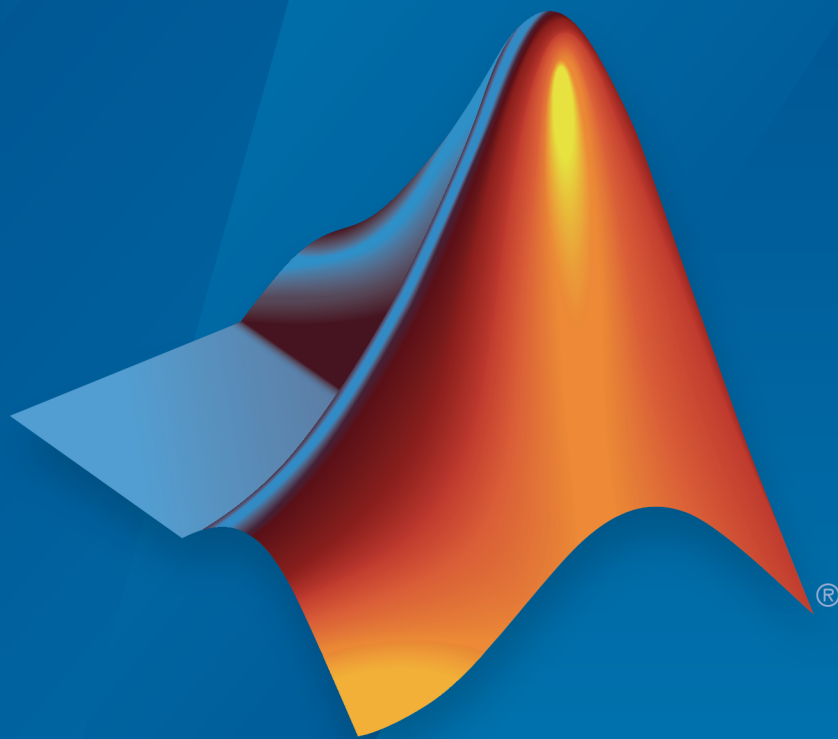


Polyspace[®] Bug Finder[™]

User's Guide



MATLAB[®]&SIMULINK[®]

R2015b



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Polyspace[®] Bug Finder[™] User's Guide

© COPYRIGHT 2013–2015 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

September 2013	Online only	New for Version 1.0 (Release 2013b)
March 2014	Online Only	Revised for Version 1.1 (Release 2014a)
October 2014	Online only	Revised for Version 1.2 (Release 2014b)
March 2015	Online only	Revised for Version 1.3 (Release 2015a)
September 2015	Online only	Revised for Version 2.0 (Release 2015b)

Project Configuration

What Is a Project?	1-3
What is a Project Template?	1-4
Open Polyspace Bug Finder	1-5
Create New Project	1-6
Create Project Automatically	1-7
Requirements for Project Creation from Build Systems ...	1-10
Compiler Not Supported for Project Creation from Build Systems	1-13
Issue	1-13
Cause	1-13
Solution	1-13
Slow Build Process When Polyspace Traces the Build	1-20
Issue	1-20
Cause	1-20
Solution	1-20
Checking if Polyspace Supports Windows Build Command	1-21
Issue	1-21
Possible Cause	1-21
Solution	1-21
Create Project Using Visual Studio Information	1-23

Troubleshooting Project Creation from Visual Studio	
Build	1-27
Cannot Create Project from Visual Studio Build	1-27
Compilation Error After Creating Project from Visual Studio Build	1-27
Add Source Files and Include Folders	1-29
Add Sources and Includes	1-29
Manage Include File Sequence	1-29
Specify Analysis Options	1-31
About Analysis Options	1-31
Specify Options in User Interface	1-32
Specify Options from DOS and UNIX Command Line	1-32
Specify Options from MATLAB Command Line	1-33
Save Analysis Options as Project Template	1-34
Organize Layout of Polyspace User Interface	1-38
Create Your Own Layout	1-38
Save and Reset Layout	1-39
Specify External Text Editor	1-40
Change Default Font Size	1-42
Define Custom Review Status	1-43
Modeling Multitasking Code	1-47
Example	1-47
Limitations	1-50
Set Up Multitasking Analysis Manually	1-52
Prerequisites	1-52
Set Up Multitasking Analysis in User Interface	1-53
Set Up Multitasking Analysis at Command Line	1-53
Set Up Multitasking Analysis at MATLAB Command Line ..	1-54
Annotate Code for Known Defects	1-55
How to Add Annotations	1-55
Syntax for Code Annotations	1-55

Annotate Code for Rule Violations	1-58
How to Add Annotations	1-58
Syntax for Code Annotations	1-59
Copy and Paste Annotations	1-61
Modify Predefined Target Processor Attributes	1-63
Specify Generic Target Processors	1-65
Define Generic Target	1-65
Common Generic Targets	1-66
View or Modify Existing Generic Targets	1-67
Delete Generic Target	1-68
Compile Operating System-Dependent Code	1-69
My Target Application Runs on Solaris	1-69
My Target Application Runs on Vxworks	1-69
My Target Application Does Not Run on Linux, VxWorks, or Solaris	1-70
Address Alignment	1-71
Ignore or Replace Keywords Before Compilation	1-72
Content of myTpl.pl file	1-72
Perl Regular Expression Summary	1-73
Analyze Keil or IAR Dialects	1-75
Supported C++ 2011 Extensions	1-81
Gather Compilation Options Efficiently	1-85
Specify Constraints	1-87
Create Constraint Template	1-87
Update Existing Template	1-89
Constraints	1-90
Storage of Polyspace Preferences	1-94

Rule Checking	2-2
Polyspace Coding Rule Checker	2-2
Differences Between Bug Finder and Code Prover	2-2
Polyspace MISRA C 2004 and MISRA AC AGC Checkers ...	2-4
Software Quality Objective Subsets (C:2004)	2-5
Rules in SQO-Subset1	2-5
Rules in SQO-Subset2	2-6
Software Quality Objective Subsets (AC AGC)	2-10
Rules in SQO-Subset1	2-10
Rules in SQO-Subset2	2-11
MISRA C:2004 and MISRA AC AGC Coding Rules	2-14
Supported MISRA C:2004 and MISRA AC AGC Rules	2-14
Unsupported MISRA C:2004 and MISRA AC AGC Rules ...	2-50
Polyspace MISRA C:2012 Checker	2-53
Software Quality Objective Subsets (C:2012)	2-54
Guidelines in SQO-Subset1	2-54
Guidelines in SQO-Subset2	2-55
Unsupported MISRA C:2012 Guidelines	2-59
Polyspace MISRA C++ Checker	2-60
Software Quality Objective Subsets (C++)	2-61
SQO Subset 1 – Direct Impact on Selectivity	2-61
SQO Subset 2 – Indirect Impact on Selectivity	2-63
MISRA C++ Coding Rules	2-68
Supported MISRA C++ Coding Rules	2-68
Unsupported MISRA C++ Rules	2-89
Polyspace JSF C++ Checker	2-95

JSF C++ Coding Rules	2-96
Supported JSF C++ Coding Rules	2-96
Unsupported JSF++ Rules	2-119

Check Coding Rules from the Polyspace Environment

3

Activate Coding Rules Checker	3-2
Select Specific MISRA or JSF Coding Rules	3-6
Create Custom Coding Rules	3-9
Format of Custom Coding Rules File	3-11
Exclude Files From Analysis	3-12
Allow Custom Pragma Directives	3-13
Specify Boolean Types	3-14
Find Coding Rule Violations	3-15
Review Coding Rule Violations	3-16
Filter and Group Coding Rule Violations	3-18
Filter Coding Rules	3-18
Group Coding Rules	3-18
Suppress Certain Rules from Display in One Click	3-18
Rules to Disable for Faster Analysis	3-21
MISRA C: 2004 and MISRA AC AGC	3-21
MISRA C: 2012	3-21

Find Bugs From the Polyspace Environment

4

Choose Specific Defects	4-2
Run Local Analysis	4-3
Run Remote Batch Analysis	4-4
Monitor Analysis	4-5
Specify Results Folder	4-6

View Results in the Polyspace Environment

5

Open Results	5-2
Open Results From Active Project	5-2
Open Results File From File Browser	5-2
View Results Summary in Polyspace Metrics	5-4
Download Results From Polyspace Metrics	5-6
Filter and Group Results	5-9
Filter Results	5-9
Group Results	5-10
Classification of Defects by Impact	5-12
High Impact Defects	5-12
Medium Impact Defects	5-14
Low Impact Defects	5-17
Limit Display of Defects	5-20
Generate Reports	5-22
Review and Fix Results	5-24
Assign and Save Comments	5-24

Import Review Comments from Previous Analysis	5-25
Review Concurrency Defects	5-27
Review Code Metrics	5-30
Impose Limits on Metrics	5-30
Comment and Justify Limit Violations	5-33
Navigate to Root Cause of Defect	5-34
Navigate Code Sequence Causing Defect	5-34
Navigate to Identifier Definition	5-35
Navigate to Identifier References	5-35
Results Folder Contents	5-37
Files in the Results Folder	5-37
Windows Used to Review Results	5-38
Dashboard	5-38
Results Summary	5-42
Source	5-44
Result Details	5-50
Bug Finder Defect Groups	5-52
Concurrency	5-52
Data flow	5-53
Dynamic Memory	5-53
Good Practice	5-53
Numerical	5-54
Object Oriented	5-54
Programming	5-54
Resource Management	5-54
Static Memory	5-55
Security	5-55
Tainted data	5-55
HIS Metrics	5-57
Project	5-57
File	5-57
Function	5-57
Common Weakness Enumeration from Bug Finder Defects	5-59
Common Weakness Enumeration	5-59
Polyspace Bug Finder and CWE Compatibility	5-59

Find CWE Identifiers from Defects	5-61
View CWE Identifiers	5-61
Filter CWE Identifiers	5-61
Generate Report with CWE Identifiers	5-61
Mapping Between CWE Identifiers and Defects	5-63

Command-Line Analysis

6

Create Project Automatically at Command Line	6-2
Run Local Analysis from Command Line	6-4
Specify Sources and Analysis Options Directly	6-4
Specify Sources and Analysis Options in Text File	6-5
Create Options File from Build System	6-5
Run Remote Analysis at Command Line	6-6
Run Remote Analysis	6-6
Manage Remote Analysis	6-7
Download Results	6-9
Create Project Automatically from MATLAB Command Line	6-10

Polyspace Bug Finder Analysis in Simulink

7

Embedded Coder Considerations	7-2
Default Options	7-2
Recommended Polyspace Bug Finder Options for Analyzing Generated Code	7-3
Hardware Mapping Between Simulink and Polyspace	7-4
TargetLink Considerations	7-5
TargetLink Support	7-5
Default Options	7-5

Lookup Tables	7-6
Code Generation Options	7-6
Generate and Analyze Code	7-7
Main Generation for Model Analysis	7-14
Review Generated Code Results	7-16
Troubleshoot Back to Model	7-18
Back-to-Model Links Do Not Work	7-18
Your Model Already Uses Highlighting	7-18

Configure Model for Code Analysis

8

Configure Simulink Model	8-2
Recommended Model Settings for Code Analysis	8-3
Check Simulink Model Settings	8-6
Check Simulink Model Settings Using the Code Generation Advisor	8-6
Check Simulink Model Settings Before Analysis	8-7
Check Simulink Model Settings Automatically	8-9
Annotate Blocks for Known Results	8-12

Configure Code Analysis Options

9

Polyspace Configuration for Generated Code	9-2
Include Handwritten Code	9-3
Configure Analysis Depth for Referenced Models	9-4

Check Coding Rules Compliance	9-5
Configure Polyspace Analysis Options and Properties	9-7
Set Advanced Analysis Options	9-7
Save a Polyspace Configuration File Template	9-8
Use a Custom Configuration File	9-9
Remove Polyspace Options From Simulink Model	9-9
Set Custom Target Settings	9-11
Set Up Remote Batch Analysis	9-14
Manage Results	9-15
Open Polyspace Results Automatically	9-15
Specify Location of Results	9-16
Save Results to a Simulink Project	9-17
Specify Signal Ranges	9-18
Specify Signal Range through Source Block Parameters ...	9-18
Specify Signal Range through Base Workspace	9-20

Run Polyspace on Generated Code

10

Specify Type of Analysis to Perform	10-2
Run Analysis for Embedded Coder	10-5
Run Analysis for TargetLink	10-6
Monitor Progress	10-7
Local Analyses	10-7
Remote Batch Analyses	10-7

Check Coding Rules from Eclipse

11

Activate Coding Rules Checker	11-2
Select Specific MISRA or JSF Coding Rules	11-6
Create Custom Coding Rules File	11-9
Contents of Custom Coding Rules File	11-11
Exclude Files From Analysis	11-12
Allow Custom Pragma Directives	11-13
Specify Boolean Types	11-14
Find Coding Rule Violations	11-15
Review Coding Rule Violations	11-16
Filter and Group Coding Rule Violations	11-18
Filter Coding Rules	11-18
Group Coding Rules	11-18
Suppress Certain Rules from Display in One Click	11-18

Find Bugs from Eclipse

12

Run Analysis	12-2
Customize Analysis Options	12-3

View Results in Eclipse

13

View Results	13-2
View Results in Eclipse	13-2
View Results in Polyspace Environment	13-2
Review and Fix Results	13-3
Filter and Group Results	13-5
Filter Results	13-5
Group Results	13-6
Understanding the Results Views	13-8
Results Summary	13-8
Result Details	13-10

Check Coding Rules from Microsoft Visual Studio

14

Activate C++ Coding Rules Checker	14-2
Exclude Files From Analysis	14-4

Find Bugs from Microsoft Visual Studio

15

Run Polyspace in Visual Studio	15-2
Monitor Progress in Visual Studio	15-5
Local Analysis	15-5
Remote Analysis	15-7
Customize Polyspace Options	15-8
Configuration File and Default Options	15-9

Bug Finding in Visual Studio	15-10
------------------------------------	-------

16 | **Open Results from Microsoft Visual Studio**

Open Results in Polyspace Environment	16-2
---	------

17 | **Troubleshooting in Polyspace Bug Finder**

View Error Information When Verification Stops	17-2
View Error Information in User Interface	17-2
View Error Information in Log File	17-2
Troubleshoot Compilation and Linking Errors	17-4
Contact Technical Support	17-5
Provide System Information	17-5
Provide Information About the Issue	17-5
Header File Location Not Specified	17-7
Message	17-7
Possible Cause	17-7
Solution	17-7
Polyspace Cannot Find the Server	17-8
Message	17-8
Possible Cause	17-8
Solution	17-8
Insufficient Memory During Report Generation	17-9
Message	17-9
Possible Cause	17-9
Solution	17-9

Errors From Disk Defragmentation and Antivirus	
Software	17-10
Message	17-10
Possible Cause	17-10
Solution	17-10
Syntax Errors Due to Unknown Keywords	17-11
Message	17-11
Code Used	17-11
Cause	17-11
Solution	17-11
Undeclared Identifier	17-12
Message	17-12
Code Used	17-12
Cause	17-12
Solution	17-12
Missing Identifiers with Keil or IAR Dialect	17-13
Message	17-13
Possible Cause	17-13
Solution	17-13
Unknown Prototype	17-14
Message	17-14
Code Used	17-14
Cause	17-14
Solution	17-14
Cannot Find Include File	17-16
Messages	17-16
Code Used	17-16
Cause	17-16
Solution	17-16
#error Directive	17-17
Message	17-17
Code Used	17-17
Cause	17-17
Solution	17-17
Object is Too Large	17-18
Issue	17-18

Message	17-18
Code Used	17-18
Solution	17-18
Errors From Special Characters	17-21
Workaround	17-21
Initialization of Static Class Members (C++)	17-22
Double Declarations of Standard Template Library Functions	17-23
GNU Dialect	17-24
Partial Support	17-24
Syntactic Support Only	17-25
Not Supported	17-25
Examples	17-25
ISO versus Default Dialects	17-27
Visual Dialects	17-29
Import Folder	17-29
pragma Pack	17-29
Eclipse Java Version Incompatible with Polyspace Plug- in	17-31
Issue	17-31
Cause	17-31
Solution	17-31

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
- “Create New Project” on page 1-6
- “Create Project Automatically” on page 1-7
- “Requirements for Project Creation from Build Systems” on page 1-10
- “Compiler Not Supported for Project Creation from Build Systems” on page 1-13
- “Slow Build Process When Polyspace Traces the Build” on page 1-20
- “Checking if Polyspace Supports Windows Build Command” on page 1-21
- “Create Project Using Visual Studio Information” on page 1-23
- “Troubleshooting Project Creation from Visual Studio Build” on page 1-27
- “Add Source Files and Include Folders” on page 1-29
- “Specify Analysis Options” on page 1-31
- “Save Analysis Options as Project Template” on page 1-34
- “Organize Layout of Polyspace User Interface” on page 1-38
- “Specify External Text Editor” on page 1-40
- “Change Default Font Size” on page 1-42
- “Define Custom Review Status” on page 1-43
- “Modeling Multitasking Code” on page 1-47
- “Set Up Multitasking Analysis Manually” on page 1-52
- “Annotate Code for Known Defects” on page 1-55
- “Annotate Code for Rule Violations” on page 1-58
- “Copy and Paste Annotations” on page 1-61
- “Modify Predefined Target Processor Attributes” on page 1-63
- “Specify Generic Target Processors” on page 1-65

- “Compile Operating System-Dependent Code” on page 1-69
- “Address Alignment” on page 1-71
- “Ignore or Replace Keywords Before Compilation” on page 1-72
- “Analyze Keil or IAR Dialects” on page 1-75
- “Supported C++ 2011 Extensions” on page 1-81
- “Gather Compilation Options Efficiently” on page 1-85
- “Specify Constraints” on page 1-87
- “Constraints” on page 1-90
- “Storage of Polyspace Preferences” on page 1-94

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

In the Polyspace interface, use the Project Browser and Configuration panes 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 1-34.

Open Polyspace Bug Finder

After you install MATLAB® and Polyspace, you can open Polyspace Bug Finder™ from the desktop shortcut created during installation. Other ways to open Polyspace are:

- via MATLAB.
 - In the apps gallery, select Polyspace Bug Finder.
 - In the Command Window, enter:
`polyspaceBugFinder`
- via the command-line.
 - DOS: *MATLAB Install*\polyspace\bin\polyspace-bug-finder
 - UNIX: *MATLAB Install*/polyspace/bin/polyspace-bug-finder

Where *MATLAB Install* is your MATLAB installation folder.

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.

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**

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 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**.

4 Click **Finish**.

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

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


To close the project at any time, in the **Project Browser**, right-click the project node and select **Close**.

Create Project Automatically

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. For more information on these options, see:
 - C Code: “Target & Compiler”
 - C++ Code: “Target & Compiler”

- 1 Select **File > New Project**.
- 2 On the Project – Properties dialog box, after specifying the project name, location and language, 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	<p>If you use an IDE such as Visual Studio® or Eclipse™ to build your project, specify the full path to your IDE executable or navigate to it using the  button. For a tutorial using Visual Studio, see “Create Project Using Visual Studio Information” on page 1-23.</p> <p>Example: "C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\VCEXpress.exe"</p> <p>If you use command-line tools to build your project, specify the appropriate command.</p> <p>Example:</p> <ul style="list-style-type: none"> • <code>make -B -f makefileName</code> or <code>make -W makefileName</code> • <code>"mingw32-make.exe -B -f makefilename"</code>
Specify working directory for	Specify the folder from which you run your build automation script.

Field	Description
running build command	If you specify the full path to your executable in the previous field, this field is redundant. Specify any folder.
Add advanced configure options	Specify additional options for advanced tasks such as incremental build. For the full list of options, see the <code>-options</code> value argument for <code>polyspaceConfigure</code> .

4 Click .

- 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.

If there is a failure, see if your build command meets the requirements for automatic project setup. In some cases, you can modify your build command to work around the limitations. For more information, see “Requirements for Project Creation from Build Systems” on page 1-10.

5 Click **Finish**.

The new project appears on the **Project Browser** pane. To close the project at any time, in the **Project Browser**, right-click the project node and select **Close**.

6 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**.

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.
- By default, Polyspace assigns the latest dialect for your compiler. If you have compilation errors in your project, check the dialect. If it does not apply to you, change it to a more appropriate one.

- If your build process requires user interaction, you cannot run the build command from the Polyspace user interface and do an automatic project setup.
-

Related Examples

- “Create Project Using Visual Studio Information” on page 1-23

More About

- “Compiler Not Supported for Project Creation from Build Systems” on page 1-13
- “Slow Build Process When Polyspace Traces the Build” on page 1-20
- “Checking if Polyspace Supports Windows Build Command” on page 1-21

Requirements for Project Creation from Build Systems

For automatic project creation from build systems, your build commands or makefiles must meet certain requirements.

For more information on automatic project creation, see:

- “Create Project Automatically” on page 1-7
- “Create Project Automatically at Command Line” on page 6-2
- “Create Project Automatically from MATLAB Command Line” on page 6-10

The requirements for your build command are as follows:

- Your compiler must be called locally.

If you use a compiler cache such as `ccache` or a distributed build system such as `distmake`, the software cannot trace your build. You must deactivate them.

- Your compiler must perform a clean build.

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` or `-W makefileName` option to force a clean build. For the list of options allowed with the GNU® `make`, see `make` options.

- Your compiler configuration must be available to Polyspace. The compilers currently supported include the following:
 - Visual C++® compiler
 - `gcc`
 - `clang`
 - MinGW compiler
 - IAR compiler

If your compiler configuration is not available to Polyspace:

- Write a compiler configuration file for your compiler in a specific format. For more information, see “Compiler Not Supported for Project Creation from Build Systems” on page 1-13.
- Contact MathWorks Technical Support. For more information, see “Contact Technical Support”.

- In Linux[®], only UNIX[®] shell (sh) commands must be used. If your build uses advanced commands such as commands supported only by bash, tcsh or zsh, Polyspace cannot trace your build.

In Windows[®], only DOS commands must be used. If your build uses advanced commands such as commands supported only by PowerShell or Cygwin[™], Polyspace cannot trace your build. To see if Polyspace supports your build command, run the command from `cmd.exe` in Windows. For more information, see “Checking if Polyspace Supports Windows Build Command” on page 1-21.

- Your build command must not use aliases.

The `alias` command is used in Linux to create an alternate name for commands. If your build command uses those alternate names, Polyspace cannot recognize them.

- Your build command must be executable completely on the current machine and must not require privileges of another user.

If your build uses `sudo` to change user privileges or `ssh` to remotely login to another machine, Polyspace cannot trace your build.

- If your build command uses redirection with the `>` or `|` character, the redirection occurs after Polyspace traces the command. Therefore, Polyspace does not handle the redirection.

For example, if your command occurs as

```
command1 | command2
```

And you enter

```
polyspace-configure command1 | command2
```

When tracing the build, Polyspace traces the first command only.

- If your computer hibernates during the build process, Polyspace might not be able to trace your build.

Note: Your environment variables are preserved when Polyspace traces your build command.

See Also

`polyspaceConfigure`

Related Examples

- “Create Project Automatically” on page 1-7

More About

- “Slow Build Process When Polyspace Traces the Build” on page 1-20

Compiler Not Supported for Project Creation from Build Systems

Issue

Your compiler is not supported for automatic project creation from build commands.

For more information on automatic project creation, see:

- “Create Project Automatically” on page 1-7
- “Create Project Automatically at Command Line” on page 6-2
- “Create Project Automatically from MATLAB Command Line” on page 6-10

Cause

For automatic project creation from your build system, your compiler configuration must be available to Polyspace. Polyspace provides a compiler configuration file only for certain compilers.

For information on which compilers are supported, see “Requirements for Project Creation from Build Systems” on page 1-10.

Solution

To enable automatic project creation for an unsupported compiler, you can write your own compiler configuration file.

- 1 Copy one of the existing configuration files from `matlabroot\polyspace\configure\compiler_configuration\`.
- 2 Save the file as `my_compiler.xml`. `my_compiler` can be any name that helps you identify the file.

To edit the file, save it outside the installation folder. After you have finished editing, you must copy the file back to `matlabroot\polyspace\configure\compiler_configuration\`.

- 3 Edit the contents of the file to represent your compiler. Replace the entries between the XML elements with appropriate content.

The following table lists the XML elements in the file with a description of what the content within the element represents.

XML Element	Content Description	Content Example for GNU C Compiler
<pre><compiler_names><name> ... </name></compiler_names></pre>	<p>Name of the compiler executable. This executable transforms your .c files into object files. You can add several binary names, each in a separate <code><name>...</name></code> element. The software checks for each of the provided names and uses the compiler name for which it finds a match.</p> <p>You must not specify the linker binary inside the <code><name>...</name></code> elements.</p>	<ul style="list-style-type: none"> • gcc • gpp
<pre><include_options><opt> ... </opt></include_options></pre>	<p>The option that you use with your compiler to specify include folders.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-I</p>
<pre><system_include_options> <opt> ... </opt> </system_include_options></pre>	<p>The option that you use with your compiler to specify system headers.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-isystem</p>

XML Element	Content Description	Content Example for GNU C Compiler
<pre><preinclude_options><opt> ... </opt></preinclude_options></pre>	<p>The option that you use with your compiler to force inclusion of a file in the compiled object.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-include</p>
<pre><define_options><opt> ... </opt></define_options></pre>	<p>The option that you use with your compiler to predefine a macro.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-D</p>
<pre><undefine_options><opt> ... </opt></undefine_options></pre>	<p>The option that you use with your compiler to undo any previous definition of a macro.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-U</p>

XML Element	Content Description	Content Example for GNU C Compiler
<pre><semantic_options><opt> ... </opt></semantic_options></pre>	<p>The options that you use to modify the compiler behavior. These options specify the language settings to which the code must conform.</p> <p>You can use the <code>isPrefix</code> attribute to specify multiple options that have the same prefix and the <code>numArgs</code> attribute to specify options with multiple arguments. For instance:</p> <ul style="list-style-type: none"> • Instead of <pre><opt>-m32</opt> <opt>-m64</opt></pre> You can write <code><opt isPrefix="true">-m</opt></code>. • Instead of <pre><opt>-std=c90</opt> <opt>-std=c99</opt></pre> You can write <code><opt numArgs="1">-std</opt></code>. If your makefile uses <code>-std c90</code> instead of <code>-std=c90</code>, this notation also supports that usage. 	<ul style="list-style-type: none"> • <code>-ansi</code> • <code>-std =C90</code> • <code>-std =c++11</code> • <code>-fun signed -char</code>

XML Element	Content Description	Content Example for GNU C Compiler
<code><dialect> ... </dialect></code>	The options that specify the Polyspace dialect used by your compiler. For the complete list of dialects, on the Configuration pane, select Target & Compiler .	gnu4.7
<code><preprocess_options_list></code> <code><opt> ... </opt></code> <code></preprocess_options_list></code>	<p>The options that specify how your compiler generates a preprocessed file.</p> <p>You can use the macro <code>\$(OUTPUT_FILE)</code> if your compiler does not allow sending the preprocessed file to the standard output. Instead it defines the preprocessed file internally.</p>	<p>-E</p> <p>For an example of the <code>\$(OUTPUT_FILE)</code> macro, see the existing compiler configuration file <code>c12000.xml</code>.</p>

XML Element	Content Description	Content Example for GNU C Compiler
<pre><preprocessed_output_file> ... </preprocessed_output_file></pre>	<p>The name of file where the preprocessed output is stored.</p> <p>You can use the following macros when the name of the preprocessed output file is adapted from the source file:</p> <ul style="list-style-type: none"> • <code>\$(SOURCE_FILE)</code>: Source file name • <code>\$(SOURCE_FILE_EXT)</code>: Source file extension • <code>\$(SOURCE_FILE_NO_EXT)</code>: Source file name without extension <p>For instance, use <code>\$(SOURCE_FILE_NO_EXT).pre</code> when the preprocessor file name has the same name as the source file, but with extension <code>.pre</code>.</p>	<p>For an example of this element, see the existing compiler configuration file <code>xc8.xml</code>.</p>
<pre><src_extensions><ext> ... </ext></src_extensions></pre>	<p>The file extensions for source files.</p>	<ul style="list-style-type: none"> • <code>c</code> • <code>cpp</code> • <code>c++</code>
<pre><obj_extensions><ext> ... </ext></obj_extensions></pre>	<p>The file extensions for object files.</p>	
<pre><precompiled_header_extensions> ... </precompiled_header_extensions></pre>	<p>The file extensions for precompiled headers (if available).</p>	

XML Element	Content Description	Content Example for GNU C Compiler
<pre><polyspace_c_extra_options_list> <opt> ... </opt> </polyspace_c_extra_options_list></pre>	<p>Additional options that will be added to your project configuration</p>	<p>To avoid compilation errors due to non-ANSI[®] extension keywords, enter <i>-D keyword</i>. For more information, see “Preprocessor definitions (C/C++)”.</p>
<pre><polyspace_cpp_extra_options_list> <opt> ... </opt> </polyspace_cpp_extra_options_list></pre>	<p>Additional options that will be added to your C++ project configuration</p>	<p>To avoid compilation errors due to non-ANSI extension keywords, enter <i>-D keyword</i>. For more information, see “Preprocessor definitions (C/C++)”.</p>

- 4 After saving the edited XML file to `matlabroot\polyspace\configure\compiler_configuration\`, create a project automatically using your build command.

Tip To quickly see if your compiler configuration file works, run the automatic project setup on a sample build that does not take much time to complete. After you have set up a project successfully with your compiler configuration file, you can use this file for larger builds.

Slow Build Process When Polyspace Traces the Build

Issue

In some cases, your build process can run slower when Polyspace traces the build.

Cause

Polyspace caches information in files stored in the system temporary folder, such as `C:\Users\User_Name\AppData\Local\Temp`, in Windows. Your build can take a long time to perform read/write operations to this folder. Therefore, the overall build process is slow.

Solution

You can work around the slow build process by changing the location where Polyspace stores cache information. For instance, you can use a cache path local to the drive from which you run build tracing. To create and use a local folder `ps_cache` for storing cache information, use the advanced option `-cache-path ./ps_cache`.

- If you trace your build from the Polyspace user interface, enter this flag in the field **Add advanced configure options**. For more information, see “Create Project Automatically” on page 1-7.
- If you trace your build from the DOS, UNIX or MATLAB command line, use this flag with the `polyspace-configure` command or `polyspaceConfigure` function.

Checking if Polyspace Supports Windows Build Command

Issue

Your build command executes successfully in a Windows console application other than `cmd.exe`. However, when Polyspace traces the build, the command fails.

For instance, your build command executes successfully from the Cygwin shell. However, when Polyspace traces the build, the build command throws an error.

For more information on automatic project creation, see:

- “Create Project Automatically” on page 1-7
- “Create Project Automatically at Command Line” on page 6-2
- “Create Project Automatically from MATLAB Command Line” on page 6-10

Possible Cause

When you launch a Windows console application, your environment variables are appropriately set. Alternate Windows console applications such as the Cygwin shell can set your environment differently from `cmd.exe`.

Polyspace attempts to trace your build with the assumption that your commands run successfully in `cmd.exe`. Therefore, even if your build command runs successfully in the alternate console application, when Polyspace traces the build, the command fails.

Solution

Make sure that your build command executes successfully in the `cmd.exe` interface. For instance, before you trace a build command that executes successfully in the Cygwin shell, do one of the following:

- Launch the Cygwin shell from `cmd.exe` and then run your build command. For instance, enter the following command at the DOS command line:

```
cmd.exe /C C:\cygwin64\bin\bash.exe -c make
```

- Find the full path to your build executable and then run this executable from `cmd.exe`.

- 1 Open the Cygwin shell. Enter the following:

`which make`

The output of this command shows the full path to your executable.

- 2 Using the above output, run the executable from `cmd.exe`. For instance, enter the following command at the DOS command line:

```
cmd.exe /C path_to_executable
```

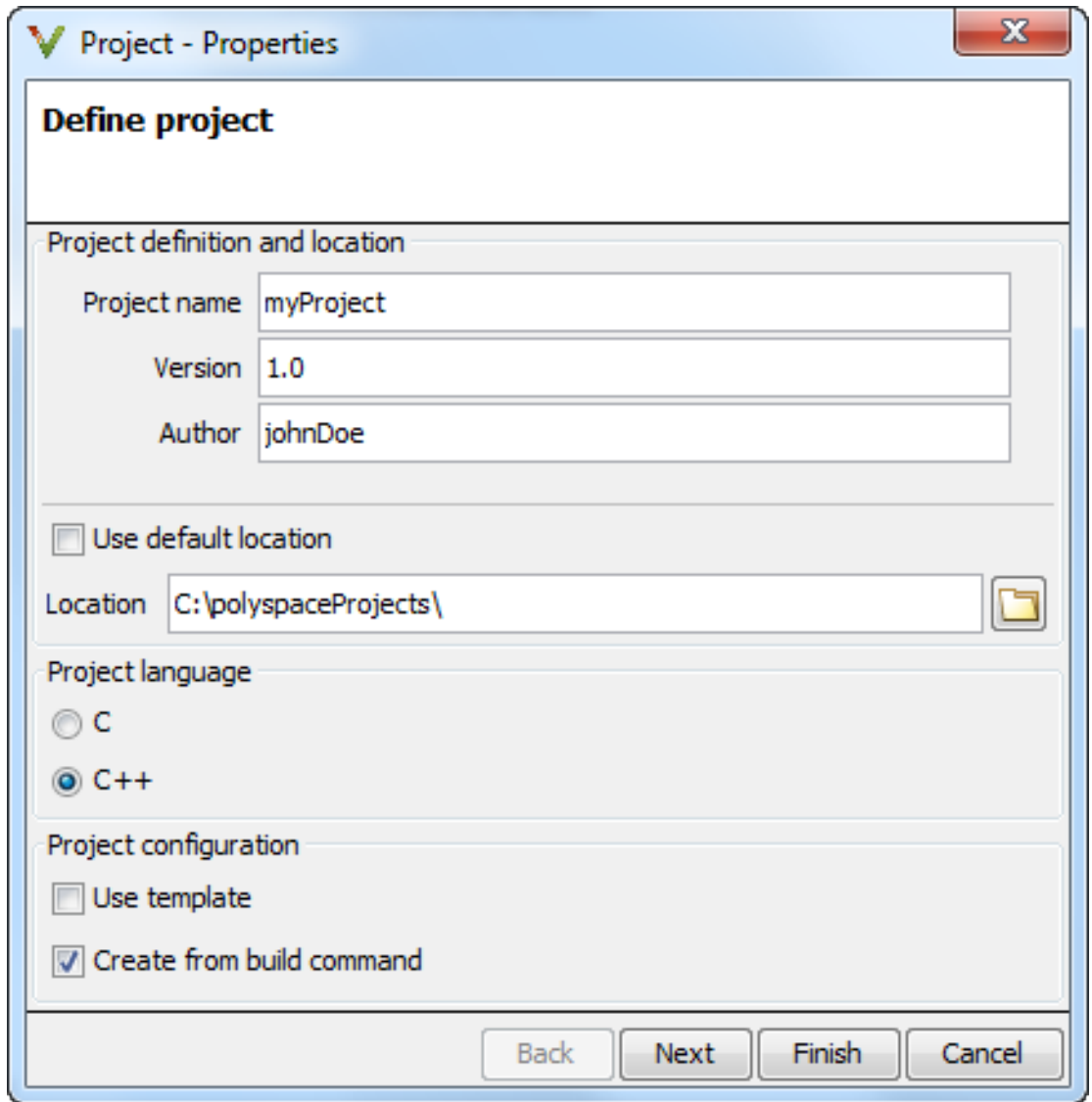
path_to_executable is the full path to the executable that you found in the previous step. For instance, `C:\cygwin64\bin\make.exe`.

If the steps do not execute successfully, Polyspace cannot trace your build.

Create Project Using Visual Studio Information

To create a Polyspace project, you can trace your Visual Studio build. For Polyspace to trace your Visual Studio build, you must install both **x86** and **x64** versions of the Visual C++ Redistributable for Visual Studio 2012 from the Microsoft website.

- 1** In the Polyspace interface, 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**.



- 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 10.0\Common7\IDE\VCEXpress.exe".

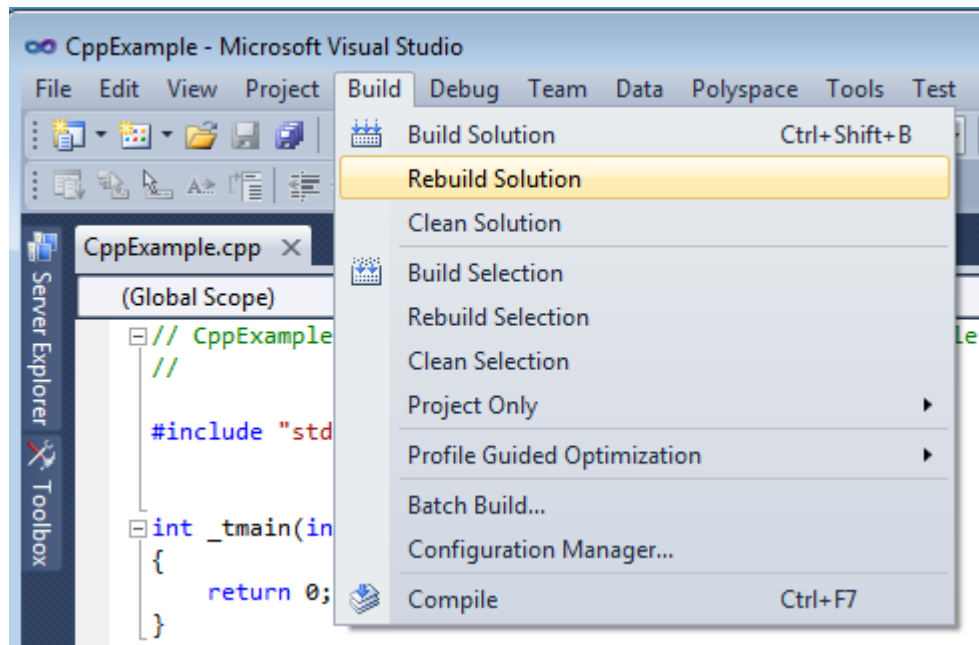
- 4 In the field **Specify working directory for running build command**, enter `C:\.`

Click .

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**.



- 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**.

More About

- “Troubleshooting Project Creation from Visual Studio Build” on page 1-27

Troubleshooting Project Creation from Visual Studio Build

In this section...

“Cannot Create Project from Visual Studio Build” on page 1-27

“Compilation Error After Creating Project from Visual Studio Build” on page 1-27

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
```

Compilation Error After Creating Project from Visual Studio Build

Issue

After you automatically set up your project from a Visual Studio 2010 build, you face compilation errors.

Possible Cause

By default, Polyspace assigns the latest dialect `visual11.0` to your project. This assignment can cause compilation errors. For more information on the **Dialect** option, see “Dialect (C++)”.

Solution

To avoid the errors, do one of the following:

- After automatic project setup:
 - 1 Open the project in the user interface. On the **Configuration** pane, select **Target & Compiler**.

- 2 Check the **Dialect**. If it is set to `visual11.0`, change it to `visual10`.

Note: If you are creating an options file from your Visual Studio 2010 build, check the `-dialect` argument. If it is set to `visual11.0`, change it to `visual10`.

- Before automatic project setup:
 - 1 Open the file `cl.xml` in `matlabroot\polyspace\configure\compiler_configuration\` where `matlabroot` is your MATLAB installation folder such as `C:\Program Files\R2015a`.
 - 2 Change the line

```
<dialect>visual11.0</dialect>
```

to

```
<dialect>visual10</dialect>
```
 - 3 Add the following lines:

```
<polyspace_cpp_extra_options_list>  
<opt>-OS-target Visual</opt>  
</polyspace_cpp_extra_options_list>
```
 - 4 Create your project or options file. The dialect is already assigned to `visual10`.

Add Source Files and Include Folders

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

In this section...

“Add Sources and Includes” on page 1-29

“Manage Include File Sequence” on page 1-29

Add Sources and Includes

- 1 In the **Project Browser**, right-click your project or the **Source** or **Include** folder in your project.
- 2 Select **Add Source**.
- 3 Add source files to your project.
 - Navigate to the location where you stored your source files. Select each source file. Click **Add Source Files**.
 - To add all files in a folder and its subfolders, select the option **Add recursively**. Select the folder. Click **Add Source Files**.
- 4 Add include folders to your project. The software adds standard include files to your project. However, you must explicitly add folders containing your custom include files.
 - Navigate to the folder containing your include files. Select the folder and click **Add Include Folders**.
 - If you do not want to add subfolders of the folder, clear **Add recursively**. Select the folder and click **Add Include Folders**.
- 5 Click **Finish**.

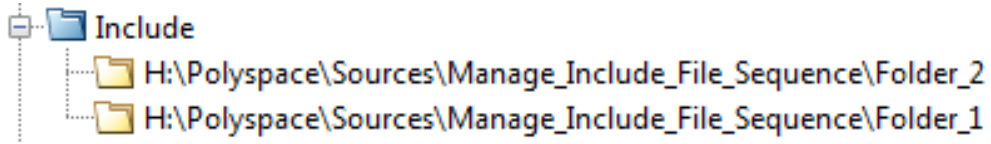
Manage Include File Sequence

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, you might want 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`.



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 4-6
- “Create New Project” on page 1-6

Specify Analysis Options

In this section...

“About Analysis Options” on page 1-31

“Specify Options in User Interface” on page 1-32

“Specify Options from DOS and UNIX Command Line” on page 1-32

“Specify Options from MATLAB Command Line” on page 1-33

About Analysis Options

You can either use the default analysis options used by the software or change them to your requirements.

At the command line or using the command-line names in the **Advanced options** pane in the user interface, you can specify analysis options multiple times. This flexibility allows you to customize pre-made configurations without having to remove options.

If you specify an option multiple times, only the last setting is used. For example, if your configuration is:

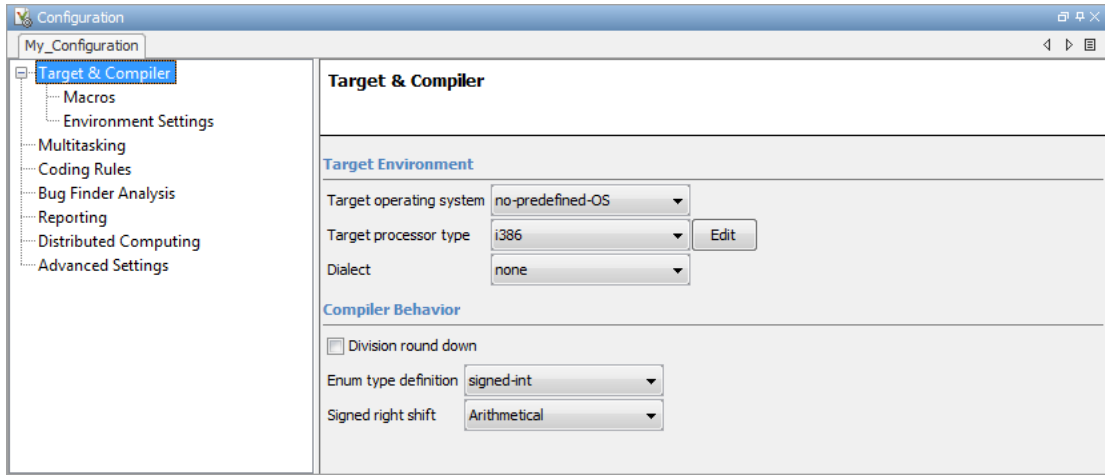
```
-lang c
-prog test_bf_cp
-verif-version 1.0
-author username
-sources-list-file sources.txt
-OS-target no-predefined-OS
-target i386
-dialect none
-misra-cpp required-rules
-target powerpc
```

Polyspace uses the last target setting, **powerpc**, and ignores the other target specified, **i386**.

Similarly, in the user interface, you can specify the target as **c18** on the Target and Compiler pane and in **Advanced options** enter **-target i386**. These two targets count as multiple analysis option specifications. Polyspace uses the target specified in the Advanced options dialog box, **i386**.

Specify Options in User Interface

To specify analysis options, use the different nodes on the **Configuration** pane.



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 1-34

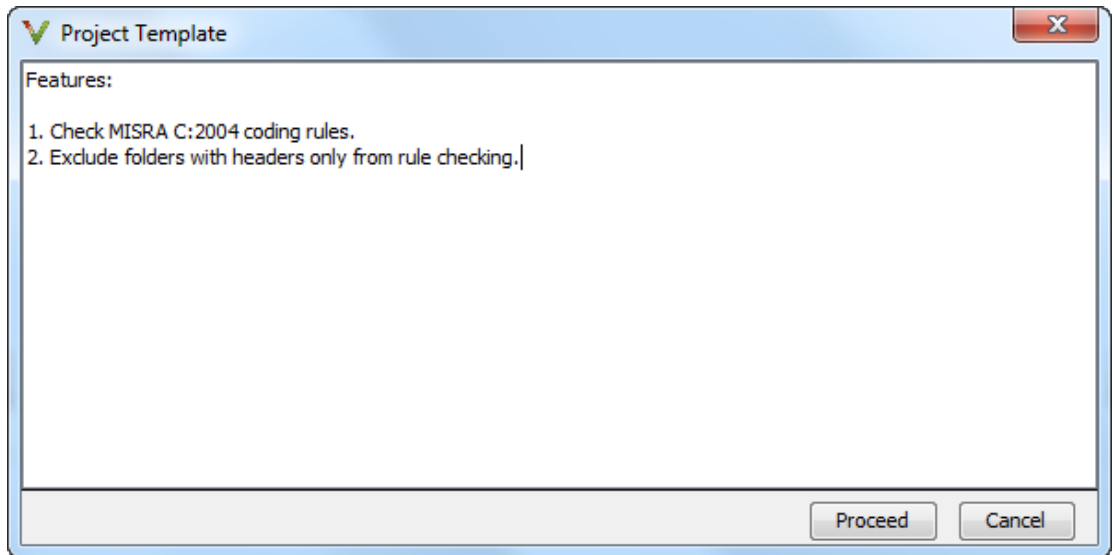
More About

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

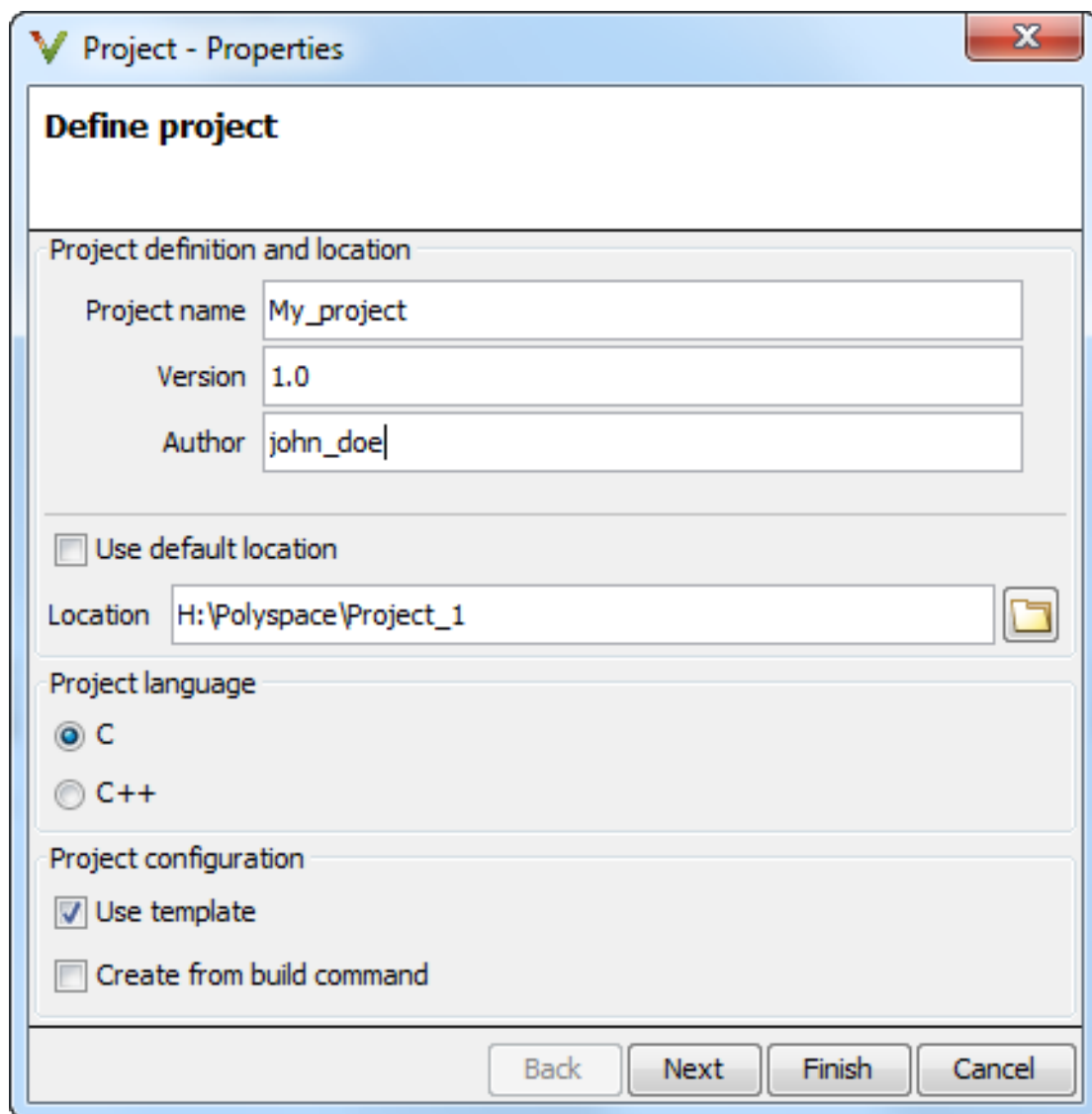
Save Analysis Options as Project Template

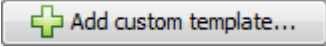
This example shows how to save your analysis options for use in other projects. Once you have configured analysis options for a project, you can save the configuration as a **Project Template**. You can use this saved configuration to automatically set up analysis options for other projects. You can also share the template with other users and enforce consistent usage of Polyspace Bug Finder in your organization.

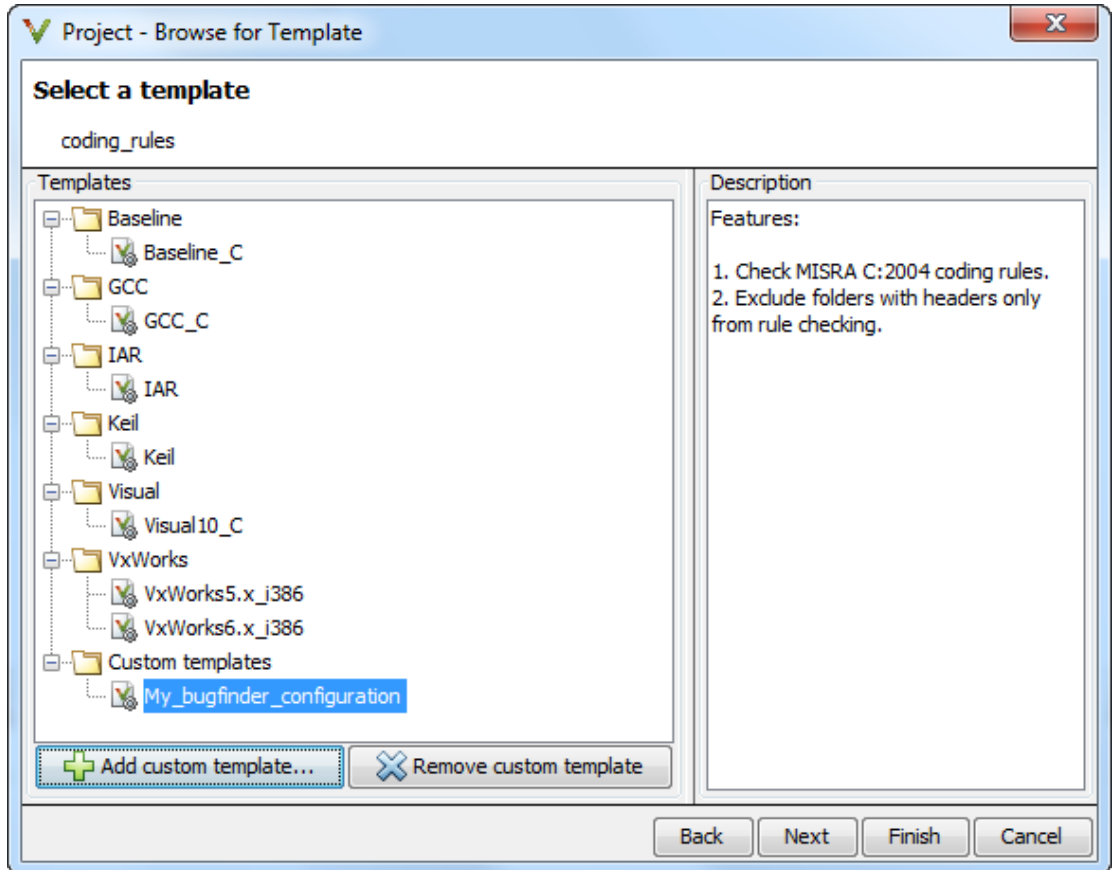
- To create a **Project Template** from an open project:
 - 1 Right-click the configuration that you want to use, and then select **Save As Template**.
 - 2 Enter a description for the template, then click **Proceed**. Save your Template file.



- When you create a new project, to use a saved template:
 - 1 Under **Project configuration**, check the **Use template** box. Click **Next**.



- 2 Select . 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.



Related Examples

- “Specify Analysis Options” on page 1-31

More About

- “Analysis Options for C”

- “Analysis Options for C++”

Organize Layout of Polyspace User Interface

The Polyspace user interface has a default set of panes. For instance, in the default layout:

- The **Project Browser** and **Results Summary** panes appear on the left.
- The **Configuration** pane appears on the top right of the user interface.

You can create and save your own layout of panes. If the current layout of the user interface does not meet your requirements, you can use a saved layout.

You can also change to the default layout of the Polyspace user interface. Select **Window > Reset Layout To > Default Layout**.

In this section...
“Create Your Own Layout” on page 1-38
“Save and Reset Layout” on page 1-39

Create Your Own Layout

To create your own layout, you can close some of the panes, open some panes that are not visible by default, and move existing panes to new locations.

To open a closed pane, select **Window > Show/Hide View > pane_name**.

To move a pane to another location:


1 Float the pane in one of three ways:

- Click and drag the blue bar on the top of the pane to float all tabs in that pane.


For instance, if **Project Browser** and **Results Summary** are tabbed on the same pane, this action floats the pane together with its tabs.

- Click and drag the tab at the bottom of the pane to float only that tab.

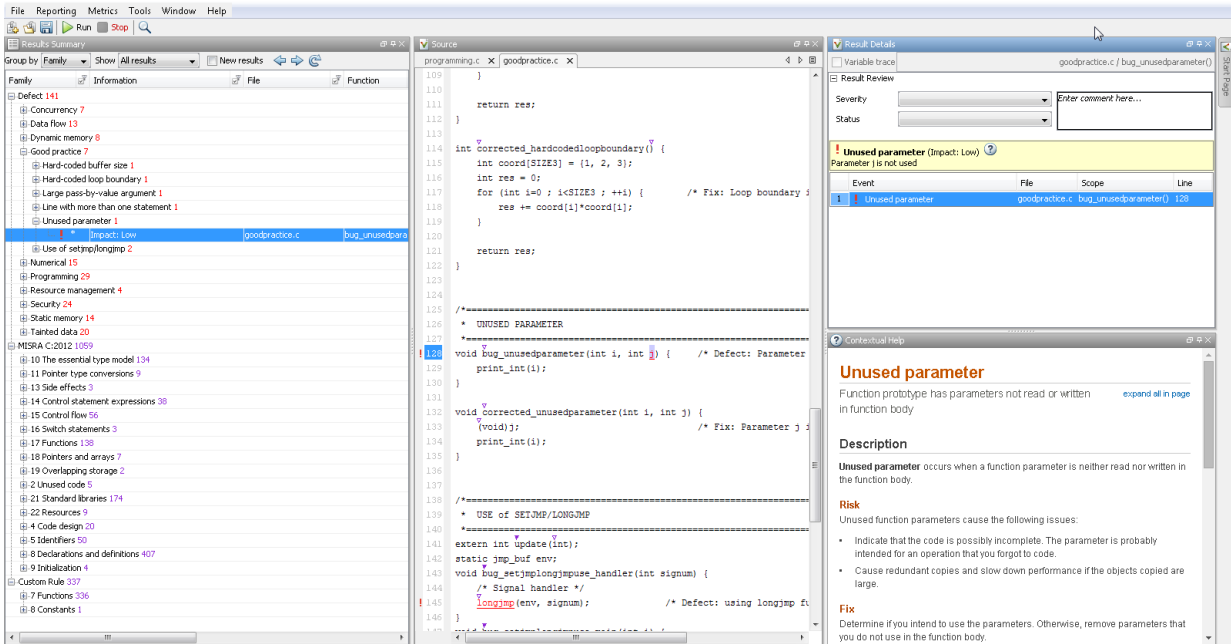
For instance, if **Project Browser** and **Results Summary** are tabbed on the same pane, dragging out **Project Browser** creates a pane with only **Project Browser** on it and floats this new pane.

- Click  on the top right of the pane to float all tabs in that pane.

2 Drag the pane to another location until it snaps into a new position.

If you want to place the pane in its original location, click  in the upper-right corner of the floating pane.

For instance, you can create a layout exclusively for reviewing results.



Save and Reset Layout

After you have created your own layout, you can save it. You can change from another layout to this saved layout.

- To save your layout, select **Window > Save Current Layout As**. Enter a name for this layout.
- To use a saved layout, select **Window > Reset Layout To > *layout_name***.
- To remove a saved layout from the **Reset Layout To** list, select **Window > Remove Custom Layout > *layout_name***.

Specify External Text Editor

This example shows how to change the default text editor for opening source files from the Polyspace interface. By default, if you open your source file from the user interface, it opens on a **Code Editor** tab. If you prefer editing your source files in an external editor, you can change this default behavior.

- 1 Select **Tools > Preferences**.
- 2 On the Polyspace Preferences dialog box, select the **Editors** tab.
- 3 From the **Text editor** drop-down list, select **External**.
- 4 In the **Text editor** field, specify the path to your text editor. For example:

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

- 5 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 Editor**, 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.

- 6 To revert back to the built-in editor, on the **Editors** tab, from the **Text editor** drop-down list, select **Built In**.

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`.

Change Default Font Size

This example shows how to change the default font size in the Polyspace user interface.

- 1 Select **Tools > Preferences**.
- 2 On the **Miscellaneous** tab:
 - To increase the font size of labels on the user interface, select a value for **GUI font size**.

For example, to increase the default size by 1 point, select +1.
 - To increase the font size of the code on the **Source** pane and the **Code Editor** pane, select a value for **Source code font size**.
- 3 Click **OK**.

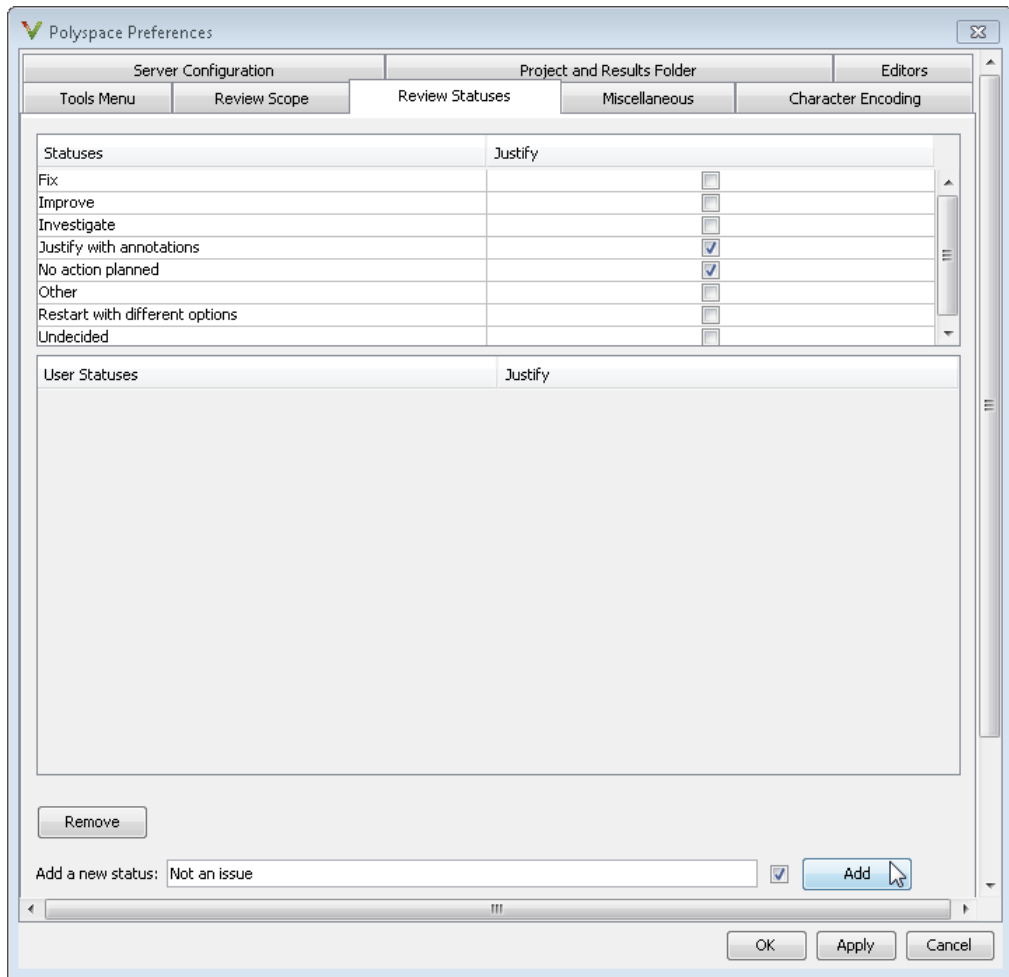
When you restart Polyspace, you see the increased font size.

Define Custom Review Status

This example shows how to customize the statuses you assign on the **Results Summary** pane.

Define Custom Status

- 1 Select **Tools > Preferences**.
- 2 Select the **Review Statuses** tab.
- 3 Enter your new status at the bottom of the dialog box, then click **Add**.



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

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

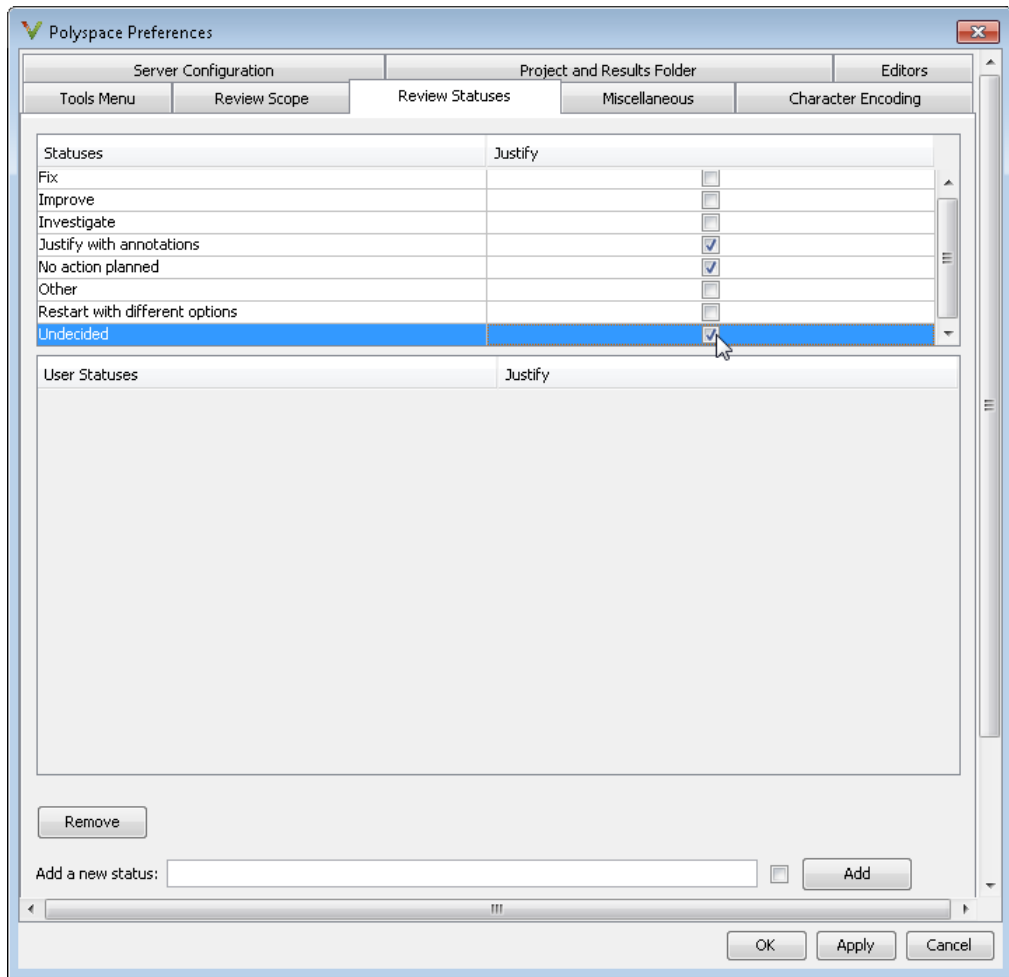
When reviewing checks, you can select the new status from the **Status** drop-down list on the **Results Summary** pane.

Add Justification to Existing Status

By default, a check is automatically justified if you assign the status, **Justified** 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 **Tools > Preferences**.
- 2** Select the **Review Statuses** tab. For the **Improve** status, select the check box in the **Justify** column. Click **OK**.



If you assign the **Improve** status to a check on the **Results Summary** pane, the check gets automatically justified.

Modeling Multitasking Code

In this section...

“Example” on page 1-47

“Limitations” on page 1-50

Polyspace Bug Finder can analyze your multitasking code for “Concurrency Defects”, such as locking and data races, if Bug Finder knows how your concurrency model is set up. In some situations, Polyspace can detect the concurrency model automatically.

If you use POSIX[®] or VxWorks[®], these concurrency primitives are supported:

- `pthread_create`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`
- `taskSpawn`
- `semTake`
- `semGive`

Polyspace uses these functions to model the thread creation, and opening and closing critical sections.

If you use a different library or different multitasking functions, you must manually model your multitasking threads by using configuration options. See “Set Up Multitasking Analysis Manually” on page 1-52.

Note: There are some aspects of multitasking that Polyspace cannot model. See “Limitations” on page 1-50.

Example

The following multitasking code models five philosophers sharing five forks.

```
#include "pthread.h"
#include <stdio.h>

pthread_mutex_t forks[4];
```

```
void* philo1(void* args) {
    while(1) {
        printf("Philosopher 1 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[0]);
        printf("Philosopher 1 takes left fork\n");
        pthread_mutex_lock(&forks[1]);
        printf("Philosopher 1 takes right fork\n");
        printf("Philosopher 1 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[1]);
        printf("Philosopher 1 puts down right fork\n");
        pthread_mutex_unlock(&forks[0]);
        printf("Philosopher 1 puts down left fork\n");
    }
    return NULL;
}

void* philo2(void* args) {
    while(1) {
        printf("Philosopher 2 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[1]);
        printf("Philosopher 2 takes left fork\n");
        pthread_mutex_lock(&forks[2]);
        printf("Philosopher 2 takes right fork\n");
        printf("Philosopher 2 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[2]);
        printf("Philosopher 2 puts down right fork\n");
        pthread_mutex_unlock(&forks[1]);
        printf("Philosopher 2 puts down left fork\n");
    }
    return NULL;
}

void* philo3(void* args) {
    while(1) {
        printf("Philosopher 3 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[2]);
        printf("Philosopher 3 takes left fork\n");
        pthread_mutex_lock(&forks[3]);
        printf("Philosopher 3 takes right fork\n");
    }
}
```

```
        printf("Philosopher 3 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[3]);
        printf("Philosopher 3 puts down right fork\n");
        pthread_mutex_unlock(&forks[2]);
        printf("Philosopher 3 puts down left fork\n");
    }
    return NULL;
}

void* philo4(void* args) {
    while(1) {
        printf("Philosopher 4 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[3]);
        printf("Philosopher 4 takes left fork\n");
        pthread_mutex_lock(&forks[4]);
        printf("Philosopher 4 takes right fork\n");
        printf("Philosopher 4 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[4]);
        printf("Philosopher 4 puts down right fork\n");
        pthread_mutex_unlock(&forks[3]);
        printf("Philosopher 4 puts down left fork\n");
    }
    return NULL;
}

void* philo5(void* args) {
    while(1) {
        printf("Philosopher 5 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[4]);
        printf("Philosopher 5 takes left fork\n");
        pthread_mutex_lock(&forks[0]);
        printf("Philosopher 5 takes right fork\n");
        printf("Philosopher 5 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[0]);
        printf("Philosopher 5 puts down right fork\n");
        pthread_mutex_unlock(&forks[4]);
        printf("Philosopher 5 puts down left fork\n");
    }
    return NULL;
}
```

```
}  
  
int main(void)  
{  
    pthread_t ph[5];  
    pthread_create(&ph[0],NULL,phil01,NULL);  
    pthread_create(&ph[1],NULL,phil02,NULL);  
    pthread_create(&ph[2],NULL,phil03,NULL);  
    pthread_create(&ph[3],NULL,phil04,NULL);  
    pthread_create(&ph[4],NULL,phil05,NULL);  
  
    pthread_join(ph[0],NULL);  
    pthread_join(ph[1],NULL);  
    pthread_join(ph[2],NULL);  
    pthread_join(ph[3],NULL);  
    pthread_join(ph[4],NULL);  
    return 1;  
}
```

Each philosopher needs two forks to eat, a right and a left fork. The functions `phil01`, `phil02`, `phil03`, `phil04`, and `phil05` represent the philosophers. Each function requires two `pthread_mutex_t` resources, representing the two forks required to eat. All five functions run at the same time in five concurrent threads.

However, a deadlock occurs in this example. When each philosopher picks up their first fork (each thread locks one `pthread_mutex_t` resource), all the forks are being used. So, the philosophers (threads) wait for their second fork (second `pthread_mutex_t` resource) to become available. However, all the forks (resources) are being held by the waiting philosophers (threads), causing a deadlock.

Without additional configuration options, Polyspace Bug Finder detects that your program performs multitasking, and that a deadlock defect occurs.

To run this example in Polyspace Bug Finder:

- 1 Copy this code into a `.c` file.
- 2 Create a Polyspace Bug Finder project with that `.c` file.
- 3 Run the analysis.

Limitations

The multitasking model that this option creates does not follow the exact semantics of POSIX or VxWorks. Polyspace cannot model:

- Thread priorities and attributes — Ignored by Polyspace.
- Recursive semaphores.
- Unbounded thread identifiers, such as `extern pthread_t ids[]` — Warning.
- Calls to concurrency primitive through high-order calls — Warning.
- Aliases on thread identifiers — Polyspace over-approximates when the alias is used.
- Termination of threads — Polyspace ignores `pthread_join`, and replaces `pthread_exit` by a standard `exit`.

See Also

“Disable automatic concurrency detection (C/C++)” | “Configure multitasking manually (C/C++)” | “Entry points (C/C++)” | “Critical section details (C/C++)” | “Temporally exclusive tasks (C/C++)” | “Find defects (C/C++)”

Related Examples

- “Review Concurrency Defects” on page 5-27
- “Set Up Multitasking Analysis Manually” on page 1-52

More About

- “Concurrency” on page 5-52

Set Up Multitasking Analysis Manually

In this section...
“Prerequisites” on page 1-52
“Set Up Multitasking Analysis in User Interface” on page 1-53
“Set Up Multitasking Analysis at Command Line” on page 1-53
“Set Up Multitasking Analysis at MATLAB Command Line” on page 1-54

This example shows how to prepare for an analysis of multitasking code. Polyspace Bug Finder can check if the protection mechanisms for your multitasking model are well designed.

Polyspace Bug Finder automatically sets up the multitasking configuration for some types of multitasking functions. For information about the supported concurrency functions, see “Modeling Multitasking Code” on page 1-47.

If your code has functions that are intended for concurrent execution, but that cannot be detected automatically, you must specify them before analysis. If these functions operate on a common variable, you must also specify protection mechanisms for those operations.

Prerequisites

For this example, save the following code in a file `multi.c`:

```
int a;

begin_critical_section();
end_critical_section();

void performTaskCycle(void) {
    begin_critical_section();
    a++;
    end_critical_section();
}

void task1(void) {
    while(1) {
        performTaskCycle();
    }
}
```

```

void task2(void) {
    while(1) {
        performTaskCycle();
    }
}

void task3() {
    a=0;
}

```

Set Up Multitasking Analysis in User Interface

- 1 Specify your entry points and protection mechanisms.
 - a On the **Configuration** pane, select the **Multitasking** node.
 - b Select **Configure multitasking manually**.
 - c For **Entry points**, specify task1, task2, and task3, each on its own line.
 - d For **Critical section details**, specify begin_critical_section as **Starting procedure** and end_critical_section as **Ending procedure**.
 - e For **Temporally exclusive tasks**, specify task1 task3 and task2 task3, each on its own line.
- 2 Specify the concurrency defects that you want Polyspace Bug Finder to detect. For more information, see “Concurrency Defects”.
 - a On the **Configuration** pane, select the **Bug Finder Analysis** node.
 - b From the **Find defects** list, select **custom**.
 - c Under the **Concurrency** node, select **Data race** and **Deadlock**.

Set Up Multitasking Analysis at Command Line

At the DOS or UNIX command prompt, specify options with the polyspace-bug-finder-nodesktop command.

```

polyspace-bug-finder-nodesktop -sources multi.c
    -entry-points task1,task2,task3
    -critical-section-begin begin_critical_section:cs1
    -critical-section-end end_critical_section:cs1
    -temporal-exclusions-file tasklist.txt

```

```
-checkers data_race,deadlock
```

Set Up Multitasking Analysis at MATLAB Command Line

At the DOS or UNIX command prompt, specify options with the `polyspaceBugFinder` function.

```
polyspaceBugFinder('-sources','multi.c',...  
  '-entry-points','task1,task2,task3',...  
  '-critical-section-begin','begin_critical_section:cs1',...  
  '-critical-section-end','end_critical_section:cs1',...  
  '-temporal-exclusions-file','tasklist.txt',...  
  '-checkers','data_race,deadlock')
```

See Also

“Disable automatic concurrency detection (C/C++)” | “Configure multitasking manually (C/C++)” | “Entry points (C/C++)” | “Critical section details (C/C++)” | “Temporally exclusive tasks (C/C++)” | “Find defects (C/C++)”

Related Examples

- “Review Concurrency Defects” on page 5-27

More About

- “Concurrency” on page 5-52
- “Modeling Multitasking Code” on page 1-47

Annotate Code for Known Defects

How to Add Annotations

You can place annotations in your code that inform Polyspace software of known or acceptable defects. Through the use of these annotations, you can:

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

You can add annotations in one of the following ways:

- When you are reviewing results in the Polyspace user interface, you can:
 - 1 Enter a **Severity**, **Status** and **Comment** for each defect on the **Results Summary** or **Results Details** pane.
 - 2 Copy the information you entered and paste it in your source code in a syntax that Polyspace can read later. For more information, see “Copy and Paste Annotations” on page 1-61.
- You can directly open your source file in a text editor and enter comments in a syntax that Polyspace can read later. For more information, see “Syntax for Code Annotations” on page 1-55.

After you have placed the annotations in your code:

- Polyspace populates the **Status**, **Severity** and **Comment** fields for the defect.
- You or another reviewer can avoid reviewing the defect. You can either ignore the known defects or filter them from the **Results Summary** pane. For more information on filtering, see “Filter and Group Results” on page 5-9.

Syntax for Code Annotations

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

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

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

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

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

... Code section ...

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

If you run Polyspace Code Prover on the code, this code annotation is ignored.

Square brackets *[]* indicate optional information.

Replace	Replace with
<i>Kind1, Kind2, ...</i>	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.
<i>Severity</i>	<ul style="list-style-type: none"> • Unset • High • Medium • Low • Not a defect
<i>Status</i>	Action for correcting the defect in your code. Possible values are: <ul style="list-style-type: none"> • Fix • Improve • Investigate • Justified • No action planned • Other
<i>Additional text</i>	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 annotations in your code that inform Polyspace software of known or acceptable coding rule violations. Through the use of these annotations, 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
-

You can add annotations in one of the following ways:

- When you are reviewing results in the Polyspace user interface, you can:
 - 1 Enter a **Severity**, **Status** and **Comment** for each coding-rule violation on the **Results Summary** pane.
 - 2 Copy the information you entered and paste it in your source code in a syntax that Polyspace can read later. For more information, see “Copy and Paste Annotations” on page 1-61.
- You can directly open your source file in a text editor and enter comments in a syntax that Polyspace can read later. For more information, see “Syntax for Code Annotations” on page 1-59.

After you have placed the annotations in your code:

- Polyspace populates the **Status**, **Severity** and **Comment** fields for the coding-rule violation.
- You or another reviewer can avoid reviewing the rule violation. You can either ignore the known rule violations or filter them from the **Results Summary** pane. For

more information on filtering, see “Filter and Group Coding Rule Violations” on page 3-18.

Syntax for Code Annotations

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

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

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

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

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

... Code section ...

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

Square brackets `[]` indicate optional information.

Replace	Replace with
<i>Rule_Set</i>	<ul style="list-style-type: none"> • MISRA-C • MISRA-AC-AGC • MISRA-CPP • JSF • Custom <p>If you want the comment to apply to all coding rule violations on the following line, specify ALL.</p>
<i>Rule1,Rule2,...</i>	<p>Rule number. For more information, see:</p> <ul style="list-style-type: none"> • “MISRA C:2004 and MISRA AC AGC Coding Rules” • “MISRA C++ Coding Rules” • “JSF C++ Coding Rules” • “Custom Coding Rules”

Replace	Replace with
<i>Severity</i>	<ul style="list-style-type: none">• Unset• High• Medium• Low• Not a defect
<i>Status</i>	Action for correcting the coding rule violation. Possible values are: <ul style="list-style-type: none">• Fix• Improve• Investigate• Justified• No action planned• Other
<i>Additional text</i>	Additional comments.

Syntax Examples:

MISRA C rule violation:

```
polyspace<MISRA-C:6.3 : Low : Justified> Known issue
```

JSF® rule violation:

```
polyspace<JSF:9 : Low : Justified> Known issue
```

Copy and Paste Annotations

This example shows how to place annotations in your code to mark defects that you are already aware of but do not intend to fix immediately. Using your comments, Polyspace populates the defect **Severity**, **Status** and **Comment** fields on the **Results Summary** pane. After you have placed your comments in your code, you or another reviewer can avoid reviewing the same defect twice.

- 1** On the **Results Summary** or **Result Details** pane, assign a **Severity**, **Status** and **Comment** to a result.
 - a** Select the result.
 - b** Select options from the **Severity** and **Status** dropdown lists.
 - c** In the **Comment** field, enter a comment that helps you recognize the result easily.
- 2** Copy the **Severity**, **Status** and **Comment**.
 - a** On the **Results Summary** pane, right-click the defect or coding rule violation.
 - b** Select **Add Pre-Justification to Clipboard**. The software copies the justification string to the clipboard.
- 3** Paste the **Severity**, **Status** and **Comment** in your source code.
 - a** On the **Results Summary** pane, right-click the defect or coding rule violation and select **Open Editor**.

Your source file opens on the **Code Editor** pane or an external text editor depending on your **Preferences**. The current line is the line containing the defect.

- b** Using the paste option in the text editor, paste the justification template string on the line immediately before the line containing the defect or coding rule violation.

You can see your **Severity**, **Status** and **Comment** as a code comment in a format that Polyspace can read later.

```
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);
```

- c** Save your source file.
- 4** Run the analysis again. Open your results.

On the **Results Summary** pane, the software populates the **Severity**, **Status** and **Comment** fields for the defect or rule violation. You can either ignore these findings, or filter them from the **Results Summary** pane. For more information on filtering, see “Filter and Group Results” on page 5-9.

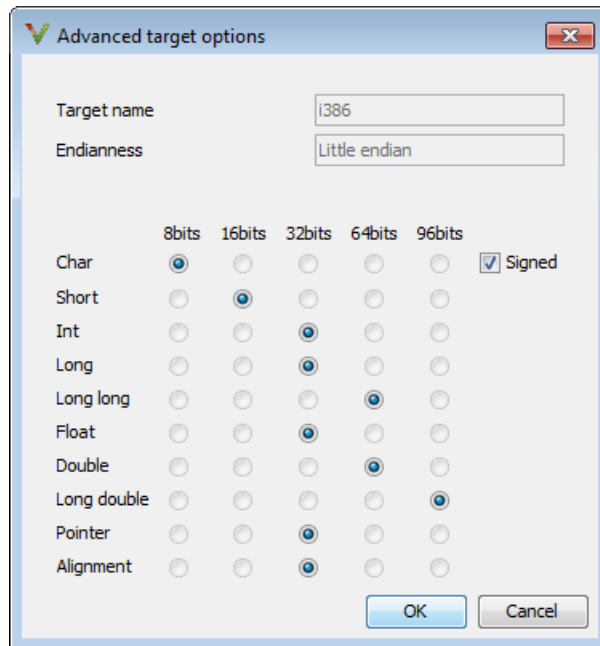
Modify Predefined Target Processor Attributes

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. The settings that you can modify for each target are shown in [brackets] in the target processor settings. See “Target processor type (C/C++)”.

To modify target processor attributes:

- 1 On the **Configuration** pane, select the **Target & Compiler** node.
- 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.



- 4 Modify the attributes as required.

For information on each target option, see “Generic target options (C/C++)”.

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

Specify Generic Target Processors

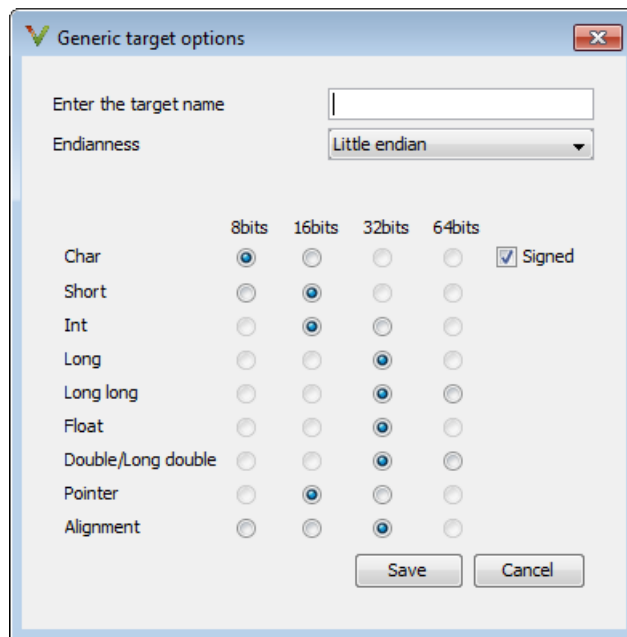
Define Generic Target

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 On the **Configuration** pane, select the **Target & Compiler** node.
- 2 From the **Target processor type** drop-down list, select `mcpu...` (Advanced).

The Generic target options dialog box opens.



- 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 (C/C++)”.

- 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
alignmen	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
alignmen	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
alignmen	8	16	16	16	16	16	16	16	16	N/A	N/A

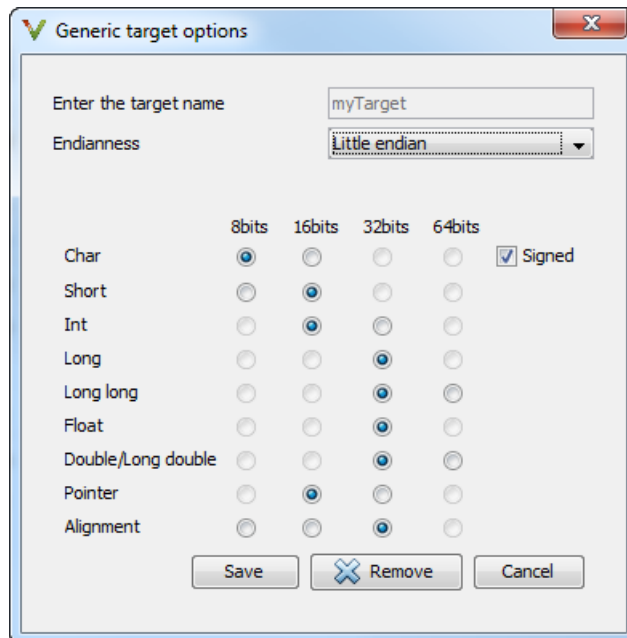
Hitachi H8/300H, H8S, H8C, H8/Tiny

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
alignmen	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 On the **Configuration** pane, select the **Target & Compiler** node.
- 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.

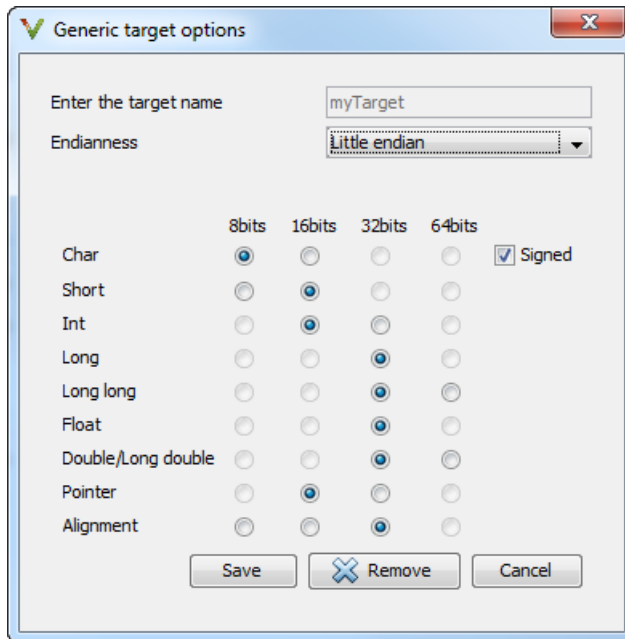


- 4 If required, specify new attributes for your target. Then click **Save**.
- 5 Otherwise, click **Cancel**.

Delete Generic Target

To delete a generic target:

- 1 On the **Configuration** pane, select the **Target & Compiler** node.
- 2 From the **Target processor type** drop-down list, select the target that you want to remove, for example, myTarget.



- 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.

In this section...

“My Target Application Runs on Solaris” on page 1-69

“My Target Application Runs on Vxworks” on page 1-69

“My Target Application Does Not Run on Linux, VxWorks, or Solaris” on page 1-70

This section describes the configuration options required to compile and analyze code designed to run on specific operating systems. Use the **Target operating system** analysis option to add certain predefined compilation flags required for Linux, Windows, or Solaris™.

My Target Application Runs on Solaris

If Polyspace software runs on a Linux machine:

User interface:

- **Target operating system** > **Solaris**
- In your project, include your Solaris include folder.

Command-line:

```
polyspace-code-prover-nodesktop \  
  -OS-target Solaris \  
  -I /your_path_to_solaris_includes
```

My Target Application Runs on Vxworks

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

User interface:

- **Target operating system** > **VxWorks**
- In your project, include your VxWorks include folder.

Command-line:

```
polyspace-code-prover-nodesktop \  
  -OS-target vxworks \  
  -I /your_path_to_VxWorks_includes
```

My Target Application Does Not Run on Linux, VxWorks, or Solaris

If your project uses target-specific routines or code, use the following options:

User interface:

- **Target operating system > no-predefined-OS**
- In your project, include your target include folders.

Command-line:

```
polyspace-code-prover-nodesktop \  
  -OS-target no-predefined-OS \  
  -I /your_path_to_target_includes
```


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 size¹
- 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.

1. 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 “Command/script to apply to preprocessed files (C/C++)”.

Content of `myTpl.pl` file

```
#!/usr/bin/perl

#####
# Post Processing template script
#
#####
# Usage from Polyspace UI:
#
# 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 occurrence 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 `sfr` and `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** 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 <code>bit</code>	<ul style="list-style-type: none"> An expression to type <code>bit</code> gives values in range [0,1]. 	<pre>bit x = 0, y = 1, z = 2; assert(x == 0); assert(y == 1); assert(z == 1);</pre>	pointers to bits and arrays of bits are not allowed

Type/Language	Description	Example	Restrictions
	<ul style="list-style-type: none"> Converting an expression in the type, gives 1 if it is not equal to 0, else 0. This behavior is similar to <code>c++ booltype</code>. 	<pre>assert(sizeof(bit) == sizeof(int));</pre>	
Type <code>sfr</code>	<ul style="list-style-type: none"> The <code>-sfr-types</code> 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. 	<pre>sfr x = 0xf0; // declaration of variable x at address 0xF0 sfr16 y = 0x4EEF;</pre> <p>For this example, options need to be:</p> <pre>-dialect keil -sfr-types sfr=8, sfr16=16</pre>	<p><code>sfr</code> and <code>sbit</code> types are only allowed in declarations of external global variables.</p>
Type <code>sbit</code>	<ul style="list-style-type: none"> Each read/write access of a variable is replaced by an access of the corresponding <code>sfr</code> variable access. Only external global variables can be mapped with a <code>sbit</code> variable. Allowed expressions are integer variables, cells of array of <code>int</code> and <code>struct/union</code> integral fields. a variable can also be declared as <code>extern bit</code> in an another file. 	<pre>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;</pre> <pre>/* file1.c */ sbit x = P0 ^ 1; /* file2.c */ extern bit x; x = 1; // set the 1st bit of P0 to 1</pre>	

Type/Language	Description	Example	Restrictions
Absolute variable location	Allowed constants are integers, strings and identifiers.	<pre>int var _at_ 0xF0 int x @ 0xFE ; static const int y @ 0xA0 = 3;</pre>	Absolute variable locations are ignored (even if declared with a <code>#pragma location</code>).
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>"	<pre>void foo1 (void) interrupt XX = YY using 99 {...} void foo2 (void) _ task_ 99 _priority_ 2 {...}</pre>	Entry points and interrupts are not taken into account as -entry-points.
Keywords ignored	alien, bdata, far, idata, epdata, huge, sdata, small, compact, large, reentrant. Defining <code>-D __C51__</code> , 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	<ul style="list-style-type: none"> 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 <code>c++ bool</code> type. If initialized with values 0 or 1, a variable of type bit is a simple variable (like a <code>c++ bool</code>). A variable of type bit is a register bit 	<pre>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 };</pre>	pointers to bits and arrays of bits are not allowed

Type/Language	Description	Example	Restrictions
	variable (mapped with a bit or a sfr type)		
Type sfr	<ul style="list-style-type: none"> The <code>-sfr-types</code> 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. 	<pre>sfr x = 0xf0; // declaration of variable x at address 0xF0</pre>	sfr and sbit types are only allowed in declarations of external global variables.
Individual bit access	<ul style="list-style-type: none"> 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. 	<pre>int x[3], y; x[2].2 = x[0].3 + y.1;</pre>	
Absolute variable location	Allowed constants are integers, strings and identifiers.	<pre>int var @ 0xF0; int xx @ 0xFE ; static const int y \ @0xA0 = 3;</pre>	Absolute variable locations are ignored (even if declared with a <code>#pragma location</code>).
Interrupt functions	<ul style="list-style-type: none"> A warning is displayed in the log file when an interrupt function has been found: "interrupt handler detected : funcname" 	<pre>interrupt [1] \ using [99] void \ foo1(void) { ... }; monitor [3] void \ foo2(void) { ... };</pre>	Entry points and interrupts are not taken into account as <code>-entry-points</code> .

Type/Language	Description	Example	Restrictions
	<ul style="list-style-type: none"> A monitor function is a function that disables interrupts while it is executing, and then restores the previous interrupt state at function exit. 		
Keywords ignored	saddr, reentrant, reentrant_idata, non_banked, plm, bdata, idata, pdata, code, data, xdata, xhuge, interrupt, __interrupt and __intrinsic		
Unnamed struct/ union	<ul style="list-style-type: none"> Fields of unions/ structs without a tag or a name can be accessed without naming their parent struct. On a conflict between a field of an anonymous struct with other identifiers: <ul style="list-style-type: none"> 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 	<pre>union { int x; }; union { int y; struct { int z; }; } @ 0xF0;</pre>	

Type/Language	Description	Example	Restrictions
	displayed in the log file.		
no_init attribute	<ul style="list-style-type: none"> a global variable declared with this attribute is handled like an external variable. It is handled like a type qualifier. 	<pre>no_init int x; no_init union { int y; } @ 0xFE;</pre>	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.

Supported C++ 2011 Extensions

The following table list which C++ 2011 standards Polyspace can analyze. If your code contains non-supported constructions, Polyspace reports a compilation error.

Standard	Description	Supported
C++2011-N2118	Rvalue references	Yes
C++2011-N2439	Rvalue references for <code>*this</code>	Yes
C++2011-N1610	Initialization of class objects by rvalues	Yes
C++2011-N2756	Non-static data member initializers	Yes
C++2011-N2242	Variadic templates	Yes
C++2011-N2555	Extending variadic template template parameters	Yes
C++2011-N2672	Initializer lists	Yes
C++2011-N1720	Static assertions	Yes
C++2011-N1984	auto-typed variables	Yes
C++2011-N1737	Multi-declarator auto	Yes
C++2011-N2546	Removal of auto as a storage-class specifier	Yes
C++2011-N2541	New function declarator syntax	Yes
C++2011-N2927	New wording for C++0x lambdas	Yes
C++2011-N2343	Declared type of an expression	Yes

Standard	Description	Supported
C++2011-N3276	decltype and call expressions	Yes
C++2011-N1757	Right angle brackets	Yes
C++2011-DR226	Default template arguments for function templates	Yes
C++2011-DR339	Solving the SFINAE problem for expressions	Yes
C++2011-N2258	Template aliases	Yes
C++2011-N1987	Extern templates	Yes
C++2011-N2431	Null pointer constant	Yes
C++2011-N2347	Strongly-typed enums	Yes
C++2011-N2764	Forward declarations for enums	Yes
C++2011-N2761	Generalized attributes	Yes
C++2011-N2235	Generalized constant expressions	Yes
C++2011-N2341	Alignment support	Yes
C++2011-N1986	Delegating constructors	Yes
C++2011-N2540	Inheriting constructors	Yes
C++2011-N2437	Explicit conversion operators	Yes
C++2011-N2249	New character types	Yes

Standard	Description	Supported
C++2011-N2442	Unicode string literals	Yes
C++2011-N2442	Raw string literals	Yes
C++2011-N2170	Universal character name literals	No
C++2011-N2765	User-defined literals	Yes
C++2011-N2342	Standard Layout Types	No
C++2011-N2346	Defaulted and deleted functions	Yes
C++2011-N1791	Extended friend declarations	Yes
C++2011-N2253	Extending sizeof	Yes
C++2011-N2535	Inline namespaces	Yes
C++2011-N2544	Unrestricted unions	Yes
C++2011-N2657	Local and unnamed types as template arguments	Yes
C++2011-N2930	Range-based for	Yes
C++2011-N2928	Explicit virtual overrides	Yes
C++2011-N3050	Allowing move constructors to throw [noexcept]	Yes
C++2011-N3053	Defining move special member functions	Yes
C++2011-N2239	Concurrency - Sequence points	No

Standard	Description	Supported
C++2011-N2427	Concurrency - Atomic operations	No
C++2011-N2748	Concurrency - Strong Compare and Exchange	No
C++2011-N2752	Concurrency - Bidirectional Fences	No
C++2011-N2429	Concurrency - Memory model	No
C++2011-N2664	Concurrency - Data-dependency ordering: atomics and memory model	No
C++2011-N2179	Concurrency - Propagating exceptions	No
C++2011-N2440	Concurrency - Abandoning a process and <code>at_quick_exit</code>	Yes
C++2011-N2547	Concurrency - Allow atomics use in signal handlers	No
C++2011-N2659	Concurrency - Thread-local storage	No
C++2011-N2660	Concurrency - Dynamic initialization and destruction with concurrency	No
C++2011-N2340	<code>__func__</code> predefined identifier	Yes
C++2011-N1653	C99 preprocessor	Yes
C++2011-N1811	<code>long long</code>	Yes
C++2011-N1988	Extended integral types	No

See Also

“C++11 Extensions (C++)”

Gather Compilation Options Efficiently

The code is often tuned for the target (as discussed in “Analyze Keil or IAR Dialects” on page 1-75). 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 Constraints

This example shows how to specify constraints on variables in your code. Polyspace uses the code that you provide to make assumptions about variable ranges, allowed buffer size for pointers, and other items. However, sometimes the assumptions are broader than what you expect because:

- You have not provided the complete code. For example, you have not provided some of the function definitions.
- Some of the information about variables is available only at run-time. For example, some variables in your code obtain values from the user at run time.

Because of these broad assumptions, Polyspace can sometimes produce false positives.

To reduce the number of such false positives, you can specify additional constraints on global variables, function inputs and return values of stubbed functions. After you specify your constraints, you can save them as an XML file and use them for subsequent verifications. If your source code changes, you can update the previous constraints. You do not have to create a new constraint template from scratch.

In this section...

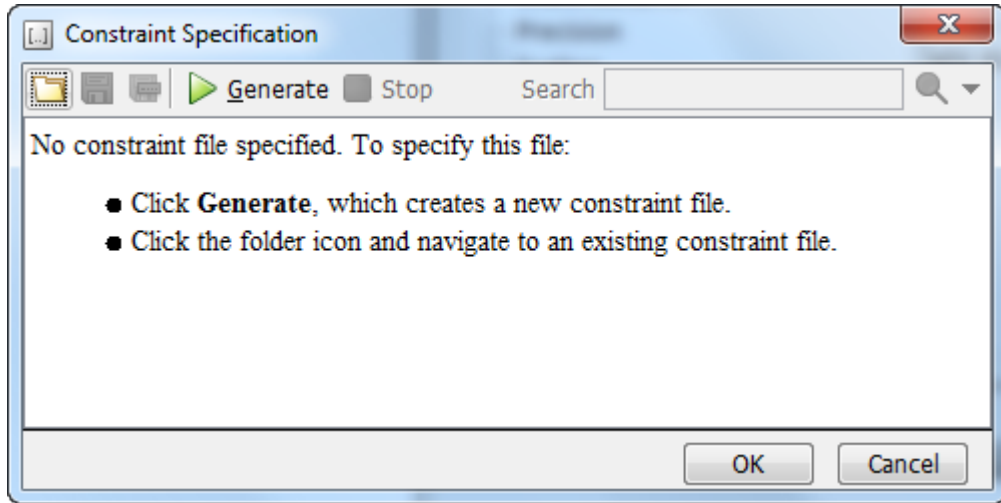
“Create Constraint Template” on page 1-87

“Update Existing Template” on page 1-89

Create Constraint Template

- 1 On the **Configuration** pane, select **Inputs & Stubbing**.
- 2 To the right of **Constraint setup**, click the **Edit** button.

The Constraint Specification dialog box opens.



- 3 Click  **Generate**. The software compiles your project and creates a template.

The template contains a list of all variables on which you can provide constraints.

- 4 Specify your constraints and save the template as an XML file. For more information, see “Constraints” on page 1-90.
- 5 Click **OK**.

You see the full path to the template XML file in the **Constraint setup** field. If you run a verification, Polyspace uses this template for extracting variable constraints.

Note: Specifying constraints outside your code in this way allows more precise verification. However, because the constraints are outside your code, you must use the code within the specified constraints. Otherwise, the verification results might not apply. For example, if you use function inputs outside your specified range, a run-time error can occur on an operation even though checks on the operation are green.

To specify constraints in your code, you can use:


- Appropriate error handling tests in your code.

Polyspace checks if the errors can actually occur. If they do not occur, the test blocks appear as **Unreachable code**.

- The `assert` macro. For example, to constrain a variable `var` in the range `[0,10]`, you can use `assert(var >= 0 && var <=10);`.

Polyspace checks your `assert` statements to see if the condition can be false. Following the `assert` statement, Polyspace considers that the `assert` condition is true. Therefore, if you use appropriate `assert` statements, for the remaining code in the same scope, your variables are constrained. For examples, see [User assertion](#).

Update Existing Template

- 1 On the **Configuration** pane, select **Inputs & Stubbing**.
- 2 Open the existing template in one of the following ways:
 - Enter the path to the template XML file in the **Constraint setup** field. Click **Edit**.
 - Click **Edit**. In the Constraint Specification dialog box, click the  icon, to navigate to your template file.
- 3 Click **Update**.
 - a Variables that are no longer present in your source code appear under the **Non Applicable** node. To remove an entry under the **Non Applicable** node or the node itself, right-click and select **Remove This Node**.
 - b Specify your new constraints for any of the other variables.

See Also

“Constraint setup (C/C++)”

Related Examples

- “Constrain Global Variable Range”

Constraints

The Polyspace DRS Configuration interface allows you to specify constraints for:

- Global Variables.
- User-defined Functions.
- Stubbed Functions.

For more information, see “Specify Constraints” on page 1-87.

The following table lists the constraints that can be specified through this interface.

Column	Settings
Name	<p>Displays the list of variables and functions in your Project for which you can specify data ranges.</p> <p>This Column displays three expandable menu items:</p> <ul style="list-style-type: none"> • Globals – Displays global variables in the project. • User defined functions – Displays user-defined functions in the project. Expand a function name to see its inputs. • Stubbed functions – Displays a list of stub functions in the project. Expand a function name to see the inputs and return values.
File	Displays the name of the source file containing the variable or function.
Attributes	<p>Displays information about the variable or function.</p> <p>For example, static variables display static.</p>
Data Type	Displays the variable type.
Main Generator Called	<p>Applicable only for user-defined functions.</p> <p>Specifies whether the main generator calls the function:</p> <ul style="list-style-type: none"> • MAIN GENERATOR – Main generator may call this function, depending on the value of the <code>-functions-called-in-loop (C)</code> or <code>-main-generator-calls (C++)</code> parameter. • NO – Main generator will not call this function.

Column	Settings
	<ul style="list-style-type: none"> • YES – Main generator will call this function.
Init Mode	<p>Specifies how the software assigns a range to the variable:</p> <ul style="list-style-type: none"> • MAIN GENERATOR – Variable range is assigned depending on the settings of the main generator options <code>-variables-written-before-loop</code> and <code>-no-def-init-glob</code>. (For C++, the options are <code>-main-generator-writes-variables</code>, and <code>-no-def-init-glob</code>.) • 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 <code>globalassert</code> mode if you need a warning. <p>User-defined functions support only INIT mode.</p> <p>Stub functions support only PERMANENT mode.</p> <p>For C verifications, global pointers support MAIN GENERATOR, IGNORE, or INIT mode.</p> <ul style="list-style-type: none"> • MAIN GENERATOR – Pointer follows the options of the main generator. • IGNORE – Pointer is not initialized • INIT – Specify if the pointer is <code>NULL</code>, and how the pointed object is allocated (Initialize Pointer and Init Allocated options).

Column	Settings
<p>Init Range</p>	<p>Specifies the minimum and maximum values for the variable.</p> <p>You can use the keywords <code>min</code> and <code>max</code> to denote the minimum and maximum values of the variable type. For example, for the type <code>long</code>, <code>min</code> and <code>max</code> correspond to -2^{31} and $2^{31}-1$ respectively.</p> <p>You can also use hexadecimal values. For example: <code>0x12..0x100</code></p> <p>For <code>enum</code> variables, you cannot specify ranges directly using the enumerator constants. Instead use the values represented by the constants.</p> <p>For <code>enum</code> variables, you can also use the keywords <code>enum_min</code> and <code>enum_max</code> to denote the minimum and maximum values that the variable can take. For example, for an <code>enum</code> variable of the type defined below, <code>enum_min</code> is 0 and <code>enum_max</code> is 5:</p> <pre>enum week{ sunday, monday=0, tuesday, wednesday, thursday, friday, saturday};</pre>
<p>Initialize Pointer</p>	<p>Applicable only to pointers. Enabled only when you specify Init Mode:INIT.</p> <p>Specifies whether the pointer should be NULL:</p> <ul style="list-style-type: none"> • 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. <hr/> <p>Note: Not applicable for C++ projects.</p>

Column	Settings
Init Allocated	<p>Applicable only to pointers. Enabled only when you specify Init Mode:INIT.</p> <p>Specifies how the pointed object is allocated:</p> <ul style="list-style-type: none"> • 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. <p>See .</p> <hr/> <p>Note: Not applicable for C++ projects.</p>
# Allocated Objects	<p>Applicable only to pointers.</p> <p>Specifies how many objects are pointed to by the pointer (the pointed object is considered as an array).</p> <p>Note: The Init Allocated parameter specifies how many allocated objects are actually initialized. See .</p> <hr/> <p>Note: Not applicable for C++ projects.</p>
Global Assert	<p>Specifies whether to perform an assert check on the variable at global initialization, and after each assignment.</p>
Global Assert Range	<p>Specifies the minimum and maximum values for the range you want to check.</p>
Comment	<p>Remarks that you enter, for example, justification for your DRS values.</p>

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 and *\$Release* is the release number.

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 2-2
- “Polyspace MISRA C 2004 and MISRA AC AGC Checkers” on page 2-4
- “Software Quality Objective Subsets (C:2004)” on page 2-5
- “Software Quality Objective Subsets (AC AGC)” on page 2-10
- “MISRA C:2004 and MISRA AC AGC Coding Rules” on page 2-14
- “Polyspace MISRA C:2012 Checker” on page 2-53
- “Software Quality Objective Subsets (C:2012)” on page 2-54
- “Unsupported MISRA C:2012 Guidelines” on page 2-59
- “Polyspace MISRA C++ Checker” on page 2-60
- “Software Quality Objective Subsets (C++)” on page 2-61
- “MISRA C++ Coding Rules” on page 2-68
- “Polyspace JSF C++ Checker” on page 2-95
- “JSF C++ Coding Rules” on page 2-96

Rule Checking

Polyspace Coding Rule Checker

Polyspace software allows you to analyze code to demonstrate compliance with established C and C++ coding standards:

- MISRA C 2004
- MISRA C 2012
- MISRA® C++:2008
- 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 which rules to enforce. Polyspace software performs rule checking before and during the analysis. Violations appear in the **Results Summary** pane.

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: When you enable the Compilation Assistant *and* coding rules checking, the software does not report coding rule violations if there are compilation errors.

Differences Between Bug Finder and Code Prover

Coding rule checker results can differ between Polyspace Bug Finder and Polyspace Code Prover. The rule checking engines are identical in Bug Finder and Code Prover, but the context in which the checkers execute is not the same. If a project is launched from Bug Finder and Code Prover with the same source files and same configuration options, the coding rule results can differ. For example, the main generator used in Code Prover activates global variables, which causes the rule checkers to identify such global

variables as initialized. The Bug Finder does not have a main generator, so handles the initialization of the global variables differently. Another difference is how violations are reported. The coding rules violations found in header files are not reported to the user in Bug Finder, but these violations are visible in Code Prover.

This difference can occur in MISRA C:2004, MISRA C:2012, MISRA C++, and JSF++. See the **Polyspace Specification** column or the **Description** for each rule.

Even though there are differences between rules checkers in Bug Finder and Code Prover, both reports are valid in their own context. For quick coding rules checking, use Polyspace Bug Finder.

Polyspace MISRA C 2004 and MISRA AC AGC Checkers

The Polyspace MISRA C:2004 checker helps you comply with the MISRA C 2004 coding standard.²

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:2004)” on page 2-5
- “Software Quality Objective Subsets (AC AGC)” on page 2-10

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

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

Software Quality Objective Subsets (C:2004)

In this section...

“Rules in SQO-Subset1” on page 2-5

“Rules in SQO-Subset2” on page 2-6

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
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.
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
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.
11.3	A cast should not be performed between a pointer type and an integral type.
12.12	The underlying bit representations of floating-point values shall not be used.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type.
13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control.
14.4	The <i>goto</i> statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.

Rule number	Description
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
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.
17.3	>, >=, <, <= 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.
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.3	An area of memory shall not be reused for unrelated purposes.
18.4	Unions shall not be used.
20.4	Dynamic heap memory allocation shall not be used.

Note: Polyspace software does not check MISRA rule 18.3.

Rules in SQ0-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. The `SQ0-subset2` option checks the rules in `SQ0-subset1` and some additional rules.

Rule number	Description
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.
6.3	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types
8.7	Objects shall be defined at block scope if they are only accessed from within a single function

Rule number	Description
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
9.2	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
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
10.5	Bitwise operations shall not be performed on signed integer types
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type
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.
11.3	A cast should not be performed between a pointer type and an integral type.
11.5	Type casting from any type to or from pointers shall not be used
12.1	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
12.5	The operands of a logical && or shall be primary-expressions
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 !)
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned
12.10	The comma operator shall not be used

Rule number	Description
12.12	The underlying bit representations of floating-point values shall not be used.
13.1	Assignment operators shall not be used in expressions that yield Boolean values
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type.
13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control.
13.6	Numeric variables being used within a “ <i>for</i> ” loop for iteration counting should not be modified in the body of the loop
14.4	The <i>goto</i> statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
14.8	The statement forming the body of a <i>switch</i> , <i>while</i> , <i>do while</i> or <i>for</i> statement shall be a compound statement
14.10	All <i>if else if</i> constructs should contain a final <i>else</i> clause
15.3	The final clause of a <i>switch</i> statement shall be the <i>default</i> clause
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration
16.7	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.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement 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

Rule number	Description
17.3	>, >=, <, <= 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.
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.3	An area of memory shall not be reused for unrelated purposes.
18.4	Unions shall not be used.
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
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives
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 ##
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.
20.3	The validity of values passed to library functions shall be checked.
20.4	Dynamic heap memory allocation shall not be used.

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 2-10
“Rules in SQO-Subset2” on page 2-11

Rules in SQO-Subset 1

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

Rule number	Description
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.
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
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.
11.3	A cast should not be performed between a pointer type and an integral type.
12.12	The underlying bit representations of floating-point values shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
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.
18.4	Unions shall not be used.

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

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. The `SQO-subset2` option checks the rules in `SQO-subset1` and some additional rules.

Rule number	Description
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.
6.3	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types
8.7	Objects shall be defined at block scope if they are only accessed from within a single function
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
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
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type
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.
11.3	A cast should not be performed between a pointer type and an integral type.
11.5	Type casting from any type to or from pointers shall not be used
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits

Rule number	Description
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned
12.10	The comma operator shall not be used
12.12	The underlying bit representations of floating-point values shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
16.9	A function identifier shall only be used with either a preceding <code>&</code> , or with a parenthesized parameter list, which may be empty
17.3	<code>></code> , <code>>=</code> , <code><</code> , <code><=</code> shall not be applied to pointer types except where they point to the same array.
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.
18.4	Unions shall not be used.
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives
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 <code>#</code> or <code>##</code>
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in <code>#ifdef</code> and <code>#ifndef</code> preprocessor directives and the <code>defined()</code> operator
19.12	There shall be at most one occurrence of the <code>#</code> or <code>##</code> preprocessor operators in a single macro definition.
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; }
```

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 and MISRA AC AGC Coding Rules

In this section...
“Supported MISRA C:2004 and MISRA AC AGC Rules” on page 2-14
“Unsupported MISRA C:2004 and MISRA AC AGC Rules” on page 2-50

Supported MISRA C:2004 and MISRA AC AGC 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 “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” on page 2-15
- “Language Extensions” on page 2-18
- “Documentation” on page 2-18
- “Character Sets” on page 2-19
- “Identifiers” on page 2-19

- “Types” on page 2-20
- “Constants” on page 2-21
- “Declarations and Definitions” on page 2-22
- “Initialization” on page 2-24
- “Arithmetic Type Conversion” on page 2-25
- “Pointer Type Conversion” on page 2-29
- “Expressions” on page 2-30
- “Control Statement Expressions” on page 2-34
- “Control Flow” on page 2-37
- “Switch Statements” on page 2-39
- “Functions” on page 2-40
- “Pointers and Arrays” on page 2-42
- “Structures and Unions” on page 2-43
- “Preprocessing Directives” on page 2-43
- “Standard Libraries” on page 2-47
- “Runtime Failures” on page 2-50

Environment

N.	MISRA Definition	Messages in report file	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 <i>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</i> precedes each of the following messages: <ul style="list-style-type: none"> • ANSI C does not allow '#include_next' • ANSI C does not allow macros with variable arguments list 	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	Polyspace Specification
		<ul style="list-style-type: none">• 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.• text following '#else' or '#endif' violates ANSI standard.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
1.1 (cont.)		<p>The text <i>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</i> precedes each of the following messages:</p> <ul style="list-style-type: none"> • 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. • Too many nesting levels of #includes: N_1. The limit is N_0. • Too many macro definitions: N_1. The limit is N_0. • Too many nesting levels for control flow: N_1. The limit is N_0. 	

N.	MISRA Definition	Messages in report file	Polyspace Specification
		<ul style="list-style-type: none"> Too many enumeration constants: N_1. The limit is N_0. 	

Language Extensions

N.	MISRA Definition	Messages in report file	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 <code>asm</code> functions or in <code>asm pragma</code> (only warning is given on <code>asm</code> statements even if it is encapsulated by a MACRO).
2.2	Source code shall only use <code>/* */</code> 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 <code>/*</code> shall not be used within a comment	The character sequence <code>/*</code> shall not appear within a comment.	This rule violation is also raised when the character sequence <code>/*</code> inside a C++ comment. Note: This rule cannot be annotated in the source code.

Documentation

Rule	MISRA Definition	Messages in report file	Polyspace Specification
3.4	All uses of the <code>#pragma</code> directive shall be documented and explained.	All uses of the <code>#pragma</code> directive shall be documented and explained.	To check this rule, the option <code>-allowed-pragmas</code> 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	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	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.	<ul style="list-style-type: none"> 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
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 Bug Finder and Code Prover check this coding rule

N.	MISRA Definition	Messages in report file	Polyspace Specification
			differently. The analyses can produce different results.
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 an <code>idf</code> 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: <ul style="list-style-type: none"> • 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	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.	<ul style="list-style-type: none"> • Value of type plain char is implicitly converted to signed char. • Value of type plain char is implicitly converted to unsigned char. 	Warning if value of type plain char is implicitly converted to value of type signed char or unsigned char.

N.	MISRA Definition	Messages in report file	Polyspace Specification
		<ul style="list-style-type: none"> • Value of type signed char is implicitly converted to plain char. • Value of type unsigned char is implicitly converted to plain char. 	
6.3	<i>typedefs</i> 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 <i>unsigned int</i> or <i>signed int</i> .	Bit fields shall only be defined to be of type unsigned int or signed int.	
6.5	Bit fields of type <i>signed int</i> 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	Polyspace Specification
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.	<ul style="list-style-type: none"> • 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	Polyspace Specification
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	<ul style="list-style-type: none"> • 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
8.4	If objects or functions are declared more than once their types shall be compatible.	<ul style="list-style-type: none"> • 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. 	<p>Violations of this rule might be generated during the link phase.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
8.5	There shall be no definitions of objects or functions in a header file	<ul style="list-style-type: none"> • Object 'XX' should not be defined in a header file. • Function 'XX' should not be defined in a header file. 	Tentative of definitions are considered as definitions.

N.	MISRA Definition	Messages in report file	Polyspace Specification
		<ul style="list-style-type: none"> • Fragment of function should not be defined in a header file. 	
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.	<p>Restricted to explicit extern declarations (tentative of definitions are ignored).</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
8.9	Definition: An identifier with external linkage shall have exactly one external definition.	<ul style="list-style-type: none"> • 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 	<p>Tentative of definitions are considered as definitions, no warning on predefined symbols.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
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.	<p>Assumes that 8.1 is not violated. No warning if 0 uses.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>

N.	MISRA Definition	Messages in report file	Polyspace Specification
8.11	The <i>static</i> 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	Size of array 'XX' should be explicitly stated.	

Initialization

N.	MISRA Definition	Messages in report file	Polyspace Specification
9.1	All automatic variables shall have been assigned a value before being used.		<p>Checked during code analysis.</p> <p>Violations displayed as Non-initialized variable results.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
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	Polyspace Specification
10.1	<p>The value of an expression of integer type shall not be implicitly converted to a different underlying type if:</p> <ul style="list-style-type: none"> • it is not a conversion to a wider integer type of the same signedness, or • the expression is complex, or • the expression is not constant and is a function argument, or • the expression is not constant and is a return expression 	<ul style="list-style-type: none"> • 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 • 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. 	<p>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.</p> <p>An expression of bool or enum types has int as underlying type.</p> <p>Plain char may have signed or unsigned underlying type (depending on Polyspace target configuration or option setting).</p> <p>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).</p>
10.1 (cont)		<ul style="list-style-type: none"> • 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 	<p>No violation reported when:</p> <ul style="list-style-type: none"> • The implicit conversion is a type widening, without change of signedness if integer • The expression is an argument expression or a return expression

N.	MISRA Definition	Messages in report file	Polyspace Specification
		<p>type XX to XX, but it is a complex expression.</p> <ul style="list-style-type: none"> • 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 type XX as argument of function whose corresponding parameter type is XX. 	<p>No violation reported when the following are all true:</p> <ul style="list-style-type: none"> • 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

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.2	<p>The value of an expression of floating type shall not be implicitly converted to a different type if</p> <ul style="list-style-type: none"> • 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 	<ul style="list-style-type: none"> • 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 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. 	<p>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.</p> <p>No violation reported when:</p> <ul style="list-style-type: none"> • The implicit conversion is a type widening • The expression is an argument expression or a return expression.

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	<ul style="list-style-type: none"> • 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 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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.5	If the bitwise operator <code>~</code> and <code><<</code> are applied to an operand of underlying type <i>unsigned char</i> or <i>unsigned short</i> , the result shall be immediately cast to the underlying type of the operand	Bitwise [<code><<</code> <code>~</code>] is applied to the operand of underlying type [<code>unsigned char</code> <code>unsigned short</code>], the result shall be immediately cast to the underlying type.	
10.6	The “U” suffix shall be applied to all constants of <i>unsigned</i> types	No explicit 'U suffix on constants of an unsigned type.	<p>Warning when the type determined from the value and the base (octal, decimal or hexadecimal) is unsigned and there is no suffix <code>u</code> or <code>U</code>.</p> <p>For example, when the size of the <code>int</code> and <code>long int</code> data types is 32 bits, the coding rule checker will report a violation of rule 10.6 for the following line:</p> <pre>int a = 2147483648;</pre> <p>There is a difference between decimal and hexadecimal constants when <code>int</code> and <code>long int</code> are not the same size.</p>

Pointer Type Conversion

N.	MISRA Definition	Messages in report file	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.	<p>Casts and implicit conversions involving a function pointer.</p> <p>Casts or implicit conversions from <code>NULL</code> or <code>(void*)0</code> do not give any warning.</p>

N.	MISRA Definition	Messages in report file	Polyspace Specification
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
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 <i>const</i> or <i>volatile</i> qualification from the type addressed by a pointer	A cast shall not be performed that removes any <i>const</i> or <i>volatile</i> qualification from the type addressed by a pointer	Extended to all conversions

Expressions

N.	MISRA Definition	Messages in report file	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.	<ul style="list-style-type: none"> 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 <code>i = i++</code> ; no detection on <code>tab[2] = tab[2]++</code>);. 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).

N.	MISRA Definition	Messages in report file	Polyspace Specification
12.3	The <code>sizeof</code> operator should not be used on expressions that contain side effects.	The <code>sizeof</code> operator should not be used on expressions that contain side effects.	No warning on volatile accesses
12.4	The right hand operand of a logical <code>&&</code> or <code> </code> operator shall not contain side effects.	The right hand operand of a logical <code>&&</code> or <code> </code> operator shall not contain side effects.	No warning on volatile accesses
12.5	The operands of a logical <code>&&</code> or <code> </code> shall be primary-expressions.	<ul style="list-style-type: none"> • operand of logical <code>&&</code> is not a primary expression • operand of logical <code> </code> is not a primary expression • The operands of a logical <code>&&</code> or <code> </code> shall be primary-expressions. 	<p>During preprocessing, violations of this rule are detected on the expressions in <code>#if</code> directives.</p> <p>Allowed exception on associatively (<code>a && b && c</code>), (<code>a b c</code>).</p>

N.	MISRA Definition	Messages in report file	Polyspace Specification
12.6	<p>Operands of logical operators (&&, and !) should be effectively Boolean.</p> <p>Expression that are effectively Boolean should not be used as operands to operators other than (&&, or !).</p>	<ul style="list-style-type: none"> • 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 '?:'. 	<p>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.</p> <p>Some operators may return Boolean-like expressions, for example, (<code>var == 0</code>).</p> <p>Consider the following code:</p> <pre>unsigned char flag; if (!flag)</pre> <p>The rule checker reports a violation of rule 12.6:</p> <p>Operand of '!' logical operator should be effectively Boolean. The operand <code>flag</code> is not a Boolean but an unsigned char.</p> <p>To be compliant with rule 12.6, the code must be rewritten either as</p> <pre>if (!(flag != 0)) or if (flag == 0)</pre> <p>The use of the option -<code>boolean-types</code> may increase or decrease the</p>

N.	MISRA Definition	Messages in report file	Polyspace Specification
			number of warnings generated.
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed	<ul style="list-style-type: none"> • [~/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 	<p>The underlying type for an integer is signed when:</p> <ul style="list-style-type: none"> • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number
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.	<ul style="list-style-type: none"> • 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).. 	<p>The numbers that are manipulated in preprocessing directives are 64 bits wide so that valid shift range is between 0 and 63</p> <p>Check is also extended onto bitfields with the field width or the width of the base type when it is within a complex expression</p>
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	<ul style="list-style-type: none"> • Unary - on operand of unsigned underlying type XX. • Minus operator applied to an expression whose underlying type is unsigned 	<p>The underlying type for an integer is signed when:</p> <ul style="list-style-type: none"> • 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.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
12.11	Evaluation of constant unsigned expression should not lead to wraparound.	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	
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: <ul style="list-style-type: none"> • 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: <pre>union { float f; int i; } ...</pre>
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	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, unless the operand is effectively Boolean	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	No warning is given on integer constants. Example: if (2) The use of the option -boolean-types may

N.	MISRA Definition	Messages in report file	Polyspace Specification
			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 <i>for</i> statement shall not contain any objects of floating type	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type	If <i>for</i> index is a variable symbol, checked that it is not a float.

N.	MISRA Definition	Messages in report file	Polyspace Specification
13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control	<ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> (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. 	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.
13.6	Numeric variables being used within a <i>for</i> 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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
13.7	Boolean operations whose results are invariant shall not be permitted	<ul style="list-style-type: none"> • 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	Polyspace Specification
14.1	There shall be no unreachable code.	There shall be no unreachable code.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
14.2	All non-null statements shall either have at least one side effect however executed, or cause control flow to change	<ul style="list-style-type: none"> • All non-null statements shall either: • have at least one side effect however executed, or • cause control flow to change 	
14.3	<p>All non-null statements shall either</p> <ul style="list-style-type: none"> • have at least 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:

N.	MISRA Definition	Messages in report file	Polyspace Specification
			<ul style="list-style-type: none"> • 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 <i>goto</i> statement shall not be used.	The goto statement shall not be used.	
14.5	The <i>continue</i> statement shall not be used.	The continue statement shall not be used.	
14.6	For any iteration statement there shall be at most one <i>break</i> statement used for loop termination	For any iteration statement there shall be at most one break statement used for loop termination	
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 <i>switch</i> , <i>while</i> , <i>do while</i> or <i>for</i> statement shall be a compound statement	<ul style="list-style-type: none"> • 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 	

N.	MISRA Definition	Messages in report file	Polyspace Specification
14.9	An <i>if (expression)</i> construct shall be followed by a compound statement. The <i>else</i> keyword shall be followed by either a compound statement, or another <i>if</i> statement	<ul style="list-style-type: none"> • 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 <i>if else if</i> constructs should contain a final <i>else</i> clause.	All if else if constructs should contain a final else clause.	

Switch Statements

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

N.	MISRA Definition	Messages in report file	Polyspace Specification
15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a <i>switch</i> 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 <i>break</i> 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.
15.3	The final clause of a <i>switch</i> statement shall be the <i>default</i> clause	The final clause of a switch statement shall be the default clause	
15.4	A <i>switch</i> 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 <i>switch</i> statement shall have at least one <i>case</i> clause	Every switch statement shall have at least one case clause	

Functions

N.	MISRA Definition	Messages in report file	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 (Use the call graph in Polyspace Code Prover). Polyspace also partially checks this rule during the 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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.
16.5	Functions with no parameters shall be declared with parameter type <i>void</i> .	Functions with no parameters shall be declared with parameter type <i>void</i> .	Definitions are also checked.
16.6	The number of arguments passed to a function shall match the number of parameters.	<ul style="list-style-type: none"> • 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.	

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

Pointers and Arrays

N.	MISRA Definition	Messages in report file	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	Pointer to a parameter is an illegal return value. Pointer	Warning when assigning address to a global variable,

N.	MISRA Definition	Messages in report file	Polyspace Specification
	be assigned to an object that may persist after the object has ceased to exist.	to a local is an illegal return value.	returning a local variable address, or returning a parameter address.

Structures and Unions

N.	MISRA Definition	Messages in report file	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.	<ul style="list-style-type: none"> 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	Polyspace Specification
19.1	<code>#include</code> statements in a file shall only be preceded by other preprocessors directives or comments	<code>#include</code> statements in a file shall only be preceded by other preprocessors directives or comments	A message is displayed when a <code>#include</code> 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 <code>#include</code> directives	<ul style="list-style-type: none"> A message is displayed on characters ', " or / * between < and > in <code>#include <filename></code> A message is displayed on characters ', or / * between " and " in <code>#include "filename"</code> 	

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.3	The <i>#include</i> directive shall be followed by either a <filename> or "filename" sequence.	<ul style="list-style-type: none"> • '#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.	<p>We assume that a macro definition does not violate this rule when it expands to:</p> <ul style="list-style-type: none"> • 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
19.5	Macros shall not be #defined and #undefd within a block.	<ul style="list-style-type: none"> • Macros shall not be #define'd within a block. • Macros shall not be #undef'd within a block. 	
19.6	#undef shall not be used.	#undef shall not be used.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.7	A function should be used in preference to a function like-macro.	A function should be used in preference to a function like-macro	Message on all function-like macro definitions.
19.8	A function-like macro shall not be invoked without all of its arguments	<ul style="list-style-type: none"> • 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)
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.	<p>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.</p> <p>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,.</p>

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in <code>#ifdef</code> and <code>#ifndef</code> preprocessor directives and the <code>defined()</code> operator.	'<name>' is not defined.	
19.12	There shall be at most one occurrence of the <code>#</code> or <code>##</code> preprocessor operators in a single macro definition.	More than one occurrence of the <code>#</code> or <code>##</code> preprocessor operators.	
19.13	The <code>#</code> and <code>##</code> preprocessor operators should not be used	Message on definitions of macros using <code>#</code> or <code>##</code> operators	
19.14	The <code>defined</code> 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.	<p>When a header file is formatted as,</p> <pre>#ifndef <control macro> #define <control macro> <contents> #endif</pre> <p>or,</p> <pre>#ifndef <control macro> #error ... #else #define <control macro> <contents> #endif</pre> <p>it is assumed that precautions have been taken to prevent multiple inclusions. Otherwise, a violation of this MISRA rule is detected.</p>

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	directive is not syntactically meaningful.	
19.17	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> or <code>#ifdef</code> directive to which they are related.	<ul style="list-style-type: none"> • <code>#elif</code> not within a conditional. • <code>#else</code> not within a conditional. • <code>#elif</code> not within a conditional. • <code>#endif</code> not within a conditional. • unbalanced <code>#endif</code>. • unterminated <code>#if</code> conditional. • unterminated <code>#ifdef</code> conditional. • unterminated <code>#ifndef</code> conditional. 	

Standard Libraries

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be redefined. • The macro '<code><name></code>' 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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.3	The validity of values passed to library functions shall be checked.	Validity of values passed to library functions shall be checked	<p>Warning for argument in library function call if the following are all true:</p> <ul style="list-style-type: none"> • 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. <p>The library function can be one of the following : <code>sqrt</code>, <code>tan</code>, <code>pow</code>, <code>log</code>, <code>log10</code>, <code>fmod</code>, <code>acos</code>, <code>asin</code>, <code>acosh</code>, <code>atanh</code>, or <code>atan2</code>.</p>
20.4	Dynamic heap memory allocation shall not be used.	<ul style="list-style-type: none"> • 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 <code>errno</code> shall not be used	The error indicator <code>errno</code> shall not be used	Assumes that rule 20.2 is not violated
20.6	The macro <code>offsetof</code> , in library <code><stddef.h></code> , shall not be used.	<ul style="list-style-type: none"> • The macro '<name>' shall not be used. • Identifier XX should not be used. 	Assumes that rule 20.2 is not violated

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.7	The <i>setjmp</i> macro and the <i>longjmp</i> function shall not be used.	<ul style="list-style-type: none"> • 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.	<ul style="list-style-type: none"> • 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.	<ul style="list-style-type: none"> • 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
20.10	The library functions atof, atoi and toll from library <stdlib.h> shall not be used.	<ul style="list-style-type: none"> • 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.	<ul style="list-style-type: none"> • 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.	<ul style="list-style-type: none"> • 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	Polyspace Specification
21.1	Minimization of runtime failures shall be ensured by the use of at least one of: <ul style="list-style-type: none"> • static verification tools/ techniques; • dynamic verification tools/ techniques; • explicit coding of checks to handle runtime faults. 		Done by Polyspace. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Unsupported MISRA C:2004 and MISRA AC AGC Rules

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 “**Polyspace Specification**” column describes the reason each rule is not checked.

Environment

Rule	Description	Polyspace Specification
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/compiler/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	The documentation of compiler must be checked.

Rule	Description	Polyspace Specification
	significance and case sensitivity are supported for external identifiers.	
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	Polyspace Specification
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	Polyspace Specification
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.
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	Polyspace Specification
18.3 (Required)	An area of memory shall not be reused for unrelated purposes.	"purpose" is functional design issue.

Polyspace MISRA C:2012 Checker

The Polyspace MISRA C:2012 checker helps you to comply with the MISRA C 2012 coding standard.³

When MISRA C:2012 guidelines are violated, the Polyspace MISRA C:2012 checker provides messages with information about the violated rule or directive. Most violations are found during the compile phase of an analysis.

The checker can check **138** of the **159** MISRA C:2012 guidelines.

Each guideline is categorized into one of these three categories: mandatory, required, or advisory. When you set up rule checking, you can select subsets of these categories to check. For automatically generated code, some rules change categories, including to one additional category: readability. The “Use generated code requirements (C)” option activates the categorization for automatically generated code.

There are additional subsets of MISRA C:2012 guidelines defined by Polyspace called Software Quality Objectives (SQO) that can have a direct or indirect impact on the precision of your results. When you set up checking, you can select these subsets. These subsets are defined in “Software Quality Objective Subsets (C:2012)” on page 2-54.

See Also

“Check MISRA C:2012” | “Use generated code requirements (C)”

Related Examples

- “Activate Coding Rules Checker” on page 3-2
- “Set Up Coding Rules Checking”

More About

- “MISRA C:2012 Directives and Rules”
- “Software Quality Objective Subsets (C:2012)” on page 2-54

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

Software Quality Objective Subsets (C:2012)

In this section...
“Guidelines in SQO-Subset1” on page 2-54
“Guidelines in SQO-Subset2” on page 2-55

These subsets of MISRA C:2012 guidelines can have a direct or indirect impact on the precision of your Polyspace results. When you set up coding rules checking, you can select these subsets.

Guidelines in SQO - Subset 1

Rule	Description
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage
8.11	When an array with external linkage is declared, its size should be explicitly specified
8.13	A pointer should point to a const-qualified type whenever possible
11.1	Conversions shall not be performed between a pointer to a function and any other type
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type
11.4	A conversion should not be performed between a pointer to object and an integer type
11.5	A conversion should not be performed from pointer to void into pointer to object
11.6	A cast shall not be performed between pointer to void and an arithmetic type
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type
14.1	A loop counter shall not have essentially floating type
14.2	A for loop shall be well-formed
15.1	The goto statement should not be used

Rule	Description
15.2	The goto statement shall jump to a label declared later in the same function
15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement
15.5	A function should have a single point of exit at the end
17.1	The features of <starg.h> shall not be used
17.2	Functions shall not call themselves, either directly or indirectly
18.3	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object
18.4	The +, -, += and -= operators should not be applied to an expression of pointer type
18.5	Declarations should contain no more than two levels of pointer nesting
18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist
19.2	The union keyword should not be used
21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used

Guidelines 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. The `SQO-subset2` option checks the rules in `SQO-subset1` and some additional rules.

Rule	Description
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage
8.11	When an array with external linkage is declared, its size should be explicitly specified
8.13	A pointer should point to a const-qualified type whenever possible
11.1	Conversions shall not be performed between a pointer to a function and any other type

Rule	Description
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type
11.4	A conversion should not be performed between a pointer to object and an integer type
11.5	A conversion should not be performed from pointer to void into pointer to object
11.6	A cast shall not be performed between pointer to void and an arithmetic type
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer
12.1	The precedence of operators within expressions should be made explicit
12.3	The comma operator should not be used
13.2	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders
13.4	The result of an assignment operator should not be used
14.1	A loop counter shall not have essentially floating type
14.2	A for loop shall be well-formed
14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type
15.1	The goto statement should not be used
15.2	The goto statement shall jump to a label declared later in the same function
15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement
15.5	A function should have a single point of exit at the end
15.6	The body of an iteration- statement or a selection- statement shall be a compound- statement
15.7	All if ... else if constructs shall be terminated with an else statement
16.4	Every switch statement shall have a default label

Rule	Description
16.5	A default label shall appear as either the first or the last switch label of a switch statement
17.1	The features of <starg.h> shall not be used
17.2	Functions shall not call themselves, either directly or indirectly
17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
18.3	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object
18.4	The +, -, += and -= operators should not be applied to an expression of pointer type
18.5	Declarations should contain no more than two levels of pointer nesting
18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist
19.2	The union keyword should not be used
20.4	A macro shall not be defined with the same name as a keyword
20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument
20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses
20.9	All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation
20.11	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator
21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used

See Also

“Check MISRA C:2012” | “Use generated code requirements (C)”

Related Examples

- “Activate Coding Rules Checker” on page 3-2

- “Set Up Coding Rules Checking”

More About

- “MISRA C:2012 Directives and Rules”

Unsupported MISRA C:2012 Guidelines

The Polyspace coding rules checker does not check the following MISRA C:2012 coding rules. These rules cannot be enforced because they are outside the scope of Polyspace software. These guidelines concern documentation, dynamic aspects, or functional aspects of MISRA rules.

Number	Category	AGC Category	Definition
Directive 1.1	Required	Required	Any implementation-defined behavior on which the output of the program depends shall be documented and understood
Directive 3.1	Required	Required	All code shall be traceable to documented requirements
Directive 4.2	Advisory	Advisory	All usage of assembly language should be documented
Directive 4.4	Advisory	Advisory	Sections of code should not be “commented out”
Directive 4.7	Required	Required	If a function returns error information, then that error information shall be tested
Directive 4.8	Advisory	Advisory	If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden
Directive 4.12	Required	Required	Dynamic memory allocation shall not be used

Polyspace MISRA C++ Checker

The Polyspace MISRA C++ checker helps you comply with the MISRA C++:2008 coding standard.⁴

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 2-61.

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>.

4. 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 2-61

“SQO Subset 2 – Indirect Impact on Selectivity” on page 2-63

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.
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.

MISRA C++ Rule	Description
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.
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.

MISRA C++ Rule	Description
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. The `SQO-subset2` option checks the rules in `SQO-subset1` and `SQO-subset2`.

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 <code>bool</code> shall not be used as operands to built-in operators other than the assignment operator <code>=</code> , the logical operators <code>&&</code> , <code> </code> , <code>!</code> , the

MISRA C++ Rule	Description
	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.
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.

MISRA C++ Rule	Description
5-2-11	The comma operator, && operator and the operator shall not be overloaded.
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.
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.

MISRA C++ Rule	Description
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.
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.

MISRA C++ Rule	Description
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.
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 2-68
- “Unsupported MISRA C++ Rules” on page 2-89

Supported MISRA C++ Coding Rules

- “Language Independent Issues” on page 2-68
- “General” on page 2-69
- “Lexical Conventions” on page 2-69
- “Basic Concepts” on page 2-71
- “Standard Conversions” on page 2-72
- “Expressions” on page 2-73
- “Statements” on page 2-77
- “Declarations” on page 2-79
- “Declarators” on page 2-81
- “Classes” on page 2-82
- “Derived Classes” on page 2-82
- “Member Access Control” on page 2-83
- “Special Member Functions” on page 2-83
- “Templates” on page 2-84
- “Exception Handling” on page 2-85
- “Preprocessing Directives” on page 2-86
- “Library Introduction” on page 2-88
- “Language Support Library” on page 2-88
- “Diagnostic Library” on page 2-89
- “Input/output Library” on page 2-89

Language Independent Issues

N.	Category	MISRA Definition	Polyspace Specification
0-1-1	Required	A project shall not contain unreachable code.	Bug Finder and Code Prover check this coding rule differently. The

N.	Category	MISRA Definition	Polyspace Specification
			analyses can produce different results.
0-1-2	Required	A project shall not contain infeasible paths.	
0-1-7	Required	The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
0-1-10	Required	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.	Category	MISRA Definition	Polyspace Specification
1-0-1	Required	All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Lexical Conventions

N.	Category	MISRA Definition	Polyspace Specification
2-3-1	Required	Trigraphs shall not be used.	
2-5-1	Advisory	Digraphs should not be used.	
2-7-1	Required	The character sequence /* shall not be used within a C-style comment.	This rule cannot be annotated in the source code.
2-10-1	Required	Different identifiers shall be typographically unambiguous.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-2	Required	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.

N.	Category	MISRA Definition	Polyspace Specification
			Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-3	Required	A typedef name (including qualification, if any) shall be a unique identifier.	No detection across namespaces. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-4	Required	A class, union or enum name (including qualification, if any) shall be a unique identifier.	No detection across namespaces. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-5	Advisory	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. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-6	Required	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. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-13-1	Required	Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.	

N.	Category	MISRA Definition	Polyspace Specification
2-13-2	Required	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.	
2-13-3	Required	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.	
2-13-4	Required	Literal suffixes shall be upper case.	
2-13-5	Required	Narrow and wide string literals shall not be concatenated.	

Basic Concepts

N.	Category	MISRA Definition	Polyspace Specification
3-1-1	Required	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.	
3-1-2	Required	Functions shall not be declared at block scope.	
3-1-3	Required	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.	
3-2-1	Required	All declarations of an object or function shall have compatible types.	
3-2-2	Required	The One Definition Rule shall not be violated.	Report type, template, and inline function defined in source file
3-2-3	Required	A type, object or function that is used in multiple translation units shall be declared in one and only one file.	
3-2-4	Required	An identifier with external linkage shall have exactly one definition.	

N.	Category	MISRA Definition	Polyspace Specification
3-3-1	Required	Objects or functions with external linkage shall be declared in a header file.	
3-3-2	Required	If a function has internal linkage then all re-declarations shall include the static storage class specifier.	
3-4-1	Required	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.	
3-9-1	Required	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.
3-9-2	Advisory	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	Required	The underlying bit representations of floating-point values shall not be used.	

Standard Conversions

N.	Category	MISRA Definition	Polyspace Specification
4-5-1	Required	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	Required	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 !=	

N.	Category	MISRA Definition	Polyspace Specification
		=, the unary & operator, and the relational operators <, <=, >, >=.	
4-5-3	Required	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.	Category	MISRA Definition	Polyspace Specification
5-0-1	Required	The value of an expression shall be the same under any order of evaluation that the standard permits.	
5-0-2	Advisory	Limited dependence should be placed on C++ operator precedence rules in expressions.	
5-0-3	Required	A cvalue expression shall not be implicitly converted to a different underlying type.	Assumes that ptrdiff_t is signed integer
5-0-4	Required	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	Required	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	Required	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.

N.	Category	MISRA Definition	Polyspace Specification
5-0-7	Required	There shall be no explicit floating-integral conversions of a cvalue expression.	
5-0-8	Required	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.	
5-0-9	Required	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.	
5-0-10	Required	If the bitwise operators <code>~</code> and <code><<</code> 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	Required	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	Required	Signed char and unsigned char type shall only be used for the storage and use of numeric values.	
5-0-14	Required	The first operand of a conditional-operator shall have type <code>bool</code> .	
5-0-15	Required	Array indexing shall be the only form of pointer arithmetic.	Warning on operations on pointers. (<code>p+I</code> , <code>I+p</code> and <code>p-I</code> , where <code>p</code> is a pointer and <code>I</code> an integer, <code>p[i]</code> accepted).
5-0-18	Required	<code>></code> , <code>>=</code> , <code><</code> , <code><=</code> 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	Required	The declaration of objects shall contain no more than two levels of pointer indirection.	

N.	Category	MISRA Definition	Polyspace Specification
5-0-20	Required	Non-constant operands to a binary bitwise operator shall have the same underlying type.	
5-0-21	Required	Bitwise operators shall only be applied to operands of unsigned underlying type.	
5-2-1	Required	Each operand of a logical <code>&&</code> or <code> </code> shall be a postfix - expression.	During preprocessing, violations of this rule are detected on the expressions in <code>#if</code> directives. Allowed exception on associativity (<code>a && b && c</code>), (<code>a b c</code>).
5-2-2	Required	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of <code>dynamic_cast</code> .	
5-2-3	Advisory	Casts from a base class to a derived class should not be performed on polymorphic types.	
5-2-4	Required	C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.	
5-2-5	Required	A cast shall not remove any <code>const</code> or <code>volatile</code> qualification from the type of a pointer or reference.	
5-2-6	Required	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	Required	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. <code>int</code>) are not detected."

N.	Category	MISRA Definition	Polyspace Specification
5-2-8	Required	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	Advisory	A cast should not convert a pointer type to an integral type.	
5-2-10	Advisory	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	
5-2-11	Required	The comma operator, && operator and the operator shall not be overloaded.	
5-2-12	Required	An identifier with array type passed as a function argument shall not decay to a pointer.	
5-3-1	Required	Each operand of the ! operator, the logical && or the logical operators shall have type bool.	
5-3-2	Required	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	
5-3-3	Required	The unary & operator shall not be overloaded.	
5-3-4	Required	Evaluation of the operand to the sizeof operator shall not contain side effects.	No warning on volatile accesses and function calls
5-8-1	Required	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.	

N.	Category	MISRA Definition	Polyspace Specification
5-14-1	Required	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	Required	The comma operator shall not be used.	
5-19-1	Required	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	

Statements

N.	Category	MISRA Definition	Polyspace Specification
6-2-1	Required	Assignment operators shall not be used in sub-expressions.	
6-2-2	Required	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.	
6-2-3	Required	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	Required	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.	
6-4-1	Required	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	Required	All if ... else if constructs shall be terminated with an else clause.	Also detects cases where the last if is in the block of the last else

N.	Category	MISRA Definition	Polyspace Specification
			(same behavior as JSF, stricter than MISRA C). Example: "if ... else { if ...}" raises the rule
6-4-3	Required	A switch statement shall be a well-formed switch statement.	Return statements are considered as jump statements.
6-4-4	Required	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	
6-4-5	Required	An unconditional throw or break statement shall terminate every non - empty switch-clause.	
6-4-6	Required	The final clause of a switch statement shall be the default-clause.	
6-4-7	Required	The condition of a switch statement shall not have bool type.	
6-4-8	Required	Every switch statement shall have at least one case-clause.	
6-5-1	Required	A for loop shall contain a single loop-counter which shall not have floating type.	
6-5-2	Required	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	Required	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	Required	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.	

N.	Category	MISRA Definition	Polyspace Specification
6-5-5	Required	A loop-control-variable other than the loop-counter shall not be modified within condition or expression.	
6-5-6	Required	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.	
6-6-1	Required	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	Required	The goto statement shall jump to a label declared later in the same function body.	
6-6-3	Required	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	Required	For any iteration statement there shall be no more than one break or goto statement used for loop termination.	
6-6-5	Required	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.	Category	MISRA Definition	Polyspace Specification
7-3-1	Required	The global namespace shall only contain main, namespace declarations and extern "C" declarations.	
7-3-2	Required	The identifier main shall not be used for a function other than the global function main.	

N.	Category	MISRA Definition	Polyspace Specification
7-3-3	Required	There shall be no unnamed namespaces in header files.	
7-3-4	Required	using-directives shall not be used.	
7-3-5	Required	Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.	
7-3-6	Required	using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.	
7-4-2	Required	Assembler instructions shall only be introduced using the asm declaration.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
7-4-3	Required	Assembly language shall be encapsulated and isolated.	
7-5-1	Required	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.	
7-5-2	Required	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.	
7-5-3	Required	A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.	
7-5-4	Advisory	Functions should not call themselves, either directly or indirectly.	

Declarators

N.	Category	MISRA Definition	Polyspace Specification
8-0-1	Required	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.	
8-3-1	Required	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	Required	Functions shall not be defined using the ellipsis notation.	
8-4-2	Required	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.	
8-4-3	Required	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8-4-4	Required	A function identifier shall either be used to call the function or it shall be preceded by &.	
8-5-1	Required	All variables shall have a defined value before they are used.	Non-initialized variable in results and error messages for obvious cases
8-5-2	Required	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	
8-5-3	Required	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.	Category	MISRA Definition	Polyspace Specification
9-3-1	Required	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	Required	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	Required	Unions shall not be used.	
9-6-2	Required	Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.	
9-6-3	Required	Bit-fields shall not have enum type.	
9-6-4	Required	Named bit-fields with signed integer type shall have a length of more than one bit.	

Derived Classes

N.	Category	MISRA Definition	Polyspace Specification
10-1-1	Advisory	Classes should not be derived from virtual bases.	
10-1-2	Required	A base class shall only be declared virtual if it is used in a diamond hierarchy.	Assumes 10.1.1 not required
10-1-3	Required	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.	
10-2-1	Required	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	Required	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.

N.	Category	MISRA Definition	Polyspace Specification
10-3-2	Required	Each overriding virtual function shall be declared with the virtual keyword.	
10-3-3	Required	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.	

Member Access Control

N.	Category	MISRA Definition	Polyspace Specification
11-0-1	Required	Member data in non- POD class types shall be private.	

Special Member Functions

N.	Category	MISRA Definition	Polyspace Specification
12-1-1	Required	An object's dynamic type shall not be used from the body of its constructor or destructor.	
12-1-2	Advisory	All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.	
12-1-3	Required	All constructors that are callable with a single argument of fundamental type shall be declared explicit.	
12-8-1	Required	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	Required	The copy assignment operator shall be declared protected or private in an abstract class.	

Templates

N.	Category	MISRA Definition	Polyspace Specification
14-5-2	Required	A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.	
14-5-3	Required	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	Required	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	Required	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.	
14-7-3	Required	All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.	
14-8-1	Required	Overloaded function templates shall not be explicitly specialized.	<p>All specializations of overloaded templates are rejected even if overloading occurs after the call.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
14-8-2	Advisory	The viable function set for a function call should either contain no	

N.	Category	MISRA Definition	Polyspace Specification
		function specializations, or only contain function specializations.	

Exception Handling

N.	Category	MISRA Definition	Polyspace Specification
15-0-2	Advisory	An exception object should not have pointer type.	NULL not detected (see 15-1-2).
15-0-3	Required	Control shall not be transferred into a try or catch block using a goto or a switch statement.	
15-1-2	Required	NULL shall not be thrown explicitly.	
15-1-3	Required	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.	
15-3-2	Advisory	There should be at least one exception handler to catch all otherwise unhandled exceptions.	<p>Detect that there is no try/catch in the main and that the catch does not handle all exceptions. Not detected if no "main".</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
15-3-3	Required	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	Required	A class type exception shall always be caught by reference.	
15-3-6	Required	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	

N.	Category	MISRA Definition	Polyspace Specification
		bases, the handlers shall be ordered most-derived to base class.	
15-3-7	Required	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	Required	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	Required	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	Required	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.	Category	MISRA Definition	Polyspace Specification
16-0-1	Required	<code>#include</code> directives in a file shall only be preceded by other preprocessor directives or comments.	
16-0-2	Required	Macros shall only be <code>#define 'd</code> or <code>#undef 'd</code> in the global namespace.	
16-0-3	Required	<code>#undef</code> shall not be used.	
16-0-4	Required	Function-like macros shall not be defined.	

N.	Category	MISRA Definition	Polyspace Specification
16-0-5	Required	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	
16-0-6	Required	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	Required	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.	
16-0-8	Required	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.	
16-1-1	Required	The defined preprocessor operator shall only be used in one of the two standard forms.	
16-1-2	Required	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	Required	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	Required	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.	
16-2-3	Required	Include guards shall be provided.	
16-2-4	Required	The ', ", /* or // characters shall not occur in a header file name.	
16-2-5	Advisory	The \ character should not occur in a header file name.	

N.	Category	MISRA Definition	Polyspace Specification
16-2-6	Required	The #include directive shall be followed by either a <filename> or "filename" sequence.	
16-3-1	Required	There shall be at most one occurrence of the # or ## operators in a single macro definition.	
16-3-2	Advisory	The # and ## operators should not be used.	

Library Introduction

N.	Category	MISRA Definition	Polyspace Specification
17-0-1	Required	Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
17-0-2	Required	The names of standard library macros and objects shall not be reused.	
17-0-5	Required	The setjmp macro and the longjmp function shall not be used.	

Language Support Library

N.	Category	MISRA Definition	Polyspace Specification
18-0-1	Required	The C library shall not be used.	
18-0-2	Required	The library functions atof, atoi and atol from library <stdlib> shall not be used.	
18-0-3	Required	The library functions abort, exit, getenv and system from library <stdlib> shall not be used.	The option <code>-dialect iso</code> must be used to detect violations, for example, <code>exit</code> .
18-0-4	Required	The time handling functions of library <ctime> shall not be used.	

N.	Category	MISRA Definition	Polyspace Specification
18-0-5	Required	The unbounded functions of library <code><cstring></code> shall not be used.	
18-2-1	Required	The macro <code>offsetof</code> shall not be used.	
18-4-1	Required	Dynamic heap memory allocation shall not be used.	
18-7-1	Required	The signal handling facilities of <code><csignal></code> shall not be used.	

Diagnostic Library

N.	Category	MISRA Definition	Polyspace Specification
19-3-1	Required	The error indicator <code>errno</code> shall not be used.	

Input/output Library

N.	Category	MISRA Definition	Polyspace Specification
27-0-1	Required	The stream input/output library <code><cstdio></code> shall not be used.	

Unsupported MISRA C++ Rules

- “Language Independent Issues” on page 2-90
- “General” on page 2-91
- “Lexical Conventions” on page 2-91
- “Standard Conversions” on page 2-92
- “Expressions” on page 2-92
- “Declarations” on page 2-92
- “Classes” on page 2-93
- “Templates” on page 2-93
- “Exception Handling” on page 2-94

- “Preprocessing Directives” on page 2-94
- “Library Introduction” on page 2-94

Language Independent Issues

N.	Category	MISRA Definition	Polyspace Specification
0-1-3	Required	A project shall not contain unused variables.	
0-1-4	Required	A project shall not contain non-volatile POD variables having only one use.	
0-1-5	Required	A project shall not contain unused type declarations.	
0-1-6	Required	A project shall not contain instances of non-volatile variables being given values that are never subsequently used.	
0-1-8	Required	All functions with void return type shall have external side effects.	
0-1-9	Required	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	Required	There shall be no unused parameters (named or unnamed) in nonvirtual functions.	
0-1-12	Required	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	Required	An object shall not be assigned to an overlapping object.	
0-3-1	Required	Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/	

N.	Category	MISRA Definition	Polyspace Specification
		techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.	
0-3-2	Required	If a function generates error information, then that error information shall be tested.	
0-4-1	Document	Use of scaled-integer or fixed-point arithmetic shall be documented.	
0-4-2	Document	Use of floating-point arithmetic shall be documented.	
0-4-3	Document	Floating-point implementations shall comply with a defined floating-point standard.	

General

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

Lexical Conventions

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

Standard Conversions

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

Expressions

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

Declarations

N.	Category	MISRA Definition	Polyspace Specification
7-1-1	Required	A variable which is not modified shall be const qualified.	
7-1-2	Required	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	Polyspace Specification
7-2-1	Required	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	
7-4-1	Document	All usage of assembler shall be documented.	

Classes

N.	Category	MISRA Definition	Polyspace Specification
9-3-3	Required	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	Document	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	Polyspace Specification
14-5-1	Required	A non-member generic function shall only be declared in a namespace that is not an associated namespace.	
14-7-1	Required	All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.	
14-7-2	Required	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.	Category	MISRA Definition	Polyspace Specification
15-0-1	Document	Exceptions shall only be used for error handling.	
15-1-1	Required	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	
15-3-1	Required	Exceptions shall be raised only after start-up and before termination of the program.	
15-3-4	Required	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	
15-5-3	Required	The terminate() function shall not be called implicitly.	

Preprocessing Directives

N.	Category	MISRA Definition	Polyspace Specification
16-6-1	Document	All uses of the #pragma directive shall be documented.	

Library Introduction

N.	Category	MISRA Definition	Polyspace Specification
17-0-3	Required	The names of standard library functions shall not be overridden.	
17-0-4	Required	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 Joint Strike Fighter program. They are designed to improve the robustness of C++ code, and improve maintainability.

5

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 Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program.

5. JSF and Joint Strike Fighter are registered trademarks of Lockheed Martin.

JSF C++ Coding Rules

In this section...
“Supported JSF C++ Coding Rules” on page 2-96
“Unsupported JSF++ Rules” on page 2-119

Supported JSF C++ Coding Rules

- “Code Size and Complexity” on page 2-97
- “Environment” on page 2-97
- “Libraries” on page 2-98
- “Pre-Processing Directives” on page 2-98
- “Header Files” on page 2-100
- “Style” on page 2-100
- “Classes” on page 2-104
- “Namespaces” on page 2-108
- “Templates” on page 2-108
- “Functions” on page 2-108
- “Comments” on page 2-109
- “Declarations and Definitions” on page 2-109
- “Initialization” on page 2-110
- “Types” on page 2-111
- “Constants” on page 2-111
- “Variables” on page 2-111
- “Unions and Bit Fields” on page 2-112
- “Operators” on page 2-112
- “Pointers and References” on page 2-113
- “Type Conversions” on page 2-114
- “Flow Control Standards” on page 2-115
- “Expressions” on page 2-117
- “Memory Allocation” on page 2-118

- “Fault Handling” on page 2-118
- “Portable Code” on page 2-118

Code Size and Complexity

N.	JSF++ Definition	Polyspace Specification
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	Polyspace Specification
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 <code>-dialect iso</code> .
13	Multi-byte characters and wide string literals will not be used.	Report L 'c', 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	Polyspace Specification
17	The error indicator <code>errno</code> shall not be used.	<code>errno</code> should not be used as a macro or a global with external "C" linkage.
18	The macro <code>offsetof</code> , in library <code><stddef.h></code> , shall not be used.	<code>offsetof</code> should not be used as a macro or a global with external "C" linkage.
19	<code><locale.h></code> and the <code>setlocale</code> function shall not be used.	<code>setlocale</code> and <code>localeconv</code> should not be used as a macro or a global with external "C" linkage.
20	The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.	<code>setjmp</code> and <code>longjmp</code> should not be used as a macro or a global with external "C" linkage.
21	The signal handling facilities of <code><signal.h></code> shall not be used.	<code>signal</code> and <code>raise</code> should not be used as a macro or a global with external "C" linkage.
22	The input/output library <code><stdio.h></code> shall not be used.	all standard functions of <code><stdio.h></code> should not be used as a macro or a global with external "C" linkage.
23	The library functions <code>atof</code> , <code>atoi</code> and <code>atol</code> from library <code><stdlib.h></code> shall not be used.	<code>atof</code> , <code>atoi</code> and <code>atol</code> should not be used as a macro or a global with external "C" linkage.
24	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> from library <code><stdlib.h></code> shall not be used.	<code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> should not be used as a macro or a global with external "C" linkage.
25	The time handling functions of library <code><time.h></code> shall not be used.	<code>clock</code> , <code>difftime</code> , <code>mktime</code> , <code>asctime</code> , <code>ctime</code> , <code>gmtime</code> , <code>localtime</code> and <code>strftime</code> should not be used as a macro or a global with external "C" linkage.

Pre-Processing Directives

N.	JSF++ Definition	Polyspace Specification
26	Only the following preprocessor directives shall be used: <code>#ifndef</code> , <code>#define</code> , <code>#endif</code> , <code>#include</code> .	

N.	JSF++ Definition	Polyspace Specification
27	<code>#ifndef</code> , <code>#define</code> and <code>#endif</code> 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 <code>#if !defined</code> , <code>#pragma once</code> , <code>#ifdef</code> , and missing <code>#define</code> .
28	The <code>#ifndef</code> and <code>#endif</code> 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 <code>#ifndef</code> .
29	The <code>#define</code> 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: <ul style="list-style-type: none"> • 29.1 : The <code>#define</code> preprocessor directive shall not be used to create inline macros. • 29.2 : Inline functions shall be used instead of inline macros.
30	The <code>#define</code> preprocessor directive shall not be used to define constant values. Instead, the <code>const</code> qualifier shall be applied to variable declarations to specify constant values.	Reports <code>#define</code> of simple constants.
31	The <code>#define</code> preprocessor directive will only be used as part of the technique to prevent multiple inclusions of the same header file.	Detects use of <code>#define</code> that are not used to guard for multiple inclusion, assuming that rules 35 and 27 are not violated.
32	The <code>#include</code> preprocessor directive will only be used to include header (*.h) files.	

Header Files

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

Style

N.	JSF++ Definition	Polyspace Specification
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. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
41	Source lines will be kept to a length of 120 characters or less.	
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 <code>if/else</code> , <code>do/while</code> , <code>for</code> , and <code>while</code> 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.	

N.	JSF++ Definition	Polyspace Specification
47	Identifiers will not begin with the underscore character '_'.	
48	Identifiers will not differ by: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • Identifier Idf1 (<i>file1.cpp line 11 column c1</i>) and Idf2 (<i>file2.cpp line 12 column c2</i>) only differ by the presence/absence of the underscore character. • Identifier Idf1 (<i>file1.cpp line 11 column c1</i>) and Idf2 (<i>file2.cpp line 12 column c2</i>) only differ by a mixture of case. • Identifier Idf1 (<i>file1.cpp line 11 column c1</i>) and Idf2 (<i>file2.cpp line 12 column c2</i>) only differ by letter O, with the number 0.
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: <ul style="list-style-type: none"> • 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. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

N.	JSF++ Definition	Polyspace Specification
51	All letters contained in function and variables names will be composed entirely of lowercase letters.	<p>Messages in report file:</p> <ul style="list-style-type: none"> • 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.	<p>Messages in report file:</p> <ul style="list-style-type: none"> • 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 <code>-dos</code> .
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 <code>-dos</code> .
57	The public, protected, and private sections of a class will be declared in that order.	
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.

N.	JSF++ Definition	Polyspace Specification
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.	<p>Messages in report file:</p> <ul style="list-style-type: none"> • 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.	
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.

N.	JSF++ Definition	Polyspace Specification
63	Spaces will not be used around ‘.’ or ‘->’, nor between unary operators and operands.	<p>Reports when the following characters are not directly connected to a white space:</p> <ul style="list-style-type: none"> • . • -> • ! • ~ • - • ++ • — <hr/> <p>Note: A violation will be reported for “.” used in float/double definition.</p>

Classes

N.	JSF++ Definition	Polyspace Specification
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.
74	Initialization of nonstatic class members will be performed through the member initialization list rather than through assignment in the body of a constructor.	<p>All data should be initialized in the initialization list except for array. Does not report that an assignment exists in ctor body.</p> <p>Message in report file:</p> <p>Initialization of nonstatic class members "<i><field></i>" will be performed through the member initialization list.</p>

N.	JSF++ Definition	Polyspace Specification
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.	<p>Messages in report file:</p> <ul style="list-style-type: none"> • no copy constructor and no copy assign • no copy constructor • no copy assign <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
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.	<p>Reports when the number of “new” called in a constructor is greater than the number of “delete” called in its destructor.</p> <hr/> <p>Note: A violation is raised even if “new” is done in a “if/else”.</p>

N.	JSF++ Definition	Polyspace Specification
81	The assignment operator shall handle self-assignment correctly	<p>Reports when copy assignment body does not begin with “if (this != arg)”</p> <p>A violation is not raised if an empty else statement follows the if, or the body contains only a return statement.</p> <p>A violation is raised when the if statement is followed by a statement other than the return statement.</p>
82	An assignment operator shall return a reference to *this .	<p>The following operators should return *this on method, and *first_arg on plain function.</p> <p>operator=operator+=operator- =operator*=operator >=operator <<=operator /=operator %=operator =operator &=operator ^=prefix operator++ prefix operator--</p> <p>Does not report when no return exists.</p> <p>No special message if type does not match.</p> <p>Messages in report file:</p> <ul style="list-style-type: none"> • An assignment operator shall return a reference to *this. • An assignment operator shall return a reference to its first arg.
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.

N.	JSF++ Definition	Polyspace Specification
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: <ul style="list-style-type: none"> • Multiple inheritance on public implementation shall not be allowed: <code><public_base_class></code> is not an interface. • Multiple inheritance on protected implementation shall not be allowed : <code><protected_base_class_1></code>. • <code><protected_base_class_2></code> 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.
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 Specification
98	Every nonlocal name, except <code>main()</code> , should be placed in some namespace.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
99	Namespaces will not be nested more than two levels deep.	

Templates

N.	JSF++ Definition	Polyspace Specification
104	A template specialization shall be declared before its use.	Reports the actual compilation error message.

Functions

N.	JSF++ Definition	Polyspace Specification
107	Functions shall always be declared at file scope.	
108	Functions with variable numbers of arguments shall not be used.	
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 <code>sizeof <= 2 * sizeof(int)</code> . Does not report for copy-constructor.

N.	JSF++ Definition	Polyspace Specification
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 Specification
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 Specification
135	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
136	Declarations should be at the smallest feasible scope.	Reports when: <ul style="list-style-type: none"> • A global variable is used in only one function. • A local variable is not used in a statement (<code>expr</code>, <code>return</code>, <code>init ...</code>) of the same level of its declaration (in the

N.	JSF++ Definition	Polyspace Specification
		<p>same block) or is not used in two sub-statements of its declaration.</p> <hr/> <p>Note:</p> <ul style="list-style-type: none"> • Non-used variables are reported. • Initializations at definition are ignored (not considered an access)
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 all translation units.
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 Specification
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 Specification
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 <code>void*</code>).
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 Specification
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.	<p>Reports direct numeric constants (except integer/float value 1, 0) in expressions, non-<code>const</code> initializations, and switch cases. char constants are allowed. Does not report on templates non-type parameter.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
151.1	A string literal shall not be modified.	<p>Report when a <code>char*</code>, <code>char []</code>, or <code>string</code> type is used not as <code>const</code>.</p> <p>A violation is raised if a string literal (for example, “”) is cast as a non <code>const</code>.</p>

Variables

N.	JSF++ Definition	Polyspace Specification
152	Multiple variable declarations shall not be allowed on the same line.	

Unions and Bit Fields

N.	JSF++ Definition	Polyspace Specification
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 Specification
157	The right hand operand of a <code>&&</code> or <code> </code> operator shall not contain side effects.	Assumes rule 159 is not violated. Messages in report file: <ul style="list-style-type: none"> • The right hand operand of a <code>&&</code> operator shall not contain side effects. • The right hand operand of a <code> </code> operator shall not contain side effects.
158	The operands of a logical <code>&&</code> or <code> </code> shall be parenthesized if the operands contain binary operators.	Messages in report file: <ul style="list-style-type: none"> • The operands of a logical <code>&&</code> shall be parenthesized if the operands contain binary operators. • The operands of a logical <code> </code> shall be parenthesized if the operands contain binary operators. <p>Exception for: <code>X Y Z , Z&&Y &&Z</code></p>
159	Operators <code> </code> , <code>&&</code> , and unary <code>&</code> shall not be overloaded.	Messages in report file: <ul style="list-style-type: none"> • Unary operator <code>&</code> shall not be overloaded. • Operator <code> </code> shall not be overloaded. • Operator <code>&&</code> shall not be overloaded.

N.	JSF++ Definition	Polyspace Specification
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 <code>sizeof</code> 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	Polyspace Specification
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: <ul style="list-style-type: none"> • the same object, • the same function, • members of the same object, or 	Reports when relational operator are used on pointer types (casts ignored).

N.	JSF++ Definition	Polyspace Specification
	<ul style="list-style-type: none"> elements of the same array (including one past the end of the same array). 	
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	Polyspace Specification
177	User-defined conversion functions should be avoided.	<p>Reports user defined conversion function, non-explicit constructor with one parameter or default value for others (even undefined ones).</p> <p>Does not report copy-constructor.</p> <p>Additional message for constructor case:</p> <p>This constructor should be flagged as "explicit".</p>
178	<p>Down casting (casting from base to derived class) shall only be allowed through one of the following mechanism:</p> <ul style="list-style-type: none"> 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.)

N.	JSF++ Definition	Polyspace Specification
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 <code>dynamic_cast</code> .
180	Implicit conversions that may result in a loss of information shall not be used.	<p>Reports the following implicit casts :</p> <pre>integer => smaller integer unsigned => smaller or eq signed signed => smaller or eq un-signed integer => float float => integer</pre> <p>Does not report for cast to <code>bool</code> reports for implicit cast on constant done with the options <code>-scalar-overflows-checks signed-and-unsigned</code> or <code>-ignore-constant-overflows</code></p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
181	Redundant explicit casts will not be used.	Reports useless cast: <code>cast T to T</code> . Casts to equivalent <code>typedefs</code> 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 <code>float->int</code> conversions. Does not report implicit ones.
185	C++ style casts (<code>const_cast</code> , <code>reinterpret_cast</code> , and <code>static_cast</code>) shall be used instead of the traditional C-style casts.	

Flow Control Standards

N.	JSF++ Definition	Polyspace Specification
186	There shall be no unreachable code.	Done with gray checks in the software.

N.	JSF++ Definition	Polyspace Specification
		Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
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).	
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.

N.	JSF++ Definition	Polyspace Specification
199	The increment expression in a <code>for</code> 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 <code>for</code> loops will not be used; a <code>while</code> loop will be used instead.	
201	Numeric variables being used within a <i>for</i> 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 Specification
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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • Variable is written more than once in an expression • Variable is read and write in sub-expressions

N.	JSF++ Definition	Polyspace Specification
		<ul style="list-style-type: none"> • Volatile variable is accessed more than once <hr/> <p>Note: Read-write operations such as ++, are only considered as a write.</p>
205	The volatile keyword shall not be used unless directly interfacing with hardware.	Reports if volatile keyword is used.

Memory Allocation

N.	JSF++ Definition	Polyspace Specification
206	Allocation/deallocation from/to the free store (heap) shall not occur after initialization.	Reports calls to C library functions: <code>malloc</code> / <code>calloc</code> / <code>realloc</code> / <code>free</code> and all <code>new</code> / <code>delete</code> operators in functions or methods.

Fault Handling

N.	JSF++ Definition	Polyspace Specification
208	C++ exceptions shall not be used.	Reports <code>try</code> , <code>catch</code> , <code>throw spec</code> , and <code>throw</code> .

Portable Code

N.	JSF++ Definition	Polyspace Specification
209	The basic types of <code>int</code> , <code>short</code> , <code>long</code> , <code>float</code> and <code>double</code> shall not be used, but specific-length equivalents should be <code>typedef</code> 'd accordingly for each compiler, and these type names used in the code.	Only allows use of basic types through direct <code>typedef</code> s.
213	No dependence shall be placed on C++'s operator precedence rules, below arithmetic operators, in expressions.	<p>Reports when a binary operation has one operand that is not parenthesized and is an operation with inferior precedence level.</p> <p>Reports bitwise and shifts operators that are used without parenthesis and binary operation arguments.</p>

N.	JSF++ Definition	Polyspace Specification
215	Pointer arithmetic will not be used.	Reports: p + Ip - Ip++p - -p+=p -= Allows p[i].

Unsupported JSF++ Rules

- “Code Size and Complexity” on page 2-119
- “Rules” on page 2-120
- “Environment” on page 2-120
- “Libraries” on page 2-120
- “Header Files” on page 2-120
- “Style” on page 2-121
- “Classes” on page 2-121
- “Namespaces” on page 2-123
- “Templates” on page 2-123
- “Functions” on page 2-123
- “Comments” on page 2-124
- “Initialization” on page 2-124
- “Types” on page 2-125
- “Unions and Bit Fields” on page 2-125
- “Operators” on page 2-125
- “Type Conversions” on page 2-125
- “Expressions” on page 2-125
- “Memory Allocation” on page 2-126
- “Portable Code” on page 2-126
- “Efficiency Considerations” on page 2-126
- “Miscellaneous” on page 2-126
- “Testing” on page 2-126

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: <ul style="list-style-type: none">• 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: <ul style="list-style-type: none">• 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.

N.	JSF++ Definition
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.
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.

N.	JSF++ Definition
72	<p>The invariant for a class should be:</p> <ul style="list-style-type: none"> • 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.
92	<p>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:</p> <ul style="list-style-type: none"> • 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. <p>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.</p>
93	“has-a” or “is-implemented-in-terms-of” relationships will be modeled through membership or non-public inheritance.

Namespaces

N.	JSF++ Definition
100	<p>Elements from a namespace should be selected as follows:</p> <ul style="list-style-type: none"> • using declaration or explicit qualification for few (approximately five) names, • using directive for many names.

Templates

N.	JSF++ Definition
101	<p>Templates shall be reviewed as follows:</p> <ol style="list-style-type: none"> 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.
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	<p>Arguments should be passed by reference if NULL values are not possible:</p> <ul style="list-style-type: none"> • 117.1 – An object should be passed as <code>const T&</code> if the function should not change the value of the object. • 117.2 – An object should be passed as <code>T&</code> if the function may change the value of the object.
118	<p>Arguments should be passed via pointers if NULL values are possible:</p> <ul style="list-style-type: none"> • 118.1 – An object should be passed as <code>const T*</code> if its value should not be modified. • 118.2 – An object should be passed as <code>T*</code> if its value may be modified.

N.	JSF++ Definition
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 parameters.
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: <ol style="list-style-type: none"> 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.

N.	JSF++ Definition
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 3-2
- “Select Specific MISRA or JSF Coding Rules” on page 3-6
- “Create Custom Coding Rules” on page 3-9
- “Format of Custom Coding Rules File” on page 3-11
- “Exclude Files From Analysis” on page 3-12
- “Allow Custom Pragma Directives” on page 3-13
- “Specify Boolean Types” on page 3-14
- “Find Coding Rule Violations” on page 3-15
- “Review Coding Rule Violations” on page 3-16
- “Filter and Group Coding Rule Violations” on page 3-18
- “Rules to Disable for Faster Analysis” on page 3-21

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 On the **Configuration** pane, select **Coding Rules & Code Metrics**.
- 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
- MISRA C:2012

If you have generated code, use the **Use generated code requirements** option to use the MISRA C:2012 categories for generated code.

- Custom coding rules

For C++ code, you can check compliance with:

- MISRA C++: 2008
- 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.

Checking for certain rules can cause the analysis to run longer than usual. For more information, see “Rules to Disable for Faster Analysis” on page 3-21.

MISRA C:2004

Option	Description
required-rules	All required MISRA C:2004 coding rules.
all-rules	All MISRA C:2004 coding rules (required and advisory).

Option	Description
SQO-subset1	A small subset of MISRA C:2004 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:2004 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.
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:2012

Option	Description
mandatory	All mandatory MISRA C:2012 coding rules. If you have generated code, also use the Use generated code requirements option categorization for generated code.

Option	Description
mandatory-required	All mandatory and required MISRA C:2012 coding rules. If you have generated code, also use the Use generated code requirements option categorization for generated code.
all	All MISRA C:2012 coding rules (mandatory, required, and advisory).
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:2012 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.

Option	Description
shall-will-rules	All Shall and Will rules. Will rules are intended to be mandatory requirements. However, these rules do not require verification.
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 3-6
- “Create Custom Coding Rules” on page 3-9
- “Exclude Files From Analysis” on page 3-12

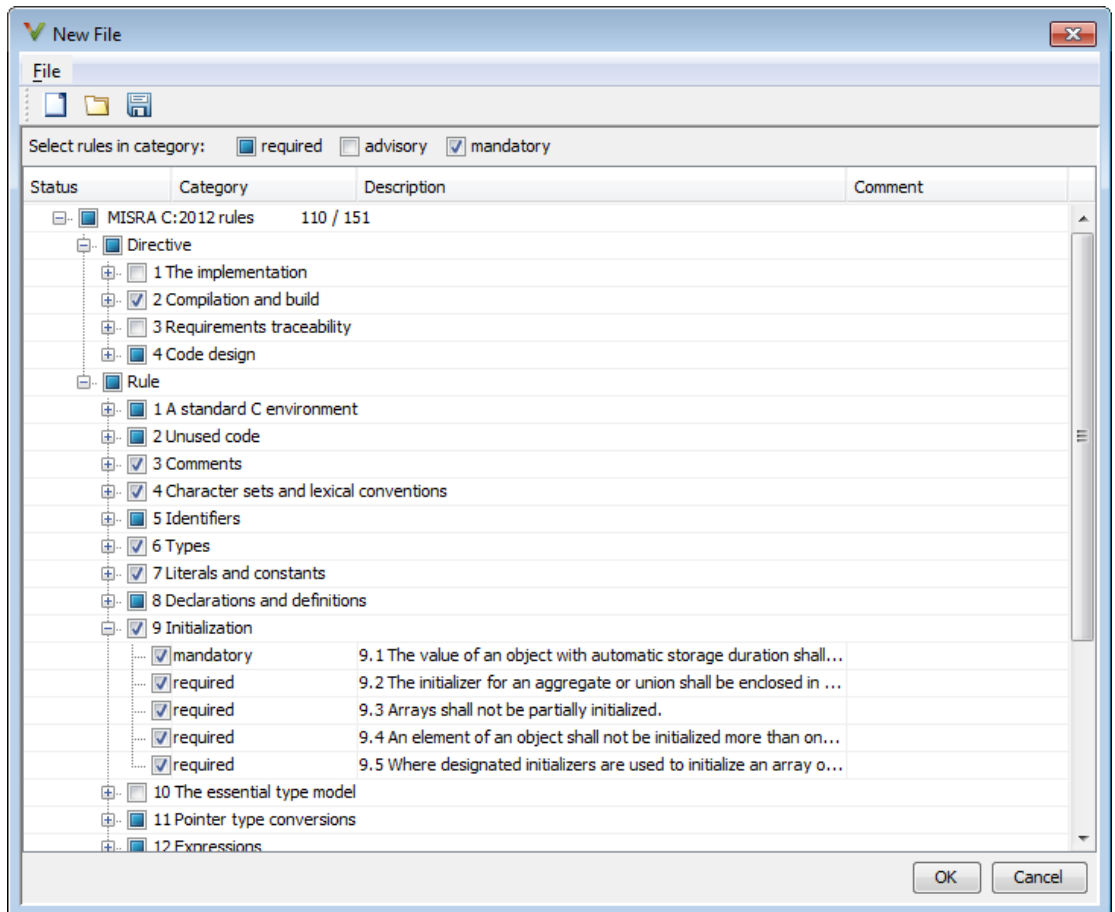
More About


- “Rule Checking” on page 2-2

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 the project configuration.
- 2 In the **Configuration** tree view, select **Coding Rules & Code Metrics**.
- 3 Select the check box for the type of coding rules you want 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.



- 6 If you already have a customized rule file you want to edit, reload your customization using the  button.
- 7 Select the rules you want to check.

You can select categories of rules (required, advisory, mandatory), subsets of rules by rule chapter, or individual rules.

- 8 When you are finished, click **OK**.

- 9 For new files, use the Save As dialog box that opens to save your customization as a rules file.
- 10 In the Configuration window, the full path to the rules file appears in the `custom` field. To reuse this customized set of rules for other projects, enter this path name in the dialog box.

Related Examples

- “Activate Coding Rules Checker” on page 3-2
- “Create Custom Coding Rules” on page 3-9


More About

- “Rule Checking” on page 2-2

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.

1 Create Coding Rules File

- 1 Create a Polyspace project. Add `printInitialValue.c` to the project.
- 2 On the **Configuration** pane, select **Coding Rules & Code Metrics**. Select the **Check custom rules** box.
- 3 Click .

The New File window opens, displaying a table of rule groups.

- 4 Clear the **Custom rules** check box to turn off checking of all custom rules.
- 5 Expand the 4 **Structs** node. For the option **4.3 All struct fields must follow the specified pattern**:

Column Title	Action
Status	Select <input checked="" type="checkbox"/> .
Convention	Enter All struct fields must begin with s_ and have capital letters or digits
Pattern	Enter <code>s_[A-Z0-9_]+</code>
Comment	Leave blank. This column is for comments that appear in the coding rules file alone.

2 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 **Result 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 Editor**. The file `printInitialValue.c` opens in the **Code Editor** pane or an external text editor depending on your **Preferences**.
- 3** In the file, replace all instances of `a` with `s_A` and `b` with `s_B`. Rerun the verification.

The custom rule violations no longer appear on the **Results Summary** pane.

Related Examples

- “Activate Coding Rules Checker” on page 3-2
- “Select Specific MISRA or JSF Coding Rules” on page 3-6
- “Exclude Files From Analysis” on page 3-12

More About

- “Rule Checking” on page 2-2
- “Format of Custom Coding Rules File” on page 3-11

Format of Custom Coding Rules File

In a custom coding rules file, each rule appears in the following format:

```
N.n off|on
convention=violation_message
pattern=regular_expression
```

- *N.n* — Custom rule number, for example, 1.2.
- `off` — Rule is not considered.
- `on` — 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 Coding Rules”.

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 on          # 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 on          # 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 3-9

Exclude Files From Analysis

This example shows how to exclude certain files from defect and coding rules checking.

- 1 Open the project configuration.
- 2 In the **Configuration** tree view, select **Inputs & Stubbing**.
- 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) — Excludes header files in the Include folders of your project. For example `.h` or `.hpp` files.
 - **all** — Excludes all include files in the Include folders of your project. 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** — 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 3-2

More About

- “Rule Checking” on page 2-2

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 & Code Metrics**.
- 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 3-2

More About

- “Rule Checking” on page 2-2

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 is related to checking of the following rules:


- MISRA C:2004 and MISRA AC AGC —12.6, 13.2, 15.4.

For more information, see “MISRA C:2004 and MISRA AC AGC Coding Rules” on page 2-14.

- MISRA C:2012 — 10.1, 10.3, 10.5, 14.4 and 16.7


1 Open project configuration.

2 In the **Configuration** tree view, select **Coding Rules & Code Metrics**.

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 3-2

More About

- “Rule Checking” on page 2-2

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 & Code Metrics**. Activate the desired coding rule checker.

For more information, see “Activate Coding Rules Checker” on page 3-2.

- 3 Checking for certain rules can cause the analysis to run longer than usual. Disable those rules if you want.

For more information, see “Rules to Disable for Faster Analysis” on page 3-21.

- 4 Specify that the analysis must not look for defects.
 - In the **Configuration** tree view, select **Bug Finder Analysis**.
 - Clear the **Find defects** check box.
- 5 Click  to run the coding rules checker without checking defects.

Related Examples

- “Activate Coding Rules Checker” on page 3-2
- “Select Specific MISRA or JSF Coding Rules” on page 3-6
- “Review Coding Rule Violations” on page 3-16

More About

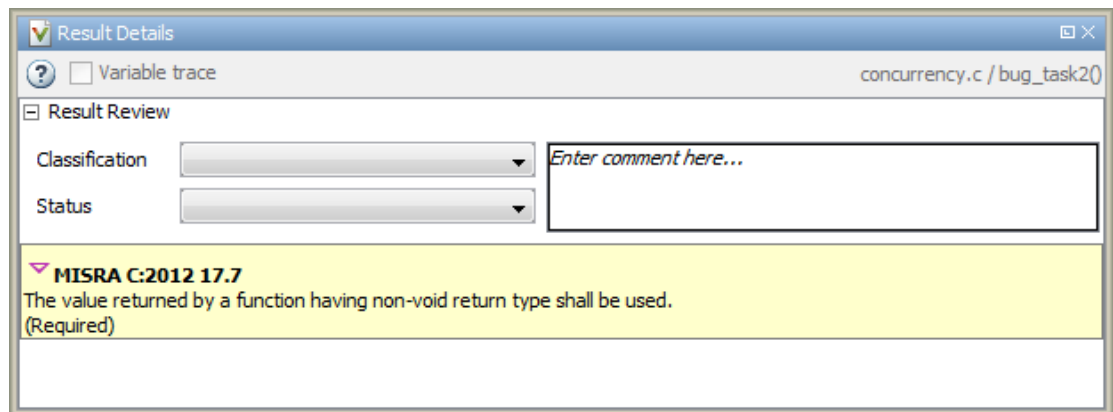
- “Rule Checking” on page 2-2


Review Coding Rule Violations

This example shows how to review coding rule violations once code analysis is complete. After analysis, the **Results Summary** pane displays the rule violations with a

- ▾ symbol for predefined coding rules, MISRA or JSF.
- ▾ symbol for custom coding rules.

- 1 Select a coding-rule violation on the **Results Summary** pane.
- 2 On the **Result Details** pane, view the location and description of the violated rule. In the source code, the line containing the violation appears highlighted.



- 3 For MISRA C: 2012 rules, on the **Result Details** pane, click the  icon to see the rationale for the rule. In some cases, you can also see code examples illustrating the violation.
- 4 Review the violation in your code.
 - a Determine whether you must fix the code to avoid the violation.
 - b If you choose to retain the code, on the **Result Details** pane, add a comment explaining why you retain the code. This comment helps you or other reviewers avoid reviewing the same coding rule violation twice.

You can also assign a **Severity** and **Status** to the coding rule violation.

- 5 After you have fixed or justified the coding rule violations, run the analysis again.

Related Examples

- “Activate Coding Rules Checker” on page 3-2
- “Find Coding Rule Violations” on page 3-15
- “Filter and Group Coding Rule Violations” on page 3-18

Filter and Group Coding Rule Violations

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.

In this section...

“Filter Coding Rules” on page 3-18

“Group Coding Rules” on page 3-18

“Suppress Certain Rules from Display in One Click” on page 3-18

Filter Coding Rules

- 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**.

Group Coding Rules

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

The rules are grouped by numbers. Each group corresponds to a certain code construct.

- 2 Expand the group nodes to select an individual coding rule violation.

Suppress Certain Rules from Display in One Click

Instead of filtering individual rules from display each time you open your results, you can limit the display of rule violations in one click. To limit the display of rule violations, use the **Show** menu on the **Results Summary** pane. You can create your own options on this menu. You can share the option file to help developers in your organization review violations of at least certain coding rules.

- 1 In the Polyspace user interface, select **Tools > Preferences**.
- 2 On the **Review Scope** tab, do one of the following:
 - To add predefined options to the **Show** menu, select **Include Quality Objectives Scopes**.

The **Scope Name** list shows additional options, **HIS**, **SQ0-4**, **SQ0-5**, and **SQ0-6**. Select an option to see which rules are suppressed from display.

In addition to coding rule violations, the options impose limits on the display of code metrics and defects.

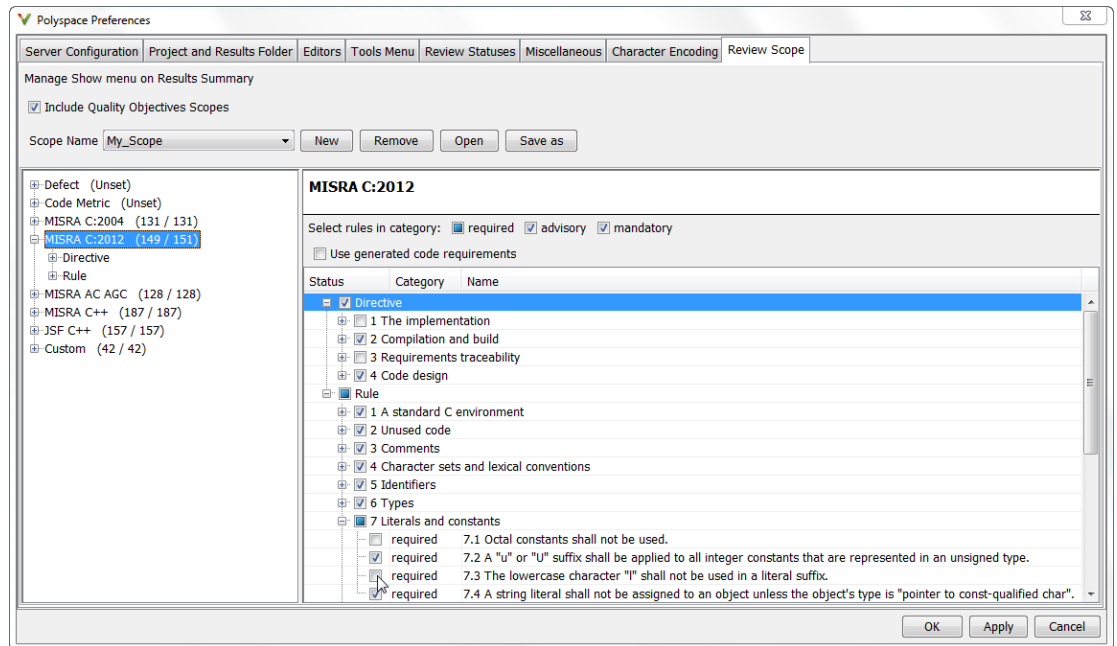
- To create your own option on the **Show** menu, select **New**. Save your option file.

On the left pane, select a rule set such as **MISRA C:2012**. On the right pane, to suppress a rule from display, clear the box next to the rule.

To suppress all rules belonging to a group such as **The essential type model**, clear the box next to the group name. For more information on the groups, see “Coding Rules”. If only a fraction of rules in a group is selected, the check box next to the group name displays a symbol.

To suppress all rules belonging to a category such as **advisory**, clear the box next to the category name on the top of the right pane. If only a fraction of rules in a category is selected, the check box next to the category name displays a symbol.

3 Check Coding Rules from the Polyspace Environment



3 Select **Apply** or **OK**.

On the **Results Summary** pane, the **Show** menu displays the additional options.

4 Select the option that you want. The rules that you suppress do not appear on the **Results Summary** pane.

Related Examples

- “Activate Coding Rules Checker” on page 3-2
- “Review Coding Rule Violations” on page 3-16

Rules to Disable for Faster Analysis

Checking for the following coding rules can cause the analysis to run longer than usual. To check these rules, Polyspace Bug Finder must check for certain defects, too.

For faster analysis, you can disable the checking of these rules if you want. For more information, see “Select Specific MISRA or JSF Coding Rules” on page 3-6.

MISRA C: 2004 and MISRA AC AGC

Rule	Definition
MISRA C: 2004 Rule 9.1 MISRA AC AGC Rule 9.1	All automatic variables shall have been assigned a value before being used.
MISRA C: 2004 Rule 21.1 MISRA AC AGC Rule 21.1	Minimization of runtime failures shall be ensured by the use of at least one of: <ul style="list-style-type: none"> • Static verification tools/techniques. • Dynamic verification tools/techniques. • Explicit coding of checks to handle runtime faults.

For more information, see “MISRA C:2004 and MISRA AC AGC Coding Rules” on page 2-14.

MISRA C: 2012

Rule	Definition
MISRA C:2012 Directive 4.1	Run-time failures shall be minimized.
MISRA C:2012 Directive 4.13	Functions which are designed to provide operations on a resource should be called in an appropriate sequence.
MISRA C:2012 Rule 2.2	There shall be no dead code.
MISRA C:2012 Rule 9.1	The value of an object with automatic storage duration shall not be read before it has been set.

Rule	Definition
MISRA C:2012 Rule 18.1	A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.
MISRA C:2012 Rule 22.1	All resources obtained dynamically by means of Standard Library functions shall be explicitly released.
MISRA C:2012 Rule 22.2	A block of memory shall only be freed if it was allocated by means of a Standard Library function.
MISRA C:2012 Rule 22.3	The same file shall not be open for read and write access at the same time on different streams.
MISRA C:2012 Rule 22.4	There shall be no attempt to write to a stream which has been opened as read-only.
MISRA C:2012 Rule 22.6	The value of a pointer to a FILE shall not be used after the associated stream has been closed.

Find Bugs From the Polyspace Environment

- “Choose Specific Defects” on page 4-2
- “Run Local Analysis” on page 4-3
- “Run Remote Batch Analysis” on page 4-4
- “Monitor Analysis” on page 4-5
- “Specify Results Folder” on page 4-6


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.

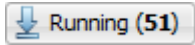

- 1 On the **Configuration** pane, select **Bug Finder Analysis**.
- 2 From the **Find defects** menu, select a set of defects. The options are:
 - **default** for the default list of defects. This list contains defects that are applicable to most coding projects. To see the defects in the default list, expand the nodes.
 - **all** for all defects.
 - **custom** to add defects to the default list or remove defects from it.

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 1-6.

- 1 Select a project to analyze.
- 2 Click the  button.
- 3 Monitor the analysis on the **Output Summary** tab.

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. Files with compilation problems do not appear in the results.

- 4 Once some results are available, start reviewing the results by selecting the link in the Output Summary window or by clicking the  button on the toolbar. This button reactivates as more results are ready.
- 5 If you viewed some of the results during the analysis, click the toolbar button  to load the rest of your results.

If you did not load results during the analysis, the **Results Summary** tab automatically opens with your completed results.

Related Examples


- “Run Remote Batch Analysis” on page 4-4
- “Create New Project” on page 1-6
- “Open Results” on page 5-2
- “Review and Fix Results” on page 5-24

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 1-6 and “Set Up Polyspace Metrics”.

- 1 Select a project to analyze.
- 2 On the **Configuration** pane, select **Distributed Computing**.
- 3 Select **Batch**.
- 4 If you want to store your results in the Polyspace Metrics repository, select **Add to results repository**.

Otherwise, clear this check box.

- 5 Select the  button.
- 6 To monitor the analysis, select **Tools > Open Job Monitor**.

Once the analysis is complete, you can open your results from the Results folder, or download them from Polyspace Metrics.

Related Examples

- “Open Results” on page 5-2
- “Download Results From Polyspace Metrics” on page 5-6

Monitor Analysis

To monitor the progress of a local analysis, use the following panes in the Polyspace Bug Finder interface. To open or close one of the tabs, select **Window > Show/Hide View**.

- **Output Summary** — Displays progress of verification, 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.
- **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.

At the end of a local analysis, the **Dashboard** tab displays statistics, for example, code coverage and check distribution.

To monitor the progress of a remote analysis:

- 1 From the Polyspace interface, select **Tools > Open Job Monitor**.
- 2 In the Polyspace Job Monitor, follow your queued job to monitor 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 On the **Project Browser**, right-click the Results folder.
- 2 Select **Choose a Result Folder**.
- 3 In the Choose a Result Folder window, navigate to the new results folder and click **Select**.

On 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 Result**.

View Results in the Polyspace Environment

- “Open Results” on page 5-2
- “View Results Summary in Polyspace Metrics” on page 5-4
- “Download Results From Polyspace Metrics” on page 5-6
- “Filter and Group Results” on page 5-9
- “Classification of Defects by Impact” on page 5-12
- “Limit Display of Defects” on page 5-20
- “Generate Reports” on page 5-22
- “Review and Fix Results” on page 5-24
- “Review Concurrency Defects” on page 5-27
- “Review Code Metrics” on page 5-30
- “Navigate to Root Cause of Defect” on page 5-34
- “Results Folder Contents” on page 5-37
- “Windows Used to Review Results” on page 5-38
- “Bug Finder Defect Groups” on page 5-52
- “HIS Metrics” on page 5-57
- “Common Weakness Enumeration from Bug Finder Defects” on page 5-59
- “Find CWE Identifiers from Defects” on page 5-61
- “Mapping Between CWE Identifiers and Defects” on page 5-63

Open Results

This example shows how to open Polyspace Bug Finder results. Before you open the results, you must start a Polyspace Bug Finder analysis on your source files. The analysis produces a results file with the extension `.psbf`.

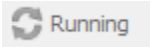
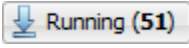
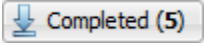
In this section...

“Open Results From Active Project” on page 5-2

“Open Results File From File Browser” on page 5-2

Open Results From Active Project

Suppose that you have a project called `Bug_Finder_Example` open in the **Project Browser**. After an analysis, the results appear under the project as `Result_Bug_Finder_Example`. While a local analysis is running, you can start reviewing your results in real time. After you start a local analysis, a button appears on the toolbar to show you the status of the analysis:

-  — The analysis is running. No results to load.
-  — The analysis is running and new results are available to start reviewing. Click this button to load the new results in the Results Summary. This button reactivates every time more results are available.
-  — The analysis is complete, but you have not loaded all results. Click this button to load the last set of results.

If you do not view partial results during the analysis, at the end of the analysis, your results open automatically. To manually open results, double-click `Result_Bug_Finder_Example`.

Open Results File From File Browser

- 1 Select **File > Open**. The Open File browser opens.
- 2 Navigate to the result folder containing the file with extension `.psbf`. For example, navigate to `matlabroot\polyspace\examples\cxx\Bug_Finder_Example\Results\`.

- 3 Select the file. Click **Open**.

More About

- “Results Folder Contents” on page 5-37
- “Windows Used to Review Results” on page 5-38

View Results Summary in Polyspace Metrics

This example shows how to view results summary in Polyspace Metrics. On the **Configuration** pane, under **Distributed Computing**, if you select **Add to results repository**, 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.

The screenshot shows the Polyspace Metrics web interface. At the top, there is a header with "Select Project" and "Polyspace Metrics". Below the header, there is a table with columns for "Project", "Product", "Mode", "Language", and "Date". The table contains several rows of project data. A sidebar on the left shows details for the selected project "bug_finder_project".

Project	Product	Mode	Language	Date
bug_finder_project	Bug Finder	C	C	May 23, 2013
Demo_C	Bug Finder	C	C	May 23, 2013
Demo_C-WF	Bug Finder	C	C	Jun 18, 2013
sf_project	Bug Finder	C	C	May 28, 2013
bug_finder_project	Bug Finder	C	C	Jun 16, 2013
quality_testing	Bug Finder	C	C	Jun 24, 2013
Ch-Testing				

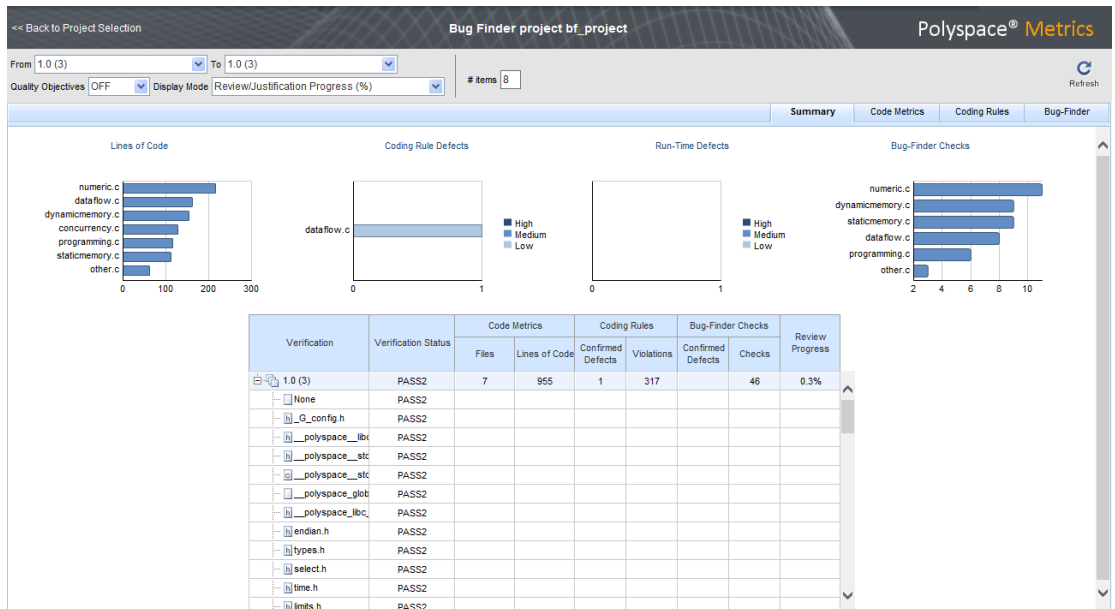
Details for "bug_finder_project":

Language	C
Mode	Integration
Last Run Name	1.0
Number of Runs	1

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.



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”
- “Download Results From Polyspace Metrics” on page 5-6
- “Review and Fix Results” on page 5-24

Download Results From Polyspace Metrics

This example shows how to download results from Polyspace Metrics. On the **Configuration** pane, under **Distributed Computing**, if you select **Add to results repository**, 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)

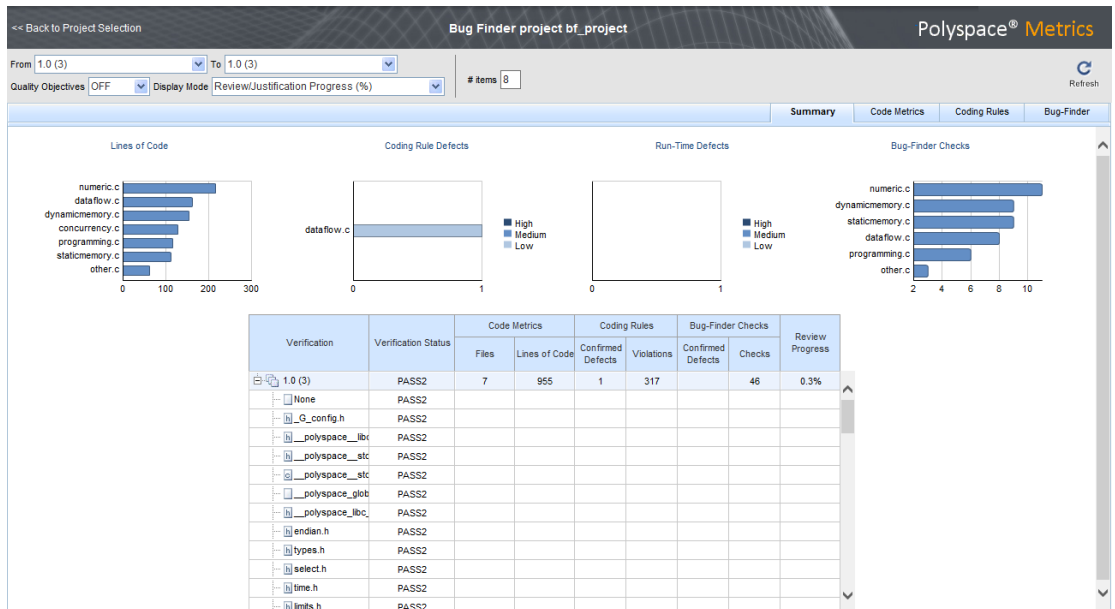
On the webpage, you can view the projects saved to your Polyspace Metrics repository.

Project	Product	Mode	Language	Date
bug_finder_project	Bug Finder	C	C	May 23, 2013
Demo_C	Bug Finder	C	C	May 23, 2013
Demo_C-BF	Bug Finder	C	C	Jun 10, 2013
BF_project	Bug Finder	C	C	May 26, 2013
bug_finder_project	Bug Finder	C	C	Jun 10, 2013
quality_testing	Bug Finder	C	C	Jun 24, 2013
Ch-Testing	Bug Finder	C	C	

Download Results

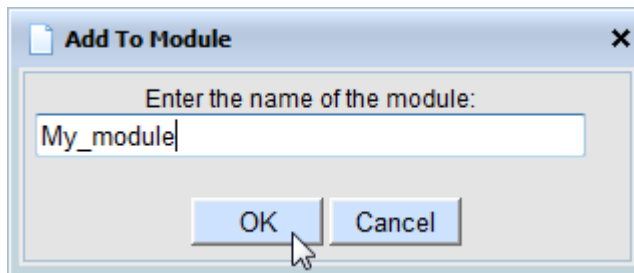
- 1 Select the **Projects** tab.
- 2 To view the results summary for your project, on the **Projects** column, select the project name.

The results summary for the project appears on the webpage under the **Summary** tab.



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 on the **Results Summary** pane in Polyspace Bug Finder. The filter **Show > Web checks** on this pane indicate that you have downloaded the results from Polyspace Metrics.

Related Examples

- “Set Up Polyspace Metrics”
- “View Results Summary in Polyspace Metrics” on page 5-4
- “Review and Fix Results” on page 5-24

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 only high-impact defects.

For more information on impact, see “Classification of Defects by Impact” on page 5-12.

- Review certain types of defects in preference to others.

For instance, you first want to address the defects resulting from **Missing or invalid return statement**.

- Review only new results found since the last analysis.
- 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.

Filter Results

- 1 To review only new results found since the last verification, on the **Results Summary** pane, select **New results**.
- 2 To suppress code metrics from your results, on the **Results Summary** pane, select **Show > Defects & Rules**.

You can increase the options on the **Show** menu or create your own options. For examples, see:

- “Suppress Certain Rules from Display in One Click” on page 3-18
- “Limit Display of Defects” on page 5-20
- “Review Code Metrics” on page 5-30

- 3 For all other filters, click the  icon on the appropriate column.

Item to Filter	Column
Results in a certain file or function	File or Function
Results with a certain severity or status	Severity or Status
Results in a certain group such as numerical or data flow	Group
Results with a certain impact	Information
Results that correspond to certain CWE IDs.	CWE ID For more information, see “Find CWE Identifiers from Defects” on page 5-61.

- 4 Clear **All**. Select the boxes for the results that you want displayed.

Alternatively, clear the boxes for the results that you do not want displayed.

Note: You can also apply multiple filters.

Group Results

On the **Results Summary** pane:

- To show results without grouping, select **Group by > None**.
- To show results grouped by result type, select **Group by > Family**.
 - The defects are organized by the defect groups. For more information on the groups, see “Defects”.
 - The coding rule violations are grouped by type of coding rule. For more information, see “Coding Rules”.
 - The code metrics are grouped by scope of metric. For more information, see “Code Metrics”.
- To show results grouped by file, select **Group by > File**.

Within each file, the results are grouped by function. The results that are not associated with a particular function are grouped under **File Scope**.

- For C++ code, to show results grouped by class, select **Group by > Class**. The results that are not associated with a particular class are grouped under **Global Scope**.

Within each class, the results are grouped by method.

Related Examples

- “Review and Fix Results” on page 5-24

More About

- “Windows Used to Review Results” on page 5-38

Classification of Defects by Impact

To prioritize your review of Polyspace Bug Finder defects, you can use the **Impact** attribute assigned to the defect. This attribute appears on:

- The **Dashboard** pane, in a **Defect distribution by impact** pie chart.

You can view at a glance whether you have many high impact defects.

- The **Results Summary** pane, in the **Information** column. When you select **Group by > None**, the defects are sorted by impact.

You can filter out low and/or medium impact defects using this column or through the **Review Scope** tab in your preferences. For more information, see “Filter and Group Results” on page 5-9.

- The **Result Details** pane, beside the defect name.

The impact is assigned to a defect based on the following considerations:

- Criticality, or whether the defect is likely to cause a code failure.

If a defect is likely to cause a code to fail, it is treated as a high impact defect. If the defect currently does not cause code failure but can cause problems with code maintenance in the future, it is a low impact defect.

- Certainty, or the rate of false positives.

For instance, the defect **Integer division by zero** is a high-impact defect because it is almost certain to cause a code crash. On the other hand, the defect **Dead code** has low impact because by itself, presence of dead code does not cause code failure. However, the dead code can hide other high-impact defects.

You cannot change the impact assigned to a defect.

High Impact Defects

The following list shows the high-impact defects.

Numerical

- Float conversion overflow
- Float division by zero

- Integer conversion overflow
- Integer division by zero
- Invalid use of standard library floating point routine
- Invalid use of standard library integer routine

Static memory

- Array access out of bounds
- Buffer overflow from incorrect string format specifier
- Destination buffer overflow in string manipulation
- Destination buffer underflow in string manipulation
- Invalid use of standard library memory routine
- Invalid use of standard library string routine
- Null pointer
- Pointer access out of bounds
- Pointer or reference to stack variable leaving scope
- Use of path manipulation function without maximum sized buffer checking
- Wrong allocated object size for cast

Dynamic memory

- Deallocation of previously deallocated pointer
- Invalid free of pointer
- Use of previously freed pointer

Data flow

- Non-initialized pointer
- Non-initialized variable

Resource management

- Closing a previously closed resource
- Resource leak
- Use of previously closed resource
- Writing to read-only resource

Programming

- Assertion
- Declaration mismatch
- Invalid use of == operator
- Invalid use of floating point operation
- Invalid use of standard library routine
- Invalid va_list argument
- Possible misuse of sizeof
- Possibly unintended evaluation of expression because of operator precedence rules
- Variable length array with nonpositive size
- Writing to const qualified object
- Wrong type used in sizeof

Concurrency

- Data race
- Deadlock
- Double lock
- Double unlock
- Missing lock
- Missing unlock

Security

- Use of non-secure temporary file

Object Oriented

- Base class assignment operator not called
- Copy constructor not called in initialization list
- Object slicing

Medium Impact Defects

The following list shows the medium-impact defects.

Numerical

- Integer overflow
- Sign change integer conversion overflow

Static memory

- Unreliable cast of function pointer
- Unreliable cast of pointer

Dynamic memory

- Memory leak

Data flow

- Pointer to non-initialized value converted to const pointer
- Unreachable code
- Useless if

Programming

- Bad file access mode or status
- Copy of overlapping memory
- Exception caught by value
- Exception handler hidden by previous handler
- Improper array initialization
- Incorrect pointer scaling
- Invalid assumptions about memory organization
- Invalid use of = operator
- Overlapping assignment
- Standard function call with incorrect arguments
- Use of memset with size argument zero

Concurrency

- Data race including atomic operations

Security

- Deterministic random output from constant seed
- Execution of a binary from a relative path can be controlled by an external actor
- File access between time of check and use (TOCTOU)
- File manipulation after `chroot()` without `chdir("/")`
- Incorrect order of network connection operations
- Load of library from a relative path can be controlled by an external actor
- Mismatch between data length and size
- Predictable random output from predictable seed
- Sensitive data printed out
- Sensitive heap memory not cleared before release
- Uncleared sensitive data in stack
- Unsafe standard encryption function
- Unsafe standard function
- Vulnerable permission assignments
- Vulnerable pseudo-random number generator

Tainted data

- Array access with tainted index
- Command executed from externally controlled path
- Execution of externally controlled command
- Host change using externally controlled elements
- Library loaded from externally controlled path
- Loop bounded with tainted value
- Memory allocation with tainted size
- Tainted sign change conversion
- Tainted size of variable length array
- Use of externally controlled environment variable

Object Oriented

- Base class destructor not virtual

- Incompatible types prevent overriding
- Member not initialized in constructor
- Missing virtual inheritance
- Partial override of overloaded virtual functions
- Return of non const handle to encapsulated data member
- Self assignment not tested in operator

Low Impact Defects

The following list shows the low-impact defects.

Numerical

- Float overflow
- Shift of a negative value
- Shift operation overflow
- Unsigned integer conversion overflow
- Unsigned integer overflow

Static memory

- Arithmetic operation with NULL pointer

Dynamic memory

- Unprotected dynamic memory allocation

Data flow

- Code deactivated by constant false condition
- Dead code
- Missing return statement
- Partially accessed array
- Static uncalled function
- Variable shadowing
- Write without a further read

Programming

- Format string specifiers and arguments mismatch
- Call to memset with unintended value
- Missing null in string array
- Modification of internal buffer returned from nonreentrant standard function
- Qualifier removed in conversion

Security

- Missing case for switch condition
- Umask used with chmod-style arguments
- Use of dangerous standard function
- Vulnerable path manipulation
- Function pointer assigned with absolute address
- Use of obsolete standard function

Tainted data

- Pointer dereference with tainted offset
- Tainted division operand
- Tainted NULL or non-null-terminated string
- Tainted modulo operand
- Tainted string format
- Use of tainted pointer

Good practice

- Delete of void pointer
- Hard coded buffer size
- Hard coded loop boundary
- Large pass-by-value argument
- Line with more than one statement
- Unused parameter
- Use of setjmp/longjmp

Object Oriented

- *this not returned in copy assignment operator
- Missing explicit keyword

Limit Display of Defects

This example shows how to control the number and type of defects displayed on the **Results Summary** pane. To reduce your review effort, you can limit the number of defects to display for certain checks or suppress them altogether.

To prevent the analysis from looking for some defects, see “Choose Specific Defects” on page 4-2.

If you want to change your analysis configuration, you can still change which defects are displayed in your results. There are two ways to filter defects from your results:

- Filter individual defects from display after each run.

For more information, see “Filter and Group Results” on page 5-9.

- Create a set of filters that you can apply in one click.

This example shows the second approach.

1 Select **Tools > Preferences**.

2 On the **Review Scope** tab, create your filter file.

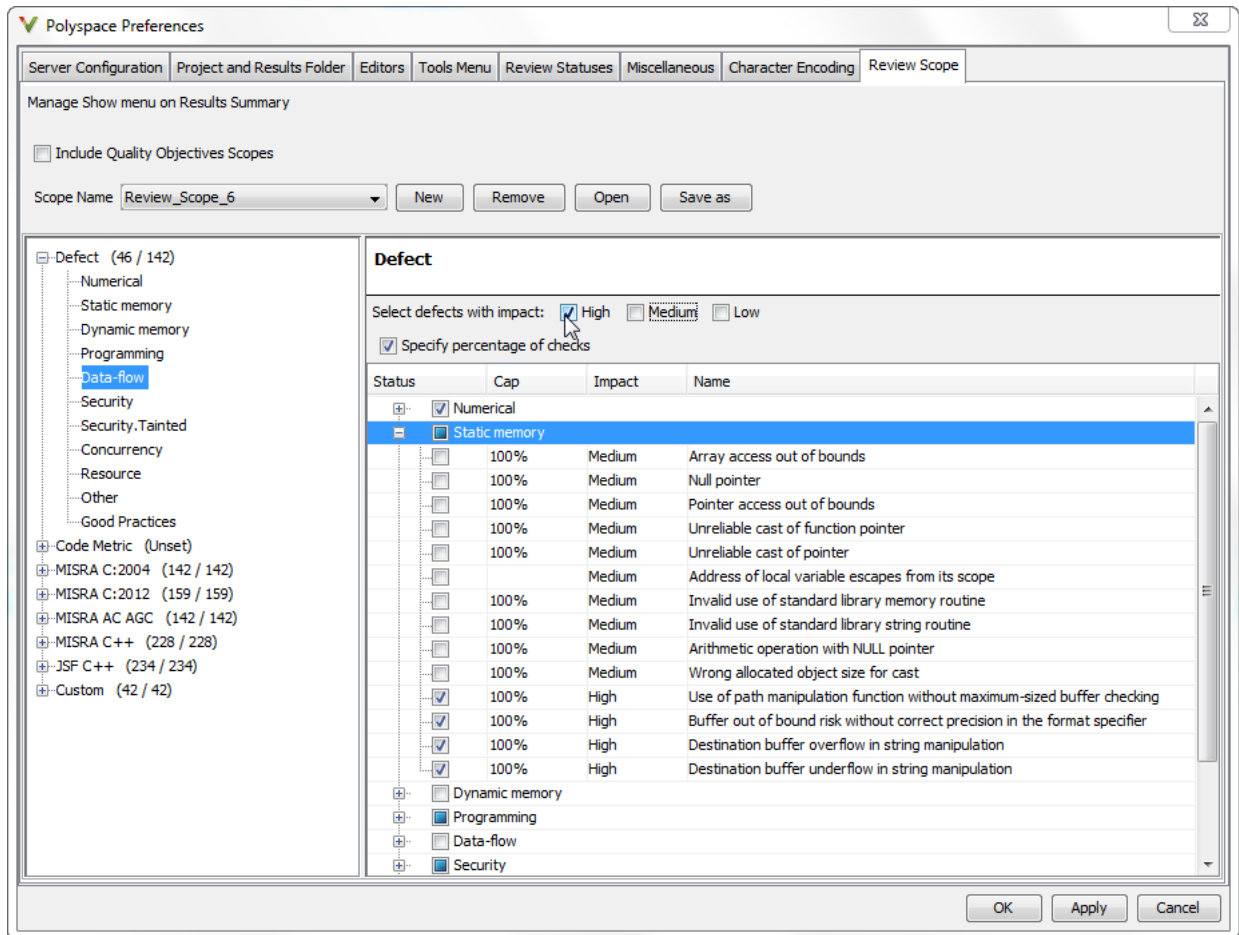
a Select **New**. Save your filter file.

b On the left pane, select **Defect**. On the right pane, to suppress a defect completely, clear the box for the defect. To suppress a defect partly, specify a percentage less than 100 to display.

Instead of a percentage, you can specify a number or the string **ALL**. To specify a number, clear the box **Specify percentage of checks**.

To suppress all defects belonging to a category such as **Numerical**, clear the box next to the category name. For more information on the categories, see “Defects”. If only a fraction of defects in a category are selected, the check box next to the category name displays a symbol.

To suppress all defects with a certain impact such as **Low**, clear the box next to the impact. For more information on impacts, see “Classification of Defects by Impact” on page 5-12. If only a fraction of defects with a certain impact are selected, the check box next to the impact displays a symbol.



3 Select **Apply** or **OK**.

On the **Results Summary** pane, the **Show** menu displays additional options.

4 Select the option corresponding to the filters that you want. Only the number or percentage of defects that you specify remain on the **Results Summary** pane.

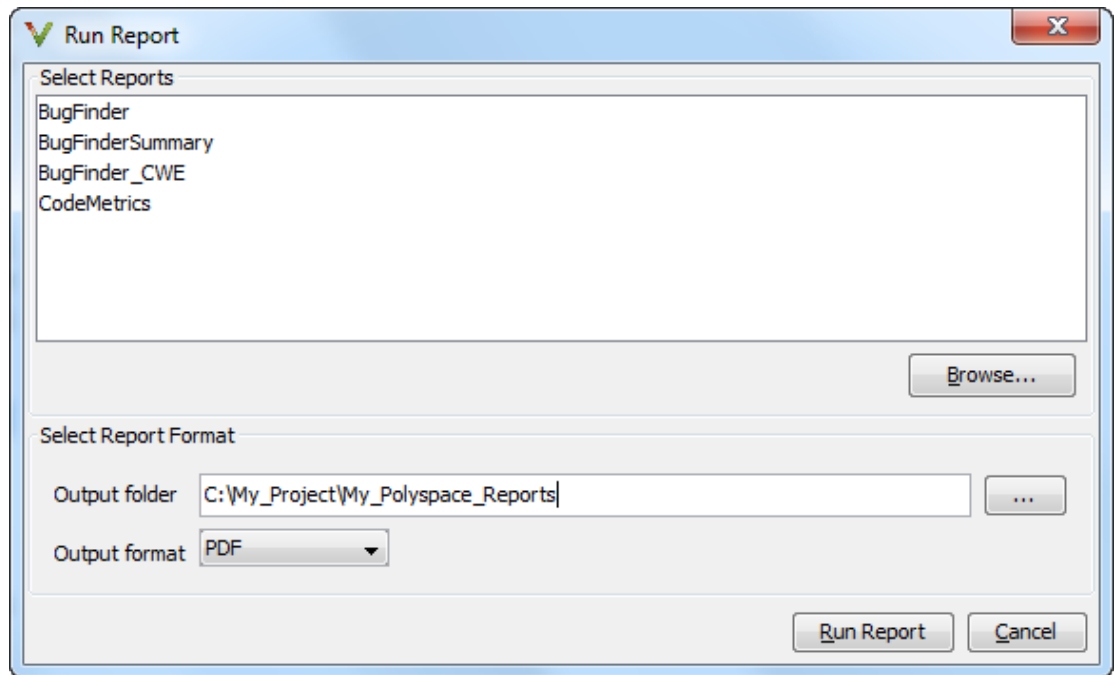
- If you specify an absolute number, Polyspace displays that number of defects.
- If you specify a percentage, Polyspace displays that percentage of the total number of defects.

Generate Reports

This example shows how to generate reports for a Polyspace Bug Finder analysis.

- 1 Open your results file.
- 2 Select **Reporting > Run Report**.

The Run Report dialog box opens.



- 3 In the **Select Reports** section, select the types of reports that you want to generate. Press the **Ctrl** key to select multiple types. For example, you can select **BugFinder** and **CodeMetrics**.
- 4 Select the **Output folder** in which to save the report.
- 5 Select an **Output format** for the report.
- 6 Click **Run Report**.

The software creates the specified report and opens it.

See Also

“Generate report (C/C++)” | “Report template (C/C++)” | “Output format (C/C++)”

Review and Fix Results

This example shows how to review and comment your Bug Finder results. 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.

In this section...

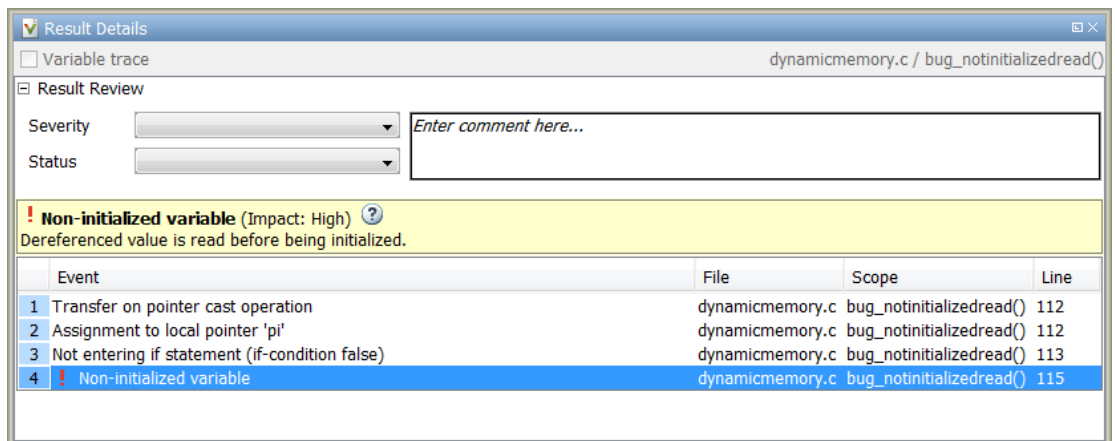
“Assign and Save Comments” on page 5-24

“Import Review Comments from Previous Analysis” on page 5-25

Assign and Save Comments

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

The **Result Details** pane displays information about the current defect.



- 2 Investigate the result further. Determine whether to fix your code, review the result later, or retain the code but provide some explanation.
- 3 On the **Results Summary** or **Result Details** pane, provide the following review information for the result:
 - **Severity** to describe how critical you consider the issue.
 - **Status** to describe how you intend to address the issue.

You can also create your own status or associate justification with an existing status. Select **Tools > Preferences** and create or modify statuses on the **Review Statuses** tab.

- **Comment** to describe any other information about the result.
- 4 To provide review information for several results together, select the results. Then, provide review information for a single result.

To select the results in a group:

- If the results are contiguous, left-click the first result. Then **Shift**-left click the last result.

To group certain results together, use the column headers on the **Results Summary** pane.

- If the results are not contiguous, **Ctrl**-left click each result.
- If the results belong to the same group and have the same color, right-click one result. From the context menu, select **Select All Type Results**.

For instance, select **Select All "Memory leak" Results**.

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

Import Review Comments from Previous Analysis

After you have reviewed verification results, you can reuse your review comments for subsequent verifications. By default, Polyspace Code Prover imports comments from the last verification on the module.

Disable Automatic Comment Import from Last Analysis

- 1 Select **Tools > Preferences**, which opens the Polyspace Preferences dialog box.
- 2 Select the **Project and Results Folder** tab.
- 3 Under **Import Comments**, clear **Automatically import comments from last verification**.
- 4 Click **OK**.

After you set this preference, for every run, the software imports review comments from the last run.

Import Comments from Another Analysis

- 1 Open your verification results.
- 2 Select **Tools > Import Comments**.
- 3 Navigate to the folder containing your previous results.
- 4 Select the results file and then click **Open**.

The review comments from the previous results are imported into the current results, and the Import checks and comments report opens showing the comments that do not apply to the current analysis.

Related Examples

- “Filter and Group Results” on page 5-9
- “Copy and Paste Annotations” on page 1-61


More About

- “Windows Used to Review Results” on page 5-38

Review Concurrency Defects

This example shows how to review defects that arise only in a multitasking analysis. For this example, use the results in the demo **Bug_Finder_Example.psprj**. To load the demo in your **Project Browser**, under **Help**, select **Examples > Bug_Finder_Example.psprj**.

Filter Concurrency Defects

- 1 Right-click any column header and select **Group** to add the Group column to your Result Summary view.
- 2 On the **Group** column, select the  icon.
- 3 From the filter menu, clear **All**. Select **Concurrency**.

Review Data Race Defects

- 1 Select the first **Data race** defect.

The **Result Details** pane lists the variable `bad_glob1` that is:

- Shared between multiple tasks and written in at least one of the tasks
- Not protected against concurrent operations

On the **Source** pane, the variable declaration appears highlighted.

- 2 To navigate to each operation involving `bad_glob1` in the source code, on the **Result Details** pane, click the row corresponding to the operation in the table. The lines with the operations are also highlighted in blue on the **Source** pane.
 - a To see if the access is in a critical section, use the **Access Protections** column. If one of the accesses is in a critical section, to fix the **Data race** defect, you can use the same critical section for the other accesses.
 - b To see which function contains the access, use the **Scope** column.
- 3 Select the second **Data race** defect.

The **Result Details** pane lists the variable `bad_glob2` involved in the defect. You can view similar information as the first **Data race** defect.

However, for this defect, the **Access** column on the **Result Details** pane lists why the operation can be non-atomic.

Review Locking Defects

- 1 Select the **Deadlock** defect.

The **Result Details** pane lists the sequence of operations that cause the **Deadlock**. You can see:

- The function call through which each task involved in the **Deadlock** enters a critical section.
 - The function call through which each task attempts to enter a critical section that is already entered by another task.
- 2 To navigate to each operation in the source code, on the **Result Details** pane, click the row corresponding to the operation in the table.
 - 3 Select the **Double lock** defect.

The **Result Details** pane lists the sequence of operations that cause the **Double lock**. You can see:

- The function call through which a task enters a critical section.
 - The function call through which the task attempts to enter the same critical section.
- 4 To navigate to each operation in the source code, on the **Result Details** pane, click the row corresponding to the operation in the table.
 - 5 Select the **Missing unlock** defect.
 - The **Source** pane shows the function call that begins a critical section.
 - On the **Result Details** pane, under the **Event** column, you can see which task contains the critical section.

See Also

Data race including atomic operations | Data race | Deadlock | Double lock | Double unlock | Missing lock | Missing unlock

Related Examples

- “Set Up Multitasking Analysis Manually” on page 1-52

More About

- “Modeling Multitasking Code” on page 1-47
- “Concurrency” on page 5-52

Review Code Metrics

This example shows how to review the code complexity metrics that Polyspace computes. For information on the individual metrics, see “Code Metrics”.

Polyspace does not compute code complexity metrics by default. To compute them during analysis, do the following:

- **User interface:** On the **Configuration** pane, select **Coding Rules & Code Metrics**. Select **Calculate Code Metrics**.
- **Command line:** Use the option `-code-metrics` with the `polyspace-bug-finder-nodesktop` command.

After analysis, the software displays code complexity metrics on the **Results Summary** pane. You can:

- Specify limits for the metric values through **Tools > Preferences**.

If you impose limits on metrics, the **Results Summary** pane displays only those metric values that violate the limits. Use predefined limits or assign your own limits. If you assign your own limits, you can share the limits file to enforce coding standards in your organization.

- Justify the value of a metric.

If a metric value exceeds specified limits and appears red, you can add a comment with the rationale.

You can also suppress code metrics from the **Results Summary** display. Select **Show > Defects & Rules**.

In this section...
“Impose Limits on Metrics” on page 5-30
“Comment and Justify Limit Violations” on page 5-33

Impose Limits on Metrics

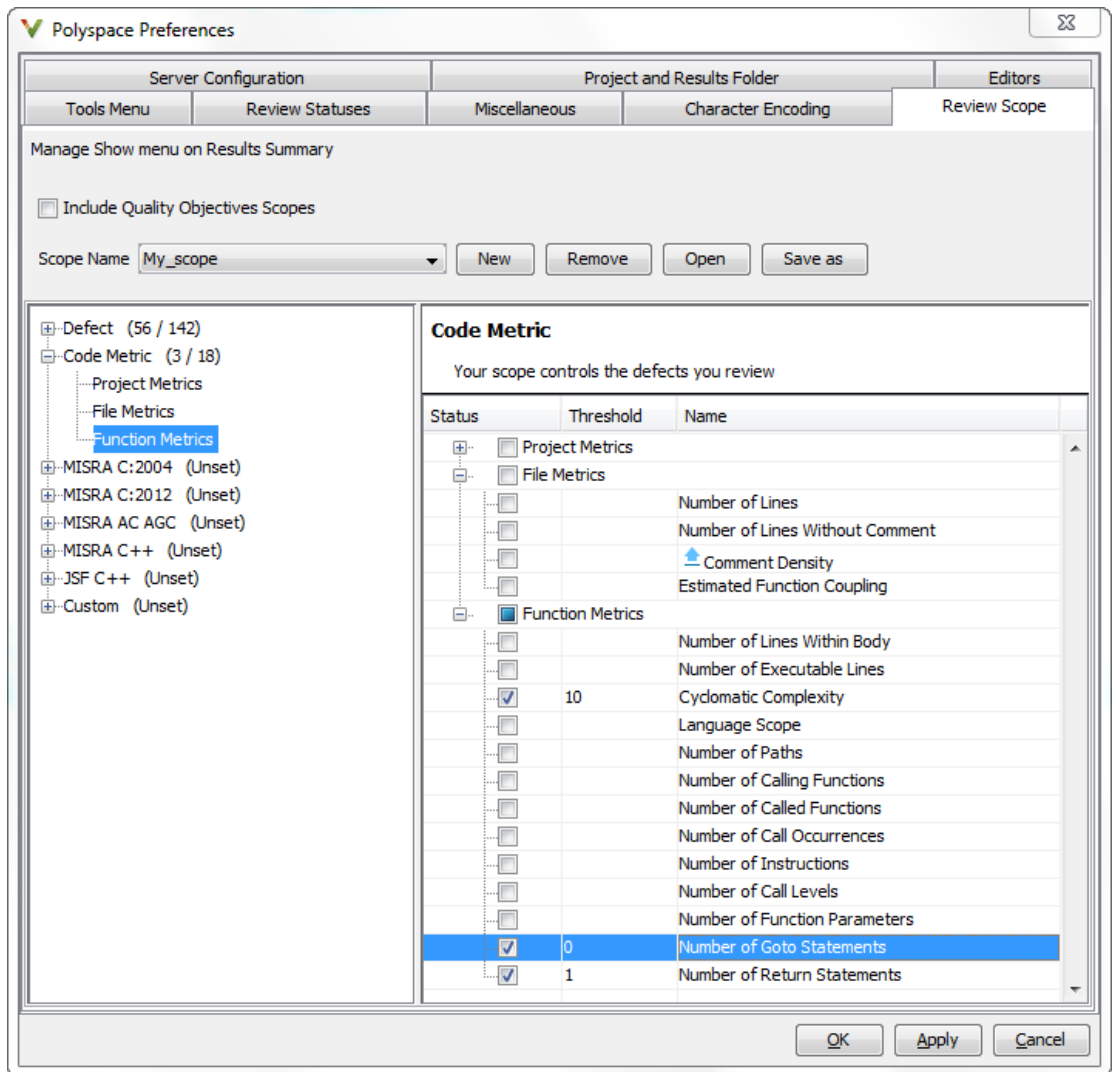
- 1 Select **Tools > Preferences**.
- 2 On the **Review Scope** tab, do one of the following:
 - To use a predefined limit, select **Include Quality Objectives Scopes**.

The **Scope Name** list shows the additional option **HIS**. The option **HIS** displays the “**HIS Metrics**” on page 5-57 only. Select the option to see the limit values.

- To define your own limits, select **New**. Save your limits file.

On the left pane, select **Code Metric**. On the right, select a metric and specify a limit value for the metric. Other than **Comment Density**, limit values are upper limits.

To select all metrics in a category such as **Function Metrics**, select the box next to the category name. For more information on the metrics categories, see “Code Metrics”. If only a fraction of metrics in a category are selected, the check box next to the category name displays a symbol.



3 Select **Apply** or **OK**.

On the **Results Summary** pane, the **Show** menu displays additional options.

- If you use predefined limits, the option **HIS** appears. This option displays code metrics only.
 - If you define your own limits, the option corresponding to your limits file name appears.
- 4** Select the option corresponding to the limits that you want. Only metric values that violate your limits appear on the **Results Summary** pane.

Note: To enforce coding standards across your organization, share your limits file that you saved in XML format.

People in your organization can use the **Open** button on the **Review Scope** tab and navigate to the location of the XML file.

Comment and Justify Limit Violations


Once you use the **Show** menu to display only metrics that violate limits, you can review each violation.

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

The code metrics appear together under one node.

- 2** Expand the node. Select each violation.

- On the **Results Summary** pane, in the **Information** column, you can see the metric value.
- On the **Result Details** pane, you can see the metric value and a brief description of the metric.

For more detailed descriptions and examples, select the  icon.

- 3** On the **Results Summary** pane, add a comment and justification describing why the violation occurs. For more information, see “Review and Fix Results” on page 5-24.

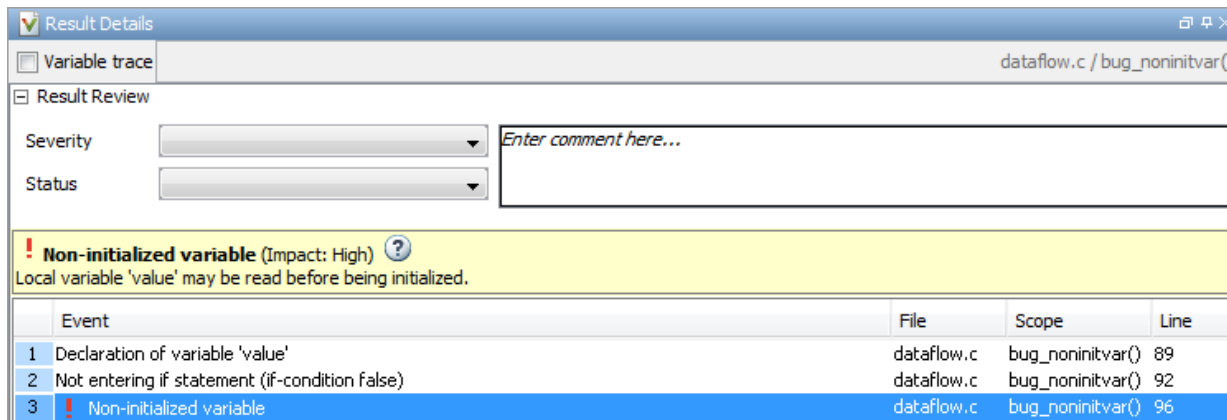
Navigate to Root Cause of Defect

Through the Polyspace Bug Finder user interface, you can navigate to the root cause of a defect in your source code. If you select a result on the **Results Summary** pane, you see the immediate location of the defect on the **Source** pane. However, the defect can be related to previous statements in your source code.

For instance, a **Non-initialized variable** defect appears at the location where you read a noninitialized variable. However, it is possible that you initialized the variable previously. For instance, the initialization occurred in a branch of a previous **if** statement and the variable is noninitialized only if that branch is not entered.

Navigate Code Sequence Causing Defect

Often, the **Result Details** pane shows the event history leading to the defect. To see the code statement that the event describes, click the event.



The screenshot shows the 'Result Details' window for a defect in 'dataflow.c / bug_noninitvar()'. The 'Variable trace' checkbox is checked. The 'Result Review' section includes dropdowns for 'Severity' and 'Status', and a text area for 'Enter comment here...'. A yellow banner displays the defect: 'Non-initialized variable (Impact: High) Local variable 'value' may be read before being initialized.' Below this is a table of events:

	Event	File	Scope	Line
1	Declaration of variable 'value'	dataflow.c	bug_noninitvar()	89
2	Not entering if statement (if-condition false)	dataflow.c	bug_noninitvar()	92
3	! Non-initialized variable	dataflow.c	bug_noninitvar()	96

On the **Source** pane, the statements are highlighted in blue and the corresponding line numbers outlined in boxes.

On the **Result Details** pane, you can select the **Variable trace** box, if available. The event sequence expands to show more events related to the defect. The statements that the additional events describe are highlighted in light blue on the **Source** pane.

Navigate to Identifier Definition

Often, to diagnose a defect, you have to navigate to an identifier definition. On the **Source** pane, right-click the identifier name. Select **Go To Definition**.

For instance, the C++ defect **Object slicing** appears at the location where you pass a derived class object by value to a function. The function expects a base class object as parameter. To diagnose this defect, you can navigate to the base and derived class definitions.

To navigate to the derived class definition starting from the defect location:

- 1 Right-click the derived class object name and select **Go To Definition**.
- 2 In the derived class object definition, right-click the derived class name and select **Go To Definition**.

Navigate to Identifier References

Often, to diagnose a defect, you have to see the locations where an identifier is used.


For instance, an `if` statement shows the **Dead code** defect. You want to understand why the variable that controls entry to the `if` statement has a certain set of values. Therefore, you want to see previous assignments to that variable.

To navigate to previous locations where an identifier is used:

- 1 Right-click the identifier name and select **Search For All References**.
The search results appear on the **Search** pane with the current location highlighted.
- 2 Click each search result, starting backward from the highlighted result.
- 3 The option **Search for All References** is not available in some cases. For instance, if you right-click a C++ `virtual` function, this option is not available.

Use one of the following options to search for occurrences of the identifier name:

- **Search For *Identifier_name* in Current Source File**
 - **Search For *Identifier_name* in All Source Files**
- 4 If reviewing a defect requires deeper navigation in your source code, you can create a duplicate source code window that focuses on the defect while you navigate in the original source code window.

- a Right-click on the **Source** pane and select **Create Duplicate Code Window**.
- b Right-click on the tab showing the duplicate file name and select **New Vertical Group**.
- c Perform the navigation steps in the original file window while the defect still appears on the duplicate file window.
- d After reviewing the defect, click the  button on the **Results Summary** pane to return to the defect location in the original file window. Close the duplicate window.

Related Examples

- “Review and Fix Results” on page 5-24

More About

- “Source” on page 5-44
- “Result Details” on page 5-50

Results Folder Contents

Every time you run an analysis, Polyspace generates files and folders that contain information about configuration options and analysis results. The contents of results folders depend on the configuration options and how the analysis was started.

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 4-6.

Files in the Results Folder

Some of the files and folders in the results folder are described below:

- `Polyspace_release_project_name_date-time.log` — A log file associated with each analysis.
- `ps_results.psbf` — An encrypted file containing your Polyspace results. Open this file in the Polyspace environment to view your results.
- `ps_sources.db` — A non-encrypted database file listing source files and macros.
- `drs-template.xml` — A template generated when you use constraint specification.
- `ps_comments.db` — An encrypted database file containing your comments and justifications.
- `comments_bak` — A subfolder used to import comments between results.
- `.status` and `.settings` — Two folders used to store files needed to relaunch the analysis.
- `Polyspace-Doc` — When you generate a report, by default, your report is saved in this folder with the name *ProjectName_ReportType*. For example, a developer report in PDF format would be, `myProject_Developer.pdf`.

See Also

`-results-dir`

Related Examples

- “Specify Results Folder” on page 4-6
- “Open Results” on page 5-2

Windows Used to Review Results

In this section...

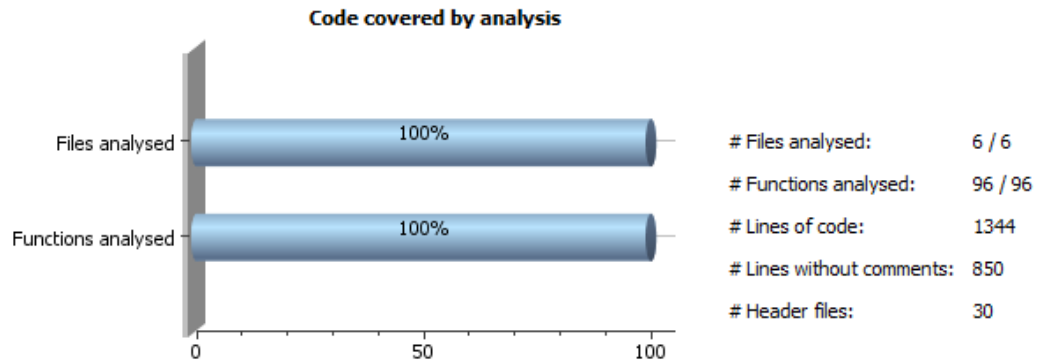
“Dashboard” on page 5-38
 “Results Summary” on page 5-42
 “Source” on page 5-44
 “Result Details” on page 5-50

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 Polyspace, this tab is displayed by default. You can view the following graphs:

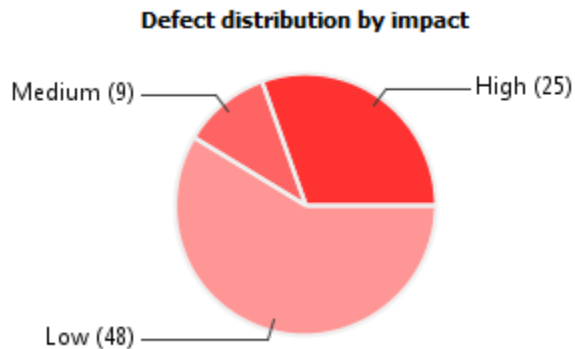
- **Code covered by analysis**



From this graph you can obtain the following information:

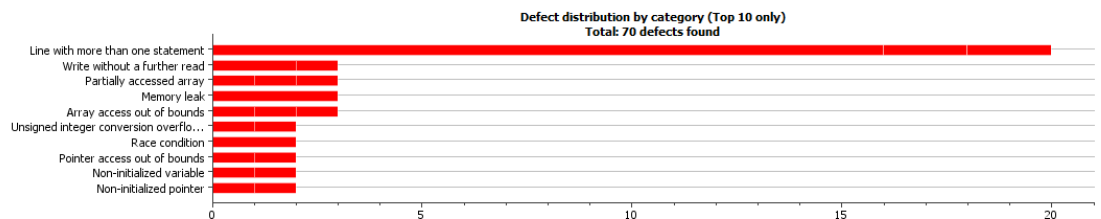
- **# Files analyzed:** Ratio of analyzed files to total number of files. If a file contains a compilation error, Polyspace Bug Finder does not analyze the file.

- **# Functions analyzed:** Ratio of analyzed functions to total number of functions in the analyzed files. If the analysis of a function takes longer than a certain 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 impact**



From this pie chart, you can obtain a graphical visualization of the defect distribution by impact. You can find at a glance whether the defects that Polyspace Bug Finder found in your code are low-impact defects. For more information on impact, see “Classification of Defects by Impact” on page 5-12.

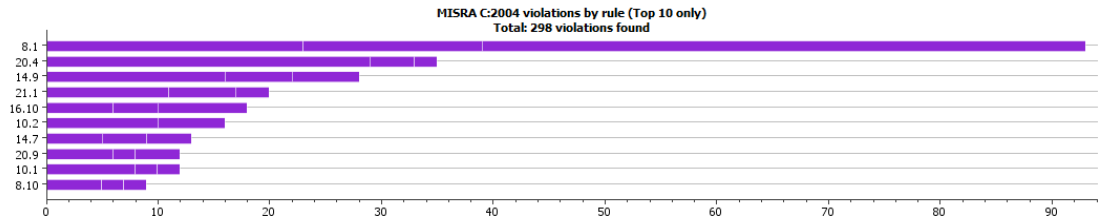
- **Defect distribution by category or file**



From this graph you can obtain the following information.

	Category	File
Top 10	<p>The ten defect types with the highest number of individual defects.</p> <ul style="list-style-type: none"> Each column represents a defect type and is divided into the: <ul style="list-style-type: none"> 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. <p>Place your cursor on a column to see the file name and number of defects of this type in this file.</p> <ul style="list-style-type: none"> The x-axis represents the number of defects. <p>Use this view to organize your check review starting at defect types with more individual defects.</p>	<p>The ten source files with the highest number of defects.</p> <ul style="list-style-type: none"> Each column represents a file and is divided into the: <ul style="list-style-type: none"> 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. <p>Place your cursor on a column to see the defect type name and number of defects of this type in this file.</p> <ul style="list-style-type: none"> The x-axis represents the number of defects. <p>Use this view to organize your check review starting at files with more defects.</p>
Bottom 10	<p>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.</p> <p>Use this view to organize your check review starting at defect types with fewer individual defects.</p>	<p>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.</p> <p>Use this view to organize your check review starting at files with fewer defects.</p>

- **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	<p>The ten rules with the highest number of violations.</p> <ul style="list-style-type: none"> • Each column represents a rule number and is divided into the: <ul style="list-style-type: none"> • 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. <p>Place your cursor on a column to see the file name and number of violations of this rule in the file.</p> <ul style="list-style-type: none"> • The x-axis represents the number of rule violations. <p>Use this view to organize your review starting at rules with more violations.</p>	<p>The ten source files containing the highest number of violations.</p> <ul style="list-style-type: none"> • Each column represents a file and is divided into the: <ul style="list-style-type: none"> • 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. <p>Place your cursor on a column to see the rule number and number of violations of the rule in this file.</p> <ul style="list-style-type: none"> • The x-axis represents the number of rule violations. <p>Use this view to organize your review starting at files with more rule violations.</p>

	Category	File
Bottom 10	<p>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.</p> <p>Use this view to organize your review starting at rules with fewer violations.</p>	<p>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.</p> <p>Use this view to organize your review starting at files with fewer rule violations.</p>

For a list of supported coding rules, see “Supported MISRA C:2004 and MISRA AC AGC Rules” on page 2-14, “Supported MISRA C++ Coding Rules” on page 2-68 and “Supported JSF C++ Coding Rules” on page 2-96.

Results Summary

The **Results Summary** pane lists all defects along with their attributes. To organize your results review, from the **Group by** list on this pane, select one of the following options:

- **None:** Lists defects and coding rule violations without grouping. By default the results are listed in order of severity.
- **Family:** Lists results grouped by grouping. For more information on the defects covered by a group, see “Bug Finder Defect Groups” on page 5-52.
- **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.

- **File:** 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.

Attribute	Description
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.
Group	Category of the defect. For more information on the defects covered by a group, see “Polyspace Bug Finder Results”.
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 <code>class_name::function_name</code> .
Severity	Level of severity you have assigned to the defect. The possible levels are: <ul style="list-style-type: none"> • High • Medium • Low • Not a defect


Attribute	Description
Status	Review status you have assigned to the check. The possible statuses are: <ul style="list-style-type: none">• Fix• Improve• Investigate• Justified• No action planned• Other
Comments	Comments you have entered about the check

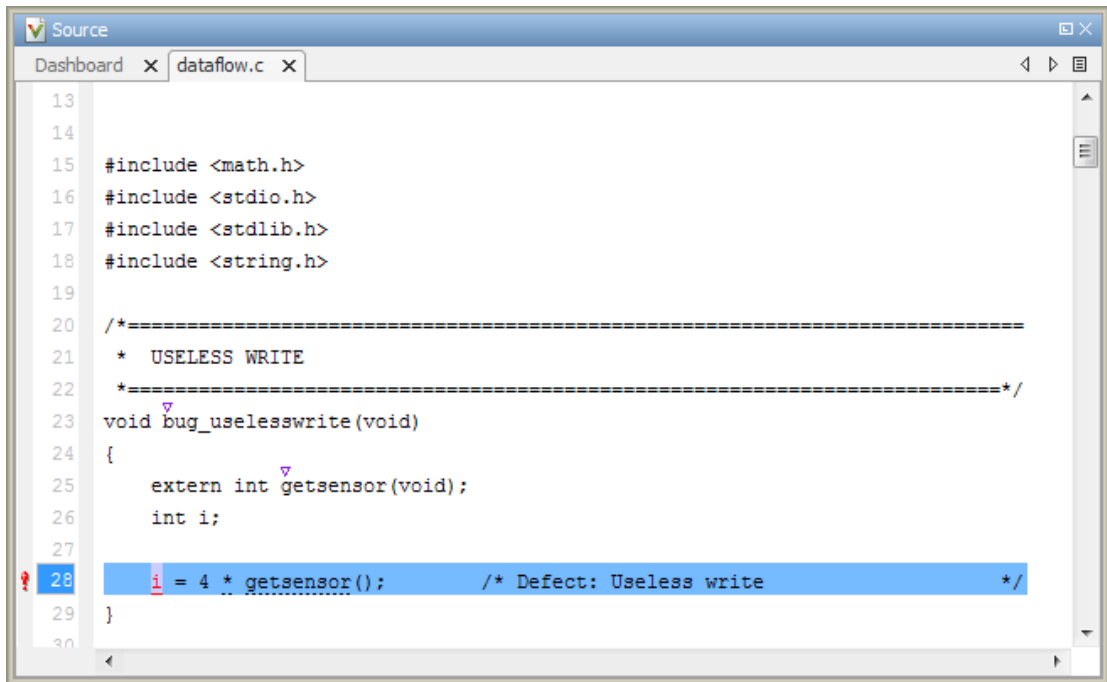
To show or hide any of the columns, right-click anywhere on the column titles. From the context menu, select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through the checks. For more information, see “Review and Fix Results” on page 5-24.
- Organize your check review using filters on the columns. For more information, see “Filter and Group Results” on page 5-9.

Source

The **Source** pane shows the source code with the defects colored in red and the corresponding line number marked by .



The screenshot shows a window titled "Source" with a tab for "dataflow.c". The code is as follows:

```
13
14
15 #include <math.h>
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <string.h>
19
20 /*=====
21  * USELESS WRITE
22  *=====*/
23 void bug_uselesswrite(void)
24 {
25     extern int getsensor(void);
26     int i;
27
28     i = 4 * getsensor();    /* Defect: Useless write */
29 }
30
```

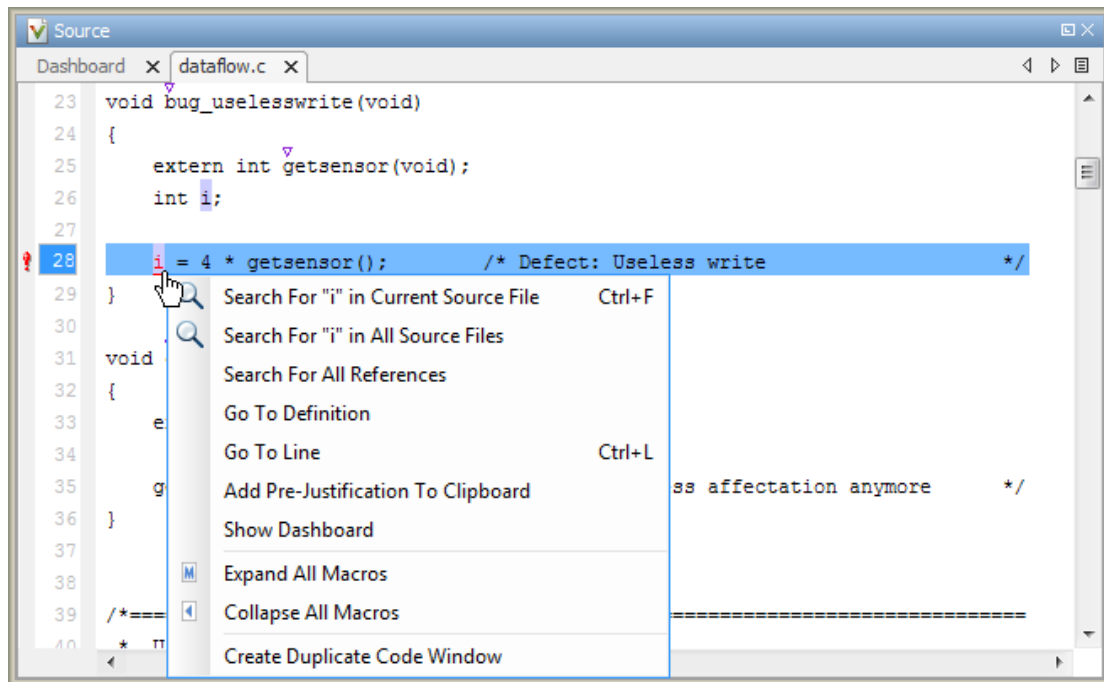
Line 28 is highlighted in blue, and a tooltip is visible over the variable `i`, indicating a defect: "Defect: Useless write".

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


On the **Source** pane, if you right-click a text string, the context menu provides options to examine your code:

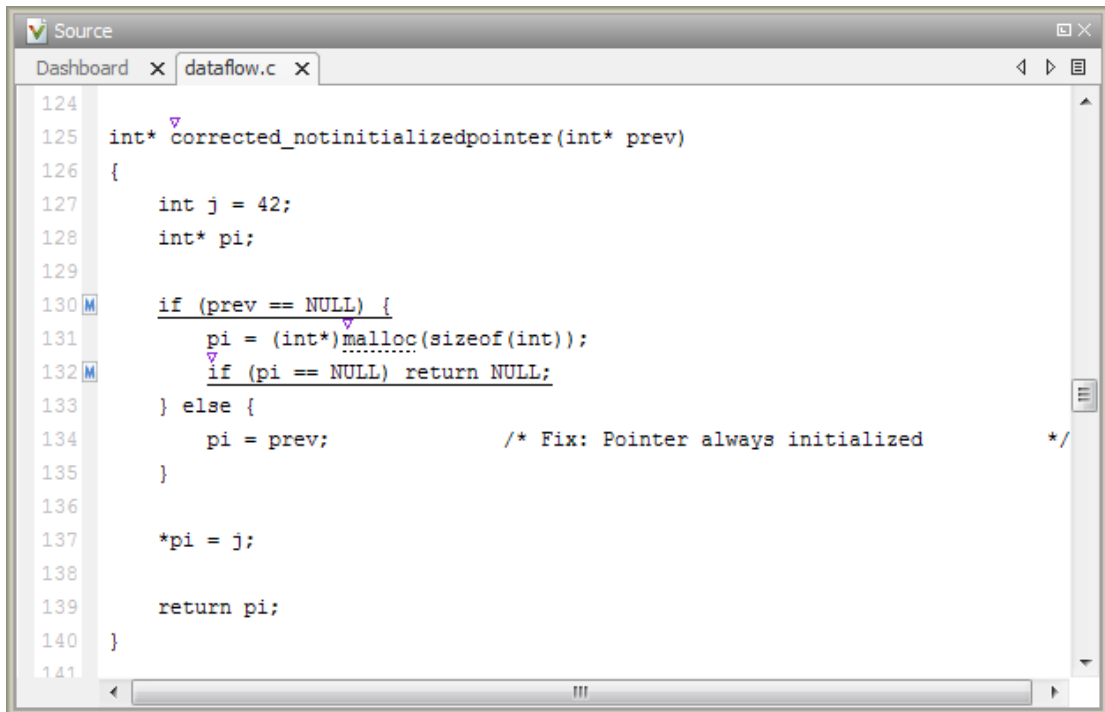


For example, if you right-click the variable `i`, you can use the following options to examine and navigate through your code:

- **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.

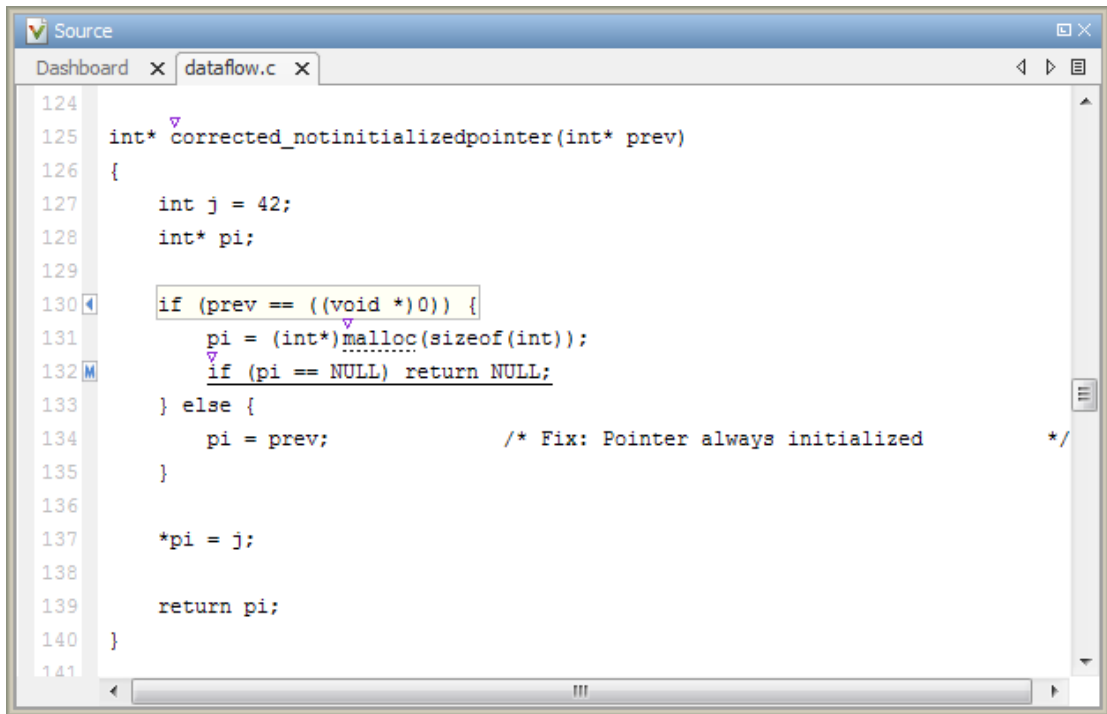
Expand Macros

You can view the contents of source code macros in the source code view. A code information bar displays  icons that identify source code lines with macros.




```
124
125 int* corrected_notinitializedpointer(int* prev)
126 {
127     int j = 42;
128     int* pi;
129
130 M     if (prev == NULL) {
131         pi = (int*)malloc(sizeof(int));
132 M         if (pi == NULL) return NULL;
133     } else {
134         pi = prev;          /* Fix: Pointer always initialized          */
135     }
136
137     *pi = j;
138
139     return pi;
140 }
141
```

When you click a line with this icon, the software displays the contents of macros on that line in a box.



```
124
125 int* corrected_notinitializedpointer(int* prev)
126 {
127     int j = 42;
128     int* pi;
129
130     if (prev == ((void *)0)) {
131         pi = (int*)malloc(sizeof(int));
132         if (pi == NULL) return NULL;
133     } else {
134         pi = prev;          /* Fix: Pointer always initialized */
135     }
136
137     *pi = j;
138
139     return pi;
140 }
141
```

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:

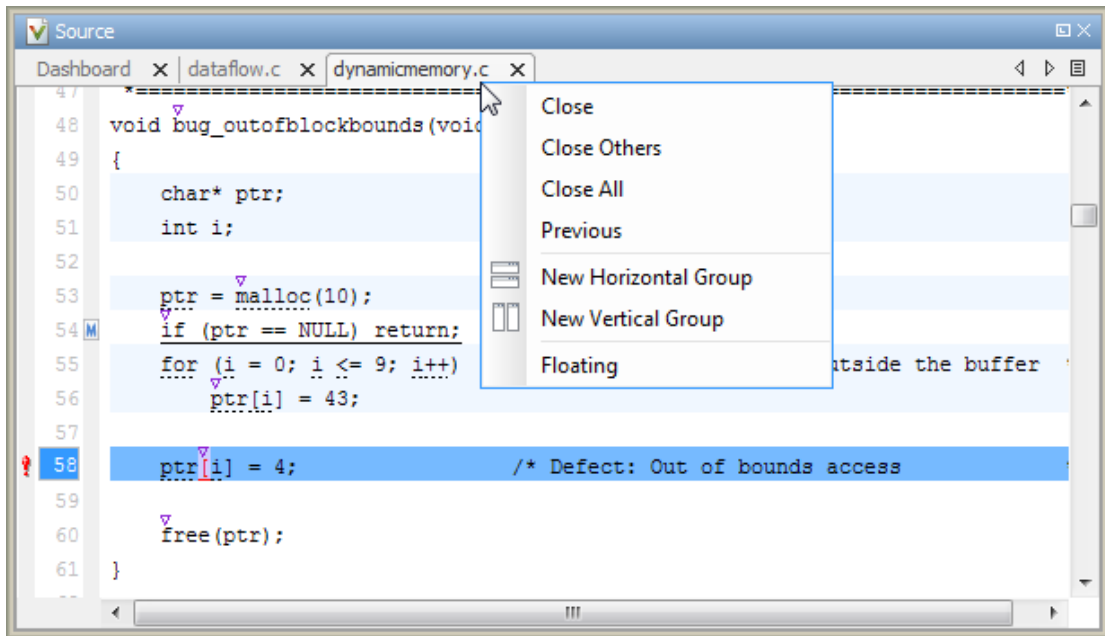
- 1 Right-click anywhere on the source.
- 2 From the context menu, select either **Expand All Macros** or **Collapse All Macros**.

Note: The **Result 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.

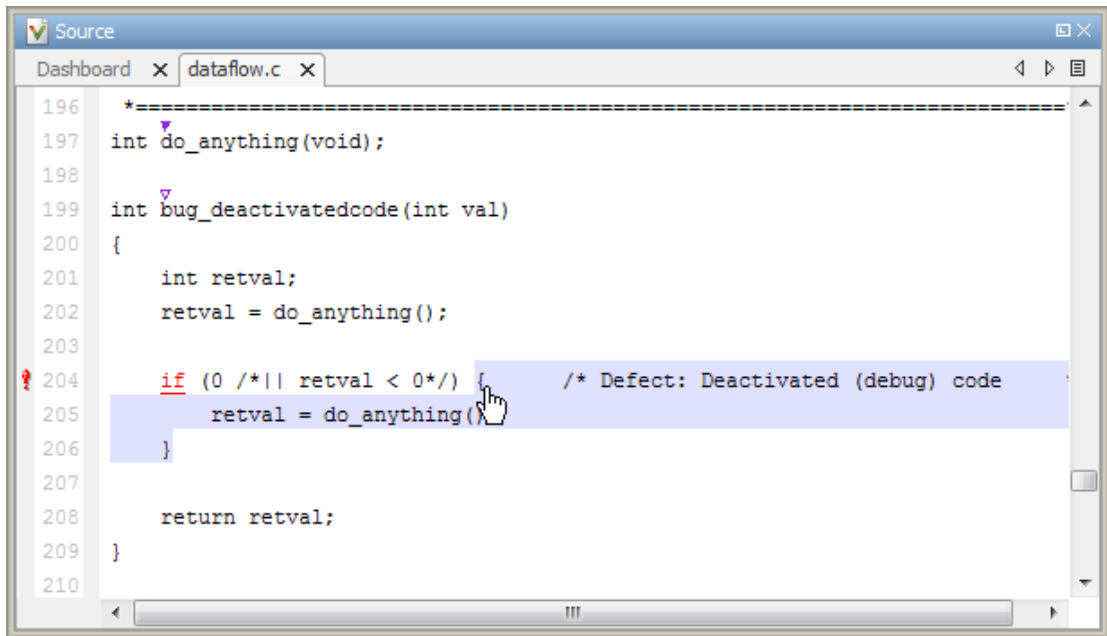


From the **Source** pane context menu, you can:

- **Close** – Close the currently selected source file. You can also use the **X** button to close tabs.
- **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

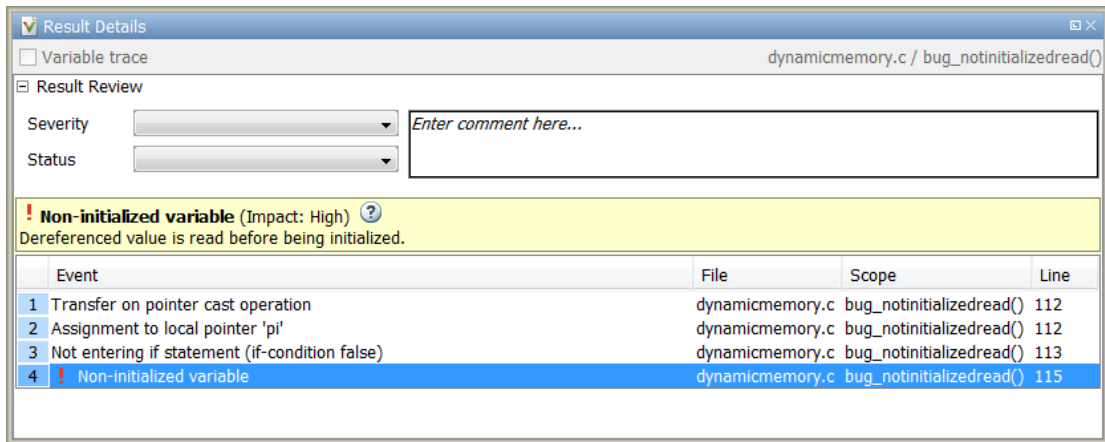
On the **Source** pane, to highlight a block of code, click either its opening or closing brace. If the brace itself is highlighted, click the brace twice.




Result Details

The **Result Details** pane contains comprehensive information about a specific defect. To see this information, on the **Results Summary** pane, select the defect.

On this pane, you can also assign a **Severity** and **Status** to each check. You can also enter comments to describe the results of your review. This action helps you track the progress of your review and avoid reviewing the same check twice.



- 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.
- The  button allows you to access documentation for the defect.

For more information, see “Navigate to Root Cause of Defect” on page 5-34.

Bug Finder Defect Groups

In this section...

“Concurrency” on page 5-52
“Data flow” on page 5-53
“Dynamic Memory” on page 5-53
“Good Practice” on page 5-53
“Numerical” on page 5-54
“Object Oriented” on page 5-54
“Programming” on page 5-54
“Resource Management” on page 5-54
“Static Memory” on page 5-55
“Security” on page 5-55
“Tainted data” on page 5-55

Concurrency

These defects are related to multitasking code.

Data Race Defects

The data race defects occur when multiple tasks operate on a shared variable without protection. For the defect to occur:

- One of the operations must be a write operation.
- The operations must not be protected by the same mechanism.

For the specific defects, see “Concurrency Defects”.

Locking Defects

The locking defects occur when the critical sections are not set up appropriately. For example:

- The critical sections are involved in a deadlock.
- A lock function does not have the corresponding unlock function.

- A lock function is called twice without an intermediate call to an unlock function.

Critical sections protect shared variables from concurrent access. Polyspace expects critical sections to follow a certain format. The critical section must lie between a call to a lock function and a call to an unlock function.

For the specific defects, see “Concurrency 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”.

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”.

Good Practice

These defects allow you to observe good coding practices. The defects by themselves might not cause a crash, but they sometimes highlight more serious logic errors in your code. The defects also make your code vulnerable to attacks and hard to maintain.

The defects include:

- Hard-coded constants such as buffer size and loop boundary
- Unused function parameters

For specific defects, see “Good Practice Defects”.

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”.

Object Oriented

These defects are related to the object-oriented aspect of C++ programming. The defects highlight class design issues or issues in the inheritance hierarchy.

The defects include:

- Data member not initialized or incorrectly initialized in constructor
- Incorrect overriding of base class methods
- Breaking of data encapsulation

For specific defects, see “Object Oriented Defects”.

Programming

These defects are errors relating to programming syntax. These defects include:

- Assignment versus equality operators
- Mismatches between variable qualifiers or declarations
- Badly formatted strings

For specific defects, see “Programming Defects”.

Resource Management

These defects are related to file handling. The defects include:

- Unclosed file stream
- Operations on a file stream after it is closed

For specific defects, see “Resource Management 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”.

Security

These defects highlight places in your code which are vulnerable to hacking or other security attacks. Many of these defects do not cause runtime errors, but instead point out risky areas in your code. The defects include:

- Managing sensitive data
- Using dangerous or obsolete functions
- Generating random numbers
- Externally controlled paths and commands

For more details about specific defects, see “Security Defects”.

Tainted data

These defects highlight elements in your code which are from unsecured sources. Malicious attackers can use input data or paths to attack your program and cause failures. These defects highlight elements in your code that are vulnerable. Defects include:

- Use of tainted variables or pointers
- Externally controlled paths

For more details about specific defects, see “Tainted Data Defects”.

HIS Metrics

The following list shows the Hersteller Initiative Software (HIS) standard metrics that Polyspace evaluates. These metrics and the recommended limits for their values are part of a standard defined by a major group of Original equipment manufacturers or OEMs. For more information on how to focus your review to this subset of code metrics, see “Review Code Metrics” on page 5-30.

Project

Polyspace evaluates the following HIS metrics at the project level.

Metric	Recommended Upper Limit
Number of Direct Recursions	0
Number of Recursions	0

File

Polyspace evaluates the HIS metric, comment density, at the file level. The recommended lower limit is 20.

Function

Polyspace evaluates the following HIS metrics at the function level.

Metric	Recommended Upper Limit
Cyclomatic Complexity	10
Language Scope	4
Number of Call Levels	80
Number of Calling Functions	5
Number of Called Functions	7
Number of Function Parameters	5
Number of Goto Statements	0
Number of Instructions	50

Metric	Recommended Upper Limit
Number of Paths	80
Number of Return Statements	1

Common Weakness Enumeration from Bug Finder Defects

In this section...

“Common Weakness Enumeration” on page 5-59

“Polyspace Bug Finder and CWE Compatibility” on page 5-59

Common Weakness Enumeration

Common Weakness Enumeration (CWE™) is a dictionary of common software weaknesses that can occur in software architecture, design, code, or implementation. These weaknesses can lead to security vulnerabilities.

The dictionary assigns a unique identifier to each software weakness. Therefore, this dictionary serves as a common language for describing software security weaknesses, and a standard for software security tools targeting these weaknesses.

For more information, see [Common Weakness Enumeration](#).

Polyspace Bug Finder and CWE Compatibility

With Polyspace Bug Finder, you can check and document whether your software contains weaknesses listed in the CWE dictionary. Polyspace Bug Finder supports some aspects of the CWE Compatibility and Effectiveness Program:

CWE Compatibility Requirement	Polyspace Bug Finder Support
CWE Searchable	<p>You can list instances of a software weakness corresponding to a certain CWE identifier.</p> <p>For more information, see “Filter CWE Identifiers” on page 5-61.</p>
CWE Output	<ul style="list-style-type: none"> • You can view CWE identifiers corresponding to certain Polyspace Bug Finder defects. <p>For more information, see “View CWE Identifiers” on page 5-61.</p>

CWE Compatibility Requirement	Polyspace Bug Finder Support
	<ul style="list-style-type: none"><li data-bbox="798 302 1326 392">• You can include CWE identifiers corresponding to Polyspace Bug Finder defects in your report. <p data-bbox="832 421 1307 512">For more information, see “Generate Report with CWE Identifiers” on page 5-61.</p>

For more information on the CWE Compatibility and Effectiveness Program, see [CWE Compatibility](#).

Related Examples

- “Find CWE Identifiers from Defects” on page 5-61

More About

- “Mapping Between CWE Identifiers and Defects” on page 5-63

Find CWE Identifiers from Defects

This example shows how to check whether your software has weaknesses listed by the Common Weakness Enumeration or CWE dictionary. The dictionary assigns a unique identifier to each software weakness. When a Polyspace Bug Finder result can be associated with CWE identifiers, the software displays those identifiers for the result. Using the identifiers, you can evaluate your code against CWE standards.

In this section...

“View CWE Identifiers” on page 5-61

“Filter CWE Identifiers” on page 5-61

“Generate Report with CWE Identifiers” on page 5-61


View CWE Identifiers

To view the CWE identifiers for defects on the **Results Summary** pane:

- 1 Right-click any column header.
- 2 Select **CWE ID**.

Filter CWE Identifiers

To filter a particular CWE identifier:

- 1 On the **CWE ID** column, click the  icon.
- 2 From the drop-down list, select **Custom**.
- 3 From the **Condition** drop-down list, select **contains**.
- 4 In the **Value** field, enter the CWE ID that you want to filter. Click **OK**.

Generate Report with CWE Identifiers

To generate a report containing CWE identifiers, do the following.

- To enable report generation before analysis:
 - 1 On the **Configuration** pane, select **Reporting**.

- 2 Select **Generate report**.
 - 3 From the **Report template** list, select **BugFinder_CWE**.
- To generate a report after analysis:
 - 1 Open your results.
 - 2 Select **Reporting > Run Report**.
 - 3 From the **Select Reports** list, select **BugFinder_CWE**.

More About

- “Common Weakness Enumeration from Bug Finder Defects” on page 5-59
- “Mapping Between CWE Identifiers and Defects” on page 5-63

Mapping Between CWE Identifiers and Defects

The following table lists the CWE IDs (version 2.6) addressed by Polyspace Bug Finder and the corresponding defects.

CWE ID	Polyspace Bug Finder Defect
15: External Control of System or Configuration Setting	Host change using externally controlled elements Use of externally controlled environment variable
22: Improper Limitation of a Pathname to a Restricted Directory	Vulnerable path manipulation
23: Relative Path Traversal	Vulnerable path manipulation
36: Absolute Path Traversal	Vulnerable path manipulation
77: Improper Neutralization of Special Elements used in a Command	Execution of externally controlled command
78: Improper Neutralization of Special Elements used in an OS Command	Command executed from externally controlled path Execution of externally controlled command
88: Argument Injection or Modification	Execution of externally controlled command
114: Process Control	Execution of a binary from a relative path can be controlled by an external actor Library loaded from externally controlled path Load of library from a relative path can be controlled by an external actor
119: Improper Restriction of Operations within the Bounds of a Memory Buffer	Array access out of bounds Pointer access out of bounds
120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	Invalid use of standard library memory routine

CWE ID	Polyspace Bug Finder Defect
	Invalid use of standard library string routine Tainted NULL or non-null-terminated string
121: Stack-based Buffer Overflow	Array access with tainted index Destination buffer overflow in string manipulation
122: Heap-based Buffer Overflow	Pointer dereference with tainted offset
124: Buffer Underwrite ('Buffer Underflow')	Array access with tainted index Buffer overflow from incorrect string format specifier Destination buffer underflow in string manipulation Pointer dereference with tainted offset
125: Out-of-bounds Read	Array access with tainted index Buffer overflow from incorrect string format specifier Destination buffer overflow in string manipulation Use of tainted pointer
126: Buffer Over-read	Buffer overflow from incorrect string format specifier
127: Buffer Under-read	Buffer overflow from incorrect string format specifier
129: Improper Validation of Array Index	Array access with tainted index Pointer dereference with tainted offset
130: Improper Handling of Length Parameter Inconsistency	Mismatch between data length and size

CWE ID	Polyspace Bug Finder Defect
134: Uncontrolled Format String	Tainted string format
170: Improper Null Termination	Missing null in string array Tainted NULL or non-null-terminated string
188: Reliance on Data/Memory Layout	Invalid assumptions about memory organization Pointer access out of bounds
190: Integer Overflow or Wraparound	Integer conversion overflow Integer overflow Shift operation overflow Tainted division operand Unsigned integer conversion overflow Unsigned integer overflow
191: Integer Underflow (Wrap or Wraparound)	Integer conversion overflow Integer overflow Unsigned integer conversion overflow Unsigned integer overflow
194: Unexpected Sign Extension	Sign change integer conversion overflow Tainted sign change conversion
195: Signed to Unsigned Conversion Error	Sign change integer conversion overflow Tainted sign change conversion
196: Unsigned to Signed Conversion Error	Sign change integer conversion overflow

CWE ID	Polyspace Bug Finder Defect
197: Numeric Truncation Error	Integer conversion overflow Float conversion overflow Unsigned integer conversion overflow
226: Sensitive Information Uncleared Before Release	Uncleared sensitive data in stack
227: Improper Fulfillment of API Contract	Invalid use of standard library floating point routine Invalid use of standard library memory routine Invalid use of standard library routine Invalid use of standard library string routine Writing to const qualified object
240: Improper Handling of Inconsistent Structural Elements	Mismatch between data length and size
242: Use of Inherently Dangerous Function	Use of dangerous standard function
243: Creation of chroot Jail Without Changing Working Directory	File manipulation after chroot() without chdir("/")
244: Improper Clearing of Heap Memory Before Release	Sensitive heap memory not cleared before release
251: Often Misused: String Management	Destination buffer overflow in string manipulation
327: Use of a Broken or Risky Cryptographic Algorithm	Unsafe standard encryption function

CWE ID	Polyspace Bug Finder Defect
330: Use of Insufficiently Random Values	Deterministic random output from constant seed Predictable random output from predictable seed Vulnerable pseudo-random number generator
336: Same Seed in PRNG	Deterministic random output from constant seed
337: Predictable Seed in PRNG	Predictable random output from predictable seed
338: Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)	Vulnerable pseudo-random number generator
366: Race Condition within a Thread	Data race including atomic operations Data race
367: Time-of-check Time-of-use (TOCTOU) Race Condition	File access between time of check and use (TOCTOU)
369: Divide By Zero	Float division by zero Integer division by zero Invalid use of standard library integer routine Invalid use of standard library floating point routine Tainted division operand Tainted modulo operand
377: Insecure Temporary File	Use of non-secure temporary file
398: Indicator of Poor Code Quality	Write without a further read
400: Uncontrolled Resource Consumption	Loop bounded with tainted value

CWE ID	Polyspace Bug Finder Defect
401: Improper Release of Memory Before Removing Last Reference	Memory leak
404: Improper Resource Shutdown or Release	Invalid deletion of pointer Invalid free of pointer Memory leak
415: Double Free	Deallocation of previously deallocated pointer
416: Use After Free	Use of previously freed pointer
427: Uncontrolled Search Path Element	Execution of a binary from a relative path can be controlled by an external actor Library loaded from externally controlled path Load of library from a relative path can be controlled by an external actor Use of externally controlled environment variable
456: Missing Initialization of a Variable	Member not initialized in constructor Non-initialized pointer Non-initialized variable
457: Use of Uninitialized Variable	Member not initialized in constructor Non-initialized pointer Non-initialized variable
466: Return of Pointer Value Outside of Expected Range	Array access out of bounds Pointer access out of bounds
467: Use of sizeof() on a Pointer Type	Possible misuse of sizeof Wrong type used in sizeof

CWE ID	Polyspace Bug Finder Defect
468: Incorrect Pointer Scaling	Incorrect pointer scaling Unreliable cast of pointer
471: Modification of Assumed-Immutable Data	Writing to const qualified object
475: Undefined Behavior for Input to API	Copy of overlapping memory
476: NULL Pointer Dereference	Null pointer Tainted NULL or non-null-terminated string
477: Use of Obsolete Functions	Use of obsolete standard function
478: Missing Default Case in Switch Statement	Missing case for switch condition
481: Assigning instead of Comparing	Invalid use of = operator
482: Comparing instead of Assigning	Invalid use of == operator
532: Information Exposure Through Log Files	Sensitive data printed out
534: Information Exposure Through Debug Log Files	Sensitive data printed out
535: Information Exposure Through Shell Error Message	Sensitive data printed out
547: Use of Hard-coded, Security-relevant Constants	Hard coded buffer size Hard coded loop boundary
558: Use of getlogin() in Multithreaded Application	Unsafe standard function
560: Use of umask() with chmod-style Argument	Umask used with chmod-style arguments
561: Dead Code	Dead code Static uncalled function Unreachable code

CWE ID	Polyspace Bug Finder Defect
562: Return of Stack Variable Address	Pointer or reference to stack variable leaving scope
573: Improper Following of Specification by Caller	Modification of internal buffer returned from nonreentrant standard function
587: Assignment of a Fixed Address to a Pointer	Function pointer assigned with absolute address
590: Free of Memory not on the Heap	Invalid free of pointer
606: Unchecked Input for Loop Condition	Loop bounded with tainted value
628: Function Call with Incorrectly Specified Arguments	Bad file access mode or status Copy of overlapping memory Invalid va_list argument Modification of internal buffer returned from nonreentrant standard function Standard function call with incorrect arguments
663: Use of a Non-reentrant Function in a Concurrent Context	Unsafe standard encryption function Unsafe standard function
665: Improper Initialization	Call to memset with unintended value Improper array initialization Overlapping assignment Use of memset with size argument zero
666: Operation on Resource in Wrong Phase of Lifetime	Incorrect order of network connection operations
667: Improper Locking	Missing unlock
672: Operation on a Resource after Expiration or Release	Use of previously closed resource Closing a previously closed resource

CWE ID	Polyspace Bug Finder Defect
676: Use of Potentially Dangerous Function	Use of dangerous standard function
681: Incorrect Conversion between Numeric Types	Float conversion overflow
682: Incorrect Calculation	Float overflow Invalid use of standard library floating point routine Tainted modulo operand
685: Function Call With Incorrect Number of Arguments	Declaration mismatch Format string specifiers and arguments mismatch Standard function call with incorrect arguments
686: Function Call with Incorrect Argument Type	Bad file access mode or status Declaration mismatch Format string specifiers and arguments mismatch Standard function call with incorrect arguments Writing to const qualified object
687: Function Call with Incorrectly Specified Argument Value	Copy of overlapping memory Standard function call with incorrect arguments Tainted size of variable length array Variable length array with nonpositive size
691: Insufficient Control Flow Management	Use of setjmp/longjmp

CWE ID	Polyspace Bug Finder Defect
704: Incorrect Type Conversion or Cast	Qualifier removed in conversion Unreliable cast of pointer Wrong allocated object size for cast
732: Incorrect Permission Assignment for Critical Resource	Vulnerable permission assignments
755: Improper Handling of Exceptional Conditions	Exception handler hidden by previous handler
762: Mismatched Memory Management Routines	Invalid free of pointer
764: Multiple Locks of a Critical Resource	Double lock
765: Multiple Unlocks of a Critical Resource	Double unlock
767: Access to Critical Private Variable via Public Method	Return of non const handle to encapsulated data member
770: Allocation of Resources Without Limits or Throttling	Tainted size of variable length array
772: Missing Release of Resource after Effective Lifetime	Resource leak
783: Operator Precedence Logic Error	Possibly unintended evaluation of expression because of operator precedence rules
785: Use of Path Manipulation Function without Maximum-sized Buffer	Use of path manipulation function without maximum sized buffer checking
786: Access of Memory Location Before Start of Buffer	Destination buffer underflow in string manipulation
787: Out-of-bounds Write	Destination buffer overflow in string manipulation Destination buffer underflow in string manipulation Use of tainted pointer

CWE ID	Polyspace Bug Finder Defect
789: Uncontrolled Memory Allocation	Memory allocation with tainted size Tainted size of variable length array Unprotected dynamic memory allocation
822: Untrusted Pointer Dereference	Tainted NULL or non-null-terminated string Use of tainted pointer
823: Use of Out-of-range Pointer Offset	Pointer access out of bounds Pointer dereference with tainted offset
824: Access of Uninitialized Pointer	Non-initialized pointer
832: Unlock of a Resource that is not Locked	Missing lock
833: Deadlock	Deadlock
835: Loop with Unreachable Exit Condition	Loop bounded with tainted value
843: Access of Resource Using Incompatible Type ("Type Confusion")	Unreliable cast of pointer
873: CERT C++ Secure Coding Section 05 - Floating Point Arithmetic (FLP)	Invalid use of floating point operation Invalid use of standard library floating point routine Float overflow
908: Use of Uninitialized Resource	Member not initialized in constructor Non-initialized pointer Non-initialized variable

Command-Line Analysis

- “Create Project Automatically at Command Line” on page 6-2
- “Run Local Analysis from Command Line” on page 6-4
- “Run Remote Analysis at Command Line” on page 6-6
- “Create Project Automatically from MATLAB Command Line” on page 6-10

Create Project Automatically at Command Line

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. For more information on these options, see:
 - C Code: “Target & Compiler”
 - C++ Code: “Target & Compiler”

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 targetName buildOptions
```

For the list of options allowed with the GNU `make`, see `make options`.

- 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 targetName buildOptions
```

Use the options file to run verification:

```
polyspace-bug-finder-nodesktop -options-file myOptions
```

You can also use advanced options to modify the default behavior of `polyspace-configure`. For more information, see the `-options value` argument for `polyspaceConfigure`.

More About

- “Requirements for Project Creation from Build Systems” on page 1-10
- “Compiler Not Supported for Project Creation from Build Systems” on page 1-13
- “Slow Build Process When Polyspace Traces the Build” on page 1-20
- “Checking if Polyspace Supports Windows Build Command” on page 1-21

Run Local Analysis from 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]`

In this section...

“Specify Sources and Analysis Options Directly” on page 6-4

“Specify Sources and Analysis Options in Text File” on page 6-5

“Create Options File from Build System” on page 6-5

Specify Sources and Analysis Options Directly

At the Windows, Linux or Mac OS X command-line, append sources and 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
```

For the full list of analysis options, see “Analysis Options for C” or “Analysis Options for C++”.

You can also enter the following at the command line:

```
polyspace-bug-finder-nodesktop -help
```

Specify Sources and Analysis Options in Text File

- 1 Create an options file called `listoptions.txt` with your options. For example:

```
#These are the options for MyBugFinderProject
-lang c
-prog MyBugFinderProject
-author jsmith
-sources "mymain.c,funAlgebra.c,funGeometry.c"
-OS-target no-predefined-OS
-target x86_64
-dialect none
-dos
-misra2 required-rules
-includes-to-ignore all-headers
-checkers default
-disable-checkers concurrency
-results-dir C:\Polyspace\MyBugFinderProject
```

- 2 Run Polyspace using options in the file `listoptions.txt`.

```
polyspace-bug-finder-nodesktop -options-file listoptions.txt
```

Create Options File from Build System

- 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 interface.

```
polyspace-bug-finder myResults
```

Run Remote Analysis at Command Line

Before you run a remote analysis, you must set up a server for this purpose. For more information, see “Set Up Server for Metrics and Remote Analysis”.

In this section...
“Run Remote Analysis” on page 6-6
“Manage Remote Analysis” on page 6-7
“Download Results” on page 6-9

Run Remote Analysis

Use the following command to run a remote verification:

```
MATLAB_Install\polyspace\bin\polyspace-bug-finder-nodesktop  
-batch -scheduler NodeHost | MJSName@NodeHost [options]
```

where:

- *MATLAB_Install* is your MATLAB installation folder.
- *NodeHost* is the name of the computer that hosts the head node of your MATLAB Distributed Computing Server™ cluster.
- *MJSName* is the name of the MATLAB Job Scheduler (MJS) on the head node host.
- *options* are the analysis options. These options are the same as that of a local analysis. For more information, see “Run Local Analysis from Command Line” on page 6-4.

After compilation, the software submits the verification job to the cluster and provides you a job ID. Use the `polyspace-jobs-manager` command with the job ID to monitor your verification and download results after verification is complete. For more information, see:

- “Manage Remote Analysis” on page 6-7
- “Download Results” on page 6-9

Tip In Windows, to avoid typing the commands each time, you can save the commands in a batch file.

- 1 Save your analysis options in a file `listoptions.txt`. See “Specify Sources and Analysis Options in Text File” on page 6-5.
To specify your sources, in the options file, instead of `-sources`, use `-sources-list-file`. This option is available only for remote analysis and allows you to specify your sources in a separate text file.
- 2 Create a file `launcher.bat` in a text editor like Notepad.
- 3 Enter the following commands in the file.


```
echo off
set POLYSPACE_PATH=C:\Program Files\MATLAB\R2015a\polyspace\bin
set RESULTS_PATH=C:\Results
set OPTIONS_FILE=C:\Options\listoptions.txt
"%POLYSPACE_PATH%\polyspace-bug-finder-nodesktop.exe" -batch -scheduler localhost
               -results-dir "%RESULTS_PATH%" -options-file "%OPTIONS_FILE%"
pause
```
- 4 Replace the definitions of the following variables in the file:
 - `POLYSPACE_PATH`: Enter the actual location of the `.exe` file.
 - `RESULTS_PATH`: Enter the path to a folder. The files generated during compilation are saved in the folder.
 - `OPTIONS_FILE`: Enter the path to the file `listoptions.txt`.
 Replace `localhost` with the name of the computer that hosts the head node of your MATLAB Distributed Computing Server cluster.
- 5 Double-click `launcher.bat` to run the verification.

If you run a Polyspace verification, a `.bat` file is automatically generated for you. You can relaunch verification using this file.

Manage Remote Analysis

To manage remote analyses, use this command:

```
MATLAB_Install\polyspace\bin\polyspace-jobs-manager action [options]
               [-scheduler schedulerOption]
```

where:

- *MATLAB_Install* is your MATLAB installation folder

- `schedulerOption` is one of the following:
 - Name of the computer that hosts the head node of your MATLAB Distributed Computing Server 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**.

- *action [options]* refer to the possible action commands to manage jobs on the scheduler:

Action	Options	Task
<code>listjobs</code>	None	Generate a list of Polyspace jobs on the scheduler. For each job, the software produces the following information: <ul style="list-style-type: none"> • <code>ID</code> — Verification or analysis identifier. • <code>AUTHOR</code> — Name of user that submitted job. • <code>APPLICATION</code> — Name of Polyspace product, for example, Polyspace Code Prover or Polyspace Bug Finder. • <code>LOCAL_RESULTS_DIR</code> — Results folder on local computer, specified through the Tools > Preferences > Server Configuration tab. • <code>WORKER</code> — Local computer from which job was submitted. • <code>STATUS</code> — Status of job, for example, running and completed. • <code>DATE</code> — Date on which job was submitted. • <code>LANG</code> — Language of submitted source code.

Action	Options	Task
download	- job <i>ID</i> -results-folder <i>FolderPath</i>	Download results of analysis with specified ID to folder specified by <i>FolderPath</i> .
getlog	- job <i>ID</i>	Open log for job with specified ID.
remove	- job <i>ID</i>	Remove job with specified ID.

Download Results

To download verification results from the command line, use the `polyspace - jobs - manager` command:

```
MATLAB_Install\polyspace\bin\polyspace-jobs-manager -download  
-job Verification_ID -results-folder FolderPath
```

After downloading results, use the Polyspace user interface to view the results. See “Open Results”.

Create Project Automatically from MATLAB Command Line

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. For more information on these options, see:
 - C Code: “Target & Compiler”
 - C++ Code: “Target & Compiler”

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.pspj` from your makefile:

```
polyspaceConfigure -prog myProject ...  
                   make 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 targetName buildOptions
```

Use the options file to run verification:

```
polyspaceBugFinder -options-file myOptions
```

You can also use advanced options to modify the default behavior of `polyspaceConfigure`. For more information, see `polyspaceConfigure`.

More About

- “Requirements for Project Creation from Build Systems” on page 1-10
- “Compiler Not Supported for Project Creation from Build Systems” on page 1-13
- “Slow Build Process When Polyspace Traces the Build” on page 1-20

Polyspace Bug Finder Analysis in Simulink

- “Embedded Coder Considerations” on page 7-2
- “TargetLink Considerations” on page 7-5
- “Generate and Analyze Code” on page 7-7
- “Main Generation for Model Analysis” on page 7-14
- “Review Generated Code Results” on page 7-16
- “Troubleshoot Back to Model” on page 7-18

Embedded Coder Considerations

In this section...

“Default Options” on page 7-2

“Recommended Polyspace Bug Finder Options for Analyzing Generated Code” on page 7-3

“Hardware Mapping Between Simulink and Polyspace” on page 7-4

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_rtvec __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
-functions-to-stub=[rtIsNaN,rtIsInf,rtIsNaNF,rtIsInFF]
```

Default options for C++:

```
-sources path_to_source_code
-results-dir results
-D PST_ERRNO
-D main=main_rtvec __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-preddefined-OS
-dialect iso
-ignore-constant-overflows true
-scalar-overflows-behavior wrap-around
-allow-negative-operand-in-shift true
-functions-to-stub=[rtIsNaN,rtIsInf,rtIsNaNF,rtIsInfF]
```

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 Polyspace **Configuration** pane opens.

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
Macros > Preprocessor definitions -D	See comments	Defines macro compiler flags used during compilation. Some defines are applied by default, depending on your <code>-OS-target</code> . Use one <code>-D</code> for each line of the Embedded Coder generated <code>defines.txt</code> file. Polyspace does not do this by default.
Target & Compiler > Target operating system -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.

Option	Recommended Value	Comments
Environment Settings > Code from DOS or Windows file system - dos	On	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: <ul style="list-style-type: none"> • 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 7-5

“Default Options” on page 7-5

“Lookup Tables” on page 7-6

“Code Generation Options” on page 7-6

TargetLink Support

For Windows, Polyspace Bug Finder is tested with releases 3.5 and 4.0 of the dSPACE® Data Dictionary and TargetLink® Code Generator.

As Polyspace Bug Finder extracts information from the dSPACE Data Dictionary, you must regenerate the code before performing an analysis.

Default Options

The following default options are set by Polyspace:

```
-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
-functions-to-stub=[rtIsNaN,rtIsInf,rtIsNaNF,rtIsInfF]
-OS-target no-predfined-OS
-ignore-constant-overflows
-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).

Related Examples

- “Run Analysis for TargetLink” on page 10-6

External Websites

- dSPACE – TargetLink

Generate and Analyze Code

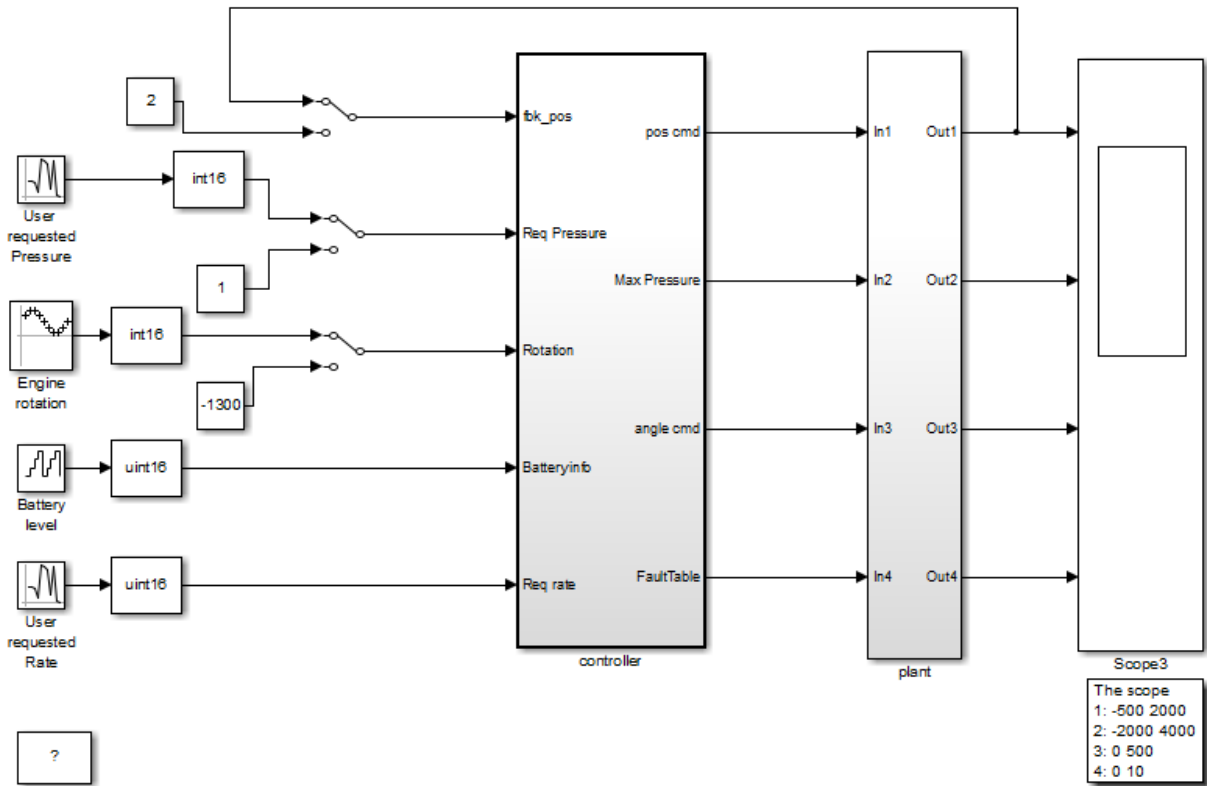
This example shows how to use Polyspace Bug Finder to generate code from submodels and S-Functions, run a Polyspace analysis from Simulink, and find code defects and MISRA-C:2012 rule violations.

Generate Code and Run Analysis

Before running Polyspace on models, define the scope of your analysis and generate code in Embedded Coder.

1. Open the example model.

```
psdemo_model_link_sl
```



Copyright 2010-2015 The MathWorks, Inc.

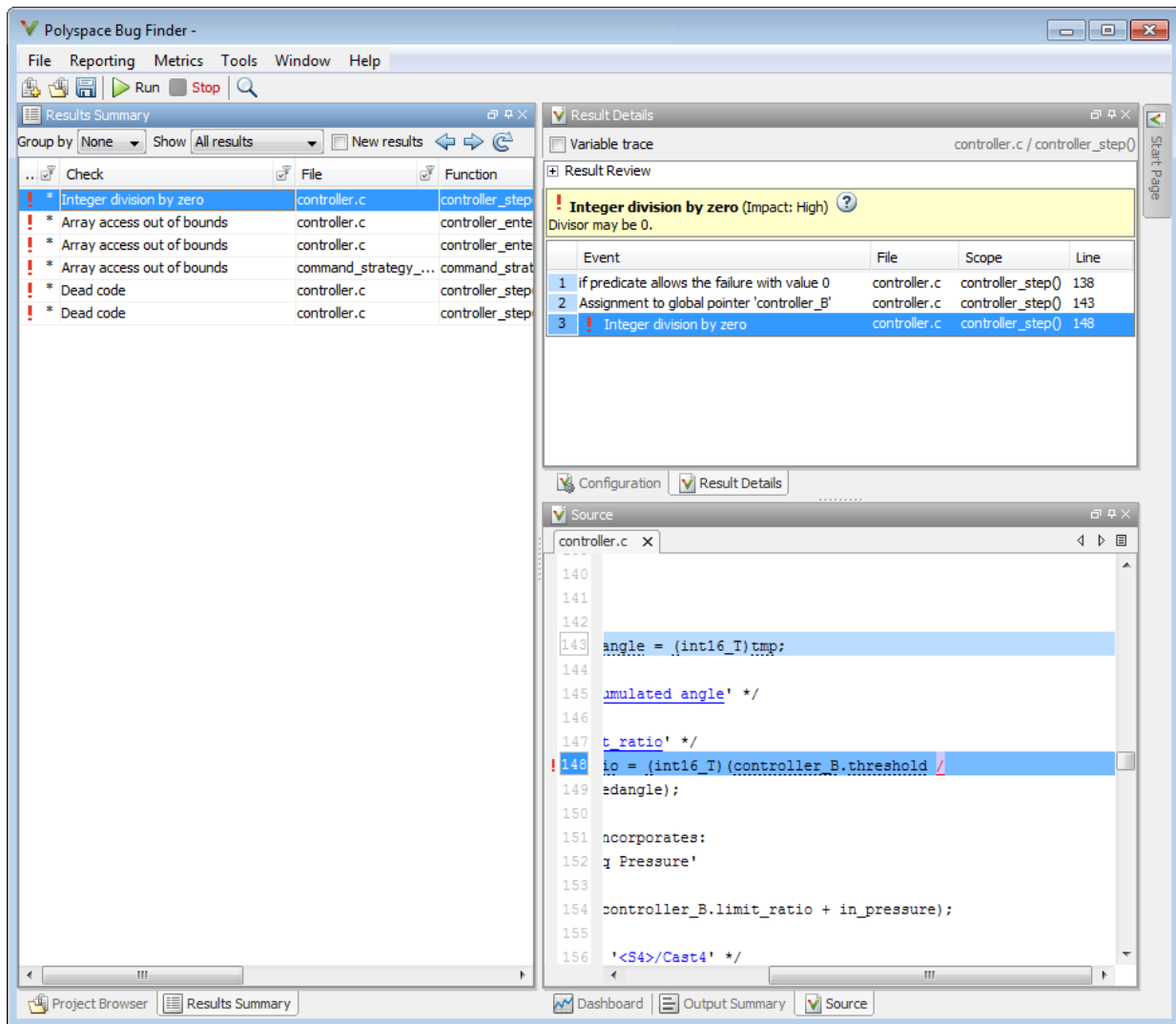
2. Right-click the **controller** subsystem.
3. From the context menu, select **C/C++ Code > Build This Subsystem**.
4. In the dialog box, select **Build**.
5. After the build is completed, right-click the **controller** subsystem.
6. From the context menu, select **Polyspace > Options**
7. In the Configuration Parameters window, select **Product Mode > Bug Finder**.

8. Apply your changes and close the Configuration Parameters window.
9. Right-click the `controller` subsystem.
10. Select **Polyspace > Verify code generated for > Selected subsystem.**

You can monitor progress from the Command Window. The results are displayed in the Polyspace environment.

Review Results

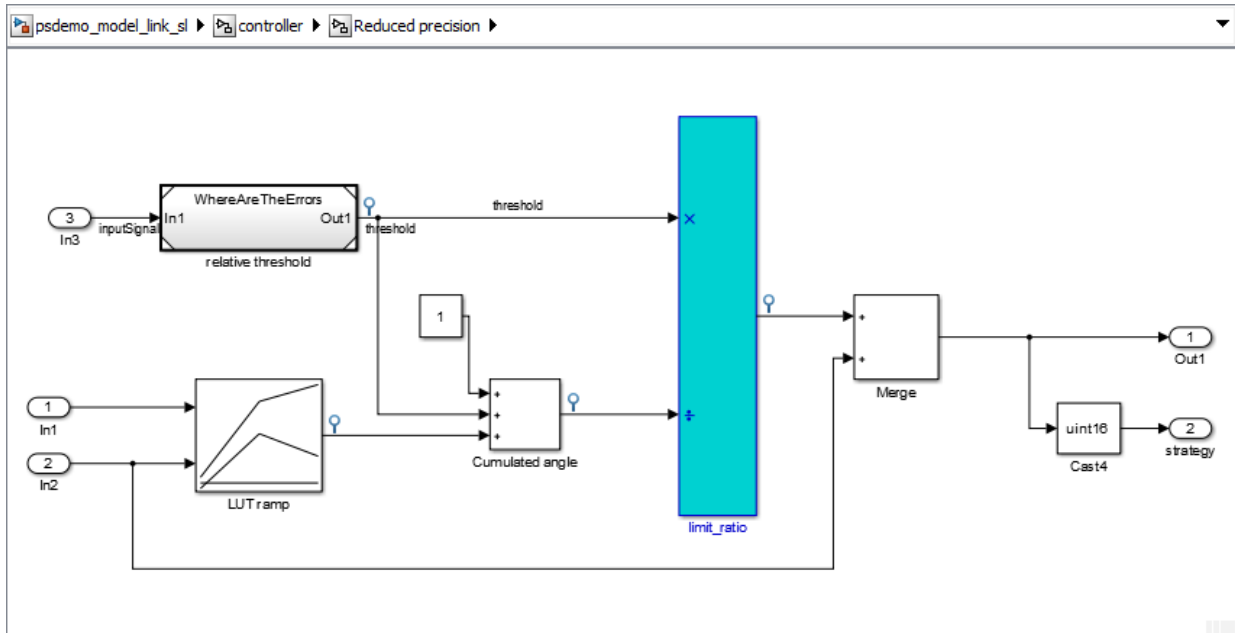
In the Polyspace Environment, explore your results and link back to the model.



1. Select the first result Integer division by zero.

This result shows a possible division by zero. The Source pane shows the division operation between variables `controller_B.threshold` and `controller_B.Cumulatedangle`.

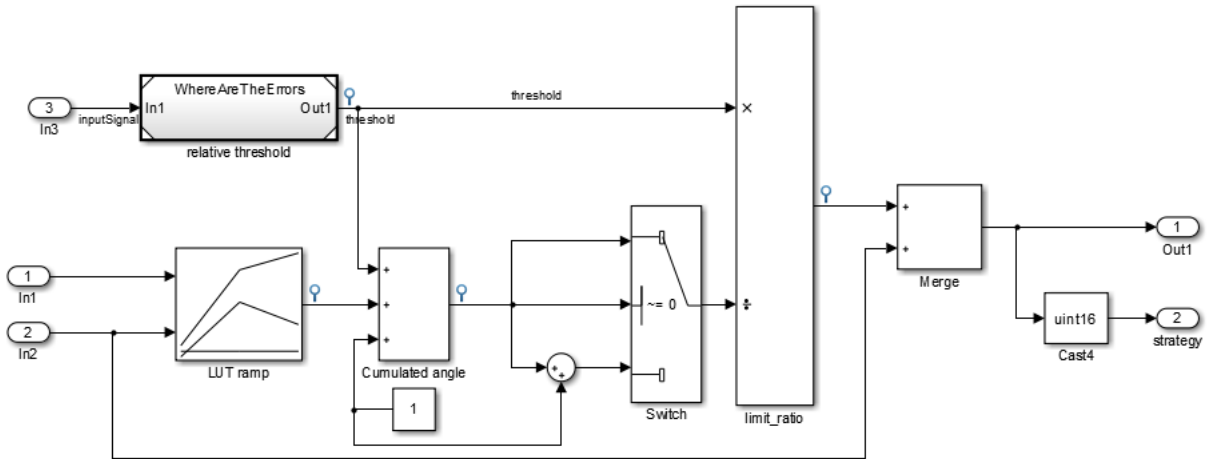
2. To see this division operation in your model, select the link `<S4>/limit_ratio`. In your model, the related block is highlighted in blue.



Fix Errors by Modifying the Model

The division by zero error stems from the **Cumulated angle** block, whose signal can be zero. To fix the error in the code, modify this block in your model.

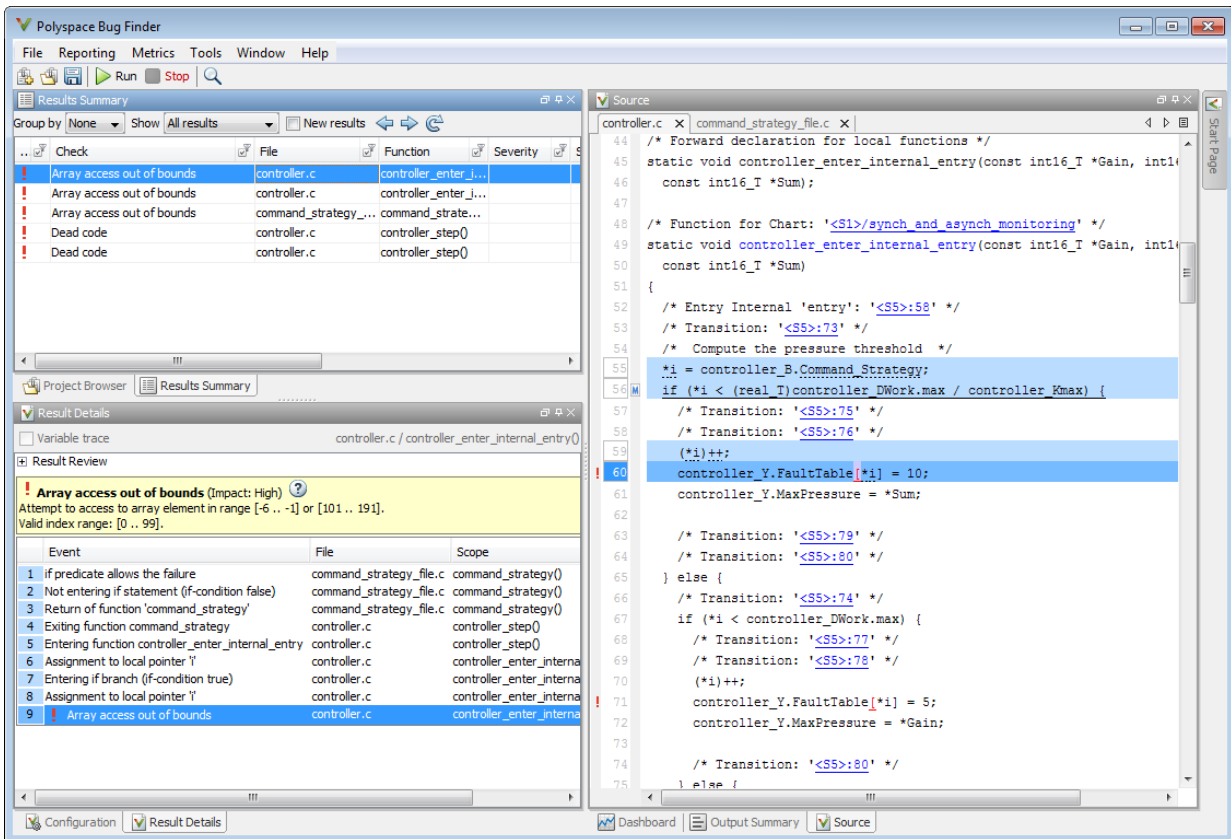
1. Before dividing with the Cumulated angle, add a switch block that checks for values equal to zero.



2. Rebuild the controller subsystem.

3. Rerun the Bug Finder analysis.

The results show that your fix in the model eliminated the division by zero defect.



Related Examples

- “Polyspace Configuration for Generated Code” on page 9-2
- “Run Analysis for Embedded Coder” on page 10-5
- “Run Analysis for TargetLink” on page 10-6

More About

- “Recommended Model Settings for Code Analysis” on page 8-3
- “Troubleshoot Back to Model” on page 7-18

Main Generation for Model Analysis

When you run an analysis, the software automatically reads the following information from the model:

- `initialize()` functions
- `terminate()` functions
- `step()` functions
- List of parameter variables
- List of input variables

The software then uses this information to generate a `main` function that:

- 1 Initializes parameters using the Polyspace option `-variables-written-before-loop`.
- 2 Calls initialization functions using the option `-functions-called-before-loop`.
- 3 Initializes inputs using the option `-variables-written-in-loop`.
- 4 Calls the `step` function using the option `-functions-called-in-loop`.
- 5 Calls the `terminate` function using the option `-functions-called-after-loop`.

If the `codeInfo` for the model does not contain the names of the inputs, the software considers all variables as entries, except for parameters and outputs.

For C++ code that is generated with Embedded Coder, the `initialize()`, `step()`, and `terminate()` functions are either class methods or have global scope. These different scopes contain the associated variables.

- For class methods in the generated code, the variables that are written before and in the loop refer to the class members.
- For functions with global scope, the associated variables are also in the global scope.

main for Generated Code

The following example shows the `main` generator options that the software uses to generate the `main` function for code generated from a Simulink model.

```
init parameters    \\ -variables-written-before-loop
init_fct()        \\ -functions-called-before-loop
while(1){         \\ start main loop
  init inputs     \\ -variables-written-in-loop
```



```
    step_fct()      \\ -functions-called-in-loop  
  }  
  terminate_fct()  \\ -functions-called-after-loop
```

Review Generated Code Results

After you run a Polyspace analysis on generated code, you review the results from the Polyspace environment. From the results you can link back to the related blocks in your model.

1 Open the results using one of the following methods.

- If you analyzed the whole model, from the Simulink toolbar, 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 showing 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**.

- If you want to open results for a Model block or subsystem, right-click the Model block or subsystem, and from the context menu, select **Polyspace > Open Results**.
- From the Polyspace Interface, select **File > Open** and navigate to your results.
- If you selected **Add to results repository** the results are stored on the Polyspace Metrics server. See “Download Results From Polyspace Metrics” on page 5-6.

2 On the **Results Summary** tab, select a result.

When you select a result, the **Result Details** pane shows additional information about the defect, including traceback information (if available).

- 3 Look at the result in the **Source** pane. Your select result is highlighted in the source code.
- 4 Hover over the result in the source code. The tooltip can provide additional information including variable ranges.
- 5 Above the defect, click a blue underlined link. For example, <Root>/Relational Operator.

The Simulink model opens, highlighting the block related to the nearby source code. This back-to-model linking allows you to fix defects in the model instead of the generated code.

Related Examples

- “View Results”
- “Polyspace Bug Finder Results”

More About

- “Troubleshoot Back to Model” on page 7-18

Troubleshoot Back to Model

In this section...

“Back-to-Model Links Do Not Work” on page 7-18

“Your Model Already Uses Highlighting” on page 7-18

Back-to-Model Links Do Not Work

You may encounter issues with the back-to-model feature if:

- Your operating system is Windows Vista™ or Windows 7; and User Account Control (UAC) is enabled or you do not have administrator privileges.
- You have multiple versions of MATLAB installed.

To reconnect MATLAB and Polyspace:

- 1 Close Polyspace.
- 2 At the MATLAB command-line, enter `PolySpaceEnableCOMserver`.

When you open your Polyspace results, the hyper-links will highlight the relevant blocks in your model.

Your Model Already Uses Highlighting

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:

```
HILITE_DATA = struct('HiliteType', 'find', 'ForegroundColor', 'black', ...  
                    'BackgroundColor', color);
```

```
set_param(0, 'HiliteAncestorsData', HILITE_DATA)
```

Where *color* is one of the following:

- 'cyan'
- 'magenta'
- 'orange'
- 'lightBlue'

- 'red'
- 'green'
- 'blue'
- 'darkGreen'

Configure Model for Code Analysis

- “Configure Simulink Model” on page 8-2
- “Recommended Model Settings for Code Analysis” on page 8-3
- “Check Simulink Model Settings” on page 8-6
- “Annotate Blocks for Known Results” on page 8-12

Configure Simulink Model

Before analyzing your generated code, there are certain settings that you should apply to your model. Use the following workflow to prepare your model for code analysis.

- If you know of results ahead of time, annotate your blocks with Polyspace annotations.
- Set the recommended configuration parameters.
- Double-check your model settings.
- Generate code.
- Set up your Polyspace options.

Recommended Model Settings for Code Analysis

For Polyspace analyses, set the following parameter configurations before generating code. If you do not use the recommended value for `SystemTargetFile`, you get an error. For all other parameters, if you do not use the recommended value, you get a warning.

Grouping	Parameter	Recommended value	Name and Location in Configuration
Code Generation	<code>SystemTargetFile</code>	An Embedded Coder Target Language Compiler (TLC) file. For example <code>ert.tlc</code> or <code>autosar.tlc</code> .	Location: Code Generation Name: System target file Value: Embedded Coder target file
	<code>MatFileLogging</code>	'off'	Location: Code Generation > Interface Name: MAT-file logging Value: <input type="checkbox"/> Not selected
	<code>GenerateReport</code>	'on'	Location: Code Generation > Report Name: Create code-generation report Value: <input checked="" type="checkbox"/> Selected
	<code>IncludeHyperlinksInReport</code>	'on'	Location: Code Generation > Report

Grouping	Parameter	Recommended value	Name and Location in Configuration
			Name: Code-to-model Value: <input checked="" type="checkbox"/> Selected
	GenerateSampleERTMain	'off'	Location: Code Generation > Templates Name: Generate an example main program Value: <input type="checkbox"/> Not selected
	GenerateComments	'on'	Location: Code Generation > Comments Name: Include comments Value: <input checked="" type="checkbox"/> Selected
Optimization	"Default parameter behavior"	'Inlined'	Location: Optimization > Signals and Parameters Name: Default parameter behavior Value: Inlined

Grouping	Parameter	Recommended value	Name and Location in Configuration
	InitFltsAndDblsToZero	'on'	Location: Optimization Name: Use memset to initialize floats and doubles to 0.0 Value: <input type="checkbox"/> Not selected
	ZeroExternalMemoryAtStartup	'on' when Configuration Parameters > Polyspace > Data Range Management > Output is Global assert	Location: Optimization Name: Remove root level I/O zero initialization Value: <input type="checkbox"/> Not selected
Solver	SolverType	'Fixed-Step'	Location: Solver Name: Type Value: Fixed-step
	Solver	'FixedStepDiscrete'	Location: Solver Name: Solver Value: discrete (no continuous states)

Check Simulink Model Settings

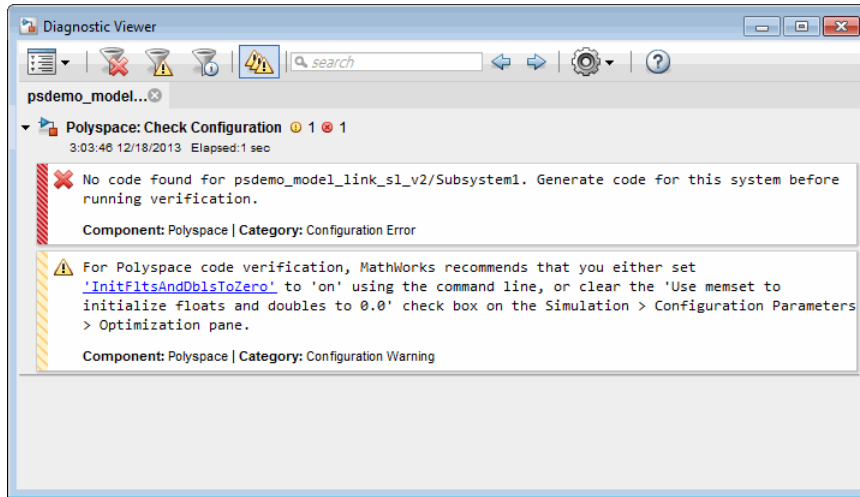
With the Polyspace plug-in, you can check your model settings before generating code or before starting an analysis. If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, warnings appear when you run the analysis.

Check Simulink Model Settings Using the Code Generation Advisor

Before generating code, you can check your model settings against the “Recommended Model Settings for Code Analysis” on page 8-3. If you do not use the recommended model settings, the back-to-model linking will not work correctly.

- 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** In the **Check model before generating code** drop-down list, select either:
 - **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.
- 5** Select **Check Model**.

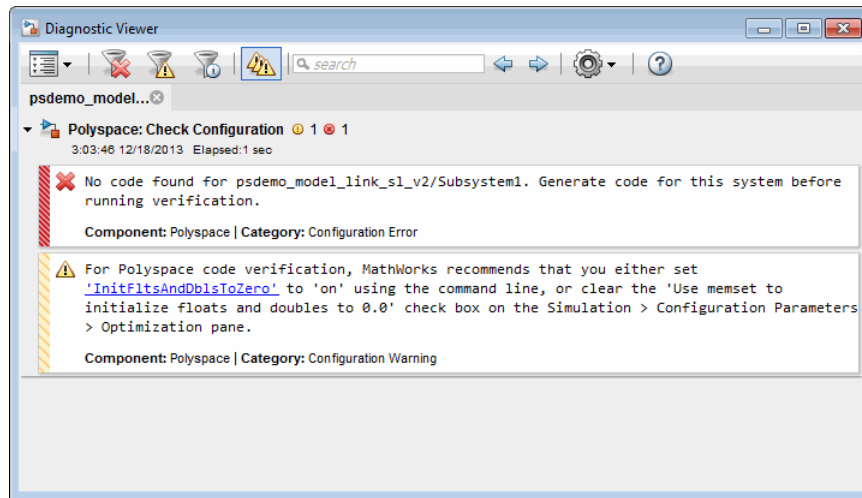
The software runs a configuration check. If your configuration check finds errors or warnings, the **Diagnostics Viewer** displays the issues and recommendations.



Check Simulink Model Settings Before Analysis

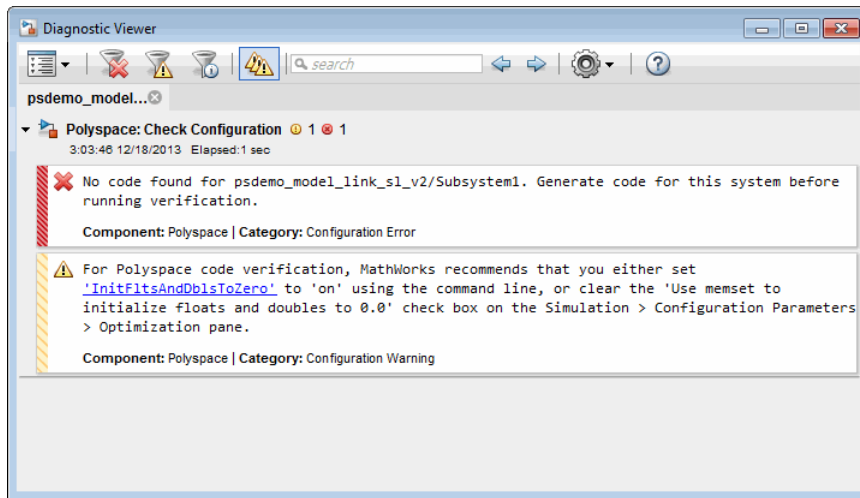
With the Polyspace plug-in, you can check your model settings before starting an analysis:

- 1 From the Simulink model window, select **Code > Polyspace > Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.
- 2 Click **Check configuration**. If your model settings are not optimal for Polyspace, the software displays warning messages with recommendations.



- 3 From the **Check configuration before verification** menu, select either:
 - 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.
- 4 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.

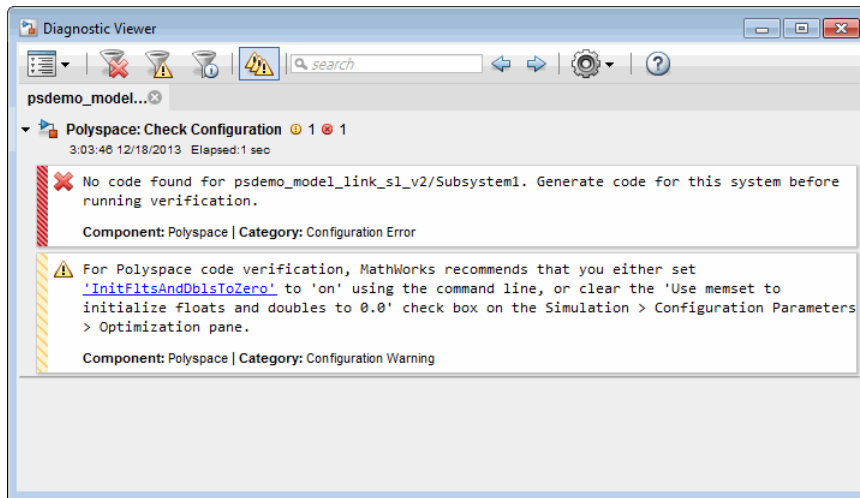


If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, the software produces warnings when you run the analysis.

Check Simulink Model Settings Automatically

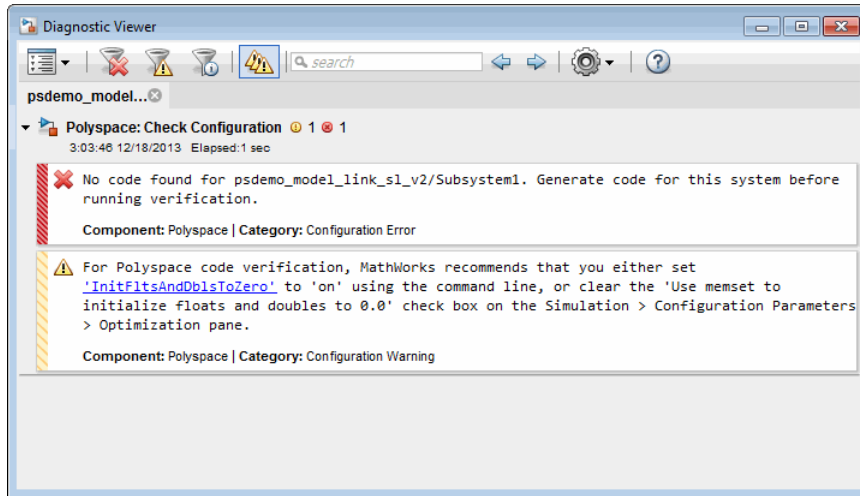
With the Polyspace plug-in, you can check your model settings before starting an analysis:

- 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.



- 3 From the **Check configuration before verification** menu, select either:
 - On (stop for warnings) — will
 - On (proceed with warnings)
- 4 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.



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.

More About

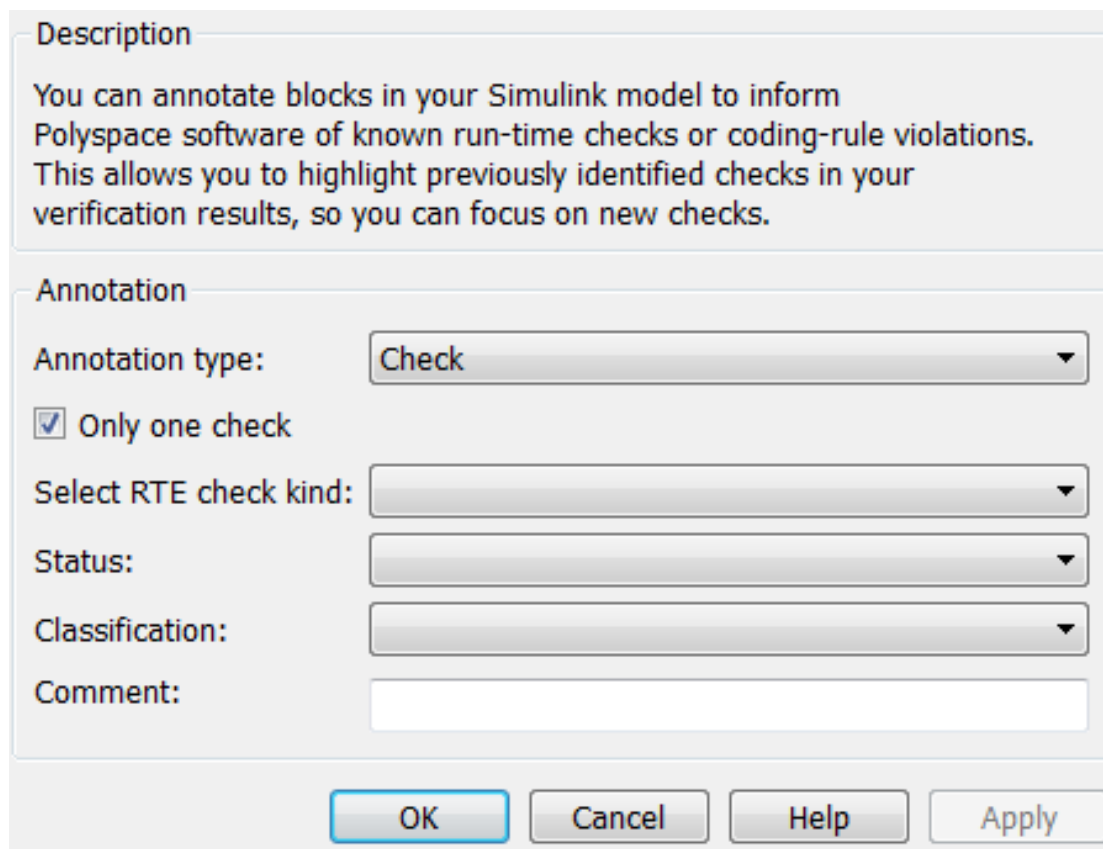
- “Recommended Model Settings for Code Analysis” on page 8-3

Annotate Blocks for Known Results

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.

Your Polyspace results displays the information that you provide with block annotations. To annotate blocks:

- 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.



The image shows the Polyspace Annotation dialog box. It has a 'Description' section with text explaining the purpose of annotations. Below that is an 'Annotation' section with several fields: 'Annotation type' (set to 'Check'), a checked 'Only one check' checkbox, 'Select RTE check kind', 'Status', 'Classification', and a 'Comment' text area. At the bottom are 'OK', 'Cancel', 'Help', and 'Apply' buttons.

Description

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

Annotation type:

Only one check

Select RTE check kind:

Status:

Classification:

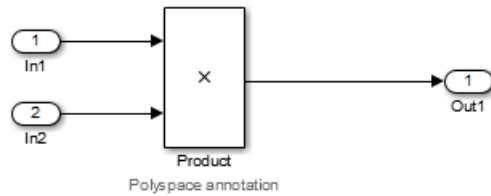
Comment:

- 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**
 - **Justified**
- (This status also marks the result as justified.)
- **No action planned**
- (This status also marks the result as justified.)
- **Other**
- 6** Select a **Severity** to describe the severity of the issue:
- **High**
 - **Medium**
 - **Low**
 - **Not a defect**
- 7** In the **Comment** field, enter additional information about the check.

- Click **OK**. The software adds the Polyspace annotation is to the block.



When you run an analysis, the **Results Summary** pane pre-populates the results with your annotation.

Results Summary						
Group by		Show		New results		
None		All results		<input type="checkbox"/>		
Family	Check	File	Function	Classification	Status	Comment
!	Dead code	controller.c	controller_step()			
!	Dead code	controller.c	controller_step()			
!	Integer division by zero	controller.c	controller_step()	Medium	Improve	Remove zero
!	Array access out of bounds	controller.c	controller_enter_internal_entry()			
!	Array access out of bounds	command_strategy_file.c	command_strategy()			

See Also
pslinkfun

Configure Code Analysis Options

- “Polyspace Configuration for Generated Code” on page 9-2
- “Include Handwritten Code” on page 9-3
- “Configure Analysis Depth for Referenced Models” on page 9-4
- “Check Coding Rules Compliance” on page 9-5
- “Configure Polyspace Analysis Options and Properties” on page 9-7
- “Set Custom Target Settings” on page 9-11
- “Set Up Remote Batch Analysis” on page 9-14
- “Manage Results” on page 9-15
- “Specify Signal Ranges” on page 9-18

Polyspace Configuration for Generated Code

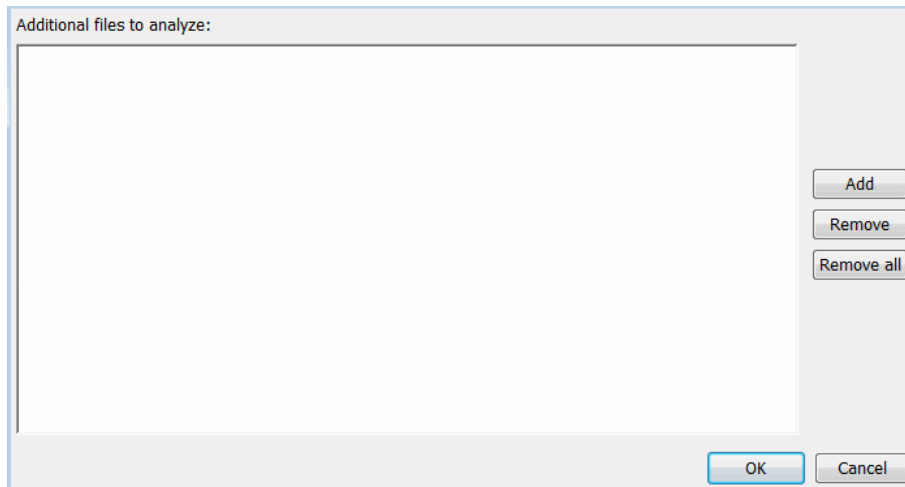
You do not have to manually create a Polyspace project or specify Polyspace options before running an analysis for your generated code. By default, Polyspace automatically creates a project and extracts the required information from your model. However, you can modify or specify additional options for your analysis:

- You may incorporate separately created code within the code generated from your Simulink model. See “Include Handwritten Code” on page 9-3.
- You may customize the options for your analysis. For example, to specify the target environment or adjust precision settings. See “Configure Polyspace Analysis Options and Properties” on page 9-7 and “Recommended Polyspace Bug Finder Options for Analyzing Generated Code” on page 7-3.
- You may create specific configurations for batch runs. See “Save a Polyspace Configuration File Template” on page 9-8.
- If you want to analyze code generated for a 16-bit target processor, you must specify header files for your 16-bit compiler. See “Set Custom Target Settings” on page 9-11.

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**.

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 10-5.

Check Coding Rules Compliance

You can check compliance with MISRA AC AGC and MISRA C:2004, and MISRA C:2012 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 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 C 2004 checking	Run Polyspace using the options specified in the Project configuration and check compliance with MISRA C:2004 coding rules.
Project configuration and MISRA C 2012 AGC checking	Run Polyspace using the options specified in the Project configuration and check compliance with MISRA C:2012 coding guidelines.
MISRA AC AGC checking	Check compliance with the MISRA AC-AGC rule set. Polyspace stops after rules checking.

Setting	Description
MISRA C 2004 checking	Check compliance with MISRA C:2004 coding rules. Polyspace stops after rules checking.
MISRA C 2012 ACG checking	Check compliance with MISRA C:2012 coding rules using generated code categories. Polyspace stops after guideline 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 Analysis Options and Properties

From Simulink, you can specify Polyspace options to change the configuration of the Polyspace Analysis. For example, you can specify the processor type and operating system of your target device.

For descriptions of options, see “Analysis Options for C” or “Analysis Options for C++”.

There are two ways to configure analysis options:

In this section...

“Set Advanced Analysis Options” on page 9-7

“Save a Polyspace Configuration File Template” on page 9-8

“Use a Custom Configuration File” on page 9-9

“Remove Polyspace Options From Simulink Model” on page 9-9

Set Advanced Analysis Options

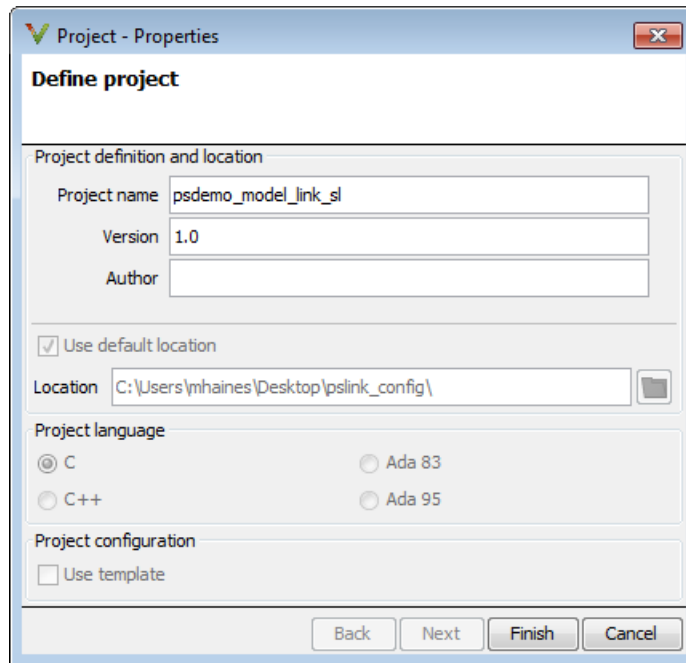
- 1 From Simulink, select **Code > Polyspace > Options**.
- 2 In the Polyspace parameter configuration pane, select **Configure**.

The Polyspace Configuration window opens.

- 3 Set options required by your application.

The first time you open the configuration, the software sets certain options by default depending on your code generator.

- 4 On the toolbar, click the **Project properties** icon .



Save 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 Parameter Configuration window opens to the Polyspace pane.
- 2 Click **Configure**.

The Polyspace Configuration window opens. Use this pane to customize the target and cross compiler.

- 3 Save your changes and close.
- 4 Make a copy of the updated project configuration file, for example, `my_first_code_polyspace.psprj`.

- 5 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')
```
- Add the file in the Parameter Configuration window. See “Use a Custom Configuration File” on page 9-9.

Use a Custom Configuration File

If you already have a configuration you want to use, you can add the configuration file to your project.

- 1 From Simulink, select **Code > Polyspace > Options**.
- 2 In the Polyspace parameter configuration pane, select **Use custom project file**.
- 3 In the text box, enter the full path to a `.psprj` file, or click **Browse for project file** to browse for a `.psprj` file.

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.

See Also

`pslinkfun` | `pslinkoptions`

Related Examples

- “Save a Polyspace Configuration File Template” on page 9-8

More About

- “Embedded Coder Considerations” on page 7-2
- “TargetLink Considerations” on page 7-5
- “Recommended Polyspace Bug Finder Options for Analyzing Generated Code” on page 7-3

Set Custom Target Settings

If your target has specific setting, you can analyze your code in context of those settings. For example, 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.

1 In the Simulink model window, select **Code > Polyspace > Options**. The Parameter Configuration window opens to the Polyspace pane.

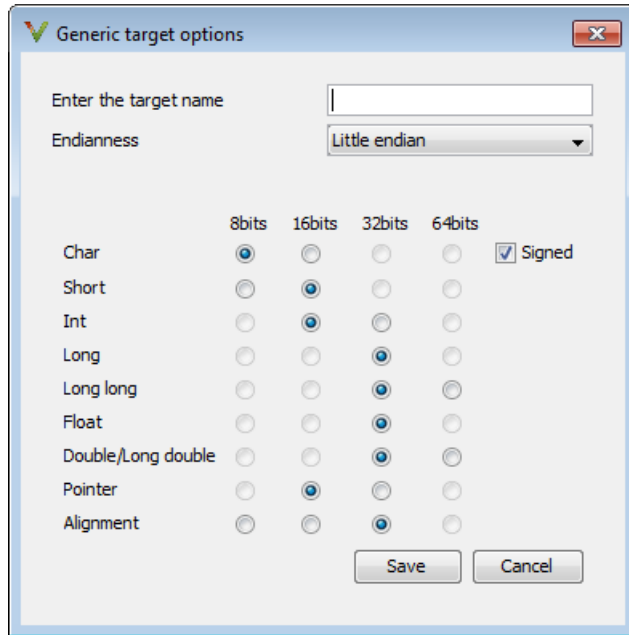
2 Click **Configure**.

The Polyspace Configuration window opens. 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.



Use this dialog box to create a new target and specify data types for the target. Then click **Save**.

- 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, select . In the new line, enter the required text.


To remove a macro, select the macro and click .

Note: If you use the LCC cross-compiler, then you must specify the `MATLAB_MEX_FILE` macro.

- Select **Target & Compiler > Environment Settings**.

7 In the **Include folders** (or **Include**) section, specify a folder (or header file) path by doing one of the following:

- Select  and enter the folder or file path.
- Select  and use the 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 .

8 Save your changes and close.

To use your configuration settings in other projects, see “Save a Polyspace Configuration File Template” on page 9-8.

Set Up Remote Batch 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** check box.
- 5 If you use Polyspace Metrics as a results repository, select **Add to results repository**.

Before running your must also make sure you are connected to a Server.

- 6 From the toolbar, select **Options > Preferences**. For help filling in this dialog, see “Configure Polyspace Preferences”.
- 7 Close the configuration window and save your changes.
- 8 Select **Apply**.

Manage Results

In this section...
“Open Polyspace Results Automatically” on page 9-15
“Specify Location of Results” on page 9-16
“Save Results to a Simulink Project” on page 9-17

Polyspace creates a set of analysis results

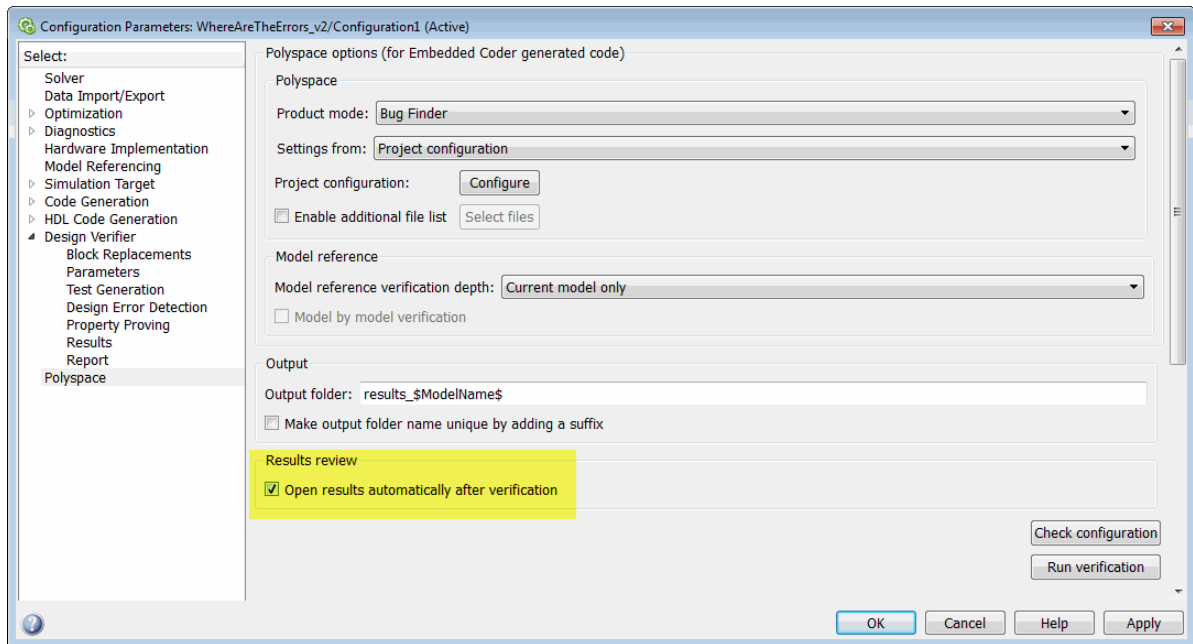
Open Polyspace Results Automatically

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 Polyspace interface.

To configure the results to open automatically:

- 1 From the model window, select **Code > Polyspace > Options**.

The Polyspace pane opens.



- 2 In the Results review section, select **Open results automatically after verification**.
- 3 Click **Apply** to save your settings.

Specify Location of Results

By default, the software stores your results in *Current Folder*\results_model_name. Every time you rerun, your old results are overwritten. To customize these options:

- 1 From the Simulink model window, select **Code > Polyspace > Options**. The Configuration Parameters dialog box opens to the Polyspace pane.
- 2 In the **Output folder** field, specify a full path for your results folder. By default, the software stores results in the current folder.
- 3 If you want to avoid overwriting results from previous analyses, select **Make output folder name unique by adding a suffix**.

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.

Save Results to a Simulink Project

By default, the software stores your results in *Current Folder\results_model_name*. If you use a Simulink project for your model work, you can store your Polyspace results there as well for better organization. To add your results to a Simulink Project:

- 1 Open your Simulink project.
- 2 From the Simulink model window, select **Code > Polyspace > Options**. The Configuration Parameters dialog box opens with the Polyspace pane displayed.
- 3 Select **Add results to current Simulink Project**.
- 4 Run your analysis.

Your results are saved to the Simulink project you opened in step 1.

Specify Signal Ranges

If you constrain signals in your Simulink model to lie within specified ranges, Polyspace software automatically applies these constraints during verification of the generated code. This can improve the precision of your results.

You can specify a range for a model signal by:

- Applying constraints through source block parameters. See “Specify Signal Range through Source Block Parameters” on page 9-18.
- Constraining signals through the base workspace. See “Specify Signal Range through Base Workspace” on page 9-20.

Note: You can also manually define data ranges using the DRS feature in the Polyspace verification environment. If you manually define a DRS file, the software automatically appends any signal range information from your model to the DRS file. However, manually defined DRS information overrides information generated from the model for all variables.

Specify Signal Range through Source Block Parameters

You can specify a signal range by applying constraints to source block parameters.

Specifying a range through source block parameters is often easier than creating signal objects in the base workspace, but must be repeated for each source block. For information on using the base workspace, see “Specify Signal Range through Base Workspace” on page 9-20.

To specify a signal range using source block parameters:

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

Inport

Provide an input port for a subsystem or model.
For Triggered Subsystems, 'Latch input by delaying outside signal' produces the value of the subsystem input at the previous time step.
For Function-Call Subsystems, turning 'On' the 'Latch input for feedback signals of function-call subsystem outputs' prevents the input value to this subsystem from changing during its execution.
The other parameters can be used to explicitly specify the input signal attributes.

Main | **Signal Attributes**

Output function call

Minimum: Maximum:

Data type:

Lock output data type setting against changes by the fixed-point tools

Port dimensions (-1 for inherited):

Variable-size signal:

Sample time (-1 for inherited):

Signal type:

Sampling mode:

- 5 Click **OK**.

Specify Signal Range through Base Workspace

You can specify a signal range by creating signal objects in the MATLAB workspace. This information is used to initialize each global variable to the range of valid values, as defined by the min-max information in the workspace.

Note: You can also specify a signal range by applying constraints to individual source block parameters. This method can be easier than creating signal objects in the base workspace, but must be repeated for each source block. For more information, see “Specify Signal Range through Source Block Parameters” on page 9-18.

To specify an input signal range through the base workspace:

- 1** Configure the signal to use, for example, the `ExportedGlobal` storage class:
 - a** Right-click the signal. From the context menu, select **Properties**. The Signal Properties dialog box opens.
 - b** In the **Signal name** field, enter a name, for example, `my_entry1`.
 - c** Select the **Code Generation** tab.
 - d** In the **Package** drop-down list, select `Simulink`.
 - e** In the **Storage class** drop-down list, select `ExportedGlobal`.

g Click **Apply**.

Run Polyspace on Generated Code

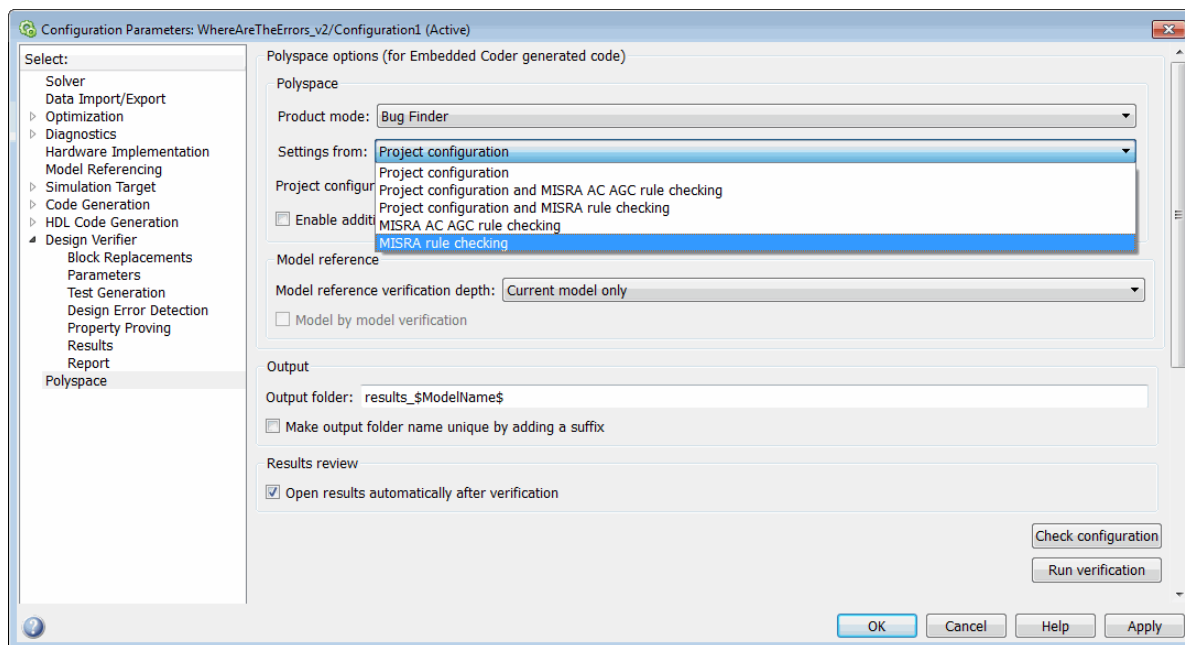
- “Specify Type of Analysis to Perform” on page 10-2
- “Run Analysis for Embedded Coder” on page 10-5
- “Run Analysis for TargetLink” on page 10-6
- “Monitor Progress” on page 10-7

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.



- 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.
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 Job Monitor.

The software writes status messages to a log file in the results folder.

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.

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 Job Monitor

Monitor Progress

In this section...
“Local Analyses” on page 10-7
“Remote Batch Analyses” on page 10-7

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.

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 Job Monitor.

From Simulink, select **Code > Polyspace > Open Job Monitor**

Check Coding Rules from Eclipse

- “Activate Coding Rules Checker” on page 11-2
- “Select Specific MISRA or JSF Coding Rules” on page 11-6
- “Create Custom Coding Rules File” on page 11-9
- “Contents of Custom Coding Rules File” on page 11-11
- “Exclude Files From Analysis” on page 11-12
- “Allow Custom Pragma Directives” on page 11-13
- “Specify Boolean Types” on page 11-14
- “Find Coding Rule Violations” on page 11-15
- “Review Coding Rule Violations” on page 11-16
- “Filter and Group Coding Rule Violations” on page 11-18

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 On the **Configuration** pane, select **Coding Rules & Code Metrics**.
- 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
- MISRA C:2012

If you have generated code, use the **Use generated code requirements** option to use the MISRA C:2012 categories for generated code.

- Custom coding rules

For C++ code, you can check compliance with:

- MISRA C++: 2008
- 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.

Checking for certain rules can cause the analysis to run longer than usual. For more information, see “Rules to Disable for Faster Analysis” on page 3-21.

MISRA C:2004

Option	Description
required-rules	All required MISRA C:2004 coding rules.
all-rules	All MISRA C:2004 coding rules (required and advisory).

Option	Description
SQO-subset1	A small subset of MISRA C:2004 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:2004 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.
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:2012

Option	Description
mandatory	All mandatory MISRA C:2012 coding rules. If you have generated code, also use the Use generated code requirements option categorization for generated code.

Option	Description
mandatory-required	All mandatory and required MISRA C:2012 coding rules. If you have generated code, also use the Use generated code requirements option categorization for generated code.
all	All MISRA C:2012 coding rules (mandatory, required, and advisory).
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:2012 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.

Option	Description
shall-will-rules	All Shall and Will rules. Will rules are intended to be mandatory requirements. However, these rules do not require verification.
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.

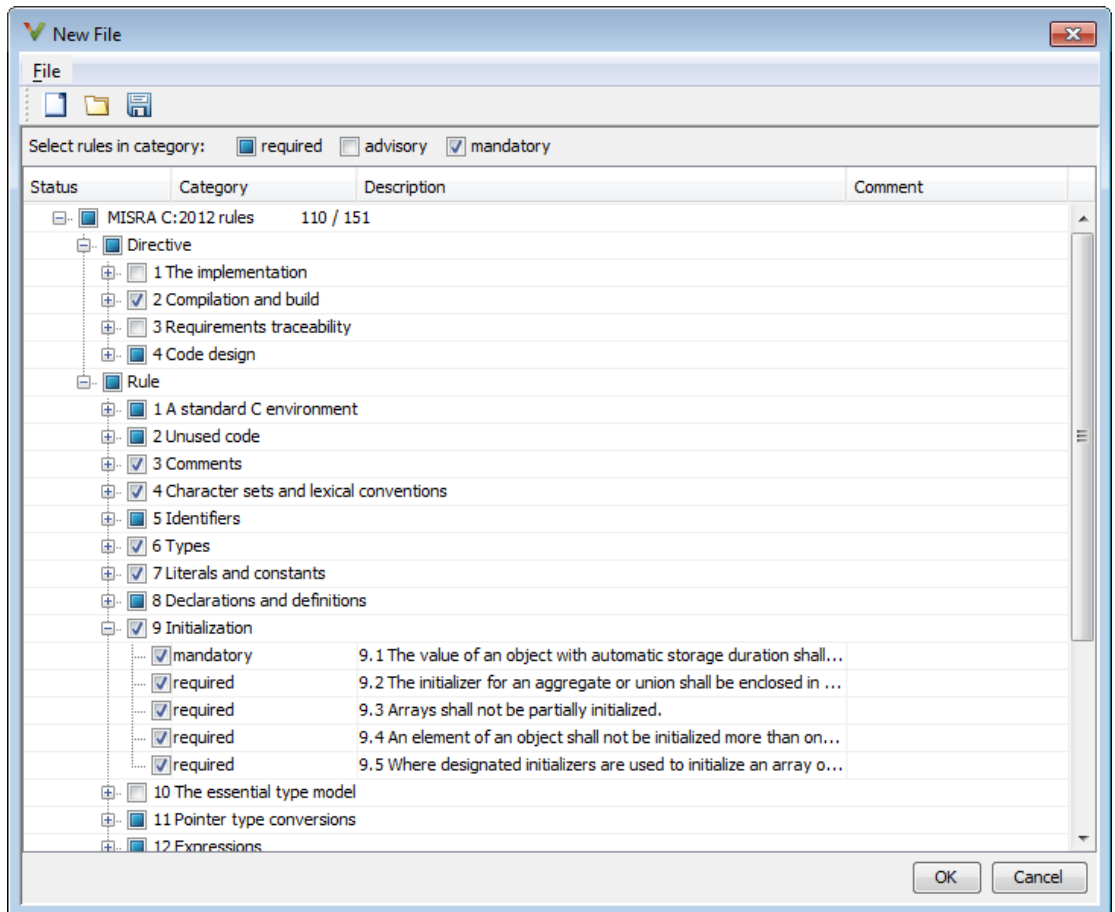
Related Examples


- “Select Specific MISRA or JSF Coding Rules” on page 11-6
- “Create Custom Coding Rules File” on page 11-9

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 the project configuration.
- 2 In the **Configuration** tree view, select **Coding Rules & Code Metrics**.
- 3 Select the check box for the type of coding rules you want 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.



- 6 If you already have a customized rule file you want to edit, reload your customization using the  button.
- 7 Select the rules you want to check.

You can select categories of rules (required, advisory, mandatory), subsets of rules by rule chapter, or individual rules.

- 8 When you are finished, click **OK**.

- 9 For new files, use the Save As dialog box that opens to save your customization as a rules file.
- 10 In the Configuration window, the full path to the rules file appears in the `custom` field. To reuse this customized set of rules for other projects, enter this path name in the dialog box.


Related Examples

- “Activate Coding Rules Checker” on page 11-2
- “Create Custom Coding Rules File” on page 11-9

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.

1 Create Coding Rules File

- 1 Create a Polyspace project. Add `printInitialValue.c` to the project.
- 2 On the **Configuration** pane, select **Coding Rules & Code Metrics**. Select the **Check custom rules** box.
- 3 Click .

The New File window opens, displaying a table of rule groups.

- 4 Clear the **Custom rules** check box to turn off checking of all custom rules.
- 5 Expand the 4 **Structs** node. For the option **4.3 All struct fields must follow the specified pattern**:

Column Title	Action
Status	Select <input checked="" type="checkbox"/> .
Convention	Enter All struct fields must begin with s_ and have capital letters or digits
Pattern	Enter <code>s_[A-Z0-9_]+</code>
Comment	Leave blank. This column is for comments that appear in the coding rules file alone.

2 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 **Result 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 Editor**. The file `printInitialValue.c` opens in the **Code Editor** pane or an external text editor depending on your **Preferences**.
- 3** In the file, replace all instances of `a` with `s_A` and `b` with `s_B`. Rerun the verification.

The custom rule violations no longer appear on the **Results Summary** pane.

Related Examples

- “Activate Coding Rules Checker” on page 11-2
- “Select Specific MISRA or JSF Coding Rules” on page 11-6

More About

- “Contents of Custom Coding Rules File” on page 11-11

Contents of Custom Coding Rules File

In a custom coding rules file, each rule appears in the following format:

```
N.n off|on
convention=violation_message
pattern=regular_expression
```

- *N.n* — Custom rule number, for example, 1.2.
- `off` — Rule is not considered.
- `on` — 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 Coding Rules”.

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 on          # 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 on          # 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 11-9

Exclude Files From Analysis

This example shows how to exclude certain files from coding rules checking and defect checking.



- 1 Open the project configuration.
- 2 In the **Configuration** tree view, select **Inputs & Stubbing**.
- 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) — Excludes header files in the Include folders of your project. For example `.h` or `.hpp` files.
 - **all** — Excludes all include files in the Include folders of your project. 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** — 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

- “Customize Analysis Options” on page 12-3

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 & Code Metrics**.
- 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 11-2

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 is related to checking of the following rules:


- MISRA C:2004 and MISRA AC AGC —12.6, 13.2, 15.4.

For more information, see “MISRA C:2004 and MISRA AC AGC Coding Rules” on page 2-14.

- MISRA C:2012 — 10.1, 10.3, 10.5, 14.4 and 16.7

- 1 Open project configuration.
- 2 In the **Configuration** tree view, select **Coding Rules & Code Metrics**.
- 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 11-2

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 & Code Metrics**. Activate the desired coding rule checker.

For more information, see “Activate Coding Rules Checker” on page 3-2.

- 3 Checking for certain rules can cause the analysis to run longer than usual. Disable those rules if you want.

For more information, see “Rules to Disable for Faster Analysis” on page 3-21.



- 4 Specify that the analysis must not look for defects.
 - In the **Configuration** tree view, select **Bug Finder Analysis**.
 - Clear the **Find defects** check box.
- 5 Click  to run the coding rules checker without checking defects.

Related Examples




- “Activate Coding Rules Checker” on page 11-2
- “Select Specific MISRA or JSF Coding Rules” on page 11-6
- “Review Coding Rule Violations” on page 11-16

Review Coding Rule Violations


This example shows how to review coding rule violations once code analysis is complete. After analysis, the **Results Summary - Bug Finder** tab displays the rule violations with a

-  symbol for predefined coding rules such as MISRA C:2004.
-  symbol for custom coding rules.

In addition, Polyspace Bug Finder highlights defects in your source code in the following ways:

- A  or  mark appears before the line number on the left.
- A  icon appears in the overview ruler to the right of the line containing the rule violation.

To further review a coding rule violation and determine your course of action:

- 1 Select the coding rule violation on the **Results Summary - Bug Finder** tab.
- 2 On the **Result Details** pane, view the location and description of the violated rule. In the source code, the line containing the violation appears highlighted.
- 3 For MISRA C: 2012 rules, on the **Result Details** pane, click the  icon to see the rationale for the rule. In some cases, you can also see code examples illustrating the violation.
- 4 Review the violation in your code.
 - a Determine whether you must fix the code to avoid the violation.
 - b If you choose to retain the code, on the **Result Details** pane, add a comment explaining why you retain the code. This comment helps you or other reviewers avoid reviewing the same coding rule violation twice.

You can also assign a **Severity** and **Status** to the coding rule violation.

- 5 After you have fixed or justified the coding rule violations, run the analysis again.

Related Examples

- “Activate Coding Rules Checker” on page 11-2
- “Find Coding Rule Violations” on page 11-15

- “Filter and Group Coding Rule Violations” on page 11-18

Filter and Group Coding Rule Violations

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.

In this section...

“Filter Coding Rules” on page 11-18

“Group Coding Rules” on page 11-18

“Suppress Certain Rules from Display in One Click” on page 11-18

Filter Coding Rules

- 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**.

Group Coding Rules

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

The rules are grouped by numbers. Each group corresponds to a certain code construct.

- 2 Expand the group nodes to select an individual coding rule violation.

Suppress Certain Rules from Display in One Click

Instead of filtering individual rules from display each time you open your results, you can limit the display of rule violations in one click. To limit the display of rule violations, use the **Show** menu on the **Results Summary** pane. You can create your own options on this menu. You can share the option file to help developers in your organization review violations of at least certain coding rules.

- 1 In the Polyspace user interface, select **Tools > Preferences**.
- 2 On the **Review Scope** tab, do one of the following:
 - To add predefined options to the **Show** menu, select **Include Quality Objectives Scopes**.

The **Scope Name** list shows additional options, **HIS**, **SQ0-4**, **SQ0-5**, and **SQ0-6**. Select an option to see which rules are suppressed from display.

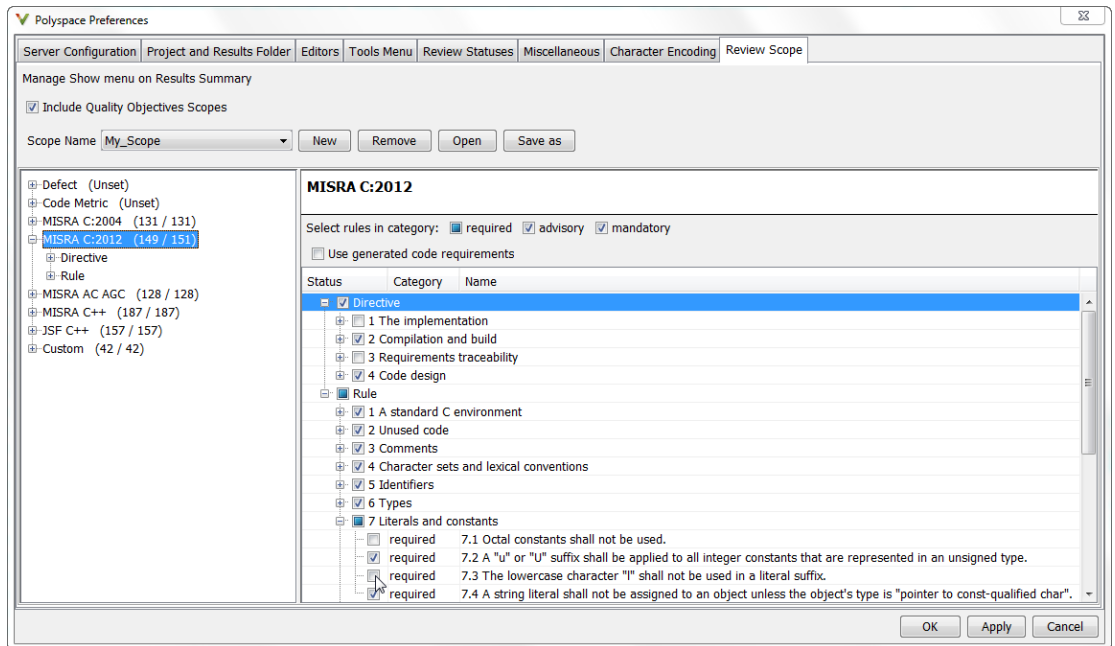
In addition to coding rule violations, the options impose limits on the display of code metrics and defects.

- To create your own option on the **Show** menu, select **New**. Save your option file.

On the left pane, select a rule set such as **MISRA C:2012**. On the right pane, to suppress a rule from display, clear the box next to the rule.

To suppress all rules belonging to a group such as **The essential type model**, clear the box next to the group name. For more information on the groups, see “Coding Rules”. If only a fraction of rules in a group is selected, the check box next to the group name displays a symbol.

To suppress all rules belonging to a category such as **advisory**, clear the box next to the category name on the top of the right pane. If only a fraction of rules in a category is selected, the check box next to the category name displays a symbol.



3 Select **Apply** or **OK**.

On the **Results Summary** pane, the **Show** menu displays the additional options.

4 Select the option that you want. The rules that you suppress do not appear on the **Results Summary** pane.

Related Examples

- “Activate Coding Rules Checker” on page 11-2
- “Review Coding Rule Violations” on page 11-16


Find Bugs from Eclipse

- “Run Analysis” on page 12-2
- “Customize Analysis Options” on page 12-3




Run Analysis



- 1 Switch to the Polyspace perspective.
 - a Select **Window > Open Perspective > Other**.
 - b In the Open Perspective dialog box, select **Polyspace**.


This allows you to view only the information related to a Polyspace verification.

- 2 If you previously ran a Polyspace Code Prover verification, open the **Polyspace Run - Code Prover** view. In the dropdown list beside the , select **Bug Finder**.
- 3 To start an analysis, do one of the following:
 - In the **Project Explorer**, right-click the project containing the files that you want to verify and select **Run Polyspace Bug Finder**.
 - In the **Project Explorer**, select the project containing the files that you want to verify. From the global menu, select **Polyspace > Run**.

You can follow the progress of the analysis in the **Polyspace Run - Bug Finder** view. If you see an error or warning during the compilation phase, double-click it to go to the corresponding location in the source code. Once the analysis is over, the results are displayed in the **Results Summary - Bug Finder** view.

- 4 If results are available, the  icon in the **Polyspace Run - Bug Finder** view turns to . Click the  icon to load available results.

With your results open, if additional results are available, the  icon is still visible. Click the  icon to load all available results.

- 5 To stop an analysis, select **Polyspace > Stop**. Alternatively you can use the  button in the **Polyspace Run - Bug Finder** view.

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 global menu, select **Polyspace > Configure Project**.

The Polyspace Bug Finder Configuration window appears.

- 2 Select the different nodes to change your analysis configuration.

For example:

- a Select the **Coding Rules** node.
- b Select **Check MISRA C:2004** to check for violations of MISRA C:2004 coding rules.

For information about the different analysis options, see “Analysis Options for C” or “Analysis Options for C++”.

View Results in Eclipse

- “View Results” on page 13-2
- “Review and Fix Results” on page 13-3
- “Filter and Group Results” on page 13-5
- “Understanding the Results Views” on page 13-8

View Results

This example shows how to view Polyspace Bug Finder results. After you run an analysis, you can view the results either in Eclipse or from the Polyspace Bug Finder interface.

In this section...
“View Results in Eclipse” on page 13-2
“View Results in Polyspace Environment” on page 13-2

View Results in Eclipse

After you run an analysis in Eclipse, your results automatically appear on the **Results Summary - Bug Finder** tab.

- If you closed the **Results Summary - Bug Finder** tab, select **Polyspace > Show View > Show Results Summary view** to reopen the tab.
- If you need to reload the results, select **Polyspace > Reload results**.

This option is useful when you reopen Eclipse or when you are switching between Polyspace projects.

View Results in Polyspace Environment

To view your results in the Polyspace Bug Finder interface, select **Polyspace > Open Results in PVE**.

Note: You can view defects, coding rule violations and code metrics from the Eclipse environment. However, you can impose limits on metrics only from the Polyspace environment. For more information, see “Review Code Metrics” on page 5-30.



Related Examples

- “Run Analysis” on page 12-2

Review and Fix Results

This example shows how to review and comment results obtained from a Polyspace Bug Finder analysis. When reviewing results, you can assign a status and severity 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 project, the review status, severity and comments from the previous analysis will be automatically imported into the next.


After analysis, the results appear on the **Results Summary - Bug Finder** tab. In addition, Polyspace Bug Finder highlights defects in your source code in the following ways:

- An ! mark appears before the line number on the left.
- The operation containing the defect has a wavy red underlining.
- A  icon appears in the overview ruler to the right of the line containing the defect.
- A  icon appears at the top of the overview ruler. If you place your cursor on the icon, a tooltip states the total number of defects in the file.

To further review a defect and determine your course of action:

- 1 On the **Results Summary - Bug Finder** tab, select the defect that you want to review.

The **Result Details** pane displays information about the current defect.

- 2 On the **Result Details** pane, click the  icon to see a brief description and code examples for the defect. In some cases, you can also see risks associated with not fixing the defect and a suggested fix.
- 3 Investigate the result further. Determine whether to fix your code, review the result later, or retain the code but provide some explanation.
- 4 On the **Result Details** pane, provide the following review information for the result:
 - **Severity** to describe how critical you consider the issue.
 - **Status** to describe how you intend to address the issue.

You can also create your own status or associate justification with an existing status from the Polyspace user interface. Select **Tools > Preferences** and create or modify statuses on the **Review Statuses** tab.

- **Comment** to describe any other information about the result.
- 5 To provide review information for several results together, select the results. Then, provide review information for a single result.

To select the results in a group:

- If the results are contiguous, left-click the first result. Then **Shift**-left click the last result.

To group certain results together, use the column headers on the **Results Summary - Bug Finder** tab.

- If the results are not contiguous, **Ctrl**-left click each result.
- If the results belong to the same group and have the same color, right-click one result. From the context menu, select **Select All Type Results**.

For instance, select **Select All "Memory leak" Results**.

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

Related Examples

- “View Results” on page 13-2
- “Filter and Group Results” on page 13-5

Filter and Group Results

This example shows how to filter and group defects on the **Results Summary - Bug Finder** tab. To organize your review of results, use filters and groups when you want to:

- Review only high-impact defects.

For more information on impact, see “Classification of Defects by Impact” on page 5-12.

- Review certain types of defects in preference to others.

For instance, you first want to address the defects resulting from **Missing or invalid return statement**.

- Review only new results found since the last analysis.
- 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.

In this section...
“Filter Results” on page 13-5
“Group Results” on page 13-6

Filter Results

- 1 To review only new results found since the last verification, on the **Results Summary - Bug Finder** tab, select **New results**.
- 2 To suppress code metrics from your results, on the **Results Summary - Bug Finder** tab, select **Show > Defects & Rules**.

You can increase the options on the **Show** menu or create your own options from the Polyspace user interface. For examples, see:

- “Suppress Certain Rules from Display in One Click” on page 3-18
- “Limit Display of Defects” on page 5-20
- “Review Code Metrics” on page 5-30

3

For all other filters, click the  icon on the appropriate column.

Item to Filter	Column
Results in a certain file or function	File or Function
Results with a certain severity or status	Severity or Status
Results in a certain group such as numerical or data flow	Group
Results with a certain impact	Information
Results that correspond to certain CWE IDs.	CWE ID For more information, see “Find CWE Identifiers from Defects” on page 5-61.

4 Clear **All**. Select the boxes for the results that you want displayed.

Alternatively, clear the boxes for the results that you do not want displayed.

Note: You can also apply multiple filters.

Group Results

On the **Results Summary - Bug Finder** tab:

- To show results without grouping, select **Group by > None**.
- To show results grouped by result type, select **Group by > Family**.
 - The defects are organized by the defect groups. For more information on the groups, see “Defects”.
 - The coding rule violations are grouped by type of coding rule. For more information, see “Coding Rules”.

- The code metrics are grouped by scope of metric. For more information, see “Code Metrics”.
- To show results grouped by file, select **Group by > File**.

Within each file, the results are grouped by function. The results that are not associated with a particular function are grouped under **File Scope**.

- For C++ code, to show results grouped by class, select **Group by > Class**. The results that are not associated with a particular class are grouped under **Global Scope**.

Within each class, the results are grouped by method.

Related Examples

- “View Results” on page 13-2
- “Review and Fix Results” on page 13-3

Understanding the Results Views

In this section...

“Results Summary” on page 13-8

“Result Details” on page 13-10

Results Summary

The **Results Summary - Bug Finder** tab lists the defects and coding rule violations along with their attributes. To organize your results review, from the **Group by** list on this tab, select one of the following options:

- **None:** Lists defects and coding rule violations without grouping. By default the results are listed in order of severity.
- **Family:** Lists results grouped by defect group. For more information on the defect groups, see “Bug Finder Defect Groups” on page 5-52.
- **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.

- **File:** 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	Grouping to which the defect belongs. For example, if you choose the Checks by File/Function grouping, 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 - Bug Finder tab, the defects appear sorted by this number.
Type	Defect or coding rule violation.

Attribute	Description
Group	Category of the defect. For more information on the defect groups, see “Bug Finder Defect Groups” on page 5-52.
Check	Description of the defect
CWE ID	CWE ID-s that correspond to the defect. For more information, see “Mapping Between CWE Identifiers and Defects” on page 5-63.
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, <code>Global Scope</code> .
Function	Function containing the instruction where the defect occurs. If the function is a method of a class, it appears in the format <code>class_name::function_name</code> .
Severity	Level of severity you have assigned to the defect. The possible levels are: <ul style="list-style-type: none"> • High • Medium • Low • Not a defect
Status	Review status you have assigned to the check. The possible statuses are: <ul style="list-style-type: none"> • Fix • Improve • Investigate • Justified • 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 13-3.
- Organize your check review using filters on the columns. For more information, see “Filter and Group Results” on page 13-5.

Result Details

The **Result Details** pane contains detailed information about a specific defect. Select a defect on the **Results Summary - Bug Finder** tab to reveal further information about the defect on the **Result Details** pane.

- 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.
- The  button allows you to access documentation for the defect.

Check Coding Rules from Microsoft Visual Studio

- “Activate C++ Coding Rules Checker” on page 14-2
- “Exclude Files From Analysis” on page 14-4

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 & Code Metrics**.
- 4 Under **Coding Rules & Code Metrics**, 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 <i>required</i> MISRA C++ coding rules. Violations are reported as warnings.
all-rules	All <i>required</i> and <i>advisory</i> 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 2-61.
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 2-61.

Option	Explanation
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 3-6.

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 3-6.

- 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 3-9 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.

Exclude Files From Analysis

This example shows how to exclude files from coding rules checking and defect checking. Excluding header files, include files, or files you are not working on allows you focus on defects in your purview.

- 1 Open the project configuration.
- 2 In the **Configuration** tree view, select **Inputs & Stubbing**.
- 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) — Excludes header files in the Include folders of your project. For example `.h` or `.hpp` files.
 - **all** — Excludes all include files in the Include folders of your project. 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** — 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

- “Customize Polyspace Options” on page 15-8

Find Bugs from Microsoft Visual Studio

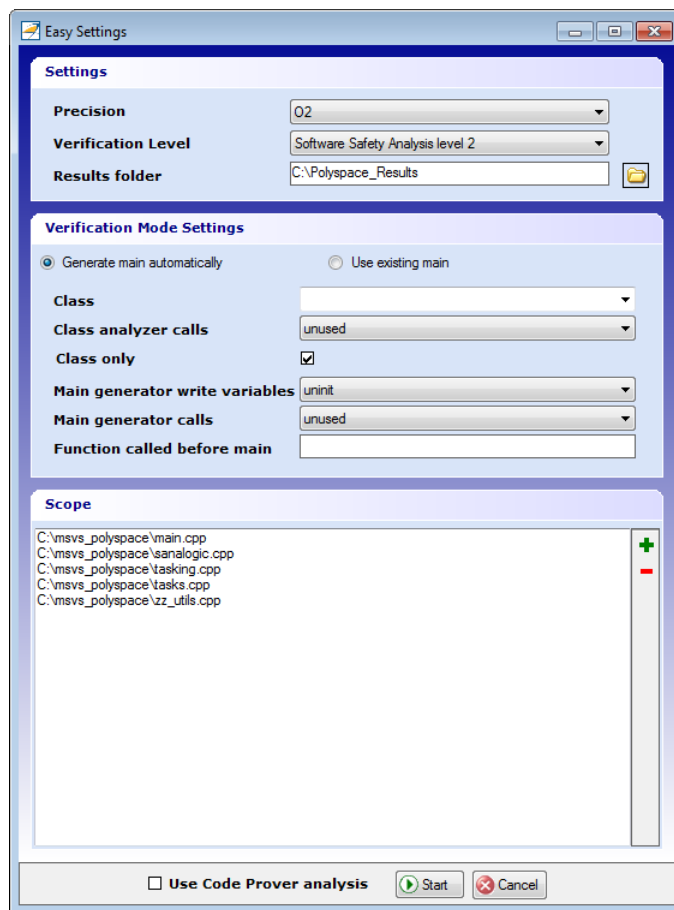
- “Run Polyspace in Visual Studio” on page 15-2
- “Monitor Progress in Visual Studio” on page 15-5
- “Customize Polyspace Options” on page 15-8
- “Configuration File and Default Options” on page 15-9
- “Bug Finding in Visual Studio” on page 15-10

Run Polyspace in Visual Studio


To set up and start an analysis:

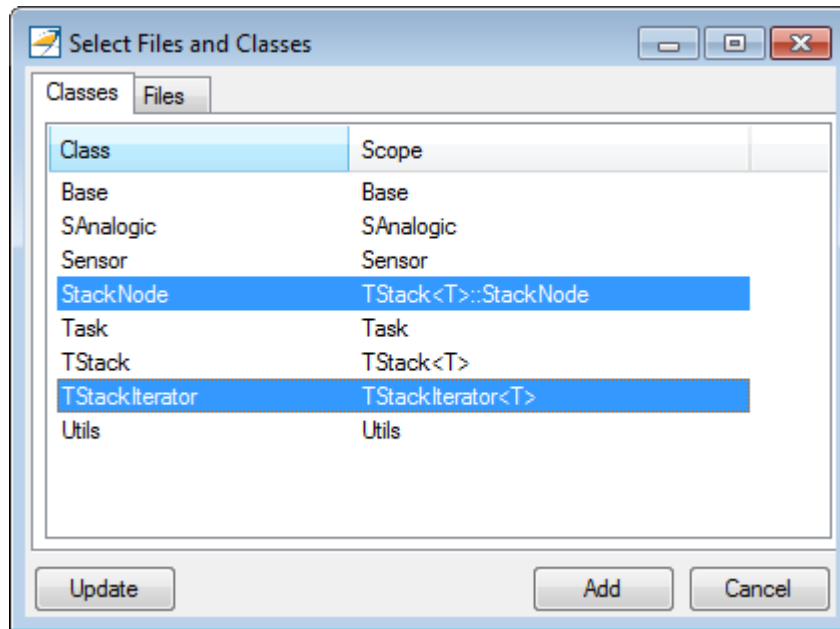
- 1 In the **Solution Explorer** view, select one or more files that you want to analyze.
- 2 Right-click the selection, and select **Polyspace Verification**.

The Easy Settings dialog box opens.



- 3 In the Easy Settings dialog box, you can specify the following options for your analysis:

- Under **Settings**, configure the following:
 - **Precision** — Precision of analysis
 - **Passes** — Level of analysis
 - **Results folder** — Location where software stores analysis results
- Under **Verification Mode Settings**, configure the following:
 - **Generate main** — Polyspace generates a `main` or **Use existing** — Polyspace uses an existing `main`
 - **Class** — Name of class to analyze
 - **Class analyzer calls** — Functions called by generated `main`
 - **Class only** — Analysis of class contents only
 - **Main generator write** — Type of initialization for global variables
 - **Main generator calls** — Functions (not in a class) called by generated `main`
 - **Function called before main** — Function called before the generated `main`
- Under **Scope**, you can modify the list of files and C++ classes to analyze.
 - ▣ Select . The Select Files and Classes dialog box opens.



- b** Select the classes that you want to analyze, then click **Add**.

In the Configuration pane in the Polyspace environment, you can configure advanced options not in the Easy Settings dialog box. See “Customize Polyspace Options” on page 15-8.

- 4** Make sure the **Use Code Prover analysis** check box is cleared.
- 5** Click **Start** to start the analysis.

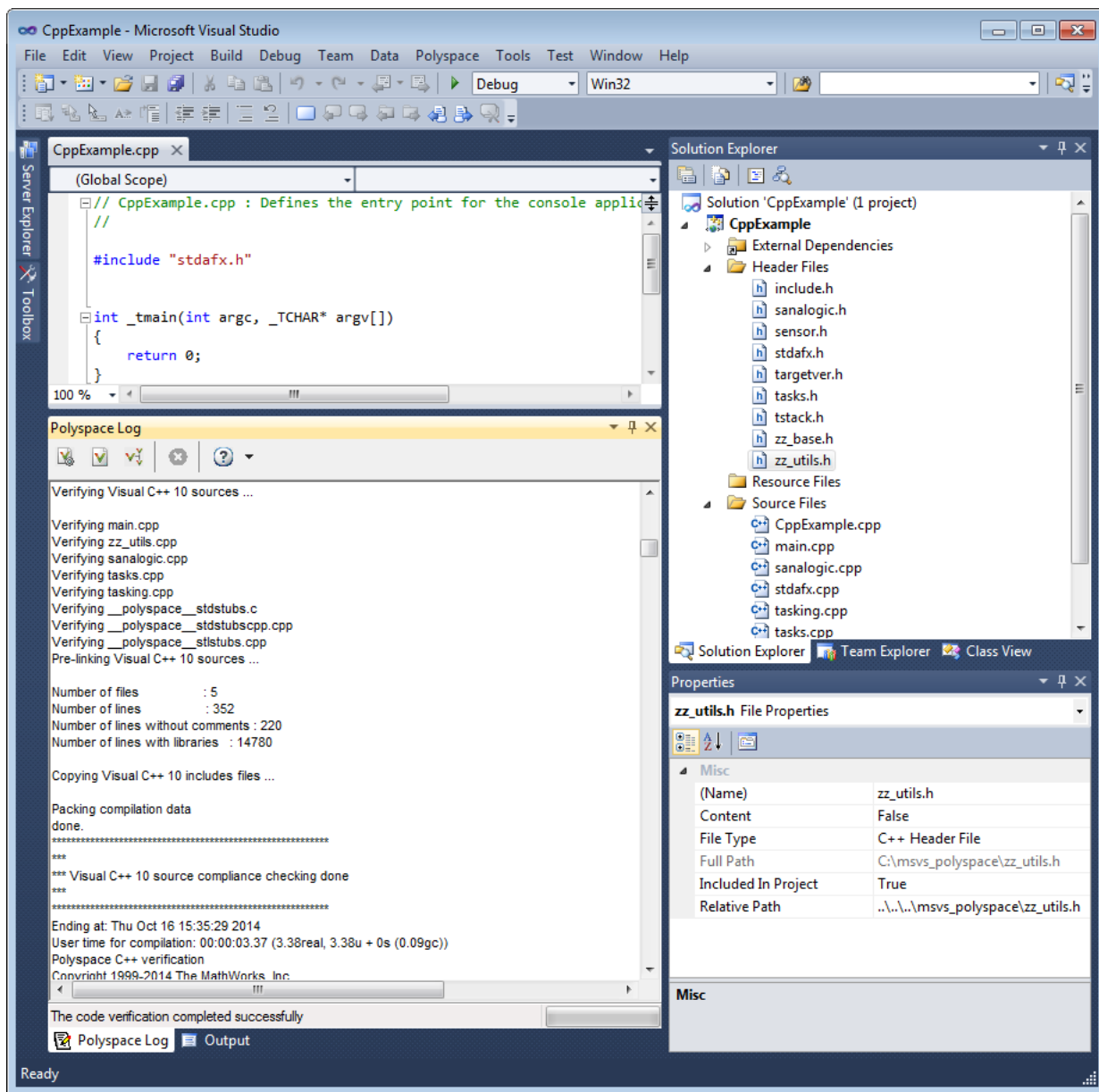
To follow the progress of an analysis, see “Monitor Progress in Visual Studio” on page 15-5

Monitor Progress in Visual Studio

Local Analysis

- 1 Open the **Polyspace Log** view to follow the progress of your analysis.

If Polyspace finds compilation issues, the errors are highlighted as links. Click a link to display the file and line that produced the error.



- 2 To stop an analysis, on the **Polyspace Log** toolbar, click **X**.

Remote Analysis

- 1 Open the **Polyspace Log** view to follow the progress of your analysis.

If Polyspace finds compilation issues, the errors are highlighted as links. Click a link to display the file and line that produced the error.

To stop a verification during the compilation phase, on the **Polyspace Log** toolbar, click **X**.

After compilation, Polyspace sends your analysis to the remote server.

- 2 Select **Polyspace > Job Monitor**.
- 3 In the Polyspace Job Monitor, right-click your project and select **View Log File**

To stop a remote analysis after compilation, use the Job Monitor interface.

Related Examples

- “Run Polyspace in Visual Studio” on page 15-2
- “Open Results in Polyspace Environment” on page 16-2

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.

Configuration File and Default Options

Some options are set by default while others are extracted from the Visual Studio project and stored in the associated Polyspace configuration file.

- The following table shows Visual Studio options that are extracted automatically, and their corresponding Polyspace options:

Visual Studio Option	Polyspace Option
/D <name>	-D <name>
/U <name>	-U <name>
/MT	-D_MT
/MTd	-D_MT -D_DEBUG
/MD	-D_MT -D_DLL
/MDd	-D_MT -D_DLL -D_DEBUG
/MLd	-D_DEBUG
/Zc:wchar_t	-wchar-t-is keyword
/Zc:forScope	-for-loop-index-scope in
/FX	-support-FX-option-results
/Zp[1,2,4,8,16]	-pack-alignment-value [1,2,4,8,16]

- Source and include folders (-I) are also extracted automatically from the Visual Studio project.
- Default options passed to the kernel depend on the Visual Studio release: -dialect Visual7.1 (or -dialect visual8) -OS-target Visual -target i386

Bug Finding in Visual Studio

You can apply the bug finding functionality of Polyspace software to code that you develop within the Visual Studio Integrated Development Environment (IDE).

A typical workflow is:

- 1** Use the Visual Studio editor to create a project and develop code within this project.
- 2** Set up the Polyspace analysis by configuring analysis options and settings, and then start the analysis.
- 3** Monitor the analysis.
- 4** Open the verification results and review in the Polyspace environment.


Before you can verify code in Visual Studio, you must install the Polyspace add-in for Visual Studio. For more information, see “Install Polyspace Add-In for Visual Studio”.

Open Results from Microsoft Visual Studio

Open Results in Polyspace Environment

After your analysis finishes running in Visual Studio, open the Polyspace environment to view your results. If you ran a server analysis, download the results before opening the Polyspace environment.

To view your results:

- From the Polyspace Log window, select .
- Select **Polyspace > Polyspace**.

Then, open your results from the Polyspace interface. For instructions, see “Open Results” on page 5-2.

Related Examples

- “Review and Fix Results” on page 5-24
- “Run Polyspace in Visual Studio” on page 15-2

Troubleshooting in Polyspace Bug Finder

- “View Error Information When Verification Stops” on page 17-2
- “Troubleshoot Compilation and Linking Errors” on page 17-4
- “Contact Technical Support” on page 17-5
- “Header File Location Not Specified” on page 17-7
- “Polyspace Cannot Find the Server” on page 17-8
- “Insufficient Memory During Report Generation” on page 17-9
- “Errors From Disk Defragmentation and Antivirus Software” on page 17-10
- “Syntax Errors Due to Unknown Keywords” on page 17-11
- “Undeclared Identifier” on page 17-12
- “Missing Identifiers with Keil or IAR Dialect” on page 17-13
- “Unknown Prototype” on page 17-14
- “Cannot Find Include File” on page 17-16
- “#error Directive” on page 17-17
- “Object is Too Large” on page 17-18
- “Errors From Special Characters” on page 17-21
- “Initialization of Static Class Members (C++)” on page 17-22
- “Double Declarations of Standard Template Library Functions” on page 17-23
- “GNU Dialect” on page 17-24
- “ISO versus Default Dialects” on page 17-27
- “Visual Dialects” on page 17-29
- “Eclipse Java Version Incompatible with Polyspace Plug-in” on page 17-31

View Error Information When Verification Stops

If verification stops, you can view error information in the user interface or in the log file.

In this section...
“View Error Information in User Interface” on page 17-2
“View Error Information in Log File” on page 17-2

View Error Information in User Interface

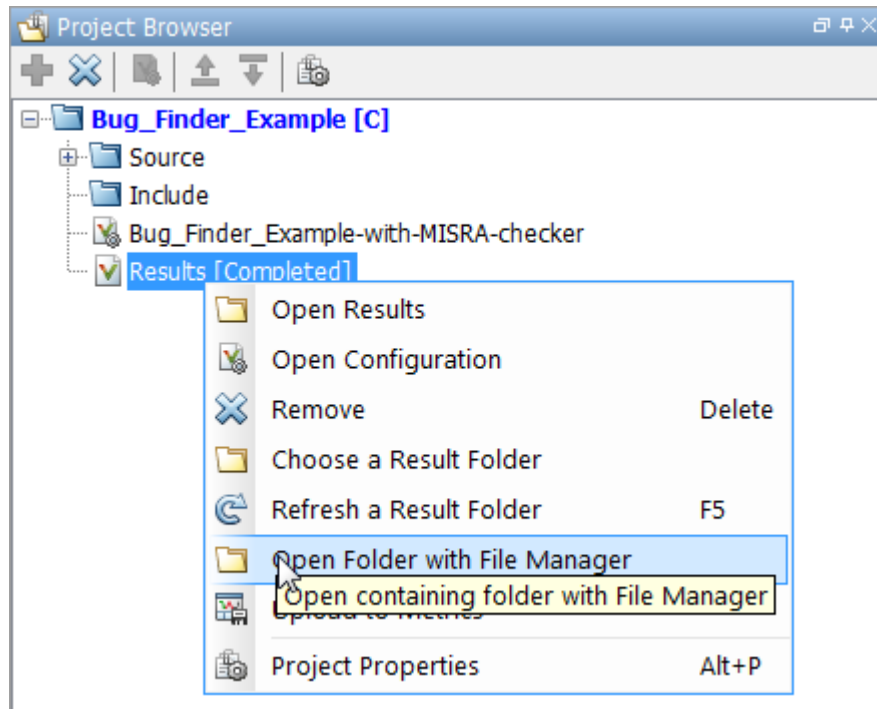
- 1 View the errors on the **Output Summary** tab.
- 2 To open the source code at the line containing the error, double-click the message.
- 3 To search the log, on the **Search** pane, enter your search term. From the drop down list on this pane, select **Output Summary** or **Run Log**.

If the **Search** pane is not open by default, select **Windows > Show/Hide View > Search**.

View Error Information in Log File

You can view errors directly in the log file. The log file is in your results folder. To open the log file:

- 1 Right-click the result folder name on the **Project Browser** pane. From the context menu, select **Open Folder with File Manager**.



- 2 Open the log file, `Polyspace_R20##n_ProjectName_date-time.log`
- 3 To view the errors, scroll through the verification log file, starting at the end and working backward.

The following example shows sample log file information. The error has occurred because a variable `var` used in the code is not defined earlier.

```
C:\missing_include.c, line 4: error: identifier "var" is undefined
|   var = func();
|   ^
```

```
1 error detected in the compilation of "missing_include.c".
C:\missing_include.c: warning: Failed compilation.
Global compilation phase...
```

Troubleshoot Compilation and Linking Errors

When you obtain an error message related to compilation or linking, try checking whether the error message is related to the target operating system or the dialect that you specified.

- Sometimes, certain macros in your code are defined only for a specific operating system. You use the macros to activate code specific to that operating system. Unless you specify your target operating system, Polyspace does not consider that those macros have been defined.

For more information on how to specify the target operating system, see “Target operating system (C/C++)”.

- Sometimes, your compilers can allow specific language extensions. Unless you specify your dialect, Polyspace produces compilation errors for those extensions.

For more information on how to specify dialect, see “Dialect (C)” or “Dialect (C++)”.

Contact Technical Support

In this section...

“Provide System Information” on page 17-5

“Provide Information About the Issue” on page 17-5

Provide System Information

When you enter a support request, provide the following system information:

- Hardware configuration
- Operating system
- Polyspace and MATLAB license numbers
- Specific version numbers for Polyspace products
- Installed Bug Report patches

To obtain your configuration information, do one of the following:

- In the Polyspace user interface, select **Help > About**.
- At the command line, run the following command:
 - UNIX — `matlabroot/polyspace/bin/polyspace-code-prover-nodesktop -ver`
 - DOS — `MATLAB_Install\polyspace\bin\polyspace-code-prover-nodesktop -ver`

Provide Information About the Issue

If you face compilation issues with your project, see “Troubleshooting in Polyspace Code Prover”. If you are still having issues, contact technical support with the following information:

- The verification log.

The verification log is a text file generated in your results folder and titled `Polyspace_version_project_date_time.txt`. It contains the error message, the options used for the verification and other relevant information.

- The source files related to the compilation error, if possible.

If you cannot provide the source files:

- Try to provide a screenshot of the source code section that causes the compilation issue.
- Try to reproduce the issue with a different code. Provide that code to technical support.

If you are having trouble understanding a result, see “Polyspace Bug Finder Results”. If you are still having trouble understanding the result, contact technical support with the following information:

- The verification log.

The verification log is a text file generated in your results folder and titled `Polyspace_version_project_date_time.txt`. It contains the the options used for the verification and other relevant information.

- The source files related to the result if possible.

If you cannot provide the source files:

- Try provide a screenshot of the relevant source code from the **Source** pane on the Polyspace user interface.
- Try to reproduce the problem with a different code. Provide that code to technical support.

Header File Location Not Specified

Message

Could not find include file "myHeader.h"

Possible Cause

Your code `#includes` a header file, for instance, `myHeader.h`. However, the include folders that you specify do not contain the header file.

Solution

Do one of the following:

- Add the missing header file to the specified include folder.
- Specify another include folder that contains the missing file.

For more information, see “Add Source Files and Include Folders” on page 1-29.

Polyspace Cannot Find the Server

Message

```
Error: Cannot instantiate Polyspace cluster
| Check the -scheduler option validity or your default cluster profile
| Could not contact an MJS lookup service using the host computer_name.
  The hostname, computer_name, could not be resolved.
```

Possible Cause

Polyspace uses information provided in **Preferences** to locate the server. If this information is incorrect, the software cannot locate the server.

Solution

Provide correct server information.

- 1 Select **Tools > Preferences**.
- 2 Select the **Server Configuration** tab. Provide your server information.

For more information, see “Set Up Server for Metrics and Remote Analysis”.

Insufficient Memory During Report Generation

Message

```
....  
Exporting views...  
Initializing...  
Polyspace Report Generator  
Generating Report  
.....  
    Converting report  
Opening log file: C:\Users\ouser\AppData\Local\Temp\java.log.7512  
Document conversion failed  
.....  
Java exception occurred:  
java.lang.OutOfMemoryError: Java heap space
```

Possible Cause

During generation of very large reports, the software can sometimes indicate that there is insufficient memory.

Solution

If this error occurs, try increasing the Java[®] heap size. The default heap size in a 64-bit architecture is 1024 MB.

To increase the size:

- 1 Navigate to `matlabroot\polyspace\bin\architecture`. Where:
 - `matlab` is the installation folder.
 - `architecture` is your computer architecture, for instance, `win32`, `win64`, etc.
- 2 Change the default heap size that is specified in the file, `java.opts`. For example, to increase the heap size to 2 GB, replace `1024m` with `2048m`.
- 3 If you do not have write permission for the file, copy the file to another location. After you have made your changes, copy the file back to `matlabroot\polyspace\bin\architecture\`.

Errors From Disk Defragmentation and Antivirus Software

Message

```
Some stats on aliases use:
  Number of alias writes:      22968
  Number of must-alias writes: 3090
  Number of alias reads:      0
  Number of invisibles:      949
Stats about alias writes:
  biggest sets of alias writes: foo1:a (733), foo2:x (728), foo1:b (728)
  procedures that write the biggest sets of aliases: foo1 (2679), foo2 (2266),
                                                    foo3 (1288)
**** C to intermediate language translation - 17 (P_PT) took 44real, 44u + 0s (1.4gc)
exception SysErr(OS.SysErr(name="Directory not empty", syserror=notempty)) raised.
unhandled exception: SysErr: No such file or directory [noent]
```

```
-----
---
--- Verifier has encountered an internal error.      ---
--- Please contact your technical support.          ---
---
-----
```

Possible Cause

A disk defragmentation tool or antivirus software is running on your machine.

Solution

Try:

- Stopping the disk defragmentation tool.
- Deactivating the antivirus software. Or, configuring exception rules for the antivirus software to allow Polyspace to run without a failure.

Note: Even if the analysis does not fail, the antivirus software can reduce the speed of your analysis. This reduction occurs because the software checks the temporary analysis files. Configure the antivirus software to exclude your temporary folder, for example, **C:\Temp**, from the checking process.

Syntax Errors Due to Unknown Keywords

Message

Verifying compilation.c compilation.c:3: syntax error; found `x'
expecting `;`

Code Used

```
void main(void)
{
    int far x;
    x = 0;
    x++;
}
```

Cause

The `far` keyword is unknown in ANSI C. Therefore, Polyspace does not recognize whether `far` is a variable or a qualifier.

Solution

Possible solutions include:

- Remove `far` from the source code, or replace `far` with a qualifier, such as `const` or `volatile`.
- Remove or replace `far` in the preprocessed code, only for the analysis. This solution keeps your source code intact.
- Replace each individual unknown keyword using an analysis option.

For information on the analysis option, see “Preprocessor definitions (C/C++)”.

- Redefine all unknown keywords in a separate header file using `#define` directives. Specify that header file using an analysis option.

For information on the analysis option, see “Include (C/C++)”.

Undeclared Identifier

Message

Verifying compilation.c compilation.c:3: undeclared identifier `x`

Code Used

```
void main(void) { x = 0; x++; }
```

Cause

Polyspace cannot find the variable declaration. Therefore, it cannot identify the variable type.

Possible causes include:

- The source code you provided does not contain the variable declaration.
- The variable represents a keyword that your compiler recognizes but is not part of the ANSI C standard. Therefore, Polyspace does not recognize it.

For instance, some compilers interpret `__SP` as a reference to the stack pointer.

Solution

Possible solutions include:

- Provide the variable declaration if it is missing in your source code.
- If the variable represents a keyword that Polyspace does not recognize, replace or remove the keyword from your source code or preprocessed code. For more information, see “Syntax Errors Due to Unknown Keywords” on page 17-11.

Missing Identifiers with Keil or IAR Dialect

Message

expected an identifier

Possible Cause

If you select Keil or IAR as your dialect, the software removes certain keywords during preprocessing. If you use these keywords as identifiers such as variable names, a compilation error occurs.

Solution

Specify that Polyspace must not remove the keywords during preprocessing. Enter `__PST_KEIL_NO_KEYWORDS__` or `__PST_IAR_NO_KEYWORDS__` for preprocessor definitions.

For more information, see “Preprocessor definitions (C/C++)”.

Unknown Prototype

Message

Error: function 'myfunc' has unknown prototype

Code Used

```
var = myfunc(s32var1, ptr->s32var2, 24);
```

var, s32var1 and s32var2 are signed long variables.

Cause

Your source code does not contain the function prototype.

Solution

Possible solutions are:

- See if your project is missing the include file that contains the function prototype. Add the folder containing the missing file.

For more information, see “Cannot Find Include File” on page 17-16.

- Specify the function prototype in a separate file. `#include` this file in all your source files, only for the purposes of analysis.

- 1 In an include file, for example, `myinclude.h`, specify the complete prototype for the function:

```
#ifndef _INC_H  
#define _INC_H
```

```
extern signed long myfunc(signed long, signed long, signed long);
```

```
#endif
```

- 2 Specify that Polyspace must `#include` the file `myinclude.h` in your source files during analysis.

The file is included only for the purposes of analysis. Your original source files remain intact. For more information on the analysis option, see “Include (C/C++)”.

Cannot Find Include File

Messages

Warning: could not find include file "one_file.h"

Code Used

```
#include "one_file.h"
```

Cause

The include folders that you specify do not contain the header file `one_file.h`.

The missing header file can contain a function prototype. If your source code uses the function, Polyspace Bug Finder determines the function prototype from the function call instance. The prototype that Polyspace Bug Finder determines can potentially be different from what you expect.

Solution

Specify the folder that contains the missing header file `one_file.h`.

- In the user interface, add the folder to your project.

For more information, see “Add Source Files and Include Folders” on page 1-29.

- At the command line, use the flag `-I` with the `polyspace-bug-finder-nodesktop` command.

For more information, see `-I`.

#error Directive

Message

```
#error directive: !Unsupported platform; stopping!
```

Code Used

```
#if defined(__BORLANDC__) || defined(__VISUALC32__)  
# define MYINT int // then use the int type  
#elif defined(__GNUC__) // GCC doesn't support myint  
# define MYINT long // but uses 'long' instead  
#else  
# error !Unsupported platform; stopping!  
#endif
```

Cause

The analysis terminates in the compilation phase. The error log displays a `#error` directive indicating an unsupported platform. The compilation stops because Polyspace does not recognize one of the three compilation flags, `__BORLANDC__`, `__VISUALC32__`, or `__GNUC__`.

Solution

For successful compilation, do one of the following:

- Explicitly define one of the compilation flags `__BORLANDC__`, `__VISUALC32__`, or `__GNUC__`.

For more information, see “Preprocessor definitions (C/C++)”.

- Specify a dialect such as `visual12.0` or `gnu4.9`. Specifying a dialect defines some of the compilation flags for the analysis.

For more information, see:

- C: “Dialect (C)”
- C++: “Dialect (C++)”

Object is Too Large

Issue

The analysis terminates during compilation with a message indicating that an object is too large. The error happens when the software detects an object such as an array, union, structure, or class, that is too big for the pointer size of the selected target.

Message

Limitation: struct or union is too large

Code Used

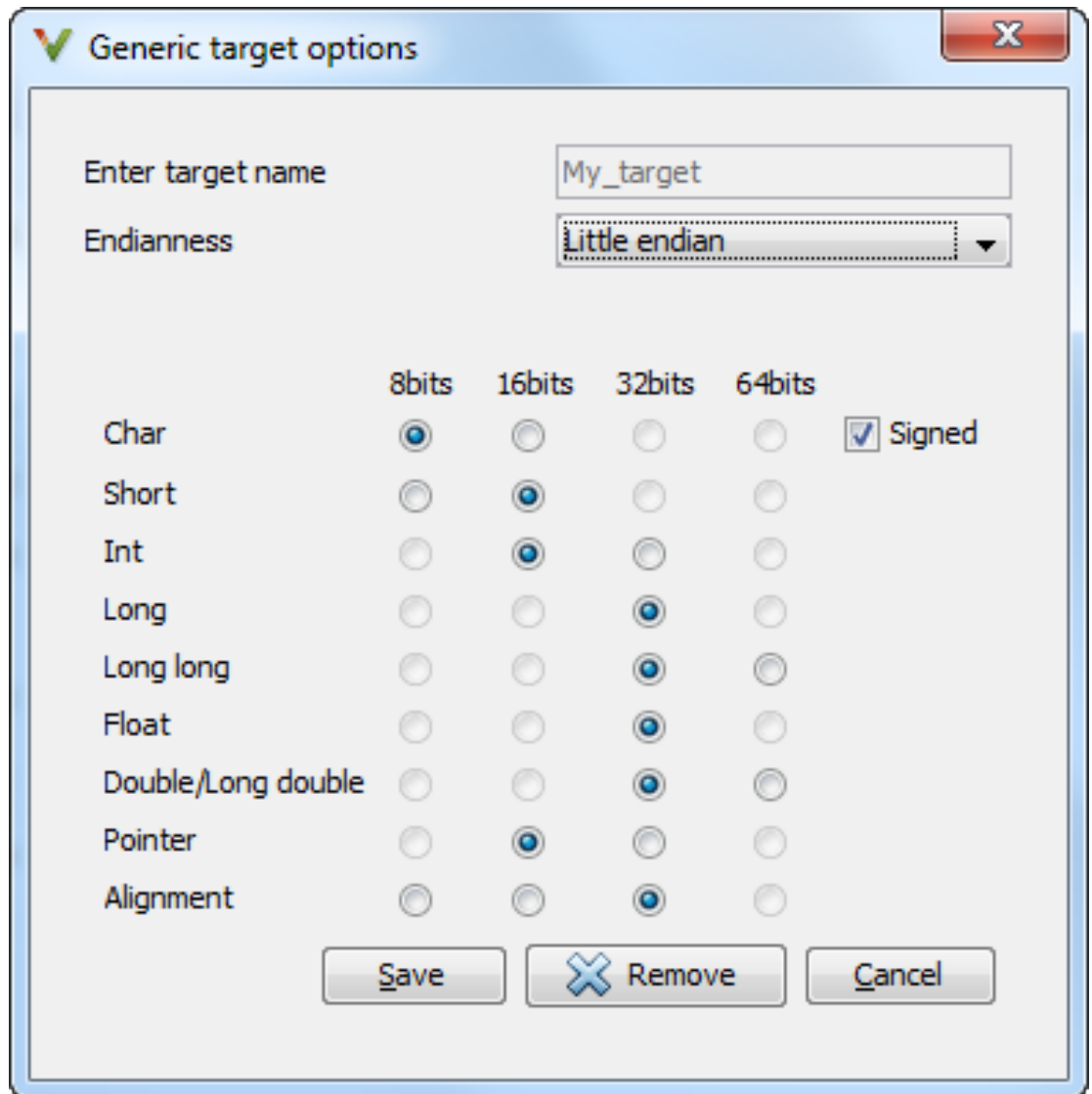
You specify a pointer size of 16 bits. The maximum object size allocated to a pointer, and therefore the maximum allowed size for an object, can be $2^{16}-1$ bytes. However, you declare a structure as follows:

- ```
struct S
{
 char tab[65536];
}s;
```
- ```
struct S
{
    char tab[65534];
    int val;
}s;
```

Solution

- 1 Check the pointer size that you specified through your target processor type. For more information, see “Target processor type (C/C++)”.

For instance, in the following, the pointer size for a custom target `My_target` is 16 bits.



- 2 Change your code or specify a different pointer size.

For instance, you can:

- Declare an array of smaller size in the structure.

If you are using a predefined target processor type, the pointer size is likely to be the same as the pointer size on your target architecture. Therefore, your declaration might cause errors on your target architecture.

- Change the pointer size of the target processor type that you specified, if possible.

Otherwise, specify another target processor type with larger pointer size or define your own target processor type. For more information on defining your own processor type, see “Generic target options (C/C++)”.

Note: Polyspace imposes an internal limit of 128 MB on the size of data structures. Even if your target processor type specification allows data structures of larger size, this internal limit constrains the data structure sizes.

Errors From Special Characters

Polyspace does not fully support extended ASCII characters, such as accented letters or Kanji characters. If you use extended ASCII in your file or folder names, your Polyspace analysis may fail due to file access errors. Error messages you might see include:

- No source files to analyze.
- Control character not valid.
- Cannot create directory *Folder_Name*.

Workaround

Change the unsupported ASCII characters to standard US-ASCII characters.

Initialization of Static Class Members (C++)

When a data member of a class is declared `static` in the class definition, it is a *static member* of the class. You must initialize static data members outside the class because they exist even when no instance of the class has been created.

```
class Test
{
public:

    static int m_number = 0;
};
```

Error message:

Error: a member with an in-class initializer must be const

Corrected code:

in file Test.h	in file Test.cpp
<pre>class Test { public: static int m_number; };</pre>	<pre>int Test::m_number = 0;</pre>

Double Declarations of Standard Template Library Functions

Consider the following code.

```
#include <list>

void f(const std::list<int*>::const_iterator it) {}
void f(const std::list<int*>::iterator it) {}
void g(const std::list<int*>::const_reverse_iterator it) {}
void g(const std::list<int*>::reverse_iterator it) {}
```

The declared functions belong to `list` container classes with different iterators. However, the software generates the following compilation errors:

```
error: function "f" has already been defined
error: function "g" has already been defined
```

You would also see the same error if, instead of `list`, the specified container was `vector`, `set`, `map`, or `deque`.

To avoid the double declaration errors, do one of the following:

- Deactivate automatic stubbing of standard template library functions. For more information, see “No STL stubs (C++)”.
- Define the following Polyspace preprocessing directives:
 - `__PST_STL_LIST_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_VECTOR_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_SET_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_MAP_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_DEQUE_CONST_ITERATOR_DIFFER_ITERATOR__`

For example, for the given code, run verification at the command line with the following flag. The flag defines the appropriate directive for the `list` container.

```
-D __PST_STL_LIST_CONST_ITERATOR_DIFFER_ITERATOR__
```

For more information on defining preprocessor directives, see “Preprocessor definitions (C/C++)”.

GNU Dialect

If you compile your code using a GNU C++ compiler, specify one of the GNU dialects for the Polyspace analysis. For more information, see “Dialect (C++)”.

If you specify one of the GNU dialects, Polyspace does not produce an error during the **Compile** phase because of assembly language keywords such as `__asm__` and `__volatile__`. However, Polyspace ignores the content of the assembly-language code for the analysis.

Polyspace software supports the following GNU elements:

- Variable length arrays
- Anonymous structures:

```
void f(int n) { char tmp[n] ; /* ... */ }
```

```
union A {  
  struct {  
    double x;  
    double y;  
    double z;  
  };  
  double tab[3];  
} a;
```

```
void main(void) {  
  assert(&(a.tab[0]) == &(a.x));  
  
}
```

- Other syntactic constructions allowed by GCC, except as noted below.
- Statement expressions:

```
int i = ({ int tmp ; tmp = f() ; if (tmp > 0 ) { tmp = 0 ; } tmp ; })
```

Partial Support

Zero-length arrays have the same support as in Visual Mode. They are allowed when used through a pointer, but not in a local variable.

Syntactic Support Only

Polyspace software provides syntactic support for the following options, but not semantic support:

- `__attribute__(...)` is allowed, but generally not taken into account.
- No special stubs are computed for predeclared functions such as `__builtin_cos`, `__builtin_exit`, and `__builtin_fprintf`).

Not Supported

The following options are not supported:

- The keyword `__thread`
- Taking the address of a label:


```
{ L : void *a = &&L ; goto *a ; }
```
- General C99 features supported by default in GCC, such as complex built-in types (`__complex__`, `__real__`, etc.).
- Extended designators initialization:

```
struct X { double a; int b[10] } x = { .b = { 1, [5] =2 },
    .b[3] = 1, .a = 42.0 };
```

- Nested functions

Examples

Example 1: `__asm__ __volatile__` keyword

In the following example, for the `inb_p` function to manage the return of the local variable `_v`, the `__asm__ __volatile__` keyword is used as follows:

```
extern inline unsigned char
inb_p (unsigned short port)
{
    unsigned char _v;

    __asm__ __volatile__ ("inb %w1,%0\noutb %%a1,$0x80":"=a"
        (_v):"Nd" (port));
    return _v;
}
```

```
}  
...
```

Although Polyspace does not produce an error during the **Compile** phase, it ignores the assembly code. An orange **Non-initialized local variable** error appears on the `return` statement after verification.

Example 2: Anonymous Structure

The following example shows an unnamed structure supported by GNU:

```
class x  
{  
public:  
  
    struct {  
        unsigned int a;  
        unsigned int b;  
        unsigned int c;  
    };  
    unsigned short pcia;  
    enum{  
        ea = 0x1,  
        eb = 0x2,  
        ec = 0x3  
    };  
  
    struct {  
        unsigned int z1;  
        unsigned int z2;  
        unsigned int z3;  
        unsigned int z4;  
    };  
};  
  
int main(int argc, char *argv[])  
{  
    class x myx;  
  
    myx.a = 10;  
    myx.z1 = 11;  
    return(0);  
}
```

ISO versus Default Dialects

The ISO dialect strictly follows the ISO/IEC 14882:1998 ANSI C++ standard. If you specify the option `iso` for “Dialect (C++)”, the Polyspace compiler might produce permissiveness errors. The following code contains five common permissiveness errors that occur if you specify the option. These errors are explained in detail following the code.

If you do not specify the option, the Polyspace compiler uses a default dialect that many C++ compilers use; the default dialect is more permissive with regard to the C++ standard.

Original code (file `permissive.cpp`):

```
class B {} ;
class A
{
    friend B ;
    enum e ;
    void f() {
        long float ff = 0.0 ;
    }
    enum e { OK = 0, KO } ;
};
template <class T>
struct traits
{
    typedef T * pointer ;
    typedef T * pointer ;
};
template<class T>
struct C
{
    typedef traits<T>::pointer pointer ;
};

void main()
{
    C<int> c;
}
```

If you use `iso` for dialect, the following errors occur.

Error message	Original code	Corrected code
error: omission of "class" is nonstandard	friend B;	friend class B;
forward declaration of enum type is nonstandard	enum e;	The line must be removed.
invalid combination of type specifiers	long float ff = 0.0;	double ff = 0.0
class member typedef may not be redeclared	Second instance of typedef T * pointer;	The line must be removed.
nontype "traits<T>::pointer [with T=T]" is not a type name	typedef traits<T>::pointer	typedef <i>typename</i> traits<T>::pointer pointer

The error messages disappear if you specify `none` for dialect.

Visual Dialects

The following messages appear if the compiler is based on a Visual[®] dialect. For more information, see “Dialect (C++)”.

Import Folder

When a Visual application uses `#import` directives, the Visual C++ compiler generates a header file with extension `.tlh` that contains some definitions. To avoid compilation errors during Polyspace analysis, you must specify the folder containing those files.

Original code:

```
#include "stdafx.h"
#include <comdef.h>
#import <MsXml.tlb>
MSXML::_xml_error e ;
MSXML::DOMDocument* doc ;
int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

Error message:

```
..\sources/ImportDir.cpp", line 7: catastrophic error: could not
open source file ".\MsXml.tlh"
    #import <MsXml.tlb>
           ^
```

The Visual C++ compiler generates these files in its “build-in” folder (usually Debug or Release). In order to provide those files:

- Build your Visual C++ application.
- Specify your build folder for the Polyspace analysis. For more information on the analysis option, see “Import folder (C++)”.

pragma Pack

Using a different value with the compile flag (`#pragma pack`) can lead to a linking error message.

Original code:

test1.cpp	type.h	test2.cpp
<pre>#pragma pack(4) #include "type.h"</pre>	<pre>struct A { char c ; int i ; } ;</pre>	<pre>#pragma pack(2) #include "type.h"</pre>

Error message:

```
Pre-linking C++ sources ...
"./sources/type.h", line 2: error: declaration of class "A" had
a different meaning during compilation of "test1.cpp"
(class types do not match)
  struct A
    ^
   detected during compilation of secondary translation unit
"test2.cpp"
```

To continue the analysis, use the option “Ignore pragma pack directives (C++)”.

Eclipse Java Version Incompatible with Polyspace Plug-in

In this section...

“Issue” on page 17-31

“Cause” on page 17-31

“Solution” on page 17-31

Issue

After installing the Polyspace plug-in for Eclipse, when you open the Eclipse or Eclipse-based IDE, you see this error message:

```
Java 7 required, but the current java version is 1.6.  
You must install Java 7 before using Polyspace plug in.
```

You see this message even if you install Java 7 or higher.

Cause

Despite installing Java 7 or higher, the Eclipse or Eclipse-based IDE still uses an older version.

Solution

Make sure that the Eclipse or Eclipse-based IDE uses the compatible Java version.

- 1 Open the `executable_name.ini` file that occurs in the root of your Eclipse installation folder.

If you are running Eclipse, the file is `eclipse.ini`.

- 2 In the file, just before the line `-vmargs`, enter:

```
-vm  
java_install\bin\javaw.exe  
Here, java_install is the Java installation folder.
```

For instance, your product installation comes with the required Java version for certain platforms. You can force the Eclipse or Eclipse-based IDE to use this version. In your `.ini` file, enter the following just before the line `-vmargs`:

-vm

Matlab_install\sys\java\jre\arch\jre\bin\javaw.exe

Here, *Matlab_install* is your product installation folder, for instance, C:\MATLAB\R2015b\ and *arch* is win32 or win64 depending on the product platform.