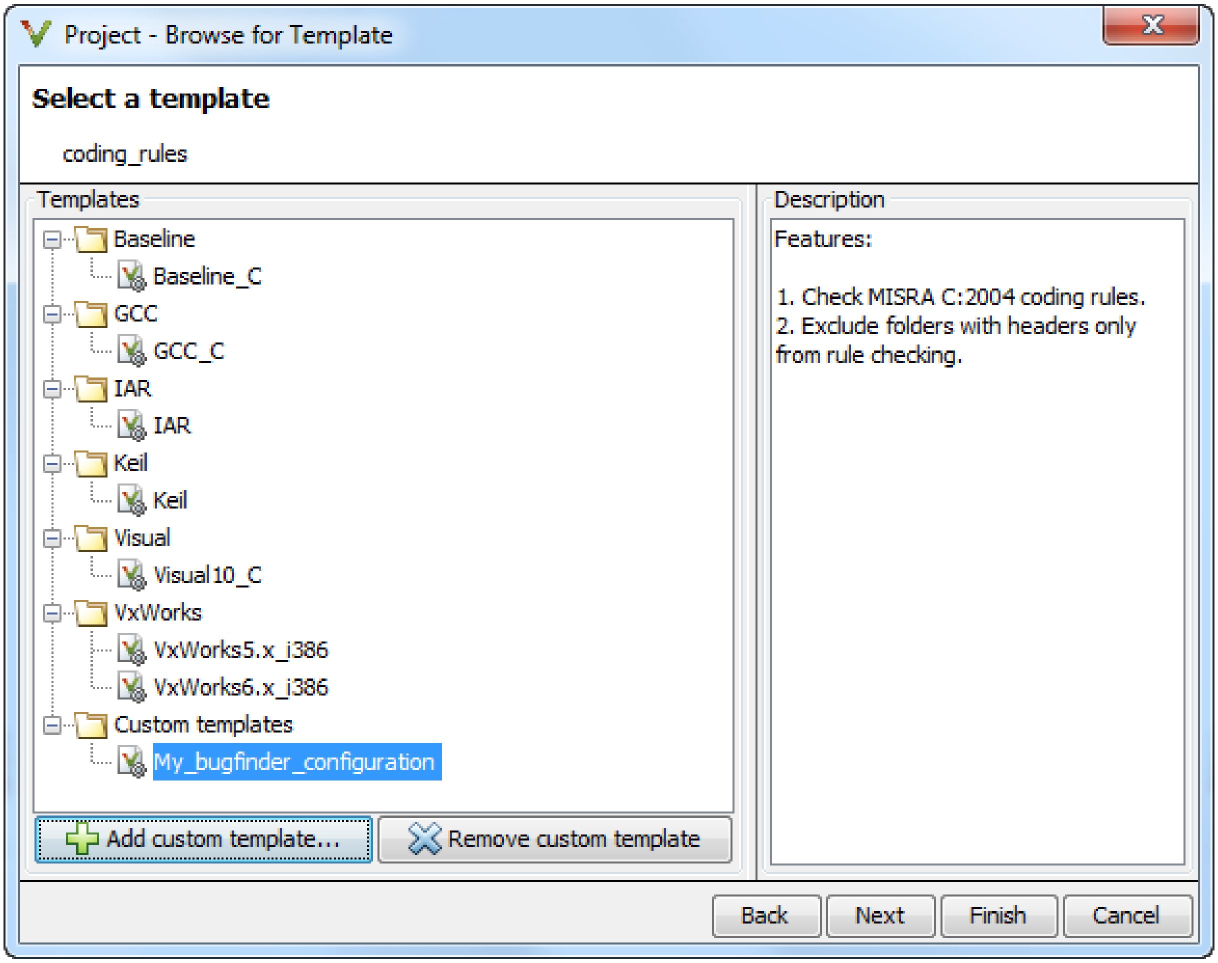
[✓ C] C++

Project configuration

[✓] Use template

[ ] Create from build command

2 Select **Add custom template...**. Navigate to the template that you saved earlier, and then click **Open**. The new template appears in the **Custom templates** folder on the **Templates** browser. Select the template for use.



Features:

1. Check MISRA C:2004 coding rules.
2. Exclude folders with headers only from rule checking.

Related Examples

- "Specify Analysis Options" on page 1-21

Concepts

- "Analysis Options for C"
- "Analysis Options for C++"

Specify Text Editor

This example shows how to change the default text editor for opening source files from the Polyspace interface. Polyspace uses WordPad as the default editor in Windows. It uses vi as the default editor in Linux.

1. Select **Options > Preferences**.

2. On the Polyspace Preferences dialog box, select the **Editors** tab.

3. In the **Text editor** field, specify the path to your text editor. For example:

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

4. To make sure that your source code opens at the correct line and column in your text editor, specify command-line arguments for the editor using Polyspace macros, `$FILE`, `$LINE` and `$COLUMN`. Once you specify the arguments, when you right-click a check on the **Results Summary** pane and select **Open Source File**, your source code opens at the location of the check.

   Polyspace has already specified the command-line arguments for the following editors:

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

   If you are using one of these editors, select it from the **Arguments** drop-down list. If you are using another text editor, select `Custom` from the drop-down list, and enter the command-line options in the field provided.

5. Click **OK**.

For console-based text editors, you must create a terminal. For example, to specify `vi`:

1. In the **Text Editor** field, enter `/usr/bin/xterm`.
2. From the **Arguments** drop-down list, select `Custom`.
3. In the field to the right, enter `-e /usr/bin/vi $FILE`.

Define Custom Review Status

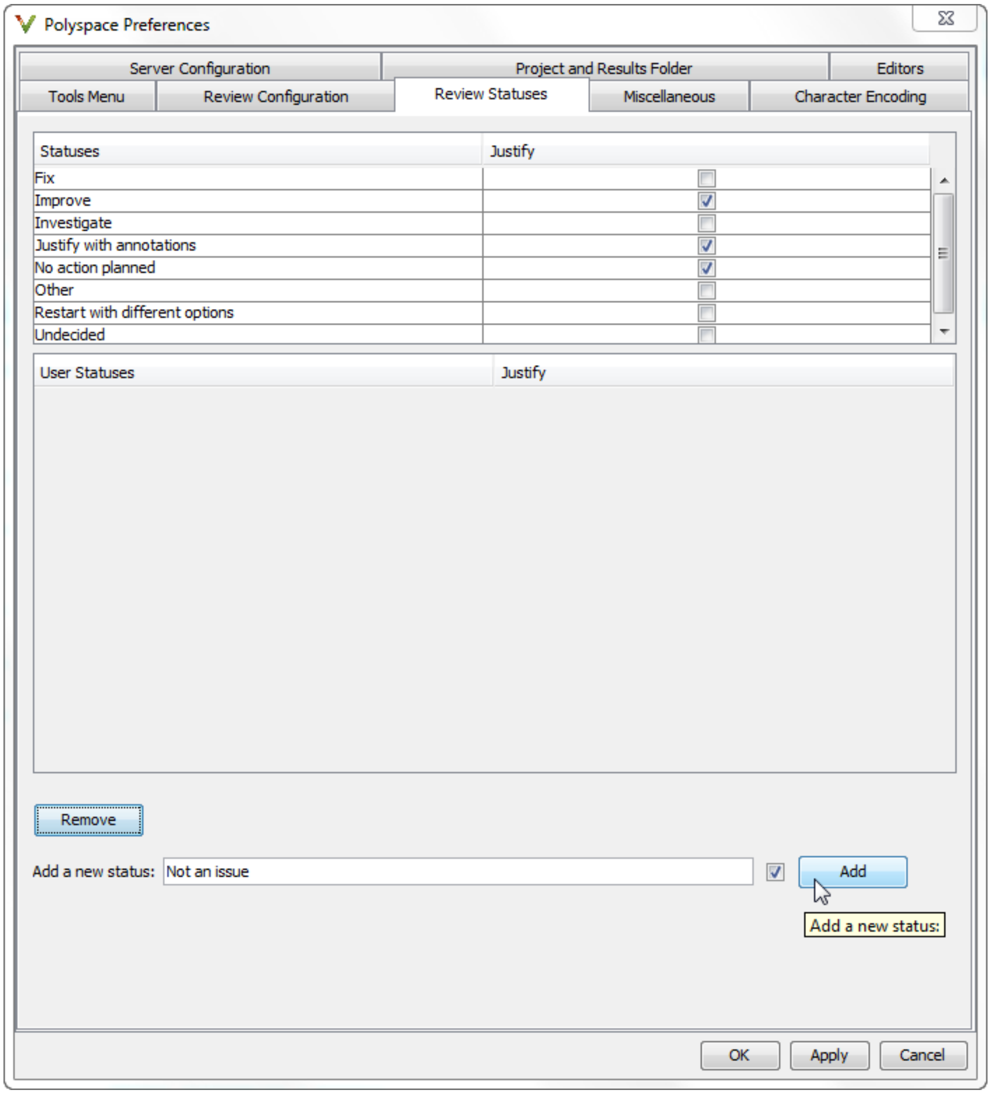
This example shows how to customize the statuses you assign on the **Check Review** pane.

Define Custom Status

1 Select **Options > Preferences**.

2 Select the **Review Statuses** tab.

3 Enter your new status at the bottom of the dialog box, then click **Add**.



Statuses	Justify
Fix	☐
Improve	✓
Investigate	☐
Justify with annotations	✓
No action planned	✓
Other	☐
Restart with different options	☐
Undecided	☐

User Statuses	Justify

Add a new status: Not an issue ✓

The new status appears in the **Status** list.

4 Click **OK** to save your changes and close the dialog box.

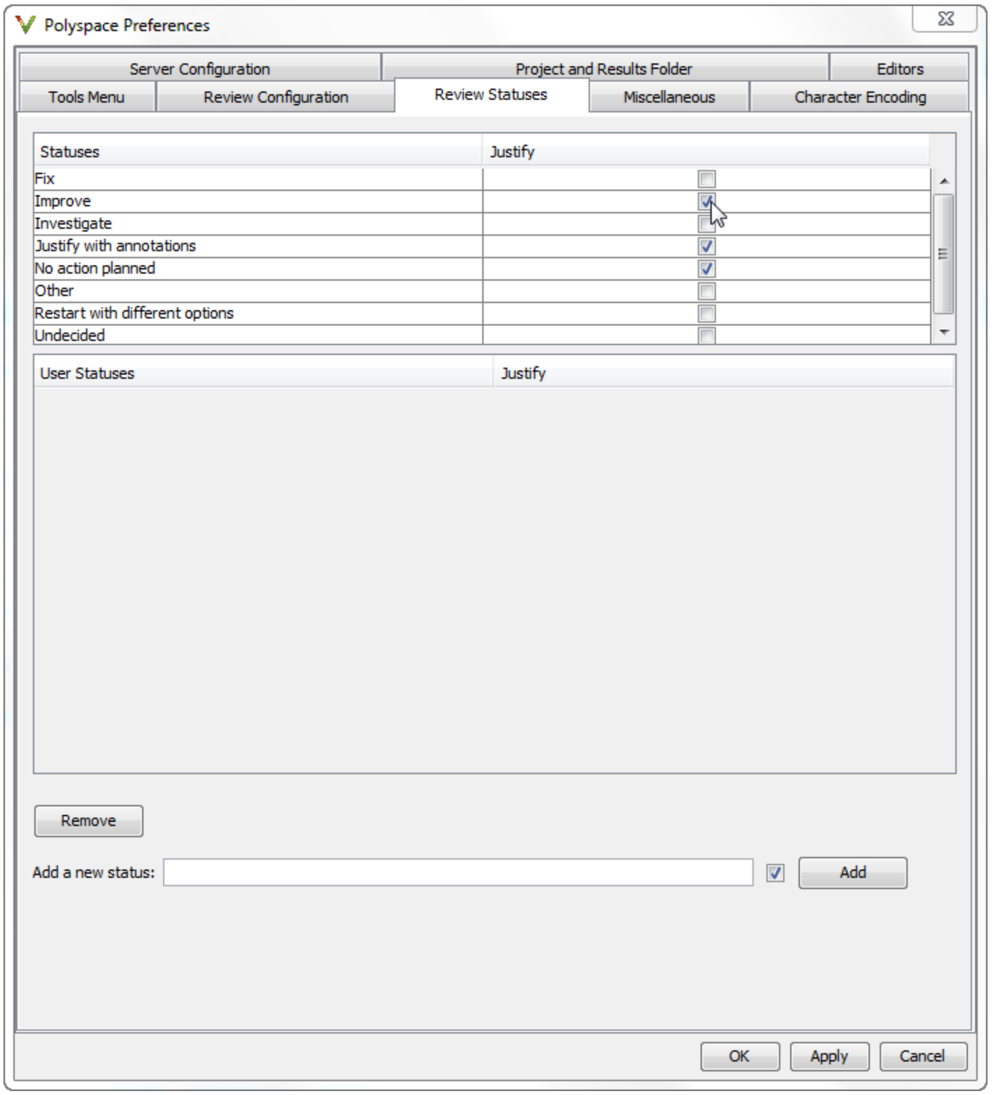
When reviewing checks, you can select the new status from the **Check Review > Status** drop-down list.

Add Justification to Existing Status

By default, a check is automatically justified if you assign the status, `Justify with annotations` or `No action planned`. However, you can change this default setting so that a check is justified when you assign one of the other existing statuses.

To add justification to existing status `Improve`:

1. Select **Options > Preferences**.
2. Select the **Review Statuses** tab. For the `Improve` status, select the check box in the **Justify** column. Click **OK**.



Statuses	Justify
Fix	☐
Improve	✓
Investigate	☐
Justify with annotations	✓
No action planned	✓
Other	☐
Restart with different options	☐
Undecided	☐

User Statuses	Justify

Add a new status: ✓

If you assign the `Improve` status to a check on the **Check Review** pane, the check gets automatically justified.

Compilation Errors

During a Polyspace Bug Finder analysis, the software first compiles the project and looks for coding rule errors. If the files have compilation errors, a message appears in the Output Summary pane and the offending files are ignored during the later analysis stages.

Consequently, Bug Finder produces results for all source files that do not have compilation errors. Files with compilation problems do not appear in the results.

For complete analysis results, fix compilation errors before rerunning the analysis.

Model Synchronous Tasks

In some circumstances, you must adapt your source code to allow synchronous tasks to be taken into account.

Suppose that an application has the following behavior:

- Once every 10 ms: `void tsk_10ms(void);`
- Once every 30 ms: `...`
- Once every 50 ms

These tasks do not interrupt each other, do not include infinite loops, and always return control to the calling context. For example:

```
void tsk_10ms(void)
{ do_things_and_exit();
 /* it's important it returns control*/
}
```

However, if you specify each entry-point at launch using the option:

```
polyspace-bug-finder-no-desktop -entry-points
tsk_10ms,tsk_30ms,tsk_50ms
```

then the results are not valid, because each task is called only once.

To address this problem, you must specify that the tasks are purely sequential. You can do this by writing a function to call each of the tasks in the right sequence, and then declaring this new function as a single task entry point.

Solution 1

Write a function that calls the cyclic tasks in the right order; an **exact sequencer**. This sequencer is then specified at launch time as a single task entry point.

This solution requires knowledge of the exact sequence of events.

For example, the sequencer might be:

```
void one_sequential_C_function(void)
{
 while (1) {
  tsk_10ms();
  tsk_10ms();
  tsk_10ms();
  tsk_30ms ();
  tsk_10ms();
  tsk_10ms();
  tsk_50ms ();
 }
}
```

and the associated launching command:

```
polyspace-bug-finder-no-desktop -entry-points
one_sequential_C_function
```

Solution 2

Make an **upper approximation sequencer**, taking into account every possible scheduling.

This solution is less precise but quick to code, especially for complicated scheduling:

For example, the sequencer might be:

```
void upper_approx_C_sequencer(void)
{
 volatile int random;
 while (1) {
  if (random) tsk_10ms();
  if (random) tsk_30ms();
  if (random) tsk_50ms();
  if (random) tsk_100ms();
  .....
 }
}
```

and the associated launching command:

```
polyspace-bug-finder-no-desktop -entry-points
upper_approx_C_sequencer
```

Note If this is the only entry-point, then it can be added at the end of the main procedure rather than specified as a task entry point.

Prepare Multitasking Code

In this section...

"Model Interruptions and Asynchronous Events and Tasks" on page 1-37

"Are Interruptions Maskable or Preemptive?" on page 1-39

"Model Shared Variables" on page 1-41

"Model Mailbox Messaging" on page 1-45

"Atomicity and Interrupted Instructions" on page 1-48

Model Interruptions and Asynchronous Events and Tasks

You can adapt your source code to allow Polyspace software to consider both *asynchronous* tasks and *interruptions*. For example:

```
void interrupt isr_1(void)
{ ... }
```

Without such an adaptation, interrupt service routines appear as dead code in the results. Dead code indicates that this code is not executed and is not taken into account, so interruptions and tasks are ignored.

The standard execution model is such that the main procedure is executed initially. Only if the main procedure terminates and returns control (i.e. if it is not an infinite loop and does not have red errors) do the entry points start, with the potential starting sequences being modelled automatically. You can adopt several different approaches to implement the required adaptations.

Solution 1: Where Interrupts (ISRs) Cannot Preempt Each Other

If the following conditions are fulfilled:

- The interrupt functions it_1 and it_2 do not interrupt each other.
- Each interrupt can be raised several times, at any time.
- The functions are returning functions, and not infinite loops.

Then these non preemptive interruptions may be grouped into a single function, and that function declared as an entry point.

```
void it_1(void);
void it_2(void);

void interruptions_and_events(void)
{ while (1) {
 if (random()) it_1();
 if (random()) it_2();
 ... }
}
```

The associated launching command would be:

```
polyspace-bug-finder-no-desktop -entry-points
interruptions_and_events
```

Solution 2: Where Interrupts Can Preempt Each Other

If two ISRs can each be interrupted by the other, then:

- Encapsulate each of them in a loop.
- Declare each loop as an entry point.

One approach is to replace the original file with a Polyspace version.

`original_file.c`

```
void it_1(void)
{
 ... return;
}

void it_2(void)
{
 ... return;
}

void one_task(void)
{
 ... return;
}
```

polyspace.c

```
void polys_it_1(void)
{
 while (1)
if (random())
 it_1();
}

 void polys_it_2(void)
{
 while (1)
  if (random())
   it_2();
}

void polys_one_task(void)
{
 while (1)
  if (random())
   one_task();
}
```

The associated launching command would be:

```
polyspace-bug-finder-no-desktop -entry-points
polys_it_1,polys_it_2,polys_one_task
```

Are Interruptions Maskable or Preemptive?

For user interruptions, an *implicit* critical section is not defined: you must write them manually.

Sometimes, an application which includes interrupts has a critical section written into its main entry point, but shared data is still flagged as unprotected.

This occurs because Polyspace does not distinguish between interrupt service routines and tasks. If you specify an interrupt to be a `-entry-points` entry point, it has the same priority level as the other procedures declared as tasks ("-entry-points" option). Because Polyspace makes an *upper approximation* of scheduling and interleaving, in this case, that includes the possibility that the ISR might be interrupted by any other task. More paths are modelled than could happen during execution, but this only means that more scenarios are considered than could happen during "real life" execution - and the shared data is not seen as being protected.

To address this, the interrupt must be embedded in a specific procedure that uses the same critical section as the interrupt used in the main task. Then, each time this function is called, the task will enter a critical section which will model the behavior of a nonmaskable interruption.

Original files:

```
int shared_x ;

void my_main_task(void)
{
 // ...
 MASK_IT;
 shared_x = 12;
 UMASK_IT;
 // ...
}
int shared_x ;

void interrupt my_real_it(void)
{ /* which is by specification unmaskable */
 shared_x = 100;
}
```

Additional C files required by the analysis:

```
extern void my_real_it(void);  // declaration required

#define MASK_IT pst_mask_it()
#define UMASK_IT pst_unmask_it()
```

```
void pst_mask_it(void);    // functions to model critical sections
void pst_unmask_it(void);  //


void other_task (void)
{
  MASK_IT;
  my_real_it();
  UMASK_IT;
}
```

The associated launch command:

```
polyspace-bug-finder-no-desktop \
 -D interrupt= \
 -entry-points my_main_task,other_task \
 -critical-section-begin "pst_mask_it:table" \
 -critical-section-end "pst_unmask_it:table"
```

Model Shared Variables

When you launch Polyspace without options, tasks are examined as though concurrent and without assumptions about priorities, sequence order, or timing. Shared variables in this context are considered unprotected.

The software uses the following explicit protection mechanisms to protect the variables:

- "Critical Sections" on page 1-41
- "Mutual Exclusion" on page 1-43
- "Semaphores" on page 1-44
- "Effects of Imprecision on Shared Variable List" on page 1-44

Critical Sections

This is the most common protection mechanism found in applications, and is simple to represent in Polyspace software:

- If one entry-point makes a call to a particular critical section, the other entry-points are blocked on the "critical-section-begin" function call until the originating entry-point calls the "critical-section-end" function.
- The code between two critical sections is not atomic.
- The code is a binary semaphore, so there is only one token per label (CS1 in the following example). Unlike many implementations of semaphores, it is not a decrementing counter that can keep track of a number of attempted accesses.

Consider the following example:

Original Code

```
void proc1(void)
{
 MASK_IT;
 x = 12; // X is protected
 y = 100;
 UMASK_IT;
}
void proc2(void)
{
 MASK_IT;
 x = 11; // X is protected
 UMASK_IT;
 y = 101; // Y is not protected
}
```

File Replacing the Original Include File

```
void begin_cs(void);
void end_cs(void);
#define MASK_IT begin_cs()
#define UMASK_IT end_cs()
```

Command-Line to Launch Polyspace Analysis

```
polyspace-bug-finder-no-desktop \
 -entry-point proc1,proc2 \
 -critical-section-begin"begin_cs:label_1" \
 -critical-section-end"end_cs:label_1"
```

Mutual Exclusion

You can implement mutual exclusion between tasks or interrupts while preparing to launch analysis.

Suppose there are entry-points which do not overlap each other, and that variables are shared by nature.

If entry-points are mutually exclusive, i.e. if they do not overlap in time, you may want the analysis to take that into account. Consider the following example:

These entry points cannot overlap:

- t1 and t3
- t2, t3 and t4

These entry-points can overlap:

- t1 and t2
- t1 and t4

Before launching, the names of mutually exclusive entry-points are placed on a single line:

```
polyspace-bug-finder-no-desktop -temporal-exclusion-file myExclusions.txt
-entry-points t1,t2,t3,t4
```

The file myExclusions.txt is also required in the current folder. This file contains:

```
t1 t3
t2 t3 t4
```

Semaphores

Although you can implement the code in C, Polyspace cannot take into account a semaphore system call. However, you can use critical sections to model the behavior of semaphores.

Effects of Imprecision on Shared Variable List

The list of shared variables that Polyspace identifies might contain more than the exact number of shared variables.

Note At a minimum, the list of shared variables contains the global variables or the exact number of shared variables.

Consider the following example.

```
// -entry-points IT_1, IT_2
int C[1];
int D[1];
extern int random(void);
void alias(int* par)
{
  int var;
  var=*par;
}

void IT_1(void)
{
while (1)
  {
     if (random())
     {
        D[0]=C[0];
        alias(D);
     }
  }
}

void IT_2(void)
```

```
{
while (1)
  {
     if (random())
     {
        C[0]=C[0]+1;
        alias(C);
     }
  }
}

void main(void)
{
   C[0]=0;
   D[0]=0;
}
```

The variable `D` is not a shared variable. However, because of array imprecision, Polyspace considers `D` a shared variable.

Model Mailbox Messaging

Suppose that an application has several tasks, some of which post messages in a mailbox while other tasks read the messages asynchronously.

This communication mechanism is possible because the OS libraries provide send and receive procedures. The source files will be unavailable because the procedures are part of the OS libraries, but the mechanism needs to be modelled for meaningful analysis.

By default, the analysis automatically stubs the missing OS send and receive procedures. The stub exhibits the following behavior:

- For `send(char *buffer, int length)`, the content of the buffer is written only when the procedure is called.
- For `receive(char *buffer, int *length)`, each element of the buffer will contain the full range of values for the corresponding data type.

You can use this mechanism and other mechanisms, with different levels of precision.

Let Polyspace software stub automatically	• Quick and easy to code. • **imprecise** because there is not a direct connection between a mailbox sender and receiver. It means that even if the sender is only submitting data within a small range, the full data range for the type(s) will be used for the receiver data
Provide a **real mailbox** mechanism	• Costly (time consuming) to implement. • Can introduce errors in the stubs. • Provides little additional benefit when compared to the upper approximation solution below.
Provide an **upper approximation of the mailbox**	Models the mechanism so that new read from the mailbox reads **one** of the recently posted messages, but not necessarily the last message. • Quick and easy to code. • **gives precise results**

Consider the following detailed implementation of the upper approximation solution:

polyspace_mailboxes.h

```
typedef struct _r {
 int length;
 char content[100];
} MESSAGE;
extern MESSAGE mailbox;
void send(MESSAGE * msg);
void receive(MESSAGE *msg);
```

polyspace_mailboxes.c

```
#include "polyspace_mailboxes.h"

MESSAGE mailbox;

void send(MESSAGE * msg)
{
 volatile int test;
 if (test) mailbox = *msg;
 // a potential write to the mailbox
}

void receive(MESSAGE *msg)
{
 *msg = mailbox;
}
```

Original code

```
#include "polyspace_mailboxes.h"

void t1(void)
{
 MESSAGE msg_to_send;
 int i;
 for (i=0; i<100; i++)
  msg_to_send.content[i] = i;
 msg_to_send.length = 100;
 send(&msg_to_send);
}
void t2(void)
{
 MESSAGE msg_to_read;
  receive (&msg_to_read);
}
```

The analysis then proceeds on the assumption that each new read from the mailbox reads a message, but not necessarily the last message.

The associated launching command is:

```
polyspace-bug-finder-no-desktop -entry-points t1,t2
```

Atomicity and Interrupted Instructions

Atomic: In computer programming, atomic describes a unitary action or object that is essentially indivisible, unchangeable, whole, and irreducible.

Atomicity: In a transaction involving two or more discrete pieces of information, either all of the pieces are committed or none are.

Instructional decomposition

Polyspace does not take into account either CPU instruction decomposition or timing considerations.

Polyspace assumes that instructions are not atomic except in the case of read and write instructions. Polyspace makes an **upper approximation of scheduling and interleaving**. There are more paths modelled than could be implemented during execution, but since Polyspace **analyzes every possible path**, the results are not negatively affected.

Consider a 16-bit target that can manipulate a 32-bit type (an `int`, for example). In this case, the CPU needs at least two cycles to write to an integer.

Suppose that x is an integer in a multitasking system, with an initial value of 0x0000. Now suppose 0xFF55 is written it. If the operation is not atomic it could be interrupted by another instruction in the middle of the write operation.

- Task 1: Writes 0xFF55 to x.
- Task 2: Interrupts task 1. Depending on the timing, the value of x could be either 0xFF00, 0x0055 or 0xFF55.

Polyspace considers write/read instructions atomic, so **task 2 can only read 0xFF55**, even if X is not protected (see “Model Shared Variables” on page 1-41).

Critical sections

In terms of critical sections, Polyspace does not model the concept of atomicity. A critical section guarantees only that once the function associated with `-critical-section-begin` is called, other functions making use of the same label are blocked. The other functions can still continue to run, even if somewhere else in another task a critical section has been started.

Polyspace of run-time errors supposes that there is not a conflict when writing the shared variables. Therefore, even if a shared variable is not protected, the analysis is complete and correct.

Priorities

Priorities are not taken into account by Polyspace. However, the timing implications of software execution are not relevant to the analysis, which is the primary reason for implementing software task prioritization. In addition, priority inversion issues can mean that the software cannot assume that priorities can protect shared variables. For that reason, Polyspace software does not makes such assumption.

Although Polyspace does not have the capability to specify differing task priorities, priorities **are** taken into account because the default behavior of the software assumes that:

- Task entry points (as defined with the option `-entry-points`) start potentially at the same time;
- The task entry points can interrupt each other in any order, in spite of the sequence of instructions. Therefore, every possible interruption is accounted for, in addition to some interruptions which do not actually occur.

If you have two tasks, t1 and t2, in which t1 has higher priority than t2, use `polyspace-bug-finder-no-desktop -entry-points t1,t2`.

- t1 interrupts t2 at any stage of t2, which models the behavior at execution time.
- t2 interrupts t1 at any stage of t1, which models a behavior which (ignoring priority inversion) would take place during execution. Polyspace has made an **upper approximation of scheduling and interleaving**. There are more paths modelled than would happen in “real life”, so results are not negatively affected.

Annotate Code for Known Defects

How to Add Annotations

You can place comments in your code that inform Polyspace software of known or acceptable bugs and coding rule violations. Through the use of these comments, you can:

- Identify defects from previous analyses.
- Categorize reviewed defects.
- Highlight defects that are not significant.

During your analysis of results, you can disregard these known errors and focus on new errors.

Annotate your code before running an analysis:

1. Open your source file using a text editor.
2. Locate the code that produces a run-time error.
3. Insert the required comment. See "Syntax for Annotations" on page 1-51.
4. Save your file.
5. Start the analysis. If your comments do not conform to the prescribed syntax, the software produces a warning and the comments do not appear in the Results Summary.

When the analysis is complete, open the results.

In the **Classification**, **Status**, and **Comment** columns, the information that you provide within your code comments is now visible.

Syntax for Annotations

Polyspace applies the comments, which are case-insensitive, to the first non-commented line of C code that follows the annotation.

Note Instead of typing the full syntax of the annotation, you can copy an annotation template from the results. See "Copy and Paste Annotations" on page 1-58 for more information.

To apply comments to a single line of code, use the following syntax:

```
/* polyspace<Defect:Kind1[,Kind2] : [Classification] : [Status] >
[Additional comments] */
```

To apply comments to a section of code, use the following syntax:

```
/* polyspace:begin<Defect:Kind1[,Kind1] :
[Classification] : [Status] >
[Additional text] */

... Code section ...

/* polyspace:end<Defect:Kind1[,Kind1] : [Classification] : [Status] >  */
```

Square brackets *[ ]* indicate optional information.

Replace	Replace with
Kind1,Kind2,...	Specific defect abbreviations such as `MEM_LEAK`, `FREED_PTR`, etc. If you want the comment to apply to all defects on the following line, specify `ALL`.
Classification	• `Unset` • `High` • `Medium` • `Low` • `Not a defect`

Replace	Replace with
`Status`	Action for correcting the defect in your code. Possible values are: • `Fix` • `Improve` • `Investigate` • `Justify with annotations` • `No action planned` • `Restart with different options` • `Other` • `Undecided`
`Additional text`	Additional comments.

Syntax Examples:

Defect:

```
polyspace<Defect:USELESS_WRITE : Low : No Action Planned > Known issue
```

Annotate Code for Rule Violations

How to Add Annotations

You can place comments in your code that inform Polyspace software of known or acceptable bugs and coding rule violations. Through the use of these comments, you can:

- Identify results from previous analyses.
- Categorize reviewed results.
- Highlight rule violations that are not significant.

Note Source code annotations do not apply to code comments. Therefore, the following coding rules cannot be annotated:

- MISRA-C Rules 2.2 and 2.3
- MISRA-C++ Rule 2-7-1
- JSF++ Rules 127 and 133

During your analysis of results, you can disregard these known errors and focus on new errors.

Annotate your code before running an analysis:

1. Open your source file using a text editor.
2. Locate the code that produces a run-time error.
3. Insert the required comment. See “Syntax for Annotations” on page 1-51.
4. Save your file.
5. Start the analysis. If your comments do not conform to the prescribed syntax, the software produces a warning and the comments do not appear in the Results Summary.

When the analysis is complete, open the results.

In the **Classification**, **Status**, and **Comment** columns, the information that you provide within your code comments is now visible.

Syntax for Annotations

Polyspace applies the comments, which are case-insensitive, to the first non-commented line of C code that follows the annotation.

Note Instead of typing the full syntax of the annotation, you can copy an annotation template from the results. See "Copy and Paste Annotations" on page 1-58 for more information.

To apply comments to a single line of code, use the following syntax:

```
/* polyspace<Rule_set:Rule1[,Rule2] : [Classification] : [Status] >
[Additional comments] */
```

To apply comments to a section of code, use the following syntax:

```
/* polyspace:begin<Rule_Set:Rule1[,Rule2] :
[Classification] : [Status] >
[Additional text] */

... Code section ...

/* polyspace:end<Rule_Set:Rule1[,Rule2] : [Classification] : [Status] >  */
```

Square brackets *[ ]* indicate optional information.

Replace	Replace with
`Rule_Set`	• `MISRA-C` • `MISRA-AC-AGC` • `MISRA-CPP` • `JSF` • `Custom` If you want the comment to apply to all coding rule violations on the following line, specify `ALL`.
`Rule1,Rule2,...`	Rule number. For more information, see: • "MISRA C:2004 Coding Rules" • "MISRA® C++ Coding Rules" • "JSF® C++ Coding Rules" • "Custom Naming Convention Rules"
`Classification`	• `Unset` • `High` • `Medium` • `Low` • `Not a defect`
`Status`	Action for correcting the coding rule violation. Possible values are: • `Fix` • `Improve` • `Investigate` • `Justify with annotations` • `No action planned` • `Restart with different options` • `Other`

Replace	Replace with
	• `Undecided`
`Additional text`	Additional comments.

Syntax Examples:

MISRA C rule violation:

```
polyspace<MISRA-C:6.3 : Low : Justify with annotations> Known issue
```

JSF rule violation:

```
polyspace<JSF:9 : Low : Justify with annotations> Known issue
```

Copy and Paste Annotations

Instead of typing the full syntax of an annotation comment in your source code, you can copy an annotation template, paste it into your source code, and modify the template to comment the check.

To copy the justification template to the clipboard:

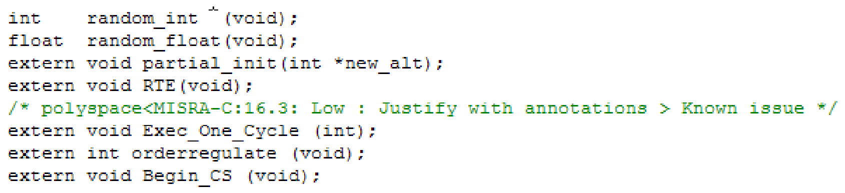
1 In the **Results Summary** pane, right–click a coding rule violation.

2 From the context menu, select **Add Pre-Justification to Clipboard**. The software copies the justification string to the clipboard.

3 Open the source file containing the violations you want to justify.

4 Navigate to the code you want to comment, and paste the justification template string on the line immediately before the line you want to comment.

5 Modify the template text to comment the code appropriately.



```
int    random_int (void);
float  random_float(void);
extern void partial_init(int *new_alt);
extern void RTE(void);
/* polyspace<MISRA-C:16.3: Low : Justify with annotations > Known issue */
extern void Exec_One_Cycle (int);
extern int orderregulate (void);
extern void Begin_CS (void);
```

6 Save the file.

Predefined Target Processor Specifications

Polyspace software supports many commonly used processors, as listed in the table below. To specify one of the predefined processors, select it from the **Target processor type** drop-down list.

Predefined Target Processor Specifications

Target	char	short	int	long	long long	float	double	long double	ptr	sign of char	endian	align
`i386`	8	16	32	32	64	32	64	96	32	signed	Little	32
`sparc`	8	16	32	32	64	32	64	128	32	signed	Big	64
`m68k / ColdFire`[1]	8	16	32	32	64	32	64	96	32	signed	Big	64
`powerpc`	8	16	32	32	64	32	64	128	32	unsigned	Big	64
`c-167`	8	16	16	32	32	32	64	64	16	signed	Little	64
`tms320c3x`	32	32	32	32	64	32	32	40[2]	32	signed	Little	32
`sharc21x61`	32	32	32	32	64	32	32 [64]	32 [64]	32	signed	Little	32
`NEC-V850`	8	16	32	32	32	32	32	64	32	signed	Little	32 [16, 8]
`hc08`[3]	8	16	16 [32]	32	32	32	32 [64]	32 [64]	16[4]	unsigned	Big	32 [16]
`hc12`[5]	8	16	16 [32]	32	32	32	32 [64]	32 [64]	32[6]	signed	Big	32 [16]
`mpc5xx`[5]	8	16	32	32	64	32	32 [64]	32 [64]	32	signed	Big	32 [16]
`c18`	8	16	16	32 [24][5]	32	32	32	32	16 [24]	signed	Little	8

1. The M68k family (68000, 68020, etc.) includes the "ColdFire" processor
2. Operations on long double values will be imprecise.
3. Non ANSI C specified keywords and compiler implementation-dependent pragmas and interrupt facilities are not taken into account by this support
4. All kinds of pointers (near or far pointer) have 2 bytes (hc08) or 4 bytes (hc12) of width physically.
5. The `c18` target supports the type `short long` as 24-bits.

Predefined Target Processor Specifications (Continued)

Target	char	short	int	long	long long	float	double	long double	ptr	sign of char	endian	align
`x86_64`	8	16	32	64 [32][6]	64	32	64	96	64	signed	Little	64 [32]
`mcpu` (Advanced)	8 [16]	8 [16]	16 [32]	32	32 [64]	32	32 [64]	32 [64]	16 [32]	signed	Little	32 [16, 8]

Note The following target processors are supported only for C code analyses: `tms320c3x`, `sharc21x61`, `NEC-V850`, `hc08`, `hc12`, `mpc5xx`, and `c18`.

After selecting a predefined target, you can modify some default attributes by selecting the browse button to the right of the **Target processor type** drop-down menu. The optional settings for each target are shown in [brackets] in the table.

If your processor is not listed, you can specify a similar processor that shares the same characteristics, or create a generic target processor.

Note If your target processor does not match the characteristics of a processor described above, contact MathWorks® technical support for advice.

---

6. Use option `-long-is-32bits` to support Microsoft C/C++ Win64 target

Modify Predefined Target Processor Attributes

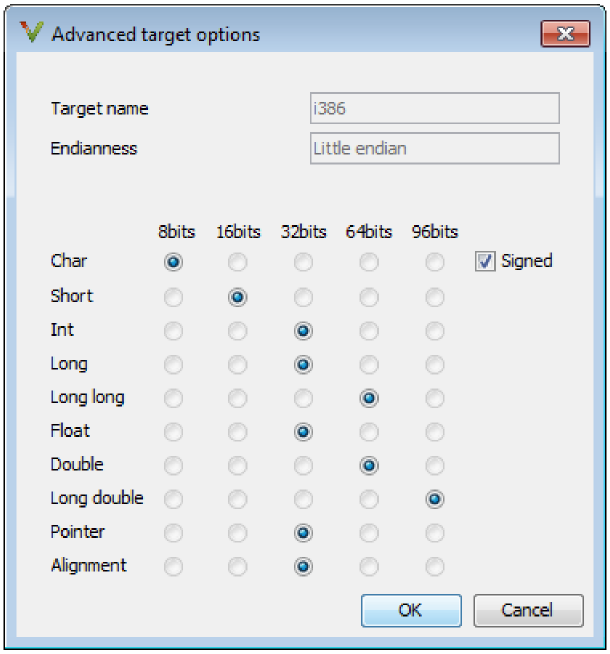
You can modify certain attributes of the predefined target processors. If your specific processor is not listed, you may be able to specify a similar processor and modify its characteristics to match your processor.

Note The settings that you can modify for each target are shown in [brackets] in the “Predefined Target Processor Specifications” on page 1-59 table.

To modify target processor attributes:

1. In the Project Manager perspective, select the **Configuration > Target & Compiler** pane.
2. From the **Target processor type** drop-down list, select the target processor that you want to use.
3. To the right of the **Target processor type** field, click **Edit**.

   The Advanced target options dialog box opens.



Target name: i386

Endianness: Little endian

	8bits	16bits	32bits	64bits	96bits	
Char	●	○	○	○	○	[✓] Signed
Short	○	●	○	○	○	
Int	○	○	●	○	○	
Long	○	○	●	○	○	
Long long	○	○	○	●	○	
Float	○	○	●	○	○	
Double	○	○	○	●	○	
Long double	○	○	○	○	●	
Pointer	○	○	●	○	○	
Alignment	○	○	●	○	○	

4 Modify the attributes as required.

For information on each target option, see “Generic target options”.

5 Click **OK** to save your changes.

Specify Generic Target Processors

Define Generic Target

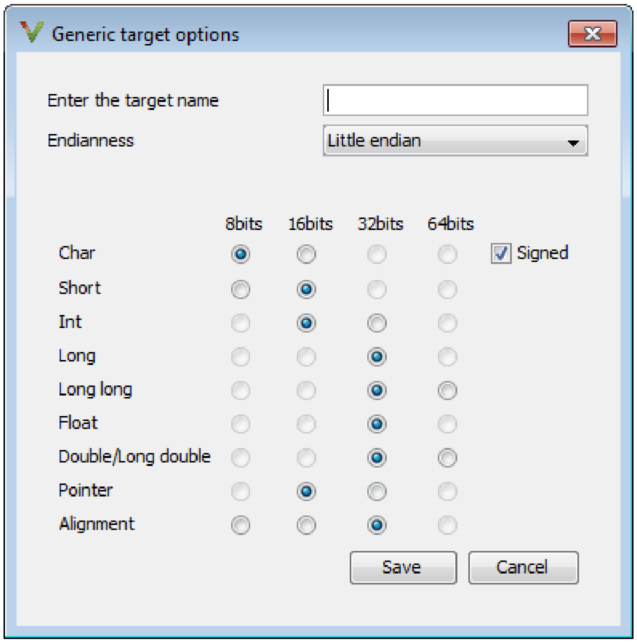
If your application is designed for a custom target processor, you can configure many basic characteristics of the target by selecting the selecting the `mcpu... (Advanced)` target, and specifying the characteristics of your processor.

To configure a generic target:

1. In the Project Manager perspective, select the **Configuration > Target & Compiler** pane.

2. From the **Target processor type** drop-down list, select `mcpu... (Advanced)`.

   The Generic target options dialog box opens.



Enter the target name

Endianness: Little endian

	8bits	16bits	32bits	64bits	
Char	●	○	○	○	☑ Signed
Short	○	●	○	○	
Int	○	●	○	○	
Long	○	○	●	○	
Long long	○	○	●	○	
Float	○	○	●	○	
Double/Long double	○	○	●	○	
Pointer	○	●	○	○	
Alignment	○	○	●	○	

3. In the **Enter the target name** field, enter a name, for example, `MyTarget`.

4 Specify the parameters for your target, such as the size of basic types, and alignment with arrays and structures.

For example, when the alignment of basic types within an array or structure is always 8, it implies that the storage assigned to arrays and structures is strictly determined by the size of the individual data objects (without fields and end padding).

Note For information on each target option, see "Generic target options".

5 Click **Save** to save the generic target options and close the dialog box.

Common Generic Targets

The following tables describe the characteristics of common generic targets.

ST7 (Hiware C compiler : HiCross for ST7)

ST7	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16	32	32	32	32	32	16/32	unsigned	Big
alignment	8	16/8	16/8	32/16/8	32/16/8	32/16/8	32/16/8	32/16/8	32/16/8	N/A	N/A

ST9 (GNU C compiler : gcc9 for ST9)

ST9	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16	32	32	32	64	64	16/64	unsigned	Big
alignment	8	8	8	8	8	8	8	8	8	N/A	N/A

Hitachi H8/300, H8/300L

Hitachi H8/300, H8/300L	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16/32	32	64	32	654	64	16	unsigned	Big
alignment	8	16	16	16	16	16	16	16	16	N/A	N/A

Hitachi H8/300H, H8S, H8C, H8/Tiny

Hitachi H8/300H, H8S, H8C, H8/Tiny	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16/ 32	32	64	32	64	64	32	unsigned	Big
alignment	8	16	32/ 16	32/16	32/16	32/16	32/16	32/16	32/16	N/A	N/A

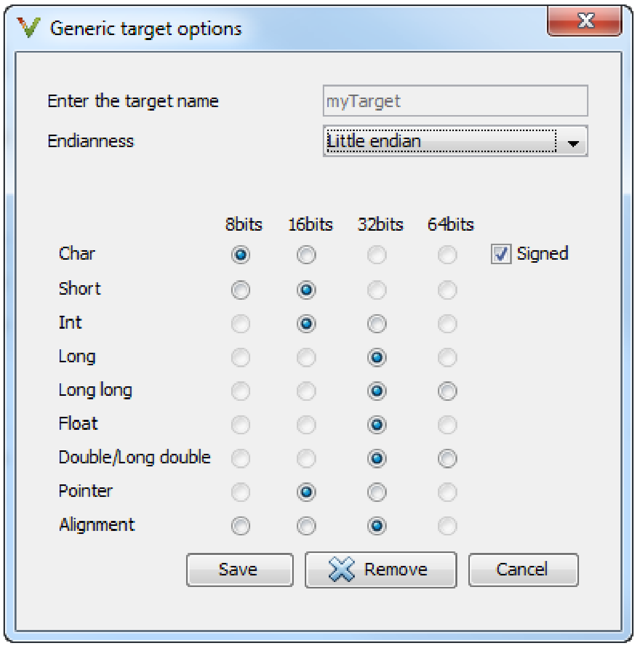
View or Modify Existing Generic Targets

To view or modify generic targets that you previously created:

1 In the Project Manager perspective, select the **Configuration > Target & Compiler** pane.

2 From the **Target processor type** drop-down list, select your target, for example, `myTarget`.

3 Click **Edit**. The Generic target options dialog box opens, displaying your target attributes.



Enter the target name: myTarget

Endianness: Little endian

	8bits	16bits	32bits	64bits	
Char	●	○	○	○	✓ Signed
Short	○	●	○	○	
Int	○	●	○	○	
Long	○	○	●	○	
Long long	○	○	●	○	
Float	○	○	●	○	
Double/Long double	○	○	●	○	
Pointer	○	●	○	○	
Alignment	○	○	●	○	

4 If required, specify new attributes for your target. Then click **Save**.

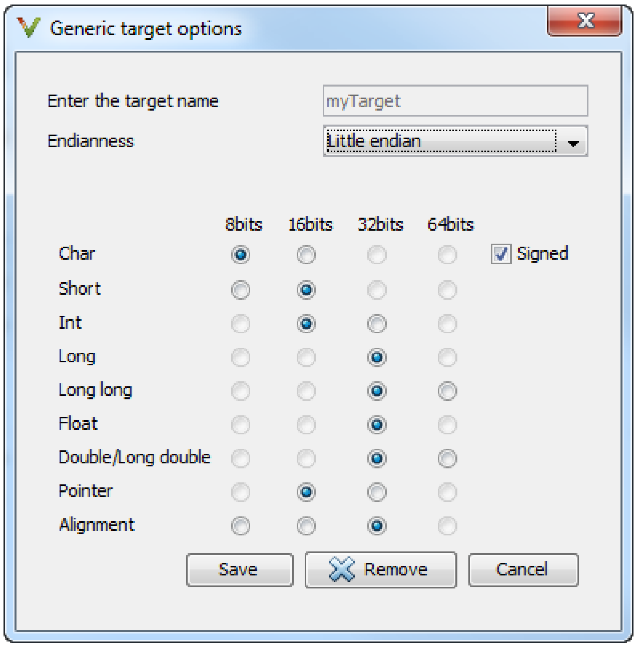
5 Otherwise, click **Cancel**.

Delete Generic Target

To delete a generic target:

1 In the Project Manager perspective, select the **Configuration > Target & Compiler** pane.

2 From the **Target processor type** drop-down list, select the target that you want to remove, for example, `myTarget`.



Enter the target name: myTarget

Endianness: Little endian

	8bits	16bits	32bits	64bits	
Char	●	○	○	○	✓ Signed
Short	○	●	○	○	
Int	○	●	○	○	
Long	○	○	●	○	
Long long	○	○	●	○	
Float	○	○	●	○	
Double/Long double	○	○	●	○	
Pointer	○	●	○	○	
Alignment	○	○	●	○	

3 Click **Remove**. The software removes the target from the list.

Compile Operating System-Dependent Code

This section describes the options required to compile and analyze code designed to run on specific operating systems. It contains the following:

In this section...

"Predefined Compilation Flags for C Code" on page 1-69

"Predefined Compilation Flags for C++ Code" on page 1-71

"My Target Application Runs on Linux" on page 1-73

"My Target Application Runs on Solaris" on page 1-74

"My Target Application Runs on Vxworks" on page 1-74

"My Target Application Does Not Run on Linux, vxworks nor Solaris" on page 1-75

Predefined Compilation Flags for C Code

These flags concern the predefined **OS-target** options: `no-predefined-OS`, `linux`, `vxworks`, `Solaris` and `visual` (`-OS-target` option).

OS-target	Compilation flags	`-include` file and content
no predefined OS	`-D__STDC__`	
visual	`-D__STDC__`	`-include` `<`*`product_dir`*`>/cinclude/pst-visual.h`

vxworks	`-D__STDC__`
	`-DANSI_PROTOTYPES`
	`-DSTATIC=`
	`-DCONST=const`
	`-D__GNUC__=2`
	`-Dunix`
	`-D__unix`
	`-D__unix__`
	`-Dsparc`
	`-D__sparc`
	`-D__sparc__`
	`-Dsun`
		`-include` `<`*`product_dir`*`>/cinclude/pst-vxworks.h`

OS-target	Compilation flags	`-include` file and content

	`-D__sun`
	`-D__sun__`
	`-D__svr4__`
	`-D__SVR4`	

linux	`-D__STDC__`
	`-D__GNUC__=2`
	`-D__GNUC_MINOR__=6`
	`-D__GNUC__=2`
	`-D__GNUC_MINOR__=6`
	`-D__ELF__`
	`-Dunix`
	`-D__unix`
	`-D__unix__`
	`-Dlinux`
	`-D__linux`
	`-D__linux__`
	`-Di386`
	`-D__i386`
	`-D__i386__`
	`-Di686`
	`-D__i686`
	`-D__i686__`
	`-Dpentiumpro`
	`-D__pentiumpro`
	`-D__pentiumpro__`
		<product_dir>`/cinclude/pst-linux.h`

Solaris	`-D__STDC__`
	`-D__GNUC__=2`
	`-D__GNUC_MINOR__=8`
	`-D__GNUC__=2`
	`-D__GNUC_MINOR__=8`
	`-D__GCC_NEW_VARARGS__`
	`-Dunix`
	`-D__unix`
	`-D__unix__`
	`-Dsparc`
	`-D__sparc`
	`-D__sparc__`
		No `-include` file mentioned

OS-target	Compilation flags	`-include` file and content

	`-Dsun`
	`-D__sun`
	`-D__sun__`
	`-D__svr4__`
	`-D__SVR4`	

Note The use of the OS-target option is entirely equivalent to the following alternative approaches.

- Setting the same -D flags manually, or
- Using the `-include` option on a copied and modified pst-*OS-target*.h file

Predefined Compilation Flags for C++ Code

The following table shown for each `OS-target`, the list of compilation flags defined by default, including pre-include header file (see also `-include`):

-OS-target	Compilation flags	-include file	Minimum set of options

Linux	`-D__SIZE_TYPE__=unsigned`
	`-D__PTRDIFF_TYPE__=int`
	`-D__inline__=inline`
	`-D__signed__=signed`
	`-D__gnuc_va_list=va_list`
	`-D__STL_CLASS_PARTIAL_`
	`SPECIALIZATION`
	`-D__GNU_SOURCE`
	`-D__STDC__ -D__ELF__`
	`-Dunix -D__unix`
	`-D__unix__ -Dlinux`
	`-D__linux -D__linux__`
	`-Di386 -D__i386`
	`-D__i386__ -Di686`
	`-D__i686 -D__i686__`
	`-Dpentiumpro`
		`<product_dir>/`
		`cinclude/`
		`pst-linux.h`
			`polyspace-[desktop-]cpp`
			`-OS-target Linux \`
			
			`-I <polyspace_install>/include/`
			`include-linux \`
			
			`-I <product_dir>/include/`
			`include-linux/next` Where the Polyspace product has been installed in the folder `<polyspace_install>`

-OS-target	Compilation flags	-include file	Minimum set of options
	-D__pentiumpro -D__pentiumpro__		
vxWorks	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D__STL_CLASS_PARTIAL_ SPECIALIZATION -DANSI_PROTOTYPES -DSTATIC= -DCONST=const -D__STDC -D__GNU_SOURCE -Dunix -D__unix -D__unux__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__ -D__svr4 -D__SVR4	<product_dir>/ cinclude/ pstvxworks. h	polyspace-[desktop-]cpp \ -OS-target vxworks \ -I /your_path_to/ Vxworks_include_folders
visual /visual6	-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__STRICT_ANSI__ -D__inline__=inline -D__signed__=signed -D__gnuc_va_list=va_list -D_POSIX_SOURCE -D__STL_CLASS_PARTIAL_ SPECIALIZATION	<product_dir>/ cinclude/ pstvisual. h	

-OS-target	Compilation flags	-include file	Minimum set of options

Solaris	`-D__SIZE_TYPE__=unsigned`
	`-D__PTRDIFF_TYPE__=int`
	`-D__inline__=inline`
	`-D__signed__=signed`
	`-D__gnuc_va_list=va_list`
	`-D__STL_CLASS_PARTIAL_ SPECIALIZATION`
	`-D__GNU_SOURCE`
	`-D__STDC`
	`-D__GCC_NEW_VARARGS__`
	`-Dunix`
	`-D__unix`
	`-D__unux__`
	`-Dsparc`
	`-D__sparc`
	`-D__sparc__`
	`-Dsun`
	`-D__sun`
	`-D__sun__`
	`-D__svr4`
	`-D__SVR4`
			If Polyspace runs on a Linux machine:
			`polyspace-bug-finder-no-desktop \`
			`-OS-target Solaris \`
			`-I /your_path_to_solaris_include`
			If Polyspace runs on a Solaris machine:
			`polyspace-bug-finder-no-desktop \`
			`-OS-target Solaris \`
			`-I /usr/include`

no-predefined-OS	`-D__SIZE_TYPE__=unsigned`
	`-D__PTRDIFF_TYPE__=int`
	`-D__STRICT_ANSI__`
	`-D__inline__=inline`
	`-D__signed__=signed`
	`-D__gnuc_va_list=va_list`
	`-D_POSIX_SOURCE`
	`-D__STL_CLASS_PARTIAL_ SPECIALIZATION`
			`polyspace-bug-finder-no-desktop \`
			`-OS-target no-predefined-OS \`
			`-I /your_path_to/ MyTarget_include_folders`

Note This list of compiler flags is written in every log file.

My Target Application Runs on Linux

The minimum set of options is as follows:

```
polyspace-bug-finder-no-desktop \
 -OS-target Linux \
 -I Polyspace_Install/polyspace/verifier/cxx/include/include-libc \

 ...
```

where the Polyspace product has been installed in the folder *Polyspace_Install*.

If your target application runs on Linux but you are launching your analysis from Windows, the minimum set of options is as follows:

```
polyspace-bug-finder-no-desktop \
 -OS-target Linux \
 -I Polyspace_Install\polyspace\verifier\cxx\include\include-libc \

 ...
```

where the Polyspace product has been installed in the folder *Polyspace_Install*.

My Target Application Runs on Solaris

If Polyspace software runs on a Linux machine:

```
polyspace-bug-finder-no-desktop \
 -OS-target Solaris \
 -I /your_path_to_solaris_include
```

If Polyspace software runs on a Solaris™ machine:

```
polyspace-bug-finder-no-desktop \
 -OS-target Solaris \
 -I /usr/include
```

My Target Application Runs on Vxworks

If Polyspace software runs on either a Solaris or a Linux machine:

```
polyspace-bug-finder-no-desktop \
 -OS-target vxworks \
 -I /your_path_to/Vxworks_include_folders
```

My Target Application Does Not Run on Linux, vxworks nor Solaris

If Polyspace software does not run on either a Solaris or a Linux machine:

```
polyspace-bug-finder-no-desktop \
 -OS-target no-predefined-OS \
 -I /your_path_to/MyTarget_include_folders
```

Address Alignment

Polyspace software handles address alignment by calculating `sizeof` and alignments. This approach takes into account 3 constraints implied by the ANSI standard which ensure that:

- that global `sizeof` and `offsetof` fields are optimum (i.e. as short as possible);
- the alignment of addressable units is respected;
- global alignment is respected.

Consider the example:

```
struct foo {char a; int b;}
```

- Each field must be aligned; that is, the starting offset of a field must be a multiple of its own size[7]
- So in the example, `char a` begins at offset 0 and its size is 8 bits. `int b` cannot begin at 8 (the end of the previous field) because the starting offset must be a multiple of its own size (32 bits). Consequently, `int b` begins at offset=32. The size of the `struct foo` before global alignment is therefore 64 bits.
- The global alignment of a structure is the maximum of the individual alignments of each of its fields;
- In the example, `global_alignment = max (alignment char a, alignment int b) = max (8, 32) = 32`
- The size of a struct must be a multiple of its global alignment. In our case, b begins at 32 and is 32 long, and the size of the struct (64) is a multiple of the `global_alignment` (32), so `sizeof` is not adjusted.

7. except in the cases of “double” and “long” on some targets.

Ignore or Replace Keywords Before Compilation

You can ignore noncompliant keywords, for example, `far` or `0x`, which precede an absolute address. The template `myTpl.pl` (listed below) allows you to ignore these keywords:

1. Save the listed template as `C:\Polyspace\myTpl.pl`.
2. Select the **Configuration > Target & Compiler > Environment Settings** pane.
3. To the right of the **Command/script to apply to preprocessed files** field, click on the file icon.
4. Use the Open File dialog box to navigate to `C:\Polyspace`.
5. In the **File name** field, enter `myTpl.pl`.
6. Click **Open**. You see `C:\Polyspace\myTpl.pl` in the **Command/script to apply to preprocessed files** field.

For more information, see .

Content of myTpl.pl file

```
#!/usr/bin/perl

##############################################################
# Post Processing template script
#
##############################################################
# Usage from Project Manager GUI:
#
# 1) Linux: /usr/bin/perl PostProcessingTemplate.pl
# 2) Windows: Polyspace_Install\sys\perl\win32\bin\perl.exe <pathtoscript>\
PostProcessingTemplate.pl
#
##############################################################

$version = 0.1;
```

```
$INFILE = STDIN;
$OUTFILE = STDOUT;

while (<$INFILE>)
{

 # Remove far keyword
 s/far//;

 # Remove "@ 0xFE1" address constructs
 s/\@\s0x[A-F0-9]*//g;

 # Remove "@0xFE1" address constructs
 # s/\@0x[A-F0-9]*//g;

 # Remove "@ ((unsigned)&LATD*8)+2" type constructs
 s/\@\s\(\(unsigned\)\&[A-Z0-9]+\*8\)\+\d//g;

 # Convert current line to lower case
# $_ =~ tr/A-Z/a-z/;

 # Print the current processed line
 print $OUTFILE $_;
}
```

Perl Regular Expression Summary

```
#########################################################
# Metacharacter What it matches
#########################################################
# Single Characters
# . Any character except newline
# [a-z0-9] Any single character in the set
# [^a-z0-9] Any character not in set
# \d A digit same as
# \D A non digit same as [^0-9]
# \w An Alphanumeric (word) character
# \W Non Alphanumeric (non-word) character
#
```

```
# Whitespace Characters
# \s Whitespace character
# \S Non-whitespace character
# \n newline
# \r return
# \t tab
# \f formfeed
# \b backspace
#
# Anchored Characters
# \B word boundary when no inside []
# \B non-word boundary
# ^ Matches to beginning of line
# $ Matches to end of line
#
# Repeated Characters
# x? 0 or 1 occurence of x
# x* 0 or more x's
# x+ 1 or more x's
# x{m,n} Matches at least m x's and no more than n x's
# abc Exactly "abc"
# to|be|great One of "to", "be" or "great"
#
# Remembered Characters
# (string) Used for back referencing see below
# \1 or $1 First set of parentheses
# \2 or $2 First second of parentheses
# \3 or $3 First third of parentheses
##########################################################
# Back referencing
#
# e.g. swap first two words around on a line
# red cat -> cat red
# s/(\w+) (\w+)/$2 $1/;
#
##########################################################
```

Analyze Keil or IAR Dialects

Typical embedded control applications frequently read and write port data, set timer registers and read input captures. To deal with this without using assembly language, some microprocessor compilers have specified special data types like `sfrand sbit`. Typical declarations are:

```
sfr A0 = 0x80;
sfr A1 = 0x81;
sfr ADCUP = 0xDE;
sbit EI = 0x80;
```

These declarations reside in header files such as `regxx.h` for the basic `80Cxxx` micro processor. The definition of `sfr` in these header files customizes the compiler to the target processor.

When accessing a register or a port, using `sfr` data is then simple, but is not part of standard ANSI C:

```
int status,P0;

void main (void) {
  ADCUP = 0x08; /* Write data to register */
  A1 = 0xFF; /* Write data to Port */
  status = P0; /* Read data from Port */
  EI = 1; /* Set a bit (enable interrupts) */
}
```

You can analyze this type of code using the **Dialect** (`-dialect`) option . This option allows the software to support the Keil or IAR C language extensions even if some structures, keywords, and syntax are not ANSI standard. The following tables summarize what is supported when analyzing code that is associated with the Keil or IAR dialects.

The following table summarizes the supported Keil C language extensions:

Example: **`-dialect keil -sfr-types sfr=8`**

Type/Language	Description	Example	Restrictions

Type `bit`	• An expression to type bit gives values in range [0,1].
	• Converting an expression in the type, gives 1 if it is not equal to 0, else 0. This behavior is similar to c++ `bool` type.
		`bit x = 0, y = 1,`
		` z = 2;`
		`assert(x == 0);`
		`assert(y == 1);`
		`assert(z == 1);`
		`assert(sizeof(bit)`
		` == sizeof(int));`
			pointers to bits and arrays of bits are not allowed

Type `sfr`	• The -sfr-types option defines unsigned types **name** and size in bits.
	• The behavior of a variable follows a variable of type integral.
	• A variable which overlaps another one (in term of address) will be considered as volatile.
		`sfr x = 0xf0; //`
		`declaration of`
		`variable x at`
		`address 0xF0`
		`sfr16 y = 0x4EEF;`
		
		For this example, options need to be:
		
		`-dialect keil`
		`-sfr-types sfr=8,  \`
		`    sfr16=16`
			sfr and sbit types are only allowed in declarations of external global variables.

Example: `-dialect keil -sfr-types sfr=8` (Continued)

Type/Language	Description	Example	Restrictions

Type `sbit`	• Each read/write access of a variable is replaced by an access of the corresponding sfr variable access.
	• Only external global variables can be mapped with a sbit variable.
	• Allowed expressions are integer variables, cells of array of int and struct/union integral fields.
	• a variable can also be declared as extern bit in an another file.
		`sfr x = 0xF0;`
		`sbit x1 = x ^ 1; // 1st bit of x`
		`sbit x2 = 0xF0 ^ 2; // 2nd bit of x`
		`sbit x3 = 0xF3; // 3rd bit of x`
		`sbit y0 = t[3] ^ 1;`
		
		`/* file1.c */`
		`sbit x = P0 ^ 1;`
		`/* file2.c */`
		`extern bit x;`
		`x = 1; // set the 1st bit of P0 to 1`	

Absolute variable location	Allowed constants are integers, strings and identifiers.	`int var _at_ 0xF0`
		`int x @ 0xFE ;`
		`static const`
		`int y @ 0xA0 = 3;`
			Absolute variable locations are ignored (even if declared with a #pragma location).

Example: `-dialect keil -sfr-types sfr=8` (Continued)

Type/Language	Description	Example	Restrictions
Interrupt functions	A warnings in the log file is displayed when an interrupt function has been found: "interrupt handler detected : <name>" or "task entry point detected : <name>"	`void foo1 (void) interrupt XX = YY using 99 { } void foo2 (void) _ task_ 99 _priority_ 2 { }`	Entry points and interrupts are not taken into account as `-entry-points`.
Keywords ignored	alien, bdata, far, idata, ebdata, huge, sdata, small, compact, large, reentrant. Defining -D __C51__, keywords large code, data, xdata, pdata and xhuge are ignored.		

The following table summarize the IAR dialect:

Example: `-dialect iar -sfr-types sfr=8`

Type/Language	Description	Example	Restrictions
Type `bit`	• An expression to type bit gives values in range [0,1]. • Converting an expression in the type, gives 1 if it is not equal to 0, else 0. This behavior is similar to c++ bool type. • If initialized with values 0 or 1, a variable of type bit is a simple variable (like a c++ bool).	`union { int v; struct { int z; } y; } s;` `void f(void) { bit y1 = s.y.z . 2; bit x4 = x.4; bit x5 = 0xF0 . 5; y1 = 1; // 2nd bit of s.y.z // is set to 1 };`	pointers to bits and arrays of bits are not allowed

Example: `-dialect iar -sfr-types sfr=8` (Continued)

Type/Language	Description	Example	Restrictions
	• A variable of type bit is a register bit variable (mapped with a bit or a sfr type)		
Type `sfr`	• The -sfr-types option defines unsigned types name and size. • The behavior of a variable follows a variable of type integral. • A variable which overlaps another one (in term of address) will be considered as volatile.	`sfr x = 0xf0; // declaration of variable x at address 0xF0`	sfr and sbit types are only allowed in declarations of external global variables.

Individual `bit` access	• Individual bit can be accessed without using sbit/bit variables.
	• Type is allowed for integer variables, cells of integer array, and struct/union integral fields.
		`int x[3], y;`
		`x[2].2 = x[0].3 + y.1;`	

Absolute variable location	Allowed constants are integers, strings and identifiers.	`int var @ 0xF0;`
		`int xx @ 0xFE ;`
		`static const int y   \`
		` @0xA0 = 3;`
			Absolute variable locations are ignored (even if declared with a #pragma location).

Example: `-dialect iar -sfr-types sfr=8` (Continued)

Type/Language	Description	Example	Restrictions

Interrupt functions	• A warning is displayed in the log file when an interrupt function has been found: "interrupt handler detected : funcname"
	• A monitor function is a function that disables interrupts while it is executing, and then restores the previous interrupt state at function exit.
		`interrupt [1]    \`
		` using [99] void   \`
		` foo1(void) { ... };`
		
		`monitor [3] void   \`
		` foo2(void) { ... };`
			Entry points and interrupts are not taken into account as `-entry-points`.

Keywords ignored	`saddr`, `reentrant`, `reentrant_idata`, `non_banked`, `plm`, `bdata`, `idata`, `pdata`, `code`, `data`, `xdata`, `xhuge`, `interrupt`, `__interrupt` and `__intrinsic`		

Unnamed struct/union	• Fields of unions/structs without a tag or a name can be accessed without naming their parent struct.
	• Option `-allow-unnamed-fields` need to be used to allow anonymous struct fields.
	• On a conflict between a field of an anonymous struct with other identifiers:
		`union { int x; };`
		`union { int y; struct { int`
		`z; }; } @ 0xF0;`	

Example: `-dialect iar -sfr-types sfr=8` (Continued)

Type/Language	Description	Example	Restrictions
	- with a variable name, field name is hidden - with a field of another anonymous struct at different scope, closer scope is chosen - with a field of another anonymous struct at same scope: an error "anonymous struct field name <name> conflict“ is displayed in the log file.		

`no_init` attribute	• a global variable declared with this attribute is handled like an external variable.
	• It is handled like a type qualifier.
		`no_init int x;`
		`no_init union`
		`{ int y; } @ 0xFE;`
			The `#pragma no_init` does not affect the code.

The option `-sfr-types` defines the size of a `sfr` type for the Keil or IAR dialect.

The syntax for an `sfr` element in the list is `type-name=typesize`.

For example:

```
-sfr-types sfr=8,sfr16=16
```

defines two `sfr` types: `sfr` with a size of 8 bits, and `sfr16` with a size of 16-bits. A value type-name must be given only once. 8, 16 and 32 are the only supported values for `type-size`.

Note As soon as an `sfr` type is used in the code, you must specify its name and size, even if it is the keyword `sfr`.

Note Many IAR and Keil compilers currently exist that are associated to specific targets. It is difficult to maintain a complete list of those supported.

Gather Compilation Options Efficiently

The code is often tuned for the target (as discussed in “Analyze Keil or IAR Dialects” on page 1-80). Rather than applying minor changes to the code, create a single polyspace.h file which contains target specific functions and options. The `-include` option can then be used to force the inclusion of the `polyspace.h` file in the source files.

Where there are missing prototypes or conflicts in variable definition, writing the expected definition or prototype within such a header file will yield several advantages.

Direct benefits:

- The error detection is much faster since it will be detected during compilation rather than in the link or subsequent phases.
- The position of the error will be identified more precisely.
- Original source files will not need to be modified.

Indirect benefits:

- The file is automatically included as the very first file in the original .c files.
- The file can contain much more powerful macro definitions than simple -D options.
- The file is reusable for other projects developed under the same environment.

Example

This is an example of a file that can be used with the `-include` option.

```
// The file may include (say) a standard include file implicitly
// included by the cross compiler

#include <stdlib.h>
#include "another_file.h"

// Generic definitions, reusable from one project to another
```

```
#define far
#define at(x)

// A prototype may be positioned here to aid in the solution of
// a link phase conflict between
// declaration and definition. This will allow detection of the
// same error at compilation time instead of at link time.
// Leads to:
// - earlier detection
// - precise localisation of conflict at compilation time
void f(int);

// The same also applies to variables.
extern int x;

// Standard library stubs can be avoided,
// and OS standard prototypes redefined.

#define POLYSPACE_NO_STANDARD_STUBS // use this flag to prevent the
              //automatic stubbing of std functions
#define __polyspace_no_sscanf
#define __polyspace_no_fgetc
void sscanf(int, char, char, char, char, char);
void fgetc(void);
```

Specify Data Ranges for Global Variables

In this section...

"Overview of Data Range Specifications (DRS)" on page 1-90

"Specify Data Ranges Using DRS Template" on page 1-90

"Remove Non Applicable Entries from DRS File" on page 1-92

"DRS Configuration Settings" on page 1-93

"Specify Data Ranges Using Existing DRS Configuration" on page 1-96

"Edit Existing DRS Configuration" on page 1-97

"XML Format of DRS File" on page 1-98

"Specify Data Ranges Using Text Files" on page 1-104

Overview of Data Range Specifications (DRS)

By default, Polyspace software assumes that all data inputs are set to their full range. Therefore, nearly any operation on these inputs could produce an overflow.

The Polyspace Data Range Specifications (DRS) feature allows you to perform a contextual analysis, analyzing the code works under normal working conditions. Using DRS, you set constraints on global variables, and analyze the code within these ranges. This can substantially reduce the number of false positives in the results.

Specify Data Ranges Using DRS Template

To use the DRS feature, you must provide a list of variables and their associated data ranges.

Polyspace software can analyze the files in your project, and generate a DRS template containing all the global variables, user-defined functions, and stub functions. You can then modify this template to set data ranges. In Polyspace Bug Finder, you can specify data ranges for global variables. In Polyspace Code Prover, you can specify data ranges for global variables, user-defined functions, and stub functions.

To use a DRS template to set data ranges for global variables:

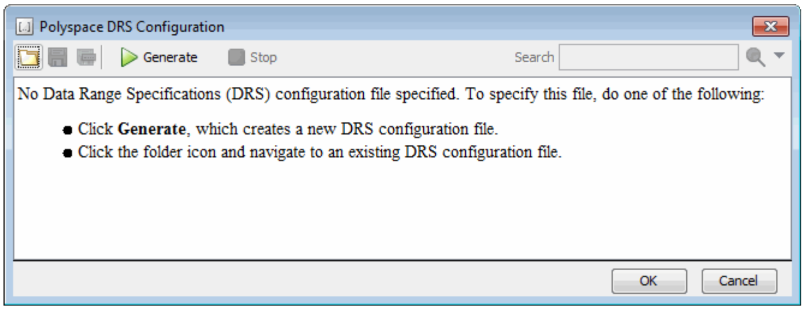
1. Open the project for which you want to set data ranges.

2. Check that the project contains the source files and include folders that you want to analyze, and specifies the configuration options that you want to use. The software generates a DRS template after compiling the code.

3. In the Project Manager perspective, select the **Configuration > Inputs & Stubbing** node.

4. To the right of the **Variable/function range setup** field, click the **Edit** button.

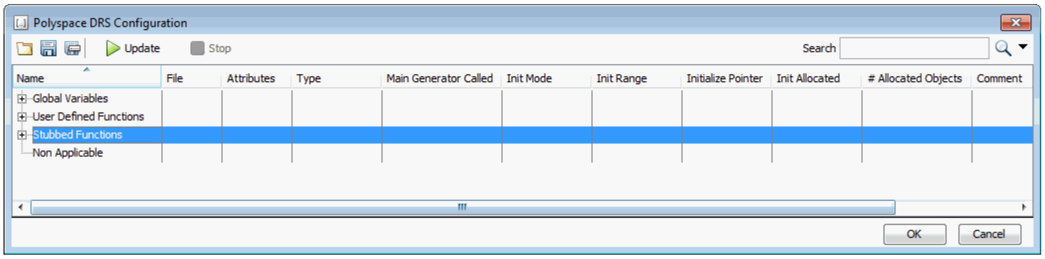
   The Polyspace DRS Configuration dialog box opens.



No Data Range Specifications (DRS) configuration file specified. To specify this file, do one of the following:

- Click **Generate**, which creates a new DRS configuration file.
- Click the folder icon and navigate to an existing DRS configuration file.

5. On the toolbar, click **Generate**. The software compiles the project and generates a DRS template, for example, `Bug_Finder_Example-with-MISRA-checker_drs_template.xml`. You can view the DRS values through the Polyspace DRS Configuration dialog box.



Name	File	Attributes	Type	Main Generator Called	Init Mode	Init Range	Initialize Pointer	Init Allocated	# Allocated Objects	Comment
Global Variables										
User Defined Functions										
Stubbed Functions										
Non Applicable										

6 Specify the data ranges for global variables. For more information, see "DRS Configuration Settings" on page 1-93.

7 To save your DRS configuration file, click (`Save DRS`).

   To save your DRS configuration file to a location that you specify, click (`Save DRS as`).

8 If you change your source code, click **Update** to generate an updated DRS configuration file. As a result of the source code changes, the updated file might contain entries that no longer apply to your code. You can remove these entries from the file. See "Remove Non Applicable Entries from DRS File" on page 1-92.

9 Click **OK** to close the **Polyspace DRS Configuration** dialog box. The **Variable/function range setup** field now contains the name of the DRS configuration file. The software uses this DRS configuration file the next time you start a verification.

10 Select **File > Save Project** to save your project settings.

Remove Non Applicable Entries from DRS File

If you change your source code, you must update your DRS configuration file. From the **Polyspace DRS Configuration** dialog box, click **Update**. The software updates the file, placing DRS entries that no longer apply to your code under the **Non Applicable** node.

You can remove:

- Entries that do not apply:
  1. Right-click **Non Applicable**.
  2. From the context menu, select **Remove This Node**.
- Entries corresponding to a subnode:
  1. Right-click the subnode, for example, **Non_Infinite_loop()**.
  2. From the context menu, select **Remove This Node**.

DRS Configuration Settings

The **Polyspace DRS Configuration** dialog box allows you to specify data ranges for global variables, user-defined functions, and stub functions in your project.

Column	Settings
Name	Displays the list of variables and functions in your Project for which you can specify data ranges. This Column displays three expandable menu items: • **Globals** – Displays a list of global variables in the Project. • **User defined functions** – Displays a list of user-defined functions in the Project. Expand a function name to see a list of the input arguments for which you can specify a data range. • **Stubbed functions** – Displays a list of stub functions in the Project. Expand a function name to see a list of the return values for which you can specify a data range.
File	Displays the name of the source file containing the variable or function.
Attributes	Displays information about the variable or function. For example, static variables display `static`.
Type	Displays the variable type.
Main Generator Called	Applicable only for user-defined functions. Specifies whether the main generator calls the function: • **`MAIN GENERATOR`** – Main generator may call this function, depending on the value of the `-functions-called-in-loop` (C) or `-main-generator-calls` (C++) parameter. • **`NO`** – Main generator will not call this function. • **`YES`** – Main generator will call this function.

Column	Settings
Init Mode	Specifies how the software assigns a range to the variable: • **`MAIN GENERATOR`** – Variable range is assigned depending on the settings of the main generator options `-variables-written-before-loop` and `-no-def-init-glob`. (For C++, the options are `-main-generator-writes-variables`, and `-no-def-init-glob`.) • **`IGNORE`** – Variable is not assigned to any range, even if a range is specified. • **`INIT`** – Variable is assigned to the specified range only at initialization, and keeps the range until first write. • **`PERMANENT`** – Variable is permanently assigned to the specified range. If the variable is assigned outside this range during the program, no warning is provided. Use the `globalassert` mode if you need a warning. User-defined functions support only `INIT` mode. Stub functions support only `PERMANENT` mode. For C verifications, global pointers support `MAIN GENERATOR`, `IGNORE`, or `INIT` mode. • **`MAIN GENERATOR`** – Pointer follows the options of the main generator. • **`IGNORE`** – Pointer is not initialized • **`INIT`** – Specify if the pointer is NULL, and how the pointed object is allocated (**Initialize Pointer** and **Init Allocated** options).

Column	Settings
Init Range	Specifies the minimum and maximum values for the variable. You can use the keywords `min` and `max` to denote the minimum and maximum values of the variable type. For example, for the type long, `min` and `max` correspond to -2^31 and 2^31-1 respectively. You can also use hexadecimal values. For example: `0x12..0x100`
Initialize Pointer	Applicable only to pointers. Enabled only when you specify **Init Mode**:`INIT`. Specifies whether the pointer should be NULL: • **`May-be NULL`** – The pointer could potentially be a NULL pointer (or not). • **`Not Null`** – The pointer is never initialized as a null pointer. • **`Null`** – The pointer is initialized as NULL. **Note** Not applicable for C++ projects.
Init Allocated	Applicable only to pointers. Enabled only when you specify **Init Mode**:`INIT`. Specifies how the pointed object is allocated: • **`MAIN GENERATOR`** – The pointed object is allocated by the main generator. • **`None`** – Pointed object is not written. • **`SINGLE`** – Write the pointed object or the first element of an array. (This setting is useful for stubbed function parameters.) • **`MULTI`** – All objects (or array elements) are initialized.

Column	Settings
	Note Not applicable for C++ projects.
# Allocated Objects	Applicable only to pointers.Specifies how many objects are pointed to by the pointer (the pointed object is considered as an array). **Note:** The Init Allocated parameter specifies how many allocated objects are actually initialized. **Note** Not applicable for C++ projects.
Global Assert	Specifies whether to perform an assert check on the variable at global initialization, and after each assignment.
Global Assert Range	Specifies the minimum and maximum values for the range you want to check.
Comment	Remarks that you enter, for example, justification for your DRS values.

Specify Data Ranges Using Existing DRS Configuration

Once you have created a DRS configuration file for a project, you can reuse the data ranges for subsequent verifications.

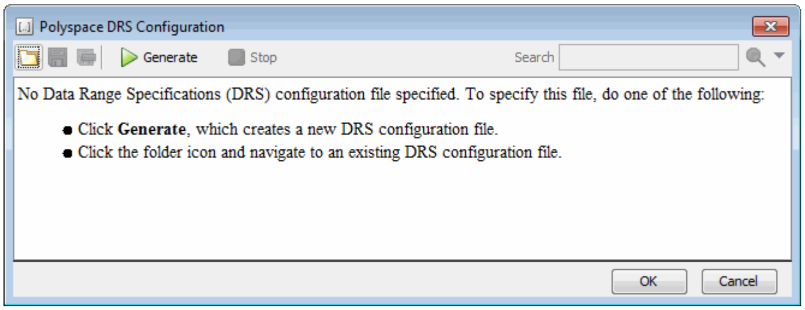
To specify an existing DRS configuration file for your project:

1 Open the project for which you want to set data ranges.

2 In the Project Manager perspective, select the **Configuration > Inputs & Stubbing** node.

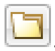
3 To the right of the **Variable/function range setup** field, click the **Edit** button.

The Polyspace DRS Configuration dialog box opens.



No Data Range Specifications (DRS) configuration file specified. To specify this file, do one of the following:

- Click **Generate**, which creates a new DRS configuration file.
- Click the folder icon and navigate to an existing DRS configuration file.

4 On the toolbar, click the button .

5 In the **Load a DRS file** dialog box, navigate to the folder that contains the required DRS configuration file, and select the file. Then click **Open**. The Load a DRS file dialog box closes.

6 In the **Polyspace DRS Configuration dialog** box, click **OK**.

7 Select **File > Save Project** to save your project settings, including the DRS file location.

The software uses the specified DRS configuration file the next time you start an analysis.

Edit Existing DRS Configuration

Once you have created a DRS configuration file for your project, you can edit the configuration using the **Polyspace DRS Configuration** dialog box.

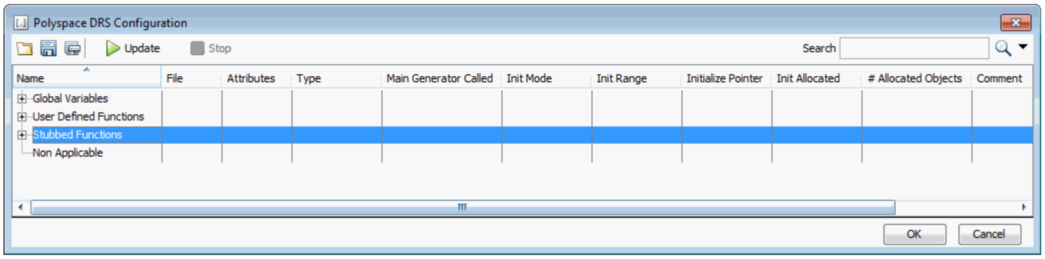
To edit an existing DRS configuration:

1 Open the project.

2 In the Project Manager perspective, select the **Configuration > Inputs & Stubbing** node.

3 To the right of the **Variable/function range setup** field, click the **Edit** button.

The Polyspace DRS Configuration dialog box opens.



Name	File	Attributes	Type	Main Generator Called	Init Mode	Init Range	Initialize Pointer	Init Allocated	# Allocated Objects	Comment
Global Variables										
User Defined Functions										
Stubbed Functions										
Non Applicable										

4 Specify the data ranges for global variables.

5 To save your DRS configuration file, click (`Save DRS`),

6 Click **OK**, which closes the Polyspace DRS Configuration dialog box.

XML Format of DRS File

Syntax Description — XML Elements

The DRS file contains the following XML elements:

- `<global>` element — Declares the global scope, and is the root element of the XML file.
- `<file>` element — Declares a file scope. Must be enclosed in the `<global>` element. May enclose any variable or function declaration. Static variables must be enclosed in a file element to avoid conflicts.
- `<scalar>` element— Declares an integer or a floating point variable. May be enclosed in any recognized element, but cannot enclose any element. Sets init/permanent/global asserts on variables.
- `<pointer>` element — Declares a pointer variable. May enclose any other variable declarations (including itself), to define the pointed objects. Specifies what value is written into pointer (NULL or not), how many objects are allocated and how the pointed objects are initialized.
- `<array>` element — Declares an array variable. May enclose any other variable definition (including itself), to define the members of the array.

- `<struct>` element — Declares a structure variable or object (instance of class). May enclose any other variable definition (including itself), to define the fields of the structure.
- `<function>` element — Declares a function or class method scope. May enclose any variable definition, to define the arguments and the return value of the function. Arguments should be named *`arg1, arg2, …argn`* and the return value should be called *`return`*.

The following notes apply to specific fields in each XML element:

- **(*)** — Fields used only by the GUI. These fields are not mandatory for verification to accept the ranges. The field line contains the line number where the variable is declared in the source code, `complete_type` contains a string with the complete variable type, and `base_type` is used by the GUI to compute the min and max values. The field comment is used to add information about any node.
- **(**)** — The field name is mandatory for scope elements `<file>` and `<function>` (except for function pointers). For other elements, the name must be specified when declaring a root symbol or a `struct` field.
- **(***)** — If more than one attribute applies to the variable, the attributes must be separated by a space. Only the static attribute is mandatory, to avoid conflicts between static variables having the same name. An attribute can be defined multiple times without impact.
- **(****)** — This element is used only by the GUI, to determine which `init` modes are allowed for the current element (according to its type). The value works as a mask, where the following values are added to specify which modes are allowed:
  - **1**: The mode "`NO`" is allowed.
  - **2** : The mode "`INIT`" is allowed.
  - **4**: The mode "`PERMANENT`" is allowed.
  - **8**: The mode "`MAIN_GENERATOR`" is allowed.

  For example, the value "**10**" means that modes "`INIT`" and "`MAIN_GENERATOR`" are allowed. To see how this value is computed, refer to "Valid Modes and Default Values" on page 1-103.

- **(*****)** — A sub-element of a pointer (i.e. a pointed object) will be taken into account only if `init_pointed` is equal to SINGLE or MULTI.

`<file>` Element.

Field	Syntax
`name`	*`filepath_or_filename`*
`comment`	*`string`*

`<scalar>` Element.

Field	Syntax
`name` (**)	*`name`*
`line` (*)	*`line`*

`base_type` (*)	`intx`
	`uintx`
	`floatx`

`Attributes` (***)	`volatile`
	`extern`
	`static`
	`const`

`complete_type` (*)	*`type`*

`init_mode`	`MAIN_GENERATOR`
	`IGNORE`
	`INIT`
	`PERMANENT`
	`disabled`
	`unsupported`

`init_modes_allowed` (*)	*`single value`* (****)

`init_range`	*`range`*
	`disabled`
	`unsupported`

Field	Syntax

`global_ assert`	`YES`
	`NO`
	`disabled`
	`unsupported`

`assert_range`	*`range`*
	`disabled`
	`unsupported`

`comment`(*)	*`string`*

`<pointer>` Element.

Field	Syntax
`Name` (**)	*`name`*
`line` (*)	*`line`*

`Attributes` (***)	`volatile`
	`extern`
	`static`
	`const`

`complete_type` (*)	*`type`*

`init_mode`	`MAIN_GENERATOR`
	`IGNORE`
	`INIT`
	`PERMANENT`
	`disabled`
	`unsupported`

`init_modes_allowed` (*)	*`single value`* (****)

`initialize_ pointer`	May be:
	`NULL`
	`Not NULL`
	`NULL`

`number_ allocated`	*`single value`*
	`disabled`
	`unsupported`

Field	Syntax
init_pointed	MAIN_GENERATOR NONE SINGLE MULTI disabled
comment	*string*

<array> and <struct> Elements.

Field	Syntax
Name (**)	*name*
line (*)	*line*
complete_type (*)	*type*
attributes (***)	volatile extern static const
comment	*string*

<function> Element.

Field	Syntax
Name (**)	*name*
line (*)	*line*
main_generator_called	MAIN_GENERATOR YES NO disabled

Field	Syntax

`attributes` (***)	`static`
	`extern`
	`unused`

`comment`	*`string`*

Valid Modes and Default Values

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
Global variables	Base type	Unqualified/ static/ const scalar	MAIN_ GENERATOR IGNORE INIT PERMANENT	YES NO			Main generator dependant
		Volatile scalar	PERMANENT	disabled			PERMANENT min..max
		Extern scalar	INIT PERMANENT	YES NO			INIT min..max
	Struct	Struct field	Refer to field type				
	Array	Array element	Refer to element type				

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
Global variables	Pointer	Unqualified/ static/ const scalar	MAIN_ GENERATOR IGNORE INIT		May be NULL Not NULL NULL	NONE SINGLE MULTI	Main generator dependant
		Volatile pointer	un-supported		un-supported	un-supported	
		Extern pointer	IGNORE INIT		May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI
		Pointed volatile scalar	un-supported	un-supported			
		Pointed extern scalar	INIT	un-supported			INIT min..max
		Pointed other scalars	MAIN_ GENERATOR INIT	un-supported			MAIN_ GENERATOR dependant
		Pointed pointer	MAIN_ GENERATOR INIT/	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	MAIN_ GENERATOR dependant
		Pointed function	un-supported	un-supported			

Specify Data Ranges Using Text Files

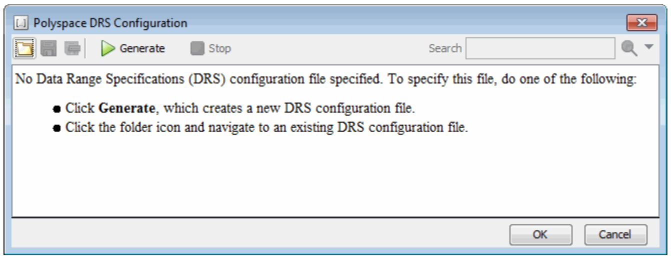
To use the DRS feature, you must provide a list of variables and their associated data ranges.

You can specify data ranges using the **Polyspace DRS Configuration** dialog box (see “Specify Data Ranges Using DRS Template” on page 1-90), or you can provide a text file that contains a list of variables and data ranges.

To specify data ranges using a DRS text file:

1. Create a DRS text file containing the list of global variables (or functions) and their associated data ranges, as described in “DRS Text File Format” on page 1-106.
2. Open the project.
3. In the Project Manager perspective, select the **Configuration > Inputs & Stubbing** node.
4. To the right of the **Variable/function range setup** field, click the **Edit** button.

   The Polyspace DRS Configuration dialog box opens.



No Data Range Specifications (DRS) configuration file specified. To specify this file, do one of the following:

- Click **Generate**, which creates a new DRS configuration file.
- Click the folder icon and navigate to an existing DRS configuration file.

5. On the toolbar, click the button 📂.
6. Navigate to the folder that contains the required DRS text file, and select the file. Then click **Open**.
7. In the Polyspace DRS Configuration dialog box, click **OK**.
8. Select **File > Save Project** to save your project settings, including the DRS file location.

When you run an analysis, the software automatically merges the data ranges in the text file with a DRS template for the project and saves the information in the file `drs-template.xml`, located in your results folder.

DRS Text File Format

The DRS file contains a list of global variables and associated data ranges. The point during verification at which the range is applied to a variable is controlled by the mode keyword: `init`, `permanent`, or `globalassert`.

The DRS file must have the following format:

```
variable_name min_value max_value <init|permanent|globalassert>
```

- *`variable_name`* — The name of the global variable.
- *`min_value`* — The minimum value for the variable.
- *`max_value`* — The maximum value for the variable.
- `init` — The variable is assigned to the specified range only at initialization, and keeps it until first write.
- `permanent` — The variable is permanently assigned to the specified range. If the variable is assigned outside this range during the program, no warning is provided. Use the `globalassert` mode if you need a warning.
- `globalassert` — After each assignment, an assert check is performed, controlling the specified range. The assert check is also performed at global initialization.

Tips for Creating DRS Text Files

- You can use the keywords "`min`" and "`max`" to denote the minimum and maximum values of the variable type. For example, for the type long, min and max correspond to -2^31 and 2^31-1 respectively.
- You can use hexadecimal values. For example, `x 0x12 0x100 init`.
- Supported column separators are tab, comma, space, or semicolon.
- To insert comments, use shell style “`#`”.

Example DRS Text File

In the following example, the global variables are named x, y, z, w, and v.

```
x  12  100    init
y  0   10000  permanent
z  0   1      globalassert
w  min max    permanent
v  0   max    globalassert
arrayOfInt  -10  20  init
s1.id       0    max init
array.c2    min  1   init
car.speed   0    350 permanent
bar.return  -100 100 permanent

# x is defined between [12;100] at initialization
# y is permanently defined between [0,10000] even any assignment
# z is checked in the range [0;1] after each assignment
# w is volatile and full range on its declaration type
# v is positive and checked after each assignment.
# All cells arrayOfInt are defined between [-10;20] at initialization
# s1.id is defined between [0;2^31-1] at initialisation.
# All cells array[i].c2 are defined between [-2^31;1] at initialization
# Speed of Struct car is permanently defined between 0 and 350 Km/h
# function bar returns -100..100
```

Setting Up Project: Additional Information

- “Create Projects Using Visual Studio Information” on page 2-2
- “Cannot create project from Visual Studio build” on page 2-6
- “Storage of Polyspace Preferences” on page 2-7

Create Projects Using Visual Studio Information

In this section…
"Use Visual Studio Project" on page 2-2
"Trace Visual Studio Build" on page 2-3

Use Visual Studio Project

You can directly create a Polyspace project from a Visual Studio® project file with extension `.vcproj`. The Visual Studio import retrieves the following information from a Visual Studio project:

- **Source** files
- **Include** folders
- Some **Target & Compiler** options
- **Preprocessor Macros**

Note For Visual Studio 2010 or Visual Studio 2012, you cannot directly import your project.

1. In the Project Manager perspective, select **File > Import Visual Studio Project**.
2. In the Import Visual Studio dialog box, specify the **Visual Studio project** that you want to use.
3. You can:
   - **Create new Polyspace project**: Enter full path to a new Polyspace project.
   - **Update existing Polyspace project**: The dropdown list contains all projects currently open in the **Project Browser**. Select the project you want to update.
4. Click **Import**.

Trace Visual Studio Build

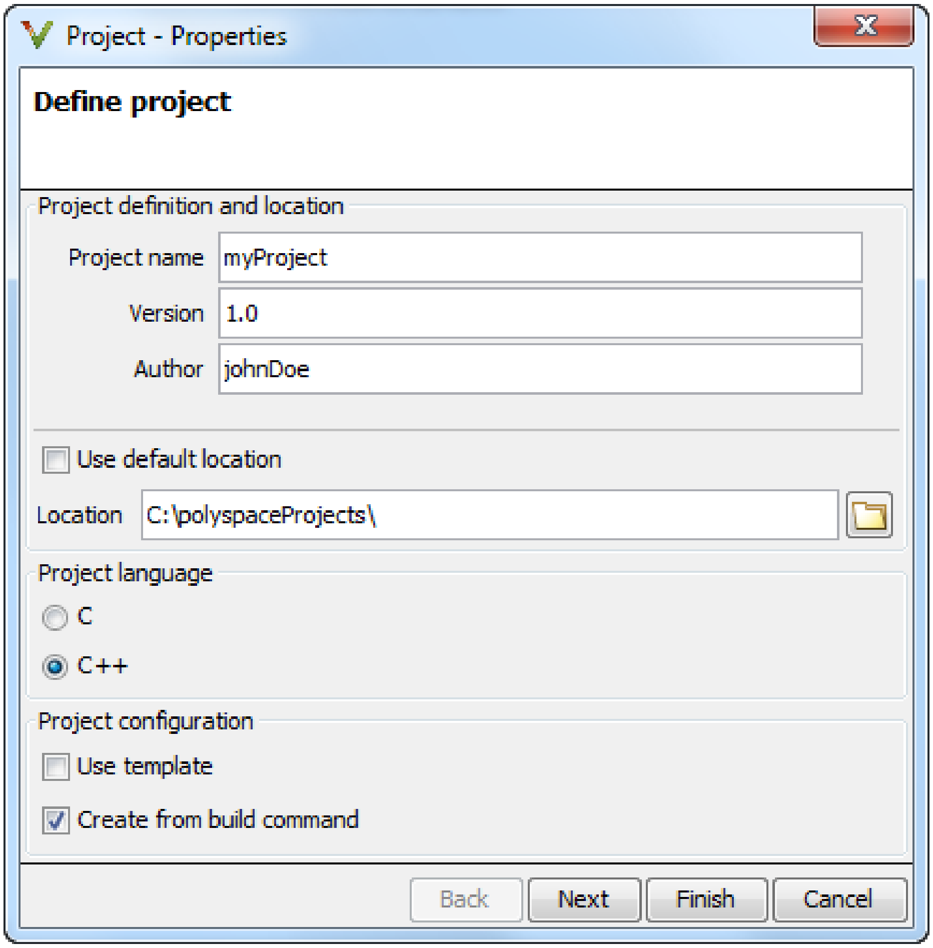
To create a Polyspace project, you can trace your Visual Studio build.

1 In the Polyspace Project Manager, select **File > New Project**.

2 In the Project – Properties window, enter your project information.

   a Choose **C++** as **Project Language**.

   b Under **Project Configuration**, select **Create from build command** and click **Next**.


Project - Properties

Define project

Project definition and location

Project name: myProject

Version: 1.0

Author: johnDoe

☐ Use default location

Location: C:\polyspaceProjects\

Project language

○ C

● C++

Project configuration

☐ Use template

☑ Create from build command

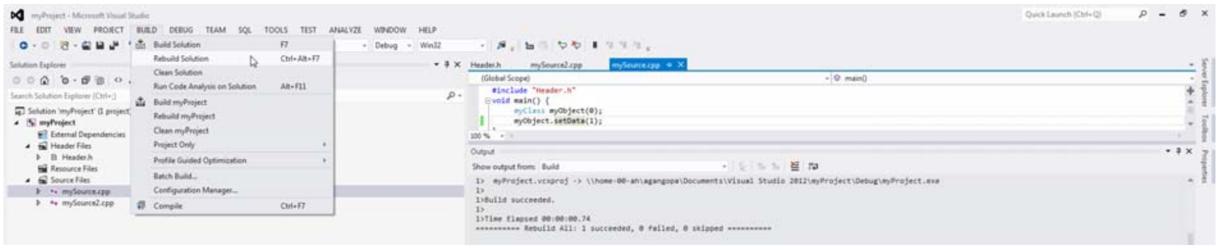
3 In the field **Specify command used for building your source files**, enter the full path to the Visual Studio executable. For instance, `"C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\devenv.exe"`.

4 In the field **Specify working directory for running build command**, enter `C:\`. Click Run.

This action opens the Visual Studio environment.

5 In the Visual Studio environment, create and build a Visual Studio project.

   If you already have a Visual Studio project, open the existing project and build a clean solution. To build a clean solution in Visual Studio 2012, select **BUILD > Rebuild Solution**.



```
#include "Header.h"
void main() {
    myClass myObject(0);
    myObject.setData(1);
```

```
1>  myProject.vcxproj -> \\home-00-ah\agangopa\Documents\Visual Studio 2012\myProject\Debug\myProject.exe
1>
1>Build succeeded.
1>
1>Time Elapsed 00:00:00.74
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
```

6 After the project builds, close Visual Studio.

   Polyspace traces your Visual Studio build and creates a Polyspace project.

   The Polyspace project contains the source files from your Visual Studio build and the relevant **Target & Compiler** options.

7 If you update your Visual Studio project, to update the corresponding Polyspace project, on the **Project Browser**, right-click the project name and select **Update Project**.

Related Examples

- “Visual Studio Environment”

Concepts

- “Cannot create project from Visual Studio build” on page 2-6

Cannot create project from Visual Studio build

If you are trying to import a Visual Studio 2010 or Visual Studio 2012 project and `polyspace-configure` does not work properly, do the following:

1. Stop the `MSBuild.exe` process.
2. Set the environment variable `MSBUILDDISABLENODEREUSE` to 1.
3. Specify `MSBuild.exe` with the`/nodereuse:false` option.
4. Restart the Polyspace configuration tool:

   ```
   polyspace-configure.exe -lang cpp <MSVS
   path>/msbuild sample.sln
   ```

Storage of Polyspace Preferences

The software stores the settings that you specify through the Polyspace Preferences dialog box in the following file:

- Windows: `$Drive\Users\$User\AppData\Roaming\MathWorks\MATLAB\$Release\Polyspace\polyspace.prf`
- Linux: `/home/$User/.matlab/$Release/Polyspace/polyspace.prf`

Here, `$Drive` is the drive where the operating system files are located such as `C:`, `$User` is the username such as `johndoe` and `$Release` is the release number such as `2014a`.

The following file stores the location of all installed Polyspace products across various releases:

- Windows: `$Drive\Users\$User\AppData\Roaming\MathWorks\MATLAB\AppData\Roaming\MathWorks\MATLAB\polyspace_shared\polyspace_products.prf`
- Linux : `/home/$User/.matlab/polyspace_shared/polyspace_products.prf`

Coding Rule Sets and Concepts

- “Rule Checking” on page 3-2
- “Custom Naming Convention Rules” on page 3-3
- “Polyspace MISRA C and MISRA AC AGC Checkers” on page 3-10
- “Software Quality Objective Subsets (C)” on page 3-11
- “Software Quality Objective Subsets (AC AGC)” on page 3-16
- “MISRA C:2004 Coding Rules” on page 3-18
- “Polyspace MISRA C++ Checker” on page 3-59
- “Software Quality Objective Subsets (C++)” on page 3-60
- “MISRA C++ Coding Rules” on page 3-69
- “Polyspace JSF C++ Checker” on page 3-95
- “JSF C++ Coding Rules” on page 3-96

Rule Checking

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

Applying coding rules can reduce the number of defects and improve the quality of your code.

While creating a project, you specify both the coding standard, and individual rules to enforce. Polyspace software then performs rule checking before starting analysis, and reports any errors or warnings in the Results Manager perspective.

If any source files in the analysis do not compile, coding rules checking will be incomplete. The coding rules checker results:

- May not contain full results for files that did not compile
- May not contain full results for the files that did compile as some rules are checked only after compilation is complete

---

Note The Compiler Assistant is selected by default. However, when you enable the Compiler Assistant *and* coding rules checking, the software does not report coding rule violations if there are compilation errors.

Custom Naming Convention Rules

The following table provides information about the custom rules that you can define.

Rule group	Number	Rule Applied	Message generated if rule is violated	Other details
Files (C/C++)	1.1	All source file names must follow the specified pattern.	The source file name "file_name" does not match the specified pattern.	Only the base name is checked. A source file is a file that is not included.
	1.2	All source folder names must follow the specified pattern.	The source dir name "dir_name" does not match the specified pattern.	Only the folder name is checked. A source file is a file that is not included.
	1.3	All include file names must follow the specified pattern.	The include file name "file_name" does not match the specified pattern.	Only the base name is checked. An include file is a file that is included.
	1.4	All include folder names must follow the specified pattern.	The include dir name "dir_name" does not match the specified pattern.	Only the folder name is checked. An include file is a file that is included.
Preprocessing (C/C++)	2.1	All macros must follow the specified pattern.	The macro "macro_name" does not match the specified pattern.	Macro names are checked before preprocessing.
	2.2	All macro parameters must follow the specified pattern.	The macro parameter "param_name" does not match the specified pattern.	Macro parameters are checked before preprocessing.

Rule group	Number	Rule Applied	Message generated if rule is violated	Other details
Type definitions (C/C++)	3.1	All integer types must follow the specified pattern.	The integer type "type_name" does not match the specified pattern.	Applies to integer types specified by `typedef` statements. Does not apply to enumeration types. For example: `typedef signed int int32_t;`
	3.2	All float types must follow the specified pattern.	The float type "type_name" does not match the specified pattern.	Applies to float types specified by `typedef` statements. For example: `typedef float f32_t;`
	3.3	All pointer types must follow the specified pattern.	The pointer type "type_name" does not match the specified pattern.	Applies to pointer types specified by `typedef` statements. For example: `typedef int* p_int;`
	3.4	All array types must follow the specified pattern.	The array type "type_name" does not match the specified pattern.	Applies to array types specified by `typedef` statements. For example: `typedef int[3] a_int_3;`
	3.5	All function pointer types must follow the specified pattern.	The function pointer type "type_name" does not match the specified pattern.	Applies to function pointer types specified by `typedef` statements. For example: `typedef void (*pf_callback) (int);`

Rule group	Number	Rule Applied	Message generated if rule is violated	Other details
Structures (C/C++)	4.1	All `struct` tags must follow the specified pattern.	The struct tag "tag_name" does not match the specified pattern.	
	4.2	All `struct` types must follow the specified pattern.	The struct type "type_name" does not match the specified pattern.	This is the `typedef` name.
	4.3	All `struct` fields must follow the specified pattern.	The struct field "field_name" does not match the specified pattern.	
	4.4	All `struct` bit fields must follow the specified pattern.	The struct bit field "field_name" does not match the specified pattern.	
Classes (C++)	5.1	All class names must follow the specified pattern.	The class tag "tag_name" does not match the specified pattern.	
	5.2	All class types must follow the specified pattern.	The class type "type_name" does not match the specified pattern.	This is the `typedef` name.
	5.3	All data members must follow the specified pattern.	The data member "member_name" does not match the specified pattern.	
	5.4	All function members must follow the specified pattern.	The function member "member_name" does not match the specified pattern.	

Rule group	Number	Rule Applied	Message generated if rule is violated	Other details
	5.5	All static data members must follow the specified pattern.	The static data member "member_name" does not match the specified pattern.	
	5.6	All static function members must follow the specified pattern.	The static function member "member_name" does not match the specified pattern.	
	5.7	All bitfield members must follow the specified pattern.	The bitfield "member_name" does not match the specified pattern.	
Enumerations (C/C++)	6.1	All enumeration tags must follow the specified pattern.	The enumeration tag "tag_name" does not match the specified pattern.	
	6.2	All enumeration types must follow the specified pattern.	The enumeration type "type_name" does not match the specified pattern.	This is the typedef name.
	6.3	All enumeration constants must follow the specified pattern.	The enumeration constant "constant_name" does not match the specified pattern.	

Rule group	Number	Rule Applied	Message generated if rule is violated	Other details
Functions (C/C++)	7.1	All global functions must follow the specified pattern.	The global function “function_name” does not match the specified pattern.	A global function is a function with external linkage.
	7.2	All static functions must follow the specified pattern.	The static function “function_name” does not match the specified pattern.	A static function is a function with internal linkage.
	7.3	All function parameters must follow the specified pattern.	The function parameter “param_name” does not match the specified pattern.	In C++, applies to non-member functions.
Constants (C/C++)	8.1	All global constants must follow the specified pattern.	The global constant “constant_name” does not match the specified pattern.	A global constant is a constant with external linkage.
	8.2	All static constants must follow the specified pattern.	The static constant “constant_name” does not match the specified pattern.	A static constant is a constant with internal linkage.
	8.3	All local constants must follow the specified pattern.	The local constant “constant_name” does not match the specified pattern.	A local constant is a constant with no linkage.
	8.4	All static local constants must follow the specified pattern.	The static local constant “constant_name” does not match the specified pattern.	A static local constant is a constant declared static in a function.

Rule group	Number	Rule Applied	Message generated if rule is violated	Other details
Variables (C/C++)	9.1	All global variables must follow the specified pattern.	The global variable “var_name” does not match the specified pattern.	A global variable is a variable with external linkage.
	9.2	All static variables must follow the specified pattern.	The static variable “var_name” does not match the specified pattern.	A static variable is a variable with internal linkage.
	9.3	All local variables must follow the specified pattern.	The local variable “var_name” does not match the specified pattern.	A local variable is a variable with no linkage.
	9.4	All static local variables must follow the specified pattern.	The static local variable “var_name” does not match the specified pattern.	A static local variable is a variable declared static in a function.
Name spaces (C++)	10.1	All namespaces must follow the specified pattern.	The namespace “namespace_name” does not match the specified pattern.	
Class templates (C++)	11.1	All class templates must follow the specified pattern.	The class template “template_name” does not match the specified pattern.	
	11.2	All class template parameters must follow the specified pattern.	The class template parameter “param_name” does not match the specified pattern.	

Rule group	Number	Rule Applied	Message generated if rule is violated	Other details
Function templates (C++)	12.1	All function templates must follow the specified pattern.	The function template "template_name" does not match the specified pattern.	Applies to non-member functions.
	12.2	All function template parameters must follow the specified pattern.	The function template parameter "param_name" does not match the specified pattern.	Applies to non-member functions.
	12.3	All function template members must follow the specified pattern.	The function template member "member_name" does not match the specified pattern.	

Polyspace MISRA C and MISRA AC AGC Checkers

The Polyspace MISRA C checker helps you comply with the MISRA C 2004 coding standard.[8]

When MISRA C rules are violated, the MISRA C checker enables Polyspace software to provide messages with information about the rule violations. Most messages are reported during the compile phase of an analysis.

The MISRA C checker can check nearly all of the **142** MISRA C:2004 rules.

The MISRA AC AGC checker checks rules from the OBL (obligatory) and REC (recommended) categories specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

There are subsets of MISRA coding rules that can have a direct or indirect impact on the selectivity (reliability percentage) of your results. When you set up rule checking, you can select these subsets directly. These subsets are defined in:

- "Software Quality Objective Subsets (C)" on page 3-11
- "Software Quality Objective Subsets (AC AGC)" on page 3-16

Note The Polyspace MISRA checker is based on MISRA C:2004, which also incorporates MISRA-C Technical Corrigendum (http://www.misra-c.com).

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

Software Quality Objective Subsets (C)

In this section...

"Rules in `SQO-Subset1`" on page 3-11

"Rules in `SQO-Subset2`" on page 3-13

Rules in `SQO-Subset1`

In Polyspace Code Prover, the following set of coding rules will typically reduce the number of unproven results.

Rule number	Description
MISRA 8.11	The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
MISRA 8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
MISRA 11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
MISRA 11.3	A cast should not be performed between a pointer type and an integral type.
MISRA 12.12	The underlying bit representations of floating-point values shall not be used.
MISRA 13.3	Floating-point expressions shall not be tested for equality or inequality.
MISRA 13.4	The controlling expression of a *for* statement shall not contain any objects of floating type.

Rule number	Description
MISRA 13.5	The three expressions of a *for* statement shall be concerned only with loop control.
MISRA 14.4	The *goto* statement shall not be used.
MISRA 14.7	A function shall have a single point of exit at the end of the function.
MISRA 16.1	Functions shall not be defined with variable numbers of arguments.
MISRA 16.2	Functions shall not call themselves, either directly or indirectly.
MISRA 16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.
MISRA 17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
MISRA 17.4	Array indexing shall be the only allowed form of pointer arithmetic.
MISRA 17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.
MISRA 17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
MISRA 18.3	An area of memory shall not be reused for unrelated purposes.
MISRA 18.4	Unions shall not be used.
MISRA 20.4	Dynamic heap memory allocation shall not be used.

Note Polyspace software does not check MISRA rule **18.3**.

Rules in `SQO-Subset2`

Good design practices generally lead to less code complexity, which can reduce the number of unproven results in Polyspace Code Prover. The following set of coding rules enforce good design practices.

Note Specifying `SQO-subset2` in your **MISRA C rules configuration** checks both the rules listed in `SQO-subset1` and `SQO-subset2`.

Rule number	Description
MISRA 6.3	*typedefs* that indicate size and signedness should be used in place of the basic types
MISRA 8.7	Objects shall be defined at block scope if they are only accessed from within a single function
MISRA 9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures
MISRA 9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized
MISRA 10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression
MISRA 10.5	Bitwise operations shall not be performed on signed integer types
MISRA 11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type
MISRA 11.5	Type casting from any type to or from pointers shall not be used
MISRA 12.1	Limited dependence should be placed on C's operator precedence rules in expressions
MISRA 12.2	The value of an expression shall be the same under any order of evaluation that the standard permits

Rule number	Description
MISRA 12.5	The operands of a logical && or || shall be primary-expressions
MISRA 12.6	Operands of logical operators (&&, || and !) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (&&, || or !)
MISRA 12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned
MISRA 12.10	The comma operator shall not be used
MISRA 13.1	Assignment operators shall not be used in expressions that yield Boolean values
MISRA 13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean
MISRA 13.6	Numeric variables being used within a "*for*" loop for iteration counting should not be modified in the body of the loop
MISRA 14.8	The statement forming the body of a *switch, while, do while* or *for* statement shall be a compound statement
MISRA 14.10	All *if else if* constructs should contain a final *else* clause
MISRA 15.3	The final clause of a *switch* statement shall be the *default* clause
MISRA 16.3	Identifiers shall be given for all of the parameters in a function prototype declaration
MISRA 16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
MISRA 16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty
MISRA 19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct

Rule number	Description
MISRA 19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives
MISRA 19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##
MISRA 19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator
MISRA 19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.
MISRA 20.3	The validity of values passed to library functions shall be checked.

Note Polyspace software does not check MISRA rule **20.3** directly.

However, you can check this rule by writing manual stubs that check the validity of values. For example, the following code checks the validity of an input being greater than 1:

```
int my_system_library_call(int in) {assert (in>1); if random \
return -1 else return 0; }
```

Software Quality Objective Subsets (AC AGC)

In this section...

“Rules in `SQO-Subset1`” on page 3-16

“Rules in `SQO-Subset2`” on page 3-16

Rules in `SQO-Subset1`

- 5.2
- 8.11 and 8.12
- 11.2 and 11.3
- 12.12
- 14.7
- 16.1 and 16.2
- 17.3 and 17.6
- 18.4

For more information about these rules, see *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

Rules in `SQO-Subset2`

- 5.2
- 6.3
- 8.7, 8.11, and 8.12
- 9.3
- 11.1, 11.2, 11.3, and 11.5
- 12.2, 12.9, 12.10, and 12.12
- 14.7
- 16.1, 16.2, 16.3, 16.8, and 16.9
- 17.3, and 17.6

- 18.4
- 19.9, 19.10, 19.11, and 19.12
- 20.3

Note When you specify `SQO-subset2` for your MISRA AC AGC rules configuration, the software checks the rules listed in `SQO-subset1` and `SQO-subset2`.

For more information about these rules, see *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

MISRA C:2004 Coding Rules

In this section...

"Supported MISRA C:2004 Rules" on page 3-18

"MISRA C:2004 Rules Not Checked" on page 3-56

Supported MISRA C:2004 Rules

The following tables list MISRA C:2004 coding rules that the Polyspace coding rules checker supports. Details regarding how the software checks individual rules and any limitations on the scope of checking are described in the "Detailed Polyspace Specification" column.

Note The Polyspace coding rules checker:

- Supports MISRA-C:2004 Technical Corrigendum 1 for rules 4.1, 5.1, 5.3, 6.1, 6.3, 7.1, 9.2, 10.5, 12.6, 13.5, and 15.0.
- Checks rules specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

The software reports most violations during the compile phase of an analysis. However, the software detects violations of rules 9.1 (`Non-initialized variable`), 12.11 (one of the overflow checks) using `-scalar-overflows-checks signed-and-unsigned`), 13.7 (dead code), 14.1 (dead code), 16.2 and 21.1 during code analysis, and reports these violations as run-time errors.

Note Some violations of rules 13.7 and 14.1 are reported during the compile phase of analysis.

Environment

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
1.1	All code shall conform to ISO® 9899:1990 “Programming languages - C”, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.	The text `All code shall conform to ISO 9899:1990 Programming languages C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996` precedes each of the following messages: • ANSI® C does not allow ‘#include_next’ • ANSI C does not allow macros with variable arguments list • ANSI C does not allow ‘#assert’ • ANSI C does not allow ’#unassert’ • ANSI C does not allow testing assertions • ANSI C does not allow ’#ident’ • ANSI C does not allow ’#sccs’ • text following ’#else’ violates ANSI standard. • text following ’#endif’ violates ANSI standard.	All the supported extensions lead to a violation of this MISRA rule. Standard compilation error messages do not lead to a violation of this MISRA rule and remain unchanged.

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
		• text following '#else' or '#endif' violates ANSI standard. • ANSI C90 forbids 'long long int' type. • ANSI C90 forbids 'long double' type. • ANSI C90 forbids long long integer constants. • Keyword 'inline' should not be used. • Array of zero size should not be used. • Integer constant does not fit within unsigned long int. • Integer constant does not fit within long int.	

Language Extensions

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
2.1	Assembly language shall be encapsulated and isolated.	Assembly language shall be encapsulated and isolated.	No warnings if code is encapsulated in `asm` functions or in `asm pragma` (only warning is given on

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
			asm statements even if it is encapsulated by a MACRO).
2.2	Source code shall only use /* */ style comments	C++ comments shall not be used.	C++ comments are handled as comments but lead to a violation of this MISRA rule**Note**: This rule cannot be annotated in the source code.
2.3	The character sequence /* shall not be used within a comment	The character sequence /* shall not appear within a comment.	This rule violation is also raised when the character sequence /* inside a C++ comment.**Note**: This rule cannot be annotated in the source code.

Documentation

Rule	MISRA Definition	Messages in report file	Detailed Polyspace Specification
3.4	All uses of the #*pragma* directive shall be documented and explained.	All uses of the #pragma directive shall be documented and explained.	To check this rule, the option `-allowed-pragmas` must be set to the list of pragmas that are allowed in source files. Warning if a pragma that does not belong to the list is found.

Character Sets

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.	\<character> is not an ISO C escape sequence Only those escape sequences which are defined in the ISO C standard shall be used.	
4.2	Trigraphs shall not be used.	Trigraphs shall not be used.	Trigraphs are handled and converted to the equivalent character but lead to a violation of the MISRA rule

Identifiers

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
5.1	Identifiers (internal and external) shall not rely on the significance of more than 31 characters	Identifier 'XX' should not rely on the significance of more than 31 characters.	All identifiers (global, static and local) are checked.
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	• Local declaration of XX is hiding another identifier. • Declaration of parameter XX is hiding another identifier.	Assumes that rule 8.1 is not violated.
5.3	A typedef name shall be a unique identifier	{ typedef name }'%s' should not be reused. (already used as { typedef name } at %s:%d)	Warning when a typedef name is reused as another identifier name.
5.4	A tag name shall be a unique identifier	{tag name }'%s' should not be reused. (already used as {tag name } at %s:%d)	Warning when a tag name is reused as another identifier name

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
5.5	No object or function identifier with a static storage duration should be reused.	{ static identifier/parameter name }'%s' should not be reused. (already used as {static identifier/parameter name } with static storage duration at %s:%d)	Warning when a static name is reused as another identifier name
5.6	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.	{member name }'%s' should not be reused. (already used as { member name } at %s:%d)	Warning when a idf in a namespace is reused in another namespace
5.7	No identifier name should be reused.	{identifier}'%s' should not be reused. (already used as { identifier} at %s:%d)	No violation reported when: • Different functions have parameters with the same name • Different functions have local variables with the same name • A function has a local variable that has the same name as a parameter of another function

Types

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
6.1	The plain char type shall be used only for the storage and use of character values	Only permissible operators on plain chars are '=', '==' or '!=' operators, explicit casts to integral types and '?' (for the 2nd and 3rd operands)	Warning when a plain char is used with an operator other than =, ==, !=, explicit casts to integral types, or as the second or third operands of the ? operator.
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.	• Value of type plain char is implicitly converted to signed char. • Value of type plain char is implicitly converted to unsigned char. • Value of type signed char is implicitly converted to plain char. • Value of type unsigned char is implicitly converted to plain char.	Warning if value of type plain char is implicitly converted to value of type signed char or unsigned char.
6.3	*typedefs* that indicate size and signedness should be used in place of the basic types	typedefs that indicate size and signedness should be used in place of the basic types.	No warning is given in typedef definition.
6.4	Bit fields shall only be defined to be of type *unsigned int* or *signed int*.	Bit fields shall only be defined to be of type unsigned int or signed int.	
6.5	Bit fields of type *signed int* shall be at least 2 bits long.	Bit fields of type signed int shall be at least 2 bits long.	No warning on anonymous signed int bitfields of width 0 - Extended to all signed bitfields of size <= 1 (if Rule **6.4** is violated).

Constants

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.	• Octal constants other than zero and octal escape sequences shall not be used. • Octal constants (other than zero) should not be used. • Octal escape sequences should not be used.	

Declarations and Definitions

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	• Function XX has no complete prototype visible at call. • Function XX has no prototype visible at definition.	Prototype visible at call must be complete.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated	Whenever an object or function is declared or defined, its type shall be explicitly stated.	
8.3	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.	Definition of function 'XX' incompatible with its declaration.	Assumes that rule 8.1 is not violated. The rule is restricted to compatible types. Can be turned to Off

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
8.4	If objects or functions are declared more than once their types shall be compatible.	• If objects or functions are declared more than once their types shall be compatible. • Global declaration of 'XX' function has incompatible type with its definition. • Global declaration of 'XX' variable has incompatible type with its definition.	Violations of this rule might be generated during the link phase.
8.5	There shall be no definitions of objects or functions in a header file	• Object 'XX' should not be defined in a header file. • Function 'XX' should not be defined in a header file. • Fragment of function should not be defined in a header file.	Tentative of definitions are considered as definitions.
8.6	Functions shall always be declared at file scope.	Function 'XX' should be declared at file scope.	
8.7	Objects shall be defined at block scope if they are only accessed from within a single function	Object 'XX' should be declared at block scope.	Restricted to static objects.
8.8	An external object or function shall be declared in one file and only one file	Function/Object 'XX' has external declarations in multiples files.	Restricted to explicit extern declarations (tentative of definitions are ignored).

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
8.9	Definition: An identifier with external linkage shall have exactly one external definition.	• Procedure/Global variable XX multiply defined. • Forbidden multiple tentative of definition for object XX • Global variable has multiples tentative of definitions • Undefined global variable XX	Tentative of definitions are considered as definitions, no warning on predefined symbols.
8.10	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required	Function/Variable XX should have internal linkage.	Assumes that 8.1 is not violated. No warning if 0 uses.
8.11	The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage	static storage class specifier should be used on internal linkage symbol XX.	
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization	Array XX has unknown size.	

Initialization

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
9.1	All automatic variables shall have been assigned a value before being used.		Checked during code analysis. Violations displayed as Non-initialized variable results.
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	

Arithmetic Type Conversion

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: • it is not a conversion to a wider integer type of the same signedness, or • the expression is complex, or	• Implicit conversion of the expression of underlying type XX to the type XX that is not a wider integer type of the same signedness. • Implicit conversion of one of the binary operands whose underlying types are XX and XX	1 ANSI C base types order (signed char, short, int, long) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1. The same interpretation is applied on the unsigned version of base types.

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
	• the expression is not constant and is a function argument, or • the expression is not constant and is a return expression	• Implicit conversion of the binary right hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary left hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary right hand operand of underlying type XX to XX that is not a wider integer type of the same signedness or Implicit conversion of the binary ? left hand operand of underlying type XX to XX, but it is a complex expression. • Implicit conversion of complex integer expression of underlying type XX to XX. • Implicit conversion of non-constant integer expression of underlying type XX in function return whose expected type is XX. • Implicit conversion of non-constant integer expression of underlying	**2** An expression of bool or enum types has int as underlying type. **3** Plain char may have signed or unsigned underlying type (depending on Polyspace target configuration or option setting). **4** The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width is not token into account and it assumes that only signed | unsigned int are used for bitfield (Rule 6.4). **5** No violation reported when: • The implicit conversion is a type widening, without change of signedness if integer • The expression is an argument expression or a return expression **6** No violation reported when the following are all true:

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
		type XX as argument of function whose corresponding parameter type is XX.	• Implicit conversion applies to a constant expression and is a type widening, with a possible change of signedness if integer • The conversion does not change the representation of the constant value or the result of the operation • The expression is an argument expression or a return expression or an operand expression of a non-bitwise operator
10.2	The value of an expression of floating type shall not be implicitly converted to a different type if • it is not a conversion to a wider floating type, or • the expression is complex, or • the expression is a function argument, or • the expression is a return expression	• Implicit conversion of the expression from XX to XX that is not a wider floating type. • Implicit conversion of the binary ? right hand operand from XX to XX, but it is a complex expression. • Implicit conversion of the binary ? right hand operand from XX to XX that is not a wider floating type or Implicit conversion of the binary ? left hand operand from	ANSI C base types order (float, double) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1. No violation reported when: • The implicit conversion is a type widening • The expression is an argument expression or a return expression.

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
		XX to XX, but it is a complex expression. • Implicit conversion of complex floating expression from XX to XX. • Implicit conversion of floating expression of XX type in function return whose expected type is XX. • Implicit conversion of floating expression of XX type as argument of function whose corresponding parameter type is XX.	
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression	Complex expression of underlying type XX may only be cast to narrower integer type of same signedness, however the destination type is XX.	• ANSI C base types order (signed char, short, int, long) defines that T1 is narrower than T2 if T2 is on the right hand of T1 or T1 = T2. The same methodology is applied on the unsigned version of base types. • An expression of bool or enum types has int as underlying type. • Plain char may have signed or unsigned underlying type (depending on target

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
			configuration or option setting). • The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width is not token into account and it assumes that only signed, unsigned int are used for bitfield (Rule 6.4).
10.4	The value of a complex expression of float type may only be cast to narrower floating type	Complex expression of XX type may only be cast to narrower floating type, however the destination type is XX.	ANSI C base types order (float, double) defines that T1 is narrower than T2 if T2 is on the right hand of T1 or T2 = T1.
10.5	If the bitwise operator ~ and << are applied to an operand of underlying type *unsigned char* or *unsigned short*, the result shall be immediately cast to the underlying type of the operand	Bitwise [<<|~] is applied to the operand of underlying type [unsigned char|unsigned short], the result shall be immediately cast to the underlying type.	
10.6	The “U” suffix shall be applied to all constants of *unsigned* types	No explicit ’U suffix on constants of an unsigned type.	Warning when the type determined from the value and the base (octal, decimal or hexadecimal) is unsigned and there is no suffix `u` or `U`. For example, when the size of the `int` and `long int` data types is 32 bits, the coding rule checker will

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
			report a violation of rule 10.6 for the following line: `int a = 2147483648;` There is a difference between decimal and hexadecimal constants when `int` and `long int` are not the same size.

Pointer Type Conversion

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type	Conversion shall not be performed between a pointer to a function and any type other than an integral type.	Casts and implicit conversions involving a function pointer. Casts or implicit conversions from `NULL` or `(void*)0` do not give any warning.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.	There is also a warning on qualifier loss
11.3	A cast should not be performed between a pointer type and an integral type	A cast should not be performed between a pointer type and an integral type.	Exception on zero constant. Extended to all conversions

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.	A cast should not be performed between a pointer to object type and a different pointer to object type.	
11.5	A cast shall not be performed that removes any *const* or *volatile* qualification from the type addressed by a pointer	A cast shall not be performed that removes any *const* or *volatile* qualification from the type addressed by a pointer	Extended to all conversions

Expressions

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
12.1	Limited dependence should be placed on C's operator precedence rules in expressions	Limited dependence should be placed on C's operator precedence rules in expressions	
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits.	• The value of 'sym' depends on the order of evaluation. • The value of volatile 'sym' depends on the order of evaluation because of multiple accesses.	The expression is a simple expression of symbols (Unlike i = i++; no detection on tab[2] = tab[2]++;). Rule 12.2 check assumes that no assignment in expressions that yield a Boolean values (rule 13.1) and the comma operator is not used (rule 12.10).
12.2	The `sizeof` operator should not be used on expressions that contain side effects.	The `sizeof` operator should not be used on expressions that contain side effects.	No warning on volatile accesses

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
12.4	The right hand operand of a logical && or || operator shall not contain side effects.	The right hand operand of a logical && or || operator shall not contain side effects.	No warning on volatile accesses
12.5	The operands of a logical && or || shall be primary-expressions.	• operand of logical && is not a primary expression • operand of logical || is not a primary expression • The operands of a logical && or || shall be primary-expressions.	During preprocessing, violations of this rule are detected on the expressions in #if directives. Allowed exception on associatively (a && b && c), (a || b || c).

12.6	Operands of logical operators (&&, || and !) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (&&, || or !).	• Operand of '!' logical operator should be effectively Boolean.
		• Left operand of '%s' logical operator should be effectively Boolean.
		• Right operand of '%s' logical operator should be effectively Boolean.
		• %s operand of '%s' is effectively Boolean. Boolean should not be used as operands to operators other than '&&', '||', '!', '=', '==', '!=' and '?:'.
			The operand of a logical operator should be a Boolean data type. Although the C standard does not explicitly define the Boolean data type, the standard implicitly assumes the use of the Boolean data type.
			Some operators may return Boolean-like expressions, for example, `(var == 0)`.
			Consider the following code:
			`unsigned char flag;`
			`if (!flag)`
			The rule checker reports a violation of rule 12.6:
			`Operand of '!' logical operator should be effectively Boolean.`

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification

			The operand `flag` is not a Boolean but an `unsigned char`.
			To be compliant with rule 12.6, the code must be rewritten either as
			`if (!( flag != 0))`
			or
			`if (flag == 0)`
			The use of the option `-boolean-types` may increase or decrease the number of warnings generated.

12.7	Bitwise operators shall not be applied to operands whose underlying type is signed	• [~/Left Shift/Right shift/&] operator applied on an expression whose underlying type is signed. • Bitwise ~ on operand of signed underlying type XX. • Bitwise [<<|>>] on left hand operand of signed underlying type XX. • Bitwise [& | ^] on two operands of s	The underlying type for an integer is signed when: • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
12.8	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	• shift amount is negative • shift amount is bigger than 64 • Bitwise [<< >>] count out of range [0 ..X] (width of the underlying type XX of the left hand operand - 1)..	The numbers that are manipulated in preprocessing directives are 64 bits wide so that valid shift range is between 0 and 63 Check is also extended onto bitfields with the field width or the width of the base type when it is within a complex expression
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	• Unary - on operand of unsigned underlying type XX. • Minus operator applied to an expression whose underlying type is unsigned	The underlying type for an integer is signed when: • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number
12.10	The comma operator shall not be used.	The comma operator shall not be used.	
12.11	Evaluation of constant unsigned expression should not lead to wraparound.	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification

12.12	The underlying bit representations of floating-point values shall not be used.	The underlying bit representations of floating-point values shall not be used.	Warning when:
			
			• A float pointer is cast as a pointer to another data type. Casting a float pointer as a pointer to `void` does not generate a warning.
			• A float is packed with another data type. For example:
			
			`union {`
			`  float f;`
			`  int i;`
			`}`

12.13	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	Warning when ++ or -- operators are not used alone.

Control Statement Expressions

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
13.1	Assignment operators shall not be used in expressions that yield Boolean values.	Assignment operators shall not be used in expressions that yield Boolean values.	
13.2	Tests of a value against zero should be made explicit,	Tests of a value against zero should be made explicit,	No warning is given on integer constants. Example: if (2)

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
	unless the operand is effectively Boolean	unless the operand is effectively Boolean	The use of the option `-boolean-types` may increase or decrease the number of warnings generated.
13.3	Floating-point expressions shall not be tested for equality or inequality.	Floating-point expressions shall not be tested for equality or inequality.	Warning on directs tests only.
13.4	The controlling expression of a *for* statement shall not contain any objects of floating type	The controlling expression of a for statement shall not contain any objects of floating type	If *for* index is a variable symbol, checked that it is not a float.
13.5	The three expressions of a *for* statement shall be concerned only with loop control	• 1st expression should be an assignment. • Bad type for loop counter (XX). • 2nd expression should be a comparison. • 2nd expression should be a comparison with loop counter (XX). • 3rd expression should be an assignment of loop counter (XX). • 3rd expression: assigned variable should be the loop counter (XX). • The following kinds of for loops are allowed:	Checked if the for loop index (V) is a variable symbol; checked if V is the last assigned variable in the first expression (if present). Checked if, in first expression, if present, is assignment of V; checked if in 2nd expression, if present, must be a comparison of V; Checked if in 3rd expression, if present, must be an assignment of V.

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
		(a) all three expressions shall be present; (b) the 2nd and 3rd expressions shall be present with prior initialization of the loop counter; (c) all three expressions shall be empty for a deliberate infinite loop.	
13.6	Numeric variables being used within a *for* loop for iteration counting should not be modified in the body of the loop.	Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop.	Detect only direct assignments if the for loop index is known and if it is a variable symbol.
13.7	Boolean operations whose results are invariant shall not be permitted	• Boolean operations whose results are invariant shall not be permitted. Expression is always true. • Boolean operations whose results are invariant shall not be permitted. Expression is always false. • Boolean operations whose results are invariant shall not be permitted.	During compilation, check comparisons with at least one constant operand.

Control Flow

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
14.1	There shall be no unreachable code.	There shall be no unreachable code.	
14.2	All non-null statements shall either have at lest one side effect however executed, or cause control flow to change	• All non-null statements shall either: • have at lest one side effect however executed, or • cause control flow to change	
14.3	All non-null statements shall either • have at lest one side effect however executed, or • cause control flow to change	A null statement shall appear on a line by itself	We assume that a ';' is a null statement when it is the first character on a line (excluding comments). The rule is violated when: • there are some comments before it on the same line. • there is a comment immediately after it • there is something else than a comment after the ';' on the same line.
14.4	The *goto* statement shall not be used.	The goto statement shall not be used.	
14.5	The *continue* statement shall not be used.	The continue statement shall not be used.	
14.6	For any iteration statement there shall be at most one *break* statement used for loop termination	For any iteration statement there shall be at most one break statement used for loop termination	

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
14.7	A function shall have a single point of exit at the end of the function	A function shall have a single point of exit at the end of the function	
14.8	The statement forming the body of a *switch, while, do while* or *for* statement shall be a compound statement	• The body of a do while statement shall be a compound statement. • The body of a for statement shall be a compound statement. • The body of a switch statement shall be a compound statement	
14.9	An *if (expression)* construct shall be followed by a compound statement. The *else* keyword shall be followed by either a compound statement, or another *if* statement	• An if (expression) construct shall be followed by a compound statement. • The else keyword shall be followed by either a compound statement, or another if statement	
14.10	All *if else if* constructs should contain a final *else* clause.	All if else if constructs should contain a final else clause.	

Switch Statements

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification

15.0	Unreachable code is detected between switch statement and first case.
	
	Note This is not a MISRA C2004 rule.
		switch statements syntax normative restrictions.	Warning on declarations or any statements before the first switch case.
			
			Warning on label or jump statements in the body of switch cases.
			
			On the following example, the rule is displayed in the log file at line 3:
			
			`1 ...`
			`2 switch(index) {`
			`3  var = var + 1;`
			`// RULE 15.0`
			`// violated`
			`4case 1: ...`
			
			The code between switch statement and first case is checked as dead code by Polyspace. It follows ANSI standard behavior.

15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	
15.2	An unconditional *break* statement shall terminate every non-empty switch clause	An unconditional break statement shall terminate every non-empty switch clause	Warning for each non-compliant case clause.

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
15.3	The final clause of a *switch* statement shall be the *default* clause	The final clause of a switch statement shall be the default clause	
15.4	A *switch* expression should not represent a value that is effectively Boolean	A switch expression should not represent a value that is effectively Boolean	The use of the option `-boolean-types` may increase the number of warnings generated.
15.5	Every *switch* statement shall have at least one *case* clause	Every switch statement shall have at least one case clause	

Functions

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
16.1	Functions shall not be defined with variable numbers of arguments.	Function XX should not be defined as varargs.	
16.2	Functions shall not call themselves, either directly or indirectly.	Function %s should not call itself.	Done by Polyspace software (Call graph in the Results Manager perspective gives the information). Polyspace also checks that partially during compilation phase.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.	Identifiers shall be given for all of the parameters in a function prototype declaration.	Assumes Rule **8.6** is not violated.
16.4	The identifiers used in the declaration and definition of a function shall be identical.	The identifiers used in the declaration and definition of a function shall be identical.	Assumes that rules **8.8**, **8.1** and **16.3** are not violated. All occurrences are detected.

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
16.5	Functions with no parameters shall be declared with parameter type *void*.	Functions with no parameters shall be declared with parameter type void.	Definitions are also checked.
16.6	The number of arguments passed to a function shall match the number of parameters.	• Too many arguments to XX. • Insufficient number of arguments to XX.	Assumes that rule **8.1** is not violated.
16.7	A pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.	Pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.	Warning if a non-`const` pointer parameter is either not used to modify the addressed object or is passed to a call of a function that is declared with a `const` pointer parameter.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Missing return value for non-void function XX.	Warning when a non-void function is not terminated with an unconditional return with an expression.
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.	Function identifier XX should be preceded by a & or followed by a parameter list.	
16.10	If a function returns error information, then that error information shall be tested.	If a function returns error information, then that error information shall be tested.	Warning if a non-`void` function is called and the returned value is ignored.No warning if the result of the call is cast to `void`. No check performed for calls of `memcpy`, `memmove`,

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
			`memset`, `strcpy`, `strncpy`, `strcat`, or `strncat`.

Pointers and Arrays

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
17.1	Pointer arithmetic shall only be applied to pointers that address an array or array element.	Pointer arithmetic shall only be applied to pointers that address an array or array element.	
17.2	Pointer subtraction shall only be applied to pointers that address elements of the same array	Pointer subtraction shall only be applied to pointers that address elements of the same array.	
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	
17.4	Array indexing shall be the only allowed form of pointer arithmetic.	Array indexing shall be the only allowed form of pointer arithmetic.	Warning on operations on pointers. (`p+I`, `I+p` and `p-I`, where `p` is a pointer and `I` an integer).
17.5	A type should not contain more than 2 levels of pointer indirection	A type should not contain more than 2 levels of pointer indirection	
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.	Pointer to a parameter is an illegal return value. Pointer to a local is an illegal return value.	Warning when assigning address to a global variable, returning a local variable address, or returning a parameter address.

Structures and Unions

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
18.1	All structure or union types shall be complete at the end of a translation unit.	All structure or union types shall be complete at the end of a translation unit.	Warning for all incomplete declarations of structs or unions.
18.2	An object shall not be assigned to an overlapping object.	• An object shall not be assigned to an overlapping object. • Destination and source of XX overlap, the behavior is undefined.	
18.4	Unions shall not be used	Unions shall not be used.	

Preprocessing Directives

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
19.1	#include statements in a file shall only be preceded by other preprocessors directives or comments	A message is displayed when a #include directive is preceded by other things than preprocessor directives, comments, spaces or “new lines”.	
19.2	Nonstandard characters should not occur in header file names in #include directives	• A message is displayed on characters ’, \, " or /* between < and > in #include <filename> • A message is displayed on characters ’, \or /* between " and " in #include "filename"	

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
19.3	The #*include* directive shall be followed by either a <filename> or "filename" sequence.	• '#include' expects "FILENAME" or <FILENAME> • '#include_next' expects "FILENAME" or <FILENAME>	
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.	Macro '<name>' does not expand to a compliant construct.	We assume that a macro definition does not violate this rule when it expands to: • a braced construct (not necessarily an initializer) • a parenthesized construct (not necessarily an expression) • a number • a character constant • a string constant (can be the result of the concatenation of string field arguments and literal strings) • the following keywords: typedef, extern, static, auto, register, const, volatile, __asm__ and __inline__ • a do-while-zero construct

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
19.5	Macros shall not be #defined and #undefd within a block.	• Macros shall not be #defined within a block. • Macros shall not be #undef'd within a block.	
19.6	#undef shall not be used.	#undef shall not be used.	
19.7	A function should be used in preference to a function like-macro.	Message on all function-like macros expansions	
19.8	A function-like macro shall not be invoked without all of its arguments	• arguments given to macro '<name>' • macro '<name>' used without args. • macro '<name>' used with just one arg. • macro '<name>' used with too many (<number>) args.	
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	Macro argument shall not look like a preprocessing directive.	This rule is detected as violated when the '#' character appears in a macro argument (outside a string or character constant)

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.	Parameter instance shall be enclosed in parentheses.	If `x` is a macro parameter, the following instances of x as an operand of the # and ## operators do not generate a warning: `#x`, `##x`, and `x##`. Otherwise, parentheses are required around `x`. The software does not generate a warning if a parameter is reused as an argument of a function or function-like macro. For example, consider a parameter x. The software does not generate a warning if `x` appears as `(x)` or `(x,` or `,x)` or `,x,`.
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	'<name>' is not defined.	
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.	More than one occurrence of the # or ## preprocessor operators.	
19.13	The # and ## preprocessor operators should not be used	Message on definitions of macros using # or ## operators	

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.	'defined' without an identifier.	

19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.	Precautions shall be taken in order to prevent multiple inclusions.	When a header file is formatted as:
			`#ifndef <control macro>`
			`#define <control macro>`
			`<contents> #endif`
			
			or:
			`#ifdef <control macro>`
			`#error ...`
			`#else`
			`#define <control macro>`
			`<contents> #endif`
			
			it is assumed that precautions have been taken to prevent multiple inclusions. Otherwise, a violation of this MISRA rule is detected.

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	directive is not syntactically meaningful.	
19.17	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	• '#elif' not within a conditional. • '#else' not within a conditional. • '#elif' not within a conditional. • '#endif' not within a conditional. • unbalanced '#endif'. • unterminated '#if' conditional. • unterminated '#ifdef' conditional. • unterminated '#ifndef' conditional.	

Standard Libraries

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
20.1	Reserved identifiers, macros and functions in the standard library, shall	• The macro '<name> shall not be redefined.	

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
	not be defined, redefined or undefined.	• The macro '<name> shall not be undefined.	
20.2	The names of standard library macros, objects and functions shall not be reused.	Identifier XX should not be used.	In case a macro whose name corresponds to a standard library macro, object or function is defined, the rule that is detected as violated is **20.1**. Tentative of definitions are considered as definitions.
20.3	The validity of values passed to library functions shall be checked.	Validity of values passed to library functions shall be checked	Warning for argument in library function call if the following are all true: • Argument is a local variable • Local variable is not tested between last assignment and call to the library function • Library function is a common mathematical function • Corresponding parameter of the library function has a restricted input domain. The library function can be one of the following : `sqrt`, `tan`, `pow`, `log`, `log10`, `fmod`, `acos`, `asin`, `acosh`, `atanh`, or `atan2`.

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
20.4	Dynamic heap memory allocation shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the dynamic heap memory allocation functions are actually macros and the macro is expanded in the code, this rule is detected as violated. Assumes rule **20.2** is not violated.
20.5	The error indicator errno shall not be used	The error indicator errno shall not be used	Assumes that rule **20.2** is not violated
20.6	The macro *offsetof*, in library <stddef.h>, shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	Assumes that rule **20.2** is not violated
20.7	The *setjmp* macro and the *longjmp* function shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the longjmp function is actually a macro and the macro is expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.8	The signal handling facilities of <signal.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case some of the signal functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.9	The input/output library <stdio.h> shall not be used in production code.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the input/output library functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
20.10	The library functions atof, atoi and toll from library <stdlib.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the atof, atoi and atoll functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.11	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the abort, exit, getenv and system functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.12	The time handling functions of library <time.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the time handling functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated

Runtime Failures

N.	MISRA Definition	Messages in report file	Detailed Polyspace Specification
21.1	Minimization of runtime failures shall be ensured by the use of at least one of: • static verification tools/techniques; • dynamic verification tools/techniques; • explicit coding of checks to handle runtime faults.		Done by Polyspace

MISRA C:2004 Rules Not Checked

The Polyspace coding rules checker does not check the following MISRA C:2004 coding rules. These rules cannot be enforced because they are outside the scope of Polyspace software. They may concern documentation, dynamic aspects, or functional aspects of MISRA rules. The "**Comments**" column describes the reason each rule is not checked.

Environment

Rule	Description	Comments
1.2 (Required)	No reliance shall be placed on undefined or unspecified behavior	Not statically checkable unless the data dynamic properties is taken into account
1.3 (Required)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the language/compilers/assemblers conform.	It is a process rule method.
1.4 (Required)	The compiler/linker/Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	The documentation of compiler must be checked.
1.5 (Advisory)	Floating point implementations should comply with a defined floating point standard.	The documentation of compiler must be checked as this implementation is done by the compiler

Language Extensions

Rule	Description	Comments
2.4 (Advisory)	Sections of code should not be "commented out"	It might be some pseudo code or code that does not compile inside a comment.

Documentation

Rule	Description	Comments
3.1 (Required)	All usage of implementation-defined behavior shall be documented.	The documentation of compiler must be checked. Error detection is based on undefined behavior, according to choices made for implementation-defined constructions. Documentation can not be checked.
3.2 (Required)	The character set and the corresponding encoding shall be documented.	The documentation of compiler must be checked.
3.3 (Advisory)	The implementation of integer division in the chosen compiler should be determined, documented and taken into account.	The documentation of compiler must be checked.

Rule	Description	Comments
3.5 (Required)	The implementation-defined behavior and packing of bitfields shall be documented if being relied upon.	The documentation of compiler must be checked.
3.6 (Required)	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.	The documentation of compiler must be checked.

Structures and Unions

Rule	Description	Comments
18.3 (Required)	An area of memory shall not be reused for unrelated purposes.	"purpose" is functional design issue.

Polyspace MISRA C++ Checker

The Polyspace MISRA C++ checker helps you comply with the MISRA C++:2008 coding standard.[9]

When MISRA C++ rules are violated, the Polyspace MISRA C++ checker enables Polyspace software to provide messages with information about the rule violations. Most messages are reported during the compile phase of an analysis. The MISRA C++ checker can check 185 of the 228 MISRA C++ coding rules.

There are subsets of MISRA C++ coding rules that can have a direct or indirect impact on the selectivity (reliability percentage) of your results. When you set up rule checking, you can select these subsets directly. These subsets are defined in “Software Quality Objective Subsets (C++)” on page 3-60.

Note The Polyspace MISRA C++ checker is based on MISRA C++:2008 – “Guidelines for the use of the C++ language in critical systems." For more information on these coding standards, see http://www.misra-cpp.com.

9. MISRA is a registered trademark of MISRA Ltd., held on behalf of the MISRA Consortium.

Software Quality Objective Subsets (C++)

In this section...

"SQO Subset 1 – Direct Impact on Selectivity" on page 3-60
"SQO Subset 2 – Indirect Impact on Selectivity" on page 3-63

SQO Subset 1 – Direct Impact on Selectivity

The following set of coding rules will typically improve the selectivity of your results.

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	The One Definition Rule shall not be violated.
3-9-3	The underlying bit representations of floating-point values shall not be used.
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.

MISRA C++ Rule	Description
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
7-5-4	Functions should not call themselves, either directly or indirectly.
8-4-1	Functions shall not be defined using the ellipsis notation.
9-5-1	Unions shall not be used.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.

MISRA C++ Rule	Description
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-3	An empty throw (throw;) shall only be used in the compound-statement of a catch handler.
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
18-4-1	Dynamic heap memory allocation shall not be used.

SQO Subset 2 – Indirect Impact on Selectivity

Good design practices generally lead to less code complexity, which can improve the selectivity of your results. The following set of coding rules may help to address design issues that impact selectivity.

Note When you specify `SQO-subset2` for your MISRA C++ rules configuration, the software checks the rules listed in SQO Subset 1 *and* SQO Subset 2.

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.
3-9-2	typedefs that indicate size and signedness should be used in place of the basic numerical types.
3-9-3	The underlying bit representations of floating-point values shall not be used.
4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.
5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.
5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions.
5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression.

MISRA C++ Rule	Description
5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.
5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.
5-0-10	If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.
5-0-13	
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-2-1	Each operand of a logical && or || shall be a postfix - expression.
5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.
5-2-5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.
5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.
5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
5-2-11	The comma operator, && operator and the || operator shall not be overloaded.

MISRA C++ Rule	Description
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
5-3-3	The unary & operator shall not be overloaded.
5-18-1	The comma operator shall not be used.
6-2-1	Assignment operators shall not be used in sub-expressions.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.
6-4-2	All if ... else if constructs shall be terminated with an else clause.
6-4-6	The final clause of a switch statement shall be the default-clause.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.

MISRA C++ Rule	Description
6-6-5	A function shall have a single point of exit at the end of the function.
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
7-5-4	Functions should not call themselves, either directly or indirectly.
8-4-1	Functions shall not be defined using the ellipsis notation.
8-4-3	All exit paths from a function with non- void return type shall have an explicit return statement with an expression.
8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.
8-5-2	Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.
8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.
11-0-1	Member data in non- POD class types shall be private.

MISRA C++ Rule	Description
12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.
12-8-2	The copy assignment operator shall be declared protected or private in an abstract class.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

MISRA C++ Rule	Description
16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.
16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.
16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.
16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.
18-4-1	Dynamic heap memory allocation shall not be used.

MISRA C++ Coding Rules

In this section...

"Supported MISRA C++ Coding Rules" on page 3-69

"MISRA C++ Rules Not Checked" on page 3-89

Supported MISRA C++ Coding Rules

- "Language Independent Issues" on page 3-70
- "General" on page 3-70
- "Lexical Conventions" on page 3-70
- "Basic Concepts" on page 3-72
- "Standard Conversions" on page 3-73
- "Expressions" on page 3-74
- "Statements" on page 3-77
- "Declarations" on page 3-80
- "Declarators" on page 3-81
- "Classes" on page 3-82
- "Derived Classes" on page 3-82
- "Member Access Control" on page 3-83
- "Special Member Functions" on page 3-83
- "Templates" on page 3-84
- "Exception Handling" on page 3-85
- "Preprocessing Directives" on page 3-86
- "Library Introduction" on page 3-88
- "Language Support Library" on page 3-88
- "Diagnostic Library" on page 3-89
- "Input/output Library" on page 3-89

Language Independent Issues

N.	MISRA Definition	Comments
0-1-1	A project shall not contain unreachable code.	
0-1-2	A project shall not contain infeasible paths.	
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.	
0-1-10	Every defined function shall be called at least once.	Detects if static functions are not called in their translation unit. Other cases are detected by the software.

General

N.	MISRA Definition	Comments
1-0-1	All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".	

Lexical Conventions

N.	MISRA Definition	Comments
2-3-1	Trigraphs shall not be used.	
2-5-1	Digraphs should not be used.	
2-7-1	The character sequence /* shall not be used within a C-style comment.	This rule cannot be annotated in the source code.
2-10-1	Different identifiers shall be typographically unambiguous.	

N.	MISRA Definition	Comments
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.	No detection for logical scopes: fields or member functions hiding outer scopes identifiers or hiding ancestors members.
2-10-3	A typedef name (including qualification, if any) shall be a unique identifier.	No detection across namespaces.
2-10-4	A class, union or enum name (including qualification, if any) shall be a unique identifier.	No detection across namespaces.
2-10-5	The identifier name of a non-member object or function with static storage duration should not be reused.	For functions the detection is only on the definition where there is a declaration.
2-10-6	If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.	If the identifier is a function and the function is both declared and defined then the violation is reported only once.
2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.	
2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.	
2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.	
2-13-4	Literal suffixes shall be upper case.	
2-13-5	Narrow and wide string literals shall not be concatenated.	

Basic Concepts

N.	MISRA Definition	Comments
3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.	
3-1-2	Functions shall not be declared at block scope.	
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.	
3-2-1	All declarations of an object or function shall have compatible types.	
3-2-2	The One Definition Rule shall not be violated.	Report type, template, and inline function defined in source file
3-2-3	A type, object or function that is used in multiple translation units shall be declared in one and only one file.	
3-2-4	An identifier with external linkage shall have exactly one definition.	
3-3-1	Objects or functions with external linkage shall be declared in a header file.	
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.	
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.	
3-9-1	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.	Comparison is done between current declaration and last seen declaration.

N.	MISRA Definition	Comments
3-9-2	typedefs that indicate size and signedness should be used in place of the basic numerical types.	No detection in non-instantiated templates.
3-9-3	The underlying bit representations of floating-point values shall not be used.	

Standard Conversions

N.	MISRA Definition	Comments
4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.	
4-5-2	Expressions with type enum shall not be used as operands to built- in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.	
4-5-3	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator. N	

Expressions

N.	MISRA Definition	Comments
5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.	
5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions.	
5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type.	Assumes that `ptrdiff_t` is signed integer
5-0-4	An implicit integral conversion shall not change the signedness of the underlying type.	Assumes that `ptrdiff_t` is signed integer If the conversion is to a narrower integer with a different sign then MISRA C++ 5-0-4 takes precedence over MISRA C++ 5-0-6.
5-0-5	There shall be no implicit floating-integral conversions.	This rule takes precedence over 5-0-4 and 5-0-6 if they apply at the same time.
5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type.	If the conversion is to a narrower integer with a different sign then MISRA C++ 5-0-4 takes precedence over MISRA C++ 5-0-6.
5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression.	
5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.	
5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.	

N.	MISRA Definition	Comments
5-0-10	If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	
5-0-11	The plain char type shall only be used for the storage and use of character values.	For numeric data, use a type which has explicit signedness.
5-0-12	Signed char and unsigned char type shall only be used for the storage and use of numeric values.	
5-0-14	The first operand of a conditional-operator shall have type bool.	
5-0-15	Array indexing shall be the only form of pointer arithmetic.	Warning on operations on pointers. (p+I, I+p and p-I, where p is a pointer and I an integer, p[i] accepted).
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.	Report when relational operator are used on pointers types (casts ignored).
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.	
5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type.	
5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type.	
5-2-1	Each operand of a logical && or || shall be a postfix - expression.	During preprocessing, violations of this rule are detected on the expressions in #if directives. Allowed exception on associativity (a && b && c), (a || b || c).
5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.	

N.	MISRA Definition	Comments
5-2-3	Casts from a base class to a derived class should not be performed on polymorphic types.	
5-2-4	C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.	
5-2-5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.	
5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.	No violation if pointer types of operand and target are identical.
5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.	"Extended to all pointer conversions including between pointer to struct object and pointer to type of the first member of the struct type. Indirect conversions through non-pointer type (e.g. int) are not detected."
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.	Exception on zero constants. Objects with pointer type include objects with pointer to function type.
5-2-9	A cast should not convert a pointer type to an integral type.	
5-2-10	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	
5-2-11	The comma operator, && operator and the || operator shall not be overloaded.	
5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer.	

N.	MISRA Definition	Comments
5-3-1	Each operand of the ! operator, the logical && or the logical || operators shall have type bool.	
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	
5-3-3	The unary & operator shall not be overloaded.	
5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects.	No warning on volatile accesses and function calls
5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	
5-14-1	The right hand operand of a logical && or || operator shall not contain side effects.	No warning on volatile accesses and function calls.
5-18-1	The comma operator shall not be used.	
5-19-1	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	

Statements

N.	MISRA Definition	Comments
6-2-1	Assignment operators shall not be used in sub-expressions.	
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.	

N.	MISRA Definition	Comments
6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character.	
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.	
6-4-1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.	
6-4-2	All if ... else if constructs shall be terminated with an else clause.	Detects also cases where the last if is in the block of the last else (same behavior as JSF, stricter than MISRA C). Example: "if … else { if …{}}" raises the rule
6-4-3	A switch statement shall be a well-formed switch statement.	Return statements are considered as jump statements.
6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	
6-4-5	An unconditional throw or break statement shall terminate every non - empty switch-clause.	
6-4-6	The final clause of a switch statement shall be the default-clause.	
6-4-7	The condition of a switch statement shall not have bool type.	
6-4-8	Every switch statement shall have at least one case-clause.	

N.	MISRA Definition	Comments
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.	
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.	
6-5-3	The loop-counter shall not be modified within condition or statement.	Detect only direct assignments if for_index is known (see 6-5-1).
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.	
6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression.	
6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.	
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.	
6-6-2	The goto statement shall jump to a label declared later in the same function body.	
6-6-3	The continue statement shall only be used within a well-formed for loop.	Assumes 6.5.1 to 6.5.6: so it is implemented only for supported 6_5_x rules.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.	
6-6-5	A function shall have a single point of exit at the end of the function.	At most one return not necessarily as last statement for void functions.

Declarations

N.	MISRA Definition	Comments
7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations.	
7-3-2	The identifier main shall not be used for a function other than the global function main.	
7-3-3	There shall be no unnamed namespaces in header files.	
7-3-4	using-directives shall not be used.	
7-3-5	Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.	
7-3-6	using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.	
7-4-2	Assembler instructions shall only be introduced using the asm declaration.	
7-4-3	Assembly language shall be encapsulated and isolated.	
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.	
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	

N.	MISRA Definition	Comments
7-5-3	A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.	
7-5-4	Functions should not call themselves, either directly or indirectly.	

Declarators

N.	MISRA Definition	Comments
8-0-1	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or	
8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.	
8-4-1	Functions shall not be defined using the ellipsis notation.	
8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.	
8-4-3	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	
8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.	
8-5-1	All variables shall have a defined value before they are used.	Non-initialized variable in results and error messages for obvious cases

N.	MISRA Definition	Comments
8-5-2	Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.	
8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	

Classes

N.	MISRA Definition	Comments
9-3-1	const member functions shall not return non-const pointers or references to class-data.	Class-data for a class is restricted to all non-static member data.
9-3-2	Member functions shall not return non-const handles to class-data.	Class-data for a class is restricted to all non-static member data.
9-5-1	Unions shall not be used.	
9-6-2	Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.	
9-6-3	Bit-fields shall not have enum type.	
9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit.	

Derived Classes

N.	MISRA Definition	Comments
10-1-1	Classes should not be derived from virtual bases.	
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.	Assumes 10.1.1 not required

N.	MISRA Definition	Comments
10-1-3	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.	
10-2-1	All accessible entity names within a multiple inheritance hierarchy should be unique.	No detection between entities of different kinds (member functions against data members, ...).
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.	Member functions that are virtual by inheritance are also detected.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.	
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.	

Member Access Control

N.	MISRA Definition	Comments
11-0-1	Member data in non- POD class types shall be private.	

Special Member Functions

N.	MISRA Definition	Comments
12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.	
12-1-2	All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.	

N.	MISRA Definition	Comments
12-1-3	All constructors that are callable with a single argument of fundamental type shall be declared explicit.	
12-8-1	A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member.	
12-8-2	The copy assignment operator shall be declared protected or private in an abstract class.	

Templates

N.	MISRA Definition	Comments
14-5-2	A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.	
14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.	
14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->	
14-6-2	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.	
14-7-3	All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.	

N.	MISRA Definition	Comments
14-8-1	Overloaded function templates shall not be explicitly specialized.	All specializations of overloaded templates are rejected even if overloading occurs after the call.
14-8-2	The viable function set for a function call should either contain no function specializations, or only contain function specializations.	

Exception Handling

N.	MISRA Definition	Comments
15-0-2	An exception object should not have pointer type.	NULL not detected (see 15-1-2).
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.	
15-1-2	NULL shall not be thrown explicitly.	
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.	
15-3-2	There should be at least one exception handler to catch all otherwise unhandled exceptions.	Detect that there is no try/catch in the main and that the catch does not handle all exceptions. Not detected if no "main".
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.	
15-3-5	A class type exception shall always be caught by reference.	

N.	MISRA Definition	Comments
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.	
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.	
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.	
15-5-1	A class destructor shall not exit with an exception.	Limit detection to throw and catch that are internals to the destructor; rethrows are partially processed; no detections in nested handlers.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).	Limit detection to throw that are internals to the function; rethrows are partially processed; no detections in nested handlers.

Preprocessing Directives

N.	MISRA Definition	Comments
16-0-1	#include directives in a file shall only be preceded by other preprocessor directives or comments.	
16-0-2	Macros shall only be #define 'd or #undef 'd in the global namespace.	
16-0-3	#undef shall not be used.	
16-0-4	Function-like macros shall not be defined.	

N.	MISRA Definition	Comments
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	
16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.	
16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.	
16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.	
16-1-1	The defined preprocessor operator shall only be used in one of the two standard forms.	
16-1-2	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	
16-2-1	The preprocessor shall only be used for file inclusion and include guards.	The rule is raised for #ifdef/#define if the file is not an include file.
16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.	
16-2-3	Include guards shall be provided.	
16-2-4	The ', ", /* or // characters shall not occur in a header file name.	
16-2-5	The \ character should not occur in a header file name.	
16-2-6	The #include directive shall be followed by either a <filename> or "filename" sequence.	

N.	MISRA Definition	Comments
16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.	
16-3-2	The # and ## operators should not be used.	

Library Introduction

N.	MISRA Definition	Comments
17-0-1	Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.	
17-0-2	The names of standard library macros and objects shall not be reused.	
17-0-5	The setjmp macro and the longjmp function shall not be used.	

Language Support Library

N.	MISRA Definition	Comments
18-0-1	The C library shall not be used.	
18-0-2	The library functions atof, atoi and atol from library <cstdlib> shall not be used.	
18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.	The option -dialect iso must be used to detect violations (e.g.:exit).
18-0-4	The time handling functions of library <ctime> shall not be used.	
18-0-5	The unbounded functions of library <cstring> shall not be used.	
18-2-1	The macro offsetof shall not be used.	

N.	MISRA Definition	Comments
18-4-1	Dynamic heap memory allocation shall not be used.	
18-7-1	The signal handling facilities of <csignal> shall not be used.	

Diagnostic Library

N.	MISRA Definition	Comments
19-3-1	The error indicator errno shall not be used.	

Input/output Library

N.	MISRA Definition	Comments
27-0-1	The stream input/output library <cstdio> shall not be used.	

MISRA C++ Rules Not Checked

- "Language Independent Issues" on page 3-90
- "General" on page 3-91
- "Lexical Conventions" on page 3-91
- "Standard Conversions" on page 3-92
- "Expressions" on page 3-92
- "Declarations" on page 3-92
- "Classes" on page 3-93
- "Templates" on page 3-93
- "Exception Handling" on page 3-94
- "Preprocessing Directives" on page 3-94

- “Library Introduction” on page 3-94

Language Independent Issues

N.	MISRA Definition	Comments
0–1–3	A project shall not contain unused variables.	
0-1-4	A project shall not contain non-volatile POD variables having only one use.	
0-1-5	A project shall not contain unused type declarations.	
0-1-6	A project shall not contain instances of non-volatile variables being given values that are never subsequently used.	
0-1-8	All functions with void return type shall have external side effects.	
0-1-9	There shall be no dead code.	Not checked by the coding rules checker. Can be enforced through detection of dead code during analysis.
0-1-11	There shall be no unused parameters (named or unnamed) in nonvirtual functions.	
0-1-12	There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.	
0-2-1	An object shall not be assigned to an overlapping object.	
0-3-1	Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.	

N.	MISRA Definition	Comments
0-3-2	If a function generates error information, then that error information shall be tested.	
0-4-1	Use of scaled-integer or fixed-point arithmetic shall be documented.	
0-4-2	Use of floating-point arithmetic shall be documented.	
0-4-3	Floating-point implementations shall comply with a defined floating-point standard.	

General

N.	MISRA Definition	Comments
1-0-2	Multiple compilers shall only be used if they have a common, defined interface.	
1-0-3	The implementation of integer division in the chosen compiler shall be determined and documented.	

Lexical Conventions

N.	MISRA Definition	Comments
2-2-1	The character set and the corresponding encoding shall be documented.	
2-7-2	Sections of code shall not be "commented out" using C-style comments.	
2-7-3	Sections of code should not be "commented out" using C++ comments.	

Standard Conversions

N.	MISRA Definition	Comments
4-10-1	ULL shall not be used as an integer value.	
4-10-2	Literal zero (0) shall not be used as the null-pointer-constant.	

Expressions

N.	MISRA Definition	Comments
5-0-13	The condition of an if-statement and the condition of an iteration- statement shall have type bool.	
5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	
5-0-17	Subtraction between pointers shall only be applied to pointers that address elements of the same array.	
5-17-1	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.	

Declarations

N.	MISRA Definition	Comments
7-1-1	A variable which is not modified shall be const qualified.	
7-1-2	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.	

N.	MISRA Definition	Comments
7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	
7-4-1	All usage of assembler shall be documented.	

Classes

N.	MISRA Definition	Comments
9-3-3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.	
9-6-1	When the absolute positioning of bits representing a bit-field is required, then the behavior and packing of bit-fields shall be documented.	

Templates

N.	MISRA Definition	Comments
14-5-1	A non-member generic function shall only be declared in a namespace that is not an associated namespace.	
14-7-1	All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.	
14-7-2	For any given template specialization, an explicit instantiation of the template with the template-arguments used in the specialization shall not render the program ill-formed.	

Exception Handling

N.	MISRA Definition	Comments
15-0-1	Exceptions shall only be used for error handling.	
15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	
15-3-1	Exceptions shall be raised only after start-up and before termination of the program.	
15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to	
15-5-3	The terminate() function shall not be called implicitly.	

Preprocessing Directives

N.	MISRA Definition	Comments
16-6-1	All uses of the #pragma directive shall be documented.	

Library Introduction

N.	MISRA Definition	Comments
17-0-3	The names of standard library functions shall not be overridden.	
17-0-4	All library code shall conform to MISRA C++.	

Polyspace JSF C++ Checker

The Polyspace JSF C++ checker helps you comply with the Joint Strike Fighter Air Vehicle C++ coding standards (JSF++). These coding standards were developed by Lockheed Martin® for the JSF program. They are designed to improve the robustness of C++ code, and improve maintainability.

When JSF++ rules are violated, the Polyspace JSF C++ checker enables Polyspace software to provide messages with information about the rule violations. Most messages are reported during the compile phase of an analysis.

Note The Polyspace JSF C++ checker is based on JSF++:2005. For more information on these coding standards, see http://www.jsf.mil/downloads/documents/JSF_AV_C++_Coding_Standards_Rev_C.doc.

JSF C++ Coding Rules

In this section...

"Supported JSF C++ Coding Rules" on page 3-96

"JSF++ Rules Not Checked" on page 3-121

Supported JSF C++ Coding Rules

- "Code Size and Complexity" on page 3-97
- "Environment" on page 3-97
- "Libraries" on page 3-98
- "Pre-Processing Directives" on page 3-99
- "Header Files" on page 3-100
- "Style" on page 3-100
- "Classes" on page 3-104
- "Namespaces" on page 3-108
- "Templates" on page 3-108
- "Functions" on page 3-108
- "Comments" on page 3-110
- "Declarations and Definitions" on page 3-110
- "Initialization" on page 3-111
- "Types" on page 3-112
- "Constants" on page 3-112
- "Variables" on page 3-112
- "Unions and Bit Fields" on page 3-113
- "Operators" on page 3-113
- "Pointers and References" on page 3-115
- "Type Conversions" on page 3-116

- "Flow Control Standards" on page 3-117
- "Expressions" on page 3-119
- "Memory Allocation" on page 3-120
- "Fault Handling" on page 3-120
- "Portable Code" on page 3-120

Code Size and Complexity

N.	JSF++ Definition	Comments
1	Any one function (or method) **will** contain no more than 200 logical source lines of code (L-SLOCs).	Message in report file: `<function name> has <num> logical source lines of code.`
3	All functions **shall** have a cyclomatic complexity number of 20 or less.	Message in report file: `<function name> has cyclomatic complexity number equal to <num>`

Environment

N.	JSF++ Definition	Comments
8	All code **shall** conform to ISO/IEC 14882:2002(E) standard C++.	Reports the compilation error message
9	Only those characters specified in the C++ basic source character set **will** be used.	
11	Trigraphs **will not** be used.	
12	The following digraphs **will not** be used: `<%`, `%>`, `<:`, `:>`, `%:`, `%:%:`.	Message in report file: `The following digraph will not be used: <digraph>` Reports the digraph. If the rule level is set to warning, the digraph will be allowed even if it is not supported in `-dialect iso`

N.	JSF++ Definition	Comments
13	Multi-byte characters and wide string literals **will not** be used.	Report `L'c'` and `L"string"` and use of `wchar_t`.
14	Literal suffixes **shall** use uppercase rather than lowercase letters.	
15	Provision **shall** be made for run-time checking (defensive programming).	Done with checks in the software.

Libraries

N.	JSF++ Definition	Comments
17	The error indicator `errno` **shall not** be used.	`errno` should not be used as a macro or a global with external "C" linkage.
18	The macro `offsetof`, in library `<stddef.h>`, **shall not** be used.	`offsetof` should not be used as a macro or a global with external "C" linkage.
19	`<locale.h>` and the `setlocale` function **shall not** be used.	`setlocale` and `localeconv` should not be used as a macro or a global with external "C" linkage.
20	The `setjmp` macro and the `longjmp` function **shall not** be used.	`setjmp` and `longjmp` should not be used as a macro or a global with external "C" linkage.
21	The signal handling facilities of `<signal.h>` **shall not** be used.	`signal` and `raise` should not be used as a macro or a global with external "C" linkage.
22	The input/output library `<stdio.h>` **shall not** be used.	all standard functions of `<stdio.h>` should not be used as a macro or a global with external "C" linkage.
23	The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>` **shall not** be used.	`atof`, `atoi` and `atol` should not be used as a macro or a global with external "C" linkage.

N.	JSF++ Definition	Comments
24	The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` **shall not** be used.	`abort`, `exit`, `getenv` and `system` should not be used as a macro or a global with external "C" linkage.
25	The time handling functions of library `<time.h>` **shall not** be used.	`clock`, `difftime`, `mktime`, `asctime`, `ctime`, `gmtime`, `localtime` and `strftime` should not be used as a macro or a global with external "C" linkage.

Pre-Processing Directives

N.	JSF++ Definition	Comments
26	Only the following preprocessor directives **shall** be used: `#ifndef`, `#define`, `#endif`, `#include`.	
27	`#ifndef`, `#define` and `#endif` **will** be used to prevent multiple inclusions of the same header file. Other techniques to prevent the multiple inclusions of header files **will not** be used.	Detects the patterns `#if !defined`, `#pragma once`, `#ifdef`, and missing `#define`.
28	The `#ifndef` and `#endif` preprocessor directives **will** only be used as defined in AV Rule 27 to prevent multiple inclusions of the same header file.	Detects any use that does not comply with AV Rule 27. Assuming 35/27 is not violated, reports only `#ifndef`.
29	The `#define` preprocessor directive **shall not** be used to create inline macros. Inline functions shall be used instead.	Rule is split into two parts: the definition of a macro function (29.def) and the call of a macrofunction (29.use).Messages in report file: • `29.1 : The #define preprocessor directive shall not be used to create inline macros.` • `29.2 : Inline functions shall be used intead of inline macros`

N.	JSF++ Definition	Comments
30	The `#define` preprocessor directive **shall not** be used to define constant values. Instead, the `const` qualifier **shall** be applied to variable declarations to specify constant values.	Reports `#define` of simple constants.
31	The `#define` preprocessor directive **will** only be used as part of the technique to prevent multiple inclusions of the same header file.	Detects use of `#define` that are not used to guard for multiple inclusion, assuming that rules 35 and 27 are not violated.
32	The `#include` preprocessor directive **will** only be used to include header (*.h) files.	

Header Files

N.	JSF++ Definition	Comments
33	The `#include` directive **shall** use the `<filename.h>` notation to include header files.	
35	A header file **will** contain a mechanism that prevents multiple inclusions of itself.	
39	Header files (`*.h`) **will not** contain non-const variable definitions or function definitions.	Reports definitions of global variables / function in header.

Style

N.	JSF++ Definition	Comments
40	Every implementation file shall include the header files that uniquely define the inline functions, types, and templates used.	Reports when type, template, or inline function is defined in source file.
41	Source lines **will** be kept to a length of 120 characters or less.	

N.	JSF++ Definition	Comments
42	Each expression-statement **will** be on a separate line.	Reports when two consecutive expression statements are on the same line.
43	Tabs **should** be avoided.	
44	All indentations will be at least two spaces and be consistent within the same source file.	Reports when a statement indentation is not at least two spaces more than the statement containing it. Does not report bad indentation between opening braces following if/else, do/while, for, and while statements. NB: in final release it will accept any indentation
46	User-specified identifiers (internal and external) **will not** rely on significance of more than 64 characters.	
47	Identifiers **will not** begin with the underscore character '_'.	
48	Identifiers **will not** differ by: • Only a mixture of case • The presence/absence of the underscore character • The interchange of the letter 'O'; with the number '0' or the letter 'D' • The interchange of the letter 'I'; with the number '1' or the letter 'l' • The interchange of the letter 'S' with the number '5' • The interchange of the letter 'Z' with the number 2 • The interchange of the letter 'n' with the letter 'h'	Checked regardless of scope. Not checked between macros and other identifiers. Messages in report file: • `Identifier "Idf1" (file1.cpp line l1 column c1) and "Idf2" (file2.h line l2 column c2) only differ by the presence/absence of the underscore character.` • `Identifier "Idf1" (file1.cpp line l1 column c1) and "Idf2" (file2.h line l2 column c2) only differ by a mixture of case.` • `Identifier "Idf1" (file1.cpp line l1 column c1) and "Idf2" (file2.h line l2 column c2) only differ by letter 'O', with the number '0'.`

N.	JSF++ Definition	Comments
50	The first word of the name of a class, structure, namespace, enumeration, or type created with `typedef` **will** begin with an uppercase letter. All others letters **will** be lowercase.	Messages in report file: • `The first word of the name of a class will begin with an uppercase letter.` • `The first word of the namespace of a class will begin with an uppercase letter.`
51	All letters contained in function and variables names **will** be composed entirely of lowercase letters.	Messages in report file: • `All letters contained in variable names will be composed entirely of lowercase letters.` • `All letters contained in function names will be composed entirely of lowercase letters.`
52	Identifiers for constant and enumerator values **shall** be lowercase.	Messages in report file: • `Identifier for enumerator value shall be lowercase.` • `Identifier for template constant parameter shall be lowercase.`
53	Header files **will** always have file name extension of "`.h`".	`.H` is allowed if you set the option `-dos`.
53.1	The following character sequences **shall** not appear in header file names: `'`, `\`, `/*`, `//`, or `"`.	
54	Implementation files **will** always have a file name extension of ".cpp".	Not case sensitive if you set the option `-dos`.
57	The public, protected, and private sections of a class **will** be declared in that order.	

N.	JSF++ Definition	Comments
58	When declaring and defining functions with more than two parameters, the leading parenthesis and the first argument **will** be written on the same line as the function name. Each additional argument will be written on a separate line (with the closing parenthesis directly after the last argument).	Detects that two parameters are not on the same line, The first parameter should be on the same line as function name. Does not check for the closing parenthesis.
59	The statements forming the body of an if, else if, else, while, do ... while or for statement **shall** always be enclosed in braces, even if the braces form an empty block.	Messages in report file: • `The statements forming the body of an if statement shall always be enclosed in braces.` • `The statements forming the body of an else statement shall always be enclosed in braces.` • `The statements forming the body of a while statement shall always be enclosed in braces.` • `The statements forming the body of a do ...  while statement shall always be enclosed in braces.` • `The statements forming the body of a for statement shall always be enclosed in braces.`
60	Braces ("`{}`") which enclose a block **will** be placed in the same column, on separate lines directly before and after the block.	Detects that statement-block braces should be in the same columns.
61	Braces ("`{}`") which enclose a block **will** have nothing else on the line except comments.	

N.	JSF++ Definition	Comments
62	The dereference operator '*' and the address-of operator '&' will be directly connected with the type-specifier.	Reports when there is a space between type and "*" "&" for variables, parameters and fields declaration.
63	Spaces will not be used around '.' or '->', nor between unary operators and operands.	Reports when the following characters are not directly connected to a white space: • . • -> • ! • ~ • - • ++ • — **Note** A violation will be reported for "." used in float/double definition.

Classes

N.	JSF++ Definition	Polyspace Comments
67	Public and protected data **should** only be used in structs - not classes.	
68	Unneeded implicitly generated member functions shall be explicitly disallowed.	Reports when default constructor, assignment operator, copy constructor or destructor is not declared.
71.1	A class's virtual functions shall not be invoked from its destructor or any of its constructors.	Reports when a constructor or destructor directly calls a virtual function.

N.	JSF++ Definition	Polyspace Comments
74	Initialization of nonstatic class members **will** be performed through the member initialization list rather than through assignment in the body of a constructor.	All data should be initialized in the initialization list except for array. Does not report that an assignment exists in `ctor` body.Message in report file: `Initialization of nonstatic class members "<field>" will be performed through the member initialization list.`
75	Members of the initialization list **shall** be listed in the order in which they are declared in the class.	
76	A copy constructor and an assignment operator **shall** be declared for classes that contain pointers to data items or nontrivial destructors.	Messages in report file: • `no copy constructor and no copy assign` • `no copy constructor` • `no copy assign`
77.1	The definition of a member function **shall not** contain default arguments that produce a signature identical to that of the implicitly-declared copy constructor for the corresponding class/structure.	Does not report when an explicit copy constructor exists.
78	All base classes with a virtual function **shall** define a virtual destructor.	
79	All resources acquired by a class shall be released by the class's destructor.	Reports when the number of "new" called in a constructor is greater than the number of "delete" called in its destructor. **Note** A violation is raised even if "new" is done in a "if/else".

N.	JSF++ Definition	Polyspace Comments
81	The assignment operator shall handle self-assignment correctly.	Reports when copy assignment body does not begin with "`if (this != arg)`" A violation is not raised if an empty `else` statement follows the `if`, or the body contains only a return statement. A violation is raised when the `if` statement is followed by a statement other than the return statement.

82	An assignment operator **shall** return a reference to `*this`.	The following operators should return `*this` on method, and `*first_arg` on plain function.
		
		`operator=`
		`operator+=`
		`operator-=`
		`operator*=`
		`operator >>=`
		`operator <<=`
		`operator /=`
		`operator %=`
		`operator |=`
		`operator &=`
		`operator ^=`
		`prefix operator++`
		`prefix operator--`
		
		Does not report when no return exists.
		
		No special message if type does not match.
		
		Messages in report file:
		
		• `An assignment operator shall return a reference to *this.`
		• `An assignment operator shall return a reference to its first arg.`

N.	JSF++ Definition	Polyspace Comments
83	An assignment operator shall assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).	Reports when a copy assignment does not assign all data members. In a derived class, it also reports when a copy assignment does not call inherited copy assignments.
88	Multiple inheritance **shall** only be allowed in the following restricted form: `n` interfaces plus `m` private implementations, plus at most one protected implementation.	Messages in report file: • `Multiple inheritance on public implementation shall not be allowed:` *`<public_base_class>`* `is not an interface.` • `Multiple inheritance on protected implementation shall not be allowed :` *`<protected_base_class_1>`* • *`<protected_base_class_2>`* are not interfaces.
88.1	A stateful virtual base **shall** be explicitly declared in each derived class that accesses it.	
89	A base class **shall not** be both virtual and nonvirtual in the same hierarchy.	
94	An inherited nonvirtual function **shall not** be redefined in a derived class.	Does not report for destructor.Message in report file: `Inherited nonvirtual function %s shall not be redefined in a derived class.`
95	An inherited default parameter **shall never** be redefined.	
96	Arrays **shall not** be treated polymorphically.	Reports pointer arithmetic and array like access on expressions whose pointed type is used as a base class.

N.	JSF++ Definition	Polyspace Comments
97	Arrays **shall not** be used in interface.	Only to prevent array-to-pointer-decay, Not checked on private methods
97.1	Neither operand of an equality operator (== or `!=`) **shall** be a pointer to a virtual member function.	Reports `==` and `!=` on pointer to member function of polymorphic classes (cannot determine statically if it is virtual or not), except when one argument is the null constant.

Namespaces

N.	JSF++ Definition	Polyspace Comments
98	Every nonlocal name, except `main()`, **should** be placed in some namespace.	
99	Namespaces **will not** be nested more than two levels deep.	

Templates

N.	JSF++ Definition	Polyspace Comments
104	A template specialization **shall** be declared before its use.	Reports the actual compilation error message.

Functions

N.	JSF++ Definition	Polyspace Comments
107	Functions **shall** always be declared at file scope.	
108	Functions with variable numbers of arguments **shall not** be used.	

N.	JSF++ Definition	Polyspace Comments
109	A function definition should not be placed in a class specification unless the function is intended to be inlined.	Reports when "inline" is not in the definition of a member function inside the class definition.
110	Functions with more than 7 arguments **will not** be used.	
111	A function **shall not** return a pointer or reference to a non-static local object.	Simple cases without alias effect detected.
113	Functions **will** have a single exit point.	Reports first return, or once per function.
114	All exit points of value-returning functions **shall** be through return statements.	
116	Small, concrete-type arguments (two or three words in size) **should** be passed by value if changes made to formal parameters should not be reflected in the calling function.	Report constant parameters references with `sizeof <= 2 * sizeof(int)`. Does not report for copy-constructor.
119	Functions **shall** not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	Direct recursion is reported statically. Indirect recursion reported through the software. Message in report file: `Function <F> shall not call directly itself.`
121	Only functions with 1 or 2 statements **should** be considered candidates for inline functions.	Reports inline functions with more than 2 statements.

Comments

N.	JSF++ Definition	Polyspace Comments
126	Only valid C++ style comments (`//`) **shall** be used.	
133	Every source file will be documented with an introductory comment that provides information on the file name, its contents, and any program-required information (e.g. legal statements, copyright information, etc).	Reports when a file does not begin with two comment lines.**Note**: This rule cannot be annotated in the source code.

Declarations and Definitions

N.	JSF++ Definition	Polyspace Comments
135	Identifiers in an inner scope **shall not** use the same name as an identifier in an outer scope, and therefore hide that identifier.	
136	Declarations should be at the smallest feasible scope.	Reports when: • A global variable is used in only one function. • A local variable is not used in a statement (`expr`, `return`, `init` ...) of the same level of its declaration (in the same block) or is not used in two sub-statements of its declaration. **Note** • Non-used variables are reported. • Initializations at definition are ignored (not considered an access)

N.	JSF++ Definition	Polyspace Comments
137	All declarations at file scope should be static where possible.	
138	Identifiers **shall not** simultaneously have both internal and external linkage in the same translation unit.	
139	External objects will not be declared in more than one file.	Reports all duplicate declarations inside a translation unit. Reports when the declaration localization is not the same in
140	The register storage class specifier **shall not** be used.	
141	A class, structure, or enumeration **will not** be declared in the definition of its type.	

Initialization

N.	JSF++ Definition	Polyspace Comments
142	All variables **shall** be initialized before use.	Done with Non-initialized variable checks in the software.
144	Braces **shall** be used to indicate and match the structure in the non-zero initialization of arrays and structures.	This covers partial initialization.
145	In an enumerator list, the '=' construct **shall not** be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	Generates one report for an enumerator list.

Types

N.	JSF++ Definition	Polyspace Comments
147	The underlying bit representations of floating point numbers **shall not** be used in any way by the programmer.	Reports on casts with float pointers (except with `void*`).
148	Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.	Reports when non enumeration types are used in switches.

Constants

N.	JSF++ Definition	Polyspace Comments
149	Octal constants (other than zero) **shall not** be used.	
150	Hexadecimal constants **will** be represented using all uppercase letters.	
151	Numeric values in code **will not** be used; symbolic values will be used instead.	Reports direct numeric constants (except integer/float value `1, 0`) in expressions, non `-const` initializations. and switch cases. char constants are allowed. Does not report on templates non-type parameter.
151.1	A string literal shall not be modified.	Report when a `char*`, `char[]`, or `string` type is used not as `const`.A violation is raised if a string literal (for example, “ “) is cast as a non `const`.

Variables

N.	JSF++ Definition	Polyspace Comments
152	Multiple variable declarations **shall not** be allowed on the same line.	

Unions and Bit Fields

N.	JSF++ Definition	Polyspace Comments
153	Unions **shall not** be used.	
154	Bit-fields **shall** have explicitly unsigned integral or enumeration types only.	
156	All the members of a structure (or class) **shall** be named and shall only be accessed via their names.	Reports unnamed bit-fields (unnamed fields are not allowed).

Operators

N.	JSF++ Definition	Polyspace Comments
157	The right hand operand of a `&&` or `||` operator shall not contain side effects.	Assumes rule 159 is not violated.Messages in report file: • `The right hand operand of a && operator shall not contain side effects.` • `The right hand operand of a || operator shall not contain side effects.`
158	The operands of a logical `&&` or `||` **shall** be parenthesized if the operands contain binary operators.	Messages in report file: • `The operands of a logical && shall be parenthesized if the operands contain binary operators.` • `The operands of a logical || shall be parenthesized if the operands contain binary operators.`

N.	JSF++ Definition	Polyspace Comments
		Exception for: `X || Y || Z , Z&&Y &&Z`
159	Operators `||`, `&&`, and unary `&` **shall not** be overloaded.	Messages in report file: • `Unary operator & shall not be overloaded.` • `Operator || shall not be overloaded.` • `Operator && shall not be overloaded.`
160	An assignment expression **shall** be used only as the expression in an expression statement.	Only simple assignment, not +=, ++, etc.
162	Signed and unsigned values **shall not** be mixed in arithmetic or comparison operations.	
163	Unsigned arithmetic **shall not** be used.	
164	The right hand operand of a shift operator **shall** lie between zero and one less than the width in bits of the left-hand operand (inclusive).	
164.1	The left-hand operand of a right-shift operator **shall not** have a negative value.	Detects constant case +. Found by the software for dynamic cases.
165	The unary minus operator **shall not** be applied to an unsigned expression.	
166	The `sizeof` operator **will not** be used on expressions that contain side effects.	
168	The comma operator **shall not** be used.	

Pointers and References

N.	JSF++ Definition	
169	Pointers to pointers should be avoided when possible.	Reports second-level pointers, except for arguments of main.
170	More than 2 levels of pointer indirection **shall not** be used.	Only reports on variables/parameters.
171	Relational operators shall not be applied to pointer types except where both operands are of the same type and point to: • the same object, • the same function, • members of the same object, or • elements of the same array (including one past the end of the same array).	Reports when relational operator are used on pointer types (casts ignored).
173	The address of an object with automatic storage **shall not** be assigned to an object which persists after the object has ceased to exist.	
174	The null pointer **shall not** be de-referenced.	Done with checks in software.
175	A pointer **shall not** be compared to `NULL` or be assigned `NULL`; use plain `0` instead.	Reports usage of `NULL` macro in pointer contexts.
176	A `typedef` **will** be used to simplify program syntax when declaring function pointers.	Reports non-`typedef` function pointers, or pointers to member functions for types of variables, fields, parameters. Returns type of function, cast, and exception specification.

Type Conversions

N.	JSF++ Definition	Comments
177	User-defined conversion functions **should** be avoided.	Reports user defined conversion function, non-explicit constructor with one parameter or default value for others (even undefined ones). Does not report copy-constructor. Additional message for constructor case: `This constructor should be flagged as "explicit".`
178	Down casting (casting from base to derived class) **shall** only be allowed through one of the following mechanism: • Virtual functions that act like dynamic casts (most likely useful in relatively simple cases). • Use of the visitor (or similar) pattern (most likely useful in complicated cases).	Reports explicit down casting, dynamic_cast included. (Visitor patter does not have a special case.)
179	A pointer to a virtual base class **shall not** be converted to a pointer to a derived class.	Reports this specific down cast. Allows dynamic_cast.

180	Implicit conversions that may result in a loss of information **shall not** be used.	Reports the following implicit casts :
		`integer => smaller integer`
		`unsigned => smaller or eq signed`
		`signed => smaller or eq un-signed`
		`integer => float`
		`float => integer`
		
		Does not report for cast to `bool` reports for implicit cast on constant done with the options `-scalar-overflows-checks signed-and-unsigned` or `-ignore-constant-overflows`

N.	JSF++ Definition	Comments
		.
181	Redundant explicit casts **will not** be used.	Reports useless cast: `cast T to T`. Casts to equivalent `typedefs` are also reported.
182	Type casting from any type to or from pointers **shall not** be used.	Does not report when Rule 181 applies.
184	Floating point numbers **shall not** be converted to integers unless such a conversion is a specified algorithmic requirement or is necessary for a hardware interface.	Reports `float->int` conversions. Does not report implicit ones.
185	C++ style casts (`const_cast`, `reinterpret_cast`, and `static_cast`) **shall** be used instead of the traditional C-style casts.	

Flow Control Standards

N.	JSF++ Definition	Comments
186	There **shall** be no unreachable code.	Done with gray checks in the software.
187	All non-null statements **shall** potentially have a side-effect.	
188	Labels **will not** be used, except in switch statements.	
189	The `goto` statement **shall** not be used.	
190	The `continue` statement **shall not** be used.	
191	The `break` statement **shall not** be used (except to terminate the cases of a switch statement).	

N.	JSF++ Definition	Comments
192	All `if`, `else if` constructs will contain either a final `else` clause or a comment indicating why a final `else` clause is not necessary.	`else if` should contain an `else` clause.
193	Every non-empty `case` clause in a switch statement **shall** be terminated with a `break` statement.	
194	All `switch` statements that do not intend to test for every enumeration value **shall** contain a final `default` clause.	Reports only for missing `default`.
195	A `switch` expression **will** not represent a Boolean value.	
196	Every `switch` statement **will** have at least two cases and a potential `default`.	
197	Floating point variables **shall not** be used as loop counters.	Assumes 1 loop parameter.
198	The initialization expression in a `for` loop **will** perform no actions other than to initialize the value of a single `for` loop parameter.	Reports if `loop` parameter cannot be determined. Assumes Rule 200 is not violated. The `loop variable` parameter is assumed to be a variable.
199	The increment expression in a `for` loop **will** perform no action other than to change a single loop parameter to the next value for the loop.	Assumes 1 loop parameter (Rule 198), with non class type. Rule 200 must not be violated for this rule to be reported.
200	Null initialize or increment expressions in `for` loops **will not** be used; a `while` loop will be used instead.	
201	Numeric variables being used within a *for* loop for iteration counting shall not be modified in the body of the loop.	Assumes 1 loop parameter (AV rule 198), and no alias writes.

Expressions

N.	JSF++ Definition	Polyspace Comments
202	Floating point variables **shall not** be tested for exact equality or inequality.	Reports only direct equality/inequality. Check done for all expressions.
203	Evaluation of expressions **shall not** lead to overflow/underflow.	Done with overflow checks in the software.
204	A single operation with side-effects shall only be used in the following contexts: • by itself • the right-hand side of an assignment • a condition • the only argument expression with a side-effect in a function call • condition of a loop • switch condition • single part of a chained operation	Reports when: • A side effect is found in a return statement • A side effect exists on a single value, and only one operand of the function call has a side effect.
204.1	The value of an expression shall be the same under any order of evaluation that the standard permits.	Reports when: • Variable is written more than once in an expression • Variable is read and write in sub-expressions • Volatile variable is accessed more than once **Note** Read-write operations such as ++, are only considered as a write.
205	The volatile keyword **shall not** be used unless directly interfacing with hardware.	Reports if volatile keyword is used.

Memory Allocation

N.	JSF++ Definition	Comments
206	Allocation/deallocation from/to the free store (heap) **shall not** occur after initialization.	Reports calls to C library functions: `malloc` / `calloc` / `realloc` / `free` and all `new`/`delete` operators in functions or methods.

Fault Handling

N.	JSF++ Definition	Comments
208	C++ exceptions **shall not** be used.	Reports `try`, `catch`, `throw spec`, and `throw`.

Portable Code

N.	JSF++ Definition	Comments
209	The basic types of `int`, `short`, `long`, `float` and `double` **shall not** be used, but specific-length equivalents should be `typedef`'d accordingly for each compiler, and these type names used in the code.	Only allows use of basic types through direct `typedefs`.
213	No dependence shall be placed on C++'s operator precedence rules, below arithmetic operators, in expressions.	Reports when a binary operation has one operand that is not parenthesized and is an operation with inferior precedence level. Reports bitwise and shifts operators that are used without parenthesis and binary operation arguments.

215	Pointer arithmetic **will not** be used.	Reports:
		`p + I`
		`p - I`
		`p++`
		`p--`
		`p+=`
		`p-=` Allows `p[i]`.

JSF++ Rules Not Checked

- “Code Size and Complexity” on page 3-122
- “Rules” on page 3-122
- “Environment” on page 3-122
- “Libraries” on page 3-122
- “Header Files” on page 3-123
- “Style” on page 3-123
- “Classes” on page 3-123
- “Namespaces” on page 3-125
- “Templates” on page 3-125
- “Functions” on page 3-126
- “Comments” on page 3-127
- “Initialization” on page 3-127
- “Types” on page 3-127
- “Unions and Bit Fields” on page 3-128
- “Operators” on page 3-128
- “Type Conversions” on page 3-128
- “Expressions” on page 3-128
- “Memory Allocation” on page 3-129
- “Portable Code” on page 3-129
- “Efficiency Considerations” on page 3-129
- “Miscellaneous” on page 3-129
- “Testing” on page 3-130

Code Size and Complexity

N.	JSF++ Definition
2	There shall not be any self-modifying code.

Rules

N.	JSF++ Definition
4	To break a “should” rule, the following approval must be received by the developer: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)
5	To break a “will” or a “shall” rule, the following approvals must be received by the developer: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool) • approval from the software product manager (obtained by the unit approval in the developmental CM tool)
6	Each deviation from a “shall” rule shall be documented in the file that contains the deviation. Deviations from this rule shall not be allowed, AV Rule 5 notwithstanding.
7	Approval will not be required for a deviation from a “shall” or “will” rule that complies with an exception specified by that rule.

Environment

N.	JSF++ Definition
10	Values of character types will be restricted to a defined and documented subset of ISO 10646 1.

Libraries

N.	JSF++ Definition
16	Only DO-178B level A [15] certifiable or SEAL 1 C/C++ libraries shall be used with safety-critical (i.e. SEAL 1) code.

Header Files

N.	JSF++ Definition
34	Header files should contain logically related declarations only.
36	Compilation dependencies should be minimized when possible.
37	Header (include) files should include only those header files that are required for them to successfully compile. Files that are only used by the associated .cpp file should be placed in the .cpp file — not the .h file.
38	Declarations of classes that are only accessed via pointers (*) or references (&) should be supplied by forward headers that contain only forward declarations.

Style

N.	JSF++ Definition
45	All words in an identifier will be separated by the '_' character.
49	All acronyms in an identifier will be composed of uppercase letters.
55	The name of a header file should reflect the logical entity for which it provides declarations.
56	The name of an implementation file should reflect the logical entity for which it provides definitions and have a ".cpp" extension (this name will normally be identical to the header file that provides the corresponding declarations.) At times, more than one .cpp file for a given logical entity will be required. In these cases, a suffix should be appended to reflect a logical differentiation.

Classes

N.	JSF++ Definition
64	A class interface should be complete and minimal.
65	A structure should be used to model an entity that does not require an invariant.
66	A class should be used to model an entity that maintains an invariant.

N.	JSF++ Definition
69	A member function that does not affect the state of an object (its instance variables) will be declared const. Member functions should be const by default. Only when there is a clear, explicit reason should the const modifier on member functions be omitted.
70	A class will have friends only when a function or object requires access to the private elements of the class, but is unable to be a member of the class for logical or efficiency reasons.
70.1	An object shall not be improperly used before its lifetime begins or after its lifetime ends.
71	Calls to an externally visible operation of an object, other than its constructors, shall not be allowed until the object has been fully initialized.
72	The invariant for a class should be: • A part of the postcondition of every class constructor, • A part of the precondition of the class destructor (if any), • A part of the precondition and postcondition of every other publicly accessible operation.
73	Unnecessary default constructors shall not be defined.
77	A copy constructor shall copy all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).
80	The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.
84	Operator overloading will be used sparingly and in a conventional manner.
85	When two operators are opposites (such as == and !=), both will be defined and one will be defined in terms of the other.
86	Concrete types should be used to represent simple independent concepts.
87	Hierarchies should be based on abstract classes.
90	Heavily used interfaces should be minimal, general and abstract.
91	Public inheritance will be used to implement “is-a” relationships.

N.	JSF++ Definition
92	A subtype (publicly derived classes) will conform to the following guidelines with respect to all classes involved in the polymorphic assignment of different subclass instances to the same variable or parameter during the execution of the system: • Preconditions of derived methods must be at least as weak as the preconditions of the methods they override. • Postconditions of derived methods must be at least as strong as the postconditions of the methods they override. In other words, subclass methods must expect less and deliver more than the base class methods they override. This rule implies that subtypes will conform to the Liskov Substitution Principle.
93	“has-a” or “is-implemented-in-terms-of” relationships will be modeled through membership or non-public inheritance.

Namespaces

N.	JSF++ Definition
100	Elements from a namespace should be selected as follows: • using declaration or explicit qualification for few (approximately five) names, • using directive for many names.

Templates

N.	JSF++ Definition
101	Templates shall be reviewed as follows: **1** with respect to the template in isolation considering assumptions or requirements placed on its arguments. **2** with respect to all functions instantiated by actual arguments.
102	Template tests shall be created to cover all actual template instantiations.

N.	JSF++ Definition
103	Constraint checks should be applied to template arguments.
105	A template definition's dependence on its instantiation contexts should be minimized.
106	Specializations for pointer types should be made where appropriate.

Functions

N.	JSF++ Definition
112	Function return values should not obscure resource ownership.
115	If a function returns error information, then that error information will be tested.
117	Arguments should be passed by reference if NULL values are not possible: • **117.1** – An object should be passed as `const T&` if the function should not change the value of the object. • **117.2** – An object should be passed as `T&` if the function may change the value of the object.
118	Arguments should be passed via pointers if NULL values are possible: • **118.1** – An object should be passed as `const T*` if its value should not be modified. • **118.2** – An object should be passed as `T*` if its value may be modified.
120	Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal
122	Trivial accessor and mutator functions should be inlined.
123	The number of accessor and mutator functions should be minimized.
124	Trivial forwarding functions should be inlined.
125	Unnecessary temporary objects should be avoided.

Comments

N.	JSF++ Definition
127	Code that is not used (commented out) shall be deleted.**Note**: This rule cannot be annotated in the source code.
128	Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented will not be allowed.
129	Comments in header files should describe the externally visible behavior of the functions or classes being documented.
130	The purpose of every line of executable code should be explained by a comment, although one comment may describe more than one line of code.
131	One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).
132	Each variable declaration, typedef, enumeration value, and structure member will be commented.
134	Assumptions (limitations) made by functions should be documented in the function's preamble.

Initialization

N.	JSF++ Definition
143	Variables will not be introduced until they can be initialized with meaningful values. (See also AV Rule 136, AV Rule 142, and AV Rule 73 concerning declaration scope, initialization before use, and default constructors respectively.)

Types

N.	JSF++ Definition
146	Floating point implementations shall comply with a defined floating point standard. The standard that will be used is the ANSI/IEEE Std 754 [1].

Unions and Bit Fields

N.	JSF++ Definition
155	Bit-fields will not be used to pack data into a word for the sole purpose of saving space.

Operators

N.	JSF++ Definition
167	The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.

Type Conversions

N.	JSF++ Definition
183	Every possible measure should be taken to avoid type casting.

Expressions

N.	JSF++ Definition
204	A single operation with side-effects shall only be used in the following contexts: **1** by itself **2** the right-hand side of an assignment **3** a condition **4** the only argument expression with a side-effect in a function call **5** condition of a loop **6** switch condition **7** single part of a chained operation

Memory Allocation

N.	JSF++ Definition
207	Unencapsulated global data will be avoided.

Portable Code

N.	JSF++ Definition
210	Algorithms shall not make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.).
210.1	Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.
211	Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.
212	Underflow or overflow functioning shall not be depended on in any special way.
214	Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.

Efficiency Considerations

N.	JSF++ Definition
216	Programmers should not attempt to prematurely optimize code.

Miscellaneous

N.	JSF++ Definition
217	Compile-time and link-time errors should be preferred over run-time errors.
218	Compiler warning levels will be set in compliance with project policies.

Testing

N.	JSF++ Definition
219	All tests applied to a base class interface shall be applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions /invariants shall be substituted in the derived class tests.
220	Structural coverage algorithms shall be applied against flattened classes.
221	Structural coverage of a class within an inheritance hierarchy containing virtual functions shall include testing every possible resolution for each set of identical polymorphic references.

Check Coding Rules from the Polyspace Environment

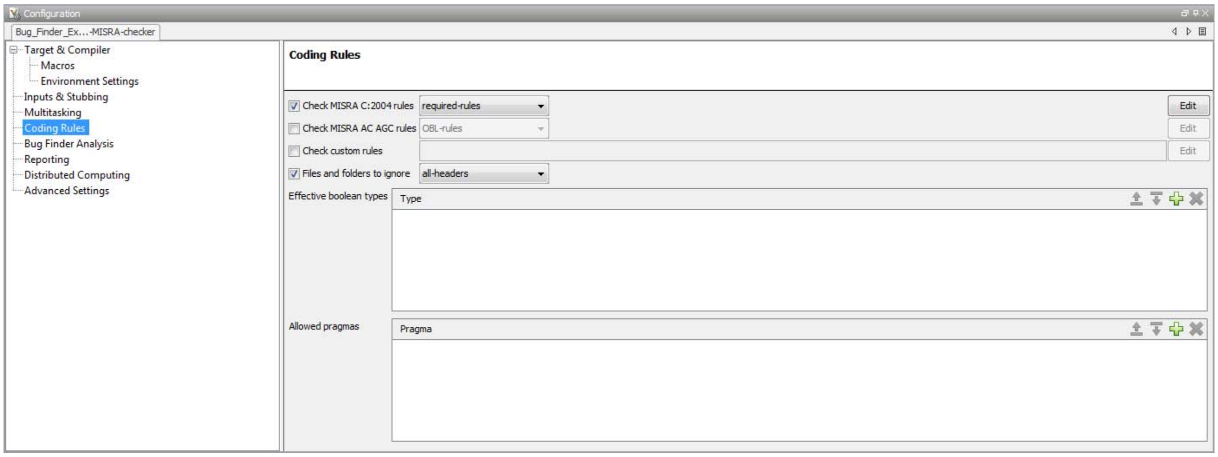
- “Activate Coding Rules Checker” on page 4-2
- “Select Specific MISRA or JSF Coding Rules” on page 4-6
- “Create Custom Coding Rules” on page 4-8
- “Format of Custom Coding Rules File” on page 4-10
- “Exclude Files From Rule Checking” on page 4-12
- “Allow Custom Pragma Directives” on page 4-13
- “Specify Boolean Types” on page 4-14
- “Find Coding Rule Violations” on page 4-15
- “Review Coding Rule Violations” on page 4-16
- “Apply Coding Rule Violation Filters” on page 4-18

Activate Coding Rules Checker

This example shows how to activate the coding rules checker before you start an analysis. This activation enables Polyspace Bug Finder to search for coding rule violations. You can view the coding rule violations in your analysis results.

1 Open project configuration.

2 In the **Configuration** tree view, select **Coding Rules**.



Configuration

Bug_Finder_Ex...-MISRA-checker

- Target & Compiler
  - Macros
  - Environment Settings
- Inputs & Stubbing
- Multitasking
- Coding Rules
- Bug Finder Analysis
- Reporting
- Distributed Computing
- Advanced Settings

Coding Rules

☑ Check MISRA C:2004 rules	required-rules	Edit
☐ Check MISRA AC AGC rules	OBL-rules	Edit
☐ Check custom rules		Edit
☑ Files and folders to ignore	all-headers	

Effective boolean types: Type

Allowed pragmas: Pragma

3 Select the check box for the type of coding rules that you want to check.

For C code, you can check compliance with:

- MISRA C:2004
- MISRA AC AGC
- Custom coding rules

For C++ code, you can check compliance with:

- MISRA C++
- JSF C++
- Custom coding rules

4 For each rule type that you select, from the drop-down list, select the subset of rules to check.

MISRA C:2004

Option	Description
`required-rules`	All required MISRA C coding rules.
`all-rules`	All required and advisory MISRA C coding rules.
`SQO-subset1`	A small subset of MISRA C rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA C coding rules that you specify.

MISRA AC AGC

Option	Description
`OBL-rules`	All required MISRA AC AGC coding rules.
`OBL-REC-rules`	All required and recommended MISRA AC AGC coding rules.
`all-rules`	All required, recommended, and readability coding rules.
`SQO-subset1`	A small subset of MISRA AC AGC rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.

Option	Description
`SQO-subset2`	A second subset of MISRA AC AGC rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA AC AGC coding rules that you specify.

MISRA C++

Option	Description
`required-rules`	All required MISRA C++ coding rules.
`all-rules`	All required and advisory MISRA C++ coding rules.
`SQO-subset1`	A small subset of MISRA C++ rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of rules with indirect impact on the selectivity in addition to `SQO-subset1`. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A specified set of MISRA C++ coding rules.

JSF C++

Option	Description
`shall-rules`	**Shall** rules are mandatory requirements. These rules require verification.
`shall-will-rules`	All **Shall** and **Will** rules. **Will** rules are intended to be mandatory requirements. However, these rules do not require verification.

Option	Description
`all-rules`	All **Shall**, **Will**, and **Should** rules. **Should** rules are advisory rules.
`custom`	A set of JSF C++ coding rules that you specify.

5 If you select **Check custom rules**, specify the path to your custom rules file or click **Edit** to create one.

When rules checking is complete, the software displays the coding rule violations in purple on the **Results Summary** pane.

Related Examples

- “Select Specific MISRA or JSF Coding Rules” on page 4-6
- “Create Custom Coding Rules” on page 4-8
- “Exclude Files From Rule Checking” on page 4-12

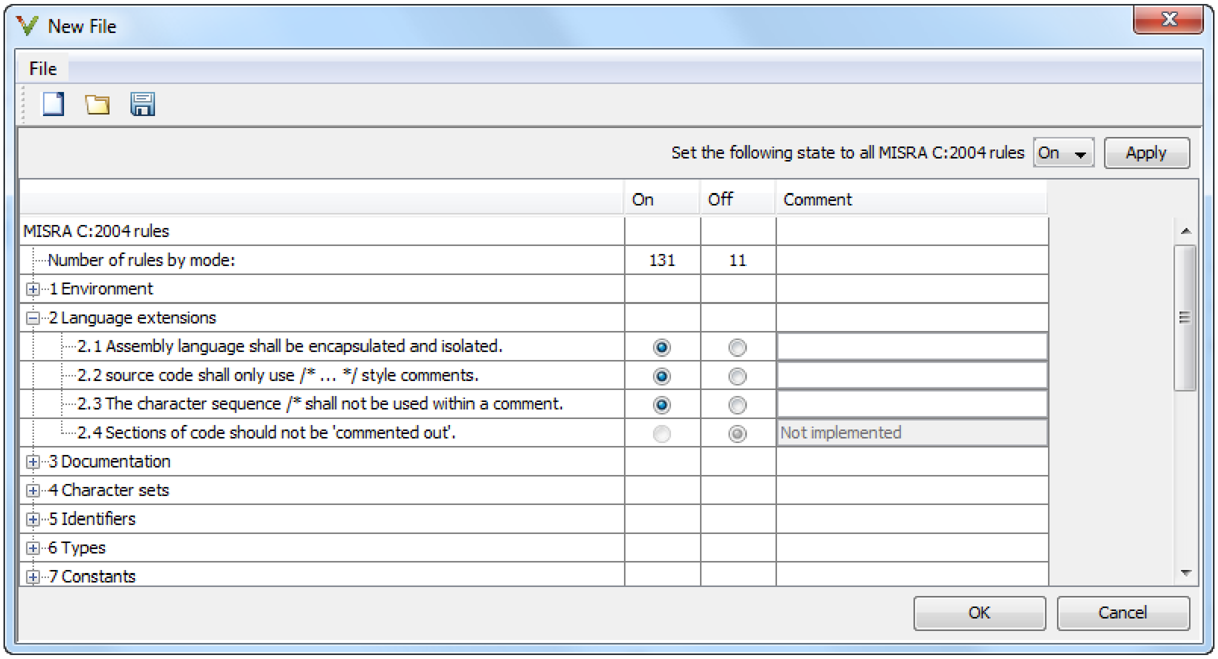
Concepts

- “Rule Checking” on page 3-2

Select Specific MISRA or JSF Coding Rules

This example shows how to specify a subset of MISRA or JSF rules for the coding rules checker. If you select `custom` from the MISRA or JSF drop-down list, you must provide a file that specifies the rules to check.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules**.
3. Select the check box for the type of coding rules you wish to check
4. From the corresponding drop-down list, select `custom`. The software displays a new field for your custom file.
5. To the right of this field, click **Edit**. A New File window opens, displaying a table of rules.



Set the following state to all MISRA C:2004 rules [On] Apply

	On	Off	Comment
MISRA C:2004 rules			
Number of rules by mode:	131	11	
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			

Select **On** for the rules you want to check.

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

   The **Save as** dialog box opens.

7 In the **File** field, enter a name for the rules file.

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

   The full path to the rules file appears. To reuse this rules file for other projects, type this path name or use the 📁 icon in the New File window.

Related Examples

- “Activate Coding Rules Checker” on page 4-2
- “Create Custom Coding Rules” on page 4-8

Concepts

- “Rule Checking” on page 3-2

Create Custom Coding Rules

This example shows how to create a custom coding rules file. You can use this file to check names or text patterns in your source code against custom rules that you specify. For each rule, you specify a pattern in the form of a regular expression. The software compares the pattern against identifiers in the source code and determines whether the custom rule is violated.

Save Example Code

Save the following code in a file `printInitialValue.c`:

```
#include <stdio.h>

typedef struct {
int a;
int b;
} collection;

void main()
{
 collection myCollection={0,0};
 printf("Initial values in the collection are %d
                     and %d.",myCollection.a,myCollection.b);
}
```

Create Coding Rules File

1 Create a Polyspace project. Add `printInitialValue.c` to the project.

2 On the **Configuration** pane, select **Coding Rules**. Select the **Check custom rules** box.

3 Click Edit.

   The New File window opens, displaying a table of rule groups.

4 From the drop-down list **Set the following state to all Custom C rules**, select `Off`. Click **Apply**.

5 Expand the **Structs** node. For the option **4.3 All struct fields must follow the specified pattern**:

Column Title	Action
On	Select ●.
Convention	Enter `All struct fields must begin with s_ and have capital letters.`
Pattern	Enter `s_[A-Z0-9_]`
Comment	Leave blank. This column is for comments that appear in the coding rules file alone.

Review Coding Rule Violations

1 Save the file and run the verification. On the **Results Summary** pane, you see two violations of rule 4.3. Select the first violation.

   a On the **Source** pane, the line `int a;` is marked.

   b On the **Check Details** pane, you see the error message you had entered, `All struct fields must begin with s_ and have capital letters.`

2 Right-click on the **Source** pane and select **Open Source File**. The file `printInitialValue.c` opens in a text editor.

3 In the file, replace all instances of `a` with `s_A` and `b` with `s_B`. Save the file and rerun the verification.

   The custom rule violations no longer appear on the **Results Summary** pane.

Related Examples

- "Activate Coding Rules Checker" on page 4-2
- "Select Specific MISRA or JSF Coding Rules" on page 4-6
- "Exclude Files From Rule Checking" on page 4-12

Concepts

- "Rule Checking" on page 3-2
- "Format of Custom Coding Rules File" on page 4-10

Format of Custom Coding Rules File

In a custom coding rules file, each rule appears in the following format:

```
N.n off|warning
convention=violation_message
pattern=regular_expression
```

- *`N.n`* — Custom rule number, for example, 1.2.
- `off` — Rule is not considered.
- `warning` — The software checks for violation of the rule. After verification, it displays the coding rule violation on the **Results Summary** pane.
- *`violation_message`* — Software displays this text in an XML file within the *`Results`*`/Polyspace-Doc` folder.
- *`regular_expression`* — Software compares this text pattern against a source code identifier that is specific to the rule. See "Custom Naming Convention Rules" on page 3-3.

  The keywords `convention=` and `pattern=` are optional. If present, they apply to the rule whose number immediately precedes these keywords. If `convention=` is not given for a rule, then a standard message is used. If `pattern=` is not given for a rule, then the default regular expression is used, that is, `.*`.

  Use the symbol `#` to start a comment. Comments are not allowed on lines with the keywords `convention=` and `pattern=`.

  The following example contains three custom rules: 1.1, 8.1, and 9.1.

```
# Custom rules configuration file
1.1  off          # Disable custom rule number 1.1
8.1  warning       # Violation of custom rule 8.1 produces a warning
convention=Global constants must begin by G_ and must be in capital letters.
pattern=G_[A-Z0-9_]*
9.1  warning    # Non-adherence to custom rule 9.1 produces a warning
convention=Global variables should begin by g_.
pattern=g_.*
```

Related Examples

- "Create Custom Coding Rules" on page 4-8

Exclude Files From Rule Checking

This example shows how to exclude certain files from coding rules checking.

1 Open project configuration.

2 In the **Configuration** tree view, select **Coding Rules**.

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

4 From the corresponding drop-down list, select one of the following:

   - `all-headers` (default) — Rule checker excludes folders that contain only header files, that is, folders without source files.
   - `all` — Rule checker excludes all include folders. For example, if you are checking a large code base with standard or Visual headers, excluding include folders can significantly improve the speed of code analysis.
   - `custom` — Rule checker excludes files or folders specified in the **File/Folder** view. To add files to the custom **File/Folder** list, select to choose the files and folders to exclude. To remove a file or folder from the list of excluded files and folders, select the row. Then click .

Related Examples

- "Activate Coding Rules Checker" on page 4-2

Concepts

- "Rule Checking" on page 3-2

Allow Custom Pragma Directives

This example shows how to exclude custom pragma directives from coding rules checking. MISRA C rule 3.4 requires checking that pragma directives are documented within the documentation of the compiler. However, you can allow undocumented pragma directives to be present in your code.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules**.
3. To the right of **Allowed pragmas**, click ➕.

   In the **Pragma** view, the software displays an active text field.
4. In the text field, enter a pragma directive.
5. To remove a directive from the **Pragma** list, select the directive. Then click ✖.

Related Examples

- "Activate Coding Rules Checker" on page 4-2

Concepts

- "Rule Checking" on page 3-2

Specify Boolean Types

This example shows how to specify data types you want Polyspace to consider as Boolean during MISRA C rules checking. The software applies this redefinition only to data types defined by `typedef` statements. The use of this option may affect the checking of MISRA C rules 12.6, 13.2, and 15.4.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules**.
3. To the right of **Effective boolean types**, click ➕.

   In the **Type** view, the software displays an active text field.
4. In the text field, specify the data type that you want Polyspace to treat as Boolean.
5. To remove a data type from the **Type** list, select the data type. Then click ✖.

Related Examples

- "Activate Coding Rules Checker" on page 4-2

Concepts

- "Rule Checking" on page 3-2

Find Coding Rule Violations

This example shows how to check for coding rule violations alone.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules**. Activate the desired coding rule checker.
3. In the **Configuration** tree view, select **Bug Finder Analysis**.
4. Clear the **Find defects** check box.
5. Click ▷ to run the coding rules checker without checking defects.

   You can view the results by selecting the `*RuleSet*-report.xml` file from the results folder.

Related Examples

- "Activate Coding Rules Checker" on page 4-2
- "Select Specific MISRA or JSF Coding Rules" on page 4-6
- "Review Coding Rule Violations" on page 4-16

Concepts

- "Rule Checking" on page 3-2

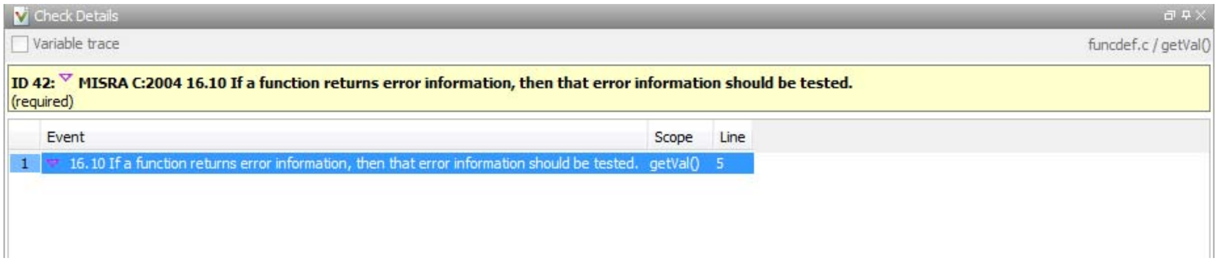
Review Coding Rule Violations

This example shows how to review coding rule violations in the Results Manager perspective once code analysis is complete. After analysis, the **Results Summary** pane displays the rule violations with a

- ▽ symbol for predefined coding rules such as MISRA C:2004.
- ▼ symbol for custom coding rules.

1 Select a coding-rule violation on the **Results Summary** pane.
   - The predefined rules such as MISRA C or C++ or JSF C++ are indicated by ▽ .
   - The custom rules are indicated by ▼ .

2 On the **Check Details** pane, view the location and description of the violated rule. In the source code, the line containing the violation appears highlighted.



ID 42: ▽ MISRA C:2004 16.10 If a function returns error information, then that error information should be tested.
(required)

	Event	Scope	Line
1	▼ 16.10 If a function returns error information, then that error information should be tested.	getVal()	5

3 Review the violation. On the **Check Review** tab, select a **Classification** to describe the severity of the issue:
   - `High`
   - `Medium`
   - `Low`
   - `Not a defect`

4 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`

You can also define your own statuses.

5 In the comment box, enter additional information about the violation.

6 To open the source file that contains the coding rule violation, on the **Source** pane, right-click the code with the purple check. From the context menu, select **Open Source File**. The file opens in your text editor.

7 Fix the coding rule violation.

8 When you have corrected the coding rule violations, run the analysis again.

Related Examples

- "Activate Coding Rules Checker" on page 4-2
- "Find Coding Rule Violations" on page 4-15
- "Apply Coding Rule Violation Filters" on page 4-18

Apply Coding Rule Violation Filters

This example shows how to use filters in the **Results Summary** pane to focus on specific kinds of coding rule violations. By default, the software displays both coding rule violations and defects.

To filter violations by rule number:

1. On the **Results Summary** pane, place your cursor on the **Check** column header. Click the filter icon that appears.
2. From the context menu, clear the **All** check box.
3. Select the violated rule numbers that you want to focus on.
4. Click **OK**.

Related Examples

- "Activate Coding Rules Checker" on page 4-2
- "Review Coding Rule Violations" on page 4-16

Find Bugs From the Polyspace Environment

- “Choose Specific Defects” on page 5-2
- “Run Local Analysis” on page 5-3
- “Run Remote Batch Analysis” on page 5-4
- “Monitor Analysis” on page 5-5
- “Specify Results Folder” on page 5-6

Choose Specific Defects

There are two preset configurations for Bug Finder defects, but you can also customize which defects to check for during the analysis.

1 In the Configuration pane, select **Bug Finder Analysis** to view the Bug Finder Analysis defects pane.

2 Select the **Find defects** check box.

3 From the drop-down menu, select a set of defects. The options are:

   - `default` for the default list of defects. This list contains defects that are applicable to most coding projects. To see if certain defects are included in this list, refer to the individual check reference pages.
   - `all` for all defects.
   - `custom` to select and deselect individual defects or categories of defects.

Run Local Analysis

Before running an analysis from the Polyspace interface, you must set up your project's source files and analysis options. For more information, see "Create New Project" on page 1-17.

1. Select a project to analyze.

2. Select the ▷ button.

   You can monitor the analysis in the Monitor tab. If the analysis fails, the **Output Summary** window lists errors or warnings.

   Once the analysis has completed, you can open your results from the Results folder.

Run Remote Batch Analysis

Before running a batch analysis, you must set up your project's source files, analysis options, and remote analysis settings. If you have not done so, see "Create New Project" on page 1-17 and "Set Up Polyspace Metrics" on page 1-7.

1 Select a project to analyze.

2 In the Configuration window, select the **Distributed Computing** pane.

3 Select the **Batch** check box.

4 If you want to store your results in the Polyspace Metrics repository, select the **Add to results repository** check box.

   Otherwise, clear this check box.

5 Select the ▷ button.

   You can monitor the analysis from the Polyspace Queue Manager ✓.

   Once the analysis has completed, you can open your results from the Results folder, or download them from Polyspace Metrics.

Monitor Analysis

To monitor the progress of a local analysis, use the following tabs in the Project Manager perspective of Polyspace Bug Finder:

- **Progress Monitor** — A blue progress bar indicates the time and percentage completed.
- **Full Log** — This tab displays messages, errors, and statistics for the phases of the analysis. 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 analysis, the **Verification Statistics** tab displays statistics, for example, code coverage and check distribution.

To monitor the progress of a remote analysis:

1. From the Polyspace interface, select the Queue Manager button .
2. In the Polyspace Queue Manager, follow your job progress.

Specify Results Folder

By default, Polyspace Bug Finder saves your results in the same directory as your project in a folder called `Results`. Each subsequent analysis overwrites the old results.

However, to specify a different location for results:

1. In the Project Browser, right-click on the Results folder.
2. From the context menu, select `Choose a Result Folder`.
3. In the **Choose a Result Folder** window, navigate to the new results folder and click **Select**.

   In the Project Browser, the new results folder appears.

   The previous results folder disappears from the Project Browser. However, the results have not been deleted, just removed from the Polyspace interface. To view the previous results, use **File > Open Results**.

View Results in the Polyspace Environment

- "Open Results" on page 6-2
- "View Results Summary in Polyspace Metrics" on page 6-3
- "Download Results From Polyspace Metrics" on page 6-5
- "Filter and Group Results" on page 6-8
- "Generate Reports" on page 6-15
- "Review and Comment Results" on page 6-16
- "Import Comments from Previous Analyses" on page 6-20
- "Code Metrics" on page 6-21
- "View Code Sequence Causing Defect" on page 6-28
- "Results Folder Contents" on page 6-31
- "Windows in the Results Manager Perspective" on page 6-33
- "Bug Finder Defect Categories" on page 6-47
- "Common Weakness Enumeration from Bug Finder Defects" on page 6-49

Open Results

This example shows how to open Polyspace Bug Finder results in the Results Manager perspective. Before you open the results, you must run Polyspace Bug Finder analysis on your source files and obtain a results file with extension `.psbf`.

Open Results from Active Project

Suppose you have a project called `Bug_Finder_Example` open in the **Project Browser.** The results are from the project are called `Bug_Finder_Example.psbf` in the folder `Results`.

1. Navigate to **Bug_Finder_Example.psbf** under **Results**.
2. Double-click **Bug_Finder_Example.psbf**. The analysis results appear in the Results Manager perspective.

Open Results File Using File Browser

If the results file `Bug_Finder_Example.psbf` is located on the path `'C:\Bug_Finder_Example\Results'`

1. Select **File > Open Result...**. The Open Results browser opens.
2. Navigate to the result folder containing the file with extension `.psbf`. In this example, navigate to `'C:\Bug_Finder_Example\Results'`.
3. Select the file. Click **Open**. The analysis results appear in the Results Manager perspective.

Concepts

- "Results Folder Contents" on page 6-31
- "Windows in the Results Manager Perspective" on page 6-33

View Results Summary in Polyspace Metrics

This example shows how to view results summary in Polyspace Metrics. If you check the configuration option **Add to results repository** under **Distributed Computing**, after remote analysis, you can view a summary of the results in Polyspace Metrics.

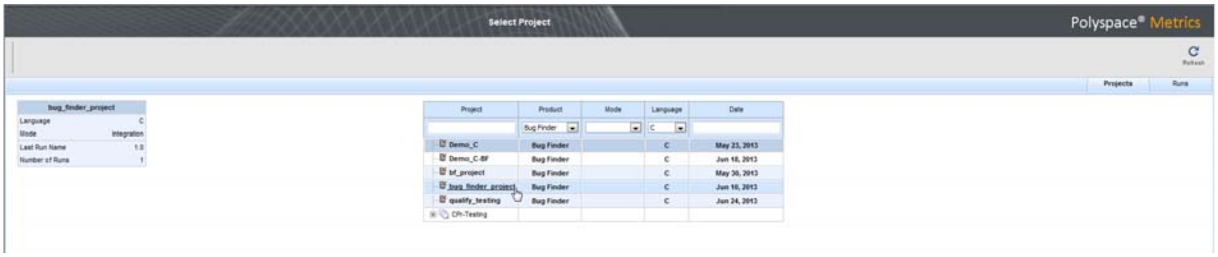
Open Polyspace Metrics

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

`protocol:// ServerName: PortNumber`

- *`protocol`* is either `http` (default) or `https`. To use HTTPS, you must set up the configuration file and the **Metrics and Remote Server Settings**.
- *`ServerName`* is the name or IP address of your Polyspace Metrics server.
- *`PortNumber`* is the Web server port number (default 8080)

On the webpage, you can view the projects saved to your Polyspace Metrics repository.



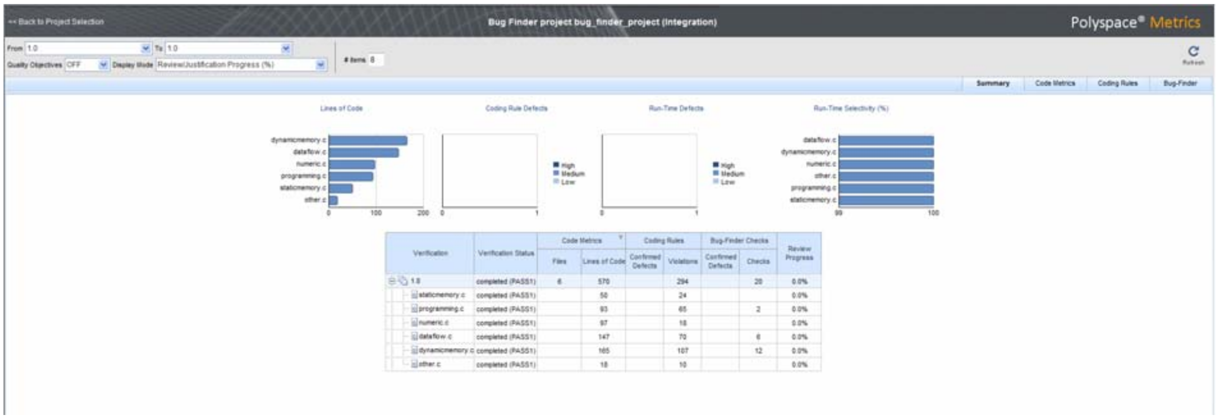
bug_finder_project	
Language	C
Mode	integration
Last Run Name	1.0
Number of Runs	1

Project	Product	Mode	Language	Date
	Bug Finder		C	
Demo_C	Bug Finder		C	May 23, 2013
Demo_C-BF	Bug Finder		C	Jun 18, 2013
bf_project	Bug Finder		C	May 30, 2013
bug_finder_project	Bug Finder		C	Jun 10, 2013
qualify_testing	Bug Finder		C	Jun 24, 2013
CPI-Testing				

View Results Summary

1. Select the **Projects** tab.
2. To view the results summary for your project, on the **Projects** column, select the project name.

The results summary for the project appears on the webpage under the **Summary** tab. The **Confirmed Defects** column lists the number of coding rule violations or checks that you have reviewed.



From: 1.0 To: 1.0
Quality Objectives: OFF Display Mode: Review/Justification Progress (%)
# items: 8

Verification	Verification Status	Code Metrics		Coding Rules		Bug-Finder Checks		Review Progress
		Files	Lines of Code	Confirmed Defects	Violations	Confirmed Defects	Checks	
1.0	completed (PASS1)	6	570		294		20	0.0%
staticmemory.c	completed (PASS1)		50		24			0.0%
programming.c	completed (PASS1)		93		65		2	0.0%
numeric.c	completed (PASS1)		97		18			0.0%
dataflow.c	completed (PASS1)		147		70		6	0.0%
dynamicmemory.c	completed (PASS1)		165		107		12	0.0%
other.c	completed (PASS1)		18		10			0.0%

3 To view the results in more detail, select the tabs:

- **Code Metrics**: Metrics such as number of lines, header files and function calls.
- **Coding Rules**: Description of coding rule violations
- **Bug-Finder**: Description of defects detected by Polyspace Bug Finder

Related Examples

- "Set Up Polyspace Metrics" on page 1-7
- "Download Results From Polyspace Metrics" on page 6-5
- "Review and Comment Results" on page 6-16

Concepts

- "Code Metrics" on page 6-21

Download Results From Polyspace Metrics

This example shows how to download results from Polyspace Metrics. If you check the configuration option **Add to results repository** under **Distributed Computing**, after remote analysis, you can view a summary of the results in Polyspace Metrics.

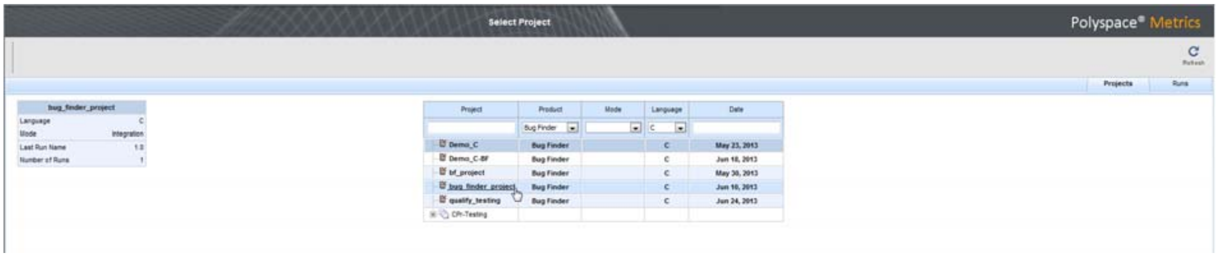
Open Polyspace Metrics

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

```
protocol:// ServerName: PortNumber
```

- *`protocol`* is either `http` (default) or `https`. To use HTTPS, you must set up the configuration file and the **Metrics and Remote Server Settings**.
- *`ServerName`* is the name or IP address of your Polyspace Metrics server.
- *`PortNumber`* is the Web server port number (default 8080)

On the webpage, you can view the projects saved to your Polyspace Metrics repository.



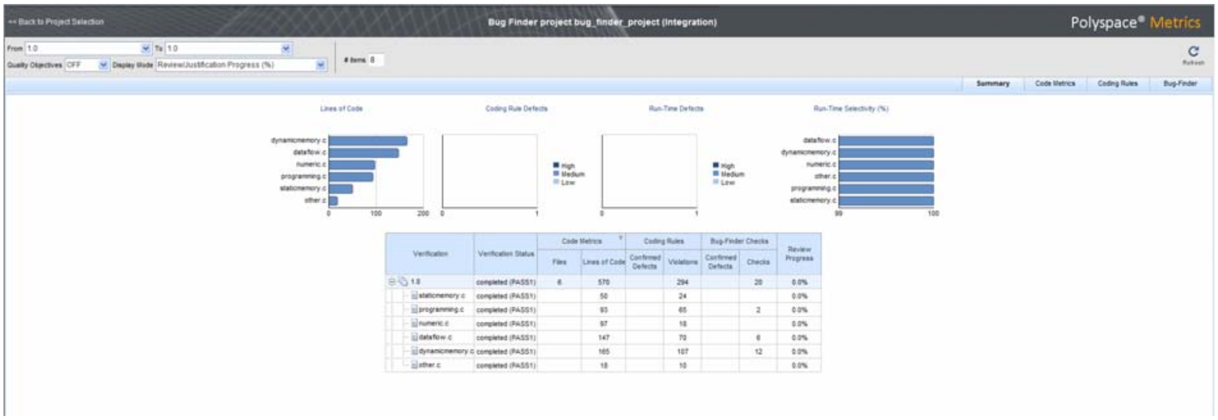
bug_finder_project	
Language	C
Mode	Integration
Last Run Name	1.0
Number of Runs	1

Project	Product	Mode	Language	Date
	Bug Finder		C	
Demo_C	Bug Finder		C	May 23, 2013
Demo_C-BF	Bug Finder		C	Jun 18, 2013
bf_project	Bug Finder		C	May 30, 2013
bug_finder_project	Bug Finder		C	Jun 10, 2013
quality_testing	Bug Finder		C	Jun 24, 2013
CPI-Testing				

Download Results

1. Select the **Projects** tab.
2. To view the results summary for your project, on the **Projects** column, select the project name.

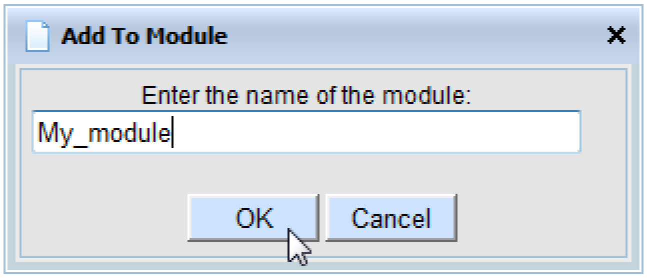
   The results summary for the project appears on the webpage under the **Summary** tab.



Verification	Verification Status	Code Metrics		Coding Rules		Bug-Finder Checks		Review Progress
		Files	Lines of Code	Confirmed Defects	Violations	Confirmed Defects	Checks	
1.0	completed (PASS1)	6	570		294		20	0.0%
staticmemory.c	completed (PASS1)		50		24			0.0%
programming.c	completed (PASS1)		93		65		2	0.0%
numeric.c	completed (PASS1)		97		18			0.0%
dataflow.c	completed (PASS1)		147		70		6	0.0%
dynamicmemory.c	completed (PASS1)		165		107		12	0.0%
other.c	completed (PASS1)		18		10			0.0%

3 To download results:

- For an individual file, on the **Verification** column, select the name of the file.
- For a group of files:
  - **a** Right-click on the row containing a file in the group. From the context menu, select **Add To Module...**.
  - **b** Enter the name of your module in the dialog box. Click **OK**.



    The name of the module appears on the **Verification** column.
  - **c** Drag and drop the other files in the group to the module.
  - **d** Select the name of the module.
- For all files in the project, on the **Verification** column, select the version number of the project.

The results open in Polyspace Bug Finder Results Manager.

Related Examples

- “Set Up Polyspace Metrics” on page 1-7
- “View Results Summary in Polyspace Metrics” on page 6-3
- “Review and Comment Results” on page 6-16

Concepts

- “Code Metrics” on page 6-21

Filter and Group Results

This example shows how to filter and group defects on the **Results Summary** pane. To organize your review of results, use filters and groups when you want to:

- Review certain categories of defects in preference to others. For instance, you first want to address the defects resulting from `Missing or invalid return statement`.
- Not address the full set of coding rule violations detected by the coding rules checker.
- Review only those defects that you have already assigned a certain status. For instance, you want to review only those defects to which you have assigned the status, `Investigate`.
- Review defects from a particular file or function. Because of continuity of code, reviewing these defects together can help you organize your review process.

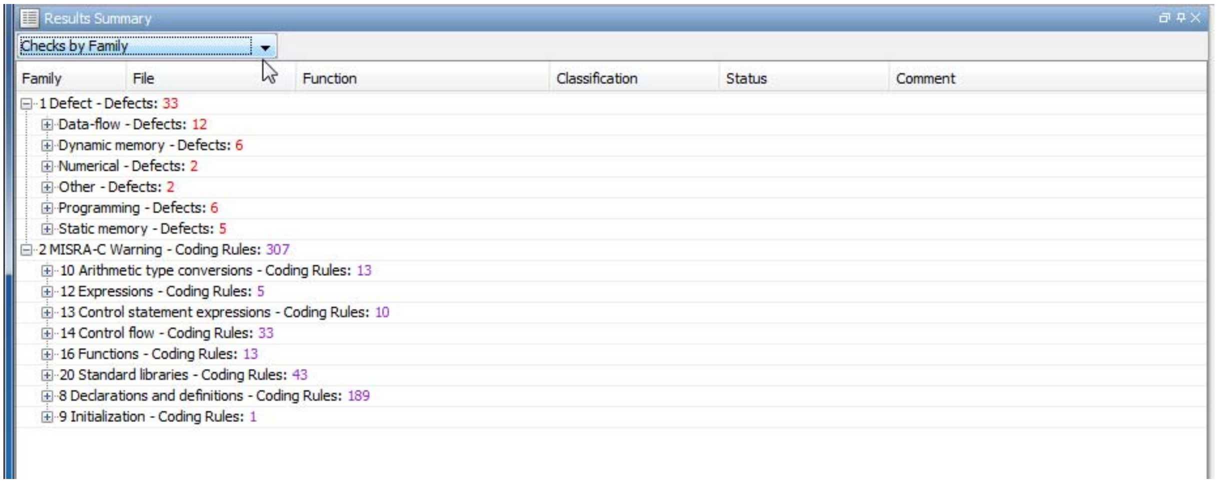
  If you have written the code for a particular source file, you can review the defects only in that file.

Review Defects in a Given Category

To review defects resulting from `Array access out of bounds`:

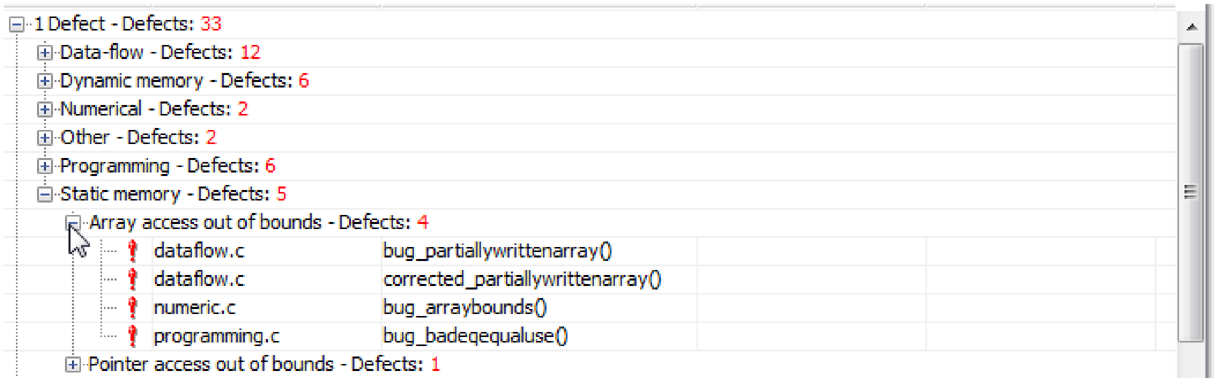
1. Open the results file, with extension, `.psbf`.
2. On the **Results Summary** pane, from the drop-down list, select `Checks by Family`.

   The defects are grouped by type.



```
1 Defect - Defects: 33
  Data-flow - Defects: 12
  Dynamic memory - Defects: 6
  Numerical - Defects: 2
  Other - Defects: 2
  Programming - Defects: 6
  Static memory - Defects: 5
2 MISRA-C Warning - Coding Rules: 307
  10 Arithmetic type conversions - Coding Rules: 13
  12 Expressions - Coding Rules: 5
  13 Control statement expressions - Coding Rules: 10
  14 Control flow - Coding Rules: 33
  16 Functions - Coding Rules: 13
  20 Standard libraries - Coding Rules: 43
  8 Declarations and definitions - Coding Rules: 189
  9 Initialization - Coding Rules: 1
```

3 Under the category **Static memory**, expand the subcategory **Array access out of bounds**.



```
1 Defect - Defects: 33
  Data-flow - Defects: 12
  Dynamic memory - Defects: 6
  Numerical - Defects: 2
  Other - Defects: 2
  Programming - Defects: 6
  Static memory - Defects: 5
    Array access out of bounds - Defects: 4
      dataflow.c        bug_partiallywrittenarray()
      dataflow.c        corrected_partiallywrittenarray()
      numeric.c         bug_arraybounds()
      programming.c     bug_badeqequaluse()
    Pointer access out of bounds - Defects: 1
```

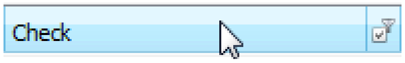
Expand **Array access out of bounds** to view all instances of this defect type.

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

4 To view only the defects resulting from `Out of bounds array index`, on the **Results Summary** pane, from the drop-down list, select `List of Checks`.

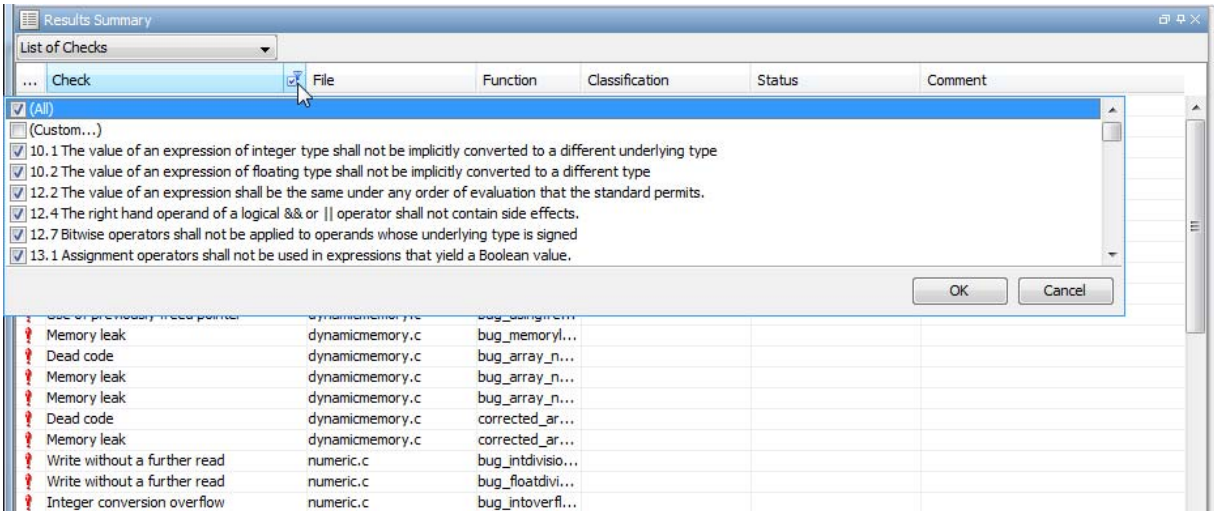
The defects appear ungrouped.

5 Place your cursor on the **Check** column head. The filter icon appears.



6 Click the filter icon.

A context menu lists the filter options available.



...	Check	File	Function	Classification	Status	Comment
	Memory leak	dynamicmemory.c	bug_memoryl...			
	Dead code	dynamicmemory.c	bug_array_n...			
	Memory leak	dynamicmemory.c	bug_array_n...			
	Memory leak	dynamicmemory.c	bug_array_n...			
	Dead code	dynamicmemory.c	corrected_ar...			
	Memory leak	dynamicmemory.c	corrected_ar...			
	Write without a further read	numeric.c	bug_intdivisio...			
	Write without a further read	numeric.c	bug_floatdivi...			
	Integer conversion overflow	numeric.c	bug_intoverfl...			

- ☑ (All)
- ☐ (Custom...)
- ☑ 10.1 The value of an expression of integer type shall not be implicitly converted to a different underlying type
- ☑ 10.2 The value of an expression of floating type shall not be implicitly converted to a different type
- ☑ 12.2 The value of an expression shall be the same under any order of evaluation that the standard permits.
- ☑ 12.4 The right hand operand of a logical && or || operator shall not contain side effects.
- ☑ 12.7 Bitwise operators shall not be applied to operands whose underlying type is signed
- ☑ 13.1 Assignment operators shall not be used in expressions that yield a Boolean value.

7 Clear the **All** check box.

8 Select the **Array access out of bounds** check box. Click **OK**.

The **Results Summary** pane displays only the defects resulting from the `Array access out of bounds` error.

Review Defects with Given Status

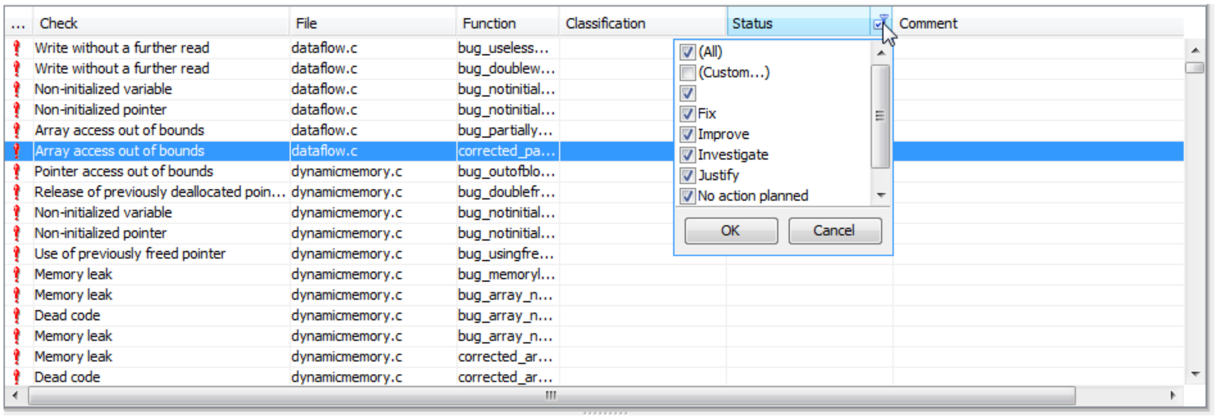
To review only the defects with `Investigate` status:

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

2 On the **Results Summary** pane, place your cursor on the **Status** column head.

3 Click the filter icon.

A context menu lists the filter options available.



...	Check	File	Function	Classification	Status	Comment
	Write without a further read	dataflow.c	bug_useless...			
	Write without a further read	dataflow.c	bug_doublew...			
	Non-initialized variable	dataflow.c	bug_notinitial...			
	Non-initialized pointer	dataflow.c	bug_notinitial...			
	Array access out of bounds	dataflow.c	bug_partially...			
	Array access out of bounds	dataflow.c	corrected_pa...			
	Pointer access out of bounds	dynamicmemory.c	bug_outofblo...			
	Release of previously deallocated poin...	dynamicmemory.c	bug_doublefr...			
	Non-initialized variable	dynamicmemory.c	bug_notinitial...			
	Non-initialized pointer	dynamicmemory.c	bug_notinitial...			
	Use of previously freed pointer	dynamicmemory.c	bug_usingfre...			
	Memory leak	dynamicmemory.c	bug_memoryl...			
	Memory leak	dynamicmemory.c	bug_array_n...			
	Dead code	dynamicmemory.c	bug_array_n...			
	Memory leak	dynamicmemory.c	bug_array_n...			
	Memory leak	dynamicmemory.c	corrected_ar...			
	Dead code	dynamicmemory.c	corrected_ar...			

4 Clear the **All** check box.

5 Select the **Investigate** check box. Click **OK**.

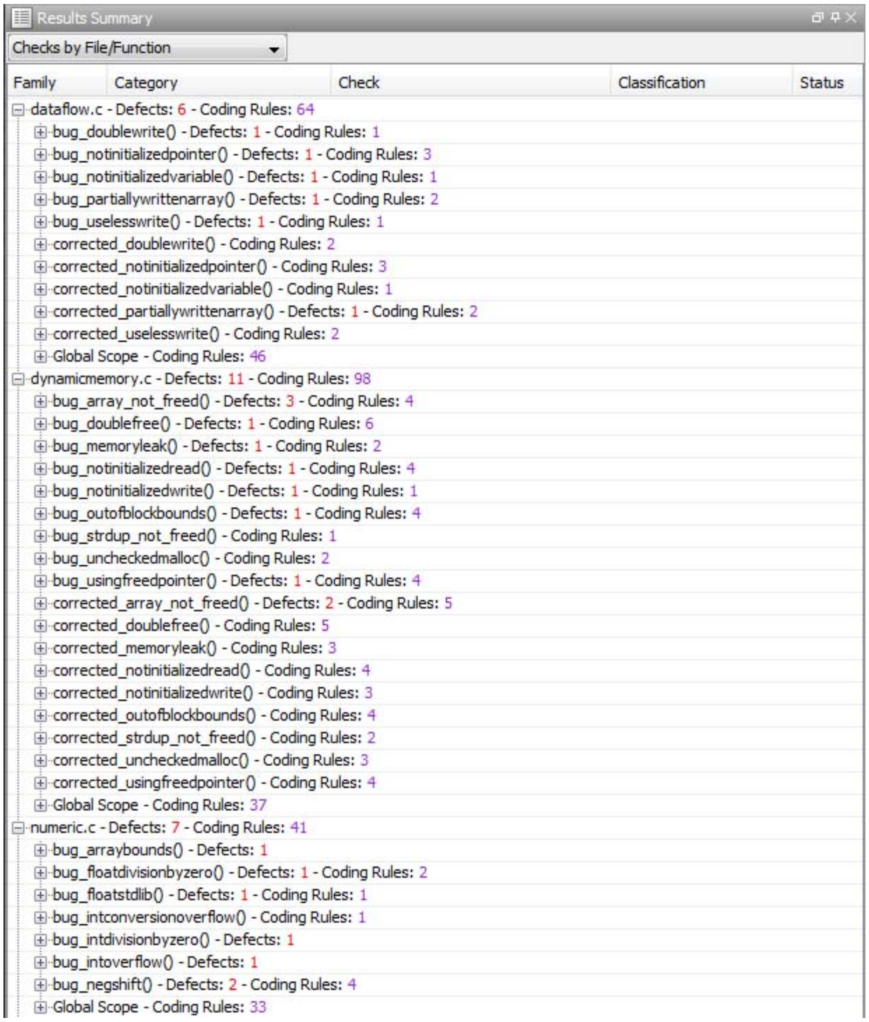
The **Results Summary** pane displays only the defects with the `Investigate` status.

Review Defects in a File

To review the defects in the file, `dataflow.c`:

1 On the **Results Summary** pane, from the drop-down list, select `Checks by File/Function`.

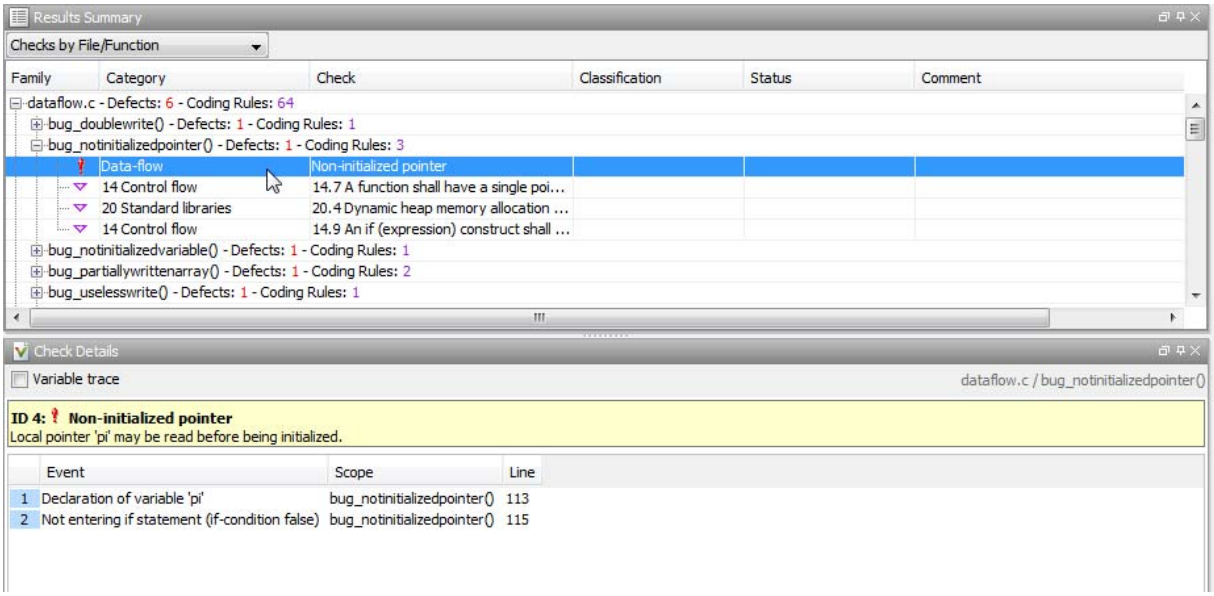
The defects displayed are grouped by files. The file names are sorted alphabetically. Within each file name, the defects are grouped by functions, sorted alphabetically.



```
dataflow.c - Defects: 6 - Coding Rules: 64
  bug_doublewrite() - Defects: 1 - Coding Rules: 1
  bug_notinitializedpointer() - Defects: 1 - Coding Rules: 3
  bug_notinitializedvariable() - Defects: 1 - Coding Rules: 1
  bug_partiallywrittenarray() - Defects: 1 - Coding Rules: 2
  bug_uselesswrite() - Defects: 1 - Coding Rules: 1
  corrected_doublewrite() - Coding Rules: 2
  corrected_notinitializedpointer() - Coding Rules: 3
  corrected_notinitializedvariable() - Coding Rules: 1
  corrected_partiallywrittenarray() - Defects: 1 - Coding Rules: 2
  corrected_uselesswrite() - Coding Rules: 2
  Global Scope - Coding Rules: 46
dynamicmemory.c - Defects: 11 - Coding Rules: 98
  bug_array_not_freed() - Defects: 3 - Coding Rules: 4
  bug_doublefree() - Defects: 1 - Coding Rules: 6
  bug_memoryleak() - Defects: 1 - Coding Rules: 2
  bug_notinitializedread() - Defects: 1 - Coding Rules: 4
  bug_notinitializedwrite() - Defects: 1 - Coding Rules: 1
  bug_outofblockbounds() - Defects: 1 - Coding Rules: 4
  bug_strdup_not_freed() - Coding Rules: 1
  bug_uncheckedmalloc() - Coding Rules: 2
  bug_usingfreedpointer() - Defects: 1 - Coding Rules: 4
  corrected_array_not_freed() - Defects: 2 - Coding Rules: 5
  corrected_doublefree() - Coding Rules: 5
  corrected_memoryleak() - Coding Rules: 3
  corrected_notinitializedread() - Coding Rules: 4
  corrected_notinitializedwrite() - Coding Rules: 3
  corrected_outofblockbounds() - Coding Rules: 4
  corrected_strdup_not_freed() - Coding Rules: 2
  corrected_uncheckedmalloc() - Coding Rules: 3
  corrected_usingfreedpointer() - Coding Rules: 4
  Global Scope - Coding Rules: 37
numeric.c - Defects: 7 - Coding Rules: 41
  bug_arraybounds() - Defects: 1
  bug_floatdivisionbyzero() - Defects: 1 - Coding Rules: 2
  bug_floatstdlib() - Defects: 1 - Coding Rules: 1
  bug_intconversionoverflow() - Coding Rules: 1
  bug_intdivisionbyzero() - Defects: 1
  bug_intoverflow() - Defects: 1
  bug_negshift() - Defects: 2 - Coding Rules: 4
  Global Scope - Coding Rules: 33
```

2 To view the defects in `dataflow.c`, expand a function name under the category, **dataflow.c**.

To view further information on a bug, select the bug. The information on the bug appears on the **Check Details** pane.



Results Summary

Checks by File/Function

Family	Category	Check	Classification	Status	Comment
dataflow.c - Defects: 6 - Coding Rules: 64					
bug_doublewrite() - Defects: 1 - Coding Rules: 1					
bug_notinitializedpointer() - Defects: 1 - Coding Rules: 3					
	Data-flow	Non-initialized pointer			
	14 Control flow	14.7 A function shall have a single poi…			
	20 Standard libraries	20.4 Dynamic heap memory allocation …			
	14 Control flow	14.9 An if (expression) construct shall …			
bug_notinitializedvariable() - Defects: 1 - Coding Rules: 1					
bug_partiallywrittenarray() - Defects: 1 - Coding Rules: 2					
bug_uselesswrite() - Defects: 1 - Coding Rules: 1					

Check Details

Variable trace

dataflow.c / bug_notinitializedpointer()

ID 4: **Non-initialized pointer**
Local pointer 'pi' may be read before being initialized.

	Event	Scope	Line
1	Declaration of variable 'pi'	bug_notinitializedpointer()	113
2	Not entering if statement (if-condition false)	bug_notinitializedpointer()	115

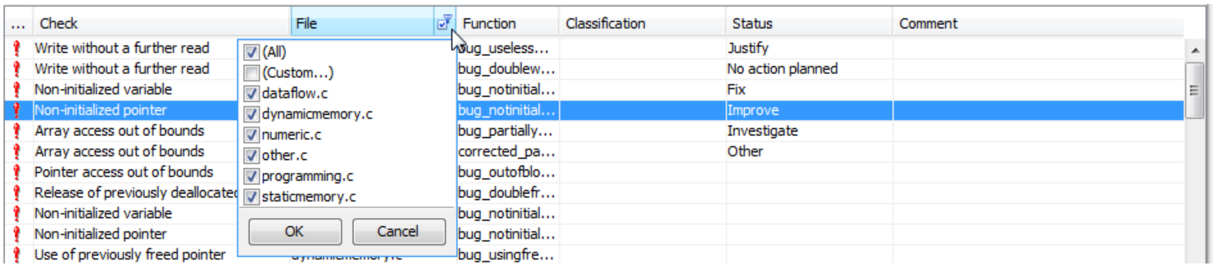
3 To view only the defects in `dataflow.c`, on the **Results Summary** pane, from the drop-down list, select `List of Checks`.

The **Results Summary** pane displays defects ungrouped.

4 Place your cursor on the **File** column head.

5 Click the filter icon.

A context menu lists the filter options available.



...	Check	File	Function	Classification	Status	Comment
	Write without a further read		bug_useless…		Justify	
	Write without a further read		bug_doublew…		No action planned	
	Non-initialized variable		bug_notinitial…		Fix	
	Non-initialized pointer		bug_notinitial…		Improve	
	Array access out of bounds		bug_partially…		Investigate	
	Array access out of bounds		corrected_pa…		Other	
	Pointer access out of bounds		bug_outofblo…			
	Release of previously deallocated		bug_doublefr…			
	Non-initialized variable		bug_notinitial…			
	Non-initialized pointer		bug_notinitial…			
	Use of previously freed pointer		bug_usingfre…			

6 Clear the **All** check box.

7 Select the **dataflow.c** check box. Click **OK**.

   The **Results Summary** pane displays only the defects in `dataflow.c`.

Tip If you apply a filter on a column on the **Results Summary** pane, the column header displays the number of check boxes selected in the filter menu. Use this information to keep track of filters you applied.

Related Examples

- “Open Results” on page 6-2
- “Review and Comment Results” on page 6-16

Concepts

- “Windows in the Results Manager Perspective” on page 6-33

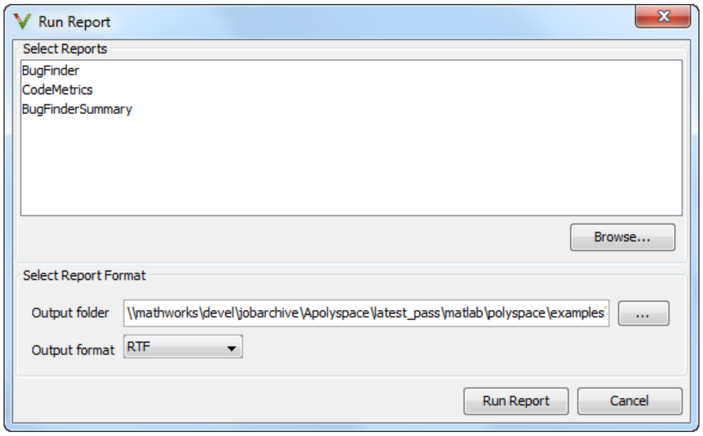
Generate Reports

This example shows how to generate reports for a Polyspace Bug Finder analysis.

1. Open your results file in the Results Manager perspective.

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

   The Run Report dialog box opens.



Output folder \\mathworks\devel\jobarchive\Apolyspace\latest_pass\matlab\polyspace\examples

Output format RTF

3. In the **Select Reports** section, select the types of reports that you want to generate. For example, you can select **BugFinder** and **CodeMetrics**.

4. Select the Output folder in which to save the report.

5. Select the Output format for the report.

6. Click **Run Report**.

   The software creates the specified report and opens it.

See Also

"Generate report" **|** "Report template" **|** "Output format"

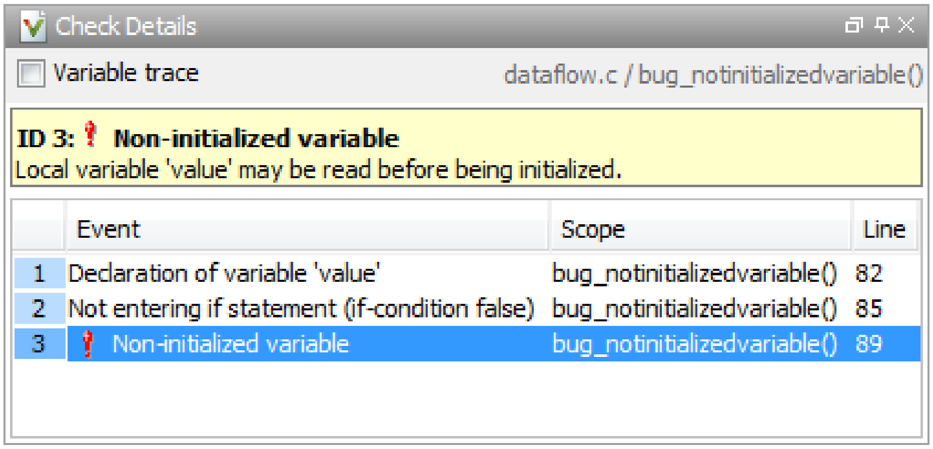
Review and Comment Results

This example shows how to review and comment results using the Results Manager perspective. When reviewing results, you can assign a status to the defects 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 defect twice.

Review and Comment Individual Defect

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

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



ID 3: Non-initialized variable
Local variable 'value' may be read before being initialized.

	Event	Scope	Line
1	Declaration of variable 'value'	bug_notinitializedvariable()	82
2	Not entering if statement (if-condition false)	bug_notinitializedvariable()	85
3	Non-initialized variable	bug_notinitializedvariable()	89

2 On the **Results Summary** pane, enter a **Classification** for the defect to describe its severity:

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

3 On the **Results Summary** pane, enter a **Status** to describe how you intend to address the defect:

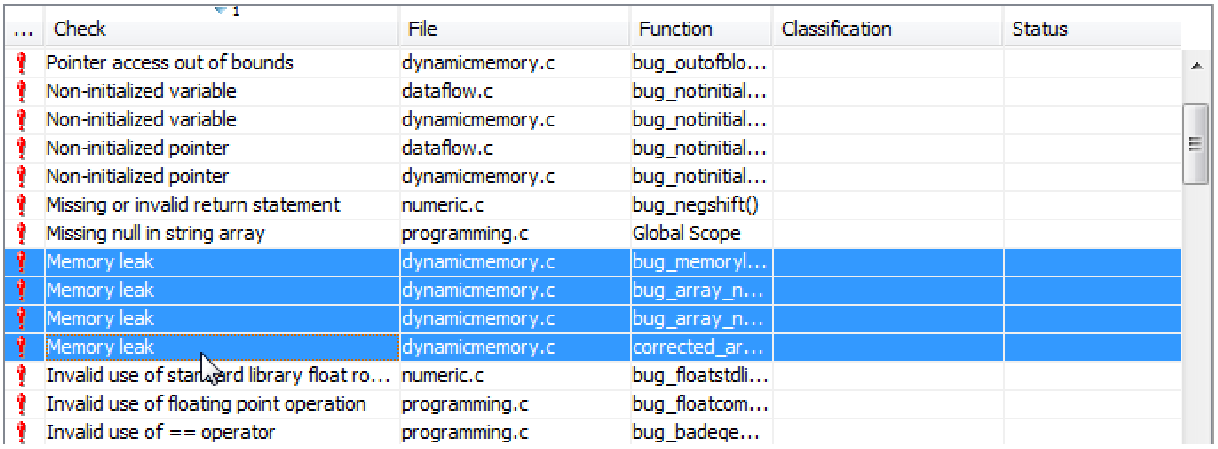
- `Fix`
- `Improve`
- `Investigate`
- `Justify`
- `No action planned`
- `Other`

4 On the **Results Summary** pane, enter remarks in the **Comment** field, for example, defect or justification information.

Review and Comment Group of Defects

1 On the **Results Summary** pane, select a group of defects using one of the following methods:

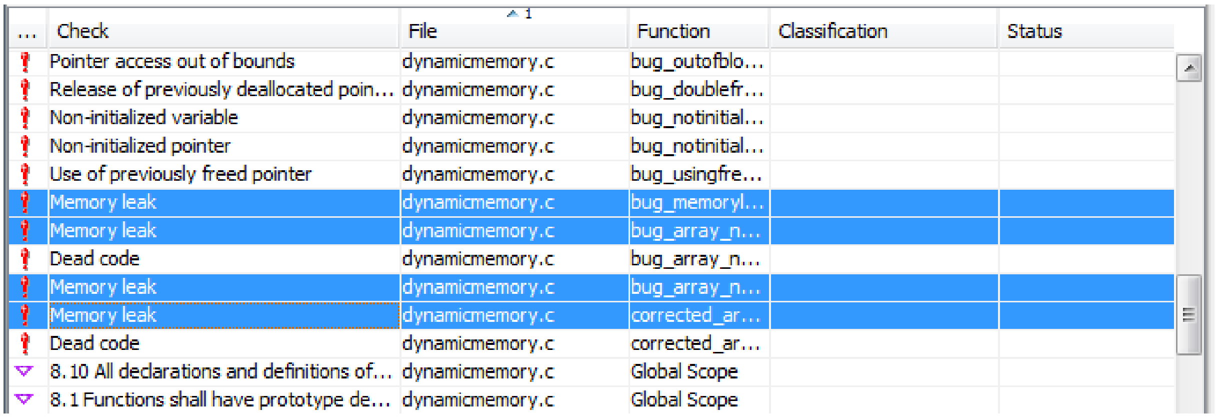
- For contiguous defects, select the first defect. Then **Shift**-select the last defect.



...	Check	File	Function	Classification	Status
!	Pointer access out of bounds	dynamicmemory.c	bug_outofblo...		
!	Non-initialized variable	dataflow.c	bug_notinitial...		
!	Non-initialized variable	dynamicmemory.c	bug_notinitial...		
!	Non-initialized pointer	dataflow.c	bug_notinitial...		
!	Non-initialized pointer	dynamicmemory.c	bug_notinitial...		
!	Missing or invalid return statement	numeric.c	bug_negshift()		
!	Missing null in string array	programming.c	Global Scope		
!	Memory leak	dynamicmemory.c	bug_memoryl...		
!	Memory leak	dynamicmemory.c	bug_array_n...		
!	Memory leak	dynamicmemory.c	bug_array_n...		
!	Memory leak	dynamicmemory.c	corrected_ar...		
!	Invalid use of standard library float ro...	numeric.c	bug_floatstdli...		
!	Invalid use of floating point operation	programming.c	bug_floatcom...		
!	Invalid use of == operator	programming.c	bug_badeqe...		

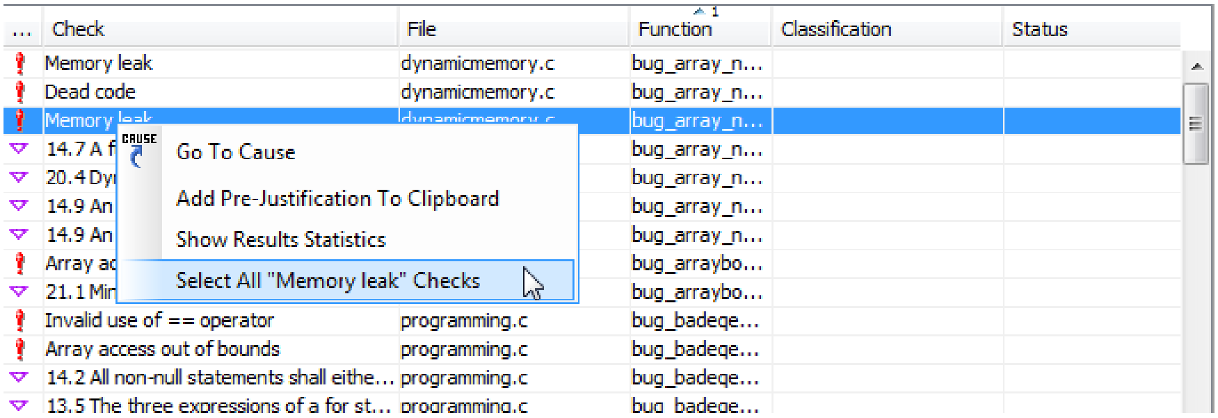
To group together the defects that belong to a certain category, click the **Check** column header on the **Results Summary** pane.

- For non-contiguous defects, **Ctrl**-select each defect.



...	Check	File	Function	Classification	Status
!	Pointer access out of bounds	dynamicmemory.c	bug_outofblo...		
!	Release of previously deallocated poin...	dynamicmemory.c	bug_doublefr...		
!	Non-initialized variable	dynamicmemory.c	bug_notinitial...		
!	Non-initialized pointer	dynamicmemory.c	bug_notinitial...		
!	Use of previously freed pointer	dynamicmemory.c	bug_usingfre...		
!	Memory leak	dynamicmemory.c	bug_memoryl...		
!	Memory leak	dynamicmemory.c	bug_array_n...		
!	Dead code	dynamicmemory.c	bug_array_n...		
!	Memory leak	dynamicmemory.c	bug_array_n...		
!	Memory leak	dynamicmemory.c	corrected_ar...		
!	Dead code	dynamicmemory.c	corrected_ar...		
▽	8.10 All declarations and definitions of...	dynamicmemory.c	Global Scope		
▽	8.1 Functions shall have prototype de...	dynamicmemory.c	Global Scope		

- For defects of a similar category, right-click one defect from that category. From the context menu, select **Select All *Defect Category* Checks**, for example, **Select All "Memory leak" Checks**.


CAUSE
Go To Cause
Add Pre-Justification To Clipboard
Show Results Statistics
Select All "Memory leak" Checks

...	Check	File	Function	Classification	Status
!	Memory leak	dynamicmemory.c	bug_array_n...		
!	Dead code	dynamicmemory.c	bug_array_n...		
!	Memory leak	dynamicmemory.c	bug_array_n...		
▽	14.7 A f		bug_array_n...		
▽	20.4 Dy		bug_array_n...		
▽	14.9 An		bug_array_n...		
▽	14.9 An		bug_array_n...		
!	Array ac		bug_arraybo...		
▽	21.1 Mir		bug_arraybo...		
!	Invalid use of == operator	programming.c	bug_badeqe...		
!	Array access out of bounds	programming.c	bug_badeqe...		
▽	14.2 All non-null statements shall eithe...	programming.c	bug_badeqe...		
▽	13.5 The three expressions of a for st...	programming.c	bug_badeqe...		

2 On the **Results Summary** pane, enter **Classification**, **Status** and **Comments**. The software applies this information to all the selected defects.

Save Review Comments

After you have reviewed your results, save your comments with the analysis 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 analysis results.

Related Examples

- “Open Results” on page 6-2
- “Filter and Group Results” on page 6-8

Concepts

- “Windows in the Results Manager Perspective” on page 6-33

Import Comments from Previous Analyses

This example shows how to import review comments from previous analyses. By default, Polyspace Bug Finder automatically imports comments from the previous analysis, allowing you to avoid reviewing the same defect twice. However, you can also manually import comments into the current review

Import Comments from Previous Analysis

1. Open your most recent results in the Results Manager perspective.
2. Select **Review > Import > Import Comments**.
3. Navigate to the folder containing your previous results.
4. Select the results file with extension `.psbf`, then click **Open**.

   The review comments from the previous results are imported into the current results, and the Import checks and comments report opens.

Change Preferences for Automatically Importing Comments

1. Select **Options > Preferences**, which opens the Polyspace Preferences dialog box.
2. Select the **Project and result folder** tab.
3. Under **Import Comments**, select or clear the **Automatically import comments from last verification** check box.
4. Click **OK**.

Code Metrics

The following table provides descriptions of the columns in the **Code Metrics** view on the Polyspace Metrics webpage.

Level	Metric name	Description	HIS metric?
Project	**Files**	Number of source files.	No
	Header Files	Directly and indirectly included header files, including Polyspace internal header files and the header files included by these internal files. The number of included headers shows how many header files are verified for the current project.	No
	Recursions	Call graph recursions. Number of call cycles over one or more functions. If one function is at the same time directly recursive (it calls itself) and indirectly recursive, the call cycle is counted only once. Call cycle through pointer is not considered.	Yes
	Direct Recursions	Number of direct recursions.	Yes
	Protected Shared Variables	Number of protected shared variables. This measure is provided only from the analysis PASS0.	No
	Non-Protected Shared Variables	Number of unprotected shared variables. This measure is provided only from the analysis `PASS0`.	No

Level	Metric name	Description	HIS metric?
File	**Lines**	Number of lines. Physical lines including comment and blank lines	No
	Lines of Code	Number of lines without comment, that is, lines excluding blank or comment lines. A line that contains code and comment is counted.	No
	Comment Density	Relationship of the number of comments (outside and within functions) to the number of statements. An internal comment is a comment that begins and/or ends with the source code line; otherwise a comment is considered external. In the comment density calculation, the comments in the header file (before the first preprocessing directive or the first token in the source file) are ignored. Two comments that are not separated by a token are considered as one occurrence. The number of statements within a file is the number of semicolons in the preprocessed source code except within `for` loops, structure or union field definitions, comments, literal strings, preprocessing directives, or parameters lists in the scope of K & R style function declarations. The comment density is: number of external comment occurrences / number of statements	Yes
	Estimated Function Coupling	Inter-file dependency. Metric is equal to: sum of call occurrences – number of functions defined in the file + 1.	No

Level	Metric name	Description	HIS metric?
		The function coupling is calculated in a preprocessed file.	
Function	**Lines Within Body**	Total number of lines in a function body, including blank and comment lines: number of lines between the first `{` and the last `}` of a function body. The number of lines within a function body is calculated in the preprocessed file. If a function body contains an `#include` directive, the included file source code is taken into account in the calculation of the lines of this function. The preprocessing directives lines are taken into account in the calculation of the lines.	No
	Executable Lines	Total number of lines with source code tokens between a function body '`{`' and '`}`' that are not declarations (w/o static initializer), comments, braces, or preprocessing directives. The number of execution lines within a function body is calculated in a preprocessed file. If the function body contains an `#include` directive, the included file source code is taken into account in the calculation of the execution lines of this function.	No
	Cyclomatic Complexity	Number of decisions + 1. The `?:` operator is considered a decision, but the combination of `&& ||` is considered to be only one decision.	Yes
	Language Scope	The language scope is an indicator of the cost of maintaining or changing functions. Metric value = (N1+N2) / (n1+n2)	Yes

Level	Metric name	Description	HIS metric?

		where:
		n1 = number of different operators
		N1 = sum of all operators
		n2 = number of different operands
		N2 = sum of all operands
		The computation is based on the preprocessed source code.
		Consider the following code.
		`int f(int i)`
		`{`
		` if (i == 1)`
		`   return i;`
		` else`
		`  return i * g(i-1);`
		`}`
		In this code, the:
		• Distinct operators are `int`, (,),{, `if`, ==, `return`, `else`, *, -, ;, }
		• Number of operators is 12
		• Number of operator occurrences is 17
		• Distinct operands are `f`, `i`, `1`, `g`
		• Number of operands is 4
		• Number of operand occurrences is 9
		For this example, the metric value is:
		`1.8 ((17 + 9) / (12 + 4))`	

	Paths	Estimated static path count. The following code contains one path.	Yes

Level	Metric name	Description	HIS metric?

		`switch (n)`
		`     {`
		`      case 1:`
		`      case 2:`
		`      case 3:`
		`      case 4:`
		`      default:`
		`        break;`
		`      }`
		
		The following code contains two paths.
		
		`switch (n)`
		`     {`
		`      case 1:`
		`      case 2:`
		`        break;`
		`      case 3:`
		`      case 4:`
		`      default:`
		`        break;`
		`      }`
		
		Implicit `else` is considered as one path.
		
		This value is not computed when a `goto` exists within the function body.	

	Calling Functions	Number of distinct callers of a function. Call through pointer is not considered.	Yes
	Called Functions	Number of distinct functions called by a function. Call through pointer is not considered. See description for **Call Occurrences**	Yes
	Call Occurrences	Number of call occurrences within function body.	No

Level	Metric name	Description	HIS metric?

		Similar to **Called Functions** except that each call of a function is counted.
		
		Consider the following code.
		
		`int callee_1() {return 0;}`
		`int callee_2() {return 0;}`
		
		`int get()`
		`{`
		`  return  callee_1() + callee_1() + callee_2() + callee_2();`
		`}`
		
		For this code, the **Called Functions** value is 2 but the **Call Occurrences** value is 4.	

	Instructions	Number of instructions per function, which is a measure of function complexity. Let STMT(*function_code_element*) represent the metric value for *function_code_element*. The following applies: STMT (*simple_statement*) = 1 STMT (*empty_statement*) = 0 STMT (*label*) = 0 STMT (*block*) = STMT (*block_body*) STMT (*declaration_ without_initializer*) = 0; STMT (*declaration_with_ initializer)* = 1; STMT (*other_statements*) = 1 where *other_statements* are `break`, `continue`, `do-while`, `for`, `goto`, `if`, `return`, `switch`, `while`.	Yes

Level	Metric name	Description	HIS metric?
	Call Levels	Depth of function nesting. Maximum depth of control structures within a function body. The value of 1 means either control structure do not exist within a function body or existing control structures are not nested within another control structure.	Yes
	Function Parameters	Number of parameters per function. A measure of the complexity of the function interface. Ellipsis (`...`) parameter is ignored.	Yes
	Goto Statements	Number of `goto` statements within a function. `break` and `continue` are not counted as `goto` statements. If this value is > 0, the number of **Paths** cannot be computed.	Yes
	Return Points	Number of `return` points within a function. Number of explicit `return` statements within a function body. The following function has zero return points: `void f(void) {}`,The following function has one return point: `void f(void) {return;}`	Yes

View Code Sequence Causing Defect

This example shows how to view the code sequence that is probably causing a defect in the Results Manager perspective. The example uses the following code, which contains the defect `Non-initialized pointer`:

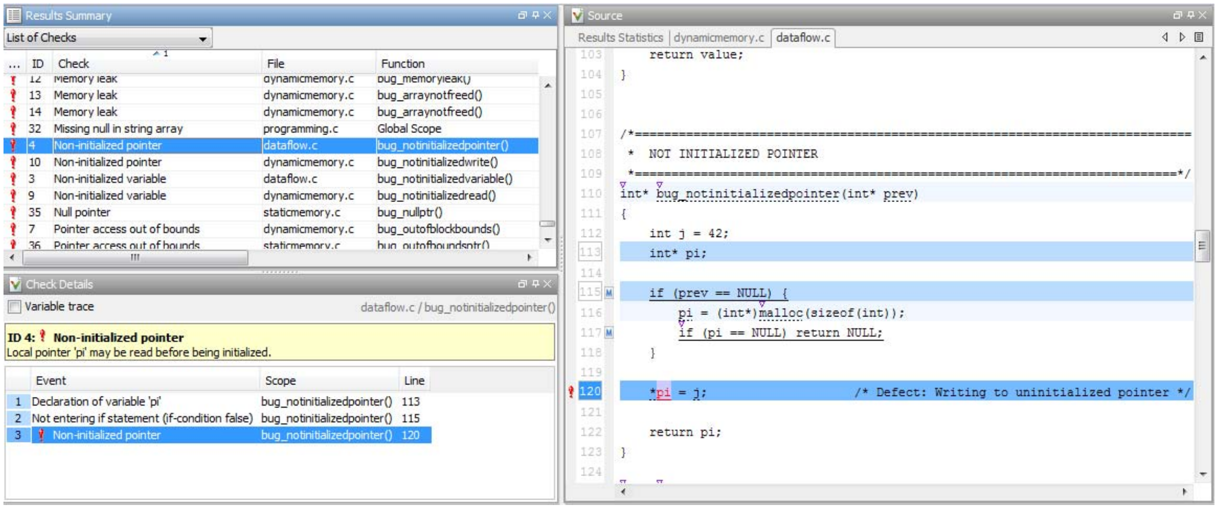
```
#include <stdlib.h>

 int* assign_value_and_return_address(int* prev, int val)
{
    int* pi;

    if (prev == NULL) {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
    }

    *pi = val;
    /* Defect: Writing to uninitialized pointer */

    return pi;
}
```

The code is stored in a source file `store_value.c`.

1 Run a Polyspace Bug Finder analysis on `store_value.c`.

2 Open the results file with extension `.psbf`.

3 On the **Results Summary** pane, select the defect **Non-initialized pointer**.



...	ID	Check	File	Function
	12	Memory leak	dynamicmemory.c	bug_memoryleak()
	13	Memory leak	dynamicmemory.c	bug_arraynotfreed()
	14	Memory leak	dynamicmemory.c	bug_arraynotfreed()
	32	Missing null in string array	programming.c	Global Scope
	4	Non-initialized pointer	dataflow.c	bug_notinitializedpointer()
	10	Non-initialized pointer	dynamicmemory.c	bug_notinitializedwrite()
	3	Non-initialized variable	dataflow.c	bug_notinitializedvariable()
	9	Non-initialized variable	dynamicmemory.c	bug_notinitializedread()
	35	Null pointer	staticmemory.c	bug_nullptr()
	7	Pointer access out of bounds	dynamicmemory.c	bug_outofblockbounds()
	36	Pointer access out of bounds	staticmemory.c	bug_outofboundsptr()

dataflow.c / bug_notinitializedpointer()

ID 4: Non-initialized pointer
Local pointer 'pi' may be read before being initialized.

	Event	Scope	Line
1	Declaration of variable 'pi'	bug_notinitializedpointer()	113
2	Not entering if statement (if-condition false)	bug_notinitializedpointer()	115
3	Non-initialized pointer	bug_notinitializedpointer()	120

```
103      return value;
104  }
105
106
107  /*==========================================================================
108   *  NOT INITIALIZED POINTER
109   *=========================================================================*/
110  int* bug_notinitializedpointer(int* prev)
111  {
112      int j = 42;
113      int* pi;
114
115      if (prev == NULL) {
116          pi = (int*)malloc(sizeof(int));
117          if (pi == NULL) return NULL;
118      }
119
120      *pi = j;                    /* Defect: Writing to uninitialized pointer */
121
122      return pi;
123  }
124
```

- The code line containing the defect is highlighted in dark blue on the **Source** pane. More information on the defect is available on the **Check Details** pane.

- The following columns describe the sequence of code instructions causing the defect:

  a **Event**: Instruction causing the defect

  b **Scope**: Function containing instruction

  c **Line**: Line number of instruction
  These instructions are also highlighted in medium blue on the **Source** pane. The corresponding line numbers are marked by squares. Place your cursor over a square to view a tooltip. The tooltip describes how the instruction is possibly related to the defect.

- Other instructions that can possibly impact the defect are highlighted in light blue on the **Source** pane. To see these instructions on the **Check Details** pane, select the **Variable trace** check box.

4 To navigate to an instruction from the probable code sequence in the source code, select the instruction on the **Event** column. The corresponding line is highlighted on the **Source** pane.

Related Examples

- "Run Local Analysis" on page 5-3
- "View Results Summary in Polyspace Metrics" on page 6-3
- "Review and Comment Results" on page 6-16

Concepts

- "Source" on page 6-39
- "Check Details" on page 6-45

Results Folder Contents

In this section...

"Files in the Results Folder" on page 6-31

"Files in the `ALL` Subfolder" on page 6-31

"Files in the `Polyspace-Doc` Subfolder" on page 6-32

Every time you run an analysis, Polyspace Bug Finder generates files and folders that contain information about configuration options and analysis results. The contents of results folders depend on the configuration options. To learn more about configuration options, see "Analysis Options for C".

By default, your results are saved in your project folder in a folder called `Result`. To use a different folder, see "Specify Results Folder" on page 5-6.

Files in the Results Folder

Some of the files in the results folder are described below:

- `Polyspace_`*`release_project_name_date-time`* — A log file associated with each analysis, for example, `Polyspace_R2013b_example_project_05_17_2013-12h01.log`.
- *`project_name`*`.psbf` — An ASCII file containing the location of the most recent results and log. The software uses this file to open results in the Results Manager.
- `options` — The list of options used for the most recent analysis.
- `source_list.txt` — A list of sources verified by the latest analysis.

Files in the `ALL` Subfolder

The `ALL` subfolder contains internal information that is used by Polyspace Bug Finder to show sources and checks.

- `SRC\MACROS\ci.zip` — A zip file containing expanded source files with a `.ci` suffix.

- `SRC\*.[c or h]` — Source code file included in the analysis. The file contains user source code and code generated by Polyspace Bug Finder.

Files in the `Polyspace-Doc` Subfolder

The `Polyspace-Doc` subfolder contains reports generated with the `-report-template`, `-report-output-name`, or `-report-ouput-format` options.

- `Code_Metrics.xml` — A list of metrics from the most recent analysis.

Windows in the Results Manager Perspective

In this section...

"Dashboard" on page 6-33
"Results Summary" on page 6-37
"Source" on page 6-39
"Check Details" on page 6-45

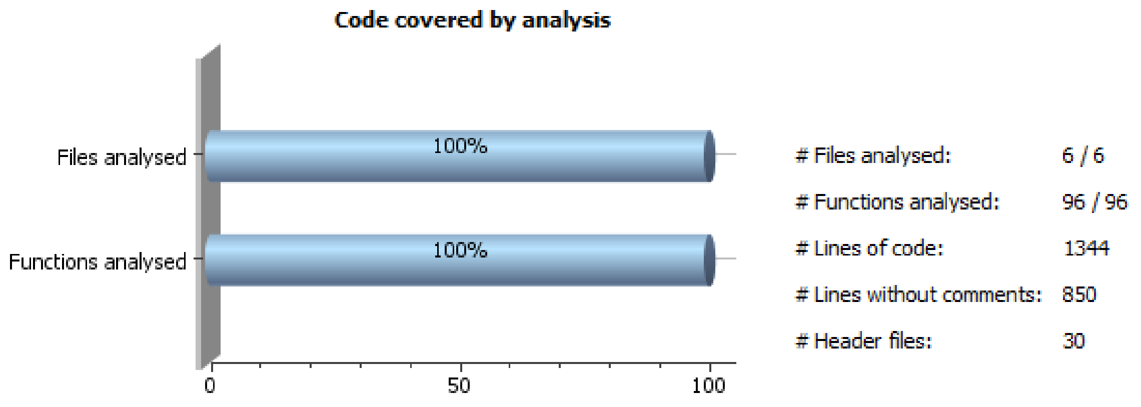
Dashboard

On the **Source** pane, the **Dashboard** tab provides statistics on the analysis results in a graphical format.

When you open a results file in the Results Manager perspective, this tab is displayed by default. You can view the following graphs:

-

  Code covered by analysis



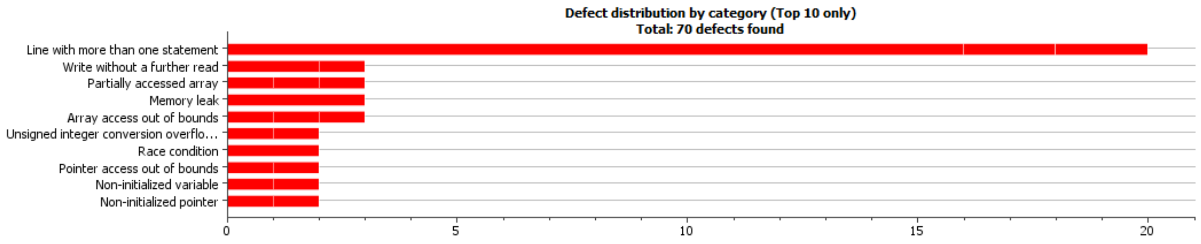
# Files analysed:	6 / 6
# Functions analysed:	96 / 96
# Lines of code:	1344
# Lines without comments:	850
# Header files:	30

From this graph you can obtain the following information:

- **# Files analysed**: Ratio of analyzed files to total number of files. If a file contains a compilation error, Polyspace Bug Finder does not analyze the file.
- **# Functions analysed**: Ratio of analyzed functions to total number of functions in the analyzed files. If the analysis of a function takes longer than a threshold value, Polyspace Bug Finder does not analyze the function.
- **# Lines of code**: Total number of code lines in source files.
- **# Lines without comments**: Total number of code lines in source files excluding lines that are only comments.
- **# Header files**: Total number of files included in your source files using `#include` directive.

•

Defect distribution by category or file

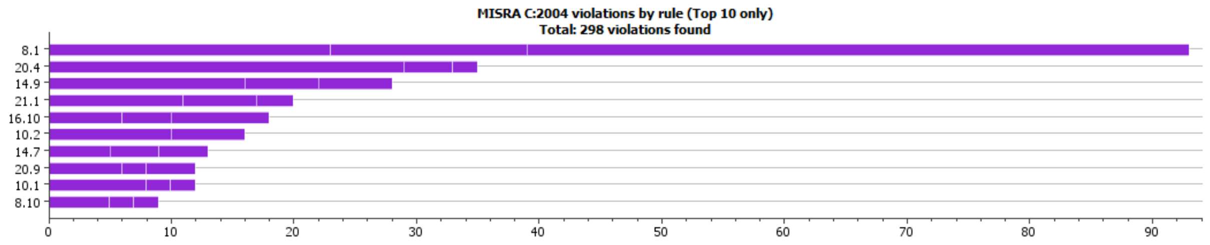


From this graph you can obtain the following information.

	Category	**File**
Top 10	The ten defect types with the highest number of individual defects. - Each column represents a defect type and is divided into the: · File with highest number of defects of this type. · File with second highest number of defects of this type. · All other files with defects of this type. Place your cursor on a column to see the file name and number of defects of this type in this file. - The x-axis represents the number of defects. Use this view to organize your check review starting at defect types with more individual defects.	The ten source files with the highest number of defects. - Each column represents a file and is divided into the: · Defect type with highest number of defects in this file. · Defect type with second highest number of defects in this file. · All other defect types in this file. Place your cursor on a column to see the defect type name and number of defects of this type in this file. - The x-axis represents the number of defects. Use this view to organize your check review starting at files with more defects.
Bottom 10	The ten defect types with the lowest number of individual defects. Each column on the graph is divided the same way as the **Top 10** defect types. Use this view to organize your check review starting at defect types with fewer individual defects.	The ten source files with the lowest number of defects. Each column on the graph is divided the same way as the **Top 10** files. Use this view to organize your check review starting at files with fewer defects.

•

Coding rule violations by rule or file



For every type of coding rule that you check (MISRA, JSF, or custom), the **Dashboard** contains a graph of the rule violations.

From this graph you can obtain the following information.

	Category	**File**
Top 10	The ten rules with the highest number of violations. - Each column represents a rule number and is divided into the: • File with highest number of violations of this rule. • File with second highest number of violations of this rule. • All other files with violations of this rule. Place your cursor on a column to see the file name and number of violations of this rule in the file. - The x-axis represents the number of rule violations.	The ten source files containing the highest number of violations. - Each column represents a file and is divided into the: • Rule with highest number of violations in this file. • Rule with second highest number of violations in this file. • All other rules violated in this file. Place your cursor on a column to see the rule number and number of violations of the rule in this file. - The x-axis represents the number of rule violations.

	Category	File
	Use this view to organize your review starting at rules with more violations.	Use this view to organize your review starting at files with more rule violations.
Bottom 10	The ten rules with the lowest number of violations. Each column on the graph is divided in the same way as the **Top 10** rules. Use this view to organize your review starting at rules with fewer violations.	The ten source files containing the lowest number of rule violations. Each column on the graph is divided in the same way as the **Top 10** files. Use this view to organize your review starting at files with fewer rule violations.

For a list of supported coding rules, see “Supported MISRA C:2004 Rules” on page 3-18, “Supported MISRA C++ Coding Rules” on page 3-69 and “Supported JSF C++ Coding Rules” on page 3-96.

Results Summary

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

- `List of checks`: Lists defects and coding rule violations in alphabetical order.
- `Checks by Family`: Lists results grouped by category. For more information on the defects covered by a category, see “Polyspace Bug Finder Defects”.
- `Checks by Class`: Lists results grouped by class. Within each class, the results are grouped by method. The first group, **Global Scope**, lists results not occurring in a class definition.

  This option is available for `C++` code only.
- `Checks by File/Function`: Lists results grouped by file. Within each file, the results are grouped by function.

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

Attribute	Description
Family	Group to which the defect belongs. For instance, if you choose the grouping `Checks by File/Function`, this column contains the name of the file and function containing the defect.
ID	Unique identification number of the defect. In the default view on the **Results Summary** pane, the defects appear sorted by this number.
Type	Defect or coding rule violation.
Category	Category of the defect. For more information on the defects covered by a category, see the defect reference pages.
Check	Description of the defect
File	File containing the instruction where the defect occurs
Class	Class containing the instruction where the defect occurs. If the defect is not inside a class definition, then this column contains the entry, `Global Scope`.
Function	Function containing the instruction where the defect occurs. If the function is a method of a class, it appears in the format *`class_name`*`::`*`function_name`*.

Attribute	Description
Classification	Level of severity you have assigned to the defect. The possible levels are: • `High` • `Medium` • `Low` • `Not a defect`
Status	Review status you have assigned to the check. The possible statuses are: • `Fix` • `Improve` • `Investigate` • `Justify` • `No action planned` • `Other`
Comments	Comments you have entered about the check

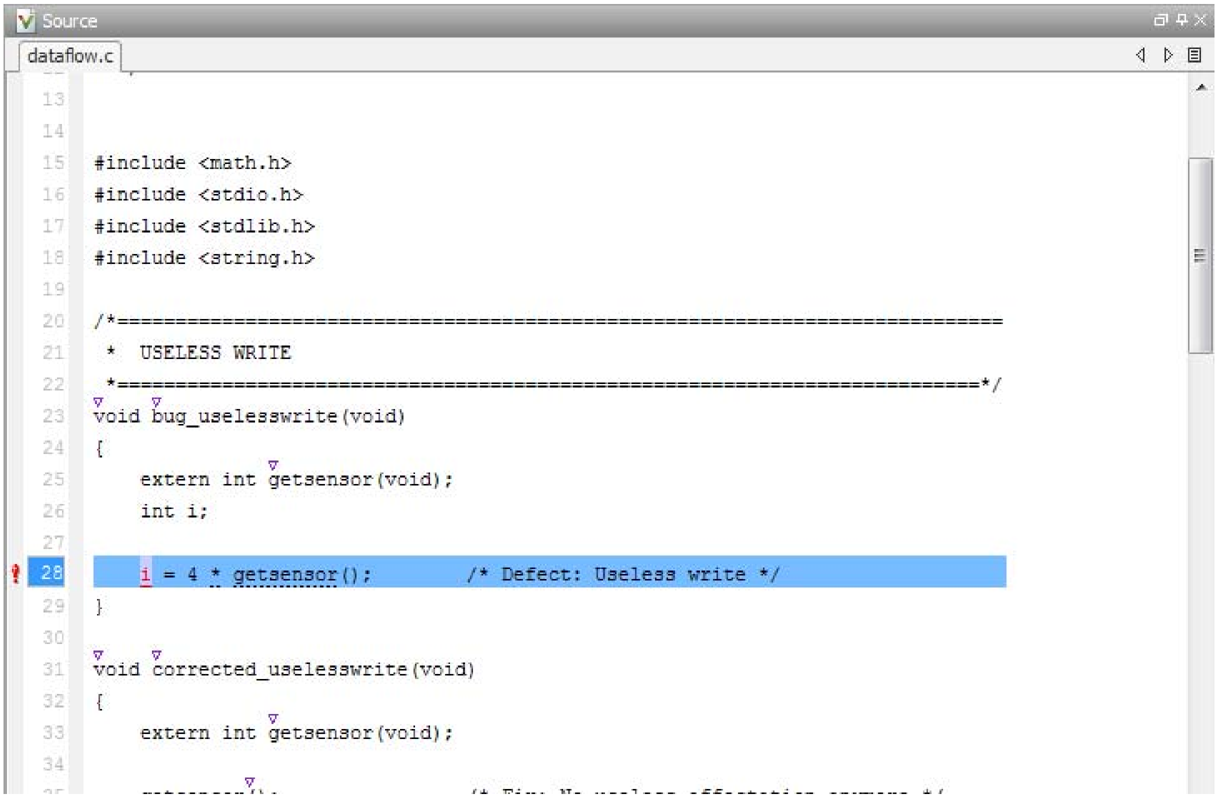
To show or hide any of the columns, right-click anywhere on the column titles. From the context menu, select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through the checks. For more information, see "Review and Comment Results" on page 6-16.
- Organize your check review using filters on the columns. For more information, see "Filter and Group Results" on page 6-8.

Source

The **Source** pane shows the source code with the defects colored in red and the corresponding line number marked by !.



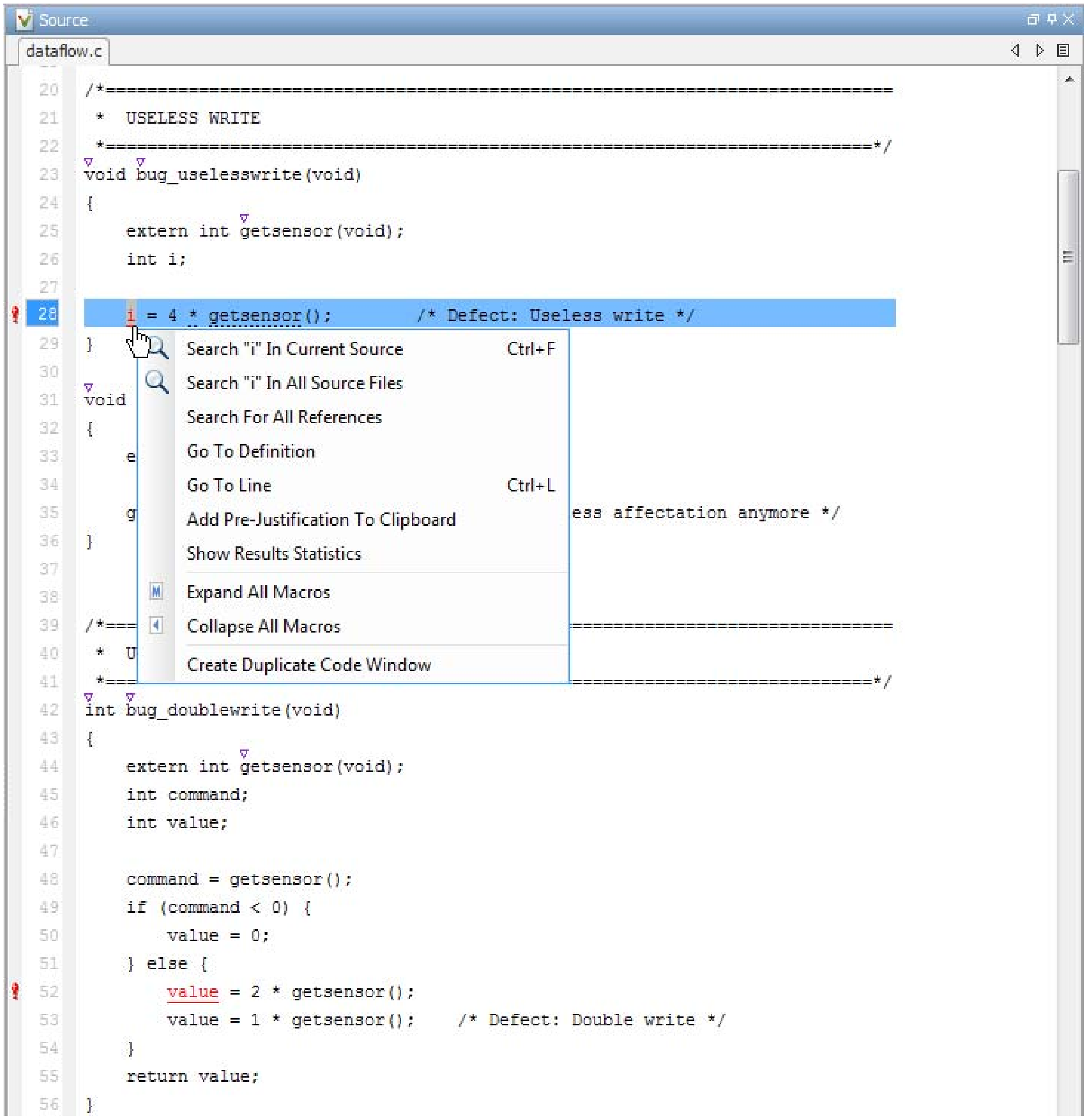
```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*==========================================================================
 *  USELESS WRITE
 *========================================================================*/
void bug_uselesswrite(void)
{
    extern int getsensor(void);
    int i;

    i = 4 * getsensor();       /* Defect: Useless write */
}

void corrected_uselesswrite(void)
{
    extern int getsensor(void);

```

Tooltips

Placing your cursor over a check displays a tooltip that provides range information for variables, operands, function parameters, and return values.

Examine Source Code

In the **Source** pane, if you right-click a text string, the context menu provides options to examine your code:


Source
dataflow.c
Search "i" In Current Source Ctrl+F
Search "i" In All Source Files
Search For All References
Go To Definition
Go To Line Ctrl+L
Add Pre-Justification To Clipboard
Show Results Statistics
Expand All Macros
Collapse All Macros
Create Duplicate Code Window

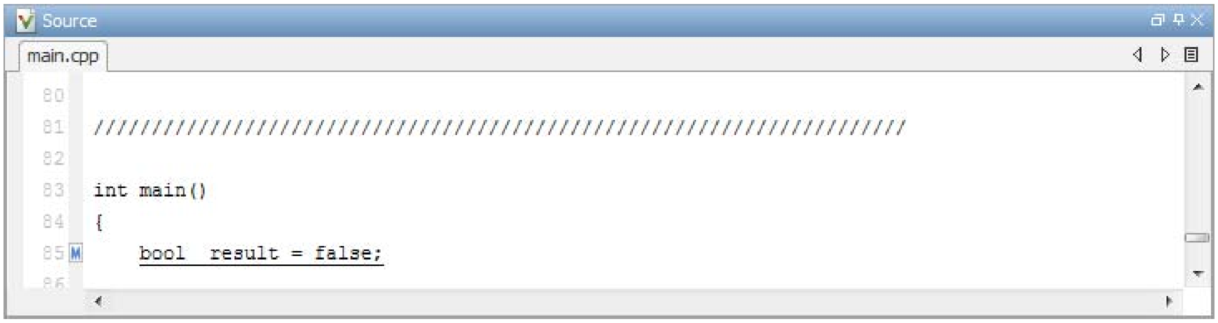
```
/*============================================================================
 *  USELESS WRITE
 *===========================================================================*/
void bug_uselesswrite(void)
{
    extern int getsensor(void);
    int i;

    i = 4 * getsensor();         /* Defect: Useless write */
}

void
{
    e
    g                                                     ess affectation anymore */
}


/*===                                                     ===============================
 *  U
 *===                                                     =============================*/
int bug_doublewrite(void)
{
    extern int getsensor(void);
    int command;
    int value;

    command = getsensor();
    if (command < 0) {
        value = 0;
    } else {
        value = 2 * getsensor();
        value = 1 * getsensor();    /* Defect: Double write */
    }
    return value;
}
```

For example, if you right-click the variable `i`, you can use the following options to examine and navigate through your code:

- **Search "i" in Current Source** — List occurrences of the string within the current source file on the **Search** pane.
- **Search "i" in All Source Files** — List occurrences of the string within the source files on the **Search** pane.
- **Search For All References** — List all references in the **Search** pane. The software supports this feature for global and local variables, functions, types, and classes.
- **Go to Definition** — Go to the line of code that contains the definition of `i`. The software supports this feature for global and local variables, functions, types, and classes.
- **Go To Line** — Open the Go to line dialog box. If you specify a line number and click **Enter**, the software displays the specified line of code.
- **Expand All Macros** or **Collapse All Macros** — Display or hide the content of macros in current source file.

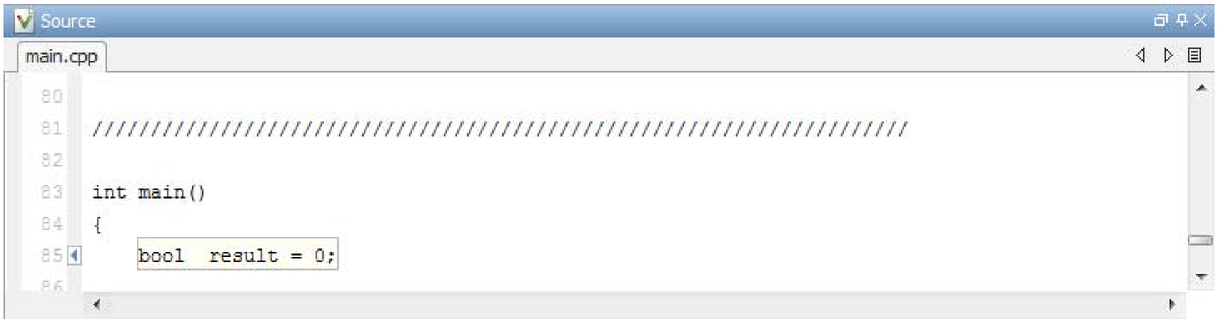
Expand Macros

You can view the contents of source code macros in the source code view. A code information bar displays M icons that identify source code lines with macros.



```
//////////////////////////////////////////////////////////////////////////

int main()
{
    bool  result = false;
```

When you click a line with this icon, the software displays the contents of macros on that line in a box.



```
////////////////////////////////////////////////////////////////////////

int main()
{
    bool  result = 0;
```

To display the normal source code again, click the line away from the box, for example, on the ◀ icon.

To display or hide the content of *all* macros:

1 Right-click the source code view.

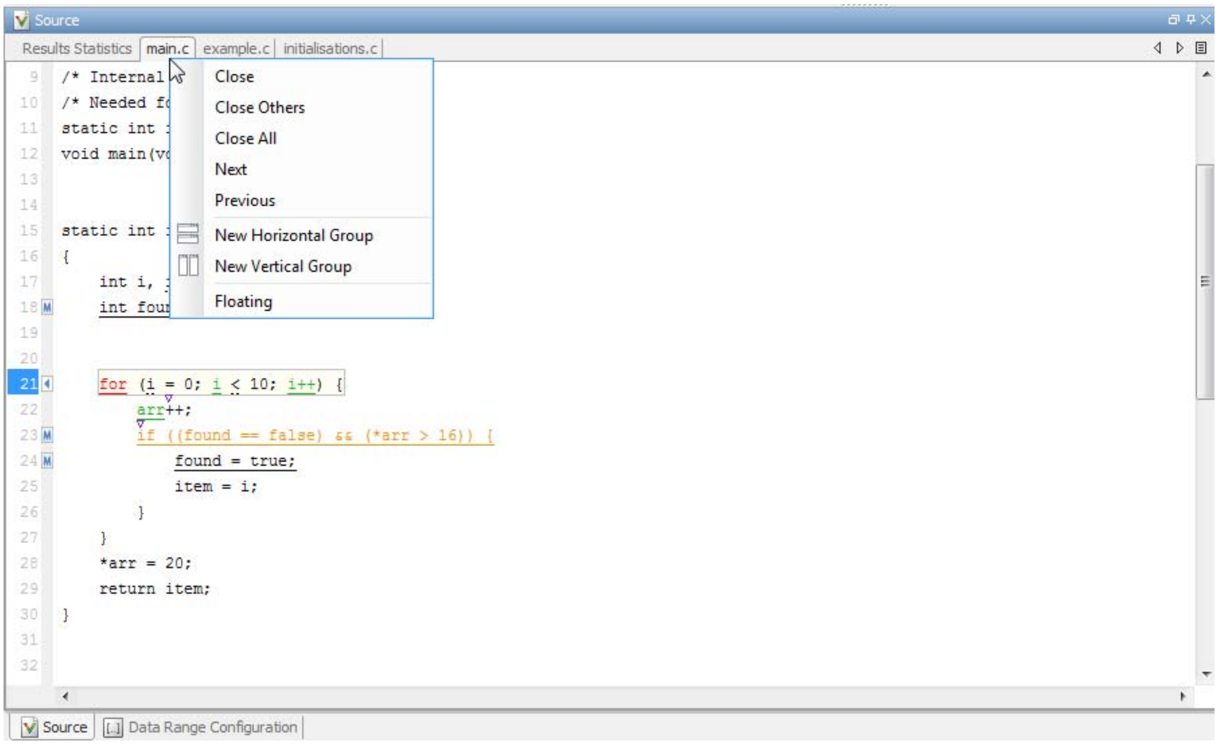
2 From the context menu, select either **Expand All Macros** or **Collapse All Macros**.

Note The **Check Details** pane also allows you to view the contents of a macro if the check you select lies within a macro.

Manage Multiple Files in Source Pane

You can view multiple source files in the **Source** pane.

Right-click on the **Source** pane toolbar.



```
/* Internal
/* Needed f
static int
void main(v


static int
{
    int i,
    int fou


    for (i = 0; i < 10; i++) {
        arr++;
        if ((found == false) && (*arr > 16)) {
            found = true;
            item = i;
        }
    }
    *arr = 20;
    return item;
}
```

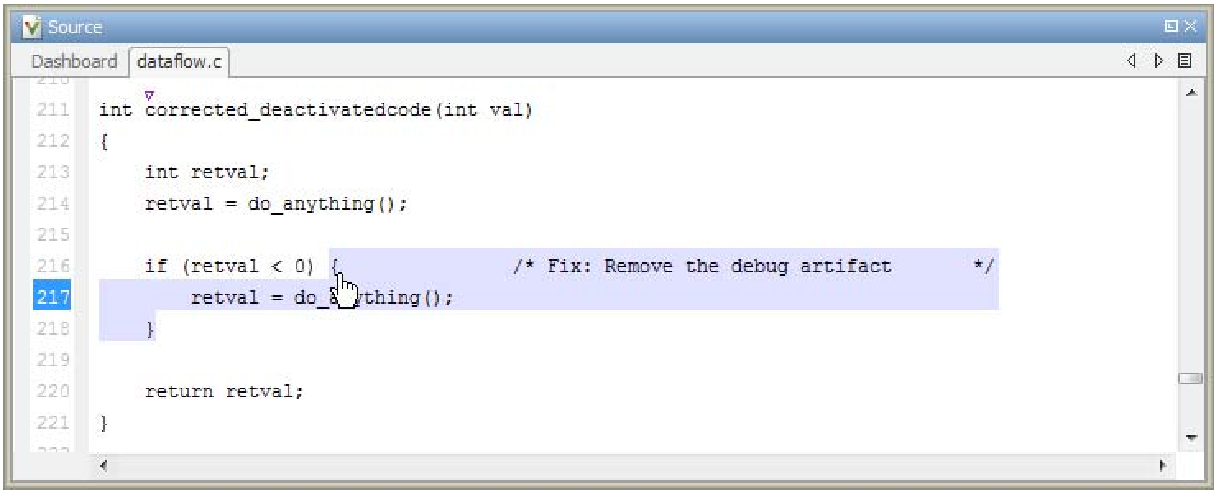
From the **Source** pane context menu, you can:

- **Close** – Close the currently selected source file.
- **Close Others** – Close all source files except the currently selected file.
- **Close All** – Close all source files.
- **Next** – Display the next view.
- **Previous** – Display the previous view.
- **New Horizontal Group** – Split the Source window horizontally to display the selected source file below another file.
- **New Vertical Group** – Split the Source window vertically to display the selected source file side-by-side with another file.

- **Floating** – Display the current source file in a new window, outside the **Source** pane.

View Code Block

On the **Source** pane, to highlight a block of code, click either its opening or closing brace.

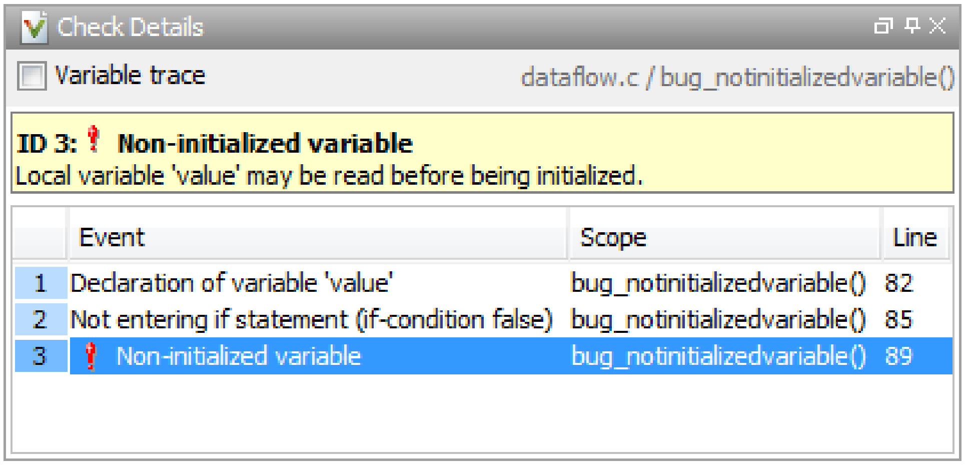


```
int corrected_deactivatedcode(int val)
{
    int retval;
    retval = do_anything();

    if (retval < 0) {           /* Fix: Remove the debug artifact      */
        retval = do_anything();
    }

    return retval;
}
```

Note This action does not highlight the code block if the brace itself is already highlighted. The opening brace can be highlighted, for example, by a **Dead code** defect on the code block.

Check Details

The **Check Details** pane contains comprehensive information about a specific defect. To see this information, on the **Results Summary** pane, select the defect.


Check Details

Variable trace   dataflow.c / bug_notinitializedvariable()

ID 3: Non-initialized variable
Local variable 'value' may be read before being initialized.

	Event	Scope	Line
1	Declaration of variable 'value'	bug_notinitializedvariable()	82
2	Not entering if statement (if-condition false)	bug_notinitializedvariable()	85
3	Non-initialized variable	bug_notinitializedvariable()	89

- The top right corner shows the file and function containing the defect, in the format `file_name/function_name`.
- The yellow box contains the name of the defect with an explanation of why the defect occurs.
- The **Event** column lists the sequence of code instructions causing the defect. The **Scope** column lists the name of the function containing the instructions. The **Line** column lists the line number of the instructions.
- The **Variable trace** check box allows you to see an additional set of instructions that are related to the defect.

For more information, see “View Code Sequence Causing Defect” on page 6-28.

Bug Finder Defect Categories

In this section...

"Numerical" on page 6-47

"Static Memory" on page 6-47

"Dynamic Memory" on page 6-48

"Programming" on page 6-48

"Data-flow" on page 6-48

"Other" on page 6-48

Numerical

These defects are errors relating to variables in your code; their values, data types, and usage. The defects include:

- Mathematical operations
- Conversion overflow
- Operational overflow

For specific defects, see "Numerical Defects".

Static Memory

These defects are errors relating to memory usage when the memory is statically allocated. The defects include:

- Accessing arrays outside their bounds
- Null pointers
- Casting of pointers

For specific defects, see "Static Memory Defects".

Dynamic Memory

These defects are errors relating to memory usage when the memory is dynamically allocated. The defects include:

- Freeing dynamically allocated memory
- Unprotected memory allocations

For specific defects, see “Dynamic Memory Defects”.

Programming

These defects are errors relating to programming syntax. These defects include:

- Assignment vs. equality operators
- Mismatches between variable qualifiers or declarations
- Badly formatted strings

For specific defects, see “Programming Defects”

Data-flow

These defects are errors relating to how information moves throughout your code. The defects include:

- Dead or unreachable code
- Unused code
- Non-initialized information

For the specific defects, see “Data-flow Defects”.

Other

These defects are those that do not fit into the other categories. They can be thing from race conditions to pass-by-value errors.

For specific defects, see “Other Defects”.

Common Weakness Enumeration from Bug Finder Defects

Polyspace Bug Finder lists the Common Weakness Enumeration IDs associated with defects on the **Results Summary** pane. To view the IDs, right-click a column header and select **CWE ID**.

The following table lists the CWE™ IDs addressed by Polyspace Bug Finder and the corresponding defects.

CWE ID	Polyspace Bug Finder Defect
119	• Array access out of bounds • Pointer access out of bounds
120	• Invalid use of standard library memory routine • Invalid use of standard library string routine
134	Format string specifiers and arguments mismatch
170	Missing null in string array
188	• Pointer access out of bounds • Unreliable cast of pointer
190	• Integer conversion overflow • Integer overflow • Shift operation overflow • Unsigned integer overflow
191	• Integer conversion overflow • Integer overflow • Unsigned integer overflow
194	Sign change integer conversion overflow

CWE ID	Polyspace Bug Finder Defect
195	• Integer conversion overflow • Sign change integer conversion overflow • Unsigned integer conversion overflow
196	• Integer conversion overflow • Sign change integer conversion overflow • Unsigned integer conversion overflow
197	Integer conversion overflow
244	Memory leak
252	Missing or invalid return statement
253	Missing or invalid return statement
366	Race conditions
367	Race conditions
369	• Float division by zero • Integer division by zero
393	Missing or invalid return statement
394	Missing or invalid return statement
398	Write without further read
401	Memory leak
404	• Invalid deletion of pointer • Invalid free of pointer • Memory leak
415	Deallocation of previously deallocated pointer
416	Use of previously freed pointer

CWE ID	Polyspace Bug Finder Defect
456	• Non-initialized pointer • Non-initialized variable
457	• Non-initialized pointer • Non-initialized variable
466	Pointer access out of bounds
467	Wrong type used in sizeof
468	• Pointer access out of bounds • Unreliable cast of pointer
476	Null pointer
481	Invalid use of = (assignment) operator
482	Invalid use of == (equality) operator
489	Code deactivated by constant false condition
561	• Dead code • Uncalled function
563	Write without further read
588	Pointer access out of bounds
590	Invalid free of pointer
617	Assertion
628	Declaration mismatch
681	Float conversion overflow
685	Declaration mismatch

CWE ID	Polyspace Bug Finder Defect
686	Declaration mismatch
761	Invalid free of pointer
762	Invalid free of pointer
789	Unprotected dynamic memory allocation
823	Pointer access out of bounds
824	Non-initialized pointer
873	• Invalid use of floating point operation • Float overflow
908	• Non-initialized pointer • Non-initialized variable • Invalid use of standard library string routine

Command-Line Analysis

- "Run Analysis from the Command Line" on page 7-2
- "Manage Remote Analyses at the Command Line" on page 7-4
- "Create Projects Automatically from Your Build System" on page 7-6
- "Requirements for Project Creation from Build Systems" on page 7-10

Run Analysis from the Command Line

Usage of Bug Finder at the Command Line

To run an analysis from a DOS or UNIX command window, use the command `polyspace-bug-finder-nodesktop` followed by other options you wish to use.

Note To run Bug Finder from the MATLAB Command Window, use the command `polyspaceBugFinder` *`[options]`*

Complete Workflow Examples

Local Analysis from Build

1 Create a list of Polyspace options using the configuration tool.

```
polyspace-configure -c -no-project -output-options-file \
        myOptions make -B myCode
```

2 Run Polyspace Bug Finder using the options read from your build.

```
polyspace-bug-finder-nodesktop -options-file myOptions \
        -results-dir myResults
```

3 Open the results in the Bug Finder Results Manager.

```
polyspace-bug-finder myResults
```

Remote Analysis

1 Create a list of Polyspace options using the configuration tool.

```
polyspace-configure -c -no-project -output-options-file \
        myOptions make -B myCode
```

2 Run Polyspace Bug Finder at the command line using the options file from `polyspace-configure`.

```
polyspace-bug-finder-nodesktop -batch -scheduler MJSname@host \
        -options-file myOptions
```

Manage Remote Analyses at the Command Line

To manage remote analyses, use this command:

```
MATLAB_Install\polyspace\bin\polyspace-jobs-manager
action [options]
          [-scheduler schedulerOption]
```

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

```
C:\Program Files\MATLAB\R2014a
```

`schedulerOption` is one of the following:

- Name of the computer that hosts the head node of your MDCS cluster (*`NodeHost`*).
- Name of the MJS on the head node host (*`MJSName@NodeHost`*).
- Name of a MATLAB cluster profile (*`ClusterProfile`*).

  For more information about clusters, see "Clusters and Cluster Profiles"

If you do not specify a job scheduler, `polyspace-job-manager` uses the scheduler specified in the **Polyspace Preferences > Server Configuration > Job scheduler host name**.

The following table lists the possible action commands to manage jobs on the scheduler.

Action	Options	Task
`listjobs`	None	Generate a list of Polyspace jobs on the scheduler. For each job, the software produces the following information: • `ID` — Verification or analysis identifier. • `AUTHOR` — Name of user that submitted job.

Action	Options	Task
		• `APPLICATION` — Name of Polyspace product, for example, Polyspace Code Prover or Polyspace Bug Finder. • `LOCAL_RESULTS_DIR` — Results folder on local computer, specified through the **Options > Preferences > Server Configuration** tab. • `WORKER` — Local computer from which job was submitted. • `STATUS` — Status of job, for example, `running` and `completed`. • `DATE` — Date on which job was submitted. • `LANG` — Language of submitted source code.

`download`	`-job` *`ID`*
	`-results-folder`
	`FolderPath`
		Download results of analysis with specified ID to folder specified by *`FolderPath`*.

`getlog`	`-job` *`ID`*	Open log for job with specified ID.
`remove`	`-job` *`ID`*	Remove job with specified ID.

Create Projects Automatically from Your Build System

In this section...

"Create Project in User Interface" on page 7-6
"Create Project from DOS and UNIX Command Line" on page 7-7
"Create Project from MATLAB Command Line" on page 7-8

If you use build automation scripts to build your source code, you can automatically setup a Polyspace project from your scripts. The automatic project setup runs your automation scripts to determine:

- Source files.
- Includes.
- **Target & Compiler** options.

Note In the Polyspace interface, it is possible that the created project will not show implicit defines or includes. The configuration tool does include them. However, they are not visible.

Create Project in User Interface

1 Select **File > New Project**.

2 On the Project – Properties dialog box, under **Project Configuration**, select **Create from build command**.

3 On the next window, enter the following information:

Field	Description
Specify command used for building your source files	Specify: • Name of your build automation script. **Example:**`make -B` • Full path to an executable such as Visual Studio. **Example:**`"C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\VCExpress.exe"`.
Specify working directory for running build command	Specify the directory from which you run your build automation script.
Add advanced configuration options	Specify additional options for advanced tasks such as incremental build. For the full list of options, see the `-options value` argument for `polyspaceConfigure`.

4 Click Run.

- If you entered your build command, Polyspace runs the command and sets up a project.
- If you entered the path to an executable, the executable runs. Build your source code and close the executable. Polyspace traces your build and sets up a project.

  For example, in Visual Studio 2010, use **Tools > Rebuild Solution** to build your source code. Then close Visual Studio.

5 If you updated your build command, you can recreate the Polyspace project from the updated command. To recreate an existing project, on the **Project Browser**, right-click the project name and select **Update Project**.

Create Project from DOS and UNIX Command Line

Use the `polyspace-configure` command to trace your build automation scripts. You can use the trace information to:

- Create a Polyspace project. You can then open the project in the user interface.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following command to create a Polyspace project `myProject.psprj` from your makefile:

  ```
  polyspace-configure  -prog myProject make -B targetName buildOptions
  ```

- Create an options file. You can then use the options file to run verification on your source code from the command-line.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following commands to create an options file `myOptions` from your makefile:

  ```
  polyspace-configure  -no-project -output-options-file
  myOptions ...
             make -B targetName buildOptions
  ```

  Use the options file to run verification:

  ```
  polyspace-bug-finder-nodesktop -options-file myOptions
  ```

For more information on advanced options for `polyspace-configure`, see the `-options value` argument for `polyspaceConfigure`.

Create Project from MATLAB Command Line

Use the `polyspaceConfigure` command to trace your build automation scripts. You can use the trace information to:

- Create a Polyspace project. You can then open the project in the user interface.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following command to create a Polyspace project `myProject.psprj` from your makefile:

  ```
  polyspaceConfigure  -prog myProject ...
               make -B targetName buildOptions
  ```

- Create an options file. You can then use the options file to run verification on your source code from the command-line.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following commands to create an options file `myOptions` from your makefile:

  ```
  polyspaceConfigure  -no-project -output-options-file
  myOptions ...
            make -B targetName buildOptions
  ```

  Use the options file to run verification:

  ```
  polyspaceBugFinder -options-file myOptions
  ```

For more information, see `polyspaceConfigure`.

Related Examples

- "Trace Visual Studio Build" on page 2-3

Concepts

- "Requirements for Project Creation from Build Systems" on page 7-10

Requirements for Project Creation from Build Systems

For `polyspaceConfigure` to correctly trace your build and gather all your source files:

- Your compiler must be called locally for a clean build.
- Your compiler configuration must be available to Polyspace. The compilers currently supported are:
  - Visual C++® compiler
  - `gcc`
  - `clang`

If your compiler does not meet these requirements, try the following:

- If your compiler performs only an incremental build, use appropriate options to build all your source files. For example, if you use `gmake`, append the `-B` option to force a clean build.
- If your compiler configuration is not available to Polyspace:
  - Write a compiler configuration file in a specific format. Use the option `-compiler-configuration` *`configurationFileName`* to provide the configuration file *`configurationFileName`*. You can find existing configuration files in *`matlabroot`*`\polyspace\configure\compiler_configuration\`.
  - Contact MathWorks Technical Support. For more information, see “Obtain system information for technical support”.
- If you use a compiler cache such as `ccache` or a distributed build system such as `distmake`, `polyspaceconfigure` cannot trace your build. You must deactivate them.

See Also

polyspaceConfigure

Related Examples

- “Create Projects Automatically from Your Build System” on page 7-6

Polyspace Bug Finder Analysis in Simulink

- "Embedded Coder Considerations" on page 8-2
- "TargetLink Considerations" on page 8-6
- "Run Analysis on Generated Code" on page 8-8
- "View Results in the Polyspace Environment" on page 8-9
- "Identify Errors in Simulink Models" on page 8-10

Embedded Coder Considerations

In this section...

"Subsystems" on page 8-2

"Default Options" on page 8-2

"Recommended Polyspace® Bug Finder™ Options for Analyzing Generated Code" on page 8-3

"Hardware Mapping Between Simulink and Polyspace" on page 8-5

Subsystems

A dialog will be presented after clicking on the Polyspace for Embedded Coder block if multiple subsystems are present in a diagram. Simply select the subsystem to analyze from the list. The subsystem list is generated from the directory structure from the code that has been generated.

Default Options

For Embedded Coder® code, the software sets certain analysis options by default.

Default options for C:

```
-sources path_to_source_code
-results-dir results
-D PST_ERRNO
-D main=main_rtwec __restrict__=
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
-OS-target no-predfined-OS
-ignore-constant-overflows true
-scalar-overflows-behavior wrap-around
-allow-negative-operand-in-shift true
-boolean-types boolean_T
```

Default options for C++:

```
-sources path_to_source_code
-results-dir results
-D PST_ERRNO
-D main=main_rtwec __restrict__=
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
-OS-target no-predfined-OS
-dialect iso
-ignore-constant-overflows true
-scalar-overflows-behavior wrap-around
-allow-negative-operand-in-shift true
```

Note *matlabroot* is the MATLAB installation folder.

Recommended Polyspace Bug Finder Options for Analyzing Generated Code

For Embedded Coder code, you can specify other analysis options for your Polyspace Project through the Polyspace **Configuration** pane. To open this pane:

1 In the Simulink® model window, select **Code > Polyspace > Options**. The **Polyspace** pane opens.

2 Click **Configure**. The Project Manager opens, displaying the Polyspace **Configuration** pane.

The following table describes options that you should specify in your Polyspace project before analyzing code generated by Embedded Coder software.

Option	Recommended Value	Comments
Target & Compiler		
`-D`	See Comments	Defines macro compiler flags used during compilation. Some defines are applied by default, depending on your `-OS-target`. Use one `-D` for each line of the Embedded Coder generated `defines.txt` file. Polyspace does not do this by default.
`-OS-target`	`Visual`	Specifies the operating system target for Polyspace stubs. This information allows the analysis to use system definitions during preprocessing to analyze the included files.
`-dos`	Selected	You must select this option if the contents of the include or source directory comes from a DOS or Windows file system. The option allows the analysis to deal with upper/lower case sensitivity and control characters issues. Concerned files are: • **Header files** – All include folders specified (`-I` option) • **Source files** – All source files selected for the analysis (`-sources` option)

Hardware Mapping Between Simulink and Polyspace

The software automatically imports target word lengths and byte ordering (endianess) from Simulink model hardware configuration settings. The software maps **Device vendor** and **Device type** settings on the Simulink **Configuration Parameters > Hardware Implementation** pane to **Target processor type** settings on the Polyspace **Configuration** pane.

The software creates a generic target for the analysis.

TargetLink Considerations

In this section...

"TargetLink Support" on page 8-6

"Subsystems" on page 8-6

"Default Options" on page 8-6

"Lookup Tables" on page 8-7

"Code Generation Options" on page 8-7

TargetLink Support

For Windows, Polyspace Bug Finder is tested with releases 3.1, 3.2, and 3.3 of the dSPACE® Data Dictionary version and TargetLink® Code Generator.

As Polyspace Bug Finder extracts information from the dSPACE Data Dictionary, you must regenerate the code before performing an analysis.

Subsystems

A dialog will be presented after clicking on the Polyspace for TargetLink block if multiple subsystems are present in a diagram. Simply select the subsystem to analyze from the list.

Default Options

The following default options are set by the tool:

```
-sources path_to_source_code
-results-dir results
-I path to source code
-D PST_ERRNO
-I dspaceroot\matlab\TL\SimFiles\Generic
-I dspaceroot\matlab\TL\srcfiles\Generic
-I dspaceroot\matlab\TL\srcfiles\i86\LCC
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
```

```
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
-OS-target no-predfined-OS
-ignore-constant-overflows true
-scalar-overflows-behavior wrap-around
-boolean-types Bool
```

Note *`dspaceroot`* and *`matlabroot`* are the dSPACE and MATLAB tool installation directories respectively.

Lookup Tables

The tool by default provides stubs for the lookup table functions. This behavior can be disabled from the Polyspace menu. The dSPACE data dictionary is used to define the range of their return values. Note that a lookup table that uses extrapolation will return full range for the type of variable that it returns.

Code Generation Options

From the TargetLink Main Dialog, it is recommended to set the option `Clean code` and deselect the option `Enable sections/pragmas/inline/ISR/user attributes`.

When installing Polyspace, the `tlcgOptions` variable has been updated with `'PolyspaceSupport', 'on'` (see variable in `'C:\dSPACE\Matlab\Tl\config\codegen\tl_pre_codegen_hook.m'` file).

Run Analysis on Generated Code

You can generate Embedded Coder code from the configured model `psdemo_model_link_sl`. You can then run a Polyspace analysis on the generated code.

To open `psdemo_model_link_sl` in the Simulink model window:

1. In the MATLAB Command Window, enter `psdemo_model_link_sl`.

   This command opens the `psdemo_model_link_sl` model that is compatible with your version of MATLAB (either `psdemo_model_link_sl`, `psdemo_model_link_sl_v1`, or `psdemo_model_link_sl_v2`).

To generate code and start the Polyspace analysis:

1. Double-click the Re-install the demo block to generate the handwritten code related to the S-function.
2. If you want to apply data ranges to the input parameters, double-click the green block Use input constraints. To remove the data range constraints, double-click the orange block Worst case inputs.
3. Right-click the subsystem `controller`.
4. From the context-menu, select **C/C++ Code > Build This Subsystem**.
5. In the Build code for Subsystem dialog box, click **Build** to generate code. When the code generation is complete, the code generation report opens.
6. Right-click the subsystem `controller`. From the context menu, select **Polyspace > Verify Code Generated for > Selected Subsystem**. The analysis starts.

   You can monitor progress from the Command Window.

Once the analysis is complete, to display the results:

1. Right-click the subsystem `controller`. From the context menu, select **Polyspace > Open Results**. The results open in the Polyspace Bug Finder interface.

View Results in the Polyspace Environment

When a Polyspace run completes, you can view the results using the Results Manager perspective of the Polyspace environment.

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

   - If you set **Model reference verification depth** to `All` and selected **Model by model verification**, the **Select the Result Folder to Open in Polyspace** dialog box opens. The dialog box displays a hierarchy of referenced models from which the software generates code. To view the analysis results for a specific model, select the model from the hierarchy. Then click **OK**.
   - You can also open results for a Model block or subsystem by right-clicking the Model block or subsystem, and from the context menu, select **Polyspace > Open Results**.

   After a few seconds, the Results Manager perspective of the Polyspace environment opens.

2. On the **Results Summary** tab, click a check to view additional information.

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

For more information on reviewing defects, see “View Results”.

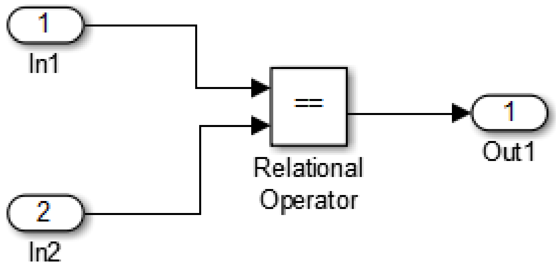
For information on specific checks, see “Polyspace Bug Finder Defects”.

Note If you selected **Add to results repository** the results are stored on the Polyspace Metrics server. For more information, see “Download Results From Polyspace Metrics” on page 6-5.

Identify Errors in Simulink Models

With Polyspace Bug Finder, you can trace your analysis results directly to your Simulink model.

Consider the following model.



where the **Check Details** pane shows information about an Invalid use of floating point operation defect, and the **Source** pane shows the source code containing error.

This defect highlights a problem comparing the signals from the inports `In1` and `In2`. To fix this issue, you must return to the model.

To trace this run-time check to the model:

1. Click the blue underlined link (`<Root>/Relational Operator`) immediately before the check in the **Source** pane. The Simulink model opens, highlighting the block with the error.
2. Examine the model to find the cause of the check.

   In this example, the highlighted block determines whether two signals are equal. In this case the signals are floating points, so the operation is imprecise. This could be a flaw in specifications; if the model is supposed to work for specific input types, you can provide these details using block parameters.

   Specifying these details should fix the defect.

If your operating system is Windows Vista™ or Windows 7, you may encounter problems with the link-back functionality if one of the following conditions apply:

- User Account Control (UAC) is enabled.
- You do not have administrator privileges.

If you have a MATLAB session running and your model is open, a possible workaround is:

1. Open a DOS window in administrator mode.
2. Go to your MATLAB installation folder.
3. From the `bin` folder, enter `matlab -regserver`.
4. Click the link again.

If your model extensively uses block coloring, the coloring from this feature may interfere with the colors already in your model. To change the color of blocks when they are linked to Polyspace results use this command:

```
HILITEDATA = struct('HiliteType', 'find', 'ForegroundColor', 'black', ...
   'BackgroundColor', color);
set_param(0, 'HiliteAncestorsData', HILITEDATA);
```

Where *`color`* is one of the following:

- `'cyan'`
- `'magenta'`
- `'orange'`
- `'lightBlue'`
- `'red'`
- `'green'`
- `'blue'`

- `'darkGreen'`

Configure Model for Code Analysis

- "Model Configuration for Code Generation and Analysis" on page 9-2
- "Configure Simulink Model" on page 9-3
- "Recommended Model Settings for Code Analysis" on page 9-5
- "Check Simulink Model Settings" on page 9-7
- "Check Simulink Model Settings Before Code Generation" on page 9-8
- "Check Simulink Model Settings Before Analysis" on page 9-10
- "Annotate Blocks for Known Errors or Coding-Rule Violations" on page 9-12

Model Configuration for Code Generation and Analysis

To facilitate Polyspace code analysis and the review of results:

- There are certain settings that you should apply to your model before generating code. See “Recommended Model Settings for Code Analysis” on page 9-5.
- The Polyspace plug-in for Simulink software allows you to check your model configuration before starting the Polyspace software. See “Check Simulink Model Settings” on page 9-7
- You can highlight blocks that you know contain checks or coding rule violations. See “Annotate Blocks for Known Errors or Coding-Rule Violations” on page 9-12.

Configure Simulink Model

To configure a Simulink model for code generation and analysis:

1. Open Model Explorer.
2. From the Model Hierarchy tree, expand the model node.
3. Select **Configuration > Code Generation**, which displays Code Generation configuration parameters.
4. Select the **General** tab, and then set the **System target file** to `ert.tlc (Embedded Coder)`.
5. In the **Report** tab, select:
   - **Create code-generation report**
   - **Code-to-model** navigation.
6. In the **Templates** tab, clear **Generate an example main program**.
7. In the **Interface** tab, select **Suppress error status in real-time model data structure**.
8. Click **Apply**.
9. Select **Configuration > Solver**, which displays Solver configuration parameters.
10. In the **Solver options** section, set:
    - **Type** to `Fixed-step`.
    - **Solver** to `discrete (no continuous states)`.
11. Click **Apply**.
12. 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.

13 Save your model.

Recommended Model Settings for Code Analysis

For Polyspace analyses, you should configure your model with the following settings before generating code.

Parameter	Recommended value	How you specify value in Configuration Parameters dialog box	If you do not use recommended value...
`InitFltsAndDblsToZero`	`'on'`	Select check box **Optimization > Use memset to initialize floats and doubles to 0.0**	Warning
`InlineParams`	`'on'`	Select check box **Optimization > Signals and Parameters > Inline parameters**	Warning
`MatFileLogging`	`'off'`	Clear check box **Code Generation > Interface > MAT-file logging**	Warning
`Solver`	`'FixedStepDiscrete'`	Select `discrete (no continuous states)` from **Solver > Solver** drop-down list	Warning
`SystemTargetFile`	`'ert.tlc'`	Specify `ert.tlc` (for Embedded Coder) in **Code Generation > System target file**	Error

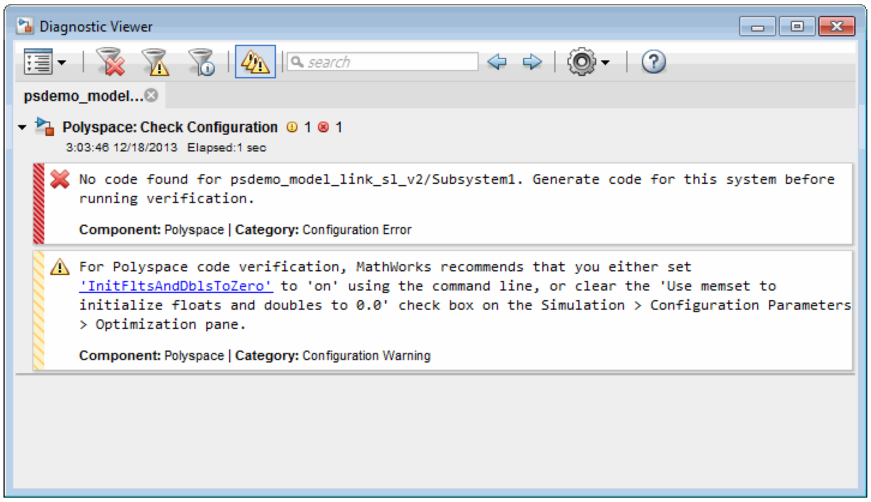
Parameter	Recommended value	How you specify value in Configuration Parameters dialog box	If you do not use recommended value...
`GenerateComments`	`'on'`	Select check box **Code Generation > Comments > Include Comments**	Error
`ZeroExternalMemoryAtStartup`	`'off'` when **Configuration Parameters > Polyspace > Data Range Management > Output** is `Global assert`	Clear check box **Optimization > Remove root level I/O zero initialization**	Warning

Check Simulink Model Settings

With the Polyspace plug-in, you can check your model settings before starting an analysis.

1 From the Simulink model window, select **Code > Polyspace > Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.

2 Click **Check configuration**. If your model settings are not optimal for Polyspace, the software displays warning messages with recommendations.


Diagnostic Viewer
search
psdemo_model...
Polyspace: Check Configuration 1 1
3:03:46 12/18/2013 Elapsed:1 sec

```
No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before
running verification.
```

Component: Polyspace | **Category:** Configuration Error

```
For Polyspace code verification, MathWorks recommends that you either set
'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to
initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters
> Optimization pane.
```

Component: Polyspace | **Category:** Configuration Warning

You can also set the configuration check to run before you run an analysis.

If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, the software produces warnings when you run the analysis.

Related Examples

- "Check Simulink Model Settings Before Analysis" on page 9-10

Concepts

- "Recommended Model Settings for Code Analysis" on page 9-5

Check Simulink Model Settings Before Code Generation

Before generating code, you can check your model settings against the "Recommended Model Settings for Code Analysis" on page 9-5.

1 From the Simulink model window, select **Code > C/C++ Code > Code Generation Options**. The Configuration Parameters dialog box opens, displaying the **Code Generation** pane.

2 Select **Set Objectives**.

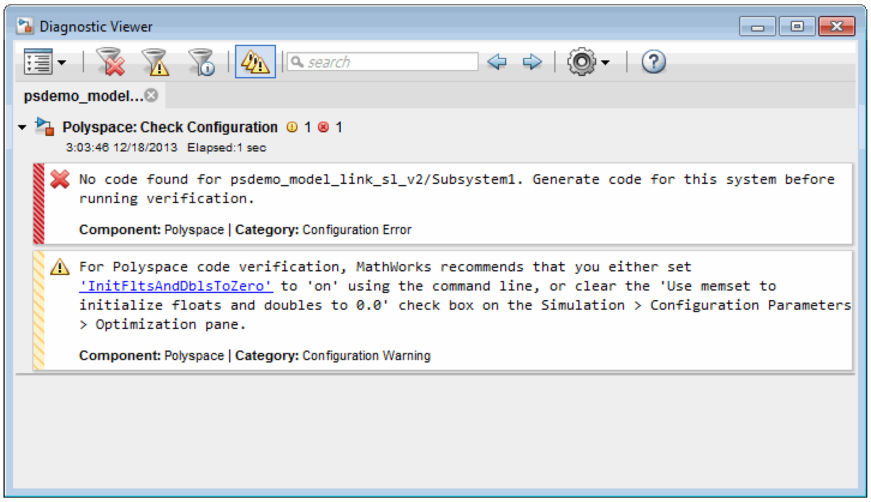
3 From the **Set Objective – Code Generation Advisor** window, add the `Polyspace` objective and any others that you want to check.

4 From the **Check model before generating code** drop-down list, select either:

   - `On (stop for warnings)`
   - `On (proceed with warnings)`

5 Select **Build** or **Generate Code**.

   The software runs a configuration check. If your configuration check finds errors or warnings, the **Diagnostics Viewer** displays the issues and recommendations.



Polyspace: Check Configuration 1 1
3:03:46 12/18/2013 Elapsed:1 sec

```
No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before
running verification.
```

Component: Polyspace | **Category:** Configuration Error

```
For Polyspace code verification, MathWorks recommends that you either set
'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to
initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters
> Optimization pane.
```

Component: Polyspace | **Category:** Configuration Warning

If you select:

- `On (stop for warnings)`, the process stops for either errors or warnings without generating code.
- `On (proceed with warnings)` — the process stops for errors, but continues generating code if the configuration only has warnings.

Related Examples

- “Check Simulink Model Settings Before Analysis” on page 9-10
- “Check Simulink Model Settings” on page 9-7

Concepts

- “Recommended Model Settings for Code Analysis” on page 9-5

Check Simulink Model Settings Before Analysis

With the Polyspace plug-in, you can check your model settings before starting an analysis:

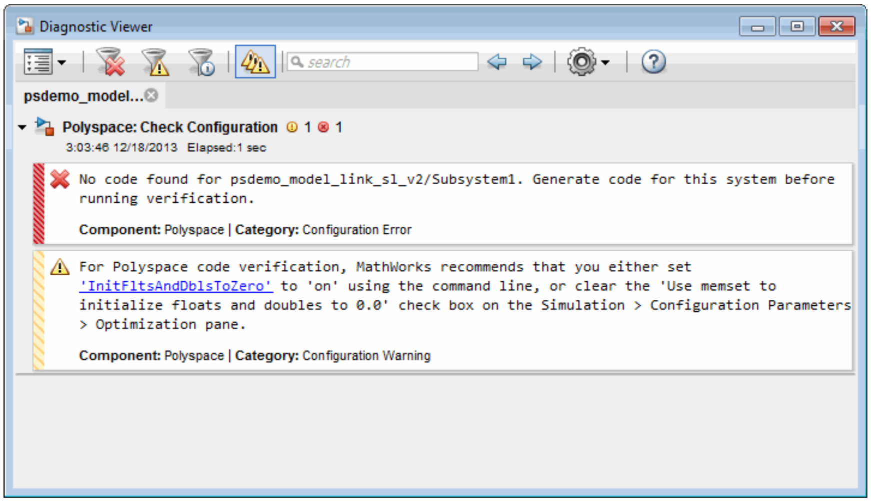
1 From the Simulink model window, select **Code > Polyspace > Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.

2 From the **Check configuration before verification** menu, select either:

   - `On (stop for warnings)` — will
   - `On (proceed with warnings)`

3 Select **Run verification**.

   The software runs a configuration check. If your configuration check finds errors or warnings, the **Diagnostics Viewer** displays the issues and recommendations.


Diagnostic Viewer
search
psdemo_model...
Polyspace: Check Configuration 1 1
3:03:46 12/18/2013 Elapsed:1 sec

```
No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before
running verification.
```

Component: Polyspace | **Category:** Configuration Error

```
For Polyspace code verification, MathWorks recommends that you either set
'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to
initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters
> Optimization pane.
```

Component: Polyspace | **Category:** Configuration Warning

If you select:

- `On (stop for warnings)`, the analysis stops for either errors or warnings.

- `On (proceed with warnings)` — the analysis stops for errors, but continues the code analysis if the configuration only has warnings.

If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, the software produces warnings when you run the analysis.

Related Examples

- "Check Simulink Model Settings" on page 9-7

Concepts

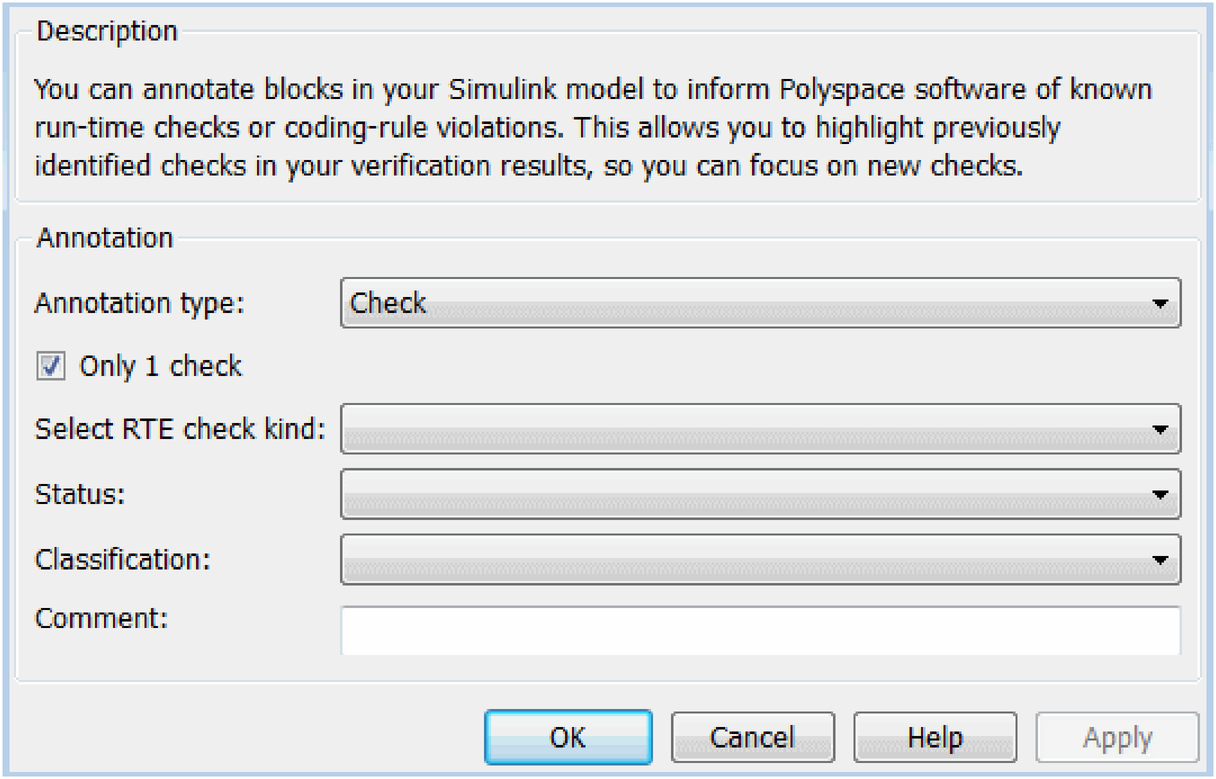
- "Recommended Model Settings for Code Analysis" on page 9-5

Annotate Blocks for Known Errors or Coding-Rule Violations

You can annotate individual blocks in your Simulink model to inform Polyspace software of known defects, run-time checks, or coding-rule violations. This allows you to highlight and categorize previously identified results, so you can focus on reviewing new results.

The Polyspace Results Manager perspective displays the information that you provide with block annotations.

1. In the Simulink model window, right-click the block you want to annotate.
2. From the context menu, select **Polyspace > Annotate Selected Block > Edit**. The Polyspace Annotation dialog box opens.



You can annotate blocks in your Simulink model to inform Polyspace software of known run-time checks or coding-rule violations. This allows you to highlight previously identified checks in your verification results, so you can focus on new checks.

Annotation type: Check

[✓] Only 1 check

Select RTE check kind:

Status:

Classification:

Comment:

3 From the **Annotation type** drop-down list, select one of the following:

- `Check` — To indicate a Code Prover run-time error
- `Defect` — To indicate a Bug Finder defect
- `MISRA-C` — To indicate a MISRA C coding rule violation
- `MISRA-C++` — To indicate a MISRA C++ coding rule violation
- `JSF` — To indicate a JSF C++ coding rule violation

4 If you want to highlight only one kind of result, select **Only 1 check** and the relevant error or coding rule from the **Select RTE check kind** (or **Select defect kind**, **Select MISRA rule**, **Select MISRA C++ rule**, or **Select JSF rule**) drop-down list.

If you want to highlight a list of checks, clear **Only 1 check**. In the **Enter a list of checks** (or **Enter a list of defects**, or **Enter a list of rule numbers**) field, specify the errors or rules that you want to highlight.

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

   - `Fix`
   - `Improve`
   - `Investigate`
   - `Justify with annotations`

     (This status also marks the result as justified.)
   - `No Action Planned`

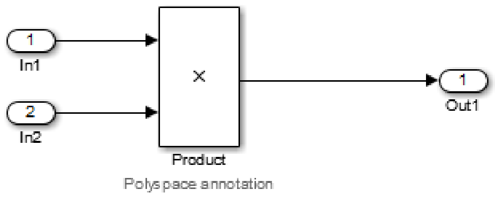
     (This status also marks the result as justified.)
   - `Other`
   - `Restart with different options`
   - `Undecided`

6. Select a **Classification** to describe the severity of the issue:

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

7. In the **Comment** field, enter additional information about the check.

8. Click **OK**. The software adds the Polyspace annotation is to the block.


1
In1
2
In2
×
Product
Polyspace annotation
1
Out1

Configure Code Analysis Options

- “Polyspace Configuration for Generated Code” on page 10-2
- “Include Handwritten Code” on page 10-3
- “Specify Remote Analysis” on page 10-5
- “Configure Analysis Depth for Referenced Models” on page 10-6
- “Specify Location of Results” on page 10-7
- “Check Coding Rules Compliance” on page 10-8
- “Configure Polyspace Options from Simulink” on page 10-10
- “Configure Polyspace Project Properties” on page 10-11
- “Create a Polyspace Configuration File Template” on page 10-12
- “Specify Header Files for Target Compiler” on page 10-15
- “Open Polyspace Results Automatically” on page 10-16
- “Remove Polyspace Options From Simulink Model” on page 10-17

Polyspace Configuration for Generated Code

You do not have to manually create a Polyspace project or specify Polyspace options before running an analysis for your generated code. By default, Polyspace automatically creates a project and extracts the required information from your model. However, you can modify or specify additional options for your analysis:

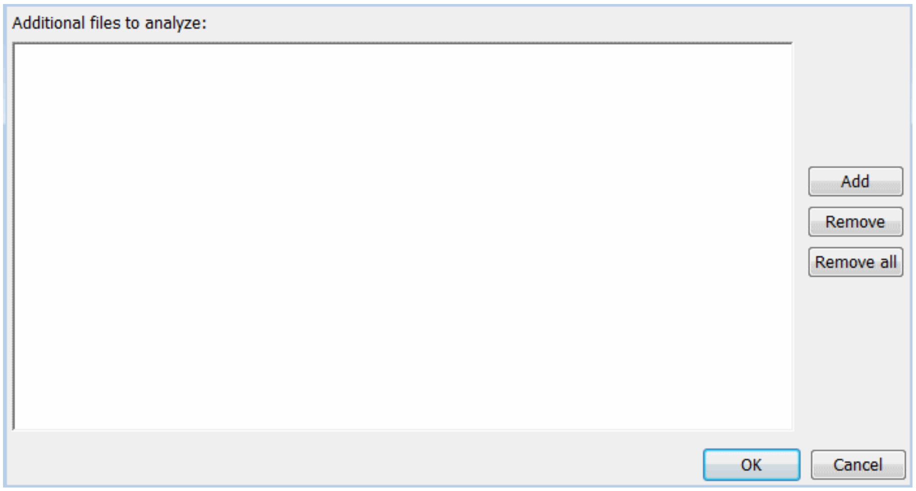
- You may incorporate separately created code within the code generated from your Simulink model. See “Include Handwritten Code” on page 10-3.
- You may customize the options for your analysis. For example, to specify the target environment or adjust precision settings. See “Configure Polyspace Options from Simulink” on page 10-10 and “Recommended Polyspace® Bug Finder™ Options for Analyzing Generated Code” on page 8-3.
- You may create specific configurations for batch runs. See “Create a Polyspace Configuration File Template” on page 10-12.
- If you want to analyze code generated for a 16-bit target processor, you must specify header files for your 16-bit compiler. See “Specify Header Files for Target Compiler” on page 10-15.

Include Handwritten Code

Files such as S-function wrappers are, by default, not part of the Polyspace analysis. However, you can add these files manually.

1 From the Simulink model window, select **Code > Polyspace > Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.

2 Select the **Enable additional file list** check box. Then click **Select files**. The Files Selector dialog box opens.



3 Click **Add**. The Select files to add dialog box opens.

4 Use the Select files to add dialog box to:

- Navigate to the relevant folder
- Add the required files.

The software displays the selected files as a list under **Additional files to analyze**.

Note To remove a file from the list, select the file and click **Remove**. To remove all files from the list, click **Remove all**.

---

5 Click **OK**.

Specify Remote Analysis

By default, the Polyspace software runs locally. To specify a remote analysis:

1. From the Simulink model window, select **Code > Polyspace > Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.
2. Select **Configure**.
3. In the Polyspace Configuration window, select the **Distributed Computing** pane.
4. Select the **Batch** checkbox.
5. Close the configuration window and save your changes.
6. Select **Apply**.

Configure Analysis Depth for Referenced Models

From the **Polyspace** pane, you can specify the analysis of generated code with respect to model reference hierarchy levels:

- **Model reference verification depth** — From the drop-down list, select one of the following:
  - `Current model only` — Default. The Polyspace runs code from the top level only. The software creates stubs to represent code from lower hierarchy levels.
  - `1` — The software analyzes code from the top level and the next level. For subsequent hierarchy levels, the software creates stubs.
  - `2` — The software analyzes code from the top level and the next two hierarchy levels. For subsequent hierarchy levels, the software creates stubs.
  - `3` — The software analyzes code from the top level and the next three hierarchy levels. For subsequent hierarchy levels, the software creates stubs.
  - `All` — The software analyzes code from the top level and all lower hierarchy levels.
- **Model by model verification** — Select this check box if you want the software to analyze code from each model separately.

Note The same configuration settings apply to all referenced models within a top model. It does not matter whether you open the **Polyspace** pane from the top model window (**Code > Polyspace > Options**) or through the right-click context menu of a particular Model block within the top model. However, you can run analyses for code generated from specific Model blocks. See “Run Analysis for Embedded Coder” on page 11-5.

Specify Location of Results

1. From the Simulink model window, select **Code > Polyspace > Options**. The Configuration Parameters dialog box opens with the Polyspace pane displayed.

2. In the **Output folder** field, specify the full path for your results folder. By default, the software stores results in `C:\Polyspace_Results\results_`*`model_name`*.

3. If you want to avoid overwriting results from previous analyses, select the **Make output folder name unique by adding a suffix** check box. Instead of overwriting an existing folder, the software specifies a new location for the results folder by appending a unique number to the folder name.

Check Coding Rules Compliance

You can check compliance with MISRA C and MISRA AC AGC coding rules directly from your Simulink model.

In addition, you can choose to run coding rules checking either with or without full code analysis.

To configure coding rules checking:

1. From the Simulink model window, select **Code > Polyspace > Options**. The **Polyspace** pane opens.

2. In the **Settings from** drop-down menu, select the type of analysis you want to perform.

   Depending on the type of code generated, different settings are available. The following tables describe the different settings.

   C Code Settings

Setting	Description
`Project configuration`	Run Polyspace using the options specified in the **Project configuration**.
`Project configuration and MISRA AC AGC rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with the MISRA AC-AGC rule set.
`Project configuration and MISRA rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with MISRA C coding rules.

C Code Settings (Continued)

Setting	Description
`MISRA AC AGC rule checking`	Check compliance with the MISRA AC-AGC rule set. Polyspace stops after rules checking.
`MISRA rule checking`	Check compliance with MISRA C coding rules. Polyspace stops after rules checking.

C++ Code Settings

Setting	Description
`Project configuration`	Run Polyspace using the options specified in the **Project configuration**.
`Project configuration and MISRA C++ rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with the MISRA C++ coding rules.
`Project configuration and JSF C++ rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with JSF C++ coding rules.
`MISRA C++ rule checking`	Check compliance with the MISRA C++ coding rules. Polyspace stops after rules checking.
`JSF C++ rule checking`	Check compliance with JSF C++ coding rules. Polyspace stops after rules checking.

3 Click **Apply** to save your settings.

Configure Polyspace Options from Simulink

From Simulink, you can use a simplified version of the Polyspace Project Manager to customize Polyspace options. For example, you can specify the target processor type, target operating system, and compilation flags.

To open the **Configuration** pane of the Project Manager:

1. From the Simulink model window, select **Code > Polyspace > Options**. The **Polyspace** pane opens.

2. Click **Configure**. The Polyspace Configuration pane opens.

   The first time you open the configuration, the software sets the following options:

   - **Target operating system** (`-OS-target`) – Set to `no-predefined-OS`
   - **Use result folder** (`-results-dir`) – Set to `results_`*`modelname`*

   The software also configures other options automatically, but the settings depend on the code generator used.

3. Set other options required by your application.

   For descriptions of advanced options, see “Analysis Options for C” or “Analysis Options for C++”.

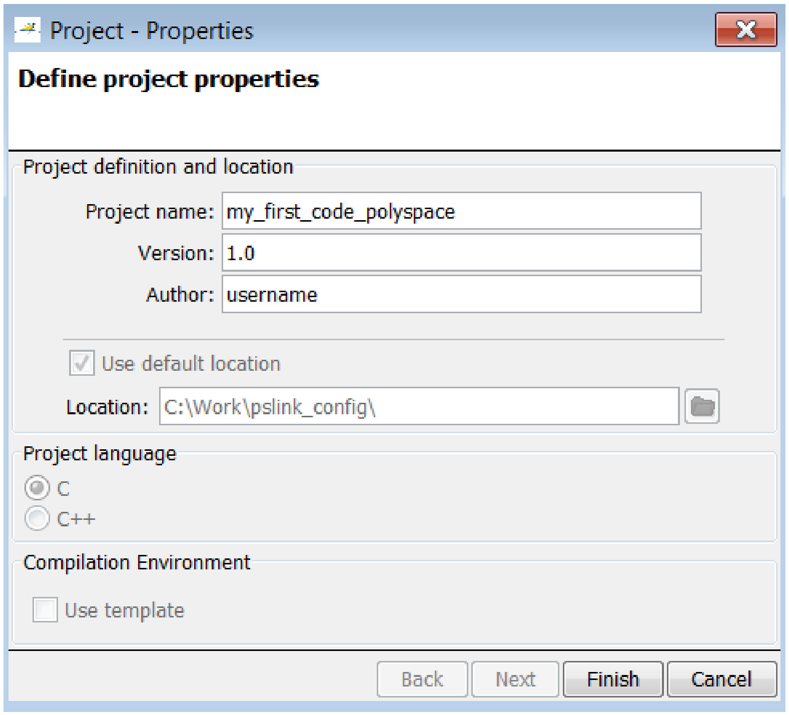
Configure Polyspace Project Properties

You can specify project properties, for example, your project name, through the Polyspace Project - Properties dialog box. To open this dialog box:

1 From the Simulink model window, select **Code > Polyspace > Options**. The **Polyspace** pane opens.

2 Click **Configure**. The Polyspace configuration window opens.

3 On the Project Manager toolbar, click the **Project properties** icon .


Project - Properties

Define project properties

Project definition and location

Project name: my_first_code_polyspace

Version: 1.0

Author: username

[✓] Use default location

Location: C:\Work\pslink_config\

Project language

(●) C

( ) C++

Compilation Environment

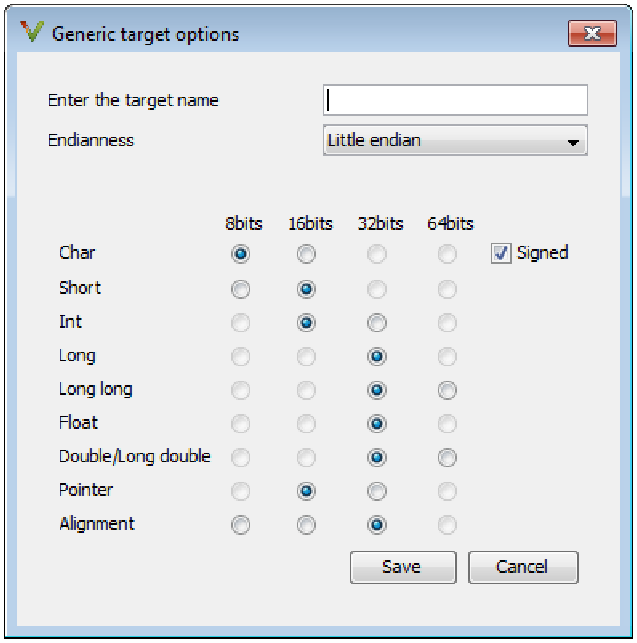
[ ] Use template

Create a Polyspace Configuration File Template

During a batch run, you may want use different configurations. At the MATLAB command-line, use `pslinkfun('settemplate',...)` to apply a configuration defined by a configuration file template.

To create a configuration file template:

1. In the Simulink model window, select **Code > Polyspace > Options**. The **Polyspace** pane opens.
2. Click **Configure**. The Project Manager opens, displaying the **Configuration** pane. Use this pane to customize the target and cross compiler.
3. From the **Configuration** tree, expand the **Target & Compiler** node.
4. In the **Target Environment** section, use the **Target processor type** option to define the size of data types.
   a. From the drop-down list, select `mcpu...(Advanced)`. The Generic target options dialog box opens.



Enter the target name

Endianness: Little endian

	8bits	16bits	32bits	64bits	
Char	●	○	○	○	✓ Signed
Short	○	●	○	○	
Int	○	●	○	○	
Long	○	○	●	○	
Long long	○	○	●	○	
Float	○	○	●	○	
Double/Long double	○	○	●	○	
Pointer	○	●	○	○	
Alignment	○	○	●	○	

Use this dialog box to create a new target and specify data types for the target. Then click **Save**.

5 From the Configuration tree, select **Target & Compiler > Macros**. Use the **Preprocessor definitions** section to define preprocessor macros for your cross-compiler.

To add a macro, in the **Macros** table, click the **+** button. In the new line, enter the required text.

To remove a macro, select the macro and click the **-** button.

Note If you use the LCC cross-compiler, then you must specify the `MATLAB_MEX_FILE` macro.

---

6 Save your changes and close the Project Manager.

7. Make a copy of the updated project configuration file, for example, `my_first_code_polyspace.psprj`.

8. Rename the copy, for example, `my_cross_compiler.psprj`. This is your new configuration file template.

To use a configuration template, run the `pslinkfun` command in the MATLAB Command Window. For example:

```
pslinkfun('settemplate','C:\Work\my_cross_compiler.psprj')
```

Specify Header Files for Target Compiler

If you want to analyze code generated for a 16-bit target processor, you must specify header files for your 16-bit compiler. The software automatically identifies the compiler from the Simulink model. If the compiler is 16-bit and you do not specify the relevant header files, the software produces an error when you try to run an analysis.

Note For a 32-bit or 64-bit target processor, the software automatically specifies the default header file.

To specify header file folders (or header files) for your compiler:

1. Open the Polyspace **Configuration** pane. From the Simulink model window, select **Code > Polyspace > Options**. The **Polyspace** pane opens.
2. Click **Configure**. The Project Manager opens, displaying the **Configuration** pane.
3. From the **Configuration** tree, expand the **Target & Compiler** node.
4. Select **Target & Compiler > Environment Settings**.
5. In the **Include folders** (or **Include**) section, specify a folder (or header file) path by doing one of the following:
   - Click the + button. Then, in the text field, enter the folder (or file) path.
   - Click the folder button and use the Open file dialog box to navigate to the required folder (or file).

     You can remove an item from the displayed list by selecting the item and then clicking -.

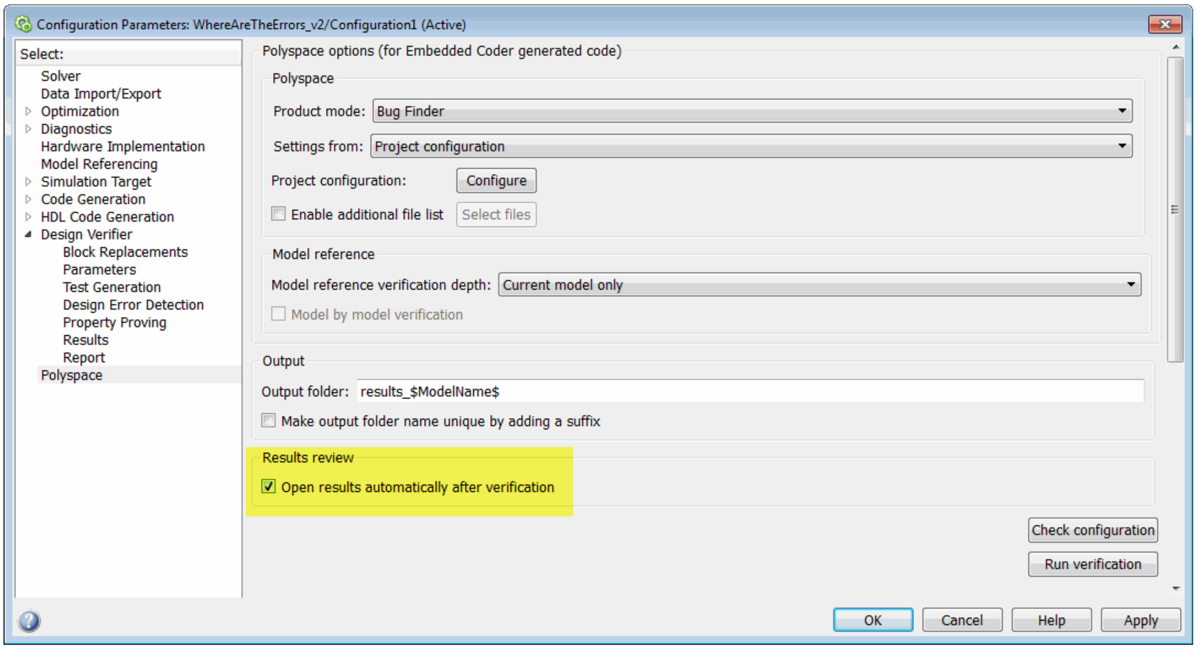
Open Polyspace Results Automatically

You can configure the software to automatically open your Polyspace results after you start the analysis. If you are doing a remote analysis, the Polyspace Metrics webpage opens. When the remote job is complete, you can download your results from Polyspace Metrics. If you are doing a local analysis, when the local job is complete, the Polyspace environment opens the results in the Results Manager perspective.

To configure the results to open automatically:

1 From the model window, select **Code > Polyspace > Options**.

   The Polyspace pane opens.



Polyspace options (for Embedded Coder generated code)

Polyspace

Product mode: Bug Finder

Settings from: Project configuration

Project configuration:

☐ Enable additional file list

Model reference

Model reference verification depth: Current model only

☐ Model by model verification

Output

Output folder: results_$ModelName$

☐ Make output folder name unique by adding a suffix

Results review

☑ Open results automatically after verification

2 In the Results review section, select **Open results automatically after verification**.

3 Click **Apply** to save your settings.

Remove Polyspace Options From Simulink Model

You can remove Polyspace configuration information from your Simulink model.

For a top model:

1. Select **Code > Polyspace > Remove Options from Current Configuration**.
2. Save the model.

For a Model block or subsystem:

1. Right-click the Model block or subsystem.
2. From the context menu, select **Polyspace > Remove Options from Current Configuration**.
3. Save the model.

Run Polyspace on Generated Code

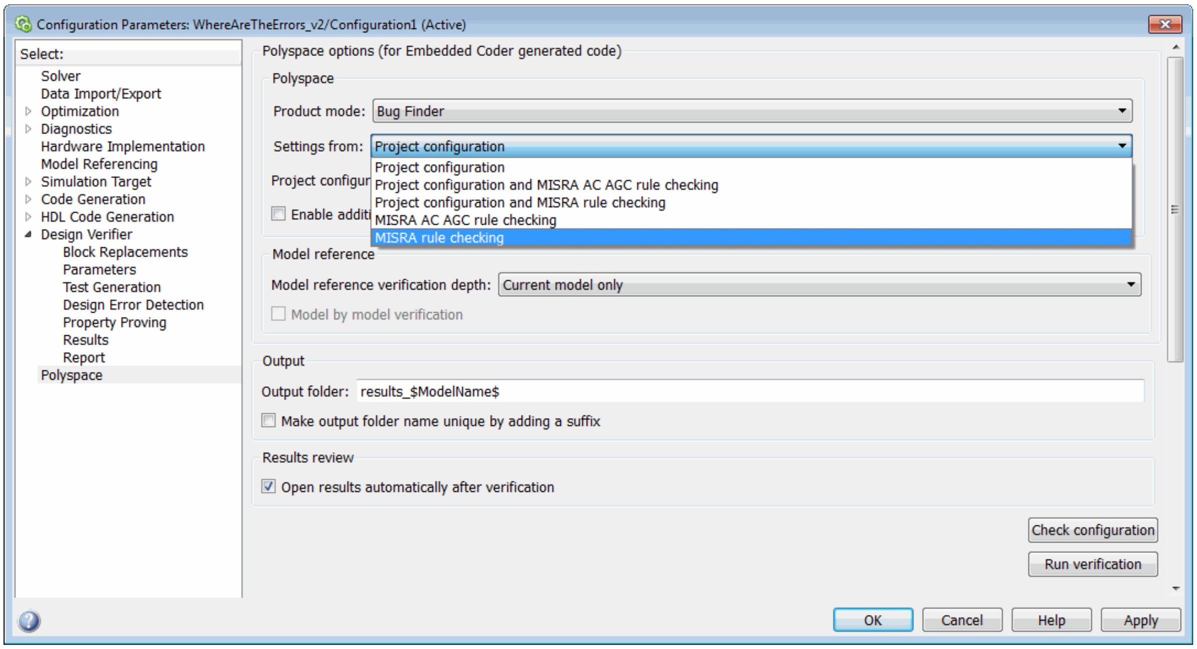
- "Specify Type of Analysis to Perform" on page 11-2
- "Run Analysis for Embedded Coder" on page 11-5
- "Run Analysis for TargetLink" on page 11-7
- "Monitor Progress" on page 11-8

Specify Type of Analysis to Perform

Before running Polyspace, you can specify what type of analysis you want to run. You can choose to run code analysis, coding rules checking, or both.

To specify the type of analysis to run:

1 From the Simulink model window, select **Code > Polyspace > Options**. The **Configuration Parameter** window opens to the **Polyspace** options pane.


Configuration Parameters: WhereAreTheErrors_v2/Configuration1 (Active)

Polyspace options (for Embedded Coder generated code)

Polyspace

Product mode: Bug Finder

Settings from: Project configuration

- Project configuration
- Project configuration and MISRA AC AGC rule checking
- Project configuration and MISRA rule checking
- MISRA AC AGC rule checking
- MISRA rule checking

Model reference

Model reference verification depth: Current model only

☐ Model by model verification

Output

Output folder: results_$ModelName$

☐ Make output folder name unique by adding a suffix

Results review

☑ Open results automatically after verification

2 In the **Settings from** drop-down menu, select the type of analysis you want to perform.

Depending on the type of code generated, different settings are available. The following tables describe the different settings.

C Code Settings

Setting	Description
`Project configuration`	Run Polyspace using the options specified in the **Project configuration**.
`Project configuration and MISRA AC AGC rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with the MISRA AC-AGC rule set.
`Project configuration and MISRA rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with MISRA C coding rules.
`MISRA AC AGC rule checking`	Check compliance with the MISRA AC-AGC rule set. Polyspace stops after rules checking.
`MISRA rule checking`	Check compliance with MISRA C coding rules. Polyspace stops after rules checking.

C++ Code Settings

Setting	Description
`Project configuration`	Run Polyspace using the options specified in the **Project configuration**.
`Project configuration and MISRA C++ rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with the MISRA C++ coding rules.

C++ Code Settings (Continued)

Setting	Description
`Project configuration and JSF C++ rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with JSF C++ coding rules.
`MISRA C++ rule checking`	Check compliance with the MISRA C++ coding rules. Polyspace stops after rules checking.
`JSF C++ rule checking`	Check compliance with JSF C++ coding rules. Polyspace stops after rules checking.

3 Click **Apply** to save your settings.

Run Analysis for Embedded Coder

To start Polyspace with:

- Code generated from the top model, from the Simulink model window, select **Code > Polyspace > Verify Code Generated for > Model**.
- All code generated as model referenced code, from the model window, select **Code > Polyspace > Verify Code Generated for > Referenced Model**.
- Model reference code associated with a specific block or subsystem, right-click the Model block or subsystem. From the context menu, select **Verify Code Generated for > Selected Subsystem**.

Note You can also start the Polyspace software from the **Polyspace** configuration parameter pane by clicking **Run verification**.

When the Polyspace software starts, messages appear in the MATLAB Command window:

```
### Starting Polyspace verification for Embedded Coder
### Creating results folder C:\PolySpace_Results\results_my_first_code
                                           for system my_first_code
### Checking Polyspace Model-Link Configuration:
### Parameters used for code verification:
 System                : my_first_code
 Results Folder        : C:\PolySpace_Results\results_my_first_code
 Additional Files      : 0
 Remote                : 0
 Model Reference Depth : Current model only
 Model by Model        : 0
 DRS input mode        : DesignMinMax
 DRS parameter mode    : None
 DRS output mode       : None
...
```

Follow the progress of the analysis in the MATLAB Command window. If you are running a remote, batch, analysis you can follow the later stages through the Polyspace Queue Manager.

The software writes status messages to a log file in the results folder, for example `Polyspace_R2013b_my_first_code_05_16_2013-18h40.log`

Run Analysis for TargetLink

To start the Polyspace software:

1. In your model, select the Target Link subsystem.
2. In the Simulink model window select **Code > Polyspace > Verify Code Generated for > Selected Target Link Subsystem**.

   Messages appear in the MATLAB Command window:

   ```
   ### Starting Polyspace verification for Embedded Coder
   ### Creating results folder results_WhereAreTheErrors_v2
                           for system WhereAreTheErrors_v2
   ### Parameters used for code verification:
    System                : WhereAreTheErrors_v2
    Results Folder        : H:\Desktop\Test_Cases\ModelLink_Testers
                                      \results_WhereAreTheErrors_v2
    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
   ```

   The exact messages depend on the code generator you use and the Polyspace product. The software writes status messages to a log file in the results folder, for example `Polyspace_R2013b_my_first_code_05_16_2013-18h40.log`

Follow the progress of the software in the MATLAB Command Window. If you are running a remote, batch analysis, you can follow the later stages through the Polyspace Queue Manager

Monitor Progress

In this section...

"Local Analyses" on page 11-8

"Remote Batch Analyses" on page 11-8

Local Analyses

For a local Polyspace runs, you can follow the progress of the software in the MATLAB Command Window. The software also saves the status messages to a log file in the results folder. For example:

```
Polyspace_R2013b_my_first_code_05_16_2013-18h40.log
```

Remote Batch Analyses

For a remote analysis, you can follow the initial stages of the analysis in the MATLAB Command window.

Once the compilation phase is complete, you can follow the progress of the software using the Polyspace Queue Manager.

From Simulink, select **Code > Polyspace > Open Spooler**

Check Coding Rules from Eclipse

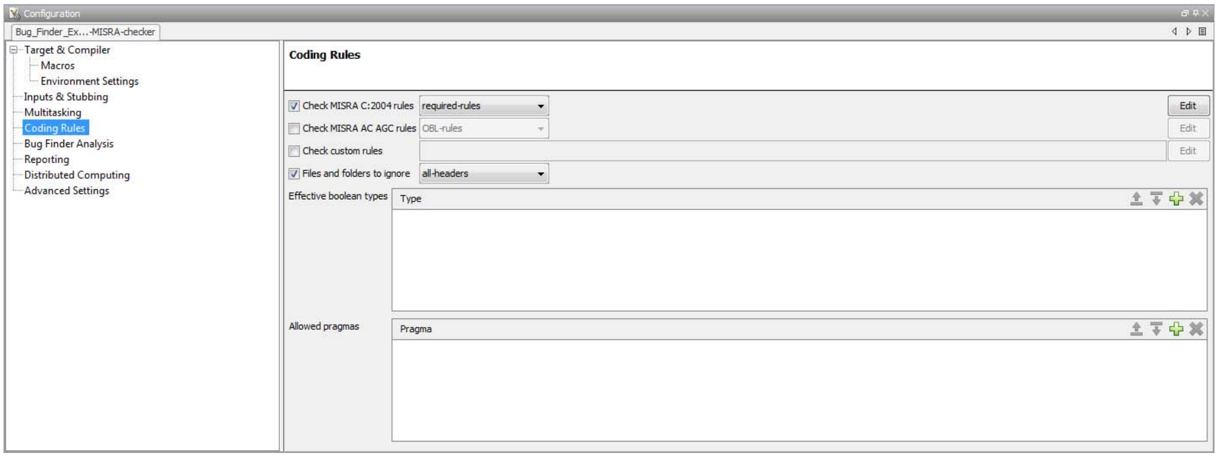
- "Activate Coding Rules Checker" on page 12-2
- "Select Specific MISRA or JSF Coding Rules" on page 12-6
- "Create Custom Coding Rules File" on page 12-8
- "Contents of Custom Coding Rules File" on page 12-10
- "Exclude Files from Rules Checking" on page 12-12
- "Allow Custom Pragma Directives" on page 12-13
- "Specify Boolean Types" on page 12-14
- "Find Coding Rule Violations" on page 12-15
- "Review Coding Rule Violations" on page 12-16
- "Apply Coding Rule Violation Filters" on page 12-18

Activate Coding Rules Checker

This example shows how to activate the coding rules checker before you start an analysis. This activation enables the Polyspace Bug Finder plug-in to search for coding rule violations. You can view the coding rule violations in your analysis results.

1 Open project configuration.

2 In the **Configuration** tree view, select **Coding Rules**.



Coding Rules

- [✓] Check MISRA C:2004 rules: required-rules
- [ ] Check MISRA AC AGC rules: OBL-rules
- [ ] Check custom rules
- [✓] Files and folders to ignore: all-headers

Effective boolean types

Allowed pragmas

3 Select the check box for the type of coding rules that you want to check.

For C code, you can check compliance with:

- MISRA C:2004
- MISRA AC AGC
- Custom coding rules

For C++ code, you can check compliance with:

- MISRA C++
- JSF C++
- Custom coding rules

4 For each rule type that you select, from the drop-down list, select the subset of rules to check.

MISRA C:2004

Option	Description
`required-rules`	All required MISRA C coding rules.
`all-rules`	All required and advisory MISRA C coding rules.
`SQO-subset1`	A small subset of MISRA C rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA C coding rules that you specify.

MISRA AC AGC

Option	Description
`OBL-rules`	All required MISRA AC AGC coding rules.
`OBL-REC-rules`	All required and recommended MISRA AC AGC coding rules.
`all-rules`	All required, recommended, and readability coding rules.
`SQO-subset1`	A small subset of MISRA AC AGC rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.

Option	Description
`SQO-subset2`	A second subset of MISRA AC AGC rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA AC AGC coding rules that you specify.

MISRA C++

Option	Description
`required-rules`	All required MISRA C++ coding rules.
`all-rules`	All required and advisory MISRA C++ coding rules.
`SQO-subset1`	A small subset of MISRA C++ rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of rules with indirect impact on the selectivity in addition to `SQO-subset1`. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A specified set of MISRA C++ coding rules.

JSF C++

Option	Description
`shall-rules`	**Shall** rules are mandatory requirements. These rules require verification.
`shall-will-rules`	All **Shall** and **Will** rules. **Will** rules are intended to be mandatory requirements. However, these rules do not require verification.

Option	Description
`all-rules`	All **Shall**, **Will**, and **Should** rules. **Should** rules are advisory rules.
`custom`	A set of JSF C++ coding rules that you specify.

5 If you select **Check custom rules**, specify the path to your custom rules file or click **Edit** to create one.

When rules checking is complete, the software displays the coding rule violations in purple on the **Results Summary** pane.

Related Examples

- "Select Specific MISRA or JSF Coding Rules" on page 12-6
- "Create Custom Coding Rules File" on page 12-8

Select Specific MISRA or JSF Coding Rules

This example shows how to specify a subset of MISRA or JSF rules for the coding rules checker. If you select `custom` from the MISRA or JSF drop-down list, you must provide a file that specifies the rules to check.

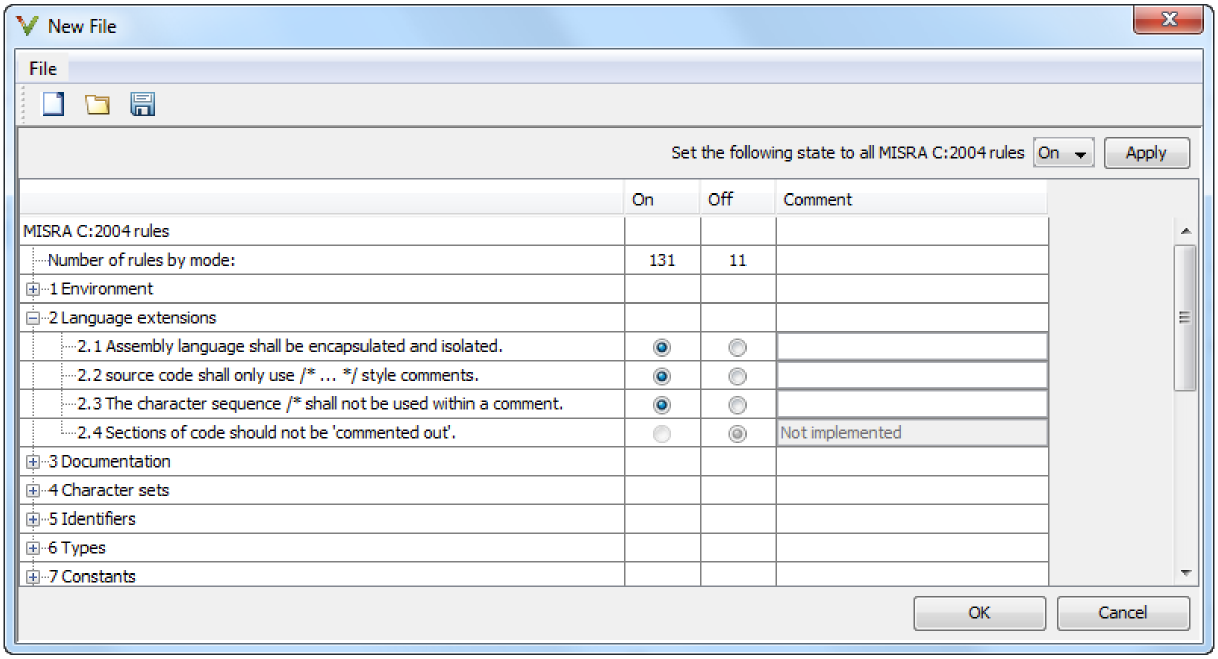
1 Open project configuration.

2 In the **Configuration** tree view, select **Coding Rules**.

3 Select the check box for the type of coding rules you wish to check

4 From the corresponding drop-down list, select `custom`. The software displays a new field for your custom file.

5 To the right of this field, click **Edit**. A New File window opens, displaying a table of rules.



	On	Off	Comment
MISRA C:2004 rules			
Number of rules by mode:	131	11	
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			

Select **On** for the rules you want to check.

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

   The **Save as** dialog box opens.

7 In the **File** field, enter a name for the rules file.

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

   The full path to the rules file appears. To reuse this rules file for other projects, type this path name or use the 📁 icon in the New File window.

Related Examples

- "Activate Coding Rules Checker" on page 12-2
- "Create Custom Coding Rules File" on page 12-8

Create Custom Coding Rules File

This example shows how to create a custom coding rules file. You can use this file to check names or text patterns in your source code against custom rules that you specify. For each rule, you specify a pattern in the form of a regular expression. The software compares the pattern against identifiers in the source code and determines whether the custom rule is violated.

Save Example Code

Save the following code in a file `printInitialValue.c`:

```
#include <stdio.h>

typedef struct {
int a;
int b;
} collection;

void main()
{
 collection myCollection={0,0};
 printf("Initial values in the collection are %d
                     and %d.",myCollection.a,myCollection.b);
}
```

Create Coding Rules File

1 Create a Polyspace project. Add `printInitialValue.c` to the project.

2 On the **Configuration** pane, select **Coding Rules**. Select the **Check custom rules** box.

3 Click Edit.

   The New File window opens, displaying a table of rule groups.

4 From the drop-down list **Set the following state to all Custom C rules**, select `Off`. Click **Apply**.

5 Expand the **Structs** node. For the option **4.3 All struct fields must follow the specified pattern**:

Column Title	Action
On	Select ●.
Convention	Enter `All struct fields must begin with s_ and have capital letters.`
Pattern	Enter `s_[A-Z0-9_]`
Comment	Leave blank. This column is for comments that appear in the coding rules file alone.

Review Coding Rule Violations

1 Save the file and run the verification. On the **Results Summary** pane, you see two violations of rule 4.3. Select the first violation.

   a On the **Source** pane, the line `int a;` is marked.

   b On the **Check Details** pane, you see the error message you had entered, `All struct fields must begin with s_ and have capital letters.`

2 Right-click on the **Source** pane and select **Open Source File**. The file `printInitialValue.c` opens in a text editor.

3 In the file, replace all instances of `a` with `s_A` and `b` with `s_B`. Save the file and rerun the verification.

   The custom rule violations no longer appear on the **Results Summary** pane.

Related Examples

- "Activate Coding Rules Checker" on page 12-2
- "Select Specific MISRA or JSF Coding Rules" on page 12-6

Concepts

- "Contents of Custom Coding Rules File" on page 12-10

Contents of Custom Coding Rules File

In a custom coding rules file, each rule appears in the following format:

```
N.n off|warning
convention=violation_message
pattern=regular_expression
```

- *N.n* — Custom rule number, for example, 1.2.
- `off` — Rule is not considered.
- `warning` — The software checks for violation of the rule. After verification, it displays the coding rule violation on the **Results Summary** pane.
- *violation_message* — Software displays this text in an XML file within the *Results*/`Polyspace-Doc` folder.
- *regular_expression* — Software compares this text pattern against a source code identifier that is specific to the rule. See "Custom Naming Convention Rules" on page 3-3.

  The keywords `convention=` and `pattern=` are optional. If present, they apply to the rule whose number immediately precedes these keywords. If `convention=` is not given for a rule, then a standard message is used. If `pattern=` is not given for a rule, then the default regular expression is used, that is, `.*`.

  Use the symbol # to start a comment. Comments are not allowed on lines with the keywords `convention=` and `pattern=`.

  The following example contains three custom rules: 1.1, 8.1, and 9.1.

```
# Custom rules configuration file
1.1  off          # Disable custom rule number 1.1
8.1  warning        # Violation of custom rule 8.1 produces a warning
convention=Global constants must begin by G_ and must be in capital letters.
pattern=G_[A-Z0-9_]*
9.1  warning    # Non-adherence to custom rule 9.1 produces a warning
convention=Global variables should begin by g_.
pattern=g_.*
```

Related Examples

- "Create Custom Coding Rules File" on page 12-8

Exclude Files from Rules Checking

This example shows how to exclude certain files from coding rules checking.

1 Open project configuration.

2 In the **Configuration** tree view, select **Coding Rules**.

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

4 From the corresponding drop-down list, select one of the following:

- `all-headers` (default) — Rule checker excludes folders that contain only header files, that is, folders without source files.
- `all` — Rule checker excludes all include folders. For example, if you are checking a large code base with standard or Visual headers, excluding include folders can significantly improve the speed of code analysis.
- `custom` — Rule checker excludes files or folders specified in the **File/Folder** view. To add files to the custom **File/Folder** list, select to choose the files and folders to exclude. To remove a file or folder from the list of excluded files and folders, select the row. Then click .

Related Examples

- "Activate Coding Rules Checker" on page 12-2

Allow Custom Pragma Directives

This example shows how to exclude custom pragma directives from coding rules checking. MISRA C rule 3.4 requires checking that pragma directives are documented within the documentation of the compiler. However, you can allow undocumented pragma directives to be present in your code.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules**.
3. To the right of **Allowed pragmas**, click ➕.

   In the **Pragma** view, the software displays an active text field.
4. In the text field, enter a pragma directive.
5. To remove a directive from the **Pragma** list, select the directive. Then click ✖.

Related Examples

- "Activate Coding Rules Checker" on page 12-2

Specify Boolean Types

This example shows how to specify data types you want Polyspace to consider as Boolean during MISRA C rules checking. The software applies this redefinition only to data types defined by `typedef` statements. The use of this option may affect the checking of MISRA C rules 12.6, 13.2, and 15.4.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules**.
3. To the right of **Effective boolean types**, click ➕.

   In the **Type** view, the software displays an active text field.
4. In the text field, specify the data type that you want Polyspace to treat as Boolean.
5. To remove a data type from the **Type** list, select the data type. Then click ✖.

Related Examples

- "Activate Coding Rules Checker" on page 12-2

Find Coding Rule Violations

This example shows how to check for coding rule violations alone.

1 Open project configuration.

2 In the **Configuration** tree view, select **Coding Rules**. Activate the desired coding rule checker.

3 In the **Configuration** tree view, select **Bug Finder Analysis**.

4 Clear the **Find defects** check box.

5 Click ▷ to run the coding rules checker without checking defects.

   You can view the results by selecting the `RuleSet-report.xml` file from the results folder.

Related Examples

- “Activate Coding Rules Checker” on page 12-2
- “Select Specific MISRA or JSF Coding Rules” on page 12-6
- “Review Coding Rule Violations” on page 12-16

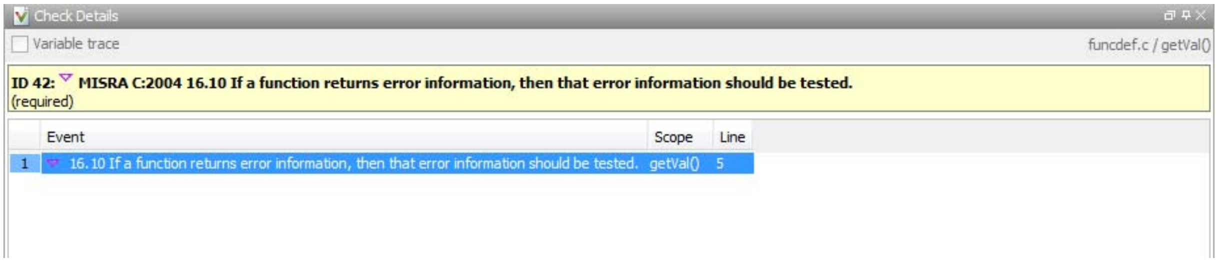
Review Coding Rule Violations

This example shows how to review coding rule violations in the Results Manager perspective once code analysis is complete. After analysis, the **Results Summary** tab displays the rule violations with a

- ▽ symbol for predefined coding rules such as MISRA C:2004.
- ▼ symbol for custom coding rules.

1 Select a coding-rule violation on the **Results Summary** pane.
   - The predefined rules such as MISRA C or C++ or JSF C++ are indicated by ▽ .
   - The custom rules are indicated by ▼ .

2 On the **Check Details** pane, view the location and description of the violated rule. In the source code, the line containing the violation appears highlighted.



ID 42: ▽ MISRA C:2004 16.10 If a function returns error information, then that error information should be tested.
(required)

	Event	Scope	Line
1	▽ 16.10 If a function returns error information, then that error information should be tested.	getVal()	5

3 Review the violation. On the **Check Review** tab, select a **Classification** to describe the severity of the issue:
   - `High`
   - `Medium`
   - `Low`
   - `Not a defect`

4 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`

You can also define your own statuses.

5 In the comment box, enter additional information about the violation.

6 To open the source file that contains the coding rule violation, on the **Source** pane, right-click the code with the purple check. From the context menu, select **Open Source File**. The file opens in your text editor.

7 Fix the coding rule violation.

8 When you have corrected the coding rule violations, run the analysis again.

Related Examples

- "Activate Coding Rules Checker" on page 12-2
- "Find Coding Rule Violations" on page 12-15
- "Apply Coding Rule Violation Filters" on page 12-18

Apply Coding Rule Violation Filters

This example shows how to use filters in the **Results Summary** pane to focus on specific kinds of coding rule violations. By default, the software displays both coding rule violations and defects.

To filter violations by rule number:

1. On the **Results Summary** pane, place your cursor on the **Check** column header. Click the filter icon that appears.
2. From the context menu, clear the **All** check box.
3. Select the violated rule numbers that you want to focus on.
4. Click **OK**.

Related Examples

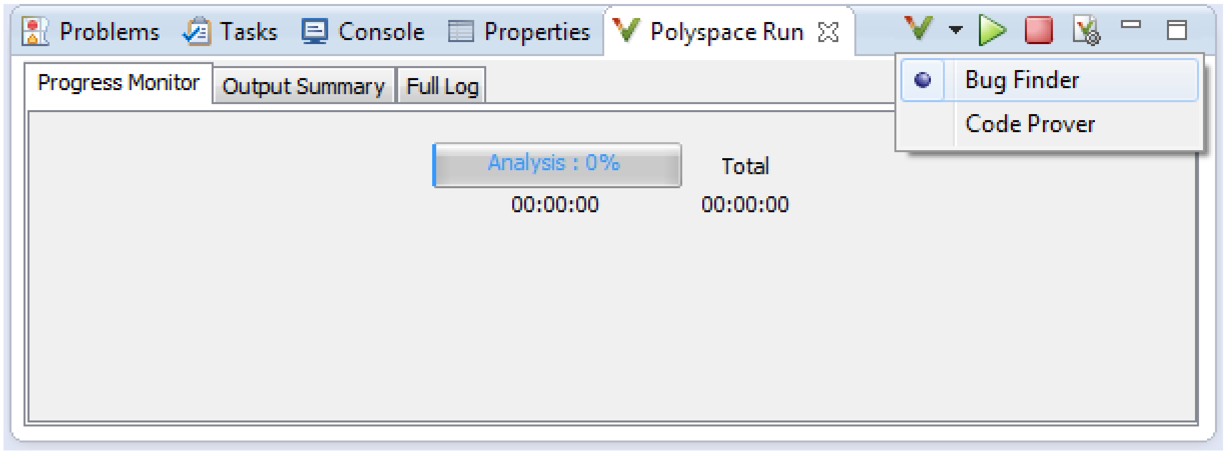
- "Activate Coding Rules Checker" on page 12-2
- "Review Coding Rule Violations" on page 12-16

Find Bugs from Eclipse

- "Run Analysis" on page 13-2
- "Customize Analysis Options" on page 13-3

Run Analysis

1 In the Polyspace Run window, select **Bug Finder** from the product configuration icon.



2 In the Project Explorer, select the files you want to analyze.

3 Do one of the following to run an analysis:

- Right-click on your selection and from the context menu select **Start Polyspace Bug Finder**
- From the toolbar, select **Polyspace > Start Polyspace**

Follow your analysis in the Progress Monitor tab of the Polyspace Log window. If your analysis fails, error and warning messages appear in the Output Summary tab.

Customize Analysis Options

The software uses a set of default analysis options preconfigured for your coding language and operating system. For each project, you can customize your configuration.

1. From the toolbar, select **Polyspace > Configure Project**.

   The Polyspace Bug Finder Configuration window appears.

2. Select the different panes to change your analysis configuration.

   For example, on the **Coding Rules** pane, select one of the coding rule sets to add coding rules checking to your analysis.

For information about the different analysis options, see “Analysis Options for C” or “Analysis Options for C++”.

View Results in Eclipse

- "Filter and Group Results" on page 14-2
- "View Results" on page 14-8
- "Review and Fix Results" on page 14-9
- "Understanding the Results Views" on page 14-12

Filter and Group Results

This example shows how to filter and group defects on the **Results Summary** tab. To organize your review of results, use filters and groups when you want to:

- Review certain categories of defects in preference to others. For instance, you first want to address the defects resulting from `Missing or invalid return statement`.
- Not address the full set of coding rule violations detected by the coding rules checker.
- Review only those defects that you have already assigned a certain status. For instance, you want to review only those defects to which you have assigned the status, `Investigate`.
- Review defects from a particular file or function. Because of continuity of code, reviewing these defects together can help you organize your review process.

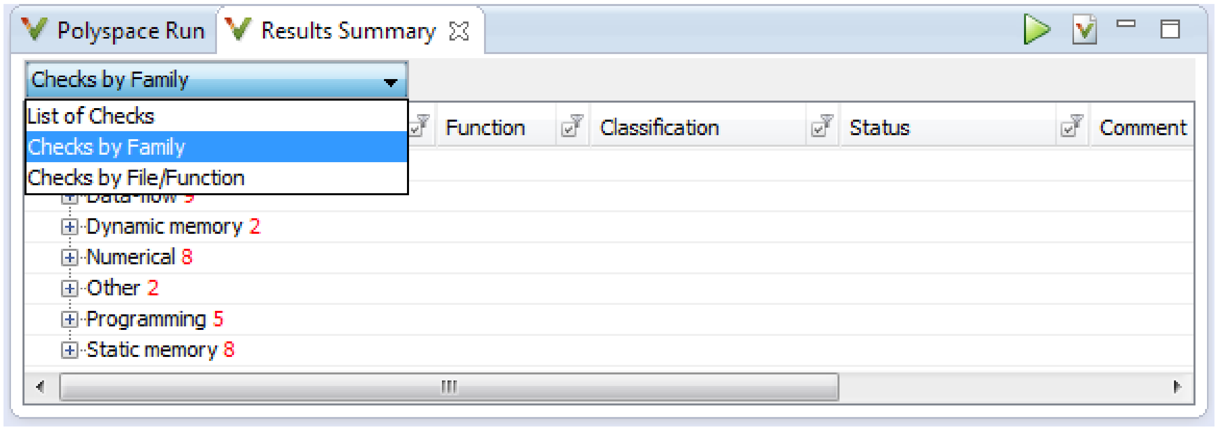
  If you have written the code for a particular source file, you can review the defects only in that file.

Review Defects in a Given Category

To review the defects resulting from `Missing or invalid return statement`:

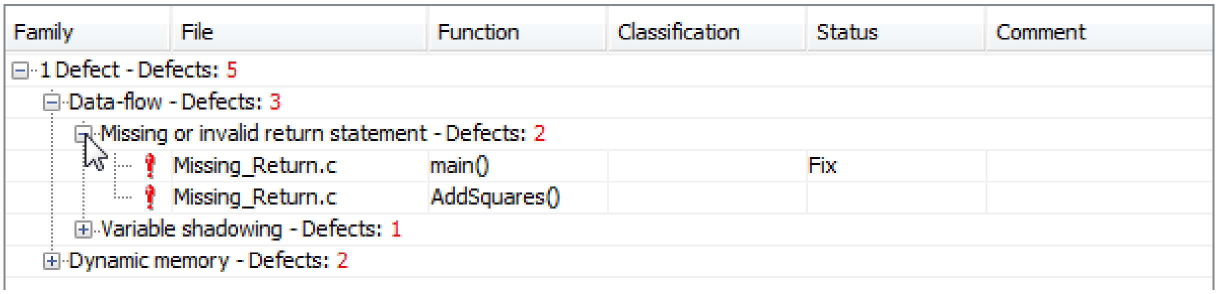
1. On the **Results Summary** tab, from the drop-down list, select `Checks by Family`.

   The defects are grouped by type.



Data-flow 9
Dynamic memory 2
Numerical 8
Other 2
Programming 5
Static memory 8

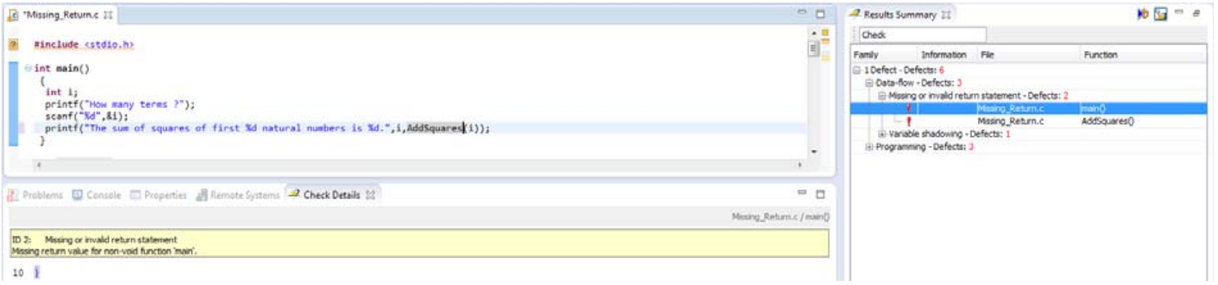
2 Under the category **Data-flow - Defects**, expand the subcategory **Missing or invalid return statement - Defects**.



Family	File	Function	Classification	Status	Comment
1 Defect - Defects: 5					
Data-flow - Defects: 3					
Missing or invalid return statement - Defects: 2					
	Missing_Return.c	main()		Fix	
	Missing_Return.c	AddSquares()			
Variable shadowing - Defects: 1					
Dynamic memory - Defects: 2					

Expand **Missing or invalid return statement - Defects** to view all instances of this defect type.

3 To see further information about an instance, select it. The information appears on the **Check Details** tab.



```
#include <stdio.h>

int main()
 {
  int i;
  printf("How many terms ?");
  scanf("%d",&i);
  printf("The sum of squares of first %d natural numbers is %d.",i,AddSquares(i));
 }
```

Missing_Return.c / main()

ID 2: Missing or invalid return statement
Missing return value for non-void function 'main'.

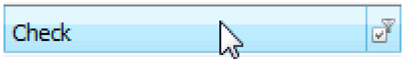
10 }

Family	Information	File	Function
1 Defect - Defects: 6			
Data-flow - Defects: 3			
Missing or invalid return statement - Defects: 2			
!		Missing_Return.c	main()
!		Missing_Return.c	AddSquares()
Variable shadowing - Defects: 1			
Programming - Defects: 3			

4 To view only the defects resulting from `Missing or invalid return statement`, on the **Results Summary** tab, from the drop-down list, select `List of Checks`.

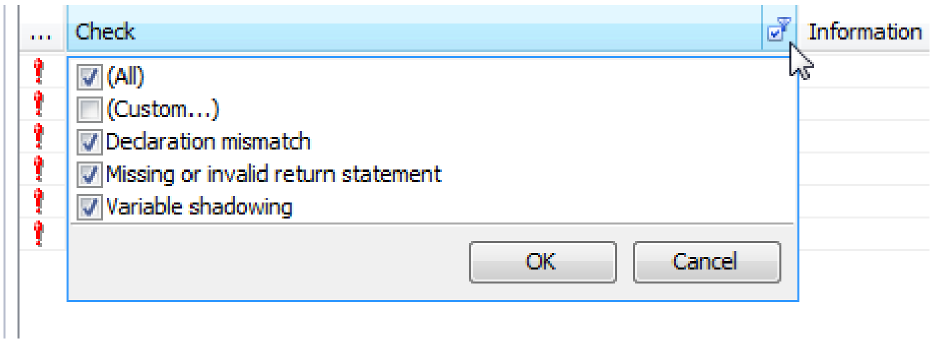
The defects appear ungrouped.

5 Place your cursor on the **Check** column head. The filter icon appears.



6 Click the filter icon.

A context menu lists the filter options available.



7 Clear the **All** check box.

8 Select the **Missing or invalid return statement** check box. Click **OK**.

The **Results Summary** tab displays only the defects resulting from the `Missing or invalid return statement` error.

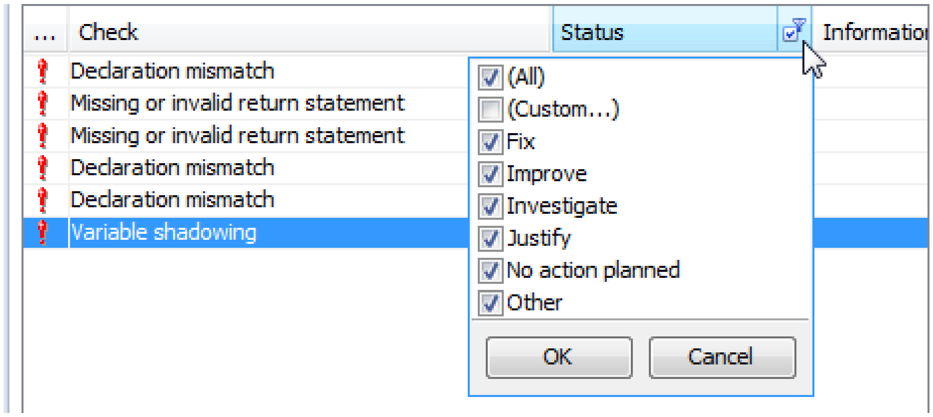
Review Defects with Given Status

To review only the defects with `Investigate` status:

1. On the **Results Summary** tab, place your cursor on the **Status** column head.

2. Click the filter icon.

   A context menu lists the filter options available.



...	Check	Status	Informatio
	Declaration mismatch		
	Missing or invalid return statement		
	Missing or invalid return statement		
	Declaration mismatch		
	Declaration mismatch		
	Variable shadowing		

3. Clear the **All** check box.

4. Select the **Investigate** check box. Click **OK**.

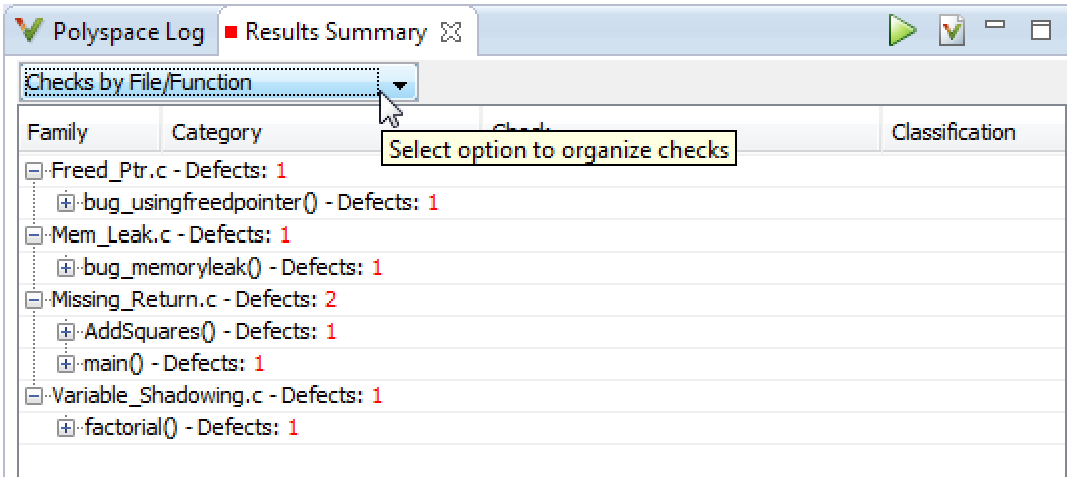
   The **Results Summary** tab displays only the defects with the `Investigate` status.

Review Defects in a File

To review the defects in the file, `Missing_Return.c`:

1. On the **Results Summary** tab, from the drop-down list, select `Checks by File/Function`.

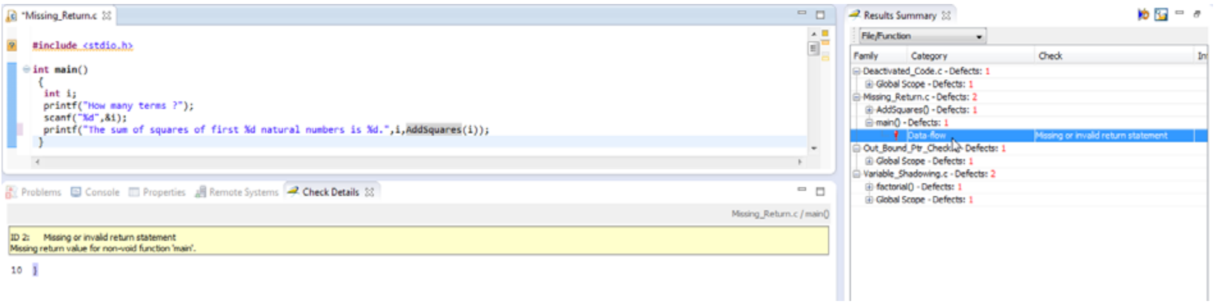
The defects displayed are grouped by files. The file names are sorted alphabetically. Within each file name, the defects are grouped by functions, sorted alphabetically.



```
⊟ Freed_Ptr.c - Defects: 1
   ⊞ bug_usingfreedpointer() - Defects: 1
⊟ Mem_Leak.c - Defects: 1
   ⊞ bug_memoryleak() - Defects: 1
⊟ Missing_Return.c - Defects: 2
   ⊞ AddSquares() - Defects: 1
   ⊞ main() - Defects: 1
⊟ Variable_Shadowing.c - Defects: 1
   ⊞ factorial() - Defects: 1
```

2 To view the defects in `Missing_Return.c`, expand a function name under the category, **Missing_Return.c - Defects**.

To view further information on a bug, select the bug. The information on the bug appears on the **Check Details** tab.



```
#include <stdio.h>

int main()
 {
  int i;
  printf("How many terms ?");
  scanf("%d",&i);
  printf("The sum of squares of first %d natural numbers is %d.",i,AddSquares(i));
 }
```

Missing_Return.c / main()

ID 2: Missing or invalid return statement
Missing return value for non-void function 'main'.

```
10 }
```

```
⊟ Deactivated_Code.c - Defects: 1
   ⊞ Global Scope - Defects: 1
⊟ Missing_Return.c - Defects: 2
   ⊞ AddSquares() - Defects: 1
   ⊟ main() - Defects: 1
        Data-flow        Missing or invalid return statement
⊟ Out_Bound_Ptr_Checking - Defects: 1
   ⊞ Global Scope - Defects: 1
⊟ Variable_Shadowing.c - Defects: 2
   ⊞ factorial() - Defects: 1
   ⊞ Global Scope - Defects: 1
```

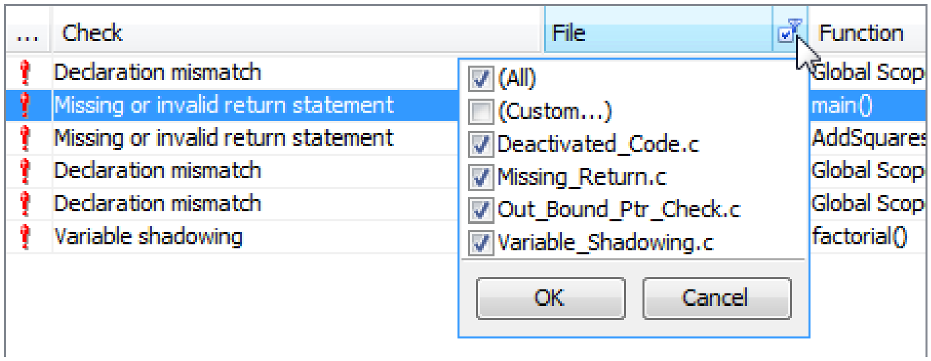
3 To view only the defects in `Missing_Return.c`, on the **Results Summary** tab, from the drop-down list, select `List of Checks`.

The **Results Summary** pane displays all results ungrouped.

4 Place your cursor on the **File** column head.

5 Click the filter icon.

A context menu lists the filter options available.



...	Check	File	Function
	Declaration mismatch		Global Scop
	Missing or invalid return statement		main()
	Missing or invalid return statement		AddSquares
	Declaration mismatch		Global Scop
	Declaration mismatch		Global Scop
	Variable shadowing		factorial()

6 Clear the **All** check box.

7 Select the **Missing_Return.c - Defects** check box. Click **OK**.

The **Results Summary** tab displays only the defects in `Missing_Return.c`.

Tip If you apply a filter on a column on the **Results Summary** pane, the column header displays the number of check boxes selected in the filter menu. Use this information to keep track of the filters you applied.

Related Examples

- "View Results" on page 14-8
- "Review and Fix Results" on page 14-9

View Results

This example shows how to view the results of Polyspace Bug Finder analysis. After you run an analysis, you can view the results either in Eclipse™ or from the Polyspace Bug Finder Results Manager.

View Results in Eclipse

To view results in Eclipse:

1. Run the Polyspace Bug Finder analysis.

   After the analysis, the results open automatically in the **Results Summary** tab.

2. To explicitly open the **Results Summary** tab after the analysis, select **Polyspace > Show View > Show Results Summary view**.

View Results in Polyspace Environment

To view results in the Polyspace Bug Finder Results Manager:

1. Run the Polyspace Bug Finder analysis.
2. Select **Polyspace > Open Results in PVE**.

Related Examples

- “Run Analysis” on page 13-2

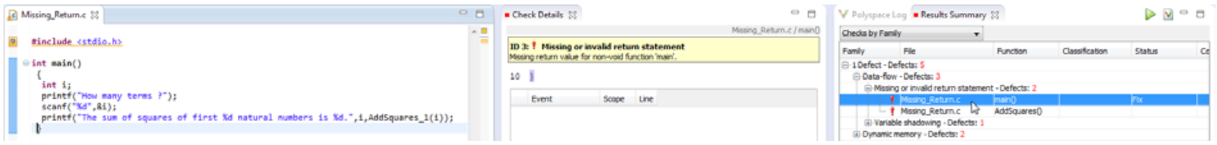
Review and Fix Results

This example shows how to review and comment results obtained from Polyspace Bug Finder analysis. When reviewing results, you can assign a status and classification to the defects 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 defect twice. If you run successive analyses on the same file, the review status, classification and comments from the previous analysis will be automatically imported into the next.

Review and Comment Individual Defect

1 On the **Results Summary** tab, select the defect that you want to review.

   The **Check Details** tab displays information about the current defect.



```
#include <stdio.h>

int main()
 {
  int i;
  printf("How many terms ?");
  scanf("%d",&i);
  printf("The sum of squares of first %d natural numbers is %d.",i,AddSquares_1(i));
 }
```

Missing_Return.c / main()

ID 3: ! Missing or invalid return statement
Missing return value for non-void function 'main'.

```
10 }
```

Event	Scope	Line

Checks by Family

Family	File	Function	Classification	Status
1 Defect - Defects: 5				
Data-flow - Defects: 3				
Missing or invalid return statement - Defects: 2				
!	Missing_Return.c	main()		Fix
!	Missing_Return.c	AddSquares()		
Variable shadowing - Defects: 1				
Dynamic memory - Defects: 2				

2 On the **Results Summary** tab, enter a **Classification** for the defect to describe its severity:

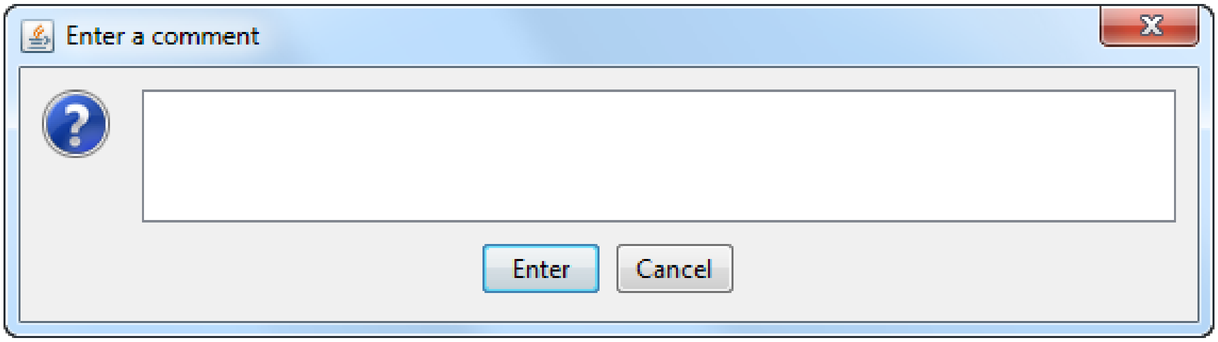
- `High`
- `Medium`
- `Low`
- `Not a defect`

3 On the **Results Summary** tab, enter a **Status** to describe how you intend to address the defect:

- `Fix`
- `Improve`
- `Investigate`
- `Justify`
- `No action planned`

- `Other`

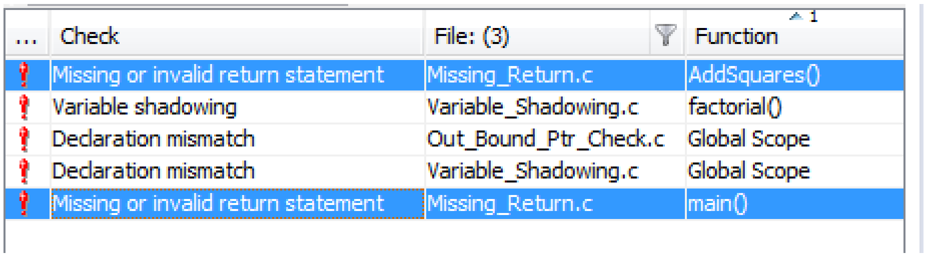
4 On the **Results Summary** tab, click the **Comment** field. Enter remarks, for example, defect or justification information, in the new window that opens.



Review and Comment Group of Defects

1 On the **Results Summary** tab, select a group of defects using one of the following methods:

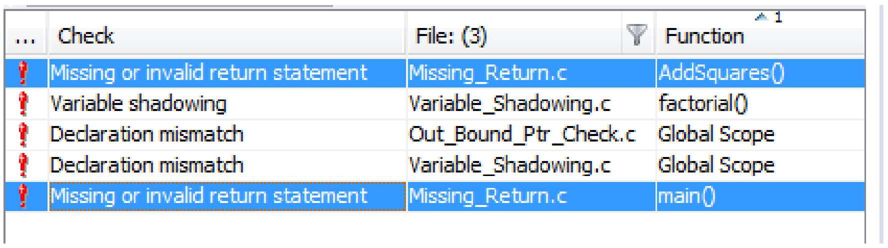
- For contiguous defects, left-click the first defect. Then **Shift**-left click the last defect.



...	Check	File: (3)	Function
!	Missing or invalid return statement	Missing_Return.c	AddSquares()
!	Variable shadowing	Variable_Shadowing.c	factorial()
!	Declaration mismatch	Out_Bound_Ptr_Check.c	Global Scope
!	Declaration mismatch	Variable_Shadowing.c	Global Scope
!	Missing or invalid return statement	Missing_Return.c	main()

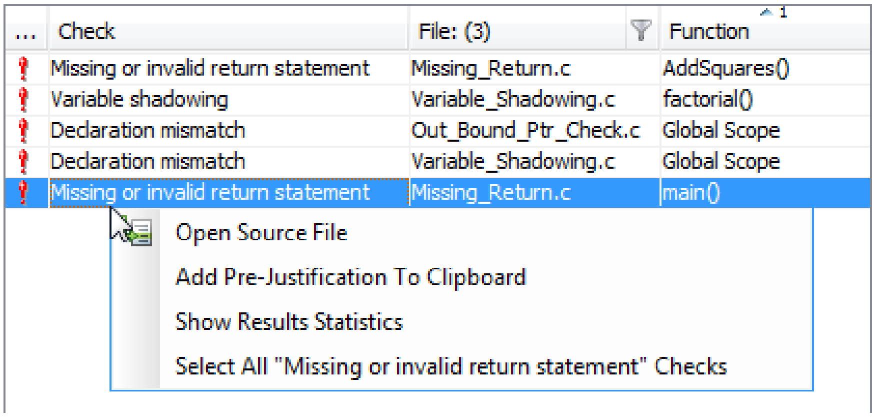
  To group together defects belonging to a certain category, click the **Check** column header on the **Results Summary** tab.

- For non-contiguous defects, **Ctrl**-left click each defect.



...	Check	File: (3)	Function
❗	Missing or invalid return statement	Missing_Return.c	AddSquares()
❗	Variable shadowing	Variable_Shadowing.c	factorial()
❗	Declaration mismatch	Out_Bound_Ptr_Check.c	Global Scope
❗	Declaration mismatch	Variable_Shadowing.c	Global Scope
❗	Missing or invalid return statement	Missing_Return.c	main()

- For defects of a similar category, right-click one defect from that category. From the context menu, select **Select All "*Defect Category*" Checks**, for instance, **Select All "Missing or invalid return statement" Checks**.



...	Check	File: (3)	Function
❗	Missing or invalid return statement	Missing_Return.c	AddSquares()
❗	Variable shadowing	Variable_Shadowing.c	factorial()
❗	Declaration mismatch	Out_Bound_Ptr_Check.c	Global Scope
❗	Declaration mismatch	Variable_Shadowing.c	Global Scope
❗	Missing or invalid return statement	Missing_Return.c	main()

Open Source File
Add Pre-Justification To Clipboard
Show Results Statistics
Select All "Missing or invalid return statement" Checks

2 On the **Results Summary** tab, enter **Classification**, **Status** and **Comments**. The software applies this information to all selected defects.

Related Examples

- "View Results" on page 14-8
- "Filter and Group Results" on page 14-2

Understanding the Results Views

In this section...

"Results Summary" on page 14-12

"Check Details" on page 14-14

Results Summary

The **Results Summary** pane lists the defects and coding rule violations along with their attributes. To organize your results review, from the drop-down list on this pane, select one of the following options:

- `List of checks`: Lists defects and coding rule violations in alphabetical order.
- `Checks by Family`: Lists results grouped by category. For more information on the defects covered by a category, see "Polyspace Bug Finder Defects".
- `Checks by Class`: Lists results grouped by class. Within each class, the results are grouped by method. The first group, **Global Scope**, lists results not occurring in a class definition.

  This option is available for `C++` code only.
- `Checks by File/Function`: Lists results grouped by file. Within each file, the results are grouped by function.

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

Attribute	Description
Family	Group to which the defect belongs. For instance, if you choose the grouping `Checks by File/Function`, this column contains the name of the file and function containing the defect.
ID	Unique identification number of the defect. In the default view on the **Results Summary** pane, the defects appear sorted by this number.
Type	Defect or coding rule violation.
Category	Category of the defect. For more information on the defects covered by a category, see the defect reference pages.
Check	Description of the defect
File	File containing the instruction where the defect occurs
Class	Class containing the instruction where the defect occurs. If the defect is not inside a class definition, then this column contains the entry, `Global Scope`.
Function	Function containing the instruction where the defect occurs. If the function is a method of a class, it appears in the format *`class_name`*`::`*`function_name`*.

Attribute	Description
Classification	Level of severity you have assigned to the defect. The possible levels are: • `High` • `Medium` • `Low` • `Not a defect`
Status	Review status you have assigned to the check. The possible statuses are: • `Fix` • `Improve` • `Investigate` • `Justify` • `No action planned` • `Other`
Comments	Comments you have entered about the check

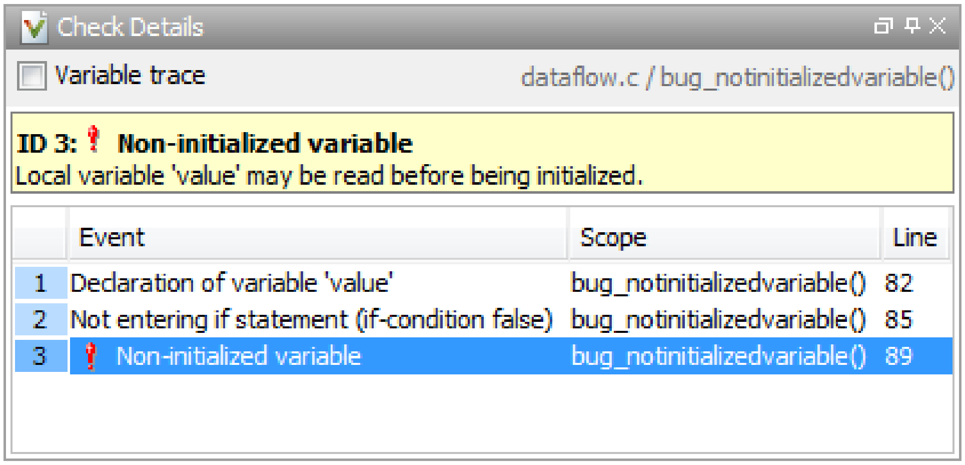
To show or hide any of the columns, right-click anywhere on the column titles. From the context menu, select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through the checks. For more information, see “Review and Fix Results” on page 14-9.
- Organize your check review using filters on the columns. For more information, see “Filter and Group Results” on page 14-2.

Check Details

The **Check Details** pane contains detailed information about a specific defect. Select a defect on the **Results Summary** pane to reveal further information about the defect on the **Check Details** pane.


Check Details

Variable trace — dataflow.c / bug_notinitializedvariable()

ID 3: Non-initialized variable
Local variable 'value' may be read before being initialized.

	Event	Scope	Line
1	Declaration of variable 'value'	bug_notinitializedvariable()	82
2	Not entering if statement (if-condition false)	bug_notinitializedvariable()	85
3	Non-initialized variable	bug_notinitializedvariable()	89

- The top right hand corner shows the file and function containing the defect, in the format `file_name/function_name`.
- The yellow box contains the name of the defect, along with an explanation.
- The **Event** column lists the sequence of code instructions causing the defect. The **Scope** column lists the name of the function containing the instructions. The **Line** column lists the line number of the instructions.
- The **Variable trace** check box when selected reveals an additional set of instructions that are related to the defect.

15

Check Coding Rules from Microsoft Visual Studio

Activate C++ Coding Rules Checker

To check coding rule compliance, before running an analysis, you must set an option in your project. Polyspace software finds the violations during the compile phase. You can view coding rule violations alongside your analysis results.

To set the rule checking option:

1. Select the files you wish to analyze.
2. Right-click on your selection and select **Edit Polyspace Configuration**.
3. In the Polyspace Bug Finder Configuration window, from the Configuration tree, select **Coding Rules**.
4. Under **Coding Rules**, select the check box next to the type of coding rules you wish to check.

   For C++ code, you can check compliance with MISRA C++ or JSF C++, and a custom rules file.
5. For MISRA and JSF rule checking, you can select a subset of rules to check from the corresponding drop-down list.

   The tables below show the options for each coding rule set:

   MISRA C++

Option	Explanation
`required-rules`	All *required* MISRA C++ coding rules. Violations are reported as warnings.
`all-rules`	All *required* and *advisory* MISRA C++ coding rules. Violations are reported as warnings.
`SQO-subset1`	A subset of MISRA C++ rules that have a direct impact on the selectivity. Violations are reported as warnings. For more information, see “Software Quality Objective Subsets (C++)” on page 3-60.

Option	Explanation
`SQO-subset2`	A second subset of rules that have an indirect impact on the selectivity, as well as the rules contained in `SQO-subset1`. Violations are reported as warnings. For more information, see "Software Quality Objective Subsets (C++)" on page 3-60.
`custom`	A specified set of MISRA C++ coding rules. When you select this option, you must specify the MISRA C++ rules to check and whether to report an error or warning for violations of each rule. For more information, see "Select Specific MISRA or JSF Coding Rules" on page 4-6.

JSF C++

Option	Explanation
`shall-rules`	All **Shall** rules, which are mandatory rules that require checking.
`shall-will-rules`	All **Shall** and **Will** rules. **Will** rules are mandatory rules that do not require checking.
`all-rules`	All **Shall**, **Will**, and **Should** rules. **Should** rules are advisory rules.
`custom`	A specified set of JSF C++ coding rules. When you select this option, you must specify the JSF C++ rules to check and whether to report an error or warning for violations of each rule. For more information, see "Select Specific MISRA or JSF Coding Rules" on page 4-6.

6 For Custom rule checking, in the corresponding field, specify the path to your custom rules file or click **Edit** to create one. See "Create Custom Coding Rules" on page 4-8 for more information.

7 Save you changes and close the configuration window.

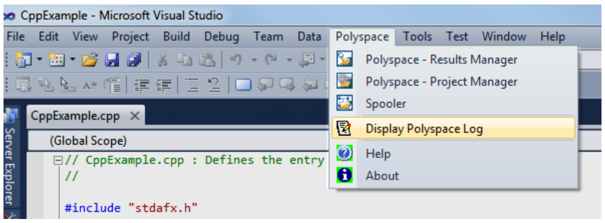
When you run an analysis, Polyspace checks coding rule compliance during the compilation phase of the analysis.

Find Bugs from Microsoft Visual Studio

- "Run Analysis" on page 16-2
- "Monitor Analysis" on page 16-5
- "Customize Polyspace Options" on page 16-6

Run Analysis

1. From Visual Studio, select **Polyspace > Display Polyspace Log** to view the Polyspace Log window.

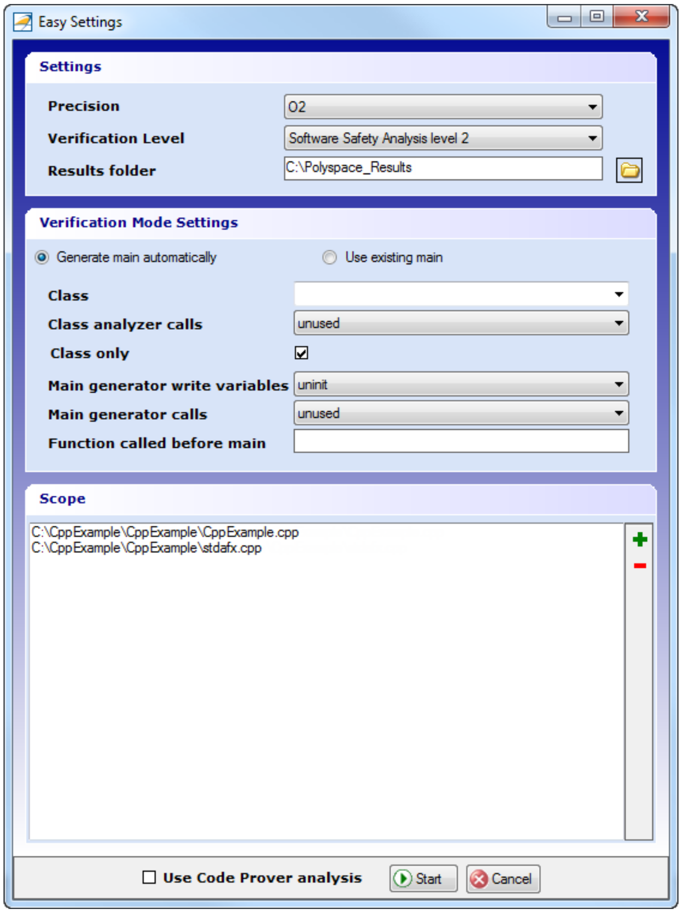


```
// CppExample.cpp : Defines the entry
//

#include "stdafx.h"
```

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

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

   The Easy Settings dialog box opens.



Settings

Precision	O2
Verification Level	Software Safety Analysis level 2
Results folder	C:\Polyspace_Results

Verification Mode Settings

(● Generate main automatically) ○ Use existing main

Class	
Class analyzer calls	unused
Class only	✓
Main generator write variables	uninit
Main generator calls	unused
Function called before main	

Scope

C:\CppExample\CppExample\CppExample.cpp
C:\CppExample\CppExample\stdafx.cpp

☐ Use Code Prover analysis

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

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

  - **Generate main** or **Use existing** — Whether Polyspace generates a `main` subprogram (`-main-generator`) or uses an existing subprogram (`-main`)
  - **Class** — Name of class to analyze (`-class-analyzer`)
  - **Class analyzer calls** — Functions called by generated `main` subprogram (`-class-analyzer-calls`)
  - **Class only** — Analysis of class contents only (`-class-only`)
  - **Main generator write** — Type of initialization for global variables (`-main-generator-writes-variables`)
  - **Main generator calls** — Functions (not in a class) called by generated `main` subprogram (`-main-generator-calls`)
  - **Function called before main** — Function called before all functions (`-function-call-before-main`)
- Under **Scope**, you can modify the list of files and classes to analyze.

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

Note In the Project Manager perspective of the Polyspace interface, you configure options that you cannot set in the Easy Settings dialog box. See “Customize Polyspace Options” on page 16-6.

5 Make sure the **Use Code Prover analysis** check box is cleared.

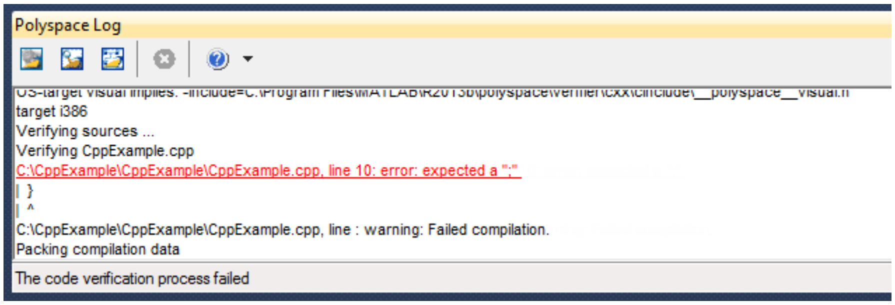
6 Click **Start** to start the analysis.

To follow the progress of an analysis, see “Monitor Analysis” on page 16-5

Monitor Analysis

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

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



```
OS-target visual implies: -include=C:\Program Files\MATLAB\R2013b\polyspace\verifier\cxx\cinclude\__polyspace__visual.h
target i386
Verifying sources ...
Verifying CppExample.cpp
C:\CppExample\CppExample\CppExample.cpp, line 10: error: expected a ";"
| }
| ^
C:\CppExample\CppExample\CppExample.cpp, line : warning: Failed compilation.
Packing compilation data
```

The code verification process failed

If the analysis is being carried out on a server, follow its progress in the Polyspace Queue Manager.

Select **Polyspace > Spooler** to follow the progress in the Polyspace Queue Manager.

Customize Polyspace Options

In the Easy Settings dialog box in Visual Studio, you specify only a subset of the Polyspace analysis options.

To customize other analysis options:

1. Select the files you wish to analyze.
2. Right-click on your selection and select **Edit Polyspace Configuration** from the context menu.
3. In the Polyspace Bug Finder configuration window, use the different panes to customize your analysis options.

   For more information about specific options, see "Analysis Options for C++".
4. Save your changes and close the configuration window.

   Next time you run an analysis, Polyspace uses the `ProjectName_UserSettings.psprj` settings.

17

Open Results from Microsoft Visual Studio

Open Results in Polyspace Environment

To view your results:

- From the Polyspace Log window, select .
- Select **Polyspace > Polyspace — Results Manager**, then open results from the Polyspace interface. For instructions, see "Open Results" on page 6-2