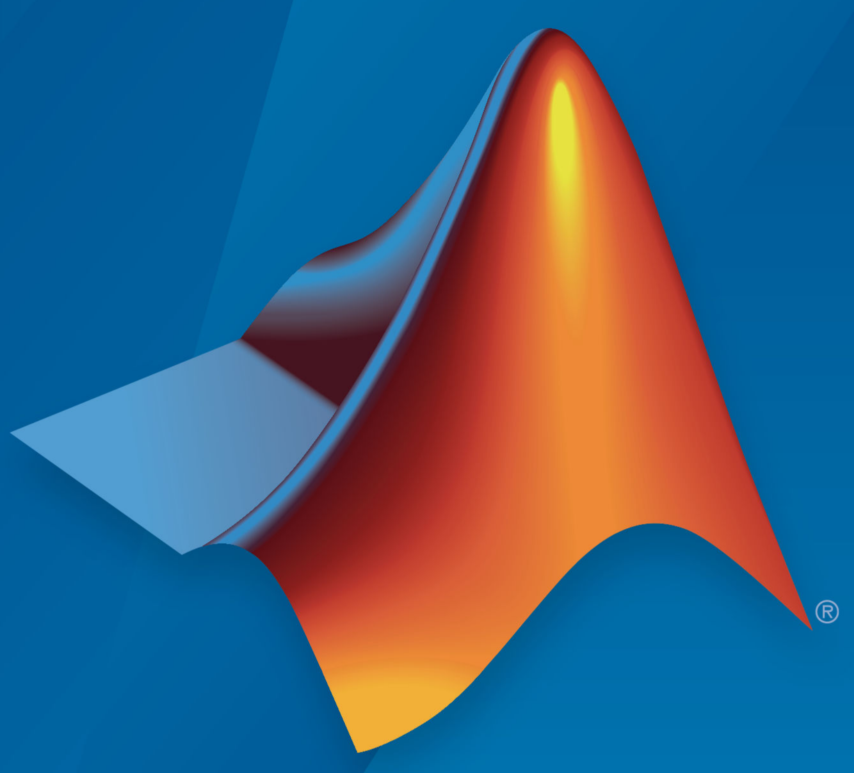
Polyspace® Bug Finder™

User's Guide



MATLAB® & SIMULINK®

R2018a

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 2013–2018 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)
October 2015	Online Only	Rereleased for Version 1.3.1 (Release 2015aSP1)
March 2016	Online Only	Revised for Version 2.1 (Release 2016a)
September 2016	Online Only	Revised for Version 2.2 (Release 2016b)
March 2017	Online Only	Revised for Version 2.3 (Release 2017a)
September 2017	Online Only	Revised for Version 2.4 (Release 2017b)
March 2018	Online Only	Revised for Version 2.5 (Release 2018a)

Contents

1 Run Polyspace Analysis on Desktop

Add Source Files for Analysis in Polyspace User Interface . . . **1-2**
Add Sources from Build Command **1-3**
Add Sources Manually **1-5**

Run Polyspace Analysis on Desktop **1-8**
Arrange Layout of Windows for Project Setup **1-9**
Set Product and Result Location **1-9**
Start and Monitor Analysis **1-10**
Fix Compilation Errors **1-11**
Open Results **1-11**

Project and Results Folder Contents **1-13**
Files in the Results Folder **1-13**

Storage of Temporary Files **1-15**

Create Project Using Visual Studio Information **1-16**

Create Project Using Configuration Template **1-19**
Why Use Templates **1-19**
Use Predefined Template **1-19**
Create Your Own Template **1-20**

Update Polyspace Project **1-24**
Change Folder Path **1-25**
Refresh Source List **1-26**
Refresh Project Created from Build Command **1-26**
Add Source and Include Folders **1-26**
Manage Include File Sequence **1-27**

Organize Layout of Polyspace User Interface **1-29**
Create Your Own Layout **1-29**

Save and Reset Layout **1-30**

Customize Polyspace User Interface **1-32**
Possible Customizations **1-33**
Storage of Polyspace User Interface Customizations **1-34**

2 Run Polyspace Analysis with Windows or Linux Scripts

Run Polyspace Analysis from Command Line **2-2**
Specify Sources and Analysis Options Directly **2-2**
Specify Sources and Analysis Options in Text File **2-3**
Create Options File from Build System **2-3**

polyspace-configure Source Files Selection Syntax **2-5**

Create Command-Line Script from Project File **2-8**
Generate Scripting Files **2-8**
Run an Analysis **2-9**

3 Run Polyspace Analysis with MATLAB Scripts

Run Polyspace Analysis by Using MATLAB Scripts **3-2**
Specify Multiple Source Files **3-2**
Check for MISRA C:2012 Violations **3-3**
Check for Specific Defects or Coding Rule Violations **3-4**
Find Files That Do Not Compile **3-4**
Run Analysis on Cluster **3-5**

Generate MATLAB Scripts from Polyspace User Interface . . . **3-6**

Troubleshoot Polyspace Analysis from MATLAB **3-9**

4 Run Polyspace Analysis on Remote Clusters

Run Polyspace Analysis on Remote Clusters **4-2**

Run Polyspace Analysis on Remote Clusters Using Scripts . . . **4-4**
Run Remote Analysis **4-4**
Manage Remote Analysis **4-6**

5 Run Polyspace Analysis on Generated Code

Run Polyspace Analysis on Code Generated with Embedded Coder **5-2**
Generate and Analyze Code **5-2**
Review Analysis Results **5-4**

Analyze Generated Code Using Polyspace Bug Finder **5-7**

Analyze Code Generated from Simulink Subsystem **5-10**
Open Model **5-10**
Generate Code **5-11**
Analyze Code **5-12**
Review Analysis Results **5-12**
Trace Errors Back to Model and Fix Them **5-13**
Check for Coding Rule Violations **5-16**
Annotate Blocks to Justify Results **5-16**

Analyze S-Function Code **5-18**
S-Function Analysis Workflow **5-18**
Compile S-Functions to Be Compatible with Polyspace **5-18**
Example S-Function Analysis **5-19**

Recommended Model Configuration Parameters for Polyspace Analysis **5-20**

Configure Advanced Polyspace Options in Simulink **5-23**
Configure Options **5-23**
Share and Reuse Configuration **5-25**

How Polyspace Analysis of Generated Code Works **5-28**

Default Polyspace Options for Code Generated with Embedded Coder **5-29**
Default Options **5-29**
Constraint Specification **5-29**
Recommended Polyspace options for Verifying Generated Code **5-30**
Hardware Mapping Between Simulink and Polyspace **5-30**

Run Polyspace Analysis on Code Generated with TargetLink **5-32**
Configure and Run Analysis **5-32**
Review Analysis Results **5-33**

Default Polyspace Options for Code Generated with TargetLink **5-34**
TargetLink Support **5-34**
Default Options **5-34**
Lookup Tables **5-35**
Data Range Specification **5-35**
Code Generation Options **5-36**

Troubleshoot Navigation from Code to Model **5-37**
Links from Code to Model Do Not Appear **5-38**
Links from Code to Model Do Not Work **5-38**
Your Model Already Uses Highlighting **5-38**

Run Polyspace on C/C++ Code Generated from MATLAB Code **5-40**
Prerequisites **5-40**
Run Polyspace Analysis **5-40**
Review Analysis Results **5-42**
Run Analysis for Specific Design Range **5-44**

Configure Advanced Polyspace Options in MATLAB Coder App **5-47**
Configure Options **5-47**
Share and Reuse Configuration **5-49**

6 Run Polyspace Analysis in IDE Plugins

Run Polyspace Analysis in Eclipse **6-2**
Configure and Run Analysis **6-4**
Review Analysis Results **6-6**

Specify Polyspace Compiler Options Through Eclipse Project **6-8**
Eclipse Refers Directly to Your Compilation Toolchain **6-8**
Eclipse Uses Your Compilation Toolchain Through Build Command **6-9**

7 Configure Polyspace Analysis

Specify Polyspace Analysis Options **7-2**
Polyspace User Interface **7-2**
Windows or Linux Scripts **7-3**
MATLAB Scripts **7-3**
Eclipse and Eclipse-based IDEs **7-4**
Simulink **7-4**
MATLAB Coder App **7-4**

8 Configure Target and Compiler Options

Specify Target Environment and Compiler Behavior **8-2**
Extract Options from Build Command **8-3**
Specify Options Explicitly **8-4**

Provide Standard Library Headers for Polyspace Analysis **8-6**

Requirements for Project Creation from Build Systems **8-8**
Compiler Requirements **8-8**
Build Command Requirements **8-9**

Language Extensions Supported by Default **8-11**

Supported Keil or IAR Language Extensions **8-13**
Special Function Register Data Type **8-13**
Keywords Removed During Preprocessing **8-14**

Supported C++ 2011 Language Extensions **8-15**

Remove or Replace Keywords Before Compilation **8-18**
Remove Unrecognized Keywords **8-18**
Remove Unrecognized Function Attributes **8-20**

Gather Compilation Options Efficiently **8-22**

9 Configure Inputs and Stubbing Options

Specify External Constraints **9-2**
Create Constraint Template **9-2**
Update Existing Template **9-3**
Specify Constraints in Code **9-4**

External Constraints for Polyspace Analysis **9-6**

XML File Format for Constraints **9-11**
Syntax Description — XML Elements **9-11**
Valid Modes and Default Values **9-16**

10 Configure Multitasking Analysis

Analyze Multitasking Programs in Polyspace **10-2**
Configure Analysis **10-3**
Review Analysis Results **10-4**

Auto-Detection of Thread Creation and Critical Section in Polyspace **10-6**
Multitasking Routines that Polyspace Can Detect **10-6**
Example of Automatic Thread Detection **10-8**
Naming Convention for Automatically Detected Threads . . . **10-11**
Limitations of Automatic Thread Detection **10-12**

Configuring Polyspace Multitasking Analysis Manually . . . **10-14**
Specify Options for Multitasking Analysis **10-14**
Adapt Code for Code Prover Multitasking Analysis **10-15**

Protections for Shared Variables in Multitasking Code **10-19**
Detect Unprotected Access **10-19**
Protect Using Critical Sections **10-20**
Protect Using Temporally Exclusive Tasks **10-21**
Protect Using Priorities **10-22**

11 Configure Coding Rules Checking and Code Metrics Computation

Check for Coding Rule Violations **11-2**
Configure Coding Rules Checking **11-2**
Review Coding Rule Violations **11-4**

Avoid Violations of MISRA C 2012 Rules 8.x **11-7**

Create Custom Coding Rules **11-11**

Format of Custom Coding Rules File **11-13**

Compute Code Complexity Metrics **11-14**
Impose Limits on Metrics **11-14**
Comment and Justify Limit Violations **11-17**

HIS Code Complexity Metrics **11-18**
Project **11-18**
File **11-18**
Function **11-18**

12 Coding Rule Sets and Concepts

Polyspace MISRA C 2004 and MISRA AC AGC Checkers **12-2**

MISRA C:2004 and MISRA AC AGC Coding Rules **12-3**
Supported MISRA C:2004 and MISRA AC AGC Rules **12-3**
Troubleshooting **12-4**
List of Supported Coding Rules **12-4**
Unsupported MISRA C:2004 and MISRA AC AGC Rules ... **12-44**

Software Quality Objective Subsets (C:2004) **12-47**
Rules in SQO-Subset1 .... **12-47**
Rules in SQO-Subset2 .... **12-48**

Software Quality Objective Subsets (AC AGC) **12-53**
Rules in SQO-Subset1 .... **12-53**
Rules in SQO-Subset2 .... **12-54**

Polyspace MISRA C:2012 Checkers **12-57**

Software Quality Objective Subsets (C:2012) **12-59**
Guidelines in SQO-Subset1 .... **12-59**
Guidelines in SQO-Subset2 .... **12-60**

Coding Rule Subsets Checked Early in Analysis **12-64**
MISRA C: 2004 and MISRA AC AGC Rules **12-64**
MISRA C: 2012 Rules **12-74**

Unsupported MISRA C:2012 Guidelines **12-84**

Polyspace MISRA C++ Checkers **12-85**

MISRA C++ Coding Rules **12-86**
Supported MISRA C++ Coding Rules **12-86**
Unsupported MISRA C++ Rules **12-111**

Software Quality Objective Subsets (C++) **12-116**
SQO Subset 1 – Direct Impact on Selectivity **12-116**
SQO Subset 2 – Indirect Impact on Selectivity **12-118**

Polyspace JSF C++ Checkers **12-123**

JSF C++ Coding Rules **12-124**
Supported JSF C++ Coding Rules **12-124**
Unsupported JSF++ Rules **12-147**

13 Configure Bug Finder Checkers

Choose Specific Bug Finder Defect Checkers **13-2**

Bug Finder Defect Groups **13-3**
Concurrency **13-3**
Cryptography **13-4**
Data flow **13-4**
Dynamic Memory **13-5**
Good Practice **13-5**
Numerical **13-5**
Object Oriented **13-6**
Programming **13-6**
Resource Management **13-6**
Static Memory **13-7**
Security **13-7**
Tainted data **13-7**

Results Found by Fast Analysis **13-9**
Polyspace Bug Finder Defects **13-9**
MISRA C: 2004 and MISRA AC AGC Rules **13-13**
MISRA C: 2012 Rules **13-21**
MISRA C++ 2008 Rules **13-29**

Check C/C++ Code for Security Standards **13-42**
Step 1: Check Code Against Standard **13-42**
Step 2: See Results with IDs from Standard **13-44**
Step 3: Fix or Justify Results with Standard IDs **13-45**
Step 4: Generate Reports **13-47**

CWE Coding Standard and Polyspace Results **13-49**
CWE and Polyspace Bug Finder **13-49**
Find CWE IDs from Polyspace Results **13-49**
Mapping Between CWE Identifiers and Polyspace Results . . **13-50**

Mapping Between CWE-658 or 659 and Polyspace Results **13-79**
CWE-658: Weaknesses in Software Written in C **13-79**
CWE-659: Weaknesses in Software Written in C++ **13-87**

CERT C Coding Standard and Polyspace Results **13-97**
CERT C and Polyspace Bug Finder **13-97**
Find CERT C Guideline Violations from Polyspace Results . . **13-97**
Mapping Between CERT C Rules and Polyspace Results . . . **13-98**
Mapping Between CERT C Recommendations and Polyspace Results **13-131**
Differences Between CERT C Standards and Defects **13-161**

ISO/IEC TS 17961 Coding Standard and Polyspace Results **13-163**
Find ISO/IEC TS 17961 Rule Violations from Polyspace Results **13-163**
Mapping Between ISO/IEC TS 17961 Rules and Polyspace Results **13-163**

14 Interpret Polyspace Bug Finder Results

Interpret Polyspace Bug Finder Results **14-2**
Interpret Result Details Message **14-3**
Find Root Cause of Result **14-4**

Investigate the Cause of Empty Results List **14-9**

Dashboard **14-11**

Concurrency Modeling **14-17**

Results List **14-19**

Source **14-22**
Tooltips **14-22**
Examine Source Code **14-22**
Expand Macros **14-24**
Manage Multiple Files in Source Pane **14-25**

View Code Block 14-27

Result Details **14-28**

15 Fix or Comment Polyspace Results

Address Polyspace Results Through Bug Fixes or Comments **15-2**
Comment in Results File **15-3**
Comment or Annotate in Code **15-4**

Annotate Code and Hide Known or Acceptable Results **15-6**
Code Annotation Syntax **15-6**
Syntax Examples **15-9**

Short Names of Bug Finder Defect Checkers **15-12**

Annotate Code for Known or Acceptable Results (Deprecated) **15-27**
Add Annotations from the Polyspace Interface **15-27**
Add Annotations Manually **15-28**

Define Custom Annotation Format **15-32**
Define Annotation Syntax Format **15-35**
Map Your Annotation to the Polyspace Annotation Syntax . . **15-40**

Annotation Description Full XML Template **15-42**
Example **15-46**

Import Comments from Previous Polyspace Analysis **15-49**
Import Comments from Another Analysis Result **15-49**
View Imported Comments That Do Not Apply **15-50**
Disable Automatic Comment Import from Last Analysis . . . **15-51**

Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results **15-52**
Mapping Multiple MISRA C: 2004 Annotations to the Same MISRA C: 2012 Result **15-53**

16 Manage Results

Filter and Group Results **16-2**
Filter Results **16-4**
Group Results **16-9**

Classification of Defects by Impact **16-11**
High Impact Defects **16-11**
Medium Impact Defects **16-14**
Low Impact Defects **16-18**

17 Generate Reports

Generate Reports **17-2**
Generate Reports from User Interface **17-2**
Generate Reports from Command Line **17-4**

Export Polyspace Analysis Results **17-6**
Export Results to Text File **17-6**
Export Results to MATLAB Table **17-8**
View Exported Results **17-8**

Visualize Bug Finder Analysis Results in MATLAB **17-10**
Export Results to MATLAB Table **17-10**
Generate Graphs from Results and Include in Report **17-10**

Customize Existing Bug Finder Report Template **17-15**
Prerequisites **17-15**
View Components of Template **17-15**
Change Components of Template **17-17**

18 Software Quality with Polyspace Metrics

Upload Results to Polyspace Metrics **18-2**
Manually Upload Results **18-2**
Automatically Upload Results (Batch Analysis Only) **18-3**

View Projects in Polyspace Metrics **18-5**
Upload Results **18-5**
Open Metrics Interface **18-5**
Review Metrics **18-6**
Compare Metrics Between Results **18-7**
Polyspace Metrics Interface **18-8**

Compare Metrics Against Software Quality Objectives **18-13**
Apply Predefined Objectives to Metrics **18-13**
Bug Finder Quality Objective Levels **18-14**
Customize Software Quality Objectives **18-19**

Web Browser Requirements for Polyspace Metrics **18-24**

View Results List in Polyspace Metrics **18-25**
Open Polyspace Metrics **18-25**
View Results List **18-26**
Download Results **18-27**

19 Troubleshooting in Polyspace Bug Finder

License Error –4,0 . . . . **19-3**
Issue **19-3**
Cause **19-3**
Solution **19-3**

View Error Information When Analysis Stops **19-4**
View Error Information in User Interface **19-4**
View Error Information in Log File **19-5**

Contact Technical Support **19-7**
Provide System Information **19-7**
Provide Information About the Issue **19-7**

Compiler Not Supported for Project Creation from Build Systems **19-9**
Issue **19-9**
Cause **19-9**
Solution **19-9**

Slow Build Process When Polyspace Traces the Build **19-19**
Issue **19-19**
Cause **19-19**
Solution **19-19**

Check if Polyspace Supports Build Scripts **19-20**
Issue **19-20**
Possible Cause **19-20**
Solution **19-20**

Troubleshooting Project Creation from MinGW Build **19-22**
Issue **19-22**
Cause **19-22**
Solution **19-22**

Troubleshooting Project Creation from Visual Studio Build **19-23**
Cannot Create Project from Visual Studio Build **19-23**
Compilation Error After Creating Project from Visual Studio Build **19-23**

Polyspace Cannot Find the Server **19-25**
Message **19-25**
Possible Cause **19-25**
Solution **19-25**

Job Manager Cannot Write to Database **19-26**
Message **19-26**
Possible Cause **19-26**
Workaround **19-26**

Undefined Identifier Error **19-28**
Issue **19-28**

Possible Cause: Missing Files **19-28**
Possible Cause: Unrecognized Keyword **19-28**
Possible Cause: Declaration Embedded in #ifdef Statements **19-29**
Possible Cause: Project Created from Non-Debug Build . . . **19-30**

Unknown Function Prototype Error **19-32**
Issue **19-32**
Cause **19-32**
Solution **19-32**

Error Related to #error Directive **19-34**
Issue **19-34**
Cause **19-34**
Solution **19-34**

Large Object Error **19-36**
Issue **19-36**
Cause **19-36**
Solution **19-36**

Errors Related to Generic Compiler **19-39**
Issue **19-39**
Cause **19-39**
Solution **19-39**

Errors Related to Keil or IAR Compiler **19-41**
Missing Identifiers **19-41**

Errors Related to Diab Compiler **19-42**
Issue **19-42**
Cause **19-42**
Solution **19-42**

Errors Related to TASKING Compiler **19-45**
Issue **19-45**
Cause **19-45**
Solution **19-46**

Conflicting Declarations in Different Translation Units . . . **19-47**
Issue **19-47**
Possible Cause: Variable Declaration and Definition Mismatch **19-48**

Possible Cause: Function Declaration and Definition Mismatch 19-49
Possible Cause: Macro-dependent Definitions 19-50
Possible Cause: Keyword Redefined as Macro 19-51
Possible Cause: Differences in Structure Packing 19-52

Errors from Conflicts with Polyspace Header Files **19-53**
Issue 19-53
Cause 19-53
Solution 19-53

Errors from Assertion or Memory Allocation Functions . . . **19-55**
Issue 19-55
Cause 19-55
Solution 19-55

Error from Special Characters **19-56**
Issue 19-56
Cause 19-56
Workaround 19-56

Errors from In-Class Initialization (C++) **19-57**

Errors from Double Declarations of Standard Template Library Functions (C++) **19-58**

Errors Related to GNU Compiler **19-59**
Issue 19-59
Cause 19-59
Solution 19-59

Errors Related to Visual Compilers **19-60**
Import Folder 19-60
pragma Pack 19-60
C++/CLI 19-61

Eclipse Java Version Incompatible with Polyspace Plug-in . **19-62**
Issue 19-62
Cause 19-62
Solution 19-62

Coding Rule Violations Not Displayed **19-64**
Issue 19-64

Possible Cause: Rule Checker Not Enabled **19-64**
Possible Cause: Rule Violations in Header Files **19-64**
Possible Cause: Rule Violations in Macros **19-64**
Possible Cause: Compilation Errors **19-65**

Insufficient Memory During Report Generation **19-66**
Message **19-66**
Possible Cause **19-66**
Solution **19-66**

Error from Disk Defragmentation and Antivirus Software . **19-67**
Issue **19-67**
Possible Cause **19-67**
Solution **19-67**

Errors with Temporary Files **19-68**
No Access Rights **19-68**
No Space Left on Device **19-68**
Cannot Open Temporary File **19-69**

1

Run Polyspace Analysis on Desktop

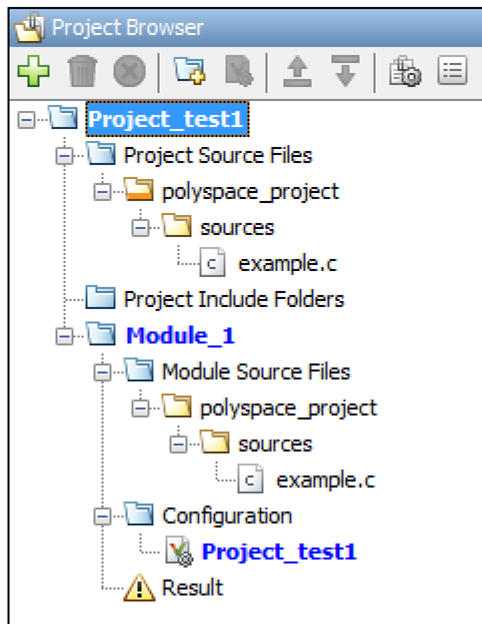
- “Add Source Files for Analysis in Polyspace User Interface” on page 1-2
- “Run Polyspace Analysis on Desktop” on page 1-8
- “Project and Results Folder Contents” on page 1-13
- “Storage of Temporary Files” on page 1-15
- “Create Project Using Visual Studio Information” on page 1-16
- “Create Project Using Configuration Template” on page 1-19
- “Update Polyspace Project” on page 1-24
- “Organize Layout of Polyspace User Interface” on page 1-29
- “Customize Polyspace User Interface” on page 1-32

Add Source Files for Analysis in Polyspace User Interface

To begin the Polyspace analysis, you must specify the path to your source files and headers.

You can specify your source paths explicitly or extract them from a build command (makefile). If you use a build command for building your source code or build your source code in an IDE (using an underlying build command), try extracting from the build command first. If Polyspace cannot trace your build command, manually add the paths to your source and include folders. You will also have to specify the target and compiler options later. See “Target and Compiler”.

Provide the source paths in a project. The source files show on the **Project Browser** pane.



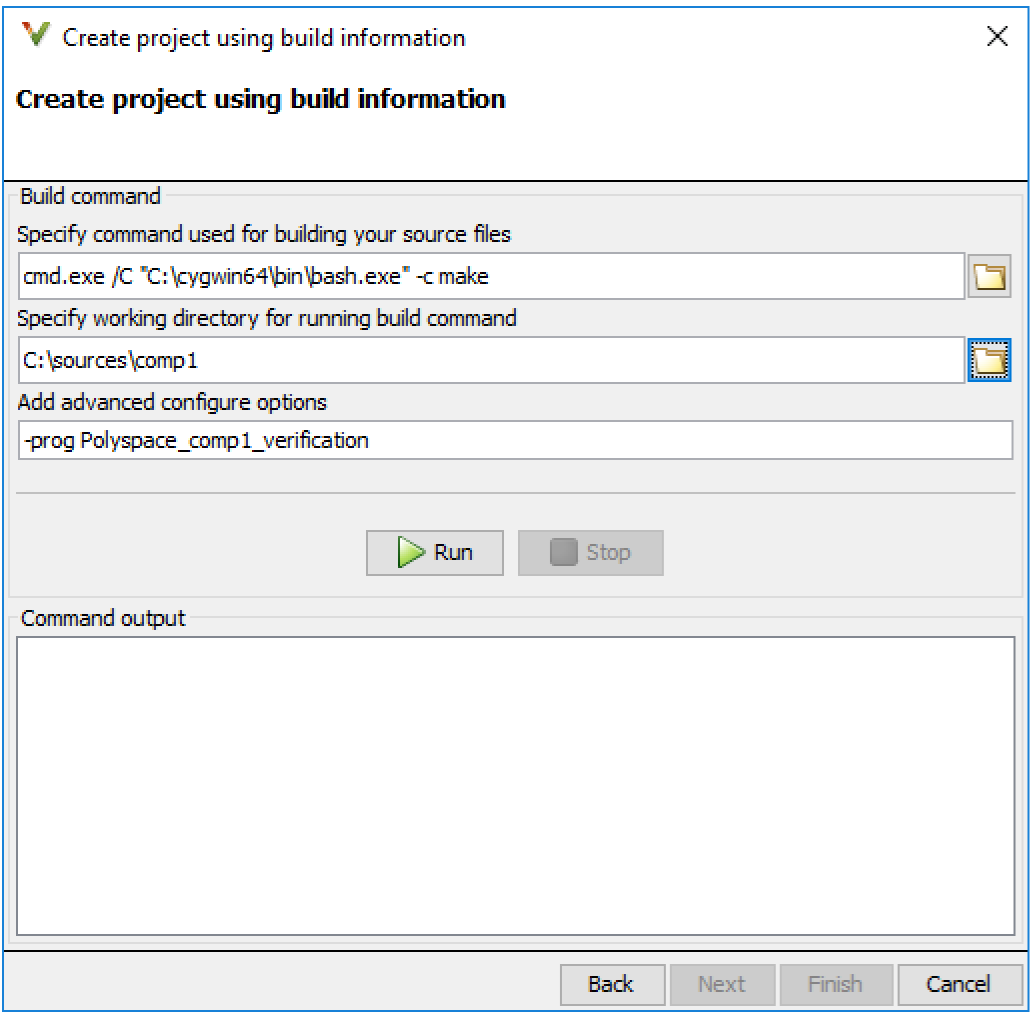
A corresponding `.psprj` file is created in the location where you saved the project. When you create a project, choose the default location for saving or enter a new location. To change the default location, select **Tools > Preferences** and use the options on the **Project and Results Folder** tab.

Add Sources from Build Command

Select **File > New Project**. Select **Create from build command**.

After providing a project name and location, on the next screen, enter this information:

- The build command, exactly as you run it on your code.
- The folder from which you run your build command.



When you click **Run**, Polyspace runs the build command and extracts the information necessary for creating a Polyspace project, specifically, source paths and compiler information.

If you build your source code within an IDE such as Visual Studio®, in the field for specifying the build command, enter the path to your executable, for instance, `C:`

`\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\VCExpress.exe`. When you click **Run**, Polyspace opens your IDE. In your IDE, perform a complete build of your code. When you close your IDE, Polyspace extracts your source paths and compiler information.

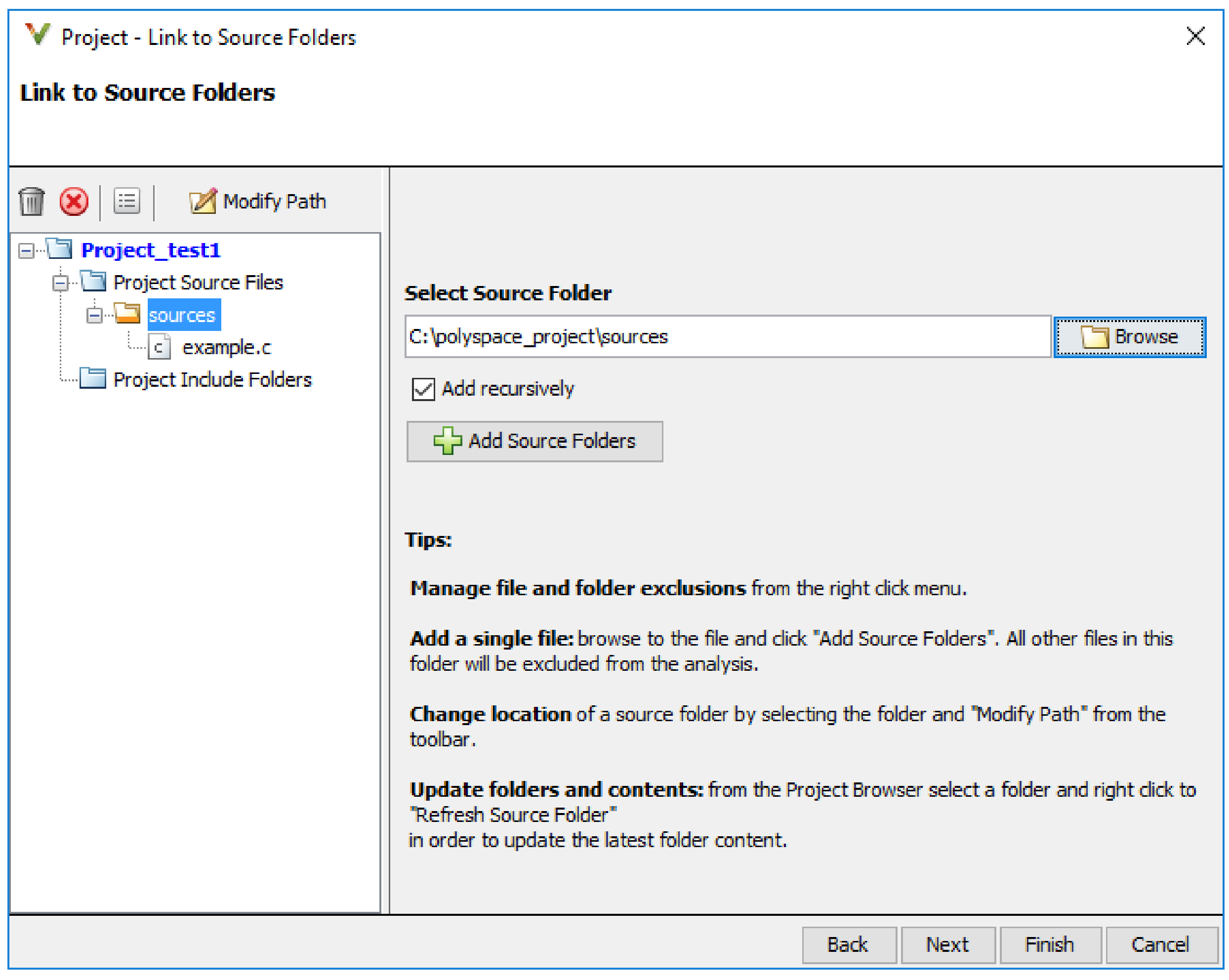
When you create a project from your build command, the **Project Browser** pane shows your source folders but not the include folders. In case you want to verify that your include folders were extracted, open the project file (with extension `.psprj`) in a text editor.

You can use additional options to modify the default project creation from build command. For instance, to create a Polyspace project despite build errors, in the **Add advanced configure options** field, you can enter the option `-allow-build-error`. To look up allowed options, see `polyspace-configure`.

Add Sources Manually

Select **File > New Project**.

After providing a project name and location, on the next screen, enter or navigate to the root folder containing your source files. With the **Add recursively** box checked, click **Add Source Folders**. All files in the folder and subfolders are added to your project.



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

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

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

For Standard Library headers such as `stdio.h`, if you know the path to the headers from your compiler, specify them explicitly. Otherwise, the analysis uses Polyspace implementation of the Standard Library headers, which in some special cases, might not match your compiler implementation.

Your project file with source and include folders show in the **Project Browser** pane. Later, if you add files to one of these folders, you can update your project. Right-click the folder that you want to update, or the entire **Project Source Files** folder, and select **Refresh Source Folder**.

You can also right-click to exclude files or add more folders to the project. The files that you add the first time are copied to the first module in your project. If you add new files later, you must explicitly right-click and add them to a module.

See Also

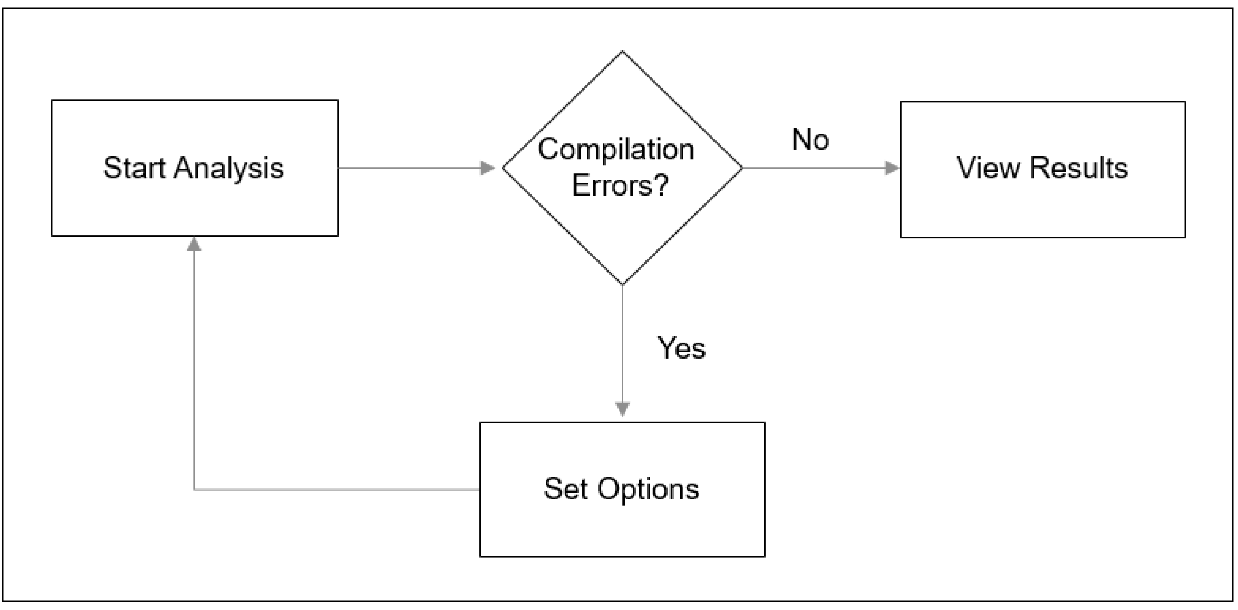
More About

- “Run Polyspace Analysis on Desktop” on page 1-8

Run Polyspace Analysis on Desktop

After you specify your source files and compiler on page 1-2, you can run the Polyspace analysis. This topic describes how to run an analysis in the Polyspace user interface, monitor progress, fix compilation issues, and open analysis results as available.

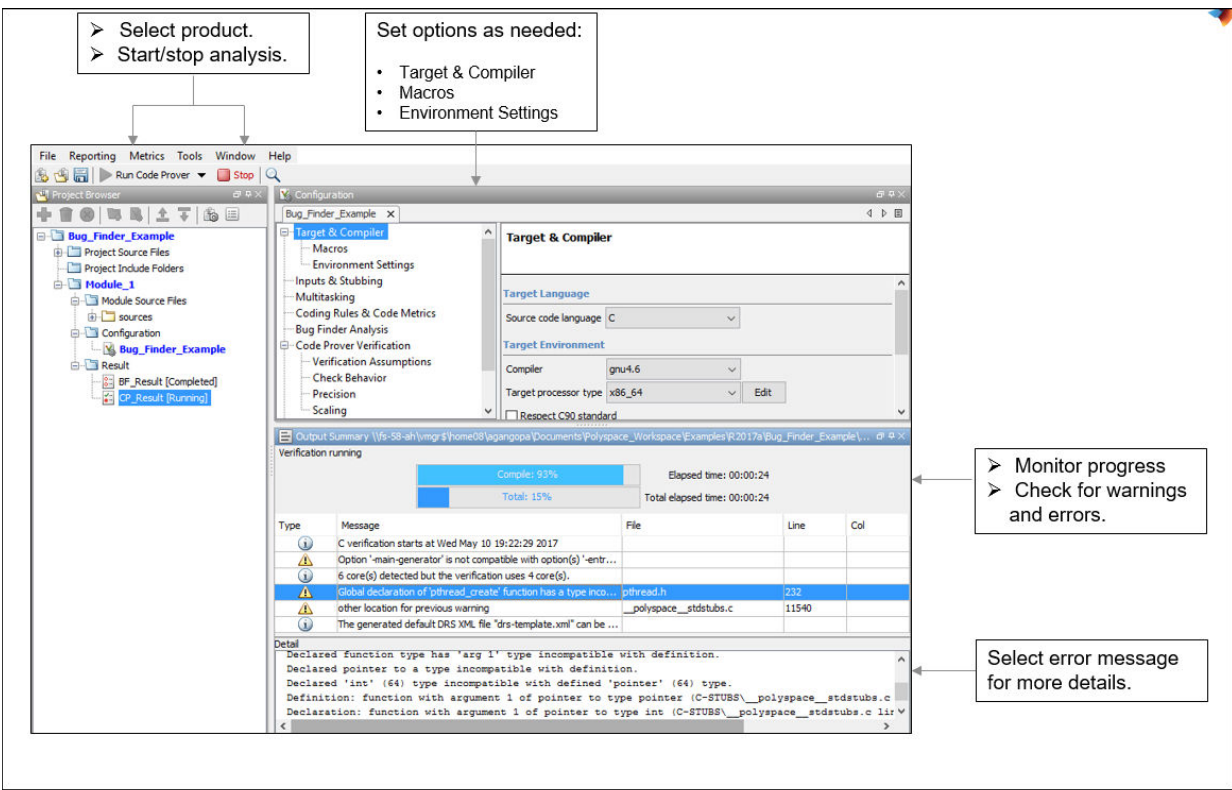
During analysis, Polyspace first compiles your code and then checks for bugs (Bug Finder) or proves code correctness (Code Prover). If you encounter compilation errors, read the error message and diagnose the root cause of the error. Often, to resolve the errors, you have to set some Polyspace configuration options and rerun the analysis.



You can run the analysis in the Polyspace user interface or by using scripts.

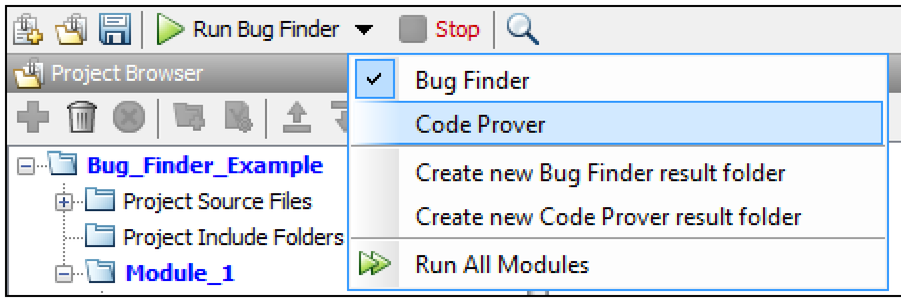
Arrange Layout of Windows for Project Setup

In the user interface, for a convenient distribution of windows, select **Window > Reset Layout > Project Setup**.



Set Product and Result Location

To switch products or create a separate folder for each run, use the dropdown beside the **Run** button.

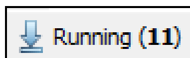


The results are stored in a subfolder `Module_#` of the project folder. To use a different folder naming convention or different storage location for results, select **Tools > Preferences** and use the options on the **Project and Results Folder** tab.

Start and Monitor Analysis

If your project has multiple modules, select the module that you want to analyze. Start the analysis. Monitor progress on the **Output Summary** pane.

- Bug Finder: You can see some results after partial analysis, because certain defect checkers do not need cross-functional information and can show results as soon as a function is analyzed. If results are available while the analysis is still running, you see the following icon beside the **Run** button:

  The icon indicates the number of results available. Click the icon to open the results. Once the analysis is over, the **Running** label in the icon changes to **Completed**. You can click the icon again to reload the full set of results.
- Code Prover: You can see results only after the analysis is complete. Code Prover is more likely to report compilation errors because it does a more rigorous analysis and must follow stricter rules for compilation. The progress bar distinguishes between the various phases of analysis starting from compilation.

Fix Compilation Errors

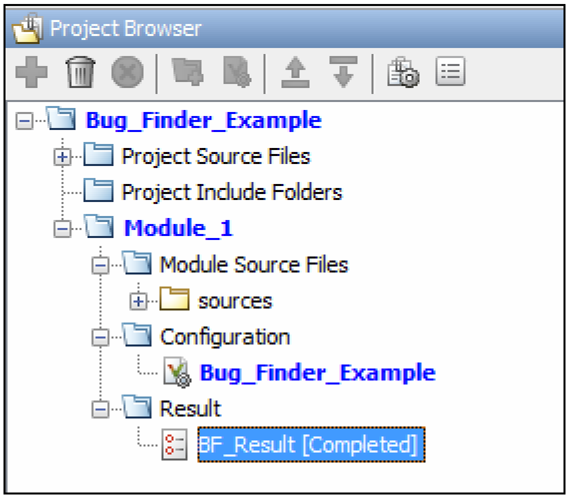
If compilation errors occur, the analysis continues with the remaining files that do not compile. The **Dashboard** pane shows that some files did not compile and links to the **Output Summary** pane for details. The **Output Summary** pane shows compilation errors with a icon.

To diagnose further, select the error message to see more details. Identify the line in your code responsible for the compilation error. You can use the error message details to understand how the line compiled with your compiler and what additional information Polyspace needs to mimic your compiler. See if you can work around the error using a Polyspace option. For more information, see “Troubleshooting in Polyspace Bug Finder”.

For more precise run-time error checking in Code Prover, it is recommended that you fix all compilation errors. Use the option `Stop analysis if a file does not compile (-stop-if-compile-error)`.

Open Results

After analysis, the results open automatically. To open results that you have closed, double-click the result on the **Project Browser** pane.



The Bug Finder (Code Prover) results are stored in a `.psbf` (`.pscp`) file in the results folder. For instance, if you save your project in `C:\Projects\`, a `.psbf` file for the Bug Finder analysis results on the first module `Module_1` is stored in `C:\Projects\Module_1\BF_Result`. See also "Project and Results Folder Contents" on page 1-13.

See Also

More About

- "Run Polyspace Analysis from Command Line" on page 2-2
- "Run Polyspace Analysis by Using MATLAB Scripts" on page 3-2
- "Interpret Polyspace Bug Finder Results" on page 14-2
- "Address Polyspace Results Through Bug Fixes or Comments" on page 15-2
- "Filter and Group Results" on page 16-2

Project and Results Folder Contents

When you run an analysis in the Polyspace user interface, Polyspace generates files that contain information about configuration options and analysis results.

The organization of Polyspace files in the physical folder location follows the hierarchy displayed in the Polyspace user interface. The project folder contains a subfolder for each module. In each module folder, there is one or more result subfolder, named `Result_#`. The number of result folders depends on whether you overwrite or retain previous results for each new run. To use a different folder naming convention or different storage location for results, select **Tools** > **Preferences** and use the options on the **Project and Results Folder** tab.

The project folder has the project file with extension `.psprj`. If you open a project from a previous release in the user interface, the project is upgraded for the new release. A backup of the old project file is saved with the extension `.bak.psprj`.

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 that store files required 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`

Storage of Temporary Files

Polyspace produces some temporary files when performing an analysis. If your analysis runs slow or you encounter errors such as running out of disk space, check your temporary file location. For more information on possible errors, see:

- “Errors with Temporary Files” on page 19-68
- “Reduce Verification Time” (Polyspace Code Prover)

To determine where to store temporary files, Polyspace looks for these environment variables in the following order:

- `RTE_TMP_DIR`: Define this environment variable only if you want to store Polyspace temporary files in a folder different from the standard temporary folders (defined by `TMPDIR` and such). You can see the current standard temporary folder by using the MATLAB® function `tempdir`.

  Note This path must be an absolute path to an existing folder on which the current user has access rights (for reading and writing).
- `TMPDIR`
- `TMP`
- `TEMP`

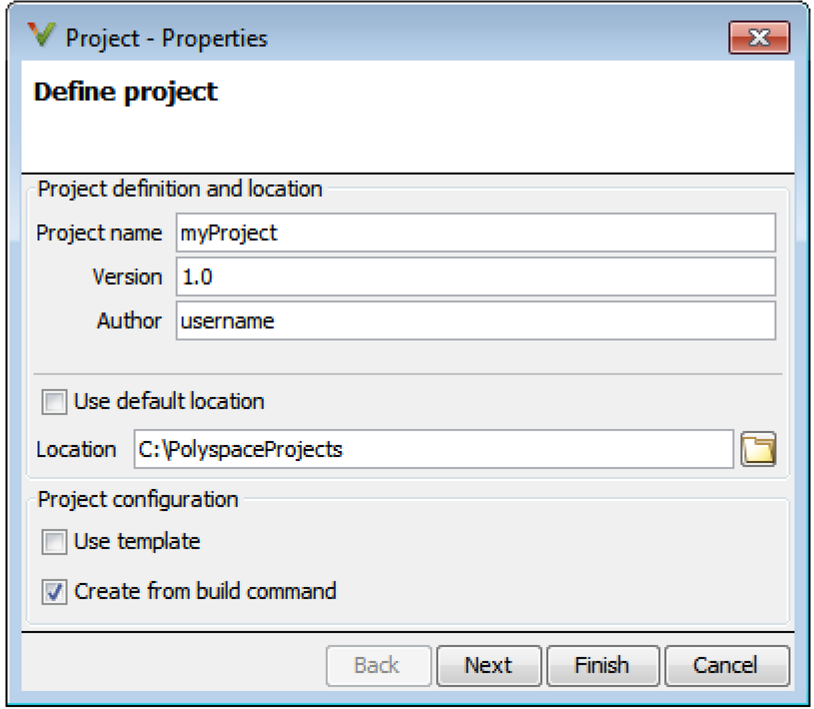
If one of these variables is defined, Polyspace uses that path for storing temporary files. If these environment variables are not defined, Polyspace stores temporary files in:

- `/tmp` on Linux® and Mac
- Folder specified with the `USERPROFILE` environment variable, folder returned from `GetWindowsDirectoryW` Windows® API, or `Temp` directory on Windows

Create Project Using Visual Studio Information

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

1. In the Polyspace interface, select **File > New Project**.
2. In the Project – Properties window, under **Project Configuration**, select **Create from build command** and click **Next**.



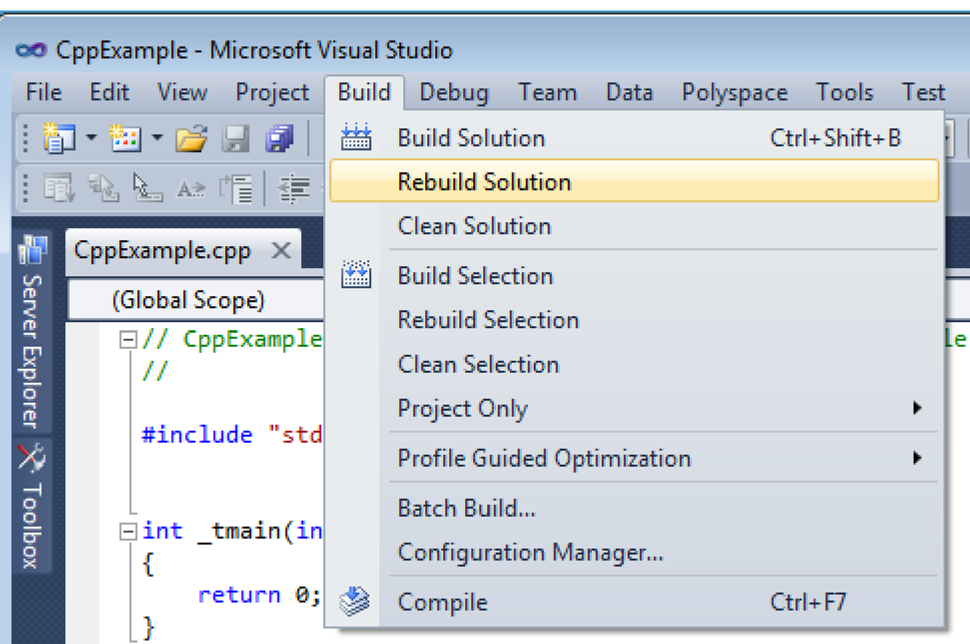
3. In the field **Specify command used for building your source files**, enter the full path to the Visual Studio executable. For instance, `"C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\VCExpress.exe"`.
4. In the field **Specify working directory for running build command**, enter `C:\`.

   Click Run.

   This action opens the Visual Studio environment.

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

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



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**.

See Also

More About

- “Troubleshooting Project Creation from Visual Studio Build” on page 19-23

Create Project Using Configuration Template

A configuration template is a predefined set of analysis options for a specific compilation environment.

Why Use Templates

Use templates to simplify your project setup. For instance, after you configure a project for a specific compilation environment, you can create a template out of the configuration. Using the template, you can reuse the configuration for projects that have the same compilation environment.

When creating a new project, you can do one of the following:

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

  Polyspace software provides predefined templates for common compilers such as `IAR`, `Kiel`, `Visual` and `VxWorks`. For additional templates, see Polyspace Compiler Templates.
- Set analysis options manually. You can then save your options as a template and reuse them later. You can also share the template with other users and enforce consistent usage of Polyspace Bug Finder in your organization.

Use Predefined Template

1. Select **File** > **New Project**.
2. On the Project – Properties dialog box, after specifying the project name and location, under **Project configuration**, select **Use template**.
3. On the next screen, select the template that corresponds to your compiler. For further details on a template, select the template and view the **Description** column on the right.

   If your compiler does not appear in the list of predefined templates, select **Baseline_C** or **Baseline_C++**.
4. On the next screen, add your source files and include folders.

Create Your Own Template

This example shows how to save a configuration from an existing project and create a new project using the saved configuration.

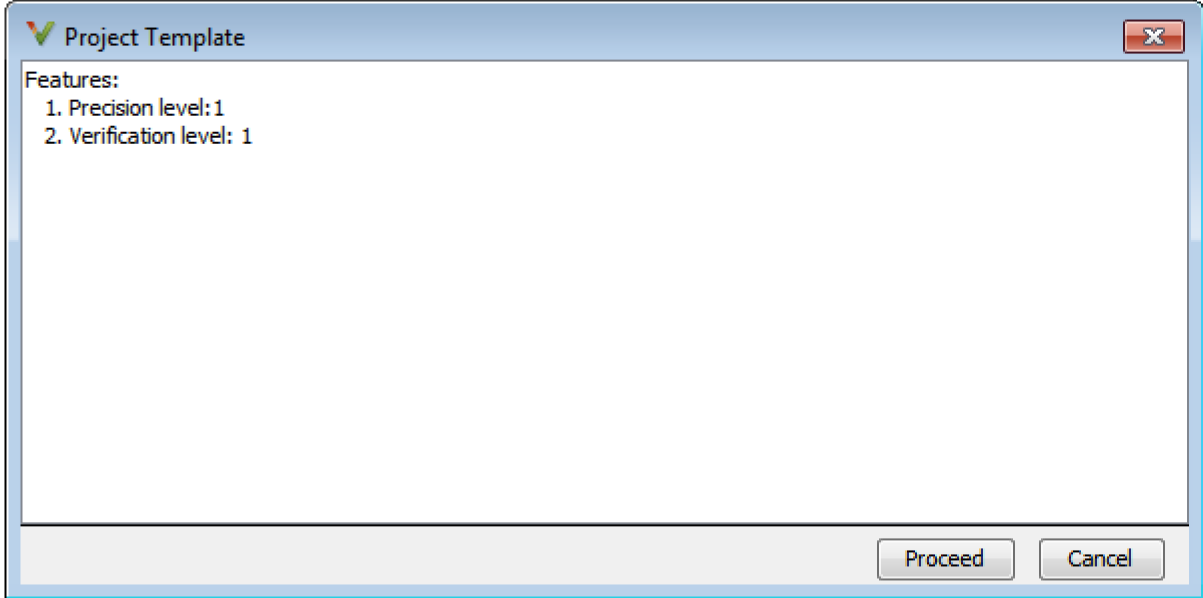
- To create a template from a project that is open on the **Project Browser** pane:

  1. Right-click the project 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.

     Suppose you create a Code Prover configuration template that runs Code Prover analysis to a precision level of 1 and a verification level of 1. See:

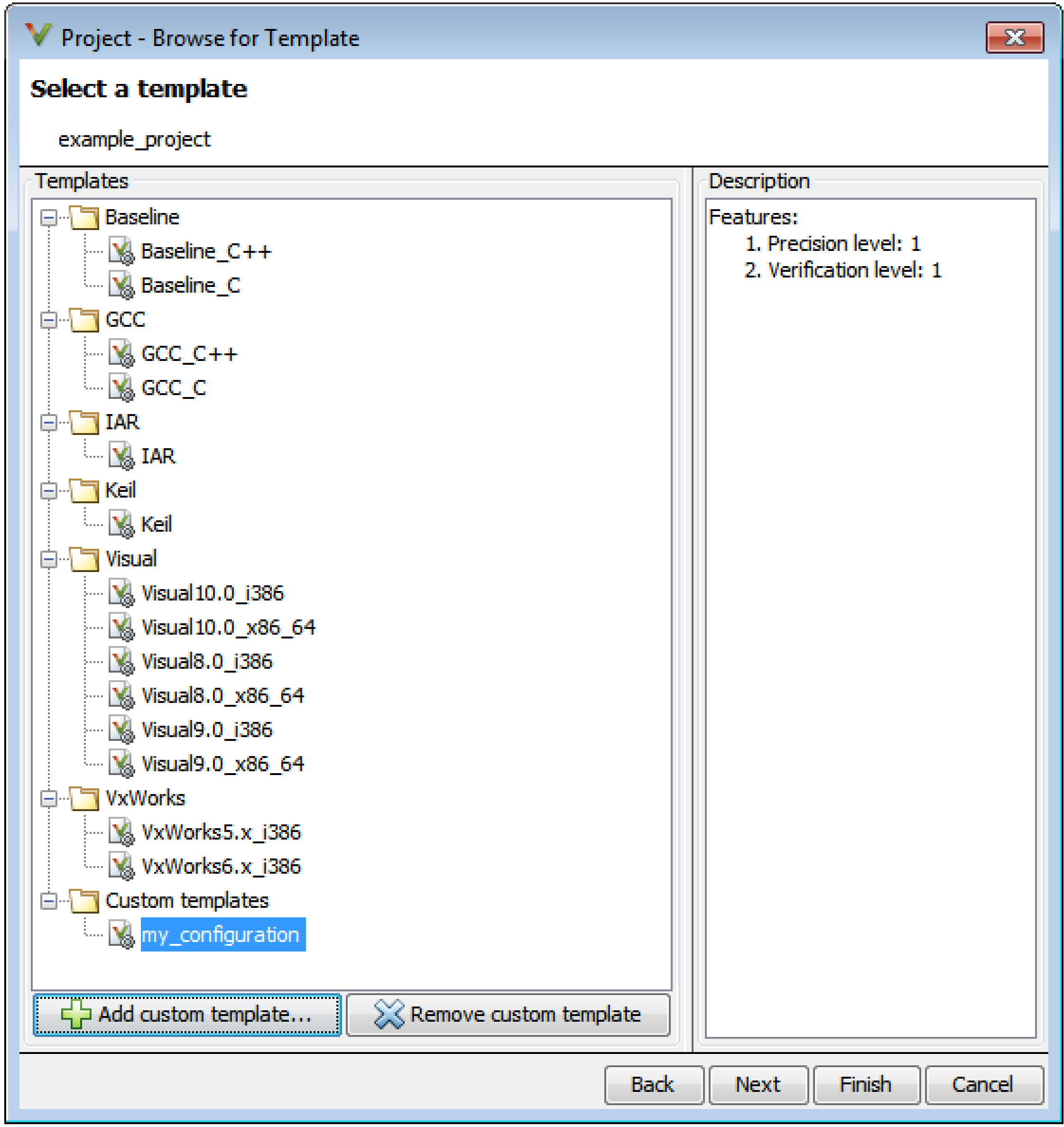
     - `Precision level (-O)`
     - `Verification level (-to)`

     You can enter this description for the template.



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

1. Select **Add custom template...**.
2. 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.


Project - Browse for Template
Select a template
example_project
Templates
Baseline
Baseline_C++
Baseline_C
GCC
GCC_C++
GCC_C
IAR
IAR
Keil
Keil
Visual
Visual10.0_i386
Visual10.0_x86_64
Visual8.0_i386
Visual8.0_x86_64
Visual9.0_i386
Visual9.0_x86_64
VxWorks
VxWorks5.x_i386
VxWorks6.x_i386
Custom templates
my_configuration
Add custom template...
Remove custom template
Description
Features:
1. Precision level: 1
2. Verification level: 1
Back
Next
Finish
Cancel

See Also

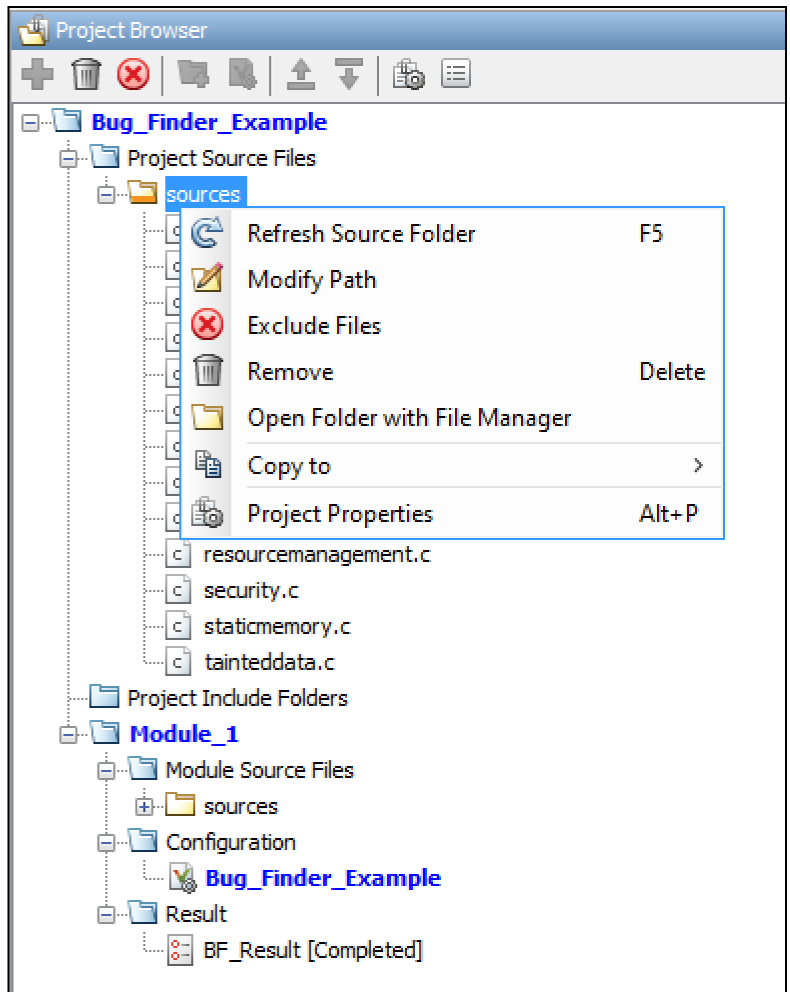
More About

- “Specify Polyspace Analysis Options” on page 7-2
- “Analysis Options”

Update Polyspace Project

To analyze your C/C++ source files with Bug Finder or Code Prover in the Polyspace user interface, you create a Polyspace project. During development, you can simply update this project and rerun the analysis for updated results. This topic describes the updates that you can make.

To begin updates, right-click your project on the **Project Browser** pane. You see a different set of options depending on the node that you right-click.



Change Folder Path

If you have moved the source folder that you added to your project, modify the path in your Polyspace project. You can also modify the folder path to point to a different version of the code in your version control system.

In the **Project Browser**, right-click the top sources folder and select **Modify Path**.Change the path to the new location.

To resync the files under this source folder, right-click your source folder and select **Refresh Source Folder**.

Refresh Source List

If you made changes to files in a folder already added to the project, you do not need to re-add the folder to your project. Refreshing your source file list looks for new files, removed files, and moved files.

Right-click your source folder and select **Refresh Source Folder**. The files in your Polyspace project refresh to match your file system.

Refresh Project Created from Build Command

If you created your project automatically from your build system, to update the project later by rerunning your build command, right-click the project folder and select **Update Project**.

You see the information that you entered when creating the original project. Click **Run** to retrace your build command and recreate the Polyspace project.

Add Source and Include Folders

If you want to change which files or folders are active in your project without removing them from your project tree, right-click the file or folder and select **Exclude Files**. The file appears with an symbol in your project indicating it is not considered for analysis. You can reinclude the files for analysis by right-clicking and selecting **Include Files**.

If you want to add additional source folders or include folders, right-click your project or the **Source** or **Include** folder in your project. Select **Add Source Folder** or **Add Include Folder**.

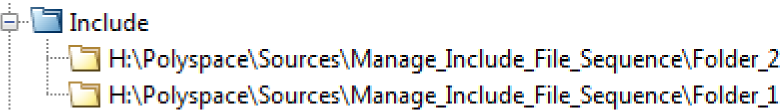
Before running an analysis, you must copy the source files to a module. Select the source files that you want to copy. To select multiple files together, press the **Ctrl** key while selecting the files. Right-click your selection. Select **Copy to > Module_*n***. *n* is the module number.

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, in your project, expand the **Include** folder. Select the include folder or folders that you want to move. To move the folder, click either or .

See Also

Related Examples

- “Add Source Files for Analysis in Polyspace User Interface” on page 1-2

Organize Layout of Polyspace User Interface

The Polyspace user interface has two default layouts of panes.

The default layout for project setup has the following arrangement of panes:

Project Browser	**Configuration**
	Output Summary

The default layout for results review has the following arrangement of panes:

Results List	**Result Details**
	Dashboard

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 one of the default layouts of the Polyspace user interface. Select **Window** > **Reset Layout** > **Project Setup** or **Window** > **Reset Layout** > **Results Review**.

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 List** 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.

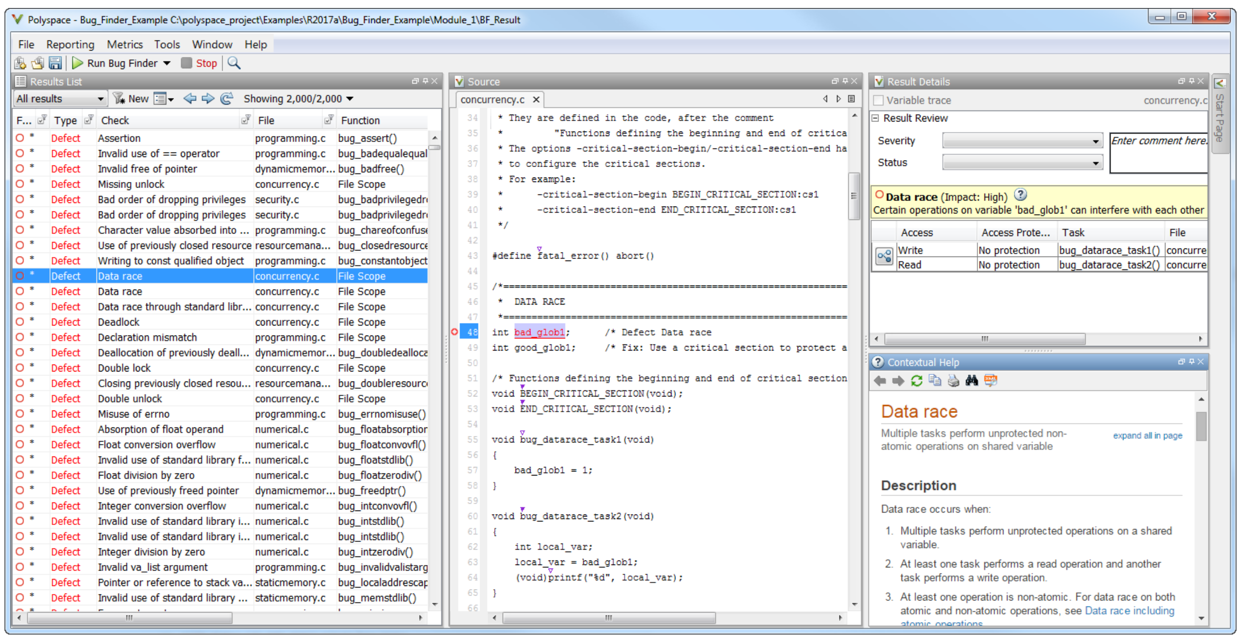
For instance, if **Project Browser** and **Results List** are tabbed on the same pane, dragging out **Project Browser** creates a pane with only **Project Browser** on it and floats this new pane.

- 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 your own layout 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** > ***`layout_name`***.
- To remove a saved layout from the **Reset Layout** list, select **Window** > **Remove Custom Layout** > ***`layout_name`***.

See Also

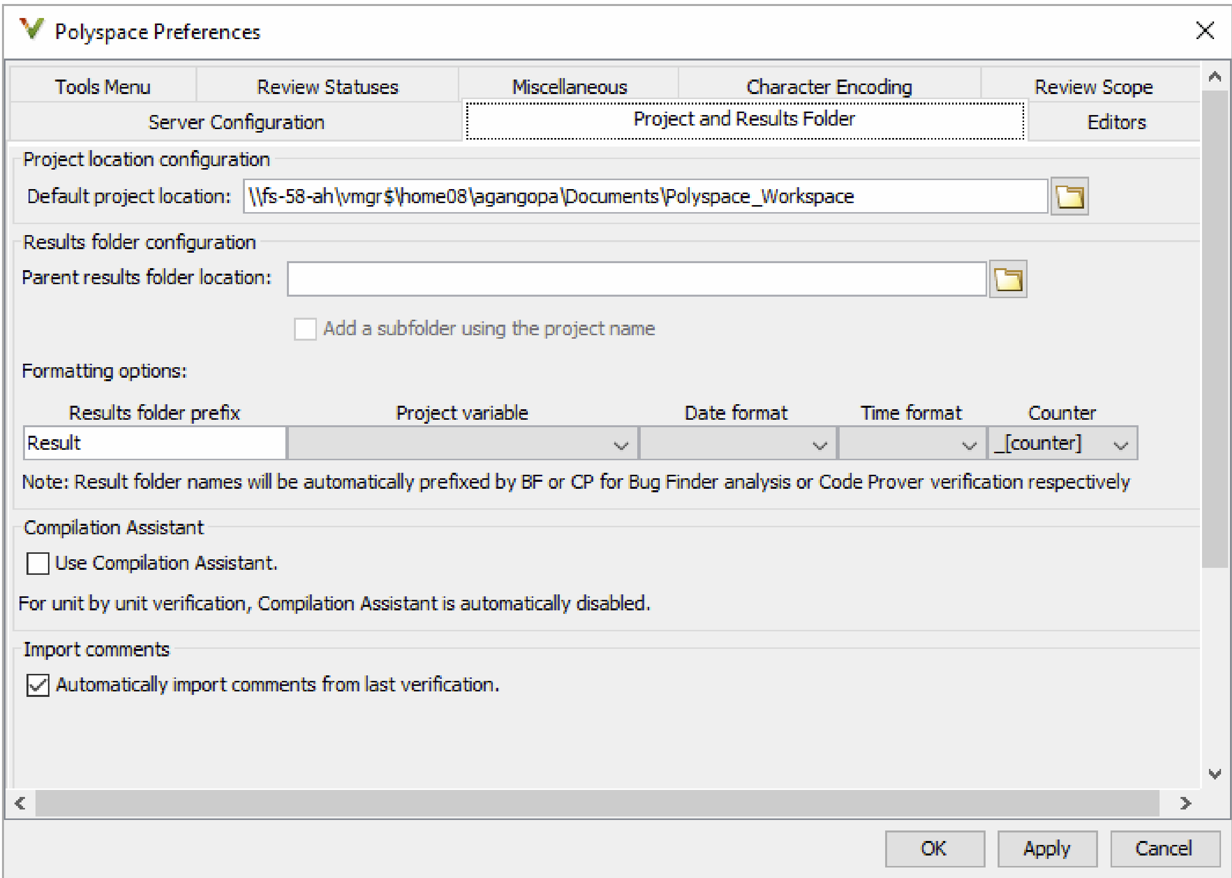
More About

- "Customize Polyspace User Interface" on page 1-32
- "Organize Layout of Polyspace User Interface" on page 1-29

Customize Polyspace User Interface

In this section...
"Possible Customizations" on page 1-33
"Storage of Polyspace User Interface Customizations" on page 1-34

You can customize various aspects of the Polyspace user interface, for instance, default project storage locations or default font size of source code. Select **Tools > Preferences**.



Possible Customizations

Change Default Font Size

To change the default font size in the Polyspace user interface, select 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**.

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

Specify External Text Editor

You can 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.

To change the text editor, select the **Editors** tab. From the **Text editor** drop-down list, select **External**. In the **Text editor** field, specify the path to your text editor. For example:

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

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 List** pane and select **Open Editor**, your source code opens at the location of the check.

Polyspace has already specified the command-line arguments for these editors: `Emacs`, `Notepad++` (Windows only), `UltraEdit`, `VisualStudio`, `WordPad` (Windows only) or `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.

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`.

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

Create Custom Review Status

When reviewing Polyspace results, you can assign a status such as `To fix` or `Justified`. See “Address Polyspace Results Through Bug Fixes or Comments” on page 15-2.

You can create your own statuses to assign. To create a new status, select the **Review Statuses** tab.

Storage of Polyspace User Interface Customizations

The software stores the settings that you specify through the Polyspace Preferences 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\polyspace_shared\polyspace_products.prf`

- Linux : `/home/$User/.matlab/polyspace_shared/polyspace_products.prf`

Run Polyspace Analysis with Windows or Linux Scripts

- “Run Polyspace Analysis from Command Line” on page 2-2
- “polyspace-configure Source Files Selection Syntax” on page 2-5
- “Create Command-Line Script from Project File” on page 2-8

Run Polyspace Analysis from Command Line

To run an analysis from a DOS or UNIX® command window, use the command `polyspace-bug-finder-nodesktop` or `polyspace-code-prover-nodesktop` followed by other options you wish to use. See also:

- `polyspace-bug-finder-nodesktop`
- `polyspace-code-prover-nodesktop`

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` or `polyspace-code-prover-nodesktop` command.

For instance:

- To specify sources, use the `-sources` option followed by a comma-separated list of sources.

  ```
  polyspace-bug-finder-nodesktop -sources C:\mySource\myFile1.c,C:\mySource\myFile2.c
  ```

  If your current folder contains a `sources` subfolder with the source files, you can omit the `-sources` flag. The analysis considers files in `sources` and all subfolders under `sources`.
- 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 the full list of options, enter the following at the command line:

```
polyspace-bug-finder-nodesktop -help
```

Specify Sources and Analysis Options in Text File

Instead of specifying the options directly, you can save the options in a text file and use the text file each time you run the analysis.

1. Create an options file called `listofoptions.txt` with your options. For example:

```
#These are the options for MyCodeProverProject
-lang c
-prog MyCodeProverProject
-author jsmith
-sources "mymain.c,funAlgebra.c,funGeometry.c"
-target x86_64
-compiler generic
-dos
-misra2 required-rules
-do-not-generate-results-for all-headers
-main-generator
-results-dir C:\Polyspace\MyCodeProverProject
```

2. Run Polyspace using options in the file `listofoptions.txt`.

```
polyspace-code-prover-nodesktop -options-file listofoptions.txt
```

See also `-options-file`.

Create Options File from Build System

If you use a build command (makefile) to build your source code, you can collect the sources and compiler options from your build command. Trace your build command to generate a text file with the required Polyspace options.

1. Create a list of Polyspace options using the configuration tool.

```
polyspace-configure -output-options-file \
        myOptions buildCommand
```

where *`buildCommand`* is the command you use to build your source code, for instance `make -B`.

See also `polyspace-configure`.

2. Run Polyspace using the options read from your build.

   ```
   polyspace-bug-finder-nodesktop -options-file myOptions \
           -results-dir myResults
   ```

   In addition to the options collected from your build command, you might want to add further options, for instance, to specify the defect checkers. You can append these options to the options file, add them directly at the command line or add them through a second options file (using another `-options-file` flag).
3. Open the results in the Polyspace user interface.

   ```
   polyspace-bug-finder myResults
   ```

See Also

`polyspace-bug-finder-nodesktop` | `polyspace-code-prover-nodesktop` | `polyspace-configure`

More About

- “Create Command-Line Script from Project File” on page 2-8

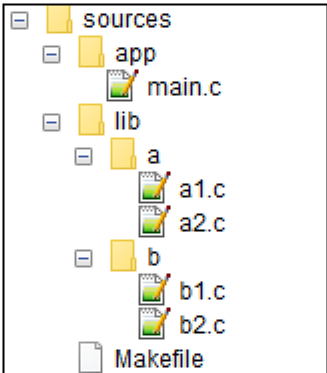
External Websites

- Set up Continuous Code Verification with Jenkins

polyspace-configure Source Files Selection Syntax

When you create projects by using `polyspace-configure`, you can include or exclude source files whose paths match the pattern that you pass to the options `-include-sources` or `-exclude-sources`. You can specify these two options multiple times and combine them at the command line.

This folder structure applies to these examples.



To try these examples, use the demo files in *`matlabroot`\help\toolbox\bugfinder\examples\sources-select*. *`matlabroot`* is your MATLAB installation folder.

Run this command:

```
polyspace-configure -allow-overwrite -include-sources glob_pattern \
-print-excluded-sources -print-included-sources make -B
```

`glob_pattern` is the glob pattern that you use to match the paths of the files you want to include or exclude from your project. In the table, the examples assume that `sources` is a top-level folder.

Glob Pattern Syntax	Example
No special characters, slashes ('/'), or backslashes ('\'). Pattern matches corresponding files, but not folders.	`-include-sources 'main.c'` matches: `/sources/app/main.c`

Glob Pattern Syntax	Example

Pattern contains `'*'` or `'?'` special characters.
 	
`'*'` matches zero or more characters in file or folder name.
 	
`'?'` matches one character in file or folder name.
 	
The matches do not include path separators.
	`-include-sources 'b?.c'` matches:
	
	`/sources/lib/b/b1.c`
	
	`/sources/lib/b/b2.c`
	
	`-include-sources 'app/*.c'` matches:
	
	`/sources/app/main.c`

Pattern starts with slash `'/'` (UNIX) or drive letter (Windows). Pattern matches absolute path only.	`-include-sources '/a'` does not match anything. `-include-sources '/sources/app'` matches: `/sources/app/main.c`

Pattern ends with a slash (UNIX), backslash (Windows), or `'**'`.
 	
Pattern matches all files under specified folder.
 	
`'**'` is ignored if it is at the start of the pattern.
	`-include-sources 'a/'` matches
	
	`/sources/lib/a/a1.c`
	
	`/sources/lib/a/a2.c`

Pattern contains `'/**/'` (UNIX) or `'**\'` (Windows). Pattern matches zero or more folders in the specified path.	`-include-sources 'lib/**/?1.c'` matches:
	
	`/sources/lib/a/a1.c`
	
	`/sources/lib/b/b1.c`

Glob Pattern Syntax	Example

Pattern starts with `'.'` or `'..'`.
 	
Pattern matches paths relative to the path where you run the command.
	If you start `polyspace-configure` from `/sources/lib/a`,
	
	`-include-sources '../lib/**/b?.c'` matches:
	
	`/sources/lib/b/b1.c`
	
	`/sources/lib/b/b2.c`

Pattern is a UNC path on Windows .	If your files are on server `myServer`:
	
	`\\myServer\sources\lib\b**` matches:
	
	`\\myServer\sources\lib\b\b1.c`
	
	`\\myServer\sources\lib\b\b2.c`

`polyspace-configure` does not support these glob patterns:

- Absolute paths relative to the current drive on Windows.

  For instance, `\foo\bar`.
- Relative paths to the current folder.

  For instance, `C:foo\bar`.
- Extended length paths in Windows.

  For instance, `\\?\foo`.
- Paths that contain '.' or '..' except at the start of the pattern.

  For instance, `/foo/bar/../a?.c`.
- The `'*'` character by itself.

Create Command-Line Script from Project File

In this section...
“Generate Scripting Files” on page 2-8
“Run an Analysis” on page 2-9

This example shows how to use a project file that you configured in the Polyspace interface to generate the necessary information to run from the command line. If you have already spent time configuring your project in the Polyspace interface, this command is useful to extract your setup work for scripting.

Generate Scripting Files

Generate a script from the demo Polyspace project, **Code_Prover_Example.psprj**.

1. In the Polyspace interface, open the example project by selecting **Help > Examples > Bug_Finder_ Example.psprj**.

   This example has been set up and configured with analysis options.
2. Open a command-line terminal and navigate to your `Polyspace_Workspace` folder. By default it is:

   - Linux — `/home/USER/Polyspace_Workspace`
   - Windows — `Users\USER\Documents\Polyspace_Workspace`
   - Mac — `USER/Polyspace_Workspace`
3. Navigate down to the example project:

   ```
   cd Examples/R2017b/Bug_Finder_Example
   ```
4. Run the script generation command .

   ```
   matlabroot/polyspace/bin/polyspace ...
       -generate-launching-script-for Bug_Finder_Example.psprj -bug-finder
   ```

   Polyspace generates a folder called `Bug_Finder_Example` containing:

   - `source_command.txt` — List of source files
   - `options_command.txt` — List of the analysis options

- `launchingCommand.sh` (UNIX) or `launchingCommand.bat` (DOS) — Shell script that calls the correct commands

For more details about what files are generated and how to use them, see `-generate-launching-script-for`.

Run an Analysis

After you have completed, "Generate Scripting Files" on page 2-8, you can use the files to run an analysis from the command line. The launching script makes integrating into continuous integration tools such as Jenkins, easier. Here are a few examples of how to use the generated files to run an analysis.

- Run the generated script locally by using the `launchingCommand.bat` file.

  ```
  Bug_Finder_Example\launchingCommand.bat
  ```
- Run the generated script and change the results folder.

  ```
  Bug_Finder_Example\launchingCommand.bat -results-dir Results_BF_Example_mine
  ```

  The extra `-results-dir` option overrides the results folder specified in the `options_command.txt` file.
- Send the analysis to a remote server and store the results in Polyspace Metrics.

  ```
  Bug_Finder_Example\launchingCommand.bat ...
      -add-to-results-repository -batch -scheduler MJS@NoteHost
  ```
- Run the analysis from the command line with the `-options-file` option.

  ```
  matlabroot/polyspace/bin/polyspace-bug-finder-nodesktop -options-file ...
      Bug_Finder_Example\options_command.txt
  ```

See Also

`-generate-launching-script-for`

Related Examples

- "Run Polyspace Analysis from Command Line" on page 2-2

External Websites

- Set up Continuous Code Verification with Jenkins

3

Run Polyspace Analysis with MATLAB Scripts

- “Run Polyspace Analysis by Using MATLAB Scripts” on page 3-2
- “Generate MATLAB Scripts from Polyspace User Interface” on page 3-6
- “Troubleshoot Polyspace Analysis from MATLAB” on page 3-9

Run Polyspace Analysis by Using MATLAB Scripts

You can automate the analysis of your C/C++ code by using MATLAB scripts. In your script, you specify your source files and analysis options such as compiler, run an analysis, and read the analysis results to MATLAB tables.

For instance, use this script to run a Polyspace Bug Finder analysis on a sample file:

```
proj = polyspace.Project

% Specify sources and includes
sourceFile = fullfile(matlabroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c');
includeFolder = fullfile(matlabroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');

% Configure analysis
proj.Configuration.Sources = {sourceFile};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.EnvironmentSettings.IncludeFolders = {includeFolder};
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = proj.run('bugFinder');

% Read results
bfSummary = proj.Results.getSummary('defects');
```

See also `polyspace.Project`.

Specify Multiple Source Files

You can specify a folder containing all your source files. For instance:

```
sourceFolder = fullfile(matlabroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');
proj.Configuration.Sources = {fullfile(sourceFolder,'*')};
```

You can also specify multiple source folders in the cell array.

You can specify a folder that contains all your source files directly *or in subfolders*. For instance:

```
sourceFolder = fullfile(matlabroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');
proj.Configuration.Sources = {fullfile(sourceFolder,'**')};
```

If you do not want to analyze all files in a folder, you can explicitly specify which files to analyze. For instance:

```
sourceFolder = fullfile(matlabroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');
file1 = fullfile(sourceFolder,'numerical.c');
file2 = fullfile(sourceFolder,'staticmemory.c');
proj.Configuration.Sources = {file1, file2};
```

You can explicitly exclude files from analysis. For instance:

```
% Specify source folder.
sourceFolder = fullfile(matlabroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');
proj.Configuration.Sources = {fullfile(sourceFolder,'**')};

% Specify files to exclude.
file1 = fullfile(sourceFolder,'security.c');
file2 = fullfile(sourceFolder,'tainteddata.c');
proj.Configuration.InputsStubbing.DoNotGenerateResultsFor = ['custom=' file1 ...
     ',' file2];
```

However, this method of exclusion does not apply to Code Prover run-time error checking.

Check for MISRA C:2012 Violations

You can customize the Polyspace analysis to check for MISRA C:2012 rule violations.

Set options for checking MISRA C:2012 rules. Disable the regular Bug Finder analysis, which looks for defects.

```
% Enable MISRA C checking
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = 'mandatory';

% Disable defect checking
proj.Configuration.BugFinderAnalysis.EnableCheckers = false;

% Run analysis
bfStatus = proj.run('bugFinder');
```

```
% Read summary of results
misraSummary = proj.Results.getSummary('misraC2012');
```

Check for Specific Defects or Coding Rule Violations

Instead of the default set of defect or coding rule checkers, you can specify your own set.

To disable MISRA C:2012 rules 8.1 to 8.4:

```
% Disable rules
misraRules = polyspace.CodingRulesOptions('misraC2012');

misraRules.rule_8_1 = false;
misraRules.rule_8_2 = false;
misraRules.rule_8_3 = false;
misraRules.rule_8_4 = false;

% Configure analysis
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = misraRules;
```

See also `polyspace.CodingRulesOptions`.

To enable Bug Finder defects, use the class `polyspace.DefectsOptions`. One difference between coding rules and defects class is that coding rule checkers are enabled by default. You disable the ones that you do not want. All defect checkers are disabled by default. You enable the ones that you want.

Find Files That Do Not Compile

If one or more of your files contain a compilation error, the analysis continues with the remaining files. You can choose to stop analysis on compilation errors.

```
proj.Configuration.EnvironmentSettings.StopWithCompileError = true;
```

However, it is more convenient to let the analysis complete and capture all compilation errors from the analysis log file. For more information, see “Troubleshoot Polyspace Analysis from MATLAB” on page 3-9.

Run Analysis on Cluster

You can run an analysis on a cluster instead of your local desktop. Once you have set up connection to a server, you can run the analysis in batch mode. For setup information, see "Set Up Server for Metrics and Remote Analysis".

Specify that the analysis must run on a server. Specify a folder on your desktop where results are downloaded after analysis.

```
proj.Configuration.MergedComputingSettings.BatchBugFinder = true;
proj.Configuration.ResultsDir = fullfile(pwd,'results');
```

Run analysis as usual.

```
proj.run('bugFinder');
```

Open the results from the results folder location.

```
pslinkfun('openresults', '-resultsfolder', proj.Configuration.ResultsDir);
```

If the analysis is complete and the results have been downloaded, they open in the Polyspace user interface.

See Also

polyspace.Project | polyspaceBugFinder

Related Examples

- "Generate MATLAB Scripts from Polyspace User Interface" on page 3-6
- "Visualize Bug Finder Analysis Results in MATLAB" on page 17-10
- "Troubleshoot Polyspace Analysis from MATLAB" on page 3-9

Generate MATLAB Scripts from Polyspace User Interface

You can specify analysis options in the Polyspace user interface and later generate a MATLAB script for easier reuse of those options.

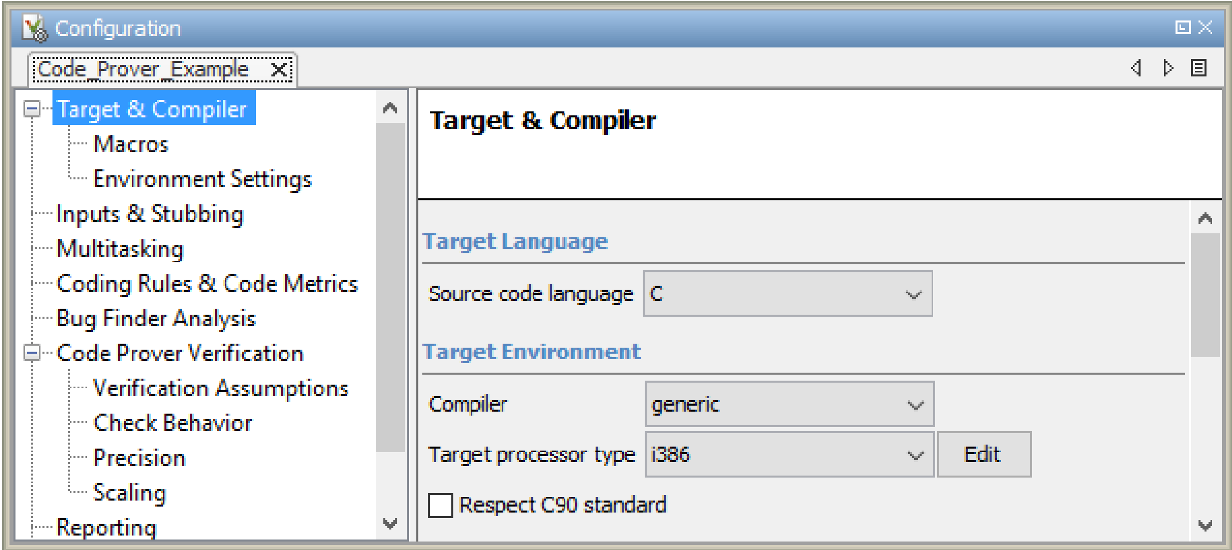
In the user interface, to determine which options to specify, you have tooltips, autocompletion of function names, compilation assistant, context-sensitive help and so on. After you specify the options, you can generate a MATLAB script. For subsequent analyses, you can modify and run the script without opening the Polyspace user interface.

To start an analysis in the Polyspace user interface, create a project. In the project:

- You specify source and include folders during project creation.
- You specify analysis options such as compiler or multitasking in your project configuration. You also enable or disable checkers.

From this project, you can generate a script that contains your sources, includes and other analysis options. To begin, select **File > New Project**. For details, see “Add Source Files for Analysis in Polyspace User Interface” on page 1-2.

This example uses a sample project. To open the project, select **Help > Examples > Code_Prover_Example.psprj**. You see the options in the project configuration. For instance, on the **Target & Compiler** node, you see a `generic` compiler and an `i386` processor.



1. Open MATLAB.

   For instance, select **Tools > Open MATLAB**.
2. Create a `polyspace.Options` object from the sample Polyspace project.

   ```
   projectFile = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', ...
             'Code_Prover_Example', 'Code_Prover_Example.psprj');
   opts = polyspace.loadProject(projectFile);
   ```
3. Append the object to a MATLAB script.

   ```
   filePath = opts.toScript('runPolyspace.m','append');
   ```

   Open the script `runPolyspace.m`. You see the options that you specified from the user interface. For instance, you see the compiler and target processor.

   ```
   opts.TargetCompiler.Compiler = 'generic';
   opts.TargetCompiler.Target = 'i386';
   ```

   Later, you can run the script to create a `polyspace.Options` object.

   ```
   run(filePath);
   ```

The preceding example converts the sample project `Code_Prover_Example` directly to a script. When you open the sample project in the user interface, a copy is loaded into your

Polyspace workspace. If you make changes to the sample project, the changes are made to the copied version. To see the changes in your MATLAB script, provide the copied project path to the `loadProject` method. To see the location of your workspace, select **Tools** > **Preferences** and view the **Project and Results Folder** tab.

See Also

Related Examples

- "Run Polyspace Analysis by Using MATLAB Scripts" on page 3-2

Troubleshoot Polyspace Analysis from MATLAB

When you run a Polyspace analysis on your C/C++ code, if one or more of your files fail to compile, the analysis continues with the remaining files. You can choose to stop the analysis on compilation errors.

```
proj = polyspace.Project;
proj.Configuration.EnvironmentSettings.StopWithCompileError = true;
```

However, it is more convenient to let the analysis complete and capture all compilation errors.

The compilation errors are displayed in the analysis log that appears on the MATLAB command window. The analysis log also contains the options used and the various stages of analysis. The lines that indicate errors begin with the `Error:` string. Find these lines and extract them to a log file for easier scanning. Produce a warning to indicate that compilation errors occurred.

The function `runPolyspace` defined later captures the output from the command window using the `evalc` function and stores lines starting with `Error:` in a file `error.log`. You can call `runPolyspace` with paths to your source and include folders.

```
[status, resultsSummary] = runPolyspace('/path/to/sources', '/path/to/includes');
```

The function is defined as follows.

```
function [status, resultsSummary] = runPolyspace(sourcePath, libPath)
% runPolyspace takes two string arguments: source and include folder.
% The files in the source folder are analyzed for defects.
% If one or more files fail to compile, the errors are saved in a log.
% A warning on the screen indicates that compilation errors occurred.

    proj = polyspace.Project;

    % Specify sources
    proj.Configuration.Sources = {fullfile(sourcePath,'*')};

    % Specify compiler and paths to libraries
    proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
    proj.Configuration.EnvironmentSettings.IncludeFolders = {fullfile(libPath,'*')};


    % Run analysis
```

```
runMode = 'bugFinder';
[logFileContent,status] = evalc('proj.run(runMode)');

% Open file for writing errors
errorFile = fopen('error.log','wt+');

% Check log file for compilation errors
numErrors = 0;

log = strsplit(logFileContent,'\n');
errorLines = find(contains(log, {'Error:'}, 'IgnoreCase', true));
for ii=1:numel(errorLines)
      fprintf(errorFile, '%s\n', log{errorLines(ii)});
      numErrors = numErrors + 1;
end

if numErrors
    warning('%d compilation error(s). See error.log for details.', numErrors);
end

fclose(errorFile);

% Read results
resultsSummary = proj.Results.getSummary('defects');
```

The analysis log is also captured in a file `Polyspace_R20##`*`n_ProjectName_date-time`*`.log`. Instead of capturing the output from the command window, you can search this file.

You can adapt this script for other purposes. For instance, you can capture warnings in addition to errors. The lines with warnings begin with `warning:`. The warnings indicate situations where the analysis proceeds despite an issue. The analysis makes an assumption to work around the issue. If the assumption is incorrect, you can see errors later or in rare cases, incorrect analysis results.

See Also

`polyspace.Project`

Related Examples

- “Run Polyspace Analysis by Using MATLAB Scripts” on page 3-2
- “Troubleshooting in Polyspace Bug Finder”

Run Polyspace Analysis on Remote Clusters

- "Run Polyspace Analysis on Remote Clusters" on page 4-2
- "Run Polyspace Analysis on Remote Clusters Using Scripts" on page 4-4

Run Polyspace Analysis on Remote Clusters

Before running a batch analysis in the Polyspace user interface, you must set up your project's source files, analysis options, and remote analysis settings. If you have not done so, see:

- "Add Source Files for Analysis in Polyspace User Interface" on page 1-2
- "Set Up Server for Metrics and Remote Analysis"

To start a remote analysis:

1. Select a project to analyze.
2. On the **Configuration** pane, select **Run Settings**.
3. Select **Run Bug Finder analysis on a remote cluster**.
4. If you want to store your results in the Polyspace Metrics repository, select **Upload results to Polyspace Metrics**.

   Otherwise, clear this check box.
5. Select the button.
6. To monitor the analysis, select **Tools > Open Job Monitor**. In the Polyspace Job Monitor, follow your queued job to monitor progress.

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

   If the analysis stops after compilation and you have to restart the analysis, to avoid restarting from the compilation phase, use the option `-submit-job-from-previous-compilation-results`.

Note If you choose to upload results to Polyspace Metrics, your results are not downloaded automatically after verification. Use the Polyspace Metrics web dashboard to view the results and download them to your desktop. For more information, see "View Projects in Polyspace Metrics" on page 18-5.

See Also

More About

- “Set Up Server for Metrics and Remote Analysis”
- “Run Polyspace Analysis on Remote Clusters Using Scripts” on page 4-4

Run Polyspace Analysis on Remote Clusters Using Scripts

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”.

Run Remote Analysis

Use the following command to run a remote analysis:

```
matlabroot\polyspace\bin\polyspace-bug-finder-nodesktop
-batch -scheduler NodeHost | MJSName@NodeHost [options]
```

where:

- `matlabroot` 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.

  If you set up communications with a cluster from the Polyspace user interface, you can determine `NodeHost` and `MJSName` from the user interface. Select **Metrics > Metrics and Remote Server Settings**. Open the MATLAB Distributed Computing Server Admin Center. Under **MATLAB Job Scheduler**, see the **Name** and **Hostname** columns for `MJSName` and `NodeHost`. For details, see “Configure for an MJS” (MATLAB Distributed Computing Server).
- `options` are the analysis options. These options are the same as that of a local analysis. For instance, you specify the results folder using the option `-results-dir`.

  For more information, see “Run Polyspace Analysis from Command Line” on page 2-2.

After compilation, the software submits the analysis job to the cluster and provides you a job ID. You can also read the ID from the file `ID.txt` in the results folder. Use the `polyspace-jobs-manager` command with the job ID to monitor your analysis and download results after analysis is complete. For more information, see “Manage Remote Analysis” on page 4-6.

If the analysis stops after compilation and you have to restart the analysis, to avoid restarting from the compilation phase, use the option `-submit-job-from-previous-compilation-results`.

Tip In Windows, to avoid typing the commands each time, you can save the commands in a batch file. In Linux, you can relaunch the analysis using a `.sh` file.

1. Save your analysis options in a file `listofoptions.txt`. See “Specify Sources and Analysis Options in Text File” on page 2-3.

   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=matlabroot\polyspace\bin
   set RESULTS_PATH=C:\Results
   set OPTIONS_FILE=C:\Options\listofoptions.txt
   "%POLYSPACE_PATH%\polyspace-bug-finder-nodesktop.exe" -batch -scheduler localhost
                        -results-dir "%RESULTS_PATH%" -options-file "%OPTIONS_FILE%"
   pause
   ```

   Where `matlabroot` is your MATLAB installation folder, and *`localhost`* is the name of the computer that hosts the head node of your MATLAB Distributed Computing Server cluster.
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 `listofoptions.txt`.
5. Double-click `launcher.bat` to run the analysis.

If you run a Polyspace analysis, a Windows `.bat` or Linux `.sh` file is automatically generated for you. The file is in the `.settings` subfolder in your results folder. You can relaunch the analysis using this file.

Manage Remote Analysis

To manage remote analyses, use this command:

```
matlabroot\polyspace\bin\polyspace-jobs-manager action [options]
          [-scheduler schedulerOption]
```

where:

- *matlabroot* 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 “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox)

  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
`listjobs`	None	Generate a list of Polyspace jobs on the scheduler. For each job, the software produces the following information: • `ID` — Verification or analysis identifier. • `AUTHOR` — Name of user that submitted job. • `APPLICATION` — Name of Polyspace product, for example, Polyspace Code Prover™ or Polyspace Bug Finder. • `LOCAL_RESULTS_DIR` — Results folder on local computer, specified through the **Tools** > **Preferences** > **Server Configuration** tab. • `WORKER` — Local computer from which job was submitted. • `STATUS` — Status of job, for example, `running` and `completed`. • `DATE` — Date on which job was submitted. • `LANG` — Language of submitted source code.

Action	Options	Task
`download`	`-job` *`ID`* `-results-folder` *`FolderPath`*	Download results of analysis with specified ID to folder specified by *`FolderPath`*. When the analysis job is queued on the server, the command `polyspace-bug-finder-nodesktop` returns a job id. Additionally, a file `ID.txt` in the results folder contains the job id in this format: *`job_id`*`;`*`server_name`*`:`*`project_name version_number`* For instance, `92;localhost:Demo 1.0`. If you do not use the `-results-folder` option, the software downloads the result to the folder you specified when starting analysis, using the `-results-dir` option. After downloading results, use the Polyspace user interface to view the results. See "Interpret Polyspace Bug Finder Results" on page 14-2.
`getlog`	`-job` *`ID`*	Open log for job with specified ID.
`remove`	`-job` *`ID`*	Remove job with specified ID.
`promote`	`-job` *`ID`*	Promote job with specified ID in the queue.
`demote`	`-job` *`ID`*	Demote job with specified ID in the queue.

See Also

More About

- "Set Up Server for Metrics and Remote Analysis"
- "Run Polyspace Analysis on Remote Clusters" on page 4-2

External Websites

- Set up Continuous Code Verification with Jenkins

Run Polyspace Analysis on Generated Code

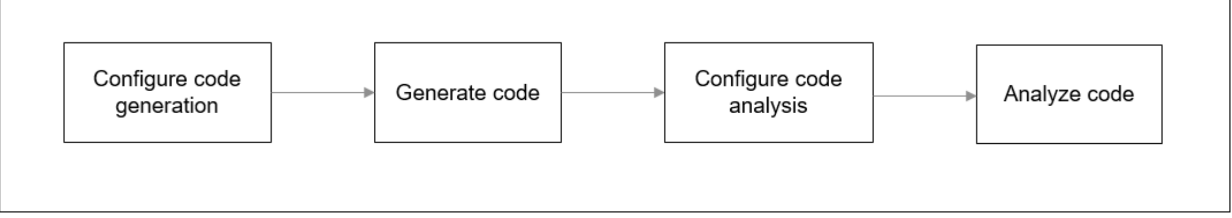
Run Polyspace Analysis on Code Generated with Embedded Coder

If you generate code from a Simulink model using Embedded Coder® or TargetLink®, you can analyze the generated code for bugs or run-time errors with Polyspace from within the Simulink environment. You do not have to manually set up a Polyspace project.

This topic uses Embedded Coder for code generation. For analysis of TargetLink-generated code, see “Run Polyspace Analysis on Code Generated with TargetLink” on page 5-32.

For a tutorial with a specific model, see “Analyze Code Generated from Simulink Subsystem” on page 5-10.

Generate and Analyze Code



Configure Code Generation and Generate Code

To configure code generation and generate code from a model or subsystem, do one of the following:

- Select **Code > C/C++ > Embedded Coder Quick Start**. Follow the on-screen instructions.
- Configure code generation through Simulink configuration parameters. The chief parameters to set are:
  - Type (Simulink): Select **Fixed-step**.

- Solver (Simulink): Select **auto (Automatic solver selection)** or **Discrete (no continuous states)**.
- System target file (Simulink Coder): Enter `ert.tlc` or `autosar.tlc`. If you derive target files from `ert.tlc`, you can also specify them.
- “Code-to-model” (Simulink Coder): Select this option to enable links from code to model.

For the full list of parameters to set, see “Recommended Model Configuration Parameters for Polyspace Analysis” on page 5-20.

Alternatively, run the Code Generation Advisor with the objective **Polyspace** and check if the required parameters are already set. See “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder).

To generate code, select **Code** > **C/C++** > **Build Model**. There is an equivalent option for a subsystem.

Configure Code Analysis

Select **Code** > **Polyspace** > **Options**. Change default values of these options if needed.

- “Product mode”: Choose Code Prover or Bug Finder.
- Settings from: Enable checking of MISRA® coding rules in addition to the default checks specified in the project configuration. The default Bug Finder checks look for bugs and the Code Prover checks look for run-time errors.
- “Input”, “Tunable parameters” and “Output”: Constrain inputs, tunable parameters or outputs for a more precise Code Prover analysis.
- “Output folder”: Specify a dedicated folder for results. The default analysis saves the results in a folder `results_`*`modelName`* in the current working folder.
- “Open results automatically after verification”

Analyze Code

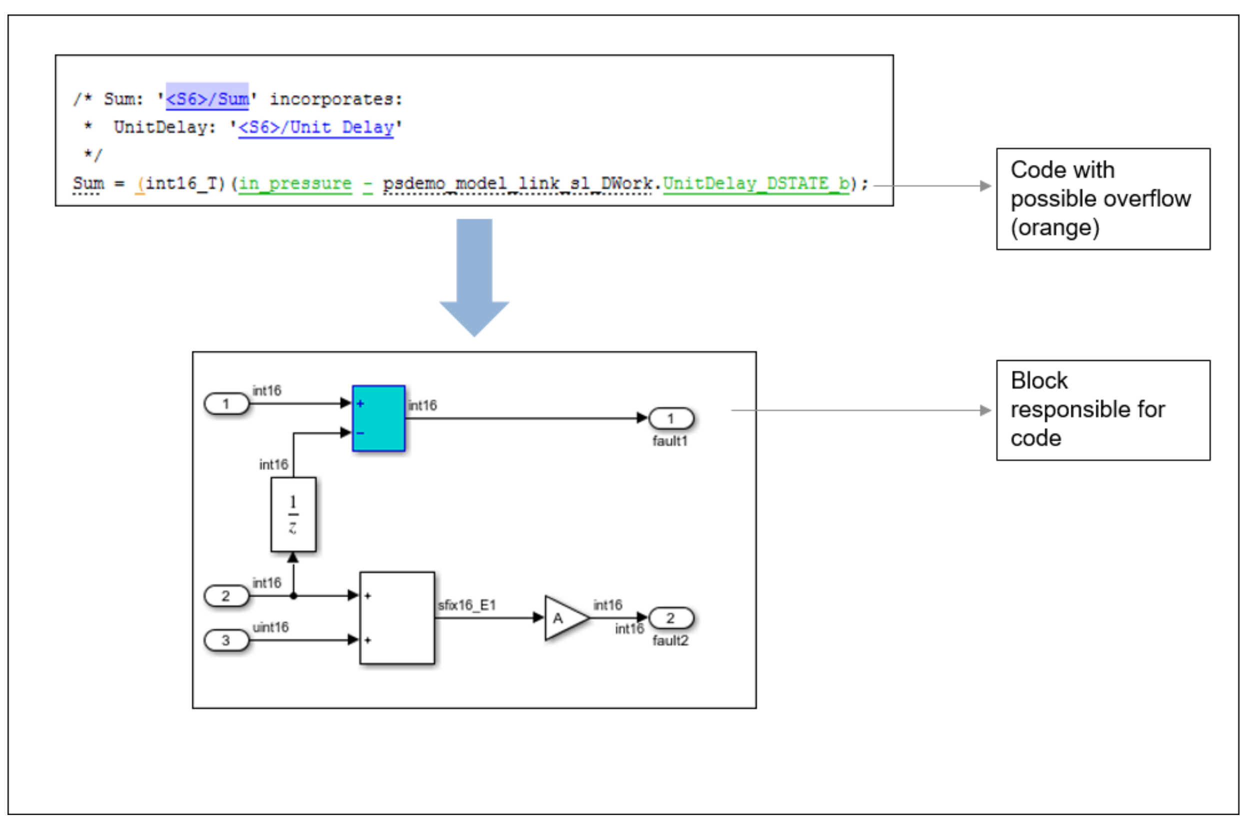
To analyze the code, select **Code** > **Polyspace** > **Verify Code Generated for** > **Model**. There is an equivalent option for a subsystem.

You can follow the progress of the analysis in the MATLAB command window.

The results open automatically unless explicitly disabled. By default, the results are saved in a folder `results_ModelName` in the current folder. Each new run overwrites previous results. You can change these behaviors or save the results to a Simulink project. To make these changes, select **Code** > **Polyspace** > **Options**.

If you want to open the results later, select **Code** > **Polyspace** > **Open Results** > **For Generated Code**.

Review Analysis Results



Review Result in Code

The results appear in the Polyspace user interface on the **Results List** pane. Click each result to see the source code and details on the **Result Details** pane. See also:

- “Interpret Polyspace Bug Finder Results” on page 14-2
-
-
- “Address Polyspace Results Through Bug Fixes or Comments” on page 15-2
- “Filter and Group Results” on page 16-2

Navigate from Code to Model

Links in code comments show blocks that generate the subsequent lines of code. To see the blocks in the model, click the block names in the links. If you run into issues, see “Troubleshoot Navigation from Code to Model” on page 5-37.

Alternatively, you can right-click a variable name and select **Go to Model**. This option is not available for all variables.

Fix Issue

Investigate whether the issues in your code are related to design flaws in the model.

There can be many design flaws in the model that lead to issues in the generated code. For instance:

- The generated code might be free of specific run-time errors only for a certain range of a block parameter. To fix this, you can change the storage class of that block parameter or use calibration data for the analysis using the configuration parameter “Tunable parameters”.
- The generated code might be free of specific run-time errors only for a certain range of inputs. To verify this, you can specify a minimum and maximum value for the Inport block signals. The Polyspace analysis uses this constrained range. See “Specify Ranges for Signals” (Simulink).
- Certain transitions in Stateflow® charts can be unreachable.

If you include handwritten C/C++ code in S-Function blocks, the Polyspace analysis can reveal possible integration issues between the handwritten and generated code. You can also analyze the handwritten code in isolation. See “Analyze S-Function Code” on page 5-18.

Annotate Blocks to Justify Issue

If you do not want to make changes in response to a Polyspace result, annotate the relevant blocks. After you annotate a block, code operations generated from the block show results prepopulated with your comments. To annotate a block, right-click the block and select **Polyspace > Annotate Selected Block > Edit**. Enter the following:

- Comma-separated list of result acronyms. To justify only type of result, select **Only 1 check**.

  See:

  - “Short Names of Bug Finder Defect Checkers” on page 15-12
  - “Short Names of Code Prover Run-Time Checks” (Polyspace Code Prover)
- Status, severity and comment to be assigned to the results.

Sometimes operations in the generated code are known to cause orange checks in Code Prover. Suppose an operation is known to possibly overflow. The generated code protects against the overflow by following the operation with a saturation. Polyspace still flags the possible overflow as an orange check. To automatically justify these checks through code comments, specify the configuration parameter “Operator annotations” (Simulink Coder).

See Also

More About

- “Configure Advanced Polyspace Options in Simulink” on page 5-23

Analyze Generated Code Using Polyspace Bug Finder

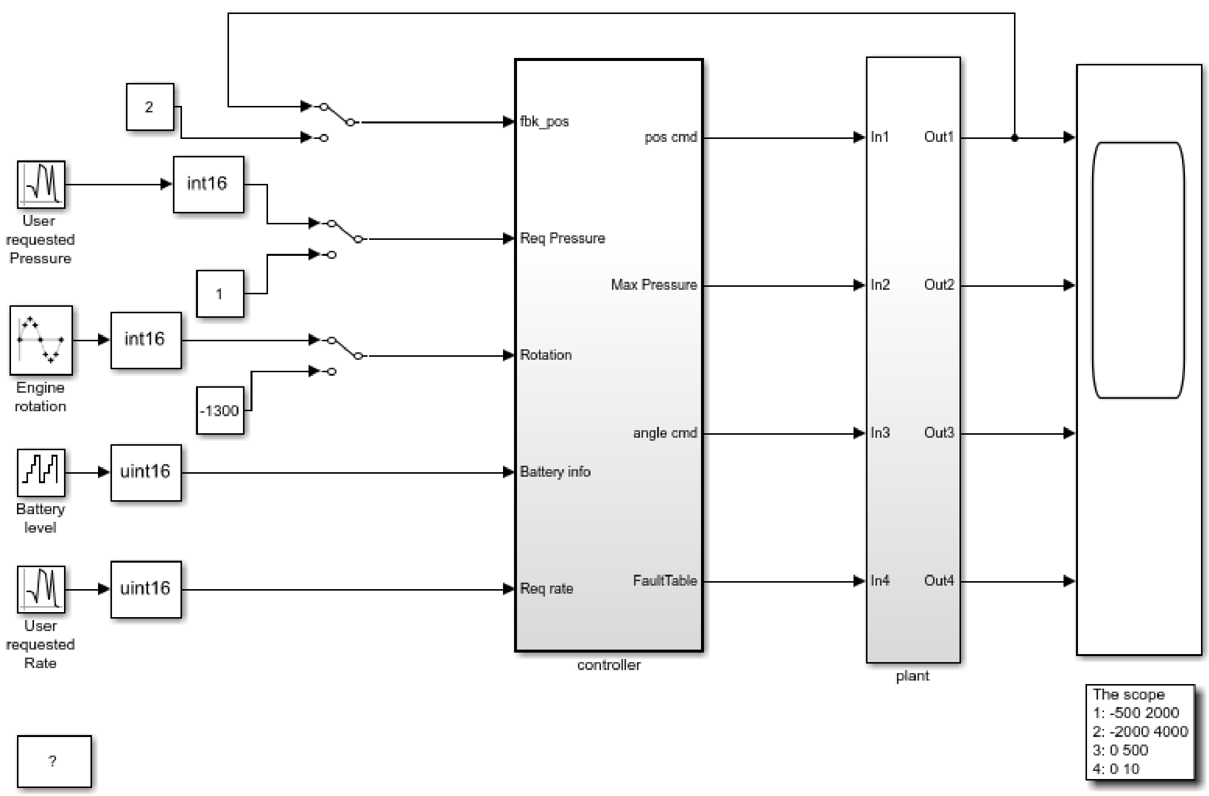
If you generate C or C++ code from models using Embedded Coder, you can check the generated code for run-time errors. Polyspace Bug Finder identifies run-time errors, concurrency issues, security vulnerabilities, and other defects in C and C++ embedded software..

This example contains a demo model from which you can generate code and then analyze the generated code.

Open Model

Open and explore the example model. The model contains a `controller` subsystem, which itself contains many subsystems. One of the subsystems has some issues that can lead to run-time errors in the generated code.

```
open_system('psdemo_model_link_sl');
```



Generate and Analyze Code

Generate code from the `controller` subsystem or one of the subsystems underneath. Then, run Polyspace Bug Finder on the generated code. You can trace back from defects found in the generated code to corresponding blocks in the model. You can also check for coding rule violations and add annotations on blocks to justify the violations. For details, see Analyze Code Generated from Simulink Subsystem.

The `controller` subsystem also contains an S-Function block. You can separately analyze the C code that the S-Function block refers to. For details, see Analyze S-Function Code.

Analyze Code Generated from Simulink Subsystem

You can run Polyspace on the code generated from a Simulink model or subsystem.

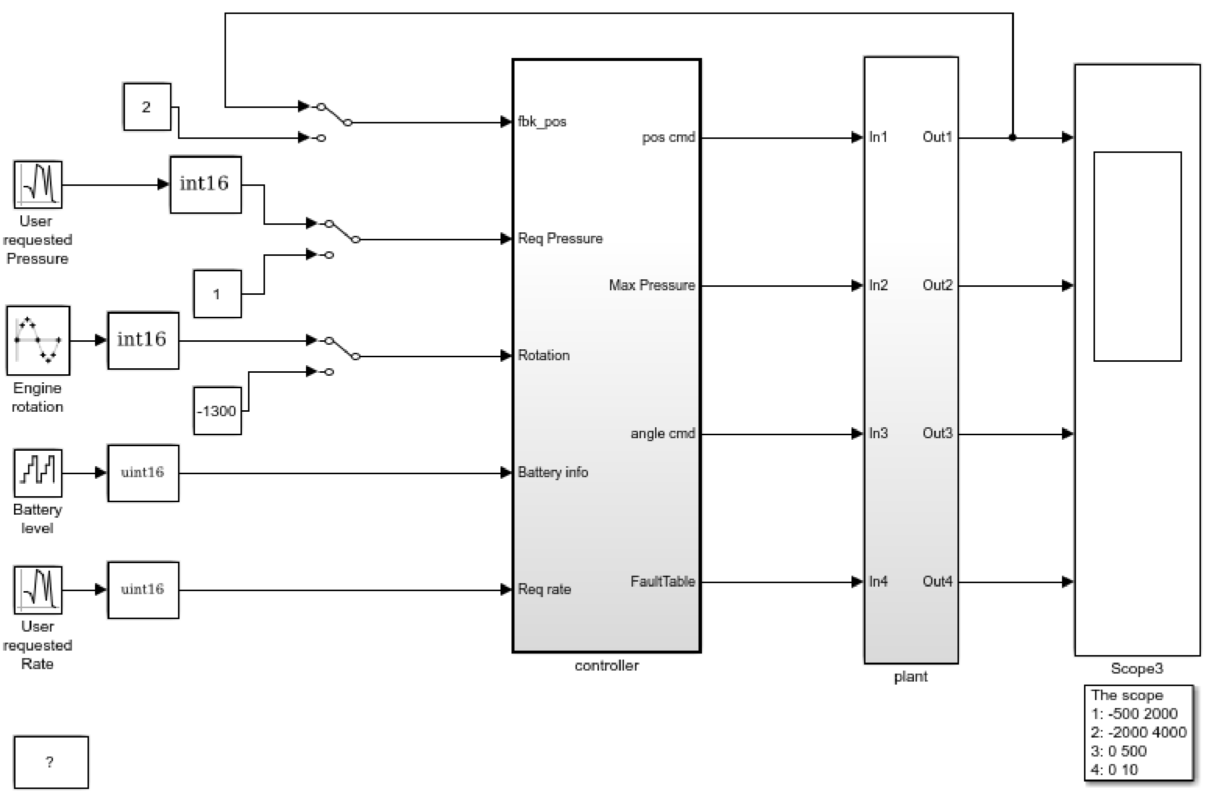
- Polyspace Bug Finder checks the code for bugs or coding rule violations (for instance, MISRA C: 2012 rules).
- Polyspace Code Prover exhaustively checks the code for run-time errors.

If you use Embedded Coder for code generation, this tutorial shows how to run Polyspace on the generated code from within Simulink.

Open Model

Open the example model.

```
modelName = 'psdemo_model_link_sl';
open_system(modelName)
```



Generate Code

Generate code for the `controller` subsystem in your model.

1. Right-click the `controller` subsystem and select **C/C++ Code > Build This Subsystem**.
2. In the dialog box, select **Build**.

Equivalent MATLAB Code:

```
subsysPath = 'psdemo_model_link_sl/controller';
rtwbuild(subsysPath);
```

Analyze Code

Analyze the code generated for the `controller` subsystem.

1. Choose a product, Bug Finder or Code Prover, to analyze the code.

   Right-click the `controller` subsystem and select **Polyspace** > **Options**. For **Product mode**, choose `Code Prover` or `Bug Finder`.
2. Analyze the generated code.

   Right-click the `controller` subsystem and select **Polyspace** > **Verify Generated Code for** > **Selected Subsystem**. Follow the progress of analysis in the MATLAB Command Window.

Equivalent MATLAB Code:

```
opts = polyspace.ModelLinkOptions('C');
mlopts = pslinkoptions(subsysPath);
mlopts.VerificationMode = 'CodeProver';
mlopts.PrjConfigFile = generateProject(opts, 'polyspaceProject');
pslinkrun(subsysPath, mlopts);
```

To analyze with Bug Finder, replace `CodeProver` with `BugFinder`. For more information on the code, see `polyspace.ModelLinkOptions`, `pslinkoptions` and `pslinkrun`.

Review Analysis Results

After analysis, the results are displayed in the Polyspace user interface.

If you run Bug Finder, the results consist of bugs detected in the generated code. If you run Code Prover, the results consist of checks that are color-coded as follows:

- **Green (proven code)**: The check does not fail for the data constraints provided. For instance, a division operation does not cause a **Division by Zero** error.
- **Red (verified error)**: The check always fails for the set of data constraints provided. For instance, a division operation always causes a **Division by Zero** error.

- **Orange (possible error)**: The check indicates unproven code and can fail for certain values of the data constraints provided. For instance, a division operation sometimes causes a **Division by Zero** error.
- **Gray (unreachable code)**: The check indicates a code operation that cannot be reached for the data constraints provided.

Review each analysis result in detail. For instance, in your Code Prover results:

1. On the **Results List** pane, select the red **Out of bounds array index** check.
2. On the **Source** pane, place your cursor on the red check to view additional information. For instance, the tooltip on the red `[` operator states the array size and possible values of the array index. The **Result Details** pane also provides this information.

The error occurs in a handwritten C file `Command_strategy_file.c`. The C file is inside an S-Function block `Command_Strategy` in the `controller` subsystem.

Trace Errors Back to Model and Fix Them

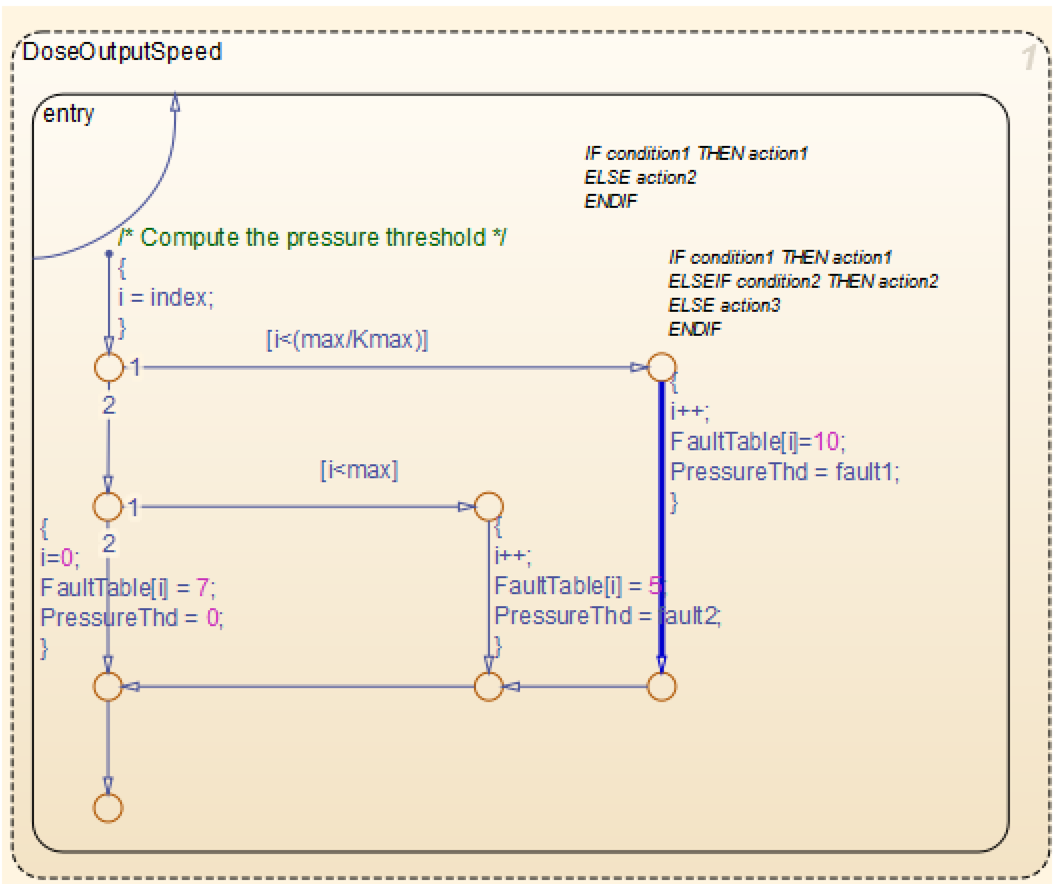
For code generated from the model, you can trace an error back to your model. These sections show how to trace specific Code Prover results back to the model.

Error 1: Out of bounds array index

1. On the **Results List** pane, select the orange **Out of bounds array index** error that occurs in the file `controller.c`.
2. On the **Source** pane, click the link **S5:76** in comments above the orange error.

```
/* Transition: '<S5>:75' */
/* Transition: '<S5>:76' */
(*i)++;

/* Outport: '<Root>/FaultTable' */
controller_Y.FaultTable[*i] = 10;
```

You see that the error occurs due to a transition in the Stateflow chart `synch_and_asynch_monitoring`. You can trace the error to the input variable index of the Stateflow chart.



You can avoid the **Out of bounds array index** in several ways. One way is to constrain the input variable `index` using a Saturation block before the Stateflow chart.

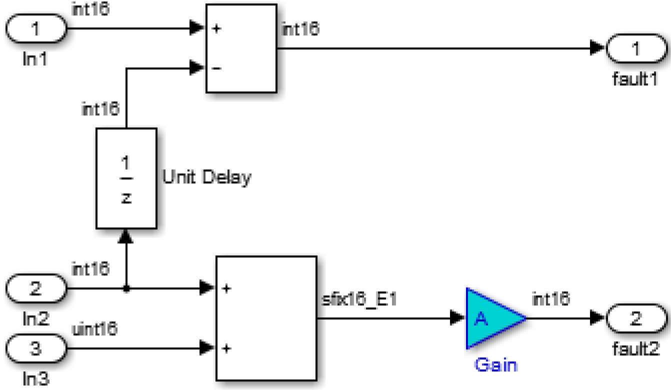
Error 2: Overflow

1. On the **Results List** pane, select the orange **Overflow** error shown below. The error appears in the file `controller.c`.
2. On the **Source** pane, review the error. To trace the error back to the model, click the link **S2/Gain** in comments above the orange error.

```
/* Gain: '<S2>/Gain' incorporates:
* Inport: '<Root>/Battery Info'
```

```
* Inport: '<Root>/Rotation'
* Sum: '<S2>/Sum1'
*/

Gain = (int16_T)(((int16_T)((in_rotation + in_battery_info) >> 1) * 24576) >>
                   10);
```

You see that the error occurs in the `Fault Management` subsystem inside a Gain block following a Sum block.



You can avoid the **Overflow** in several ways. One way is to constrain the value of the signal `in_battery_info` that is fed to the Sum block. To constrain the signal:

1. Double-click the Inport block `Battery info` that provides the input signal `in_battery_info` to the `controller` subsystem.
2. On the **Signal Attributes** tab, change the **Maximum** value of the signal.

The errors in this model occur due to one of the following:

- Faulty scaling, unknown calibrations and untested data ranges coming out of a subsystem into an arithmetic block.
- Array manipulation in Stateflow event-based modelling and handwritten lookup table functions.
- Saturations leading to unexpected data flow inside the generated code.
- Faulty Stateflow programming.

Once you identify the root cause of the error, you can modify the model appropriately to fix the issue.

Check for Coding Rule Violations

To check for coding rule violations, before starting code analysis:

1. Right-click the `controller` subsystem and select **Polyspace** > **Options**.
2. In the Configuration Parameters dialog box, select an appropriate option in the **Settings from** list. For instance, select `Project configuration and MISRA C 2012 AGC Checking`.

   It is recommended that you run Bug Finder for checking MISRA C:2012 rules. For **Product mode**, choose `Bug Finder`.
3. Click **Apply** or **OK** and rerun the analysis.

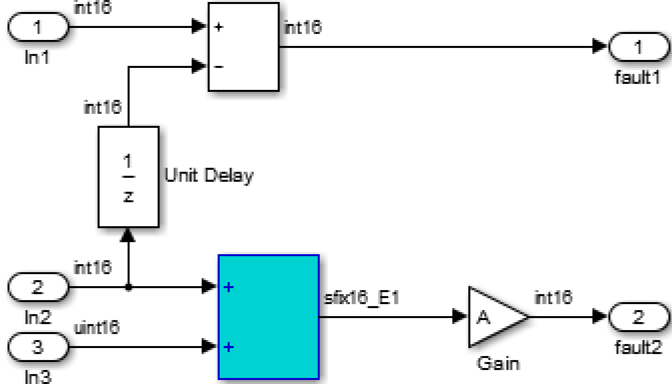
Annotate Blocks to Justify Results

You can justify your results by adding annotations to your blocks. During code analysis, Polyspace Code Prover reads your annotations and populates the result with your justification. Once you justify a result, you do not have to review it again.

1. On the **Results List** pane, from the drop-down list in the upper left corner, select **File**.
2. In the file `controller.c`, in the function `controller_step()`, select the violation of MISRA C:2012 rule 10.4. The **Source** pane shows that an addition operation violates the rule.
3. On the **Source** pane, click the link **S2/Sum1** in comments above the addition operation.

   ```
   /* Gain: '<S2>/Gain' incorporates:
   * Inport: '<Root>/Battery Info'
   * Inport: '<Root>/Rotation'
   * Sum: '<S2>/Sum1'
   */
   Gain = (int16_T)(((int16_T)((in_rotation + in_battery_info) >> 1) * 24576) >>
                      10);
   ```

   You see that the rule violation occurs in a Sum block.



To annotate this block and justify the rule violation:

a Right-click the block and select **Polyspace** > **Annotate Selected Block** > **Edit**.

b Select `MISRA-C-2012` for **Annotation type** and enter information about the rule violation. Set the **Status** to **No action planned** and the **Severity** to **Unset**.

c Click **Apply** or **OK**. The words **Polyspace annotation** appear below the block, indicating that the block contains a code annotation.

d Regenerate code and rerun the analysis. The **Severity** and **Status** columns on the **Results List** pane are prepopulated with your annotations.

See Also

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 5-2

Analyze S-Function Code

If you want to check your S-Function code for bugs or errors, you can run Polyspace directly from your S-Function block in Simulink.

S-Function Analysis Workflow

To verify an S-Function with Polyspace, follow this recommended workflow:

1. Compile your S-Function to be compatible with Polyspace.
2. Select your Polyspace options.
3. Run a Polyspace Bug Finder analysis using one of the two analysis modes:
   - **This Occurrence** — Analyzes the specified occurrence of the S-Function with the input for that block.
   - **All Occurrences** — Analyzes the S-Function code with input values from every occurrence of the S-Function.
4. Review results in the Polyspace interface.
   - For information about navigating through your results, see “Filter and Group Results” on page 16-2.
   - For help reviewing and understanding the results, see “Polyspace Bug Finder Results”.

Compile S-Functions to Be Compatible with Polyspace

Before you analyze your S-Function with Polyspace Bug Finder , you must compile your S-Function with one of following tools:

- The Legacy Code Tool with the `def.Options.supportCoverageAndDesignVerifier` set to `true`. See `legacy_code`.
- The SFunctionBuilder block, with **Enable support for Design Verifier** selected on the **Build Info** tab of the SFunctionBuilder dialog box.
- The Simulink Coverage™ function `slcovmex`, with the option `-sldv`.

Example S-Function Analysis

This example shows the workflow for analyzing S-Functions with Polyspace. You use the model `psdemo_model_link_sl` and the S-Function `Command_Strategy`.

1. Open the model and use the Legacy Code Tool to compile the S-Function `Command_Strategy`.

```
% Open Model
psdemo_model_link_sl

% Compile S-Function Command_Strategy
def = legacy_code('initialize');
def.SourceFiles = { 'command_strategy_file.c' };
def.HeaderFiles = { 'command_strategy_file.h' };
def.SFunctionName = 'Command_Strategy';
def.OutputFcnSpec = 'int16 y1 = command_strategy(uint16 u1, uint16 u2)';
def.IncPaths = { fullfile(matlabroot, ...
  'toolbox','polyspace','pslink','pslinkdemos','psdemo_model_link_sl') };
def.SrcPaths = def.IncPaths;
def.Options.supportCoverageAndDesignVerifier = true;
legacy_code('compile',def);
```
2. Open the subsystem `psdemo_model_link_sl/controller`.
3. Right-click the S-Function block Command_Strategy and select **Polyspace > Options**.
4. In the Configuration Parameters dialog box, make sure that the following parameters are set:
   - **Product mode** — `Bug Finder`
   - **Settings from** — `Project configuration and MISRA C 2012 checking`
   - **Open results automatically after verification** — ☑ On
5. Apply your settings and close the Configuration Parameters.
6. Right-click the Command_Strategy block and select **Polyspace > Verify S-Function > This Occurrence**.
7. Follow the analysis in the MATLAB Command Window. When the analysis is finished, your results open in the Polyspace interface.

Recommended Model Configuration Parameters for Polyspace 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 other parameters, if you do not use the recommended value, you get a warning.

Grouping	Command-Line	Name and Location in Configuration
Code Generation	Name: `SystemTargetFile (Simulink Coder)` Value: An Embedded Coder Target Language Compiler (TLC) file. For example `ert.tlc` or `autosar.tlc`.	Location: **Code Generation** Name: **System target file** Value: Embedded Coder target file
	Name: `MatFileLogging (Simulink Coder)` Value: `'off'`	Location: **Code Generation > Interface** Name: **MAT-file logging** Value: ☐ Not selected
	Name: `GenerateReport (Simulink Coder)` Value: `'on'`	Location: **Code Generation > Report** Name: **Create code-generation report** Value: ☑ Selected
	Name: `IncludeHyperlinksInReport (Simulink Coder)` Value: `'on'`	Location: **Code Generation > Report** Name: **Code-to-model** Value: ☑ Selected

Grouping	Command-Line	Name and Location in Configuration
	Name: `GenerateSampleERTMain (Embedded Coder)` Value: `'off'`	Location: **Code Generation > Templates** Name: **Generate an example main program** Value: ☐ Not selected
	Name: `GenerateComments (Simulink Coder)` Value: `'on'`	Location: **Code Generation > Comments** Name: **Include comments** Value: ☑ Selected
Optimization	Name: `DefaultParameterBehavior (Simulink Coder)` Value: `'Inlined'`	Location: **Optimization** Name: **Default parameter behavior** Value: `Inlined`
	Name: `InitFltsAndDblsToZero (Simulink Coder)` Value: `'on'`	Location: **Optimization** Name: **Use memset to initialize floats and doubles to 0.0** Value: ☐ Not selected
	Name: `ZeroExternalMemoryAtStartup (Simulink Coder)` Value: `'on'`	Location: **Optimization** Name: **Remove root level I/O zero initialization** Value: ☐ Not selected
Solver	Name: `SolverType (Simulink)` Value: `'Fixed-Step'`	Location: **Solver** Name: **Type** Value: `Fixed-step`

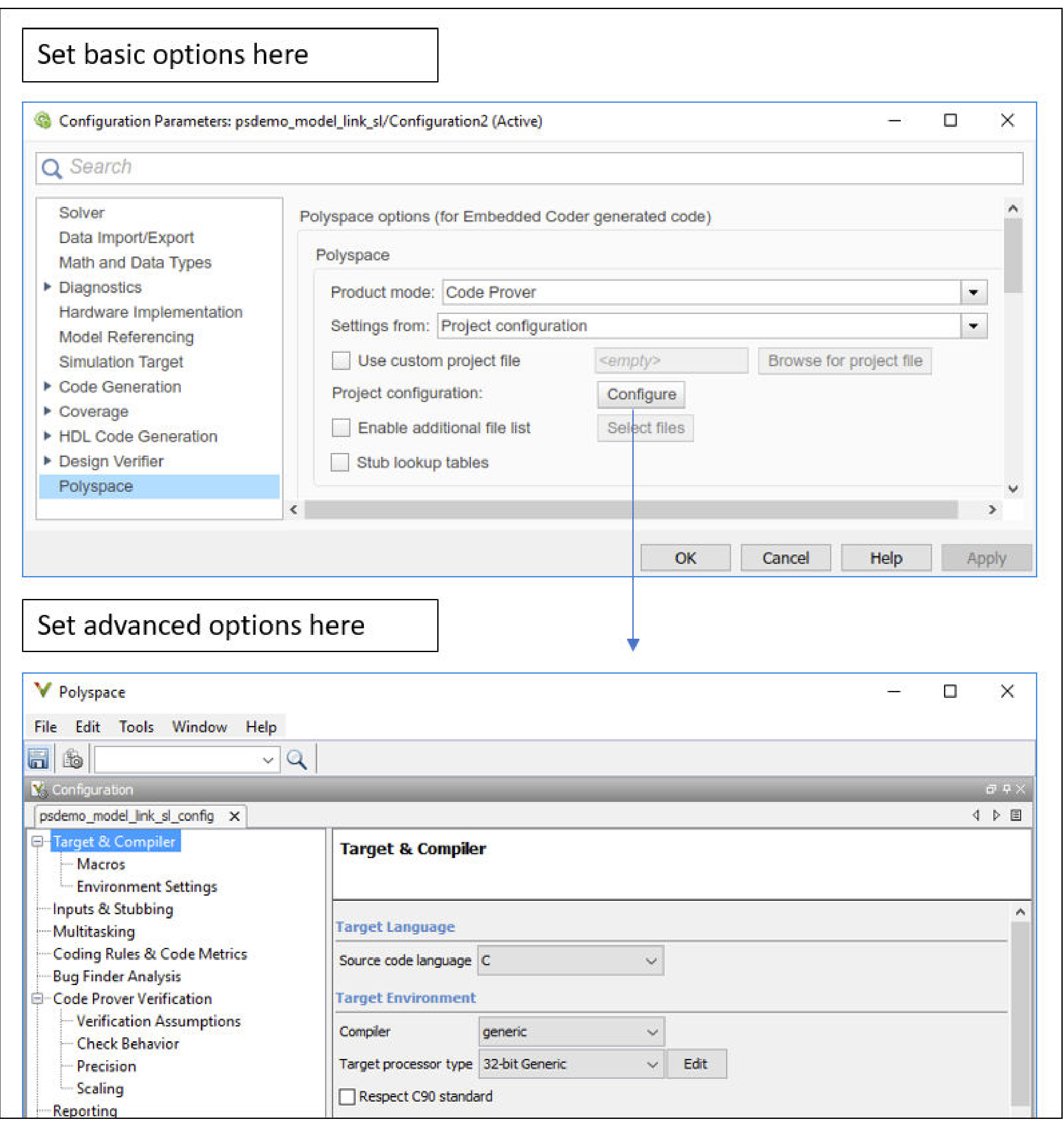
Grouping	Command-Line	Name and Location in Configuration
	Name: `Solver (Simulink)` Value: `'FixedStepDiscrete'`	Location: **Solver** Name: **Solver** Value: `discrete (no continuous states)`

Configure Advanced Polyspace Options in Simulink

Before analyzing generated code in Simulink, you can change some of the default options. This topic shows how to configure the options and save this configuration.

For getting started with Polyspace analysis in Simulink, see “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 5-2.

Configure Options


Set basic options here
Configuration Parameters: psdemo_model_link_sl/Configuration2 (Active)
Search
Solver
Data Import/Export
Math and Data Types
Diagnostics
Hardware Implementation
Model Referencing
Simulation Target
Code Generation
Coverage
HDL Code Generation
Design Verifier
Polyspace
Polyspace options (for Embedded Coder generated code)
Polyspace
Product mode: Code Prover
Settings from: Project configuration
Use custom project file
<empty>
Browse for project file
Project configuration:
Configure
Enable additional file list
Select files
Stub lookup tables
OK
Cancel
Help
Apply
Set advanced options here
Polyspace
File Edit Tools Window Help
Configuration
psdemo_model_link_sl_config
Target & Compiler
Macros
Environment Settings
Inputs & Stubbing
Multitasking
Coding Rules & Code Metrics
Bug Finder Analysis
Code Prover Verification
Verification Assumptions
Check Behavior
Precision
Scaling
Reporting
Target & Compiler
Target Language
Source code language C
Target Environment
Compiler generic
Target processor type 32-bit Generic
Edit
Respect C90 standard

Set basic options

The commonly used options appear in Simulink Configuration Parameters. Select **Code** > **Polyspace** > **Options**.

Set advanced options

Select **Code** > **Polyspace** > **Options**. From the Configuration Parameters window, you can access a wider set of options for configuring the analysis. Click the **Configure** button beside **Project configuration**.

For instance, you can:

- Run the code analysis on a remote cluster. Use the option `Run Bug Finder or Code Prover analysis on a remote cluster`.

  If you use this option, after starting the analysis, you can follow the progress of the analysis on the remote cluster through the Job Monitor window. Select **Code** > **Polyspace** > **Open Job Monitor**.
- Stub certain functions for the analysis and then constrain the function output. Use the options `Functions to stub (-functions-to-stub)` and `Constraint setup (-data-range-specifications)`.

If a basic option in the Configuration Parameters window directly conflicts with an advanced option in the Polyspace window, the former prevails. For instance, in this situation, Polyspace checks for MISRA C: 2012 rules:

- “Settings from (C)”: You select this basic option `Project configuration and MISRA C 2012 checking for generated code`.
- `Check MISRA C:2012 (-misra3)`: You disable this advanced option.

Share and Reuse Configuration

You can share the basic or advanced options across multiple models.

- Basic options: You can share and reuse the options set in the Configuration Parameters window. See “Share a Configuration for Multiple Models” (Simulink).
- Advanced options: The advanced options are saved in a separate Polyspace project associated with your analysis. Share this project across multiple models.

The next sections show how to reuse the advanced options. You can specify the advanced options just once. You can reuse these advanced options across multiple models and set only the basic options individually in each model.

Set options from model

Set the advanced options as needed. To see where the associated project file is stored or change the name of the file, on the Polyspace window toolbar, click the icon.

Reuse options in another model

To reuse the advanced options in another model, open the Configuration Parameters window from the model. Select **Code > Polyspace > Options**.

- Select **Use custom project file**. Provide the path to the project file previously created (extension `.psprj`).
- For **Settings from**, select `Project configuration` so that the settings in your project are used.

  If you want to check for additional issues, for instance MISRA C: 2012 violations, select `Project configuration and MISRA C 2012 checking for generated code`.

See Also

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 5-2

- “Run Polyspace Analysis on Code Generated with TargetLink” on page 5-32
- “Default Polyspace Options for Code Generated with Embedded Coder” on page 5-29
- “Default Polyspace Options for Code Generated with TargetLink” on page 5-34

How Polyspace Analysis of Generated Code Works

When you run Polyspace on generated code, the software automatically reads the following information from the generated code:

- `initialize()` functions
- `terminate()` functions
- `step()` functions
- List of parameter variables
- List of input variables

If you run Code Prover, the software uses this information to generate a `main` function that:

1. Initializes parameters using the Polyspace option `Parameters (-variables-written-before-loop)`.
2. Calls initialization functions using the option `Initialization functions (-functions-called-before-loop)`.
3. Initializes inputs using the option `Inputs (-variables-written-in-loop)`.
4. Calls the `step` function using the option `Step functions (-functions-called-in-loop)`.
5. Calls the `terminate` function using the option `Termination functions (-functions-called-after-loop)`.

The `main` function conceptually looks like this:

```
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
```

Code Prover uses this generated `main` function to perform the subsequent analysis.

For C++ code that is generated with Embedded Coder, the `initialize()`, `step()`, and `terminate()` functions and associated variables are either class members or have global scope.

Default Polyspace Options for Code Generated with Embedded Coder

In this section...

"Default Options" on page 5-29

"Constraint Specification" on page 5-29

"Recommended Polyspace options for Verifying Generated Code" on page 5-30

"Hardware Mapping Between Simulink and Polyspace" on page 5-30

Default Options

For Embedded Coder code, the software sets the following verification options by default:

```
-sources path_to_source_code
-D PST_ERRNO
-D main=main_rtwec
-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]
-results-dir results
```

Note *matlabroot* is the MATLAB installation folder.

Constraint Specification

You can constrain inputs, parameters, and outputs to lie within specified ranges. Use these configuration parameters:

- "Input""Input" (Polyspace Code Prover)
- "Tunable parameters"
- "Output"

The software automatically creates a Polyspace constraints file using information from the MATLAB workspace and block parameters.

You can also manually define a constraints file in the Polyspace user interface. See "Specify External Constraints" on page 9-2. If you define a constraints file, the software appends the automatically generated information to the constraints file you create. Manually defined constraint information overrides automatically generated information for all variables.

The software supports the automatic generation of constraint specifications for the following kinds of generated code:

- Code from standalone models
- Code from configured function prototypes
- Reusable code
- Code generated from referenced models and submodels

Recommended Polyspace options for Verifying Generated Code

For Embedded Coder code, the software automatically specifies values for the following verification options:

- `-main-generator`
- `-functions-called-in-loop`
- `-functions-called-before-loop`
- `-functions-called-after-loop`
- `-variables-written-in-loop`
- `-variables-written-before-loop`

In addition, for the option `-server`, the software uses the value specified in the **Send to Polyspace server** check box on the **Polyspace** pane. These values override the corresponding option values in the **Configuration** pane of the Polyspace user interface.

You can specify other verification options for your Polyspace Project through the Polyspace **Configuration** pane. See "Configure Advanced Polyspace Options in Simulink" on page 5-23.

Hardware Mapping Between Simulink and Polyspace

The software automatically imports target word lengths and byte ordering (endianness) 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 verification.

Run Polyspace Analysis on Code Generated with TargetLink

You can analyze code generated from Simulink models with TargetLink.

You have fewer capabilities for code generated with TargetLink compared to code generated with Embedded Coder. For instance, you cannot add annotations to your blocks that carry over to the generated code and justify known issues.

Configure and Run Analysis

Configure code analysis

Select **Code** > **Polyspace** > **Options**. Change default values of these options if needed.

- "Product mode": Choose Bug Finder or Code Prover.
- "Settings from (C)": Enable checking of MISRA or JSF® coding rules in addition to the default checks.
- "Output folder": Specify a dedicated folder for results. The default analysis runs Code Prover on generated code and saves the results in a folder `results_`*`modelName`* in the current working folder.
- "Enable additional file list": Add C files that are not part of the generated code.
- "Open results automatically after verification"

Analyze code

To analyze the code, select **Code** > **Polyspace** > **Verify Code Generated for** > **Selected Target Link Subsystem**. You cannot analyze code generated from the entire model.

You can follow the progress of the analysis in the MATLAB command window.

The results open automatically unless explicitly disabled. By default, the results are saved in a folder `results_`*`ModelName`* in the current folder. Each new run overwrites previous results. You can change these behaviors or save the results to a Simulink project using appropriate configuration parameters.

Review Analysis Results

Review result in code

The results appear on the **Results List** pane. Click each result to see the source code and details on the **Result Details** pane.

Navigate from code to model

Links in code comments show blocks that generate the subsequent lines of code. To see the blocks in the model, click the block names.

Fix issue

Investigate whether the issues in your code are related to design flaws in the model.

For instance, you might need to constrain the range of signal from Inport blocks. See “Specify Ranges for Signals” (Simulink).

Default Polyspace Options for Code Generated with TargetLink

In this section...

"TargetLink Support" on page 5-34
"Default Options" on page 5-34
"Lookup Tables" on page 5-35
"Data Range Specification" on page 5-35
"Code Generation Options" on page 5-36

TargetLink Support

The Windows version of Polyspace Bug Finder is supported for versions 3.5 and 4.0 of the dSPACE® Data Dictionary and TargetLink Code Generator.

Polyspace Bug Finder does support CTO generated code. However, for better results, MathWorks® recommends that you disable the CTO option in TargetLink before generating code. For more information, see the dSPACE documentation.

Because Polyspace Bug Finder extracts information from the dSPACE Data Dictionary, you must regenerate the code before performing an analysis.

Default Options

Polyspace sets the following options by default:

```
-sources path_to_source_code
-results-dir results_folder_name
-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]
-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

By default, Polyspace provides stubs for the lookup table functions. The dSPACE data dictionary is used to define the range of their return values. A lookup table that uses extrapolation returns full range for the type of variable that it returns. You can disable this behavior from the Polyspace configuration menu.

Data Range Specification

You can constrain inputs, parameters, and outputs to lie within specified data ranges. See “Specify Ranges for Signals” (Simulink).

The software automatically creates a Polyspace constraints file using the dSPACE Data Dictionary for each global variable. The constraint information is used to initialize each global variable to the range of valid values as defined by the min..max information in the data dictionary. This information allows Polyspace software to model real values for the system during analysis. Carefully defining the min-max information in the model allows the analysis to be more precise, because only the range of real values is analyzed.

Note Boolean types are modeled having a minimum value of 0 and a maximum of 1.

You can also manually define a constraint file in the Polyspace user interface. See “Specify External Constraints” on page 9-2. If you define a constraint file, the software appends the automatically generated information to the constraint file you create. Manually defined constraint information overrides automatically generated information for all variables.

Constraints cannot be applied to static variables. Therefore, the compilation flags `-D static=` is set automatically. It has the effect of removing the static keyword from the

code. If you have a problem with name clashes in the global name space, either rename the variables or disable this option in Polyspace configuration.

Code Generation Options

From the TargetLink Main Dialog, it is recommended to:

- Set the option `Clean code`
- Unset the option `Enable sections/pragmas/inline/ISR/user attributes`
- Turn off the compute to overflow (CTO) generation. Polyspace can analyze code generated with CTO, but the results may not be as precise.

When you install Polyspace, the `tlcgOptions` variable is updated with `'PolyspaceSupport', 'on'` (see variable in `'C:\dSPACE\Matlab\Tl\config\codegen\tl_pre_codegen_hook.m'` file).

See Also

Related Examples

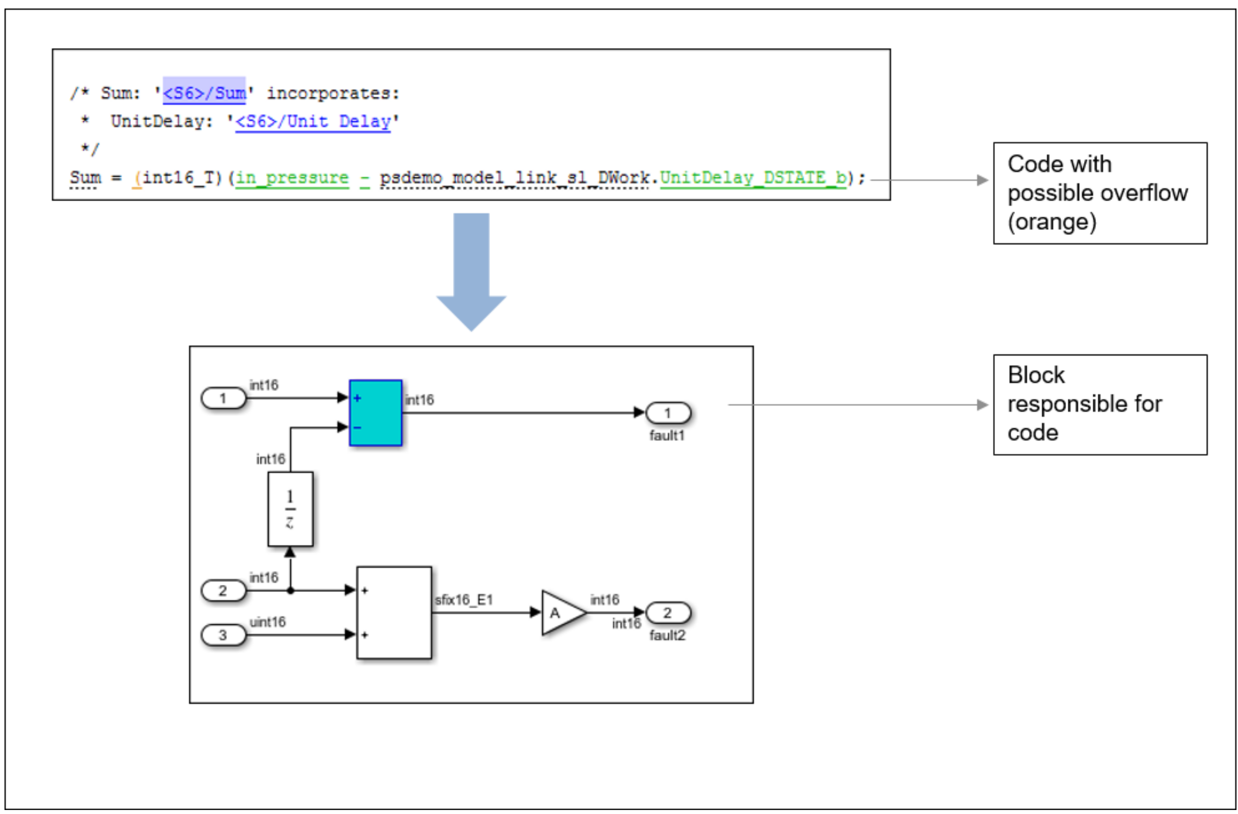
- "Run Polyspace Analysis on Code Generated with TargetLink" on page 5-32

External Websites

- dSPACE – TargetLink

Troubleshoot Navigation from Code to Model

When you run Polyspace on generated code, in the analysis results, you see links in code comments. The links show names of blocks that generate the subsequent lines of code. To see the blocks in the model, you click the block names in the links.



This topic shows the issues that can happen in navigation from code to model.

Links from Code to Model Do Not Appear

See if you are looking at source files (`.c` or `.cpp`) or header files. Header files are not directly associated with blocks in the model and do not have links back to the model.

Links from Code to Model 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 `pslinkfun('enablebacktomodel')`.

   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. You can change the color of blocks when they are linked to Polyspace results. For instance, to change the color to magenta, use this command:

```
color = 'magenta';
HILITE_DATA = struct('HiliteType', 'find', 'ForegroundColor', 'black', ...
        'BackgroundColor', color);
set_param(0, 'HiliteAncestorsData', HILITE_DATA)
```

The color can be one of the following:

- `'cyan'`
- `'magenta'`
- `'orange'`
- `'lightBlue'`

- `'red'`
- `'green'`
- `'blue'`
- `'darkGreen'`

Run Polyspace on C/C++ Code Generated from MATLAB Code

After generating C/C++ code from MATLAB code, you can independently check the generated code for:

- Bugs or defects and coding rule violations: Use Polyspace Bug Finder.
- Run-time errors: Use Polyspace Code Prover.

Whether you generate code in the MATLAB Coder™ app or use `codegen`, you can follow the same workflow for checking the generated code.

This tutorial uses the MATLAB Coder example `averaging_filter`. To copy the required MATLAB files into a temporary folder and change to the folder, enter:

```
coderdemo_setup('coderdemo_averaging_filter');
```

The example shows a Code Prover analysis. You can follow a similar workflow for Bug Finder.

Prerequisites

To run this tutorial:

- You must have an Embedded Coder license. The MATLAB Coder app does not show options for running Polyspace unless you have an Embedded Coder license.
- You must be familiar with how to open and use the MATLAB Coder app or the `codegen` command. Otherwise, see the MATLAB Coder Getting Started.

Run Polyspace Analysis

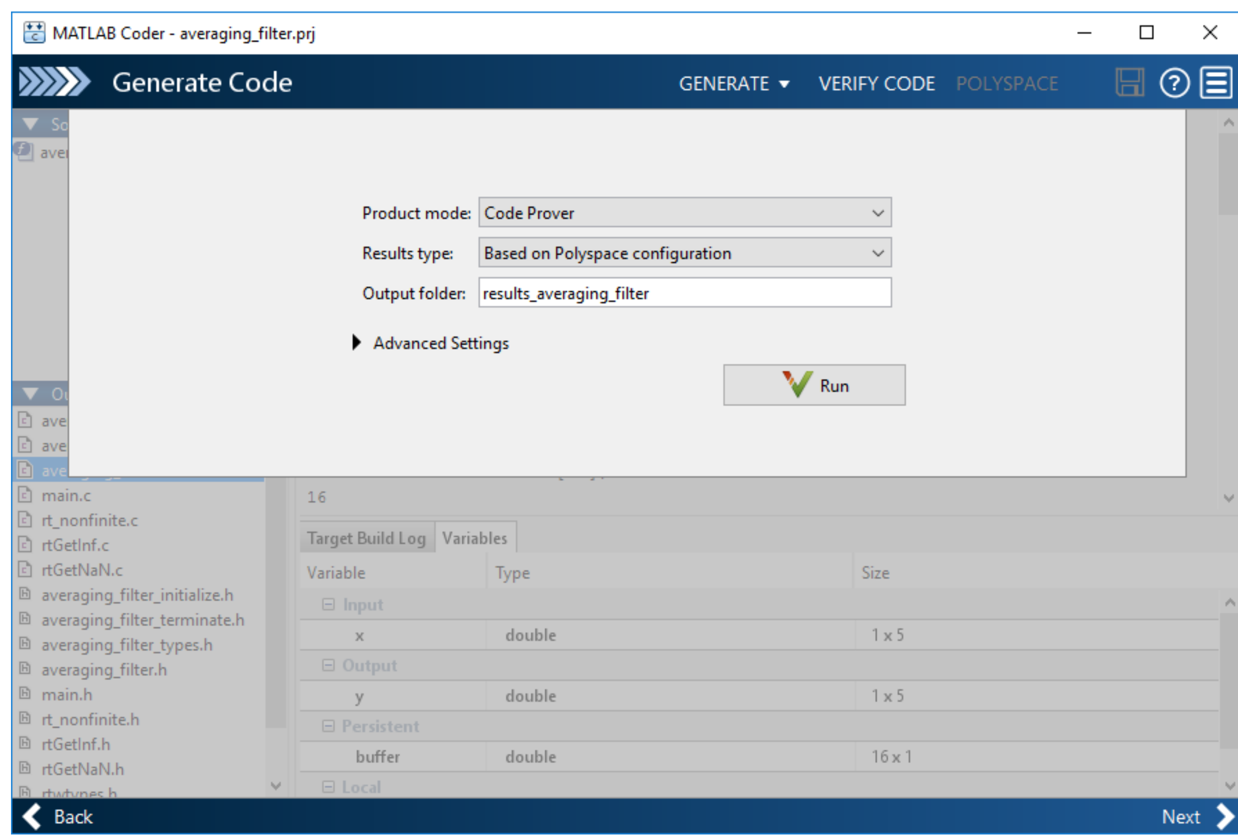
In the MATLAB Coder app, generate code from the file `averaging_filter.m` and analyze the generated code.

1. Generate code.

   From the entry-point function in the file, generate standalone C/C++ code (a static library, dynamically linked library, or executable program) in the MATLAB Coder app. The function has one input. Explicitly specify a data type for the input, for instance, a 1 X 100 vector of type `double`, or provide a file for deriving data types.

2 Analyze the generated code.

After code generation, open the **Polyspace** pane and click **Run**.



If the analysis is completed without errors, the Polyspace results open automatically. If you close the results, you can reopen them from the final page in the app, under the section **Generated Output**. The results are stored in a subfolder `results_averaging_filter` in the folder containing the MATLAB file.

To script the preceding workflow, run:

```
% Copy demo files into a temporary folder
coderdemo_setup('coderdemo_averaging_filter');

% Generate code
codeName = 'averaging_filter';
codegenFolder = fullfile(pwd, 'codegenFolder');
codegen(codeName, '-config:lib', '-c', '-args', ...
       {zeros(1,100,'double')}, '-d', codegenFolder);


% Configure Polyspace analysis
opts = pslinkoptions('ec');
opts.ResultDir = ['results_',codeName];
opts.OpenProjectManager = 1;

% Run Polyspace
pslinkrun('-codegenfolder', codegenFolder, opts);
```

Review Analysis Results

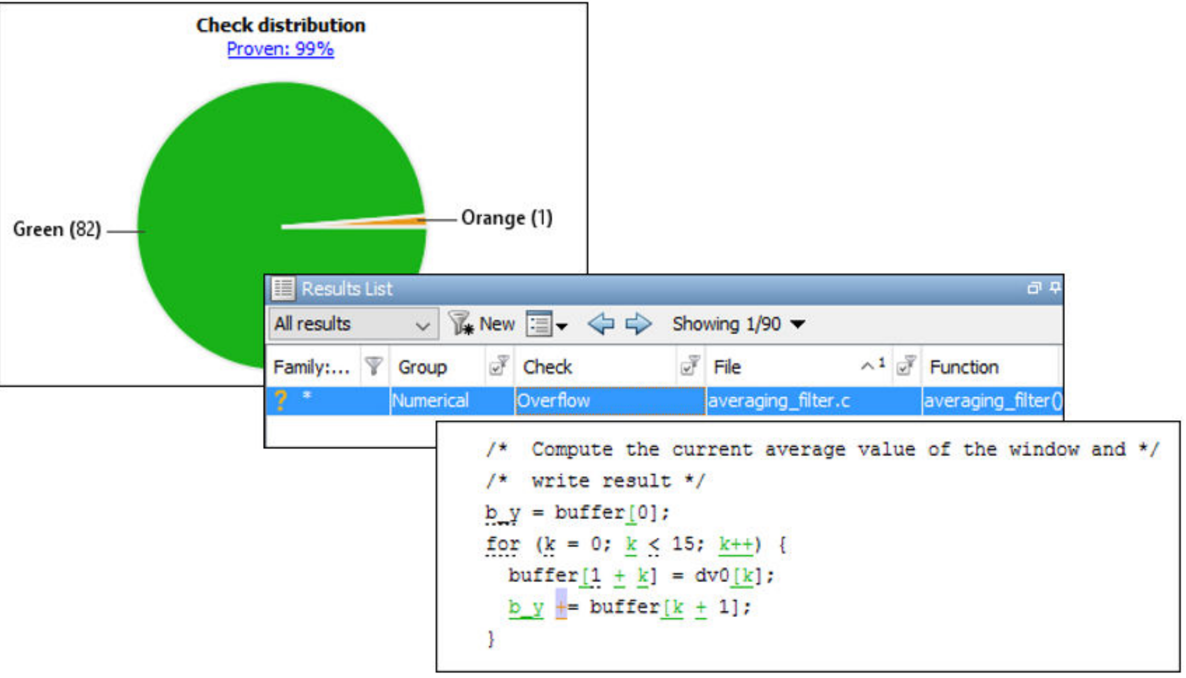
After analysis, the **Results List** pane shows a list of run-time checks. For an explanation of the result colors, see “Code Prover Result and Source Code Colors” (Polyspace Code Prover).

Review the results and determine whether to fix the issues.

1. Filter out results that you do not want to review. For instance, you might not want to see the green checks.

   See an overview of the results on the **Dashboard** pane. Click the orange section of the pie chart to filter the list of results on the **Results List** pane to the one orange check. Click this orange **Overflow** check and see the source code for the operation that can overflow.

   If results are grouped by family, to see a flat list, on the **Results List** pane, from the dropdown, select **None**.



2 Find the root cause of each run-time error.

On the **Source** pane, use right-click navigation tools and tooltips to identify the root cause of the check. In this case, you see that the `+` operation overflows because Polyspace makes an assumption about the input array to the function. The assumption is that the array elements can have any value allowed by their `double` data type. The tooltip on the line `buffer[0] = x[i]` shows the assumed range.

```
/*  Add a new sample value to the buffer */
buffer[0] = x[i];

/*  Com
/*  wri
b_y = b
for (k
  buffe
  b_y += buffer[k + 1];
}
```

Assignment to element of static array (float 64): [-1.7977E+308 .. 1.7977E+308]

array size: 16
array index value: 0

Press 'F2' for focus

With an Embedded Coder license, you can easily trace back from the generated C code to the original MATLAB code. See “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder).

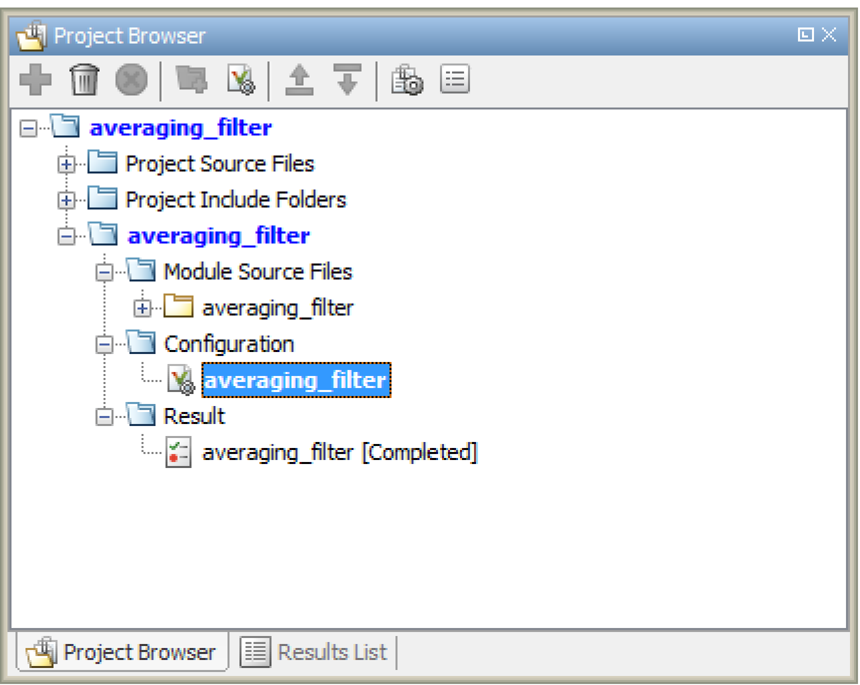
Run Analysis for Specific Design Range

You can check the generated code for a specific range of inputs. Range specification helps narrow down the default assumption that inputs are full-range.

To specify a range for inputs:

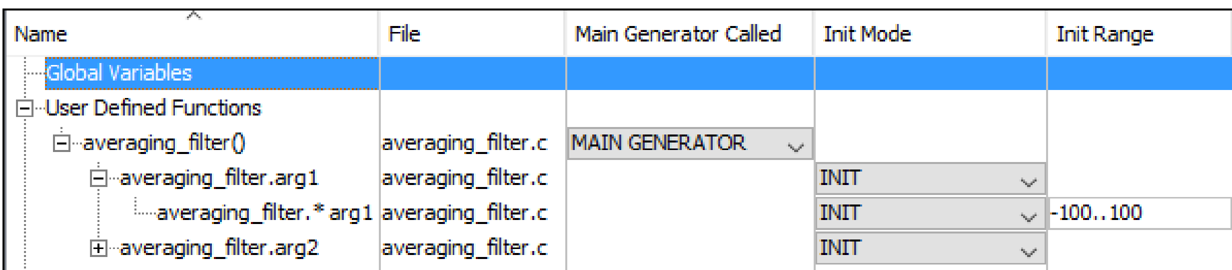
1 Open the analysis configuration.

   In the Polyspace user interface, switch to the Polyspace project created for the analysis. Select **Window** > **Reset Layout** > **Project Setup**. On the **Project Browser** pane, click the project configuration.



2 Specify a design range for the inputs.

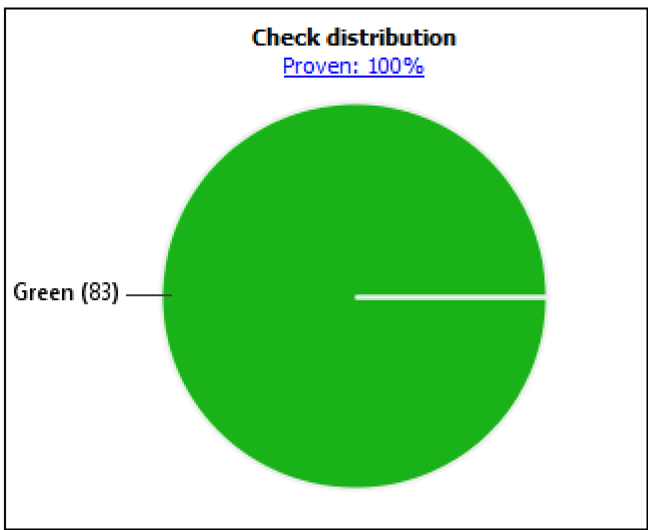
In the **Configuration** pane, on the **Inputs & Stubbing** node, set up your constraints. Click **Edit** beside **Constraint setup**. Constrain the range of the first input to [-100..100].



You can overwrite the default constraint template or save the constraints elsewhere. For information on the columns in this window, see “External Constraints for Polyspace Analysis” on page 9-6.

3. Rerun the analysis from the Coder app (or at the MATLAB command line) and see the results.

   On the **Dashboard** pane, you do not see the previous orange overflow anymore.



See Also

More About

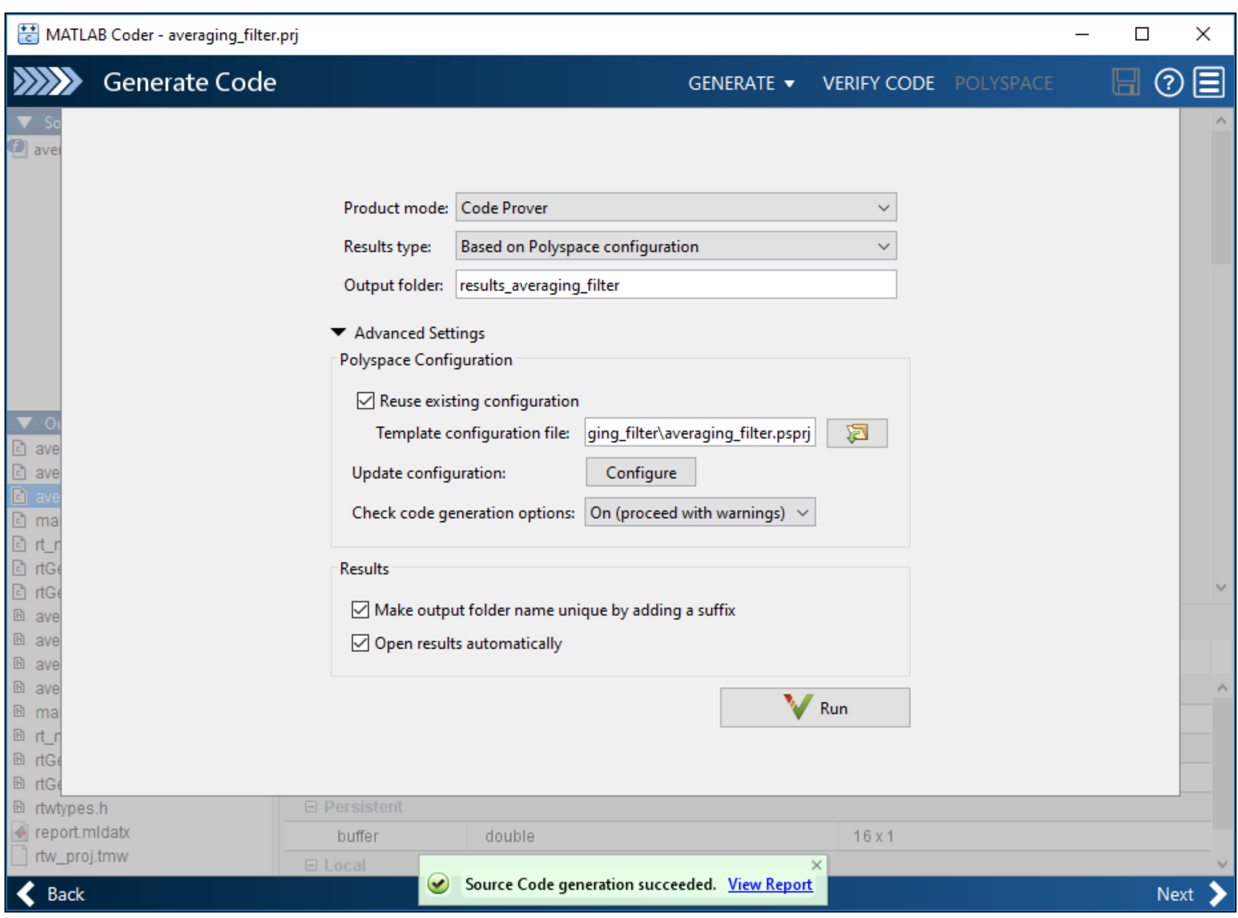
- “Configure Advanced Polyspace Options in MATLAB Coder App” on page 5-47

Configure Advanced Polyspace Options in MATLAB Coder App

Before analyzing generated code with Polyspace in the MATLAB Coder App, you can change some of the default options. This topic shows how to configure the options and save this configuration.

For getting started with Polyspace analysis in the MATLAB Coder App, see “Run Polyspace on C/C++ Code Generated from MATLAB Code” on page 5-40.

Configure Options



The default analysis runs Code Prover based on a default project configuration. The results are stored in a folder `result_project_name` in the current working folder.

You can change these options in the MATLAB Coder App itself:

- **Product mode**: Select Code Prover or Bug Finder.
- **Results type**: Check for MISRA C:2004 (MISRA AC AGC) or MISRA C:2012 rule violations, in addition to or instead of the default checkers.

- **Output folder**: Choose an output folder name. To save the results of each run in a new folder, under **Advanced Settings**, select **Make output folder name unique by adding a suffix**.
- **Check code generation options**: Choose to see warnings or errors if the code generation uses options that can result in imprecise Code Prover analysis.

  For instance, if the code generation setting **Use memset to initialize floats and doubles to 0.0** is disabled, Code Prover can show imprecise orange checks because of approximations. See "Orange Checks in Code Prover" (Polyspace Code Prover).

To see the other default options or update them, under **Advanced Settings**, click the **Configure** button. You see the options on a **Configuration** pane.

For more information on the options, see Bug Finder Analysis Options or Code Prover Analysis Options (Polyspace Code Prover).

Share and Reuse Configuration

If you change some of the default options in the **Configuration** pane, your updated configuration is saved as a `.psprj` file in the results folder. Using this file, you can reuse your configuration across multiple MATLAB Coder projects.

To reuse a previous configuration in the current project opened in the MATLAB Coder App, under **Advanced Settings**, select **Reuse existing configuration**. For **Template configuration file**, provide the `.psprj` file that stores the previous configuration.

The **Results type** option in the MATLAB Coder app still shows **Based on Polyspace configuration** but the configuration used is the one that you provided.

More About

- "Run Polyspace on C/C++ Code Generated from MATLAB Code" on page 5-40

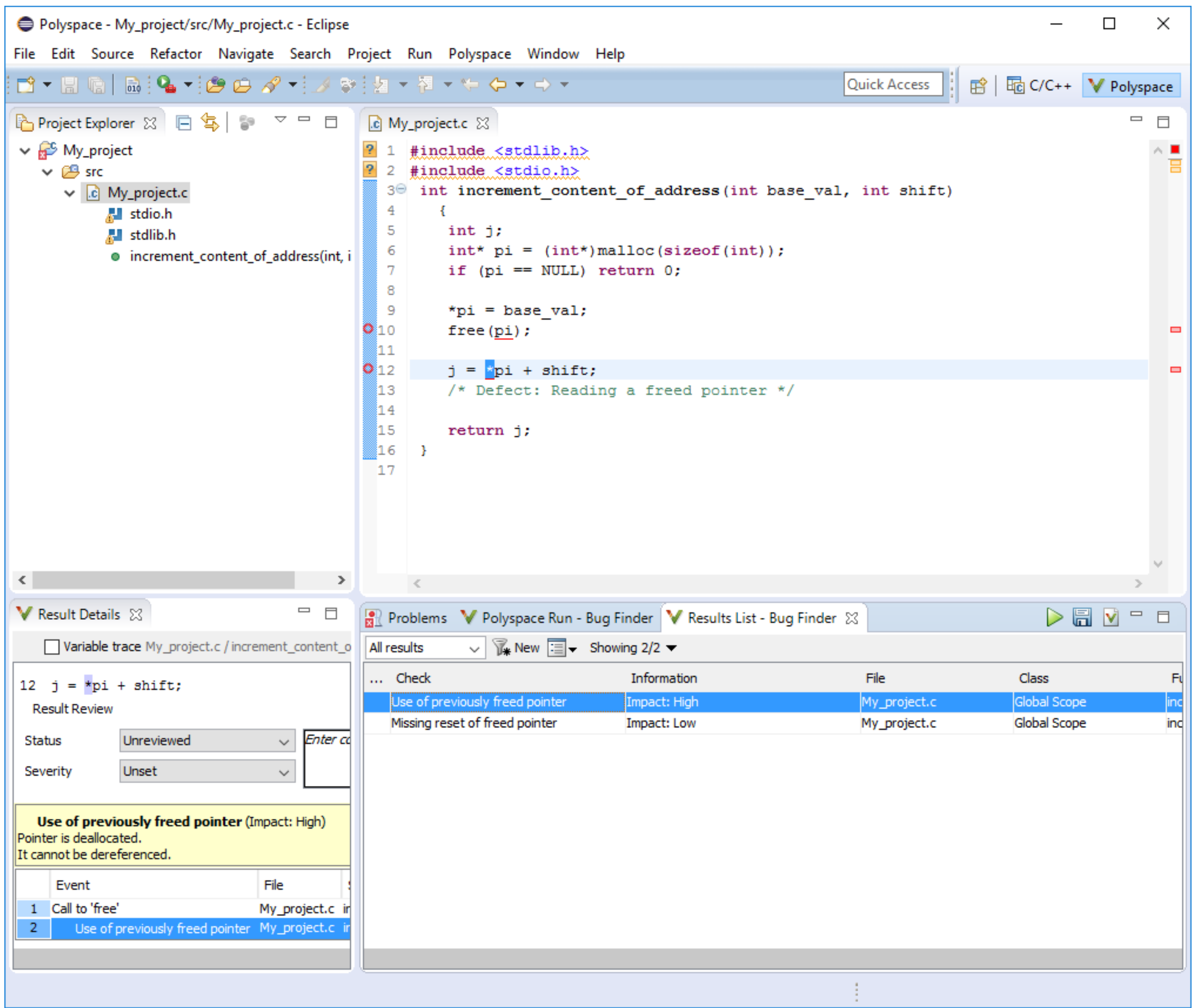
6

Run Polyspace Analysis in IDE Plugins

Run Polyspace Analysis in Eclipse

If you develop code in Eclipse or an Eclipse-based IDE, you can install the Polyspace plugin and run a Polyspace analysis on the source files in an Eclipse project. You can check for bugs each time you save your code, or explicitly run an analysis.

This topic describes how to set up a Polyspace analysis in Eclipse and review analysis results.



After you install the Polyspace plugin, you see a **Polyspace** menu and right-click options in the **Project Explorer** to run a Polyspace analysis.

The analysis progress bar, quick run buttons and analysis results appear on specific panes. To avoid cluttering your window, you can confine these panes to the Polyspace

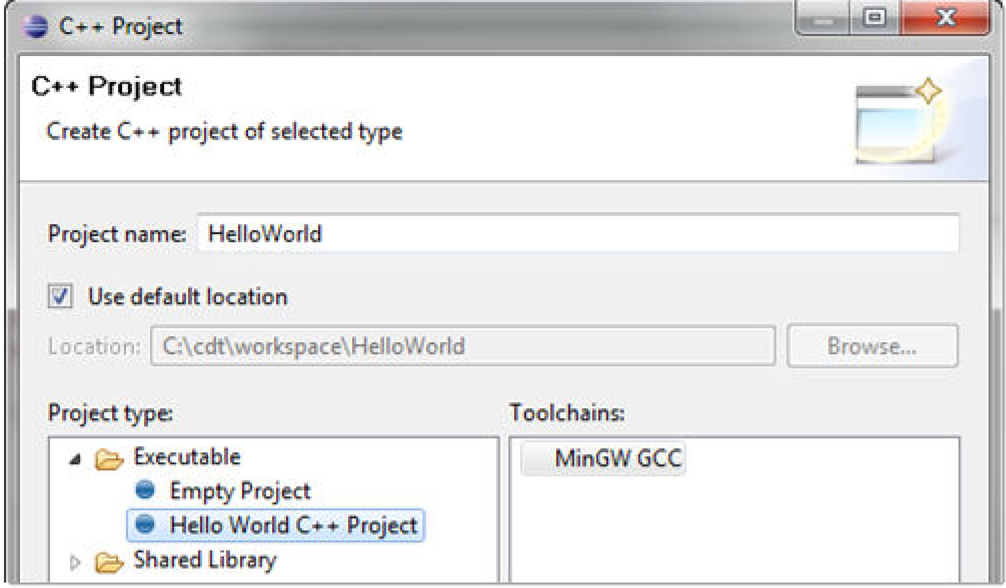
perspective. Select **Window > Open Perspective > Other**. In the Open Perspective dialog box, select **Polyspace**. You can switch back to other perspectives using tabs on the upper right.

Configure and Run Analysis

Configure analysis

Polyspace analyzes the source files in your Eclipse project. In addition to sources, the analysis uses the following information:

- Compiler: The compiler toolchain can be extracted from your Eclipse project. If the project directly refers to a compilation toolchain such as MinGW GCC, the Polyspace analysis can use the information.

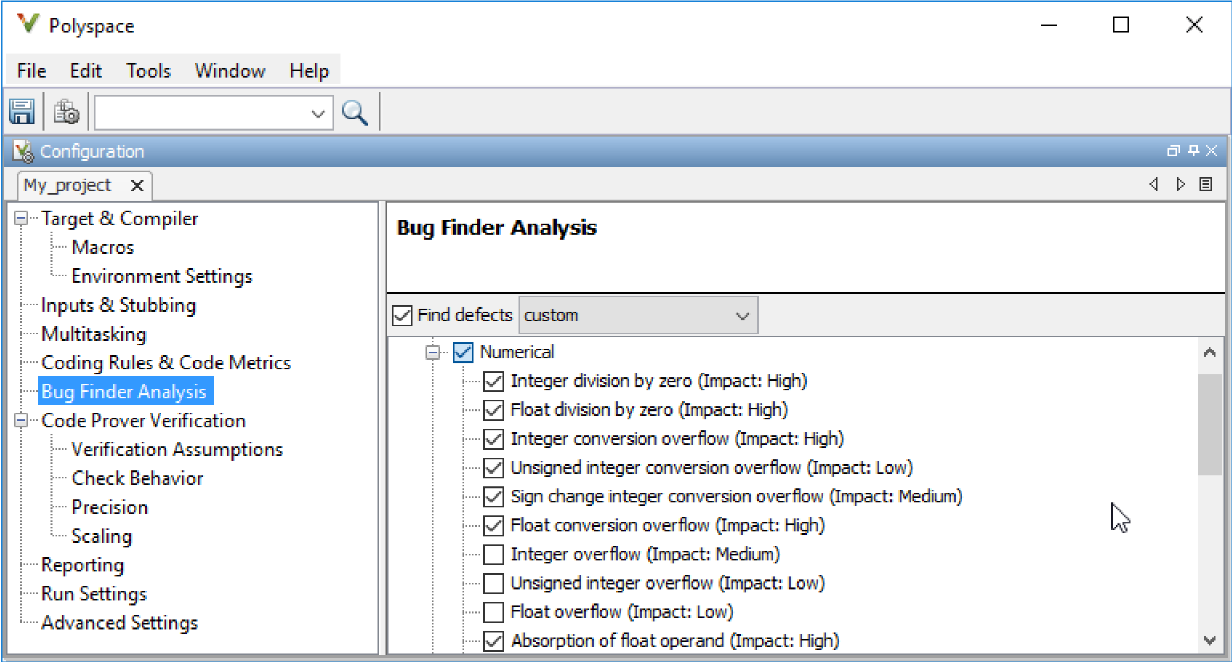


If your Eclipse project uses a build command (makefile) that has the compiler information, you must perform some additional steps to extract this information for the Polyspace analysis.

If Polyspace cannot extract the compiler information from your build command, you can also explicitly specify your compiler options explicitly like other analysis options.

See "Specify Polyspace Compiler Options Through Eclipse Project" on page 6-8.

- Other analysis options: You can retain the default analysis options or adjust them to your requirements. Select **Polyspace > Configure Project**.



The key options are:

- **Target & Compiler**: If you have not specified your compiler information through your Eclipse project, use these options.
- **Bug Finder Analysis**: Specify which defects to check for in a Bug Finder analysis.
- **Code Prover Verification**, **Check Behavior**, **Precision**: Modify the behavior of checkers in a Code Prover verification.

Note that you cannot run a remote analysis using the Polyspace plugin for Eclipse. To send the analysis job to a remote cluster, start the analysis from the Polyspace user interface or using scripts. See “Polyspace Analysis on Clusters”.

Run analysis

After configuration, you can start and stop a Polyspace analysis explicitly from the **Polyspace** menu, right-click options on your Eclipse project or quick run buttons in the Polyspace panes. You can switch between Bug Finder and Code Prover using the icon on the **Polyspace Run** pane.

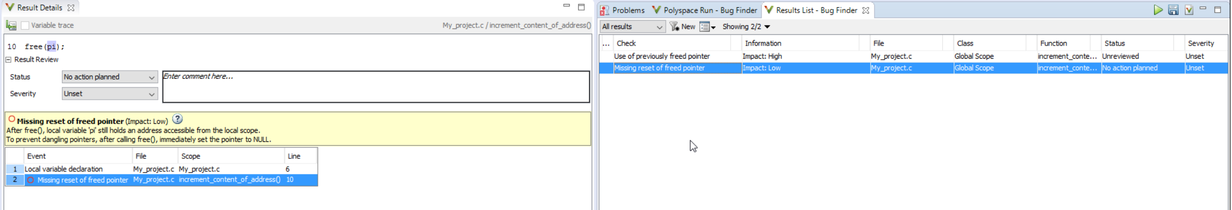
Run analysis when saving code

In the Polyspace perspective, you can set up a Bug Finder analysis that runs each time you save your code. To enable this analysis, select **Polyspace > Run Fast Analysis on Save**. The analysis runs quickly but looks for a reduced set of defects. You get the same results as if you had specified the analysis option `Use fast analysis mode for Bug Finder (-fast-analysis)`.

Review Analysis Results

View results after analysis

After analysis, the results appear on the **Results List** pane. Click each result to see the source code and details on the **Result Details** pane.



View results as available

Some results of a Bug Finder analysis are often available before the analysis is complete. If so, the icon in the **Polyspace Run - Bug Finder** pane turns to . To load available results, click this icon. The icon shows up again when more results are available.

Address results

Based on the result details, fix your code or justify the result. To justify a result, set its **Status** to `Justified`, `No Action Planned` or `Not a Defect`. To hide a justified result in the next run, add the status as annotation to your source code. For quick annotation, right-click the result and select **Hide Result and Annotate Code**.

See Also

Related Examples

- “Specify Polyspace Compiler Options Through Eclipse Project” on page 6-8
- “Interpret Polyspace Bug Finder Results” on page 14-2
- “Address Polyspace Results Through Bug Fixes or Comments” on page 15-2
- “Filter and Group Results” on page 16-2

Specify Polyspace Compiler Options Through Eclipse Project

Polyspace analysis in Eclipse uses a set of default analysis options preconfigured for your coding language and operating system. For each project, you can customize the analysis options further.

- Compiler options: You specify the compiler that you use, the libraries that you include and the macros that are defined for your compilation.
  - If your Eclipse project directly refers to a compilation toolchain, the analysis extracts the compiler options from the project.

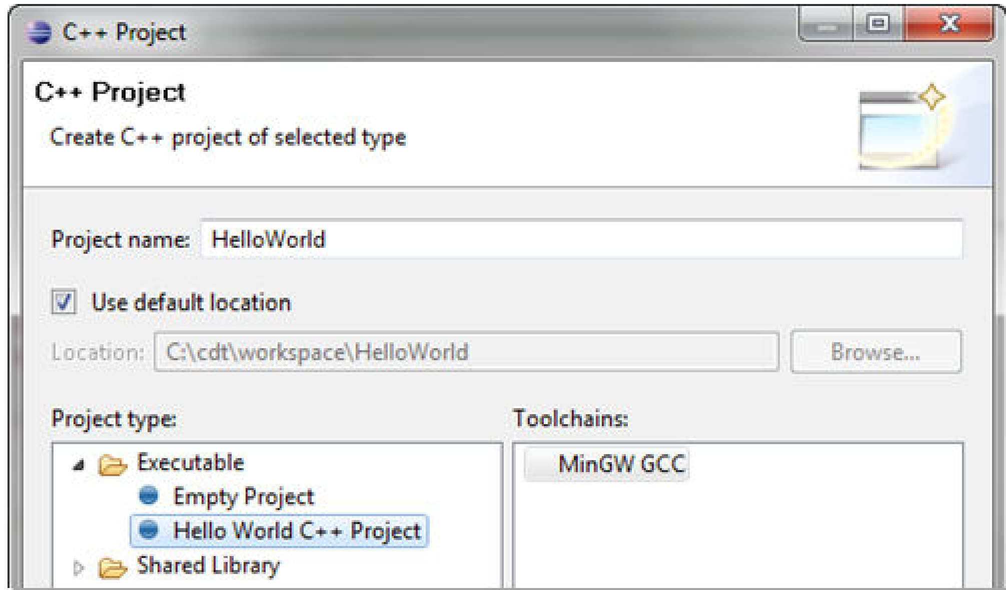
    See “Eclipse Refers Directly to Your Compilation Toolchain” on page 6-8.
  - If the project refers to your compilation toolchain through a build command, the analysis cannot extract the compiler options. Trace the build command to extract the options.

    See “Eclipse Uses Your Compilation Toolchain Through Build Command” on page 6-9.
- Other options: Through the other options, you specify which analysis results you want and how precise you want them to be.To specify these options in Eclipse, select **Polyspace > Configure Project**.

For information on how to run Polyspace from Eclipse, see “Run Polyspace Analysis in Eclipse” on page 6-2.

Eclipse Refers Directly to Your Compilation Toolchain

When setting up your Eclipse project, you might be directly referring to your compilation toolchain without using a build command. For instance, you might refer to the MinGW GCC toolchain in the project setup wizard as below.



The compiler options from your Eclipse project, such as include paths and preprocessor macros, are reused for the analysis.

You cannot view the options directly in the Polyspace configuration but you can view them in your Eclipse editor. In your project properties (**Project > Properties**), in the **Paths and Symbols** node:

- See the include paths under the **Includes** tab.

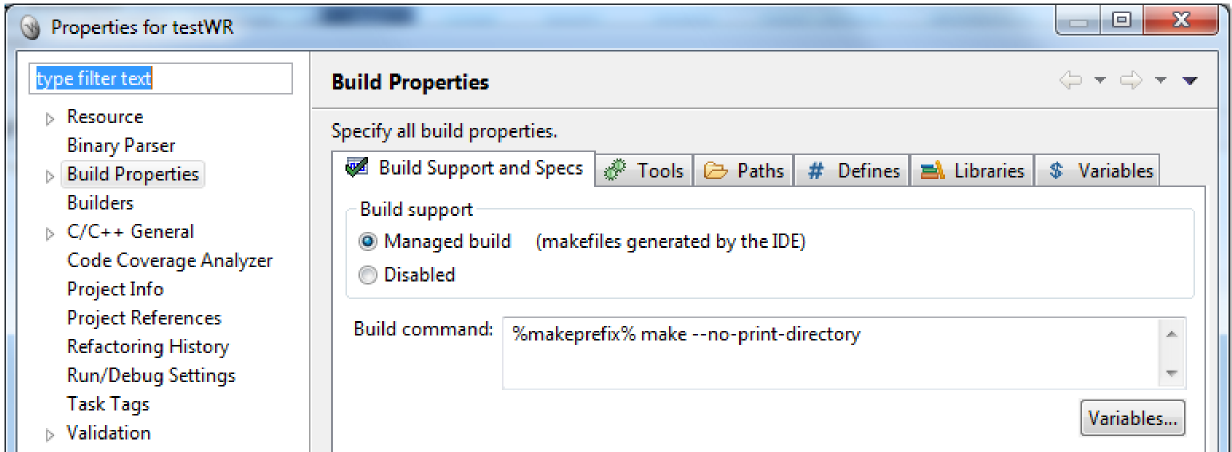
  During analysis, the paths are implicitly used with the analysis option `-I`.
- See the preprocessor macros under the **Symbols** tab.

  During analysis, the macros are implicitly used with the analysis option `Preprocessor definitions (-D)`.

Eclipse Uses Your Compilation Toolchain Through Build Command

When setting up your Eclipse project, instead of specifying your compilation toolchain directly, you might be specifying it through a build command. For instance, in the Wind

River Workbench IDE (an Eclipse-based IDE), you might specify your build command as shown in the following figure.



If you use a build command for compilation, the analysis cannot automatically extract the compiler options. You must trace your build command.

1 Replace your build command:

```
matlabroot\polyspace\bin\polyspace-configure.exe
 -output-project
PolyspaceWorkspace\Projects\EclipseProjects\Name\Name.psprj buildCommand
```

Here:

- *matlabRoot* is the MATLAB installation folder.
- *polyspaceworkspace* is the folder where your Polyspace files are stored. You specify this location on the **Project and Results Folder** tab in your Polyspace preferences (**Tools > Preferences** in the Polyspace user interface).
- *Name* is the name of your Eclipse project.
- *buildCommand* is the original build command that you want to trace.

  For instance, in the preceding example, *buildCommand* is the following:

  ```
  %makeprefix% make --no-print-directory
  ```

2 Build your Eclipse project. Perform a clean build so that files are recompiled.

For instance, select the option **Project > Clean**. Normally, the option runs your build command. With your replacement in the previous step, the option also traces the build to extract the compiler options.

3. Restore the original build command and restart Eclipse.

   You can now run analysis on your Eclipse project. The analysis uses the compiler options that it has extracted.

See Also

Related Examples

- "Run Polyspace Analysis in Eclipse" on page 6-2

Configure Polyspace Analysis

Specify Polyspace Analysis Options

You can change the default options associated with a Polyspace analysis. For instance, you can:

- Change the set of defects that Bug Finder looks for.

  See `Find defects (-checkers)`.
- Change the default behavior of run-time checkers in Code Prover.

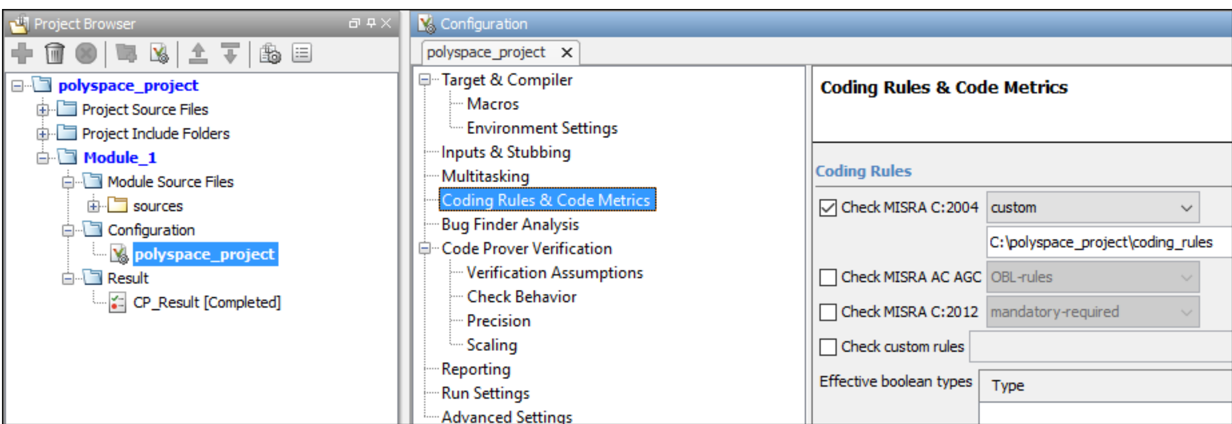
  See, for instance, `Detect overflows (-scalar-overflows-checks)`.

For the full list of analysis options, see “Analysis Options”.

Depending on how you run Polyspace, you can configure the analysis options accordingly.

Polyspace User Interface

In the Polyspace user interface, you create a project for the analysis. The project can have one or more modules. Click the **Configuration** node in a module. On the **Configuration** pane, change options as needed.



For more information, see the tooltip on each option. Click the **More help** link for context-sensitive help on the options.

For more information, see “Run Polyspace Analysis on Desktop” on page 1-8.

Windows or Linux Scripts

Provide the options to the `polyspace-bug-finder-nodesktop` or `polyspace-code-prover-nodesktop` command. See also:

- `polyspace-bug-finder-nodesktop`
- `polyspace-code-prover-nodesktop`

For instance:

```
polyspace-code-prover-nodesktop -sources file_name \
        -main-generator main-generator-writes-variables all
```

You can also provide the options in a text file. See “Run Polyspace Analysis from Command Line” on page 2-2.

MATLAB Scripts

Create a `polyspace.Project` object and set the options through the `Configuration` property of the object. See also:

- `polyspace.Project`
- polyspace.Project.Configuration Properties

For instance:

```
proj = polyspace.Project;
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.BugFinderAnalysis.EnableCheckers = false;
```

See also “Run Polyspace Analysis by Using MATLAB Scripts” on page 3-2.

Eclipse and Eclipse-based IDEs

Select **Polyspace** > **Configure Project**. Set the options in the Configuration window.

Some Target & Compiler options are automatically extracted from your Eclipse project. See “Run Polyspace Analysis in Eclipse” on page 6-2.

Simulink

In your Simulink model, specify the basic options through Simulink Configuration Parameters. Select **Code** > **Polyspace** > **Options**.

From this window, you can navigate to the full set of Polyspace analysis options.

See:

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 5-2
- “Configure Advanced Polyspace Options in Simulink” on page 5-23

MATLAB Coder App

In the MATLAB Coder app, after code generation, specify the basic options through the **Polyspace** pane. From this window, you can navigate to the full set of Polyspace analysis options.

See:

- “Run Polyspace on C/C++ Code Generated from MATLAB Code” on page 5-40
- “Configure Advanced Polyspace Options in MATLAB Coder App” on page 5-47

8

Configure Target and Compiler Options

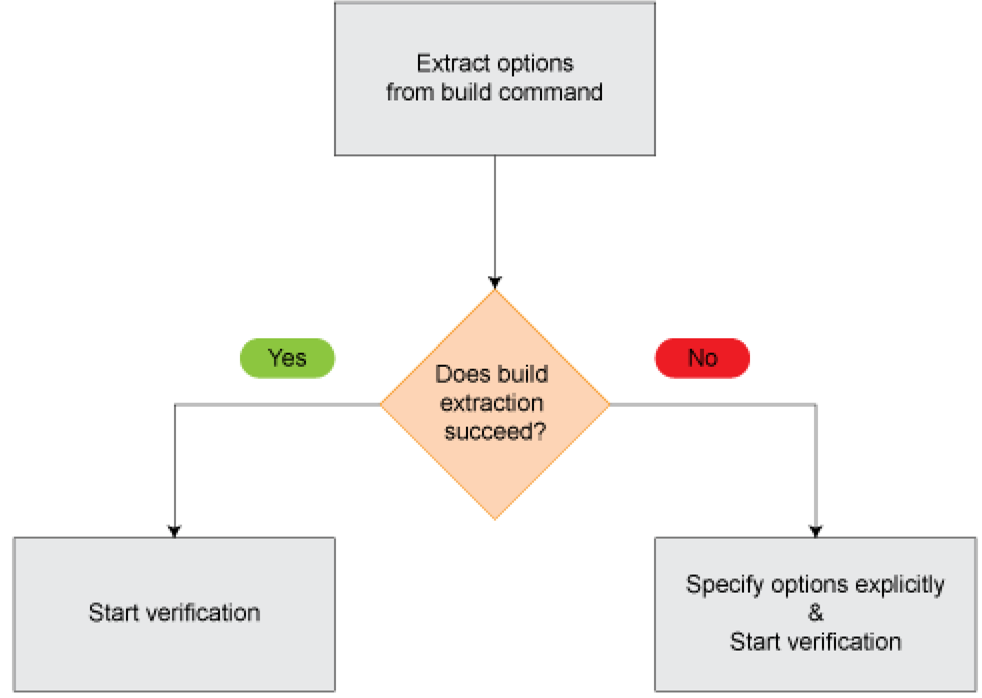
Specify Target Environment and Compiler Behavior

Before verification, specify your source code language (C or C++), target processor, and the compiler that you use for building your code. In certain cases, to emulate your compiler behavior, you might have to specify additional options.

Using your specification, the verification determines the sizes of fundamental types, considers certain macros as defined, and interprets compiler-specific extensions of the Standard. If the options do not correspond to your run-time environment, you can encounter:

- Compilation errors
- Verification results that might not apply to your target

If you use a build command such as `gmake` to build your code and the build command meets certain restrictions, you can extract the options from the build command. Otherwise, specify the options explicitly.



Extract Options from Build Command

If you use build automation scripts to build your source code, you can set up a Polyspace project from your scripts. The options associated with your compiler are specified in that project.

For information on how to trace your build command from the:

- Polyspace user interface, see "Add Source Files for Analysis in Polyspace User Interface" on page 1-2.
- DOS or UNIX command line, see `polyspace-configure`.
- MATLAB command line, see `polyspaceConfigure`.

For Polyspace project creation, your build automation script (makefile) must meet certain requirements. See "Requirements for Project Creation from Build Systems" on page 8-8.

Specify Options Explicitly

If you cannot trace your build command and therefore manually create a project, you have to specify the options explicitly.

- In the Polyspace user interface, select a project configuration. On the **Configuration** pane, select **Target & Compiler**. Specify the options.
- At the DOS or UNIX command line, specify flags with the `polyspace-code-prover-nodesktop` command.
- At the MATLAB command line, specify arguments with the `polyspaceCodeProver` function.

Specify the options in this order.

- Required options:
  - `Source code language (-lang)`: If all files have the same extension `.c` or `.cpp`, the verification uses the extension to determine the source code language. Otherwise, explicitly specify the option.
  - `Compiler (-compiler)`: Select the compiler that you use for building your source code. If you cannot find your compiler, use an option that closely matches your compiler.
  - `Target processor type (-target)`: Specify the target processor on which you intend to execute your code. For some processors, you can change the default specifications. For instance, for the processor `hc08`, you can change the size of types `double` and `long double` from 32 to 64 bits.

    If you cannot find your target processor, you can create your own target and specify the sizes of fundamental types, default signedness of `char`, and endianness of the target machine. See `Generic target options`.
- Language-specific options:
  - `C++11 extensions (-cpp11-extension)`: Select this option if you use C++11 extensions. See also “Supported C++ 2011 Language Extensions” on page 8-15.
  - `Respect C90 standard (-no-language-extensions)`: Select this option if you prefer that the verification use the C90 Standard (ISO/IEC 9899:1990). Otherwise, the verification uses the ANSI® C99 Standard (ISO®/IEC 9899:1999) for compilation and checking of certain coding rules.
- Compiler-specific options:

Whether these options are available or not depends on your specification for `Compiler (-compiler)`. For instance, if you select a `visual` compiler, the option `Pack alignment value (-pack-alignment-value)` is available. Using the option, you emulate the compiler option `/Zp` that you use in Visual Studio.

For all compiler-specific options, see “Target and Compiler”.

- Advanced options:

  Using these options, you can modify the verification results. For instance, if you use the option `Division round down (-div-round-down)`, the verification considers that quotients from division or modulus of negative numbers are rounded down. Use these options only if you use similar options when compiling your code.

  For all advanced options, see “Target and Compiler”.
- Compiler header files:

  If you specify the `diab`, `tasking` or `greenhills` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis” on page 8-6.

If you still see compilation errors after running analysis, you might have to specify other options:

- *Define macros*: Sometimes, a compilation error occurs because the analysis considers a macro as undefined. Explicitly define these macros. See “Macros”.
- *Specify include files*: Sometimes, a compilation error occurs because your compiler defines standard library functions differently from Polyspace and you do not provide your compiler include files. Explicitly specify the path to your compiler include files. See “Errors from Conflicts with Polyspace Header Files” on page 19-53.

See Also

More About

- “Language Extensions Supported by Default” on page 8-11
- “Supported Keil or IAR Language Extensions” on page 8-13
- “Supported C++ 2011 Language Extensions” on page 8-15

Provide Standard Library Headers for Polyspace Analysis

Before Polyspace analyzes the code for bugs and run-time errors, it compiles your code. Even if the code compiles with your compiler, you can see compilation errors with Polyspace. If the error comes from a standard library function, it usually indicates that Polyspace is not using your compiler headers. To work around the errors, provide the path to your compiler headers.

This topic shows how to locate the standard library headers from your compiler. The code examples cause a compilation error that shows the location of the headers.

- To locate the folder containing your C compiler system headers, compile this C code by using your compilation toolchain:

  ```
  float fopen(float f);
  #include <stdio.h>
  ```

  The code does not compile because the `fopen` declaration conflicts with the declaration inside `stdio.h`. The compilation error shows the location of your compiler implementation of `stdio.h`. Your C standard library headers are all likely to be in that folder.
- To locate the folder containing your C++ compiler system headers, compile this C++ code by using your compilation toolchain:

  ```
  namespace std {
      float cin;
  }
  #include <iostream>
  ```

  The code does not compile because the `cin` declaration conflicts with the declaration inside `iostream.h`. The compilation error shows the location of your compiler implementation of `iostream.h`. Your C++ standard library headers are all likely to be in that folder.

After you locate the path to your compiler's header files, specify the path for the Polyspace analysis. For C++ code, specify the paths to both your C and C++ headers.

- In the user interface, add the folder to your project.

  For more information, see “Add Source Files for Analysis in Polyspace User Interface” on page 1-2.

- At the command line, use the flag `-I` with the `polyspace-bug-finder-nodesktop` command.

  For more information, see `-I`.

Requirements for Project Creation from Build Systems

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

Compiler Requirements

- 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:

  - Clang
  - Wind River® Diab
  - GNU C
  - IAR Embedded Workbench
  - Green Hills®
  - NXP CodeWarrior®
  - Altium® Tasking
  - Texas Instruments™
  - Tiny C
  - Microsoft® Visual C++®

  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 19-9.

  - Contact MathWorks Technical Support. For more information, see “Contact Technical Support” on page 19-7.
- With the TASKING compiler, if you use an alternative sfr file with extension `.asfr`, Polyspace might not be able to locate your file. If you encounter an error, explicitly `#include` your `.asfr` file in the preprocessed code using the option `Include (-include)`.

  Typically, you use the statement `#include __SFRFILE__(__CPU__)` along with the compiler option `--alternative-sfr-file` to specify an alternative sfr file. The path to the file is typically `Tasking_C166_INSTALL_DIR\include\sfr\regCPUNAME.asfr`. For instance, if your TASKING compiler is installed in `C:\Program Files\Tasking\C166-VX_v4.0r1\` and you use the CPU-related flag `-Cxc2287m_104f` or `--cpu=xc2287m_104f`, the path is `C:\Program Files\Tasking\C166-VX_v4.0r1\include\sfr\regxc2287m.asfr`.

Build Command Requirements

- Your build command must run to completion without any user interaction.
- 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 “Check if Polyspace Supports Build Scripts” on page 19-20.
- If you use statically linked libraries, Polyspace cannot trace your build. In Linux, you can install the full Linux Standard Base (LSB) package to allow dynamic linking. For example, on Debian® systems, install LSB with the command `apt-get install lsb`.
- 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 process must not use the `LD_PRELOAD` mechanism.
- 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 log in 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 the System Integrity Protection (SIP) feature is active on the operating system macOS El Capitan (10.11) or a later macOS version, Polyspace cannot trace your build command. Before tracing your build command, disable the SIP feature. You can reenable this feature after tracing the build command.
- 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

- “Add Source Files for Analysis in Polyspace User Interface” on page 1-2

More About

- “Slow Build Process When Polyspace Traces the Build” on page 19-19

Language Extensions Supported by Default

Polyspace analysis can interpret a subset of common C/C++ language constructs and extended keywords by default. The default analysis follows these standards:

- C language: C99 Standard (ISO/IEC 9899:1999)

  If you select `Respect C90 standard (-no-language-extensions)`, the analysis follows the C90 Standard.
- C++ language: C++03 Standard (ISO/IEC 14882:2003)

  If you select `C++11 extensions (-cpp11-extension)`, the analysis allows C++11 extensions.

In addition, the default analysis can also interpret language extensions that are supported by many compilers. For other compiler-specific constructs, explicitly specify your compiler.

The analysis can interpret the following constructs, irrespective of your choice of compiler.

- Designated initializers (labeling initialized elements)
- Compound literals (structs or arrays as values)
- Boolean type (`_Bool`)
- Statement expressions (statements and declarations inside expressions)
- `typeof` constructs
- Case ranges
- Empty structures
- Cast to union
- Local labels (`__label__`)
- Hexadecimal floating-point constants
- Extended keywords, operators, and identifiers (`_Pragma`, `__func__`, `__const__`, `__asm__`)

The list is not complete.

In some cases, the analysis supports the construct semantically and fully emulates its run-time behavior. In other cases, the analysis only supports the construct syntactically, but

does not emulate its run-time behavior fully. For instance, the analysis recognizes the construct `asm` as introduction of assembly code, but does not interpret the assembly code encapsulated in the construct. As a result, values modified by the assembly code are considered to have all possible values allowed by their data type.

See Also

Related Examples

- "Specify Target Environment and Compiler Behavior" on page 8-2

More About

- "Supported Keil or IAR Language Extensions" on page 8-13
- "Supported C++ 2011 Language Extensions" on page 8-15

Supported Keil or IAR Language Extensions

Polyspace analysis can interpret a subset of common C/C++ language constructs and extended keywords by default. For compiler-specific keywords, you must specify your choice of compiler. If you specify `keil` or `iar` for `Compiler (-compiler)`, the Polyspace verification allows language extensions specific to the Keil or IAR compilers.

Special Function Register Data Type

Embedded control applications frequently read and write port data, set timer registers, and read input captures. To deal with these requirements without using assembly language, some microprocessor compilers define special data types such as `sfr` and `sbit`. Typical declarations are:

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

The declarations reside in header files such as `regxx.h` for the basic `80Cxxx` micro processor. The declarations customize the compiler to the target processor.

You access a register or a port by using the `sfr` and `sbit` data as follows. However, these data types are not part of the C99 Standard.

```
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 all interrupts) */
}
```

To analyze this type of code, use these options:

- `Compiler (-compiler)`: Specify `keil` or `iar`.
- `Sfr type support (-sfr-types)`: Specify the data type and size in bits.

The analysis then supports the Keil or IAR language extensions even if some structures, keywords, and syntax are not part of the C99 standard.

Keywords Removed During Preprocessing

Once you specify the Keil or IAR compiler, the analysis recognizes compiler-specific keywords in your code. If a keyword is not relevant for the analysis, it is removed from the source code during preprocessing.

If you disable the keyword and use it as an identifier instead, you can encounter a compilation error when you compile your code with Polyspace. See “Errors Related to Keil or IAR Compiler” on page 19-41.

These keywords are removed during preprocessing:

- Keil: `bdata`, `far`, `idata`, `huge`, `sdata`
- IAR: `saddr`, `reentrant`, `reentrant_idata`, `non_banked`, `plm`, `bdata`, `idata`, `pdata`, `code`, `xdata`, `xhuge`, `interrupt`, `__interrupt`, `__intrinsic`

  The `data` keyword is not removed.

Supported C++ 2011 Language Extensions

This table lists 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 *this	Yes
C++2011-N1610	Initialization of class objects by rvalues	Yes
C++2011-N2756	Nonstatic 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
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

Standard	Description	Supported
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
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	No
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
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

Standard	Description	Supported
C++2011-N2179	Concurrency: Propagating exceptions	No
C++2011-N2440	Concurrency: Abandoning a process and at_quick_exit	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	__func__ predefined identifier	Yes
C++2011-N1653	C99 preprocessor	Yes
C++2011-N1811	long long	Yes
C++2011-N1988	Extended integral types	No

See Also

`C++11 extensions (-cpp11-extension)`

Remove or Replace Keywords Before Compilation

The Polyspace compiler strictly follows the ANSI C99 Standard (ISO/IEC 9899:1999). If your compiler allows deviation from the Standard, the Polyspace compilation using default options cannot emulate your compiler. For instance, your compiler can allow certain non-ANSI keyword, which Polyspace does not recognize by default.

To emulate your compiler closely, you specify the Target & Compiler options. If you still get compilation errors from unrecognized keywords, you can remove or replace them only for the purposes of verification. The option `Preprocessor definitions (-D)` allows you to make simple substitutions. For complex substitutions, for instance to remove a group of space-separated keywords such as a function attribute, use the option `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

Remove Unrecognized Keywords

You can remove unsupported keywords from your code for the purposes of analysis. For instance, follow these steps to remove the `far` and `0x` keyword from your code (`0x` precedes an absolute address).

1 Save the following template as `C:\Polyspace\myTpl.pl`.

Content of myTpl.pl

```
#!/usr/bin/perl

##############################################################
# Post Processing template script
#
##############################################################
# Usage from GUI:
#
# 1) Linux: /usr/bin/perl PostProcessingTemplate.pl
# 2) Windows: matlabroot\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;

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

For reference, see a summary of Perl regular expressions.

Perl Regular Expressions

```
#########################################################
# 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 All of abc respectively
# 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/;
#
##########################################################
```

2. On the **Configuration** pane, select **Environment Settings**.
3. To the right of **Command/script to apply to preprocessed files**, click .
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.

Remove Unrecognized Function Attributes

You can remove unsupported function attributes from your code for the purposes of analysis.

If you run verification on this code specifying a generic compiler, you can see compilation errors from the `noreturn` attribute. The code compiles using a GNU compiler.

```
void fatal () __attribute__ ((noreturn));

void fatal (/* ... */)
{
    /* ... */ /* Print error message. */ /* ... */
    exit (1);
}
```

If the software does not recognize an attribute and the attribute does not affect the code analysis, you can remove it from your code for the purposes of verification. For instance, you can use this Perl script to remove the `noreturn` attribute.

```
while ($line = <STDIN>)
{

# __attribute__ ((noreturn))

  # Remove far keyword
  $line =~ s/__attribute__\ \(\(noreturn\)\)//g;

  # Print the current processed line to STDOUT
  print $line;
}
```

Specify the script using the option `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

See Also

Polyspace Analysis Options
`Command/script to apply to preprocessed files (-post-preprocessing-command)` | `Compiler (-compiler)` | `Preprocessor definitions (-D)`

Related Examples

- "Troubleshooting in Polyspace Bug Finder"

Gather Compilation Options Efficiently

Polyspace verification can sometimes stop in the compilation or linking phase due to the following reasons:

- The Polyspace compiler strictly follows the ANSI C99 Standard (ISO/IEC 9899:1999). If your compiler allows deviation from the Standard, the Polyspace compilation using default options cannot emulate your compiler.
- Your compiler declares standard library functions with argument or return types different from the standard types. Unless you also provide the function definition, for efficient verification, Polyspace uses its own definitions of standard library functions, which have the usual prototype. The mismatch in types causes a linking error.

You can easily work around the compilation and standard library function errors. To work around the errors, you typically specify certain analysis options. In some cases, you might have to add a few lines to your code. For instance:

- To emulate your compiler behavior more closely, you specify the Target & Compiler options. If you still face compilation errors, you might have to remove or replace certain unrecognized keywords using the option `Preprocessor definitions (-D)`. However, the option allows only simple substitution of a string with another string. For more complex replacements, you might have to add `#define` statements to your code.
- To avoid errors from stubbing standard library functions, you might have to `#define` certain Polyspace-specific macros so that Polyspace does not use its own definition of standard library functions.

For more information, see "Troubleshooting in Polyspace Bug Finder".

Instead of adding these modifications to your original code, create a single `polyspace.h` file that contains all modifications. Use the option `Include (-include)` to force inclusion of the `polyspace.h` file in all source files under verification.

Benefits of this approach include:

- The error detection is much faster since it will be detected during compilation rather than in the link or subsequent phases.
- There will be no need to modify original source files.
- The file is automatically included as the very first file in the original `.c` files.

- The file is reusable for other projects developed under the same environment.

Example 8.1. Example

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

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

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

// Workarounds for compilation errors
#define far
#define at(x)

// Workarounds for errors due to redefining standard library functions

#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);
```

Configure Inputs and Stubbing Options

Specify External Constraints

This example shows how to specify constraints (also known data range specifications or DRS) on variables in your code. Polyspace uses the code that you provide to make assumptions about items such as variable ranges and allowed buffer size for pointers. Sometimes the assumptions are broader than what you expect because:

- You have not provided the complete code. For example, you did not provide 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.

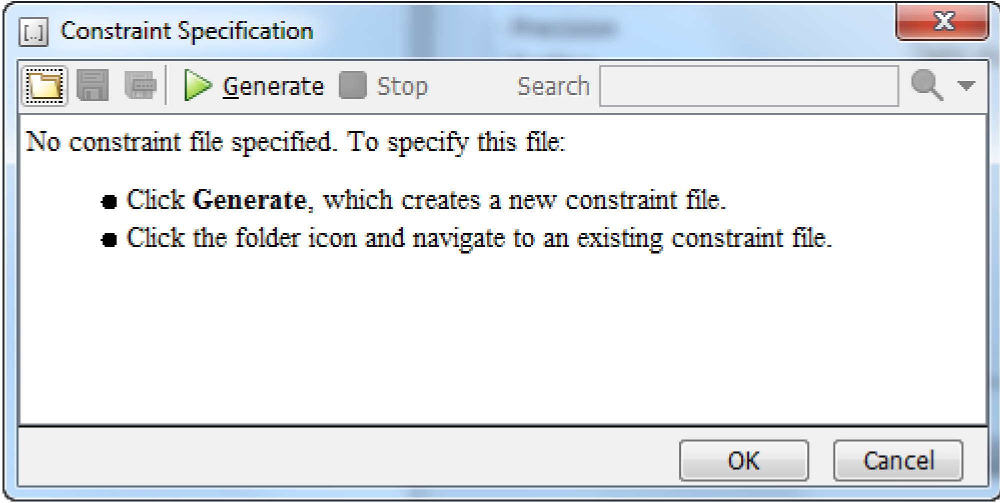
Because of these broad assumptions, Polyspace can sometimes produce false positives.

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 to use them for subsequent analyses. If your source code changes, you can update the previous constraints. You do not have to create a new constraint template.

Note In Bug Finder, you can only constrain global variables. You cannot constrain function inputs or return values of stubbed functions.

Create Constraint Template

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



3. In the Constraint Specification dialog box, create a blank constraint template. The template contains a list of all variables on which you can provide constraints.

   - If you have run analysis once and not changed your code since that analysis, instead of generating a new constraint template, use the folder icon to navigate to the previous results folder. Open the template file `drs_template.xml` from that folder. Save the file in another location, in case you delete the previous results folder.
   - Otherwise, to create a new template, click Generate. The software compiles your project and creates a template. The new template is stored in a file *`Module_number_Project_name`*`_drs_template.xml` in your project folder.

4. Specify your constraints and save the template as an XML file. For more information, see "External Constraints for Polyspace Analysis" on page 9-6.
5. Click **OK**.

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

Update Existing Template

If you remove some variables or functions from your code, constraints on them are not applicable any more. Instead of regenerating a constraint template and respecifying the

constraints, you can update an existing template and remove the variables that are not present in your code.

1 On the **Configuration** pane, select **Inputs & Stubbing**.
2 Open the existing template in one of the following ways:

   - In the **Constraint setup** field, enter the path to the template XML file. 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.

Specify Constraints in Code

Specifying constraints outside your code allows for more precise analysis. However, you must use the code within the specified constraints because the constraints are *outside* your code. Otherwise, the 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 *inside* your code, you can use:

- Appropriate error handling tests in your code.

  Polyspace checks to determine 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. Using `assert` statements, you can constrain your variables for the remaining code in the same scope. For examples, see `Assertion`.

See Also

`Constraint setup (-data-range-specifications)`

Related Examples

- “External Constraints for Polyspace Analysis” on page 9-6
- “XML File Format for Constraints” on page 9-11

External Constraints for Polyspace Analysis

The Polyspace Constraint Specification interface allows you to specify constraints for:

- Global Variables.
- User-defined Functions.

  You cannot constrain user-defined functions in Bug Finder.
- Stubbed Functions.

  You cannot constrain stubbed functions in Bug Finder.

For more information, see “Specify External Constraints” on page 9-2.

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

Column	Settings
Name	Displays the list of variables and functions in your Project for which you can specify data ranges. This Column displays three expandable menu items: • **Globals** – Displays 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	Displays information about the variable or function. For example, static variables display `static`.
Data Type	Displays the variable type.

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

<table>
<tr><th>Column</th><th>Settings</th></tr>
<tr><td>Init Range</td><td>Specifies the minimum and maximum values for the variable.

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 long, <code>min</code> and <code>max</code> correspond to -2^31 and 2^31-1 respectively.

You can also use hexadecimal values. For example: <code>0x12..0x100</code>

For <code>enum</code> variables, you cannot specify ranges directly using the enumerator constants. Instead use the values represented by the constants.

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:

<pre>enum week{ sunday, monday=0, tuesday,
     wednesday, thursday, friday, saturday};</pre></td></tr>
<tr><td>Initialize Pointer</td><td>Applicable only to pointers. Enabled only when you specify Init Mode:<code>INIT</code>.

Specifies whether the pointer should be NULL:
<code>May-be NULL</code> – The pointer could potentially be a NULL pointer (or not).
<code>Not Null</code> – The pointer is never initialized as a null pointer.
<code>Null</code> – The pointer is initialized as NULL.
Note Not applicable for C++ projects.</td></tr>
</table>

<table>
<tr><th>Column</th><th>Settings</th></tr>
<tr><td>Init Allocated</td><td>Applicable only to pointers. Enabled only when you specify Init Mode:<code>INIT</code>.

Specifies how the pointed object is allocated:

• <code>MAIN GENERATOR</code> – The pointed object is allocated by the main generator.
• <code>None</code> – Pointed object is not written.
• <code>SINGLE</code> – Write the pointed object or the first element of an array. (This setting is useful for stubbed function parameters.)
• <code>MULTI</code> – All objects (or array elements) are initialized.

Note Not applicable for C++ projects.</td></tr>
<tr><td># Allocated Objects</td><td>Applicable only to pointers.

Specifies how many objects are pointed to by the pointer (the pointed object is considered as an array).

Note: The Init Allocated parameter specifies how many allocated objects are actually initialized.

Note Not applicable for C++ projects.</td></tr>
<tr><td>Global Assert</td><td>Specifies whether to perform an assert check on the variable at global initialization, and after each assignment.</td></tr>
<tr><td>Global Assert Range</td><td>Specifies the minimum and maximum values for the range you want to check.</td></tr>
<tr><td>Comment</td><td>Remarks that you enter, for example, justification for your DRS values.</td></tr>
</table>

See Also

More About

- “Specify External Constraints” on page 9-2

XML File Format for Constraints

If you run a verification, the software automatically generates a constraint file `drs-template.xml` in your results folder. Edit this XML file to specify your constraints.

You can also see the information in this topic and the underlying XML schema in *`matlabroot`*`\polyspace\drs`. Here, *`matlabroot`* is the MATLAB installation folder, for instance, `C:\Program Files\MATLAB\R2017a`.

Note Instead of editing the constraint XML file directly, use the Polyspace user interface to specify your constraints and save the constraints as an XML file. For more information, see “Specify External Constraints” on page 9-2.

Syntax Description — XML Elements

The DRS file contains the following XML elements:

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

Arguments should be named *arg1, arg2, …argn* and the return value should be called *return*.

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

- **(*)** — Fields used only by the GUI. These fields are not mandatory for verification to accept the ranges. The field line contains the line number where the variable is declared in the source code, `complete_type` contains a string with the complete variable type, and `base_type` is used by the GUI to compute the min and max values. The field comment is used to add information about any node.
- **(**)** — The field name is mandatory for scope elements `<file>` and `<function>` (except for function pointers). For other elements, the name must be specified when declaring a root symbol or a `struct` field.
- **(***)** — If more than one attribute applies to the variable, the attributes must be separated by a space. Only the static attribute is mandatory, to avoid conflicts between static variables having the same name. An attribute can be defined multiple times without impact.
- **(****)** — This element is used only by the GUI, to determine which `init` modes are allowed for the current element (according to its type). The value works as a mask, where the following values are added to specify which modes are allowed:

  - **1**: The mode “`NO`” is allowed.
  - **2** : The mode “`INIT`” is allowed.
  - **4**: The mode “`PERMANENT`” is allowed.
  - **8**: The mode “`MAIN_GENERATOR`” is allowed.

  For example, the value “**10**” means that modes “`INIT`” and “`MAIN_GENERATOR`” are allowed. To see how this value is computed, refer to “Valid Modes and Default Values” on page 9-16.
- **(*****)** — A sub-element of a pointer (i.e. a pointed object) will be taken into account only if `init_pointed` is equal to SINGLE, MULTI, SINGLE_CERTAIN_WRITE or MULTI_CERTAIN_WRITE.
- **(******)** — SINGLE_CERTAIN_WRITE or MULTI_CERTAIN_WRITE are available for parameters and return values of stubbed functions only if they are pointers. If the parameter or return value is a structure and the structure has a pointer field, they are also available for the pointer field.

<file> Element

Field	Syntax
name	*filepath_or_filename*
comment	*string*

<scalar> Element

Field	Syntax
name (**)	*name*
line (*)	*line*
base_type (*)	intx uintx floatx
Attributes (***)	volatile extern static const
complete_type (*)	*type*
init_mode	MAIN_GENERATOR IGNORE INIT PERMANENT disabled unsupported
init_modes_allowed (*)	*single value* (****)
init_range	*range* disabled unsupported
global_ assert	YES NO disabled unsupported

Field	Syntax
assert_range	*range* disabled unsupported
comment(*)	*string*

<pointer> Element

Field	Syntax
Name (**)	*name*
line (*)	*line*
Attributes (***)	volatile extern static const
complete_type (*)	*type*
init_mode	MAIN_GENERATOR IGNORE INIT PERMANENT disabled unsupported
init_modes_allowed (*)	*single value* (****)
initialize_ pointer	May be: NULL Not NULL NULL
number_ allocated	*single value* disabled unsupported

Field	Syntax
init_pointed (******)	MAIN_GENERATOR NONE SINGLE MULTI SINGLE_CERTAIN_WRITE MULTI_CERTAIN_WRITE disabled
comment	*string*

<array> and <struct> Elements

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

<function> Element

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

Field	Syntax
main_generator_called	MAIN_GENERATOR YES NO disabled
attributes (***)	static extern unused
comment	*string*

Valid Modes and Default Values

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
Global variables	Base type	Unqualified/ static/ const scalar	MAIN_ GENERATOR IGNORE INIT PERMANENT	YES NO			Main generator dependant
		Volatile scalar	PERMANENT	disabled			PERMANENT min..max
		Extern scalar	INIT PERMANENT	YES NO			INIT min..max
	Struct	Struct field	Refer to field type				
	Array	Array element	Refer to element type				
Global variables	Pointer	Unqualified/ static/ const scalar	MAIN_ GENERATOR IGNORE INIT		May be NULL Not NULL NULL	NONE SINGLE MULTI	Main generator dependant
		Volatile pointer	un-supported		un-supported	un-supported	

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
		Extern pointer	IGNORE INIT		May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI
		Pointed volatile scalar	un-supported	un-supported			
		Pointed extern scalar	INIT	un-supported			INIT min..max
		Pointed other scalars	MAIN_ GENERATOR INIT	un-supported			MAIN_ GENERATOR dependant
		Pointed pointer	MAIN_ GENERATOR INIT/	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	MAIN_ GENERATOR dependant
		Pointed function	un-supported	un-supported			
Function parameters	Userdef function	Scalar parameters	MAIN_ GENERATOR INIT	un-supported			INIT min..max
		Pointer parameters	MAIN_ GENERATOR INIT	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI
		Other parameters	Refer to parameter type				
	Stubbed function	Scalar parameter	disabled	un-supported			

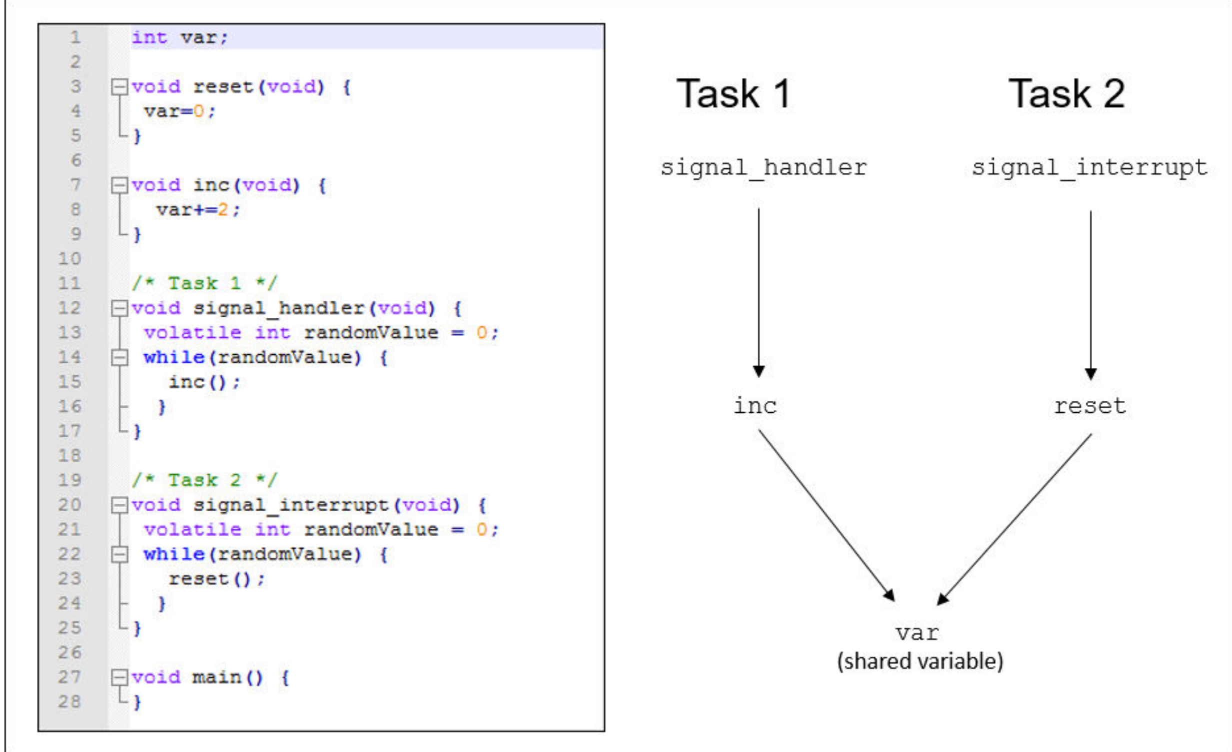
Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
		Pointer parameters	disabled		disabled	NONE SINGLE MULTI SINGLE_CERTAIN_WRITE MULTI_CERTAIN_WRITE	MULTI
		Pointed parameters	PERMANENT	un-supported			PERMANENT min..max
		Pointed const parameters	disabled	un-supported			
Function return	Userdef function	Return	disabled	un-supported	disabled	disabled	
	Stubbed function	Scalar return	PERMANENT	un-supported			PERMANENT min..max

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
		Pointer return	PERMANENT	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI SINGLE_CERTAIN_WRITE MULTI_CERTAIN_WRITE	PERMANENT May be NULL max MULTI

Configure Multitasking Analysis

Analyze Multitasking Programs in Polyspace

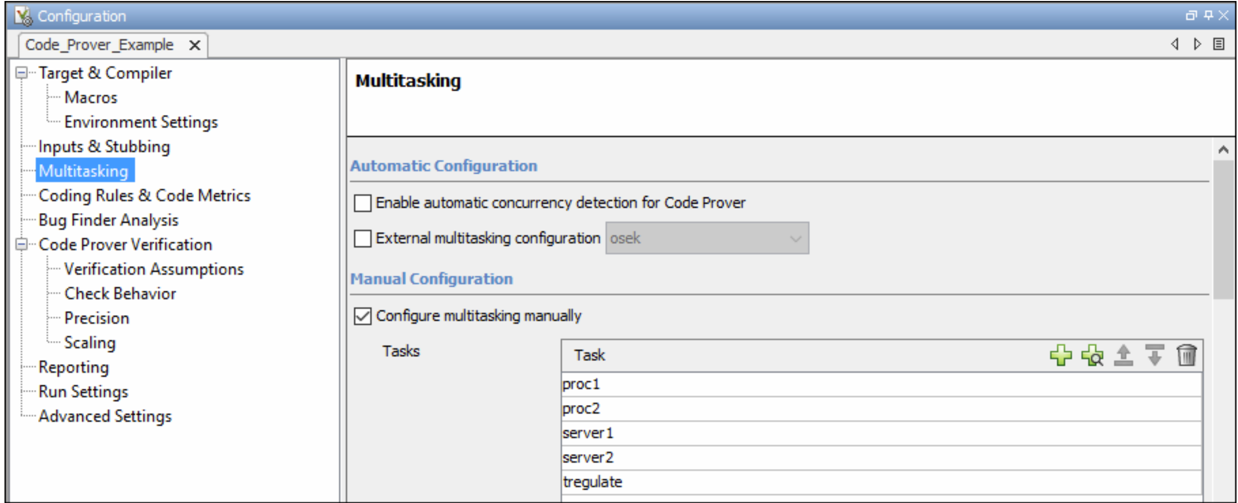
With Polyspace, you can analyze programs where multiple threads (tasks) run concurrently.



In addition to regular run-time checks, the analysis looks for issues specific to concurrent execution:

- Data races, deadlocks, consecutive or missing locks and unlocks (Bug Finder)
- Unprotected shared variables (Code Prover)

Configure Analysis



If your code uses multitasking primitives from certain families, for instance, `pthread_create` for thread creation:

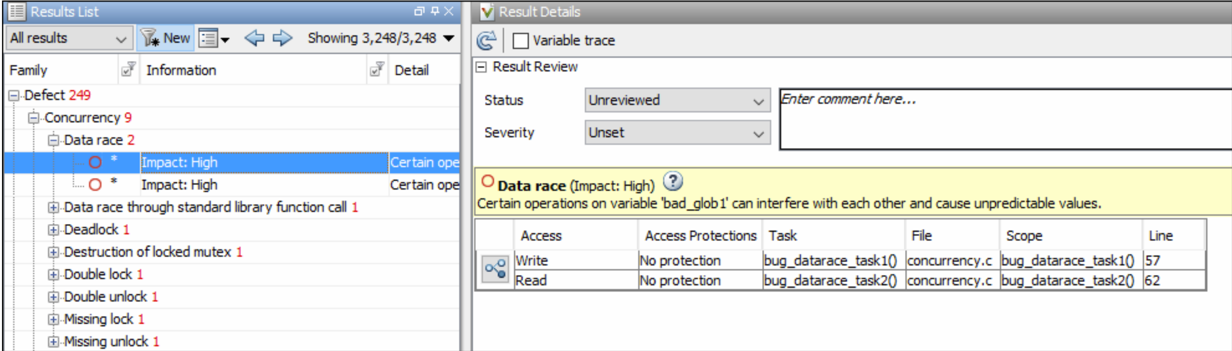
- In Bug Finder, the analysis detects them and extracts your multitasking model from the code.
- In Code Prover, you must enable this automatic detection explicitly.

See "Auto-Detection of Thread Creation and Critical Section in Polyspace" on page 10-6.

Alternatively, define your multitasking model through the analysis options. In the Polyspace user interface, the options are on the **Multitasking** node in the **Configuration** pane. For more information, see "Configuring Polyspace Multitasking Analysis Manually" on page 10-14.

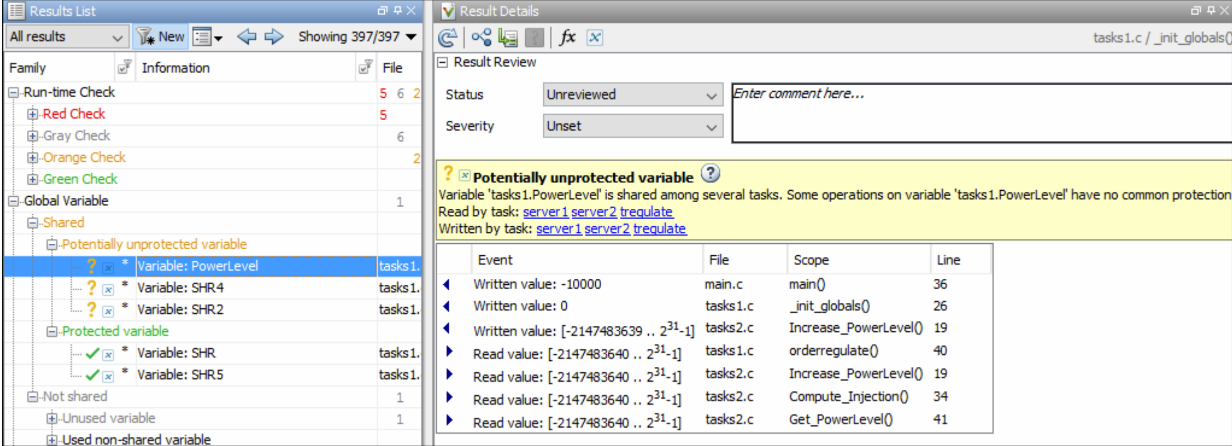
Review Analysis Results

Bug Finder



The Bug Finder analysis shows concurrency defects such as data races and deadlocks. See “Concurrency Defects”.

Code Prover



The Code Prover analysis exhaustively checks if shared global variables are protected from concurrent access. See “Global Variables” (Polyspace Code Prover).

Review the results using the message on the **Result Details** pane. See a visual representation of conflicting operations using the (graph) icon.

See Also

More About

- “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 10-6
- “Configuring Polyspace Multitasking Analysis Manually” on page 10-14
- “Protections for Shared Variables in Multitasking Code” on page 10-19

Auto-Detection of Thread Creation and Critical Section in Polyspace

With Polyspace, you can analyze programs where multiple threads run concurrently. Polyspace can analyze your multitasking code for data races, deadlocks and other concurrency defects, if the analysis is aware of the concurrency model in your code. In some situations, Polyspace can detect thread creation and critical sections in your code automatically. Bug Finder detects them by default. In Code Prover, you enable automatic detection using the option `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

For the multitasking code analysis workflow, see “Analyze Multitasking Programs in Polyspace” on page 10-2.

If your thread creation function is not detected automatically:

- You can also map the function to a thread-creation function that Polyspace can detect automatically. Use the option `-function-behavior-specifications`.
- Otherwise, you must manually model your multitasking threads by using configuration options. See “Configuring Polyspace Multitasking Analysis Manually” on page 10-14.

Multitasking Routines that Polyspace Can Detect

Polyspace can detect thread creation and critical sections if you use primitives from these groups. Polyspace recognizes calls to these routines as the creation of a new thread or as the beginning or end of a critical section.

POSIX

Thread creation: `pthread_create`

Critical section begins: `pthread_mutex_lock`

Critical section ends: `pthread_mutex_unlock`

VxWorks

Thread creation: `taskSpawn`

Critical section begins: `semTake`

Critical section ends: `semGive`

To activate automatic detection of concurrency primitives for VxWorks®, use the `VxWorks` template. For more information on templates, see “Create Project Using Configuration Template” on page 1-19.

Concurrency detection is possible only if the multitasking functions are created from an entry point named `main`. If the entry point has a different name, such as `vxworks_entry_point`, do one of the following:

- Provide a `main` function.
- `Preprocessor definitions (-D)`: In preprocessor definitions, set `vxworks_entry_point=main`.

Windows

Thread creation: `CreateThread`

Critical section begins: `EnterCriticalSection`

Critical section ends: `LeaveCriticalSection`

µC/OS II

Thread creation: `OSTaskCreate`

Critical section begins: `OSMutexPend`

Critical section ends: `OSMutexPost`

C++11

Thread creation: `std::thread::thread`

Critical section begins: `std::mutex::lock`

Critical section ends: `std::mutex::unlock`

For autodetection of C++11 threads, explicitly specify paths to your compiler header files or use `polyspace-configure`.

For instance, if you use `std::thread` for thread creation, explicitly specify the path to the folder containing `thread.h`.

Example of Automatic Thread Detection

The following multitasking code models five philosophers sharing five forks. The example uses POSIX® thread creation routines and illustrates a classic example of a deadlock. Run Bug Finder on this code to see the deadlock.

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

pthread_mutex_t forks[5];

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,philo1,NULL);
  pthread_create(&ph[1],NULL,philo2,NULL);
  pthread_create(&ph[2],NULL,philo3,NULL);
  pthread_create(&ph[3],NULL,philo4,NULL);
  pthread_create(&ph[4],NULL,philo5,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 `philo1`, `philo2`, `philo3`, `philo4`, and `philo5` 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.

Naming Convention for Automatically Detected Threads

If you use a function such as `pthread_create()` to create new threads (tasks), each thread is associated with an unique identifier. For instance, in this example, two threads are created with identifiers `id1` and `id2`.

```
pthread_t* id1,id2;

void main()
{
  pthread_create(id1,NULL,start_routine,NULL);
  pthread_create(id2,NULL,start_routine,NULL);
}
```

If a data race occurs between the threads, the analysis can detect it. When displaying the results, the threads are indicated as `task_`*`id`*, where `id` is the identifier associated with the thread. In the preceding example, the threads are identified as `task_id1` and `task_id2`.

If a thread identifiers is:

- Local to a function, the thread name shows the function.

  For instance, the thread created below appears as `task_f:id`

  ```
  void f (void) {
    pthread_t* id;
    pthread_create(id,NULL,start_routine,NULL);
  }
  ```
- A field of a structure, the thread name shows the structure.

  For instance, the thread created below appears as `task_a#id`

```
struct {pthread_t* id; int x;} a;
pthread_create(a.id,NULL,start_routine,NULL);
```

- An array member, the thread name shows the array.

  For instance, the thread created below appears as `task_tab[1]`.

  ```
  pthread_t* tab[10];
  pthread_create(tab[1],NULL,start_routine,NULL);
  ```

Limitations of Automatic Thread Detection

The multitasking model extracted by Polyspace does not include some features. 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.
- (Bug Finder only) Creation of multiple threads through multiple calls to the same function with different pointer arguments.

  For instance, in this example, Polyspace considers that only one thread is created.

  ```
  pthread_t id1,id2;
  void start(pthread_t* id)
  {
     pthread_create(id,NULL,start_routine,NULL);
  }
  void main()
  {
    start(&id1);
    start(&id2);
  }
  ```

See Also

`Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`

More About

- "Analyze Multitasking Programs in Polyspace" on page 10-2
- "Configuring Polyspace Multitasking Analysis Manually" on page 10-14

Configuring Polyspace Multitasking Analysis Manually

With Polyspace, you can analyze programs where multiple threads run concurrently. In some situations, Polyspace can detect thread creation and critical sections in your code automatically. See “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 10-6.

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.

For the multitasking code analysis workflow, see “Analyze Multitasking Programs in Polyspace” on page 10-2.

Specify Options for Multitasking Analysis

Use these options to specify cyclic tasks, interrupts and protections for shared variables. In the Polyspace user interface, the options are on the **Multitasking** node in the **Configuration** pane.

- `Tasks (-entry-points)`: Specify noncyclic entry point functions.

  Do not specify `main`. Polyspace implicitly considers `main` as an entry point function.
- `Cyclic tasks (-cyclic-tasks)`: Specify functions that are scheduled at periodic intervals.
- `Interrupts (-interrupts)`: Specify functions that can run asynchronously.
- `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`: Specify functions that disable and reenable interrupts (Bug Finder only).
- `Critical section details (-critical-section-begin -critical-section-end)`: Specify functions that begin and end critical sections.
- `Temporally exclusive tasks (-temporal-exclusions-file)`: Specify groups of functions that are temporally exclusive.
- `-preemptable-interrupts`: Specify functions that have lower priority than interrupts, but higher priority than tasks (preemptable or non-preemptable).

  Only the Bug Finder analysis considers priorities.

- `-non-preemptable-tasks`: Specify functions that have higher priority than tasks, but lower priority than interrupts (preemptable or non-preemptable).

  Only the Bug Finder analysis considers priorities.

Adapt Code for Code Prover Multitasking Analysis

The multitasking analysis in Code Prover is more exhaustive about finding potentially unprotected shared variables and therefore follows a strict model.

Tasks and interrupts must be void-void functions.

Functions that you specify as tasks and interrupts must have the prototype:

```
void func(void);
```

Suppose you want to specify a function `func` that takes `int` arguments:

```
void func(int);
```

Define a wrapper void-void function that calls `func` with a volatile value. Specify this wrapper function as a task or interrupt.

```
void func_wrapper() {
  volatile int arg;
  func(arg);
}
```

The `main` function must end.

Code Prover assumes that the `main` function ends before all tasks and interrupts begin. If the `main` function contains an infinite loop or run-time error, the tasks and interrupts are not analyzed.

Suppose you want to specify the `main` function as a cyclic task.

```
void performTask1Cycle(void);
void performTask2Cycle(void);

void main() {
```

```
 while(1) {
    performTask1Cycle();
  }
}

void task2() {
 while(1) {
    performTask2Cycle();
  }
}
```

Replace the definition of `main` with:

```
#ifdef POLYSPACE
void main() {
}
void task1() {
 while(1) {
    performTask1Cycle();
  }
}

#else
void main() {
 while(1) {
    performTask1Cycle();
  }
}
#endif
```

The replacement defines an empty `main` and places the content of `main` into another function `task1` if a macro `POLYSPACE` is defined. Define the macro `POLYSPACE` using the option `Preprocessor definitions (-D)`.

This assumption does not apply to automatically detected threads. For instance, a `main` function can create threads using `pthread_create`.

All tasks and interrupts can interrupt each other.

The Bug Finder analysis considers priorities of tasks. A function that you specify as a task cannot interrupt a function that you specify as an interrupt because an interrupt has higher priority.

The Code Prover analysis considers that all tasks and interrupts can interrupt each other.

All tasks and interrupts can run any number of times in any sequence.

The Code Prover analysis considers that all tasks and interrupts can run any number of times in any sequence.

Suppose in this example, you specify `reset` and `inc` as cyclic tasks. The analysis shows an overflow on the operation `var+=2`.

```
void reset(void) {
 var=0;
}

void inc(void) {
  var+=2;
}
```

Suppose you want to model a scheduling of tasks such that `reset` executes after `inc` has executed five times. Write a wrapper function that implements this sequence. Specify this new function as a cyclic task instead of `reset` and `inc`.

```
void task() {
 volatile int randomValue = 0;
 while(randomValue) {
   inc();
   inc();
   inc();
   inc();
   inc();
   reset();
   }
 }
```

Suppose you want to model a scheduling of tasks such that `reset` executes after `inc` has executed zero to five times. Write a wrapper function that implements this sequence. Specify this new function as a cyclic task instead of `reset` and `inc`.

```
void task() {
 volatile int randomValue = 0;
 while(randomValue) {
   if(randomValue)
```

```
    inc();
  if(randomValue)
    inc();
  if(randomValue)
    inc();
  if(randomValue)
    inc();
  if(randomValue)
    inc();
  reset();
  }
}
```

See Also

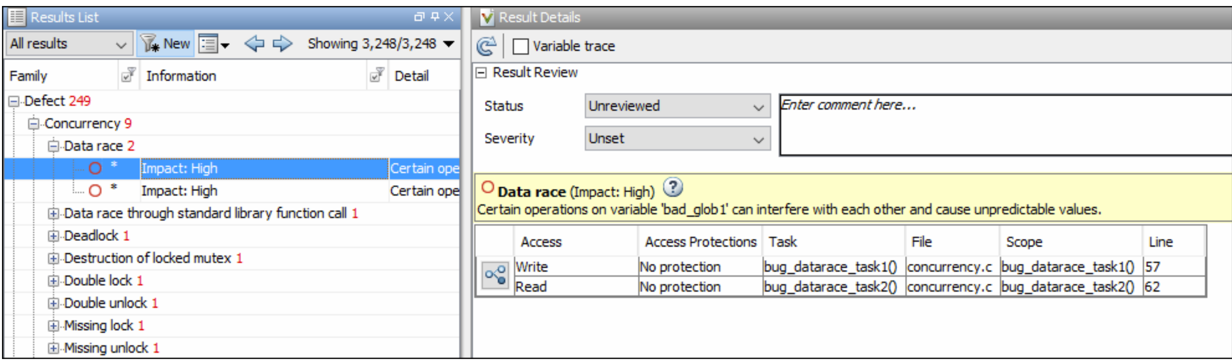
More About

- “Analyze Multitasking Programs in Polyspace” on page 10-2
- “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 10-6

Protections for Shared Variables in Multitasking Code

If your code is intended for multitasking, tasks in your code can access a common shared variable. To prevent data races, you can protect read and write operations on the variable. This topic shows the various protection mechanisms that Polyspace can recognize.

Detect Unprotected Access



You can detect an unprotected access using either Bug Finder or Code Prover. Code Prover is more exhaustive and proves if a shared variable is protected from concurrent access.

- Bug Finder detects an unprotected access using the result **Data race**. See `Data race`.
- Code Prover detects an unprotected access using the result **Shared unprotected global variable**. See `Shared unprotected global variable`.

Suppose you analyze this code, specifying `signal_handler_1` and `signal_handler_2` as cyclic tasks. Use the analysis option `Cyclic tasks (-cyclic-tasks)`.

```
#include <limits.h>
int shared_var;
```

```
void inc() {
 shared_var+=2;
}

void reset() {
 shared_var = 0;
}

void signal_handler_1(void) {
    reset();
    inc();
    inc();
}

void signal_handler_2(void) {
 shared_var = INT_MAX;
}

 void main() {
}
```

Bug Finder shows a data race on `shared_var`. Code Prover shows that `shared_var` is a potentially unprotected shared variable. Code Prover also shows that the operation `shared_var += 2` can overflow. The overflow occurs if the call to `inc` in `signal_handler_1` immediately follows the operation `shared_var = INT_MAX` in `signal_handler_2`.

Protect Using Critical Sections

One possible solution is to protect operations on shared variables using critical sections.

In the preceding example, modify your code so that operations on `shared_var` are in the same critical section. Use the functions `take_semaphore` and `give_semaphore` to begin and end the critical sections. To specify these functions that begin and end critical sections, use the analysis options `Critical section details (-critical-section-begin -critical-section-end)`.

```
#include <limits.h>
int shared_var;

void inc() {
```

```
 shared_var+=2;
}

void reset() {
 shared_var = 0;
}

/* Declare lock and unlock functions */
void take_semaphore(void);
void give_semaphore(void);

void signal_handler_1() {
    /* Begin critical section */
    take_semaphore();
    reset();
    inc();
    inc();
    /* End critical section */
    give_semaphore();

}

void signal_handler_2() {
   /* Begin critical section */
   take_semaphore();
   shared_var = INT_MAX;
   /* End critical section */
   give_semaphore();

}

void main() {
}
```

You do not see the data race in Bug Finder. Code Prover proves that the shared variable is protected. You also do not see the overflow because the call to `reset()` in `signal_handler_1` always precedes calls to `inc()`.

Protect Using Temporally Exclusive Tasks

Another possible solution is to specify a group of tasks as temporally exclusive. Temporally exclusive tasks cannot interrupt each other.

In the preceding example, specify that `signal_handler_1` and `signal_handler_2` are temporally exclusive. Use the option `Temporally exclusive tasks (-temporal-exclusions-file)`.

You do not see the data race in Bug Finder. Code Prover proves that the shared variable is protected. You also do not see the overflow because the call to `reset()` in `signal_handler_1` always precedes calls to `inc()`.

Protect Using Priorities

Another possible solution is to specify that one task has higher priority over another.

In the preceding example, specify that `signal_handler_1` is an interrupt. Retain `signal_handler_2` as a cyclic task. Use the options `Cyclic tasks (-cyclic-tasks)` and `Interrupts (-interrupts)`.

Bug Finder does not show the data race defect anymore. The reason is this:

- The operation `shared_var = INT_MAX` in `signal_handler_2` is atomic. Therefore, the operations in `signal_handler_1` cannot interrupt it.
- The operations in `signal_handler_1` cannot be inerrupted by the operation in `signal_handler_2` because `signal_handler_1` has higher priority.

Code Prover does not consider priorities of tasks. Therefore, Code Prover still shows `shared_var` as a potentially unprotected global variable.

See Also

More About

- “Analyze Multitasking Programs in Polyspace” on page 10-2

11

Configure Coding Rules Checking and Code Metrics Computation

Check for Coding Rule Violations

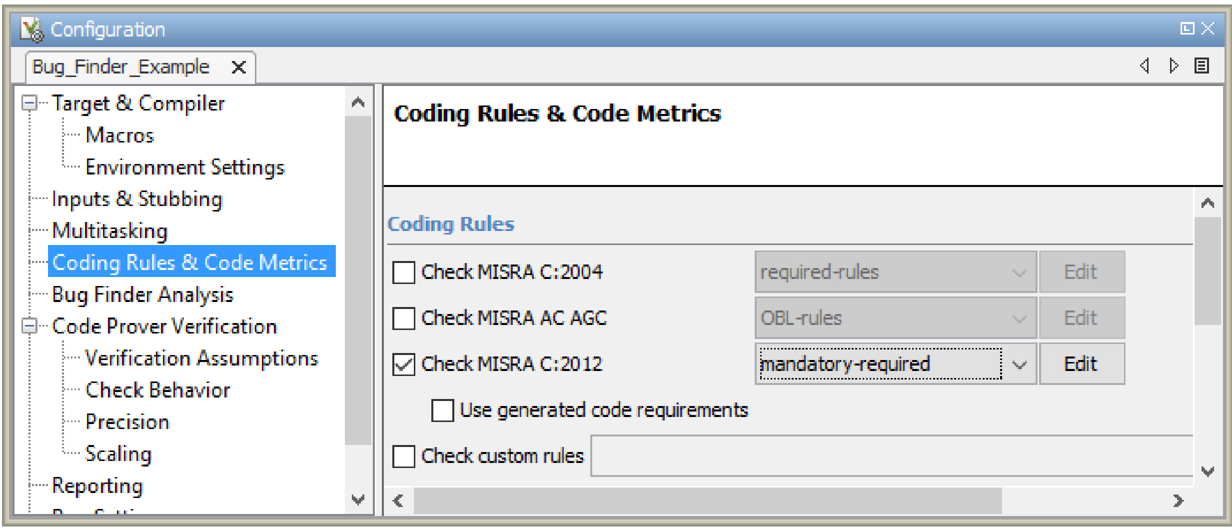
With Polyspace, you can check your C/C++ code for violations of coding rules such as MISRA C:2012 rules. Adhering to coding rules can reduce the number of defects and improve the quality of your code.

Polyspace can detect the violations of these rules:

- MISRA C: 2004
- MISRA C: 2012
- MISRA C++
- JSF++

Using Bug Finder, you can also check for security standards such as CWE, CERT C or ISO 17961. See “Check C/C++ Code for Security Standards” on page 13-42.

Configure Coding Rules Checking



Specify the coding rules through Polyspace analysis options. When you run Bug Finder or Code Prover, the analysis looks for coding rule violations in addition to other checks. You can also disable the other checks and look for coding rule violations only.

In the Polyspace user interface, the options are on the **Configuration** pane under the **Coding Rules & Code Metrics** node.

Use one of these options for C code:

- `Check MISRA C:2004 (-misra2)`
- `Check MISRA AC AGC (-misra-ac-agc)`
- `Check MISRA C:2012 (-misra3)`

Use one of these options for C++ code:

- `Check MISRA C++ rules (-misra-cpp)`
- `Check JSF++ rules (-jsf-coding-rules)`

You can specify a predefined subsets of rules, for instance, `mandatory` for MISRA C: 2012. Alternatively, you can specify your own subset in one of these ways:

- Click the **Edit** button. Select the rules to enable.

  When you save, the rules are saved in a text file that you can reuse for multiple analyses.
- Specify a text file that lists one rule per line using the syntax:

  `Rule_number` `on|off` *`#Comments`*

  For example:

  ```
  10.5 off # rule 10.5: essential type model
  17.2 on # rule 17.2: functions
  ```

  You can only enter the rules that you want to turn off. When you run an analysis, Polyspace automatically turns on the other rules and populates the file.

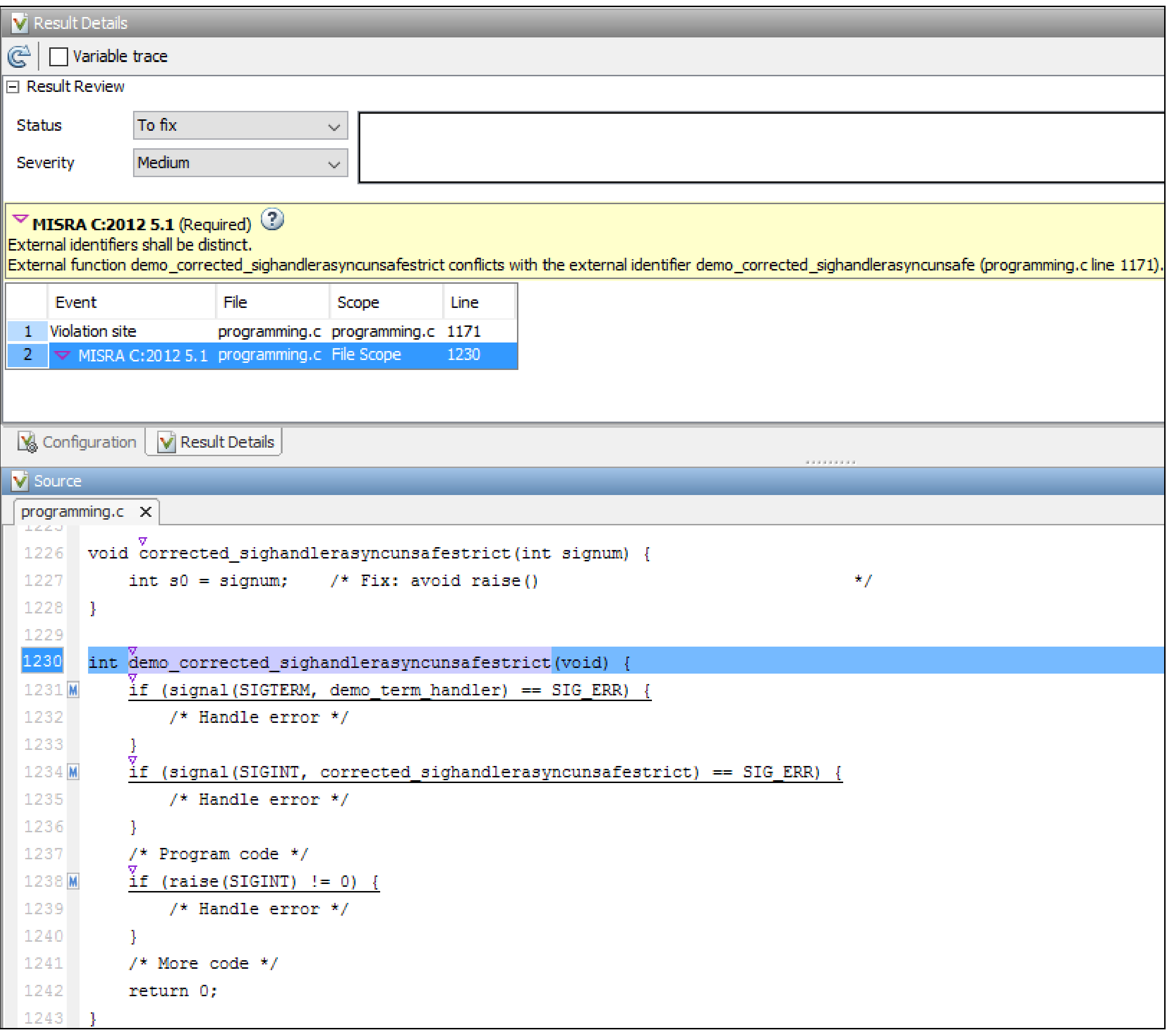
To check for coding rules only:

- In Bug Finder, disable checking of defects. Use the option `Find defects (-checkers)`.

- In Code Prover, check for source compliance only. Use the option `Verification level (-to)`.

  These rules are checked in the later stages of a Code Prover analysis: MISRA C:2004 rules 9.1, 13.7 and 21.1, and MISRA C:2012 rules 2.2, 9.1, 14.3, and 18.1. If you stop Code Prover at source compliance checking, the analysis might not find all violations of these rules.

Review Coding Rule Violations



After analysis, you see the coding rule violations on the **Results List** pane. Select a violation to see further details on the **Result Details** pane and the source code on the **Source** pane.

Violations of MISRA or JSF rules are indicated by the ▽ icon.

For further steps, see:

- “Interpret Polyspace Bug Finder Results” on page 14-2 or “Interpret Polyspace Code Prover Results” (Polyspace Code Prover)
- “Address Polyspace Results Through Bug Fixes or Comments” on page 15-2
- “Filter and Group Results” on page 16-2

Avoid Violations of MISRA C 2012 Rules 8.x

MISRA C:2012 rules 8.1-8.14 enforce good coding practices surrounding declarations and definitions. If you follow these practices, you are less likely to have conflicting declarations or to unintentionally modify variables.

If you do not follow these practices *during coding*, your code might require major changes later to be MISRA C-compliant. You might have too many MISRA C violations. Sometimes, in fixing a violation, you might violate another rule. Instead, keep these rules in mind when coding. Use the MISRA C:2012 checker to spot any issues that you might have missed.

- **Explicitly specify all data types in declarations.**

  Avoid implicit data types like this declaration of `k`:

  ```
  extern void foo (char c, const k);
  ```

  Instead use:

  ```
  extern void foo (char c, const int k);
  ```

  That way, you do not violate `MISRA C:2012 Rule 8.1`.
- **When declaring functions, provide names and data types for all parameters.**

  Avoid declarations without parameter names like these declarations:

  ```
  extern int func(int);
  extern int func2();
  ```

  Instead use:

  ```
  extern int func(int arg);
  extern int func2(void);
  ```

  That way, you do not violate `MISRA C:2012 Rule 8.2`.
- **If you want to use an object or function in multiple files, declare the object or function once in only one header file.**

  To use an object in multiple source files, declare it as `extern` in a header file. Include the header file in all the source files where you need the object. In one of those source files, define the object. For instance:

```
/* header.h */
extern int var;

/* file1.c */
#include "header.h"
/* Some usage of var */

/* file2.c */
#include "header.h"
int var=1;
```

To use a function in multiple source files, declare it in a header file. Include the header file in all the source files where you need the function. In one of those source files, define the function.

That way, you do not violate `MISRA C:2012 Rule 8.3`, `MISRA C:2012 Rule 8.4`, `MISRA C:2012 Rule 8.5`, or `MISRA C:2012 Rule 8.6`.

- **If you want to use an object or function in one file only, declare and define the object or function with the `static` specifier.**

  Make sure that you use the `static` specifier in all declarations and the definition. For instance, this function `func` is meant to be used only in the current file:

  ```
  static int func(void);
  static int func(void){
  }
  ```

  That way, you do not violate `MISRA C:2012 Rule 8.7` and `MISRA C:2012 Rule 8.8`.

- **If you want to use an object in one function only, declare the object in the function body.**

  Avoid declaring the object outside the function.

  For instance, if you use `var` in `func` only, do declare it outside the body of `func`:

  ```
  int var;
  void func(void) {
     var=1;
  }
  ```

  Instead use:

```
void func(void) {
   int var;
   var=1;
}
```

That way, you do not violate `MISRA C:2012 Rule 8.7` and `MISRA C:2012 Rule 8.9`.

- **If you want to inline a function, declare and define the function with the `static` specifier.**

  Every time you add `inline` to a function definition, add `static` too:

  ```
  static inline double func(int val);
  static inline double func(int val) {
  }
  ```

  That way, you do not violate `MISRA C:2012 Rule 8.10`

- **When declaring arrays, explicitly specify their size.**

  Avoid implicit size specifications like this:

  ```
  extern int32_t array[];
  ```

  Instead use:

  ```
  #define MAXSIZE 10
  extern int32_t array[MAXSIZE];
  ```

  That way, you do not violate `MISRA C:2012 Rule 8.11`.

- **When declaring enumerations, try to avoid mixing implicit and explicit specifications.**

  Avoid mixing implicit and explicit specifications. You can specify the first enumeration constant explicitly, but after that, use either implicit or explicit specifications. For instance, avoid this type of mix:

  ```
  enum color {red = 2, blue, green = 3, yellow};
  ```

  Instead use:

  ```
  enum color {red = 2, blue, green, yellow};
  ```

  That way, you do not violate `MISRA C:2012 Rule 8.12`.

- **When declaring pointers, point to a `const`-qualified type unless you want to use the pointer to modify an object.**

  Point to a `const`-qualified type by default unless you intend to use the pointer for modifying the pointed object. For instance, in this example, `ptr` is not used to modify the pointed object:

  ```
  char last_char(const char * const ptr){
  }
  ```

  That way, you do not violate `MISRA C:2012 Rule 8.13`.

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.

The tutorial uses the following code stored in a file `printInitialValue.c`:

```
#include <stdio.h>

typedef struct {
    int a;
    int b;
} collection;

void main()
{
    collection myCollection= {0,0};
    printf("Initial values in the collection are %d and %d.",
           myCollection.a,myCollection.b);
}
```

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 Edit.

   The New File window opens, displaying a table of rule groups.
4. Specify the rules to check for.

   a. First, clear the **Custom rules** check box to turn off checking of custom rules.

   b. Expand the **4 Structs** node. For the option **4.3 All struct fields must follow the specified pattern**:

Column Title	Action
Status	Select ☑.

Column Title	Action
Convention	Enter `All struct fields must begin with s_ and have capital letters or digits`
Pattern	Enter `s_[A-Z0-9_]+`
Comment	Leave blank. This column is for comments that appear in the coding rules file alone.

5 Save the file and run the analysis. On the **Results List** 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`

6 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**.

7 In the file, replace all instances of `a` with `s_A` and `b` with `s_B`. Rerun the analysis.

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

See Also

Polyspace Analysis Options
`Check custom rules (-custom-rules)`

More About

- “Format of Custom Coding Rules File” on page 11-13

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 analysis, it displays the coding rule violation on the **Results List** 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_.*
```

See Also

Related Examples

- "Create Custom Coding Rules" on page 11-11

Compute Code Complexity 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**: With the `polyspace-bug-finder-nodesktop` or `polyspaceBugFinder` command, use the option `-code-metrics` .

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

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

  If you impose limits on metrics, the **Results List** 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 List** display. Select **Show** > **Defects & Rules**.

Impose Limits on Metrics

1. Select **Tools** > **Preferences**.
2. On the **Review Scope** tab, do one of the following:

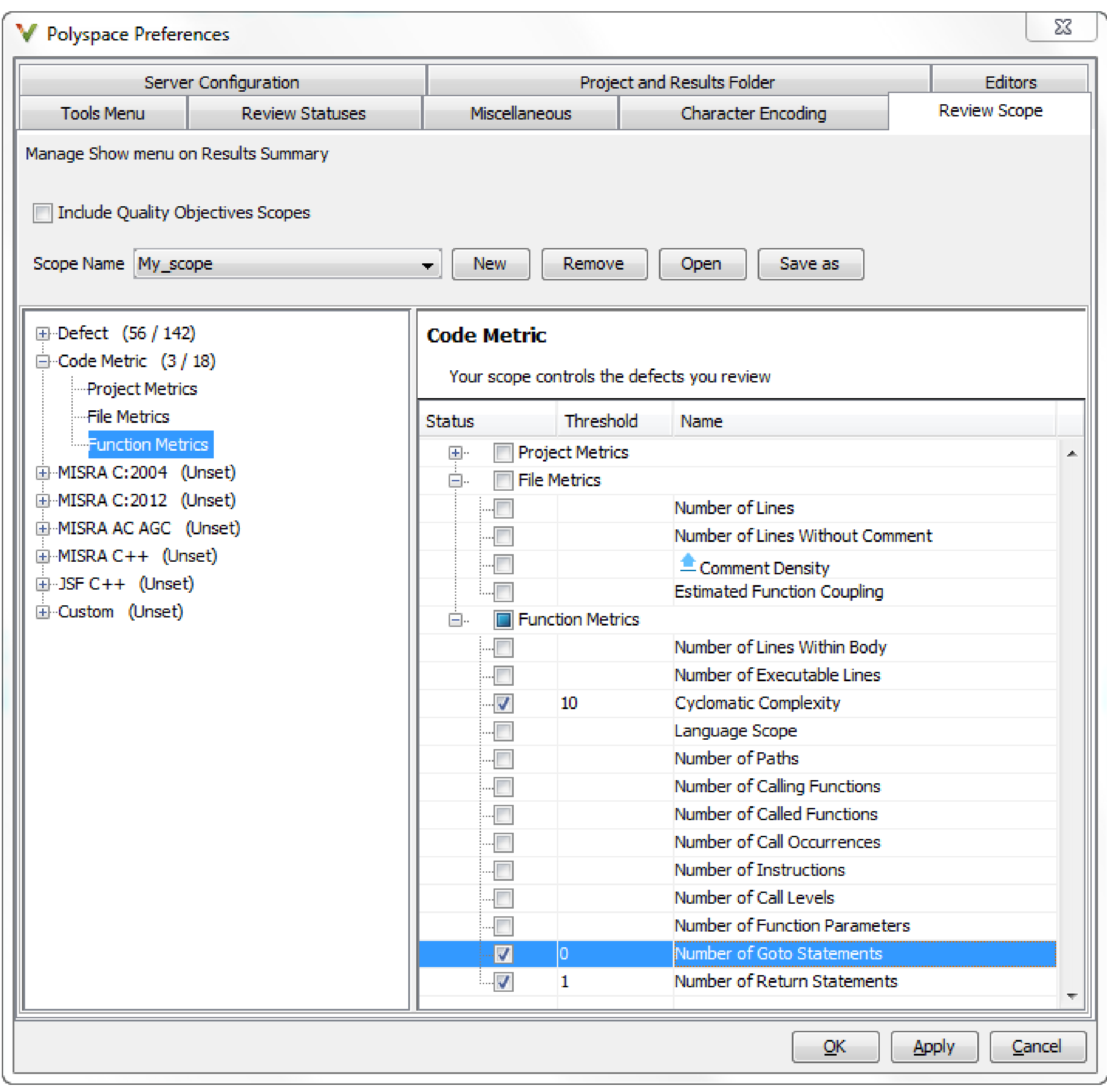
   - To use a predefined limit, select **Include Quality Objectives Scopes**.

     The **Scope Name** list shows the additional option `HIS`. The option `HIS` displays the HIS code metrics on page 11-18 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**.

The drop-down list in the left of the **Results List** pane toolbar 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 List** 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.

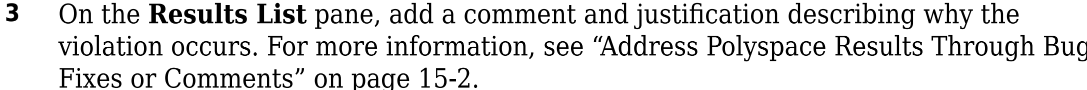
1 On the **Results List** pane, from the list, select **Family**.

The code metrics appear together under one node.

2 Expand the node. Select each violation.

- On the **Results List** 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 List** pane, add a comment and justification describing why the violation occurs. For more information, see “Address Polyspace Results Through Bug Fixes or Comments” on page 15-2.

HIS Code Complexity 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 “Compute Code Complexity Metrics” on page 11-14.

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`	4
`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

Coding Rule Sets and Concepts

- “Polyspace MISRA C 2004 and MISRA AC AGC Checkers” on page 12-2
- “MISRA C:2004 and MISRA AC AGC Coding Rules” on page 12-3
- “Software Quality Objective Subsets (C:2004)” on page 12-47
- “Software Quality Objective Subsets (AC AGC)” on page 12-53
- “Polyspace MISRA C:2012 Checkers” on page 12-57
- “Software Quality Objective Subsets (C:2012)” on page 12-59
- “Coding Rule Subsets Checked Early in Analysis” on page 12-64
- “Unsupported MISRA C:2012 Guidelines” on page 12-84
- “Polyspace MISRA C++ Checkers” on page 12-85
- “MISRA C++ Coding Rules” on page 12-86
- “Software Quality Objective Subsets (C++)” on page 12-116
- “Polyspace JSF C++ Checkers” on page 12-123
- “JSF C++ Coding Rules” on page 12-124

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.[1]

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 12-47
- “Software Quality Objective Subsets (AC AGC)” on page 12-53

Note The Polyspace MISRA checker is based on MISRA C:2004, which also incorporates MISRA C Technical Corrigendum.

See Also

More About

- “Check for Coding Rule Violations” on page 11-2
- “MISRA C:2004 and MISRA AC AGC Coding Rules” on page 12-3

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

MISRA C:2004 and MISRA AC AGC Coding Rules

In this section...

"Supported MISRA C:2004 and MISRA AC AGC Rules" on page 12-3

"Troubleshooting" on page 12-4

"List of Supported Coding Rules" on page 12-4

"Unsupported MISRA C:2004 and MISRA AC AGC Rules" on page 12-44

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.

Troubleshooting

If you expect a rule violation but do not see it, check out "Coding Rule Violations Not Displayed" on page 19-64.

List of Supported Coding Rules

- "Environment" on page 12-5
- "Language Extensions" on page 12-7
- "Documentation" on page 12-8
- "Character Sets" on page 12-8
- "Identifiers" on page 12-9
- "Types" on page 12-10
- "Constants" on page 12-12
- "Declarations and Definitions" on page 12-12
- "Initialization" on page 12-15
- "Arithmetic Type Conversion" on page 12-16
- "Pointer Type Conversion" on page 12-21
- "Expressions" on page 12-22
- "Control Statement Expressions" on page 12-26
- "Control Flow" on page 12-29
- "Switch Statements" on page 12-32
- "Functions" on page 12-33
- "Pointers and Arrays" on page 12-34
- "Structures and Unions" on page 12-35
- "Preprocessing Directives" on page 12-36
- "Standard Libraries" on page 12-40
- "Runtime Failures" on page 12-44

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 *All code shall conform to ISO 9899:1990 Programming languages C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996* precedes each of the following messages: • ANSI C does not allow ‘#include_next' • ANSI C does not allow macros with variable arguments list • ANSI C does not allow ‘#assert’ • ANSI C does not allow '#unassert' • ANSI C does not allow testing assertions • ANSI C does not allow '#ident' • ANSI C does not allow '#sccs' • text following '#else' violates ANSI standard. • text following '#endif' violates ANSI standard. • text following '#else' or '#endif' violates ANSI standard.	All the supported extensions lead to a violation of this MISRA rule. Standard compilation error messages do not lead to a violation of this MISRA rule and remain unchanged.

N.	MISRA Definition	Messages in report file	Polyspace Specification
1.1 (cont.)		The text *All code shall conform to ISO 9899:1990 Programming languages C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/ COR2:1996* precedes each of the following messages: • ANSI 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
		• 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 the following: • `asm` functions or `asm pragma` • Macros
2.2	Source code shall only use /* */ style comments	C++ comments shall not be used.	C++ comments are handled as comments but lead to a violation of this MISRA rule **Note**: This rule cannot be annotated in the source code.
2.3	The character sequence /* shall not be used within a comment	The character sequence /* shall not appear within a comment.	This rule violation is also raised when the character sequence /* inside a C++ comment. **Note**: This rule cannot be annotated in the source code.

Documentation

Rule	MISRA Definition	Messages in report file	Polyspace Specification
3.4	All uses of the #*pragma* directive shall be documented and explained.	All uses of the #pragma directive shall be documented and explained.	To check this rule, you must list the pragmas that are allowed in source files by using the option `Allowed pragmas (-allowed-pragmas)`. If Polyspace finds a pragma not in the allowed pragma list, a violation is raised.

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. For easier review, the rule checker shows all identifiers that have the same first 31 characters as one rule violation. You can see all instances of conflicting identifier names in the event history of that rule violation.
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	• Local declaration of XX is hiding another identifier. • Declaration of parameter XX is hiding another identifier.	Assumes that rule 8.1 is not violated.
5.3	A typedef name shall be a unique identifier	{typedef name}'%s' should not be reused. (already used as {typedef name} at %s:%d)	Warning when a typedef name is reused as another identifier name.
5.4	A tag name shall be a unique identifier	{tag name}'%s' should not be reused. (already used as {tag name} at %s:%d)	Warning when a tag name is reused as another identifier name
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 differently. The analyses can produce different results.

N.	MISRA Definition	Messages in report file	Polyspace Specification
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 `idf` in a namespace is reused in another namespace
5.7	No identifier name should be reused.	{identifier}'%s' should not be reused. (already used as {identifier} at %s:%d)	No violation reported when: • Different functions have parameters with the same name • Different functions have local variables with the same name • A function has a local variable that has the same name as a parameter of another function

Types

N.	MISRA Definition	Messages in report file	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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.	• Value of type plain char is implicitly converted to signed char. • Value of type plain char is implicitly converted to unsigned char. • Value of type signed char is implicitly converted to plain char. • Value of type unsigned char is implicitly converted to plain char.	Warning if value of type plain char is implicitly converted to value of type signed char or unsigned char.
6.3	*typedefs* that indicate size and signedness should be used in place of the basic types	typedefs that indicate size and signedness should be used in place of the basic types.	No warning is given in typedef definition.
6.4	Bit fields shall only be defined to be of type *unsigned int* or *signed int*.	Bit fields shall only be defined to be of type unsigned int or signed int.	
6.5	Bit fields of type *signed int* shall be at least 2 bits long.	Bit fields of type signed int shall be at least 2 bits long.	No warning on anonymous signed int bitfields of width 0 - Extended to all signed bitfields of size <= 1 (if Rule **6.4** is violated).

Constants

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

Declarations and Definitions

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

N.	MISRA Definition	Messages in report file	Polyspace Specification
8.4	If objects or functions are declared more than once their types shall be compatible.	• If objects or functions are declared more than once their types shall be compatible. • Global declaration of 'XX' function has incompatible type with its definition. • Global declaration of 'XX' variable has incompatible type with its definition.	Violations of this rule might be generated during the link phase. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.5	There shall be no definitions of objects or functions in a header file	• Object 'XX' should not be defined in a header file. • Function 'XX' should not be defined in a header file. • Fragment of function should not be defined in a header file.	Tentative definitions are considered as definitions. For objects with file scope, tentative definitions are declarations that: • Do not have initializers. • Do not have storage class specifiers, or have the `static` specifier
8.6	Functions shall always be declared at file scope.	Function 'XX' should be declared at file scope.	This rule maps to **ISO/IEC TS 17961 ID** `addrescape`.
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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
8.8	An external object or function shall be declared in one file and only one file	Function/Object 'XX' has external declarations in multiple files.	Restricted to explicit extern declarations (tentative definitions are ignored). Polyspace considers that variables or functions declared `extern` in a non-header file violate this rule. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.9	Definition: An identifier with external linkage shall have exactly one external definition.	• Procedure/Global variable XX multiply defined. • Forbidden multiple tentative definitions for object XX • Global variable has multiple tentative definitions • Undefined global variable XX	The checker flags multiple definitions only if the definitions occur in different files. No warnings appear on predefined symbols. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.10	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required	Function/Variable XX should have internal linkage.	Assumes that 8.1 is not violated. No warning if 0 uses. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.11	The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage	static storage class specifier should be used on internal linkage symbol XX.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.		Checked during code analysis. Violations displayed as Non-initialized variable results. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	

Arithmetic Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: • it is not a conversion to a wider integer type of the same signedness, or • the expression is complex, or • the expression is not constant and is a function argument, or • the expression is not constant and is a return expression	• Implicit conversion of the 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. • Implicit conversion of the binary right hand operand of underlying type XX to XX that is not a wider integer type of the same signedness or Implicit conversion of the binary ? left hand operand of underlying type XX to XX, but it is a complex expression. • Implicit conversion of complex integer expression of underlying type XX to XX.	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. An expression of bool or enum types has int as underlying type. Plain char may have signed or unsigned underlying type (depending on Polyspace 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). This rule violation is not produced on operations involving pointers. No violation reported when: • The implicit conversion is a type widening, without

N.	MISRA Definition	Messages in report file	Polyspace Specification
		• 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.	change of signedness of integer • The expression is an argument expression or a return expression No violation reported when the following are true: • Implicit conversion applies to a constant expression and is a type widening, with a possible change of signedness of 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. Conversions of constants are not reported for these cases to avoid flagging too many violations. If the constant can be represented in both the original and converted type, the conversion is less of an issue.

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.2	The value of an expression of floating type shall not be implicitly converted to a different type if • it is not a conversion to a wider floating type, or • the expression is complex, or • the expression is a function argument, or • the expression is a return expression	• Implicit conversion of the expression from XX to XX that is not a wider floating type. • Implicit conversion of the binary ? right hand operand from XX to XX, but it is a complex expression. • Implicit conversion of the binary ? right hand operand from XX to XX that is not a wider floating type or Implicit conversion of the binary ? left hand operand from 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.	ANSI C base types order (float, double) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1. No violation reported when: • The implicit conversion is a type widening • The expression is an argument expression or a return expression.

N.	MISRA Definition	Messages in report file	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.	• The rule checker raises a defect only if the result of a composite expression is cast to a different or wider essential type.
			
			For instance, in this example, a violation is shown in the first assignment to `i` but not the second. In the first assignment, a composite expression `i+1` is directly cast from a signed to an unsigned type. In the second assignment, the composite expression is first cast to the same type and then the result is cast to a different type.
			
			`typedef int int32_T;`
			`typedef unsigned char uint8_T;`
			`...`
			`...`
			`int32_T i;`
			`i = (uint8_T)(i+1);`
			`/* Noncompliant */`
			`i = (uint8_T)((int32_T)(i+1));`
			` /* Compliant */`
			• 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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
			• 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.
10.5	If the bitwise operator ~ and << are applied to an operand of underlying type *unsigned char* or *unsigned short*, the result shall be immediately cast to the underlying type of the operand	Bitwise [<<|~] is applied to the operand of underlying type [unsigned char| unsigned short], the result shall be immediately cast to the underlying type.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.6	The “U” suffix shall be applied to all constants of *unsigned* types	No explicit 'U suffix on constants of an unsigned type.	Warning when the type determined from the value and the base (octal, decimal or hexadecimal) is unsigned and there is no suffix `u` or `U`. For example, when the size of the `int` and `long int` data types is 32 bits, the coding rule checker will report a violation of rule 10.6 for the following line: `int a = 2147483648;` There is a difference between decimal and hexadecimal constants when `int` and `long int` are not the same size.

Pointer Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Specification
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type	Conversion shall not be performed between a pointer to a function and any type other than an integral type.	Casts and implicit conversions involving a function pointer. Casts or implicit conversions from `NULL` or `(void*)0` do not give any warning.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.	There is also a warning on qualifier loss This rule maps to **ISO/IEC TS 17961 ID** `alignconv`.

N.	MISRA Definition	Messages in report file	Polyspace Specification
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 This rule maps to **ISO/IEC TS 17961 ID** `alignconv`.
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.	A cast should not be performed between a pointer to object type and a different pointer to object type.	
11.5	A cast shall not be performed that removes any *const* or *volatile* qualification from the type addressed by a pointer	A cast shall not be performed that removes any *const* or *volatile* qualification from the type addressed by a pointer	Extended to all conversions

Expressions

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

<table>
<tr><th>N.</th><th>MISRA Definition</th><th>Messages in report file</th><th>Polyspace Specification</th></tr>
<tr><td>12.4</td><td>The right hand operand of a logical && or || operator shall not contain side effects.</td><td>The right hand operand of a logical && or || operator shall not contain side effects.</td><td>No warning on volatile accesses</td></tr>
<tr><td>12.5</td><td>The operands of a logical && or || shall be primary-expressions.</td><td>• operand of logical && is not a primary expression
• operand of logical || is not a primary expression
• The operands of a logical && or || shall be primary-expressions.</td><td>During preprocessing, violations of this rule are detected on the expressions in #if directives.

Allowed exception on associatively (a && b && c), (a || b || c).</td></tr>
</table>

<table>
<tr><th>N.</th><th>MISRA Definition</th><th>Messages in report file</th><th>Polyspace Specification</th></tr>
<tr>
<td>12.6</td>
<td>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 !).</td>
<td>• 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 '?:'.</td>
<td>The operand of a logical operator should be a Boolean data type. Although the C standard does not explicitly define the Boolean data type, the standard implicitly assumes the use of the Boolean data type.

Some operators may return Boolean-like expressions, for example, <code>(var == 0)</code>.

Consider the following code:

<code>unsigned char flag;</code>
<code>if (!flag)</code>

The rule checker reports a violation of rule 12.6:

<code>Operand of '!' logical operator should be effectively Boolean.</code>

The operand <code>flag</code> is not a Boolean but an <code>unsigned char</code>.

To be compliant with rule 12.6, the code must be rewritten either as

<code>if (!(flag != 0))</code>

or

<code>if (flag == 0)</code>

The use of the option <code>-boolean-types</code> may increase or decrease the</td>
</tr>
</table>

<table>
<tr><th>N.</th><th>MISRA Definition</th><th>Messages in report file</th><th>Polyspace Specification</th></tr>
<tr><td></td><td></td><td></td><td>number of warnings generated.</td></tr>
<tr><td>12.7</td><td>Bitwise operators shall not be applied to operands whose underlying type is signed</td><td>• [~/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</td><td>The underlying type for an integer is signed when:
• it does not have a u or U suffix
• it is small enough to fit into a 64 bits signed number</td></tr>
<tr><td>12.8</td><td>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.</td><td>• 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)..</td><td>The numbers that are manipulated in preprocessing directives are 64 bits wide so that valid shift range is between 0 and 63

Check is also extended onto bitfields with the field width or the width of the base type when it is within a complex expression</td></tr>
<tr><td>12.9</td><td>The unary minus operator shall not be applied to an expression whose underlying type is unsigned.</td><td>• Unary - on operand of unsigned underlying type XX.
• Minus operator applied to an expression whose underlying type is unsigned</td><td>The underlying type for an integer is signed when:
• it does not have a u or U suffix
• it is small enough to fit into a 64 bits signed number</td></tr>
<tr><td>12.10</td><td>The comma operator shall not be used.</td><td>The comma operator shall not be used.</td><td></td></tr>
</table>

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:
			• A float pointer is cast as a pointer to another data type. Casting a float pointer as a pointer to `void` does not generate a warning.
			• A float is packed with another data type. For example:
			`union {`
			`float f;`
			`int i;`
			`} …`

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

Control Statement Expressions

N.	MISRA Definition	Messages in report file	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.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
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 increase or decrease the number of warnings generated.
13.3	Floating-point expressions shall not be tested for equality or inequality.	Floating-point expressions shall not be tested for equality or inequality.	Warning on directs tests only.
13.4	The controlling expression of a *for* statement shall not contain any objects of floating type	The controlling expression of a for statement shall not contain any objects of floating type	If *for* index is a variable symbol, checked that it is not a float.

N.	MISRA Definition	Messages in report file	Polyspace Specification
13.5	The three expressions of a *for* statement shall be concerned only with loop control	• 1st expression should be an assignment. • Bad type for loop counter (XX). • 2nd expression should be a comparison. • 2nd expression should be a comparison with loop counter (XX). • 3rd expression should be an assignment of loop counter (XX). • 3rd expression: assigned variable should be the loop counter (XX). • The following kinds of for loops are allowed: (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 *for* loop for iteration counting should not be modified in the body of the loop.	Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop.	Detect only direct assignments if the for loop index is known and if it is a variable symbol.

N.	MISRA Definition	Messages in report file	Polyspace Specification
13.7	Boolean operations whose results are invariant shall not be permitted	• Boolean operations whose results are invariant shall not be permitted. Expression is always true. • Boolean operations whose results are invariant shall not be permitted. Expression is always false. • Boolean operations whose results are invariant shall not be permitted.	During compilation, check comparisons with at least one constant operand. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results. • Bug Finder flags some violations of this rule through the `Dead code` and `Useless if` checkers. • Code Prover does not use gray code to flag violations of this rule.

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	• All non-null statements shall either: • have at least one side effect however executed, or • cause control flow to change	

N.	MISRA Definition	Messages in report file	Polyspace Specification
14.3	All non-null statements shall either • 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: • there are some comments before it on the same line. • there is a comment immediately after it • there is something else than a comment after the ';' on the same line.
14.4	The *goto* statement shall not be used.	The goto statement shall not be used.	
14.5	The *continue* statement shall not be used.	The continue statement shall not be used.	
14.6	For any iteration statement there shall be at most one *break* statement used for loop termination	For any iteration statement there shall be at most one break statement used for loop termination	
14.7	A function shall have a single point of exit at the end of the function	A function shall have a single point of exit at the end of the function	
14.8	The statement forming the body of a *switch, while, do while* or *for* statement shall be a compound statement	• The body of a do while statement shall be a compound statement. • The body of a for statement shall be a compound statement. • The body of a switch statement shall be a compound statement	

N.	MISRA Definition	Messages in report file	Polyspace Specification
14.9	An *if (expression)* construct shall be followed by a compound statement. The *else* keyword shall be followed by either a compound statement, or another *if* statement	• An if (expression) construct shall be followed by a compound statement. • The else keyword shall be followed by either a compound statement, or another if statement	
14.10	All *if else if* constructs should contain a final *else* clause.	All if else if constructs should contain a final else clause.	

Switch Statements

N.	MISRA Definition	Messages in report file	Polyspace Specification

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

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

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

Functions

N.	MISRA Definition	Messages in report file	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.
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 *void.*	Functions with no parameters shall be declared with parameter type void.	Definitions are also checked.
16.6	The number of arguments passed to a function shall match the number of parameters.	• Too many arguments to XX. • Insufficient number of arguments to XX.	Assumes that rule **8.1** is not violated. This rule maps to **ISO/IEC TS 17961 ID** `argcomp`.

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

N.	MISRA Definition	Messages in report file	Polyspace Specification
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). • Array indexing on nonarray pointers.
17.5	A type should not contain more than 2 levels of pointer indirection	A type should not contain more than 2 levels of pointer indirection	
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.	Pointer to a parameter is an illegal return value. Pointer to a local is an illegal return value.	Warning when assigning address to a global variable, returning a local variable address, or returning a parameter address. This rule maps to **ISO/IEC TS 17961 ID** `accfree`.

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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
18.2	An object shall not be assigned to an overlapping object.	• An object shall not be assigned to an overlapping object. • Destination and source of XX overlap, the behavior is undefined.	
18.4	Unions shall not be used	Unions shall not be used.	

Preprocessing Directives

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

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

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.8	A function-like macro shall not be invoked without all of its arguments	• arguments given to macro '<name>' • macro '<name>' used without args. • macro '<name>' used with just one arg. • macro '<name>' used with too many (<number>) args.	
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	Macro argument shall not look like a preprocessing directive.	This rule is detected as violated when the '#' character appears in a macro argument (outside a string or character constant)
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.	Parameter instance shall be enclosed in parentheses.	If `x` is a macro parameter, the following instances of x as an operand of the # and ## operators do not generate a warning: `#x`, `##x`, and `x##`. Otherwise, parentheses are required around `x`. The software does not generate a warning if a parameter is reused as an argument of a function or function-like macro. For example, consider a parameter x. The software does not generate a warning if `x` appears as `(x)` or `(x,` or `,x)` or `,x,`.

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	'<name>' is not defined.	
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.	More than one occurrence of the # or ## preprocessor operators.	
19.13	The # and ## preprocessor operators should not be used	Message on definitions of macros using # or ## operators	
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.	'defined' without an identifier.	
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.	Precautions shall be taken in order to prevent multiple inclusions.	When a header file is formatted as, #ifndef <control macro> #define <control macro> <contents> #endif or, #ifndef <control macro> #error ... #else #define <control macro> <contents> #endif it is assumed that precautions have been taken to prevent multiple inclusions. Otherwise, a violation of this MISRA rule is detected.

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

Standard Libraries

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

N.	MISRA Definition	Messages in report file	Polyspace Specification
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 definitions are considered as definitions. For objects with file scope, tentative definitions are declarations that: • Do not have initializers. • Do not have storage class specifiers, or have the `static` specifier

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	Warning for argument in library function call if the following are all true: • Argument is a local variable • Local variable is not tested between last assignment and call to the library function • Library function is a common mathematical function • Corresponding parameter of the library function has a restricted input domain. The library function can be one of the following : `sqrt`, `tan`, `pow`, `log`, `log10`, `fmod`, `acos`, `asin`, `acosh`, `atanh`, or `atan2`.
20.4	Dynamic heap memory allocation shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the dynamic heap memory allocation functions are actually macros and the macro is expanded in the code, this rule is detected as violated. Assumes rule **20.2** is not violated.
20.5	The error indicator errno shall not be used	The error indicator errno shall not be used	Assumes that rule **20.2** is not violated
20.6	The macro *offsetof*, in library <stddef.h>, shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	Assumes that rule **20.2** is not violated

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.7	The *setjmp* macro and the *longjmp* function shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the longjmp function is actually a macro and the macro is expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.8	The signal handling facilities of <signal.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case some of the signal functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.9	The input/output library <stdio.h> shall not be used in production code.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the input/output library functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.10	The library functions atof, atoi and atoll from library <stdlib.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the atof, atoi and atoll functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.11	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the abort, exit, getenv and system functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.12	The time handling functions of library <time.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the time handling functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated

Runtime Failures

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

Rule	Description	Polyspace Specification
1.4 (Required)	The compiler/linker/Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	To observe this rule, check your compiler documentation.
1.5 (Advisory)	Floating point implementations should comply with a defined floating point standard.	To observe this rule, check your compiler documentation.

Language Extensions

Rule	Description	Polyspace Specification
2.4 (Advisory)	Sections of code should not be "commented out"	One way a tool can check this rule is to determine if the code compiles when commented out sections are uncommented. However, such checking can be expensive and inaccurate.

Documentation

Rule	Description	Polyspace Specification
3.1 (Required)	All usage of implementation-defined behavior shall be documented.	To observe this rule, check your compiler documentation. Error detection is based on undefined behavior, according to choices made for implementation- defined constructions.
3.2 (Required)	The character set and the corresponding encoding shall be documented.	To observe this rule, check your compiler documentation.
3.3 (Advisory)	The implementation of integer division in the chosen compiler should be determined, documented and taken into account.	To observe this rule, check your compiler documentation.

Rule	Description	Polyspace Specification
3.5 (Required)	The implementation-defined behavior and packing of bitfields shall be documented if being relied upon.	To observe this rule, check your compiler documentation.
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.	To observe this rule, check your compiler documentation.

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.

Software Quality Objective Subsets (C:2004)

In this section...
"Rules in SQO-Subset1" on page 12-47
"Rules in SQO-Subset2" on page 12-48

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 *static* 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 *for* statement shall not contain any objects of floating type.
13.5	The three expressions of a *for* statement shall be concerned only with loop control.
14.4	The *goto* statement 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.

Rule number	Description
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 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	*typedefs* 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 *static* 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 *for* statement shall not contain any objects of floating type.
13.5	The three expressions of a *for* statement shall be concerned only with loop control.
13.6	Numeric variables being used within a "*for*" loop for iteration counting should not be modified in the body of the loop
14.4	The *goto* 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 *switch, while, do while* or *for* statement shall be a compound statement
14.10	All *if else if* constructs should contain a final *else* clause
15.3	The final clause of a *switch* statement shall be the *default* 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; }
```

See Also

More About

- "Check for Coding Rule Violations" on page 5-16

Software Quality Objective Subsets (AC AGC)

In this section...

"Rules in SQO-Subset1" on page 12-53
"Rules in SQO-Subset2" on page 12-54

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 *static* 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	*typedefs* 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 *static* 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 &, or with a parenthesized parameter list, which may be empty
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.
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.

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*.

See Also

More About

- “Check for Coding Rule Violations” on page 5-16

Polyspace MISRA C:2012 Checkers

The Polyspace MISRA C:2012 checker helps you to comply with the MISRA C 2012 coding standard.[2]

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.

Polyspace Bug Finder can check all the MISRA C:2012 rules and most MISRA C:2012 directives. Polyspace Code Prover does not support checking of the following:

- MISRA C:2012 Directive 4.7, 4.13 and 4.14
- MISRA C:2012 Rule 21.13, 21.14, and 21.17 - 21.20
- MISRA C:2012 Rule 22.1 - 22.4 and 22.6 - 22.10

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 (-misra3-agc-mode)` 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 12-59.

See Also

`Check MISRA C:2012 (-misra3)` | `Use generated code requirements (-misra3-agc-mode)`

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

See Also

More About

- “Check for Coding Rule Violations” on page 5-16
- “MISRA C:2012 Directives and Rules”

Software Quality Objective Subsets (C:2012)

In this section...
“Guidelines in SQO-Subset1” on page 12-59
“Guidelines in SQO-Subset2” on page 12-60

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-Subset1

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

Rule	Description
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
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

Rule	Description
15.7	All if … else if constructs shall be terminated with an else statement
16.4	Every switch statement shall have a default label
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 (-misra3)` | `Use generated code requirements (-misra3-agc-mode)`

More About

- "Check for Coding Rule Violations" on page 5-16

Coding Rule Subsets Checked Early in Analysis

In the initial compilation phase of the analysis, Polyspace checks those coding rules that do not require the run-time error detection part of the analysis. If you want only those rules checked, you can perform a much quicker analysis.

The software provides two predefined subsets of rules that it checks earlier in the analysis for `Check MISRA C:2004 (-misra2)`, `Check MISRA AC AGC (-misra-ac-agc)`, and `Check MISRA C:2012 (-misra3)`.

Argument	Purpose
`single-unit-rules`	Check rules that apply only to single translation units. If you detect only coding rule violations and select this subset, the analysis stops after the compilation phase.
`system-decidable-rules`	Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. If you detect only coding rule violations and select this subset, the analysis stops after the linking phase.

See also “Check for Coding Rule Violations” on page 11-2.

MISRA C: 2004 and MISRA AC AGC Rules

The software checks the following rules early in the analysis. The rules that are checked at a system level and appear only in the `system-decidable-rules` subset are indicated by an asterisk.

Environment

Rule	Description
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.

Language Extensions

Rule	Description
2.1	Assembly language shall be encapsulated and isolated.
2.2	Source code shall only use `/* */` style comments.
2.3	The character sequence `/*` shall not be used within a comment.

Documentation

Rule	Description
3.4	All uses of the `#pragma` directive shall be documented and explained.

Character Sets

Rule	Description
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.
4.2	Trigraphs shall not be used.

Identifiers

Rule	Description
5.1*	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
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.
5.3*	A typedef name shall be a unique identifier.
5.4*	A tag name shall be a unique identifier.
5.5*	No object or function identifier with a static storage duration should be reused.
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.
5.7*	No identifier name should be reused.

Types

Rule	Description
6.1	The plain char type shall be used only for the storage and use of character values.
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.
6.3	`typedef`s that indicate size and signedness should be used in place of the basic types.
6.4	Bit fields shall only be defined to be of type `unsigned int` or `signed int`.
6.5	Bit fields of type `signed int` shall be at least 2 bits long.

Constants

Rule	Description
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and Definitions

Rule	Description
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
8.2	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.
8.4*	If objects or functions are declared more than once their types shall be compatible.
8.5	There shall be no definitions of objects or functions in a header file.
8.6	Functions shall always be declared at file scope.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function.
8.8*	An external object or function shall be declared in one file and only one file.
8.9*	An identifier with external linkage shall have exactly one external definition.
8.10*	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
8.11	The `static` 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.

Initialization

Rule	Description
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.

Arithmetic Type Conversion

Rule	Description
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: • It is not a conversion to a wider integer type of the same signedness, or • The expression is complex, or • The expression is not constant and is a function argument, or • The expression is not constant and is a return expression
10.2	The value of an expression of floating type shall not be implicitly converted to a different type if • It is not a conversion to a wider floating type, or • The expression is complex, or • The expression is a function argument, or • The expression is a return expression
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.
10.4	The value of a complex expression of float type may only be cast to narrower floating type.
10.5	If the bitwise operator ~ and `<<` are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand
10.6	The "U" suffix shall be applied to all constants of `unsigned` types.

Pointer Type Conversion

Rule	Description
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.
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.
11.5	A cast shall not be performed that removes any `const` or `volatile` qualification from the type addressed by a pointer

Expressions

Rule	Description
12.1	Limited dependence should be placed on C's operator precedence rules in expressions.
12.3	The `sizeof` operator should not be used on expressions that contain side effects.
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 `!`).
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed.
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.
12.11	Evaluation of constant unsigned expression should not lead to wraparound.
12.12	The underlying bit representations of floating-point values shall not be used.
12.13	The increment (`++`) and decrement (`--`) operators should not be mixed with other operators in an expression

Control Statement Expressions

Rule	Description
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 `for` statement shall not contain any objects of floating type.
13.5	The three expressions of a `for` statement shall be concerned only with loop control.
13.6	Numeric variables being used within a `for` loop for iteration counting should not be modified in the body of the loop.

Control Flow

Rule	Description
14.3	All non-null statements shall either • have at least one side effect however executed, or • cause control flow to change.
14.4	The `goto` statement shall not be used.
14.5	The `continue` statement shall not be used.
14.6	For any iteration statement, there shall be at most one `break` statement used for loop termination.
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 `switch`, `while`, `do while` or `for` statement shall be a compound statement.
14.9	An `if` (expression) construct shall be followed by a compound statement. The `else` keyword shall be followed by either a compound statement, or another `if` statement.
14.10	All `if else if` constructs should contain a final `else` clause.

Switch Statements

Rule	Description
15.0	Unreachable code is detected between `switch` statement and first `case`.
15.1	A `switch` label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement
15.2	An unconditional `break` statement shall terminate every non-empty `switch` clause.
15.3	The final clause of a `switch` statement shall be the `default` clause.
15.4	A `switch` expression should not represent a value that is effectively Boolean.
15.5	Every `switch` statement shall have at least one `case` clause.

Functions

Rule	Description
16.1	Functions shall not be defined with variable numbers of arguments.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.
16.4*	The identifiers used in the declaration and definition of a function shall be identical.
16.5	Functions with no parameters shall be declared with parameter type `void`.
16.6	The number of arguments passed to a function shall match the number of parameters.
16.8	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.

Pointers and Arrays

Rule	Description
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	A type should not contain more than 2 levels of pointer indirection.

Structures and Unions

Rule	Description
18.1	All structure or union types shall be complete at the end of a translation unit.
18.4	Unions shall not be used.

Preprocessing Directives

Rule	Description
19.1	`#include` statements in a file shall only be preceded by other preprocessors directives or comments.
19.2	Nonstandard characters should not occur in header file names in `#include` directives.
19.3	The `#include` directive shall be followed by either a <filename> or "filename" sequence.
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.
19.5	Macros shall not be `#define`d and `#undef`d within a block.
19.6	`#undef` shall not be used.
19.7	A function should be used in preference to a function like-macro.
19.8	A function-like macro shall not be invoked without all of its arguments.
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.
19.13	The # and ## preprocessor operators should not be used.
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.

Rule	Description
19.17	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Standard Libraries

Rule	Description
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.
20.2	The names of standard library macros, objects and functions shall not be reused.
20.4	Dynamic heap memory allocation shall not be used.
20.5	The error indicator errno shall not be used.
20.6	The macro offsetof, in library <stddef.h>, shall not be used.
20.7	The setjmp macro and the longjmp function shall not be used.
20.8	The signal handling facilities of <signal.h> shall not be used.
20.9	The input/output library <stdio.h> shall not be used in production code.
20.10	The library functions atof, atoi and atoll from library <stdlib.h> shall not be used.
20.11	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.
20.12	The time handling functions of library <time.h> shall not be used.

The rules that are checked at a system level and appear only in the system-decidable-rules subset are indicated by an asterisk.

MISRA C: 2012 Rules

The software checks the following rules early in the analysis. The rules that are checked at a system level and appear only in the system-decidable-rules subset are indicated by an asterisk.

Standard C Environment

Rule	Description
1.1	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.
1.2	Language extensions should not be used.

Unused Code

Rule	Description
2.3*	A project should not contain unused type declarations.
2.4*	A project should not contain unused tag declarations.
2.5*	A project should not contain unused macro declarations.
2.6	A function should not contain unused label declarations.
2.7	There should be no unused parameters in functions.

Comments

Rule	Description
3.1	The character sequences `/*` and `//` shall not be used within a comment.
3.2	Line-splicing shall not be used in `//` comments.

Character Sets and Lexical Conventions

Rule	Description
4.1	Octal and hexadecimal escape sequences shall be terminated.
4.2	Trigraphs should not be used.

Identifiers

Rule	Description
5.1*	External identifiers shall be distinct.
5.2	Identifiers declared in the same scope and name space shall be distinct.
5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
5.4	Macro identifiers shall be distinct.
5.5	Identifiers shall be distinct from macro names.
5.6*	A typedef name shall be a unique identifier.
5.7*	A tag name shall be a unique identifier.
5.8*	Identifiers that define objects or functions with external linkage shall be unique.
5.9*	Identifiers that define objects or functions with internal linkage should be unique.

Types

Rule	Description
6.1	Bit-fields shall only be declared with an appropriate type.
6.2	Single-bit named bit fields shall not be of a signed type.

Literals and Constants

Rule	Description
7.1	Octal constants shall not be used.
7.2	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.
7.3	The lowercase character "l" shall not be used in a literal suffix.
7.4	A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Declarations and Definitions

Rule	Description
8.1	Types shall be explicitly specified.
8.2	Function types shall be in prototype form with named parameters.
8.3*	All declarations of an object or function shall use the same names and type qualifiers.
8.4	A compatible declaration shall be visible when an object or function with external linkage is defined.
8.5*	An external object or function shall be declared once in one and only one file.
8.6*	An identifier with external linkage shall have exactly one external definition.
8.7*	Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.
8.8	The `static` storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.9*	An object should be defined at block scope if its identifier only appears in a single function.
8.10	An inline function shall be declared with the `static` storage class.
8.11	When an array with external linkage is declared, its size should be explicitly specified.
8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.
8.14	The `restrict` type qualifier shall not be used.

Initialization

Rule	Description
9.2	The initializer for an aggregate or union shall be enclosed in braces.
9.3	Arrays shall not be partially initialized.
9.4	An element of an object shall not be initialized more than once.
9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

The Essential Type Model

Rule	Description
10.1	Operands shall not be of an inappropriate essential type.
10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.
10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.
10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
10.5	The value of an expression should not be cast to an inappropriate essential type.
10.6	The value of a composite expression shall not be assigned to an object with wider essential type.
10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.
10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Pointer Type Conversion

Rule	Description
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.
11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object 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.
11.9	The macro `NULL` shall be the only permitted form of integer null pointer constant.

Expressions

Rule	Description
12.1	The precedence of operators within expressions should be made explicit.
12.3	The comma operator should not be used.
12.4	Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Side Effects

Rule	Description
13.3	A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.
13.4	The result of an assignment operator should not be used.
13.6	The operand of the `sizeof` operator shall not contain any expression which has potential side effects.

Control Statement Expressions

Rule	Description
14.4	The controlling expression of an `if` statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Control Flow

Rule	Description
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.
15.4	There should be no more than one `break` or `goto` statement used to terminate any iteration 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.

Switch Statements

Rule	Description
16.1	All `switch` statements shall be well-formed.
16.2	A `switch` label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.
16.3	An unconditional `break` statement shall terminate every `switch`-clause.
16.4	Every `switch` statement shall have a `default` label.
16.5	A `default` label shall appear as either the first or the last `switch` label of a `switch` statement.
16.6	Every `switch` statement shall have at least two `switch`-clauses.
16.7	A `switch`-expression shall not have essentially Boolean type.

Functions

Rule	Description
17.1	The features of `<starg.h>` shall not be used.
17.3	A function shall not be declared implicitly.
17.4	All exit paths from a function with non-`void` return type shall have an explicit return statement with an expression.
17.6	The declaration of an array parameter shall not contain the `static` keyword between the `[ ]`.
17.7	The value returned by a function having non-`void` return type shall be used.

Pointers and Arrays

Rule	Description
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.
18.7	Flexible array members shall not be declared.
18.8	Variable-length array types shall not be used.

Overlapping Storage

Rule	Description
19.2	The `union` keyword should not be used.

Preprocessing Directives

Rule	Description
20.1	`#include` directives should only be preceded by preprocessor directives or comments.
20.2	The `'`, `"`, or `\` characters and the `/*` or `//` character sequences shall not occur in a header file name.
20.3	The `#include` directive shall be followed by either a <filename> or \"filename \" sequence.
20.4	A macro shall not be defined with the same name as a keyword.
20.5	`#undef` should not be used.
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.
20.8	The controlling expression of a `#if` or `#elif` preprocessing directive shall evaluate to 0 or 1.
20.9	All identifiers used in the controlling expression of `#if` or `#elif` preprocessing directives shall be `#define`'d before evaluation.
20.10	The # and ## preprocessor operators should not be used.
20.11	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.
20.12	A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.
20.13	A line whose first token is # shall be a valid preprocessing directive.
20.14	All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if`, `#ifdef` or `#ifndef` directive to which they are related.

Standard Libraries

Rule	Description
21.1	#define and #undef shall not be used on a reserved identifier or reserved macro name.
21.2	A reserved identifier or macro name shall not be declared.
21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used.
21.4	The standard header file <setjmp.h> shall not be used.
21.5	The standard header file <signal.h> shall not be used.
21.6	The Standard Library input/output functions shall not be used.
21.7	The atof, atoi, atol, and atoll functions of <stdlib.h> shall not be used.
21.8	The library functions abort, exit, getenv and system of <stdlib.h> shall not be used.
21.9	The library functions bsearch and qsort of <stdlib.h> shall not be used.
21.10	The Standard Library time and date functions shall not be used.
21.11	The standard header file <tgmath.h> shall not be used.
21.12	The exception handling features of <fenv.h> should not be used.

The rules that are checked at a system level and appear only in the system-decidable-rules subset are indicated by an asterisk.

Unsupported MISRA C:2012 Guidelines

The Polyspace coding rules checker does not check the following MISRA C:2012 directives. These directives are not checked either in Bug Finder or Code Prover. These directives 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 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 4.12	Required	Required	Dynamic memory allocation shall not be used

Polyspace MISRA C++ Checkers

The Polyspace MISRA C++ checker helps you comply with the MISRA C++:2008 coding standard.[3]

When MISRA C++ rules are violated, the Polyspace software provides messages with information about why the code violates the rule. Most violations are found during the compile phase of an analysis. The MISRA C++ checker can check 202 of the 230 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 12-116.

Note The Polyspace MISRA C++ checker is based on MISRA C++:2008 – “Guidelines for the use of the C++ language in critical systems."

See Also

More About

- “Check for Coding Rule Violations” on page 11-2
- “MISRA C++ Coding Rules” on page 12-86

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

MISRA C++ Coding Rules

In this section...
"Supported MISRA C++ Coding Rules" on page 12-86
"Unsupported MISRA C++ Rules" on page 12-111

Supported MISRA C++ Coding Rules

- "Language Independent Issues" on page 12-87
- "General" on page 12-88
- "Lexical Conventions" on page 12-88
- "Basic Concepts" on page 12-90
- "Standard Conversions" on page 12-91
- "Expressions" on page 12-92
- "Statements" on page 12-96
- "Declarations" on page 12-99
- "Declarators" on page 12-101
- "Classes" on page 12-102
- "Derived Classes" on page 12-103
- "Member Access Control" on page 12-103
- "Special Member Functions" on page 12-104
- "Templates" on page 12-104
- "Exception Handling" on page 12-105
- "Preprocessing Directives" on page 12-108
- "Library Introduction" on page 12-109
- "Language Support Library" on page 12-110
- "Diagnostic Library" on page 12-110
- "Input/output Library" on page 12-111

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 analyses can produce different results.
0-1-2	Required	A project shall not contain infeasible paths.	
0-1-3	Required	A project shall not contain unused variables.	The checker flags local or global variables that are declared or defined but not used anywhere in the source files. This also applies to members of structures and classes.
0-1-5	Required	A project shall not contain unused type declarations.	
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.
0-1-9	Required	There shall be no dead code.	This rule can also be enforced through detection of dead code during analysis.
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.
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.	Polyspace checks for unused parameters in the set of virtual functions within single translation units.
0-2-1	Required	An object shall not be assigned to an overlapping object.	

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. 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.

N.	Category	MISRA Definition	Polyspace Specification
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.	
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.	
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.	

N.	Category	MISRA Definition	Polyspace Specification
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 !=, 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	

N.	Category	MISRA Definition	Polyspace Specification
4-10-1	Required	NULL shall not be used as an integer value.	The checker flags assignment of NULL to an integer variable or binary operations involving NULL and an integer. Assignments can be direct or indirect such as passing NULL as integer argument to a function.
4-10-2	Required	Literal zero (0) shall not be used as the null-pointer-constant.	The checker flags assignment of 0 to a pointer variable or binary operations involving 0 and a pointer. Assignments can be direct or indirect such as passing 0 as pointer argument to a function.

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.

N.	Category	MISRA Definition	Polyspace Specification
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.
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 ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	
5-0-11	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.	
5-0-13	Required	The condition of an if-statement and the condition of an iteration-statement shall have type bool.	
5-0-14	Required	The first operand of a conditional-operator shall have type bool.	

<table>
<tr><th>N.</th><th>Category</th><th>MISRA Definition</th><th>Polyspace Specification</th></tr>
<tr><td>5-0-15</td><td>Required</td><td>Array indexing shall be the only form of pointer arithmetic.</td><td>Warning on:
• Operations on pointers. (p+I, I +p and p-I, where p is a pointer and I an integer, p[i] accepted).
• Array indexing on nonarray pointers.</td></tr>
<tr><td>5-0-17</td><td>Required</td><td>Subtraction between pointers shall only be applied to pointers that address elements of the same array.</td><td>Use Bug Finder for this checker. Code Prover can fail to detect some violations.</td></tr>
<tr><td>5-0-18</td><td>Required</td><td>>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.</td><td>Report when relational operator are used on pointers types (casts ignored).</td></tr>
<tr><td>5-0-19</td><td>Required</td><td>The declaration of objects shall contain no more than two levels of pointer indirection.</td><td></td></tr>
<tr><td>5-0-20</td><td>Required</td><td>Non-constant operands to a binary bitwise operator shall have the same underlying type.</td><td></td></tr>
<tr><td>5-0-21</td><td>Required</td><td>Bitwise operators shall only be applied to operands of unsigned underlying type.</td><td></td></tr>
<tr><td>5-2-1</td><td>Required</td><td>Each operand of a logical && or || shall be a postfix - expression.</td><td>During preprocessing, violations of this rule are detected on the expressions in #if directives.
Allowed exception on associativity (a && b && c), (a || b || c).</td></tr>
<tr><td>5-2-2</td><td>Required</td><td>A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.</td><td></td></tr>
<tr><td>5-2-3</td><td>Advisory</td><td>Casts from a base class to a derived class should not be performed on polymorphic types.</td><td></td></tr>
</table>

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

N.	Category	MISRA Definition	Polyspace Specification
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.	
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.	

N.	Category	MISRA Definition	Polyspace Specification
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` (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.	

N.	Category	MISRA Definition	Polyspace Specification
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.	
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.	

N.	Category	MISRA Definition	Polyspace Specification
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
7-1-1	Required	A variable which is not modified shall be const qualified.	The checker flags function parameters or local variables that are not `const`-qualified but never modified in the function body. Function parameters of integer, float, enum and boolean types are not flagged. If a variable is passed to another function by reference or pointers, the checker assumes that the variable can be modified. These variables are not flagged.
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.	The checker flags pointers where the underlying object is not `const`-qualified but never modified in the function body. If a variable is passed to another function by reference or pointers, the checker assumes that the variable can be modified. Pointers that point to these variables are not flagged.

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.	
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.	

N.	Category	MISRA Definition	Polyspace Specification
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 &.	

N.	Category	MISRA Definition	Polyspace Specification
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.
9-3-3	Required	If a member function cannot be made static then it shall be made static, otherwise if it can be made const then it shall be made const.	The checker flags member functions that are not declared `static` but do not access a data member of the class. The checker flags member functions that are not declared `const` but do not modify a data member of the class.
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.	

N.	Category	MISRA Definition	Polyspace Specification
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.
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.	

N.	Category	MISRA Definition	Polyspace Specification
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.	All specializations of overloaded templates are rejected even if overloading occurs after the call. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
14-8-2	Advisory	The viable function set for a function call should either contain no 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.	

N.	Category	MISRA Definition	Polyspace Specification
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.	Detect that there is no try/catch in the main and that the catch does not handle all exceptions. Not detected if no "main". Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
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 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.	

N.	Category	MISRA Definition	Polyspace Specification
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.

15-5-3	Required	The terminate() function shall not be called implicitly.	The checker flags these situations when the `terminate()` function can be called implicitly:
			• An exception escapes uncaught. This also violates rule 15-3-2. For instance:
			  • Before an exception is caught, it escapes through another function that throws an uncaught exception. For instance, a `catch` statement or exception handler invokes a copy constructor that throws an uncaught exception.
			  • A throw expression with no operand rethrows an uncaught exception.
			• A class destructor throws an exception. This also violates rule 15-5-1.

Preprocessing Directives

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

N.	Category	MISRA Definition	Polyspace Specification
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.	
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.	
16-6-1	Document	All uses of the #pragma directive shall be documented.	To check this rule, you must list the pragmas that are allowed in source files by using the option `Allowed pragmas (-allowed-pragmas)`. If Polyspace finds a pragma not in the allowed pragma list, a violation is raised.

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.

N.	Category	MISRA Definition	Polyspace Specification
17-0-2	Required	The names of standard library macros and objects shall not be reused.	
17-0-3	Required	The names of standard library functions shall not be overridden.	
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 <cstdlib> shall not be used.	
18-0-3	Required	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.	The option `-compiler iso` must be used to detect violations, for example, `exit`.
18-0-4	Required	The time handling functions of library <ctime> shall not be used.	
18-0-5	Required	The unbounded functions of library <cstring> shall not be used.	
18-2-1	Required	The macro offsetof 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 <csignal> shall not be used.	

Diagnostic Library

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

Input/output Library

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

Unsupported MISRA C++ Rules

- “Language Independent Issues” on page 12-111
- “General” on page 12-112
- “Lexical Conventions” on page 12-112
- “Expressions” on page 12-113
- “Declarations” on page 12-113
- “Classes” on page 12-114
- “Templates” on page 12-114
- “Exception Handling” on page 12-114
- “Library Introduction” on page 12-115

Language Independent Issues

N.	Category	MISRA Definition	Polyspace Specification
0-1-4	Required	A project shall not contain non-volatile POD variables having only one use.	
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.	

N.	Category	MISRA Definition	Polyspace Specification
0-3-1	Required	Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/ techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.	
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.	To observe this rule, check your compiler documentation.
0-4-2	Document	Use of floating-point arithmetic shall be documented.	To observe this rule, check your compiler documentation.
0-4-3	Document	Floating-point implementations shall comply with a defined floating-point standard.	To observe this rule, check your compiler documentation.

General

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

Lexical Conventions

N.	Category	MISRA Definition	Polyspace Specification
2-2-1	Document	The character set and the corresponding encoding shall be documented.	To observe this rule, check your compiler documentation.

N.	Category	MISRA Definition	Polyspace Specification
2-7-2	Required	Sections of code shall not be "commented out" using C-style comments.	One way a tool can check this rule is to determine if the code compiles when commented out sections are uncommented. However, such checking can be expensive and inaccurate.
2-7-3	Advisory	Sections of code should not be "commented out" using C++ comments.	One way a tool can check this rule is to determine if the code compiles when commented out sections are uncommented. However, such checking can be expensive and inaccurate.

Expressions

N.	Category	MISRA Definition	Polyspace Specification
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.	
5-17-1	Required	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.	

Declarations

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.	To observe this rule, check your compiler documentation.

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.	To observe this rule, check your compiler documentation.

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.	To observe this rule, check your compiler documentation.

N.	Category	MISRA Definition	Polyspace Specification
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.	

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++.	To observe this rule, check your compiler documentation.

Software Quality Objective Subsets (C++)

In this section...

"SQO Subset 1 – Direct Impact on Selectivity" on page 12-116
"SQO Subset 2 – Indirect Impact on Selectivity" on page 12-118

SQO Subset 1 – Direct Impact on Selectivity

The following set of MISRA C++ 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.

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

<table>
<tr><th>MISRA C++ Rule</th><th>Description</th></tr>
<tr><td>4-5-1</td><td>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.</td></tr>
<tr><td>5-0-1</td><td>The value of an expression shall be the same under any order of evaluation that the standard permits.</td></tr>
<tr><td>5-0-2</td><td>Limited dependence should be placed on C++ operator precedence rules in expressions.</td></tr>
<tr><td>5-0-7</td><td>There shall be no explicit floating-integral conversions of a cvalue expression.</td></tr>
<tr><td>5-0-8</td><td>An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.</td></tr>
<tr><td>5-0-9</td><td>An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.</td></tr>
<tr><td>5-0-10</td><td>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.</td></tr>
<tr><td>5-0-13</td><td></td></tr>
<tr><td>5-0-15</td><td>Array indexing shall be the only form of pointer arithmetic.</td></tr>
<tr><td>5-0-18</td><td>>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.</td></tr>
<tr><td>5-0-19</td><td>The declaration of objects shall contain no more than two levels of pointer indirection.</td></tr>
<tr><td>5-2-1</td><td>Each operand of a logical && or || shall be a postfix - expression.</td></tr>
<tr><td>5-2-2</td><td>A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.</td></tr>
<tr><td>5-2-5</td><td>A cast shall not remove any const or volatile qualification from the type of a pointer or reference.</td></tr>
<tr><td>5-2-6</td><td>A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.</td></tr>
<tr><td>5-2-7</td><td>An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.</td></tr>
</table>

MISRA C++ Rule	Description
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
5-2-11	The comma operator, && operator and the || operator shall not be overloaded.
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.

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

MISRA C++ Rule	Description
15-3-5	A class type exception shall always be caught by reference.
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
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.

Polyspace JSF C++ Checkers

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.

4

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.

See Also

More About

- "Check for Coding Rule Violations" on page 11-2
- "JSF C++ Coding Rules" on page 12-124

4. JSF and Joint Strike Fighter are Lockheed Martin.

JSF C++ Coding Rules

Supported JSF C++ Coding Rules

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 `-compiler iso`.
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.	

N.	JSF++ Definition	Polyspace Specification
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 `errno` **shall not** be used.	`errno` should not be used as a macro or a global with external "C" linkage.
18	The macro `offsetof`, in library `<stddef.h>`, **shall not** be used.	`offsetof` should not be used as a macro or a global with external "C" linkage.
19	`<locale.h>` and the `setlocale` function **shall not** be used.	`setlocale` and `localeconv` should not be used as a macro or a global with external "C" linkage.
20	The `setjmp` macro and the `longjmp` function **shall not** be used.	`setjmp` and `longjmp` should not be used as a macro or a global with external "C" linkage.
21	The signal handling facilities of `<signal.h>` **shall not** be used.	`signal` and `raise` should not be used as a macro or a global with external "C" linkage.
22	The input/output library `<stdio.h>` **shall not** be used.	all standard functions of `<stdio.h>` should not be used as a macro or a global with external "C" linkage.
23	The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>` **shall not** be used.	`atof`, `atoi` and `atol` should not be used as a macro or a global with external "C" linkage.
24	The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` **shall not** be used.	`abort`, `exit`, `getenv` and `system` should not be used as a macro or a global with external "C" linkage.
25	The time handling functions of library `<time.h>` **shall not** be used.	`clock`, `difftime`, `mktime`, `asctime`, `ctime`, `gmtime`, `localtime` and `strftime` should not be used as a macro or a global with external "C" linkage.

Pre-Processing Directives

N.	JSF++ Definition	Polyspace Specification
26	Only the following preprocessor directives **shall** be used: `#ifndef`, `#define`, `#endif`, `#include`.	
27	`#ifndef`, `#define` and `#endif` **will** be used to prevent multiple inclusions of the same header file. Other techniques to prevent the multiple inclusions of header files **will not** be used.	Detects the patterns `#if !defined`, `#pragma once`, `#ifdef`, and missing `#define`.
28	The `#ifndef` and `#endif` preprocessor directives **will** only be used as defined in AV Rule 27 to prevent multiple inclusions of the same header file.	Detects any use that does not comply with AV Rule 27. Assuming 35/27 is not violated, reports only `#ifndef`.
29	The `#define` preprocessor directive **shall not** be used to create inline macros. Inline functions shall be used instead.	Rule is split into two parts: the definition of a macro function (29.def) and the call of a macrofunction (29.use). Messages in report file: • 29.1 : The `#define` preprocessor directive shall not be used to create inline macros. • 29.2 : Inline functions shall be used instead of inline macros.
30	The `#define` preprocessor directive **shall not** be used to define constant values. Instead, the `const` qualifier **shall** be applied to variable declarations to specify constant values.	Reports `#define` of simple constants.
31	The `#define` preprocessor directive **will** only be used as part of the technique to prevent multiple inclusions of the same header file.	Detects use of `#define` that are not used to guard for multiple inclusion, assuming that rules 35 and 27 are not violated.
32	The `#include` preprocessor directive **will** only be used to include header (*.h) files.	

Header Files

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

Style

N.	JSF++ Definition	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 if/else, do/while, for, and while statements. NB: in final release it will accept any indentation
46	User-specified identifiers (internal and external) **will not** rely on significance of more than 64 characters.	

N.	JSF++ Definition	Polyspace Specification
47	Identifiers **will not** begin with the underscore character '_'.	
48	Identifiers **will not** differ by: • Only a mixture of case • The presence/absence of the underscore character • The interchange of the letter 'O'; with the number '0' or the letter 'D' • The interchange of the letter 'I'; with the number '1' or the letter 'l' • The interchange of the letter 'S' with the number '5' • The interchange of the letter 'Z' with the number 2 • The interchange of the letter 'n' with the letter 'h'	Checked regardless of scope. Not checked between macros and other identifiers. Messages in report file: • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by the presence/absence of the underscore character. • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by a mixture of case. • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) 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: • 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.	Messages in report file: • All letters contained in variable names will be composed entirely of lowercase letters. • All letters contained in function names will be composed entirely of lowercase letters.
52	Identifiers for constant and enumerator values **shall** be lowercase.	Messages in report file: • Identifier for enumerator value shall be lowercase. • Identifier for template constant parameter shall be lowercase.
53	Header files **will** always have file name extension of "`.h`".	`.H` is allowed if you set the option `-dos`.
53.1	The following character sequences **shall** not appear in header file names: `'`, `\`, `/*`, `//`, or `"`.	
54	Implementation files **will** always have a file name extension of ".cpp".	Not case sensitive if you set the option `-dos`.
57	The public, protected, and private sections of a class **will** be declared in that order.	
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.	Messages in report file: • The statements forming the body of an if statement shall always be enclosed in braces. • The statements forming the body of an else statement shall always be enclosed in braces. • The statements forming the body of a while statement shall always be enclosed in braces. • The statements forming the body of a do … while statement shall always be enclosed in braces. • The statements forming the body of a for statement shall always be enclosed in braces.
60	Braces ("{}") which enclose a block **will** be placed in the same column, on separate lines directly before and after the block.	Detects that statement-block braces should be in the same columns.
61	Braces ("{}") which enclose a block **will** have nothing else on the line except comments.	
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.	Reports when the following characters are not directly connected to a white space: • . • -> • ! • ~ • - • ++ • — Note that a violation will be reported for "." used in float/double definition.

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.	All data should be initialized in the initialization list except for array. Does not report that an assignment exists in `ctor` body. Message in report file: Initialization of nonstatic class members "<*field*>" will be performed through the member initialization list.

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.	Messages in report file: • `no copy constructor and no copy assign` • `no copy constructor` • `no copy assign` Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
77.1	The definition of a member function **shall not** contain default arguments that produce a signature identical to that of the implicitly-declared copy constructor for the corresponding class/structure.	Does not report when an explicit copy constructor exists.
78	All base classes with a virtual function **shall** define a virtual destructor.	
79	All resources acquired by a class shall be released by the class's destructor.	Reports when the number of "new" called in a constructor is greater than the number of "delete" called in its destructor. **Note** A violation is raised even if "new" is done in a "if/else".

<table>
<tr><th>N.</th><th>JSF++ Definition</th><th>Polyspace Specification</th></tr>
<tr><td>81</td><td>The assignment operator shall handle self-assignment correctly</td><td>Reports when copy assignment body does not begin with “if (this != arg)”

A violation is not raised if an empty else statement follows the if, or the body contains only a return statement.

A violation is raised when the if statement is followed by a statement other than the return statement.</td></tr>
<tr><td>82</td><td>An assignment operator shall return a reference to *this.</td><td>The following operators should return *this on method, and *first_arg on plain function.

operator=operator+=operator-=operator*=operator >>=operator <<=operator /=operator %=operator |=operator &=operator ^=prefix operator++ prefix operator--

Does not report when no return exists.

No special message if type does not match.

Messages in report file:

• An assignment operator shall return a reference to *this.
• An assignment operator shall return a reference to its first arg.</td></tr>
<tr><td>83</td><td>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).</td><td>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.</td></tr>
</table>

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: • Multiple inheritance on public implementation shall not be allowed: *<public_base_class>* is not an interface. • Multiple inheritance on protected implementation shall not be allowed : *<protected_base_class_1>*. • *<protected_base_class_2>* are not interfaces.
88.1	A stateful virtual base **shall** be explicitly declared in each derived class that accesses it.	
89	A base class **shall not** be both virtual and nonvirtual in the same hierarchy.	
94	An inherited nonvirtual function **shall not** be redefined in a derived class.	Does not report for destructor. Message in report file: Inherited nonvirtual function `%s` shall not be redefined in a derived class.
95	An inherited default parameter **shall never** be redefined.	
96	Arrays **shall not** be treated polymorphically.	Reports pointer arithmetic and array like access on expressions whose pointed type is used as a base class.
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 `main()`, **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.	

N.	JSF++ Definition	Polyspace Specification
116	Small, concrete-type arguments (two or three words in size) **should** be passed by value if changes made to formal parameters should not be reflected in the calling function.	Report constant parameters references with `sizeof <= 2 * sizeof(int)`. Does not report for copy-constructor.
119	Functions **shall** not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	Direct recursion is reported statically. Indirect recursion reported through the software. Message in report file: Function <*F*> shall not call directly itself.
121	Only functions with 1 or 2 statements **should** be considered candidates for inline functions.	Reports inline functions with more than 2 statements.

Comments

N.	JSF++ Definition	Polyspace 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.

N.	JSF++ Definition	Polyspace Specification
136	Declarations should be at the smallest feasible scope.	Reports when: • A global variable is used in only one function. • A local variable is not used in a statement (`expr`, `return`, `init` ...) of the same level of its declaration (in the same block) or is not used in two sub-statements of its declaration. **Note** • Non-used variables are reported. • Initializations at definition are ignored (not considered an access)
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.

N.	JSF++ Definition	Polyspace Specification
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 `void*`).
148	Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.	Reports when non enumeration types are used in switches.

Constants

N.	JSF++ Definition	Polyspace 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.	Reports direct numeric constants (except integer/float value `1, 0`) in expressions, non `-const` initializations. and switch cases. char constants are allowed. Does not report on templates non-type parameter. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

N.	JSF++ Definition	Polyspace Specification
151.1	A string literal shall not be modified.	Report when a `char*`, `char[]`, or `string` type is used not as `const`. A violation is raised if a string literal (for example, " ") is cast as a non `const`.

Variables

N.	JSF++ Definition	Polyspace 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 && or `||` operator shall not contain side effects.	Assumes rule 159 is not violated. Messages in report file: • The right hand operand of a && operator shall not contain side effects. • The right hand operand of a `||` operator shall not contain side effects.

N.	JSF++ Definition	Polyspace Specification
158	The operands of a logical && or `||` **shall** be parenthesized if the operands contain binary operators.	Messages in report file: • The operands of a logical && shall be parenthesized if the operands contain binary operators. • The operands of a logical `||` shall be parenthesized if the operands contain binary operators. Exception for: `X || Y || Z` , `Z&&Y &&Z`
159	Operators `||`, &&, and unary & **shall not** be overloaded.	Messages in report file: • Unary operator & shall not be overloaded. • Operator `||` shall not be overloaded. • Operator && shall not be overloaded.
160	An assignment expression **shall** be used only as the expression in an expression statement.	Only simple assignment, not +=, ++, etc.
162	Signed and unsigned values **shall not** be mixed in arithmetic or comparison operations.	
163	Unsigned arithmetic **shall not** be used.	
164	The right hand operand of a shift operator **shall** lie between zero and one less than the width in bits of the left-hand operand (inclusive).	
164.1	The left-hand operand of a right-shift operator **shall not** have a negative value.	Detects constant case +. Found by the software for dynamic cases.
165	The unary minus operator **shall not** be applied to an unsigned expression.	
166	The `sizeof` operator **will not** be used on expressions that contain side effects.	
168	The comma operator **shall not** be used.	

Pointers and References

N.	JSF++ Definition	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: • the same object, • the same function, • members of the same object, or • elements of the same array (including one past the end of the same array).	Reports when relational operator are used on pointer types (casts ignored).
173	The address of an object with automatic storage **shall not** be assigned to an object which persists after the object has ceased to exist.	
174	The null pointer **shall not** be de-referenced.	Done with checks in software.
175	A pointer **shall not** be compared to `NULL` or be assigned `NULL`; use plain `0` instead.	Reports usage of `NULL` macro in pointer contexts.
176	A `typedef` **will** be used to simplify program syntax when declaring function pointers.	Reports non-`typedef` function pointers, or pointers to member functions for types of variables, fields, parameters. Returns type of function, cast, and exception specification.

Type Conversions

N.	JSF++ Definition	Polyspace Specification
177	User-defined conversion functions **should** be avoided.	Reports user defined conversion function, non-explicit constructor with one parameter or default value for others (even undefined ones). Does not report copy-constructor. Additional message for constructor case: This constructor should be flagged as "explicit".
178	Down casting (casting from base to derived class) **shall** only be allowed through one of the following mechanism: • Virtual functions that act like dynamic casts (most likely useful in relatively simple cases). • Use of the visitor (or similar) pattern (most likely useful in complicated cases).	Reports explicit down casting, dynamic_cast included. (Visitor patter does not have a special case.)
179	A pointer to a virtual base class **shall not** be converted to a pointer to a derived class.	Reports this specific down cast. Allows dynamic_cast.

N.	JSF++ Definition	Polyspace Specification

180	Implicit conversions that may result in a loss of information **shall not** be used.	Reports the following implicit casts :
		
		`integer => smaller integer`
		`unsigned => smaller or eq signed`
		`signed => smaller or eq un-signed`
		`integer => float float => integer`
		
		Does not report for cast to `bool` reports for implicit cast on constant done with the options `-scalar-overflows-checks signed-and-unsigned` or `-ignore-constant-overflows`
		
		Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

181	Redundant explicit casts **will not** be used.	Reports useless cast: `cast T to T`. Casts to equivalent `typedefs` are also reported.
182	Type casting from any type to or from pointers **shall not** be used.	Does not report when Rule 181 applies.
184	Floating point numbers **shall not** be converted to integers unless such a conversion is a specified algorithmic requirement or is necessary for a hardware interface.	Reports `float->int` conversions. Does not report implicit ones.
185	C++ style casts (`const_cast`, `reinterpret_cast`, and `static_cast`) **shall** be used instead of the traditional C-style casts.	

Flow Control Standards

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

N.	JSF++ Definition	Polyspace Specification
198	The initialization expression in a `for` loop **will** perform no actions other than to initialize the value of a single `for` loop parameter.	Reports if `loop` parameter cannot be determined. Assumes Rule 200 is not violated. The `loop variable` parameter is assumed to be a variable.
199	The increment expression in a `for` loop **will** perform no action other than to change a single loop parameter to the next value for the loop.	Assumes 1 loop parameter (Rule 198), with non class type. Rule 200 must not be violated for this rule to be reported.
200	Null initialize or increment expressions in `for` loops **will not** be used; a `while` loop will be used instead.	
201	Numeric variables being used within a *for* loop for iteration counting shall not be modified in the body of the loop.	Assumes 1 loop parameter (AV rule 198), and no alias writes.

Expressions

N.	JSF++ Definition	Polyspace 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: • by itself • the right-hand side of an assignment • a condition • the only argument expression with a side-effect in a function call • condition of a loop • switch condition • single part of a chained operation	Reports when: • A side effect is found in a return statement • A side effect exists on a single value, and only one operand of the function call has a side effect.

N.	JSF++ Definition	Polyspace Specification
204.1	The value of an expression shall be the same under any order of evaluation that the standard permits.	Reports when: • Variable is written more than once in an expression • Variable is read and write in sub-expressions • Volatile variable is accessed more than once **Note** Read-write operations such as ++, are only considered as a write.
205	The volatile keyword **shall not** be used unless directly interfacing with hardware.	Reports if volatile keyword is used.

Memory Allocation

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

Fault Handling

N.	JSF++ Definition	Polyspace Specification
208	C++ exceptions **shall not** be used.	Reports `try`, `catch`, `throw spec`, and `throw`.

Portable Code

N.	JSF++ Definition	Polyspace Specification
209	The basic types of `int`, `short`, `long`, `float` and `double` **shall not** be used, but specific-length equivalents should be `typedef`'d accordingly for each compiler, and these type names used in the code.	Only allows use of basic types through direct `typedefs`.

N.	JSF++ Definition	Polyspace Specification
213	No dependence shall be placed on C++'s operator precedence rules, below arithmetic operators, in expressions.	Reports when a binary operation has one operand that is not parenthesized and is an operation with inferior precedence level. Reports bitwise and shifts operators that are used without parenthesis and binary operation arguments.
215	Pointer arithmetic **will not** be used.	Reports:`p + Ip - Ip++p--p+=p-=` Allows `p[i]`.

Unsupported JSF++ Rules

- “Code Size and Complexity” on page 12-148
- “Rules” on page 12-148
- “Environment” on page 12-148
- “Libraries” on page 12-149
- “Header Files” on page 12-149
- “Style” on page 12-149
- “Classes” on page 12-149
- “Namespaces” on page 12-151
- “Templates” on page 12-151
- “Functions” on page 12-152
- “Comments” on page 12-152
- “Initialization” on page 12-153
- “Types” on page 12-153
- “Unions and Bit Fields” on page 12-153
- “Operators” on page 12-153
- “Type Conversions” on page 12-153
- “Expressions” on page 12-154
- “Memory Allocation” on page 12-154
- “Portable Code” on page 12-154

- “Efficiency Considerations” on page 12-155
- “Miscellaneous” on page 12-155
- “Testing” on page 12-155

Code Size and Complexity

N.	JSF++ Definition
2	There shall not be any self-modifying code.

Rules

N.	JSF++ Definition
4	To break a “should” rule, the following approval must be received by the developer: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)
5	To break a “will” or a “shall” rule, the following approvals must be received by the developer: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool) • approval from the software product manager (obtained by the unit approval in the developmental CM tool)
6	Each deviation from a “shall” rule shall be documented in the file that contains the deviation. Deviations from this rule shall not be allowed, AV Rule 5 notwithstanding.
7	Approval will not be required for a deviation from a “shall” or “will” rule that complies with an exception specified by that rule.

Environment

N.	JSF++ Definition
10	Values of character types will be restricted to a defined and documented subset of ISO 10646 1.

Libraries

N.	JSF++ Definition
16	Only DO-178B level A [15] certifiable or SEAL 1 C/C++ libraries shall be used with safety-critical (i.e. SEAL 1) code.

Header Files

N.	JSF++ Definition
34	Header files should contain logically related declarations only.
36	Compilation dependencies should be minimized when possible.
37	Header (include) files should include only those header files that are required for them to successfully compile. Files that are only used by the associated .cpp file should be placed in the .cpp file — not the .h file.
38	Declarations of classes that are only accessed via pointers (*) or references (&) should be supplied by forward headers that contain only forward declarations.

Style

N.	JSF++ Definition
45	All words in an identifier will be separated by the '_' character.
49	All acronyms in an identifier will be composed of uppercase letters.
55	The name of a header file should reflect the logical entity for which it provides declarations.
56	The name of an implementation file should reflect the logical entity for which it provides definitions and have a ".cpp" extension (this name will normally be identical to the header file that provides the corresponding declarations.) At times, more than one .cpp file for a given logical entity will be required. In these cases, a suffix should be appended to reflect a logical differentiation.

Classes

N.	JSF++ Definition
64	A class interface should be complete and minimal.
65	A structure should be used to model an entity that does not require an invariant.

N.	JSF++ Definition
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.
72	The invariant for a class should be: • A part of the postcondition of every class constructor, • A part of the precondition of the class destructor (if any), • A part of the precondition and postcondition of every other publicly accessible operation.
73	Unnecessary default constructors shall not be defined.
77	A copy constructor shall copy all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).
80	The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.
84	Operator overloading will be used sparingly and in a conventional manner.
85	When two operators are opposites (such as == and !=), both will be defined and one will be defined in terms of the other.
86	Concrete types should be used to represent simple independent concepts.
87	Hierarchies should be based on abstract classes.
90	Heavily used interfaces should be minimal, general and abstract.
91	Public inheritance will be used to implement “is-a” relationships.

N.	JSF++ Definition
92	A subtype (publicly derived classes) will conform to the following guidelines with respect to all classes involved in the polymorphic assignment of different subclass instances to the same variable or parameter during the execution of the system: • Preconditions of derived methods must be at least as weak as the preconditions of the methods they override. • Postconditions of derived methods must be at least as strong as the postconditions of the methods they override. In other words, subclass methods must expect less and deliver more than the base class methods they override. This rule implies that subtypes will conform to the Liskov Substitution Principle.
93	“has-a” or “is-implemented-in-terms-of” relationships will be modeled through membership or non-public inheritance.

Namespaces

N.	JSF++ Definition
100	Elements from a namespace should be selected as follows: • using declaration or explicit qualification for few (approximately five) names, • using directive for many names.

Templates

N.	JSF++ Definition
101	Templates shall be reviewed as follows: **1** with respect to the template in isolation considering assumptions or requirements placed on its arguments. **2** with respect to all functions instantiated by actual arguments.
102	Template tests shall be created to cover all actual template instantiations.
103	Constraint checks should be applied to template arguments.
105	A template definition’s dependence on its instantiation contexts should be minimized.
106	Specializations for pointer types should be made where appropriate.

Functions

N.	JSF++ Definition
112	Function return values should not obscure resource ownership.
115	If a function returns error information, then that error information will be tested.
117	Arguments should be passed by reference if NULL values are not possible: • **117.1** – An object should be passed as `const T&` if the function should not change the value of the object. • **117.2** – An object should be passed as `T&` if the function may change the value of the object.
118	Arguments should be passed via pointers if NULL values are possible: • **118.1** – An object should be passed as `const T*` if its value should not be modified. • **118.2** – An object should be passed as `T*` if its value may be modified.
120	Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal 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.

N.	JSF++ Definition
131	One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).
132	Each variable declaration, typedef, enumeration value, and structure member will be commented.
134	Assumptions (limitations) made by functions should be documented in the function's preamble.

Initialization

N.	JSF++ Definition
143	Variables will not be introduced until they can be initialized with meaningful values. (See also AV Rule 136, AV Rule 142, and AV Rule 73 concerning declaration scope, initialization before use, and default constructors respectively.)

Types

N.	JSF++ Definition
146	Floating point implementations shall comply with a defined floating point standard. The standard that will be used is the ANSI/IEEE® Std 754 [1].

Unions and Bit Fields

N.	JSF++ Definition
155	Bit-fields will not be used to pack data into a word for the sole purpose of saving space.

Operators

N.	JSF++ Definition
167	The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.

Type Conversions

N.	JSF++ Definition
183	Every possible measure should be taken to avoid type casting.

Expressions

N.	JSF++ Definition
204	A single operation with side-effects shall only be used in the following contexts: **1** by itself **2** the right-hand side of an assignment **3** a condition **4** the only argument expression with a side-effect in a function call **5** condition of a loop **6** switch condition **7** single part of a chained operation

Memory Allocation

N.	JSF++ Definition
207	Unencapsulated global data will be avoided.

Portable Code

N.	JSF++ Definition
210	Algorithms shall not make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.).
210.1	Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.
211	Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.
212	Underflow or overflow functioning shall not be depended on in any special way.
214	Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.

Efficiency Considerations

N.	JSF++ Definition
216	Programmers should not attempt to prematurely optimize code.

Miscellaneous

N.	JSF++ Definition
217	Compile-time and link-time errors should be preferred over run-time errors.
218	Compiler warning levels will be set in compliance with project policies.

Testing

N.	JSF++ Definition
219	All tests applied to a base class interface shall be applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions /invariants shall be substituted in the derived class tests.
220	Structural coverage algorithms shall be applied against flattened classes.
221	Structural coverage of a class within an inheritance hierarchy containing virtual functions shall include testing every possible resolution for each set of identical polymorphic references.

13

Configure Bug Finder Checkers

- “Choose Specific Bug Finder Defect Checkers” on page 13-2
- “Bug Finder Defect Groups” on page 13-3
- “Results Found by Fast Analysis” on page 13-9
- “Check C/C++ Code for Security Standards” on page 13-42
- “CWE Coding Standard and Polyspace Results” on page 13-49
- “Mapping Between CWE-658 or 659 and Polyspace Results” on page 13-79
- “CERT C Coding Standard and Polyspace Results” on page 13-97
- “ISO/IEC TS 17961 Coding Standard and Polyspace Results” on page 13-163

Choose Specific Bug Finder Defect Checkers

You can check your C/C++ code using the predefined subsets of defect checkers in Bug Finder. However, 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.
   - `CWE`, `CERT-rules`, `CERT-all`, or `ISO-17961`, for defects related to a security standard.

     For more information, see "Check C/C++ Code for Security Standards" on page 13-42.
   - `custom` to add defects to the default list or remove defects from it.

You can use a spreadsheet to keep track of the defect checkers that you enable and add notes explaining why you do not enable the other checkers. A spreadsheet of checkers is provided in *`matlabroot`*`\polyspace\resources`. Here, *`matlabroot`* is the MATLAB installation folder, such as `C:\Program Files\MATLAB\R2017b`.

To standardize the bug finding across your organization, you can save your list of defect checkers as a configuration template and share with others. See "Create Project Using Configuration Template" on page 1-19.

See Also

`Find defects (-checkers)`

More About

- "Create Project Using Configuration Template" on page 1-19

Bug Finder Defect Groups

In this section...

"Concurrency" on page 13-3
"Cryptography" on page 13-4
"Data flow" on page 13-4
"Dynamic Memory" on page 13-5
"Good Practice" on page 13-5
"Numerical" on page 13-5
"Object Oriented" on page 13-6
"Programming" on page 13-6
"Resource Management" on page 13-6
"Static Memory" on page 13-7
"Security" on page 13-7
"Tainted data" on page 13-7

For convenience, the defect checkers in Bug Finder are classified into various groups.

- In certain projects, you can choose to focus only on specific groups of defects. Specify the group name for the option `Find defects (-checkers)`.
- When reviewing results, you can review all results of a certain group together. Filter out other results during review. See "Filter and Group Results" on page 16-2.

This topic gives an overview of the various groups.

Concurrency

These defects are related to multitasking code.

Data Race Defects

The data race defects occur when multiple tasks operate on a shared variable or call a nonreentrant standard library function without protection.

For the specific defects, see "Concurrency Defects".

Command-Line Parameter: `concurrency`

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".

Command-Line Parameter: `concurrency`

Cryptography

These defects are related to incorrect use of cryptography routines from the OpenSSL library. For instance:

- Use of cryptographically weak algorithms
- Absence of essential elements such as cipher key or initialization vector
- Wrong order of cryptographic operations

For the specific defects, see "Cryptography Defects".

Command-Line Parameter: `cryptography`

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".

Command-Line Parameter: `data_flow`

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".

Command-Line Parameter: `dynamic_memory`

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".

Command-Line Parameter: `good_practice`

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".

Command-Line Parameter: `numerical`

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".

Command-Line Parameter: `object_oriented`

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".

Command-Line Parameter: `programming`

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”.

Command-Line Parameter: `resource_management`

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”.

Command-Line Parameter: `static_memory`

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”.

Command-Line Parameter: `security`

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”.

Command-Line Parameter: `tainted_data`

Results Found by Fast Analysis

In fast analysis mode, Bug Finder checks for a subset of defects and coding rules only. These defects and rules can be found within a single compilation unit, such as a single function or file. The software does not perform interprocedural or cross-functional analysis.

The tables below list the results that can be found in a fast analysis. Besides these, some other results can also be found in fast analysis mode provided Bug Finder can determine the results from a single file only.

Polyspace Bug Finder Defects

Static Memory

Name	Description
Buffer overflow from incorrect string format specifier (`str_format_buffer_overflow`)	String format specifier causes buffer argument of standard library functions to overflow
Unreliable cast of function pointer (`func_cast`)	Function pointer cast to another function pointer with different argument or return type
Unreliable cast of pointer (`ptr_cast`)	Pointer implicitly cast to different data type

Programming

Name	Description
Copy of overlapping memory (`overlapping_copy`)	Source and destination arguments of a copy function have overlapping memory
Exception caught by value (`excp_caught_by_value`)	`catch` statement accepts an object by value
Exception handler hidden by previous handler (`excp_handler_hidden`)	`catch` statement is not reached because of an earlier catch statement for the same exception
Format string specifiers and arguments mismatch (`string_format`)	String specifiers do not match corresponding arguments
Improper array initialization (`improper_array_init`)	Incorrect array initialization when using initializers
Invalid use of == (equality) operator (`bad_equal_equal_use`)	Equality operation in assignment statement
Invalid use of = (assignment) operator (`bad_equal_use`)	Assignment in conditional statement
Invalid use of floating point operation (`bad_float_op`)	Imprecise comparison of floating point variables
Missing null in string array (`missing_null_char`)	String does not terminate with null character
Overlapping assignment (`overlapping_assign`)	Memory overlap between left and right sides of an assignment
Possibly unintended evaluation of expression because of operator precedence rules (`operator_precedence`)	Operator precedence rules cause unexpected evaluation order in arithmetic expression
Unsafe conversion between pointer and integer (`bad_int_ptr_cast`)	Misaligned or invalid results from conversions between pointer and integer types
Wrong type used in sizeof (`ptr_sizeof_mismatch`)	`sizeof` argument does not match pointed type

Data Flow

Name	Description
Code deactivated by constant false condition (`deactivated_code`)	Code segment deactivated by `#if 0` directive or `if(0)` condition
Missing return statement (`missing_return`)	Function does not return value though return type is not void
Static uncalled function (`uncalled_func`)	Function with static scope not called in file
Variable shadowing (`var_shadowing`)	Variable hides another variable of same name with nested scope

Object Oriented

Name	Description
`*this` not returned in copy assignment operator (`return_not_ref_to_this`)	operator= method does not return a pointer to the current object
Base class assignment operator not called (`missing_base_assign_op_call`)	Copy assignment operator does not call copy assignment operators of base subobjects
Base class destructor not virtual (`dtor_not_virtual`)	Class cannot behave polymorphically for deletion of derived class objects
Copy constructor not called in initialization list (`missing_copy_ctor_call`)	Copy constructor does not call copy constructors of some members or base classes
Incompatible types prevent overriding (`virtual_func_hiding`)	Derived class method hides a virtual base class method instead of overriding it
Member not initialized in constructor (`non_init_member`)	Constructor does not initialize some members of a class
Missing explicit keyword (`missing_explicit_keyword`)	Constructor missing the explicit specifier
Missing virtual inheritance (`missing_virtual_inheritance`)	A base class is inherited virtually and nonvirtually in the same hierarchy
Object slicing (`object_slicing`)	Derived class object passed by value to function with base class parameter
Partial override of overloaded virtual functions (`partial_override`)	Class overrides fraction of inherited virtual functions with a given name
Return of non const handle to encapsulated data member (`breaking_data_encapsulation`)	Method returns pointer or reference to internal member of object
Self assignment not tested in operator (`missing_self_assign_test`)	Copy assignment operator does not test for self-assignment

Security

Name	Description
Function pointer assigned with absolute address (`func_ptr_absolute_addr`)	Constant expression is used as function address is vulnerable to code injection

Good Practice

Name	Description
Bitwise and arithmetic operation on the same data (`bitwise_arith_mix`)	Statement with mixed bitwise and arithmetic operations
Delete of void pointer (`delete_of_void_ptr`)	delete operates on a `void*` pointer pointing to an object
Hard-coded buffer size (`hard_coded_buffer_size`)	Size of memory buffer is a numerical value instead of symbolic constant
Hard-coded loop boundary (`hard_coded_loop_boundary`)	Loop boundary is a numerical value instead of symbolic constant
Large pass-by-value argument (`pass_by_value`)	Large argument passed by value between functions
Line with more than one statement (`more_than_one_statement`)	Multiple statements on a line
Missing break of switch case (`missing_switch_break`)	No comments at the end of switch case without a break statement
Missing reset of a freed pointer (`missing_freed_ptr_reset`)	Pointer free not followed by a reset statement to clear leftover data
Unused parameter (`unused_parameter`)	Function prototype has parameters not read or written in function body

MISRA C: 2004 and MISRA AC AGC Rules

The software checks the following rules early in the analysis.

Language Extensions

Rule	Description
2.1	Assembly language shall be encapsulated and isolated.
2.2	Source code shall only use `/* */` style comments.
2.3	The character sequence `/*` shall not be used within a comment.

Documentation

Rule	Description
3.4	All uses of the `#pragma` directive shall be documented and explained.

Character Sets

Rule	Description
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.
4.2	Trigraphs shall not be used.

Identifiers

Rule	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.

Types

Rule	Description
6.1	The plain char type shall be used only for the storage and use of character values.
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.
6.3	`typedef`s that indicate size and signedness should be used in place of the basic types.
6.4	Bit fields shall only be defined to be of type `unsigned int` or `signed int`.
6.5	Bit fields of type `signed int` shall be at least 2 bits long.

Constants

Rule	Description
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and Definitions

Rule	Description
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
8.2	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.
8.5	There shall be no definitions of objects or functions in a header file.
8.6	Functions shall always be declared at file scope.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function.
8.8	An external object or function shall be declared in one file and only one file.
8.9	An identifier with external linkage shall have exactly one external definition.
8.11	The `static` 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.

Initialization

Rule	Description
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.

Arithmetic Type Conversion

Rule	Description
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: • It is not a conversion to a wider integer type of the same signedness, or • The expression is complex, or • The expression is not constant and is a function argument, or • The expression is not constant and is a return expression
10.2	The value of an expression of floating type shall not be implicitly converted to a different type if • It is not a conversion to a wider floating type, or • The expression is complex, or • The expression is a function argument, or • The expression is a return expression
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.
10.4	The value of a complex expression of float type may only be cast to narrower floating type.
10.5	If the bitwise operator ~ and << are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand
10.6	The "U" suffix shall be applied to all constants of `unsigned` types.

Pointer Type Conversion

Rule	Description
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.
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.
11.5	A cast shall not be performed that removes any `const` or `volatile` qualification from the type addressed by a pointer

Expressions

Rule	Description
12.1	Limited dependence should be placed on C's operator precedence rules in expressions.
12.3	The `sizeof` operator should not be used on expressions that contain side effects.
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 `!`).
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed.
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.
12.11	Evaluation of constant unsigned expression should not lead to wraparound.
12.12	The underlying bit representations of floating-point values shall not be used.
12.13	The increment (`++`) and decrement (`--`) operators should not be mixed with other operators in an expression

Control Statement Expressions

Rule	Description
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 `for` statement shall not contain any objects of floating type.
13.5	The three expressions of a `for` statement shall be concerned only with loop control.
13.6	Numeric variables being used within a `for` loop for iteration counting should not be modified in the body of the loop.

Control Flow

Rule	Description
14.3	All non-null statements shall either • have at least one side effect however executed, or • cause control flow to change.
14.4	The `goto` statement shall not be used.
14.5	The `continue` statement shall not be used.
14.6	For any iteration statement, there shall be at most one `break` statement used for loop termination.
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 `switch`, `while`, `do while` or `for` statement shall be a compound statement.
14.9	An `if` (expression) construct shall be followed by a compound statement. The `else` keyword shall be followed by either a compound statement, or another `if` statement.
14.10	All `if else if` constructs should contain a final `else` clause.

Switch Statements

Rule	Description
15.0	Unreachable code is detected between `switch` statement and first `case`.
15.1	A `switch` label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement
15.2	An unconditional `break` statement shall terminate every non-empty `switch` clause.
15.3	The final clause of a `switch` statement shall be the `default` clause.
15.4	A `switch` expression should not represent a value that is effectively Boolean.
15.5	Every `switch` statement shall have at least one `case` clause.

Functions

Rule	Description
16.1	Functions shall not be defined with variable numbers of arguments.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.
16.5	Functions with no parameters shall be declared with parameter type `void`.
16.6	The number of arguments passed to a function shall match the number of parameters.
16.8	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.

Pointers and Arrays

Rule	Description
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	A type should not contain more than 2 levels of pointer indirection.

Structures and Unions

Rule	Description
18.1	All structure or union types shall be complete at the end of a translation unit.
18.4	Unions shall not be used.

Preprocessing Directives

Rule	Description
19.1	`#include` statements in a file shall only be preceded by other preprocessors directives or comments.
19.2	Nonstandard characters should not occur in header file names in `#include` directives.
19.3	The `#include` directive shall be followed by either a <filename> or "filename" sequence.
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.
19.5	Macros shall not be `#define`-d and `#undef`-d within a block.
19.6	`#undef` shall not be used.
19.7	A function should be used in preference to a function like-macro.
19.8	A function-like macro shall not be invoked without all of its arguments.
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.
19.13	The # and ## preprocessor operators should not be used.
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.
19.17	All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related.

Standard Libraries

Rule	Description
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.
20.2	The names of standard library macros, objects and functions shall not be reused.
20.4	Dynamic heap memory allocation shall not be used.
20.5	The error indicator `errno` shall not be used.
20.6	The macro `offsetof`, in library `<stddef.h>`, shall not be used.
20.7	The `setjmp` macro and the `longjmp` function shall not be used.
20.8	The signal handling facilities of `<signal.h>` shall not be used.
20.9	The input/output library `<stdio.h>` shall not be used in production code.
20.10	The library functions `atof`, `atoi` and `atoll` from library `<stdlib.h>` shall not be used.
20.11	The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` shall not be used.
20.12	The time handling functions of library `<time.h>` shall not be used.

MISRA C: 2012 Rules

Standard C Environment

Rule	Description
1.1	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.
1.2	Language extensions should not be used.

Unused Code

Rule	Description
2.6	A function should not contain unused label declarations.
2.7	There should be no unused parameters in functions.

Comments

Rule	Description
3.1	The character sequences /* and // shall not be used within a comment.
3.2	Line-splicing shall not be used in // comments.

Character Sets and Lexical Conventions

Rule	Description
4.1	Octal and hexadecimal escape sequences shall be terminated.
4.2	Trigraphs should not be used.

Identifiers

Rule	Description
5.2	Identifiers declared in the same scope and name space shall be distinct.
5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
5.4	Macro identifiers shall be distinct.
5.5	Identifiers shall be distinct from macro names.

Types

Rule	Description
6.1	Bit-fields shall only be declared with an appropriate type.
6.2	Single-bit named bit fields shall not be of a signed type.

Literals and Constants

Rule	Description
7.1	Octal constants shall not be used.
7.2	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.
7.3	The lowercase character "l" shall not be used in a literal suffix.
7.4	A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Declarations and Definitions

Rule	Description
8.1	Types shall be explicitly specified.
8.2	Function types shall be in prototype form with named parameters.
8.4	A compatible declaration shall be visible when an object or function with external linkage is defined.
8.5	An external object or function shall be declared once in one and only one file.
8.6	An identifier with external linkage shall have exactly one external definition.
8.8	The `static` storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.10	An inline function shall be declared with the `static` storage class.
8.11	When an array with external linkage is declared, its size should be explicitly specified.
8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.
8.14	The `restrict` type qualifier shall not be used.

Initialization

Rule	Description
9.2	The initializer for an aggregate or union shall be enclosed in braces.
9.3	Arrays shall not be partially initialized.
9.4	An element of an object shall not be initialized more than once.
9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

The Essential Type Model

Rule	Description
10.1	Operands shall not be of an inappropriate essential type.
10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.
10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.
10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
10.5	The value of an expression should not be cast to an inappropriate essential type.
10.6	The value of a composite expression shall not be assigned to an object with wider essential type.
10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.
10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Pointer Type Conversion

Rule	Description
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.
11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object 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.
11.9	The macro `NULL` shall be the only permitted form of integer null pointer constant.

Expressions

Rule	Description
12.1	The precedence of operators within expressions should be made explicit.
12.3	The comma operator should not be used.
12.4	Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Side Effects

Rule	Description
13.3	A full expression containing an increment (`++`) or decrement (`--`) operator should have no other potential side effects other than that caused by the increment or decrement operator.
13.4	The result of an assignment operator should not be used.
13.6	The operand of the `sizeof` operator shall not contain any expression which has potential side effects.

Control Statement Expressions

Rule	Description
14.4	The controlling expression of an `if` statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Control Flow

Rule	Description
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.
15.4	There should be no more than one `break` or `goto` statement used to terminate any iteration 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.

Switch Statements

Rule	Description
16.1	All `switch` statements shall be well-formed.
16.2	A `switch` label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.
16.3	An unconditional `break` statement shall terminate every `switch`-clause.
16.4	Every `switch` statement shall have a `default` label.
16.5	A `default` label shall appear as either the first or the last `switch` label of a `switch` statement.
16.6	Every `switch` statement shall have at least two `switch`-clauses.
16.7	A `switch`-expression shall not have essentially Boolean type.

Functions

Rule	Description
17.1	The features of `<starg.h>` shall not be used.
17.3	A function shall not be declared implicitly.
17.4	All exit paths from a function with non-`void` return type shall have an explicit return statement with an expression.
17.6	The declaration of an array parameter shall not contain the `static` keyword between the `[ ]`.
17.7	The value returned by a function having non-`void` return type shall be used.

Pointers and Arrays

Rule	Description
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.
18.7	Flexible array members shall not be declared.
18.8	Variable-length array types shall not be used.

Overlapping Storage

Rule	Description
19.2	The `union` keyword should not be used.

Preprocessing Directives

Rule	Description
20.1	`#include` directives should only be preceded by preprocessor directives or comments.
20.2	The `'`, `"`, or `\` characters and the `/*` or `//` character sequences shall not occur in a header file name.
20.3	The `#include` directive shall be followed by either a <filename> or \"filename\" sequence.
20.4	A macro shall not be defined with the same name as a keyword.
20.5	`#undef` should not be used.
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.
20.8	The controlling expression of a `#if` or `#elif` preprocessing directive shall evaluate to 0 or 1.
20.9	All identifiers used in the controlling expression of `#if` or `#elif` preprocessing directives shall be `#define`'d before evaluation.
20.10	The # and ## preprocessor operators should not be used.
20.11	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.
20.12	A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.
20.13	A line whose first token is # shall be a valid preprocessing directive.
20.14	All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if`, `#ifdef` or `#ifndef` directive to which they are related.

Standard Libraries

Rule	Description
21.1	#define and #undef shall not be used on a reserved identifier or reserved macro name.
21.2	A reserved identifier or macro name shall not be declared.
21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used.
21.4	The standard header file <setjmp.h> shall not be used.
21.5	The standard header file <signal.h> shall not be used.
21.6	The Standard Library input/output functions shall not be used.
21.7	The atof, atoi, atol, and atoll functions of <stdlib.h> shall not be used.
21.8	The library functions abort, exit, getenv and system of <stdlib.h> shall not be used.
21.9	The library functions bsearch and qsort of <stdlib.h> shall not be used.
21.10	The Standard Library time and date functions shall not be used.
21.11	The standard header file <tgmath.h> shall not be used.
21.12	The exception handling features of <fenv.h> should not be used.

MISRA C++ 2008 Rules

Language Independent Issues

Rule	Description
0-1-7	The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.
0-1-11	There shall be no unused parameters (named or unnamed) in non- virtual functions.
0-1-12	There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.
0-2-1	An object shall not be assigned to an overlapping object.

General

Rule	Description
1-0-1	All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".

Lexical Conventions

Rule	Description
2-3-1	Trigraphs shall not be used.
2-5-1	Digraphs should not be used.
2-7-1	The character sequence /* shall not be used within a C-style comment.
2-10-1	Different identifiers shall be typographically unambiguous.
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
2-10-3	A typedef name (including qualification, if any) shall be a unique identifier.
2-10-4	A class, union or enum name (including qualification, if any) shall be a unique identifier.
2-10-6	If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.
2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.
2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.
2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.
2-13-4	Literal suffixes shall be upper case.
2-13-5	Narrow and wide string literals shall not be concatenated.

Basic Concepts

Rule	Description
3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.
3-1-2	Functions shall not be declared at block scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-1	Objects or functions with external linkage shall be declared in a header file.
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.
3-9-1	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.
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.

Standard Conversions

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

Expressions

Rule	Description
5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.
5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions.
5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type.
5-0-4	An implicit integral conversion shall not change the signedness of the underlying type.
5-0-5	There shall be no implicit floating-integral conversions.
5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type.
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.
5-0-11	The plain char type shall only be used for the storage and use of character values.
5-0-12	signed char and unsigned char type shall only be used for the storage and use of numeric values.
5-0-13	The condition of an if-statement and the condition of an iteration-statement shall have type bool.
5-0-14	The first operand of a conditional-operator shall have type bool.
5-0-15	Array indexing shall be the only form of pointer arithmetic.
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.
5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type.
5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type.

<table>
<tr><th>Rule</th><th>Description</th></tr>
<tr><td>5-2-1</td><td>Each operand of a logical && or || shall be a postfix - expression.</td></tr>
<tr><td>5-2-2</td><td>A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.</td></tr>
<tr><td>5-2-3</td><td>Casts from a base class to a derived class should not be performed on polymorphic types.</td></tr>
<tr><td>5-2-4</td><td>C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.</td></tr>
<tr><td>5-2-5</td><td>A cast shall not remove any const or volatile qualification from the type of a pointer or reference.</td></tr>
<tr><td>5-2-6</td><td>A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.</td></tr>
<tr><td>5-2-7</td><td>An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.</td></tr>
<tr><td>5-2-8</td><td>An object with integer type or pointer to void type shall not be converted to an object with pointer type.</td></tr>
<tr><td>5-2-9</td><td>A cast should not convert a pointer type to an integral type.</td></tr>
<tr><td>5-2-10</td><td>The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.</td></tr>
<tr><td>5-2-11</td><td>The comma operator, && operator and the || operator shall not be overloaded.</td></tr>
<tr><td>5-2-12</td><td>An identifier with array type passed as a function argument shall not decay to a pointer.</td></tr>
<tr><td>5-3-1</td><td>Each operand of the ! operator, the logical && or the logical || operators shall have type bool.</td></tr>
<tr><td>5-3-2</td><td>The unary minus operator shall not be applied to an expression whose underlying type is unsigned.</td></tr>
<tr><td>5-3-3</td><td>The unary & operator shall not be overloaded.</td></tr>
<tr><td>5-3-4</td><td>Evaluation of the operand to the sizeof operator shall not contain side effects.</td></tr>
<tr><td>5-8-1</td><td>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.</td></tr>
<tr><td>5-14-1</td><td>The right hand operand of a logical && or || operator shall not contain side effects.</td></tr>
<tr><td>5-18-1</td><td>The comma operator shall not be used.</td></tr>
<tr><td>5-19-1</td><td>Evaluation of constant unsigned integer expressions should not lead to wrap-around.</td></tr>
</table>

Statements

Rule	Description
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.
6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character.
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.
6-4-1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.
6-4-2	All if ... else if constructs shall be terminated with an else clause.
6-4-3	A switch statement shall be a well-formed switch statement.
6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.
6-4-5	An unconditional throw or break statement shall terminate every non - empty switch-clause.
6-4-6	The final clause of a switch statement shall be the default-clause.
6-4-7	The condition of a switch statement shall not have bool type.
6-4-8	Every switch statement shall have at least one case-clause.
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.
6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression.
6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

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.
6-6-3	The continue statement shall only be used within a well-formed for loop.
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.

Declarations

Rule	Description
7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations.
7-3-2	The identifier main shall not be used for a function other than the global function main.
7-3-3	There shall be no unnamed namespaces in header files.
7-3-4	using-directives shall not be used.
7-3-5	Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.
7-3-6	using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.
7-4-2	Assembler instructions shall only be introduced using the asm declaration.
7-4-3	Assembly language shall be encapsulated and isolated.

Declarators

Rule	Description
8-0-1	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.
8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.
8-4-1	Functions shall not be defined using the ellipsis notation.
8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.
8-4-3	All exit paths from a function with non- void return type shall have an explicit return statement with an expression.
8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.
8-5-2	Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.
8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Classes

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

Derived Classes

Rule	Description
10-1-1	Classes should not be derived from virtual bases.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.
10-2-1	All accessible entity names within a multiple inheritance hierarchy should be unique.
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.

Member Access Control

Rule	Description
11-0-1	Member data in non- POD class types shall be private.

Special Member Functions

Rule	Description
12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.
12-1-2	All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.
12-1-3	All constructors that are callable with a single argument of fundamental type shall be declared explicit.
12-8-1	A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member.
12-8-2	The copy assignment operator shall be declared protected or private in an abstract class.

Templates

Rule	Description
14-5-2	A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.
14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.
14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.
14-6-2	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.
14-7-3	All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.
14-8-1	Overloaded function templates shall not be explicitly specialized.
14-8-2	The viable function set for a function call should either contain no function specializations, or only contain function specializations.

Exception Handling

Rule	Description
15-0-2	An exception object should not have pointer type.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-2	NULL shall not be thrown explicitly.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.
15-3-2	There should be at least one exception handler to catch all otherwise unhandled exceptions
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-5-1	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).

Preprocessing Directives

Rule	Description
16-0-1	#include directives in a file shall only be preceded by other preprocessor directives or comments.
16-0-2	Macros shall only be #define 'd or #undef 'd in the global namespace.
16-0-3	#undef shall not be used.
16-0-4	Function-like macros shall not be defined.
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.
16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.
16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.
16-1-1	The defined preprocessor operator shall only be used in one of the two standard forms.
16-1-2	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.
16-2-1	The pre-processor shall only be used for file inclusion and include guards.
16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.
16-2-3	Include guards shall be provided.
16-2-4	The ', ", /* or // characters shall not occur in a header file name.
16-2-5	The \ character should not occur in a header file name.
16-2-6	The #include directive shall be followed by either a <filename> or "filename" sequence.
16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.
16-3-2	The # and ## operators should not be used.
16-6-1	All uses of the #pragma directive shall be documented.
17-0-1	Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

Rule	Description
17-0-2	The names of standard library macros and objects shall not be reused.
17-0-5	The setjmp macro and the longjmp function shall not be used.

Language Support Library

Rule	Description
18-0-1	The C library shall not be used.
18-0-2	The library functions atof, atoi and atol from library <cstdlib> shall not be used.
18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.
18-0-4	The time handling functions of library <ctime> shall not be used.
18-0-5	The unbounded functions of library <cstring> shall not be used.
18-2-1	The macro offsetof shall not be used.
18-4-1	Dynamic heap memory allocation shall not be used.
18-7-1	The signal handling facilities of <csignal> shall not be used.

Diagnostic Library

Rule	Description
19-3-1	The error indicator errno shall not be used.

Input/Output Library

Rule	Description
27-0-1	The stream input/output library <cstdio> shall not be used.

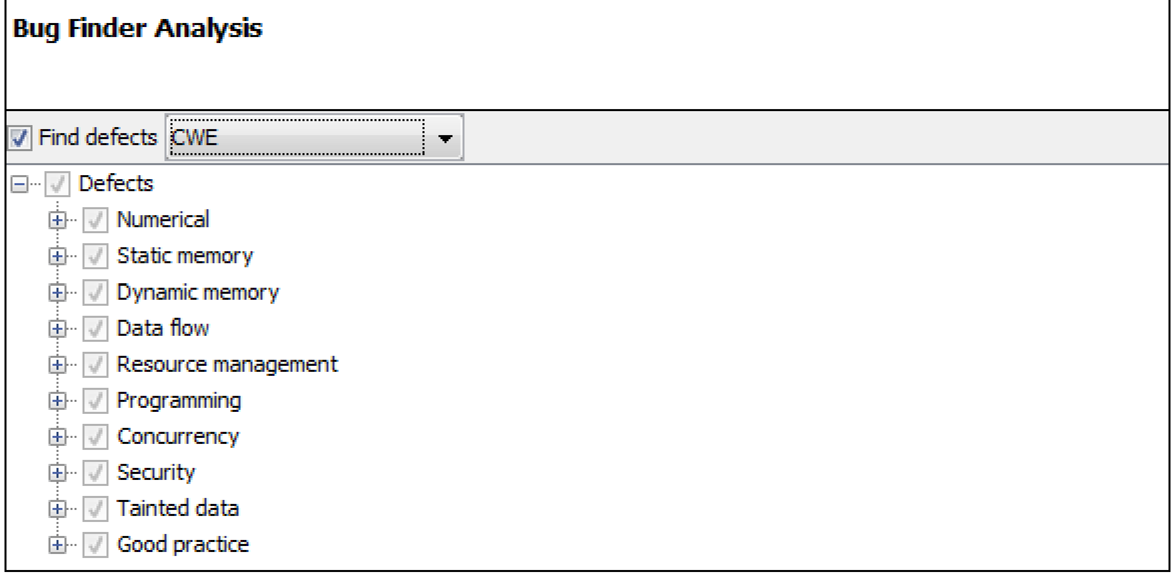
Check C/C++ Code for Security Standards

Using results of a Polyspace analysis, you can check your code for the following security standards:

- CWE: See also “CWE Coding Standard and Polyspace Results” on page 13-49.
- CERT C99: See also “CERT C Coding Standard and Polyspace Results” on page 13-97.
- ISO/IEC TS 17961: See also “ISO/IEC TS 17961 Coding Standard and Polyspace Results” on page 13-163.

To adhere to a security standard, follow this workflow.

Step 1: Check Code Against Standard



Check your code for the subset of defects and coding rules that correspond to the standard.

- CWE: Use the `CWE` subset for the option `Find defects (-checkers)`.
- CERT C99: Use both the option to check defects and the option to check coding rules.

    - `Find defects (-checkers)`: Use `CERT-rules` or `CERT-all`.
    - `Check MISRA C:2012 (-misra3)`: Use `CERT-rules` or `CERT-all`.

  If you run a Code Prover analysis, the run-time errors are mapped to the CERT C standard. All Code Prover run-time checkers are enabled by default.
- ISO/IEC TS 17961: Use both the option to check defects and the option to check coding rules.

    - `Find defects (-checkers)`: Use `ISO-17961`.
    - `Check MISRA C:2012 (-misra3)`: Use `ISO-17961`.

Additional Information

- *Can I look for more defects than the subset that corresponds to the standard?*

  Choose `all` for the options to find defects and coding rules. The analysis looks for all results that it can find, including results mapped to the standard.

  You can later filter out results that do not map to a security standard.
- *Can I look for specific IDs instead of all supported IDs from a standard?*

  Choose `custom` for the options to find defects and coding rules. Select defects and coding rules corresponding to specific IDs only.

  Save your configuration as a template so that you can reuse it later.

  For information on:

    - Which defect or coding rule maps to which ID, see CWE on page 13-49, CERT C99 on page 13-97 or ISO/IEC TS 17961 on page 13-163.
    - Using configuration templates, see “Create Project Using Configuration Template” on page 1-19.

Step 2: See Results with IDs from Standard

Results List

All results | New | Showing 1,971/1,971

Family	Type	Check	CWE ID
○	Defect	Invalid use of == operator	CWE-482
○	Defect	Invalid free of pointer	CWE-404 CWE-590 CWE-762
○	Defect	Missing unlock	CWE-667
○	Defect	Bad order of dropping privileges	CWE-250 CWE-696
○	Defect	Bad order of dropping privileges	CWE-250 CWE-696
○	Defect	Use of previously closed resource	CWE-672
○	Defect	Writing to const qualified object	CWE-227 CWE-471 CWE-686
○	Defect	Data race	CWE-366
○	Defect	Data race	CWE-366
○	Defect	Data race through standard library function call	CWE-366
○	Defect	Deadlock	CWE-833
○	Defect	Declaration mismatch	CWE-685 CWE-686
○	Defect	Deallocation of previously deallocated pointer	CWE-415
○	Defect	Double lock	CWE-764
○	Defect	Closing previously closed resource	CWE-672
○	Defect	Double unlock	CWE-765
○	Defect	Absorption of float operand	CWE-682 CWE-873

After analysis, see results that correspond to the security standard.

To see the IDs from a security standard, on the **Results List** pane, check the **CWE ID**, **CERT ID** or **ISO-17961 ID** column. If you do not see the column, right-click any column header and enable the column.

Additional Information

- *If I did not choose a security standard before analysis, can I focus on the subset after analysis?*

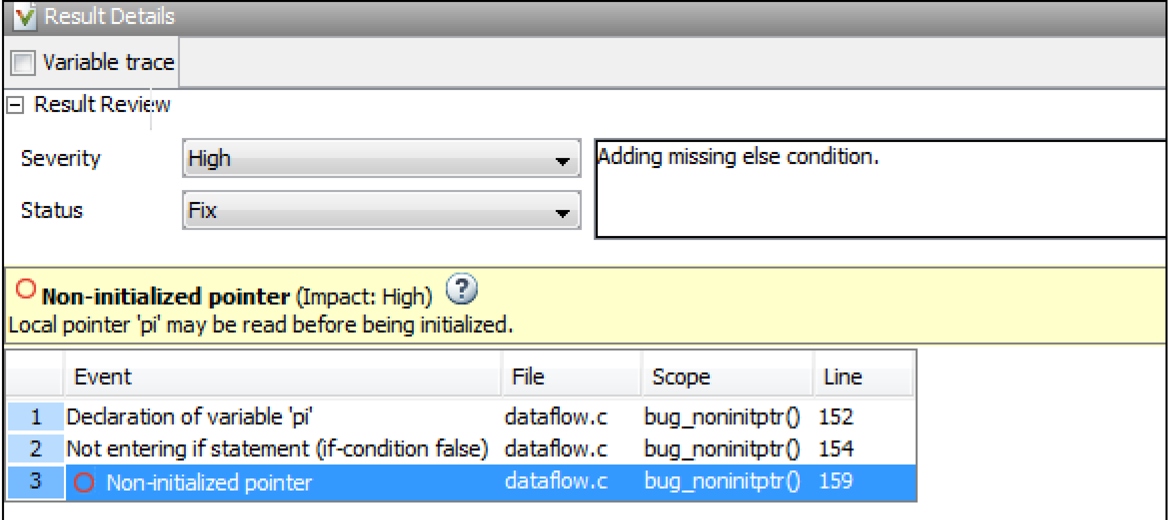
  Narrow your review scope only to results that correspond to a security standard. Instead of `All results` in **Results List**, select `CWE checks`, `CERT checks` or `ISO-17961 checks`.

- *If both a defect and coding rule corresponds to the same security standard ID, will the analysis show both results?*

  The defect and coding rule violation both appear in your results list.

  If you fix the issue, both results disappear in the next run. If you justify the issue, add your comments for one result and use auto-completion for the other.

Step 3: Fix or Justify Results with Standard IDs



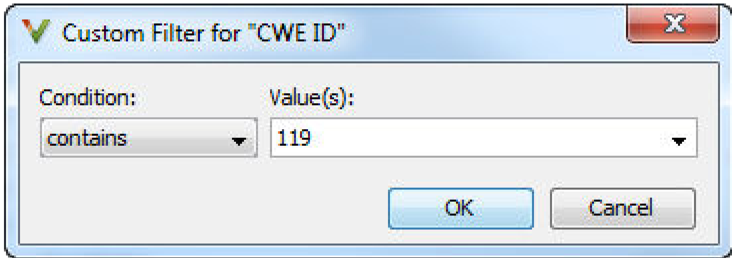
Fix or justify each result. To keep track of your progress, assign the status,`To fix` or `Justified`. For results that you justified, enter comments with your rationale.

Additional Information

- *Can I focus on a single ID after analysis? For instance, can I review all violations of a specific CWE ID together?*

  You can filter all results that correspond to a specific ID and review them together.

  For instance, on the **CWE ID** column, click the (filter) icon. From the drop-down list, select **Custom**. Use the `contains` filter.



- *Can I review only specific IDs*?

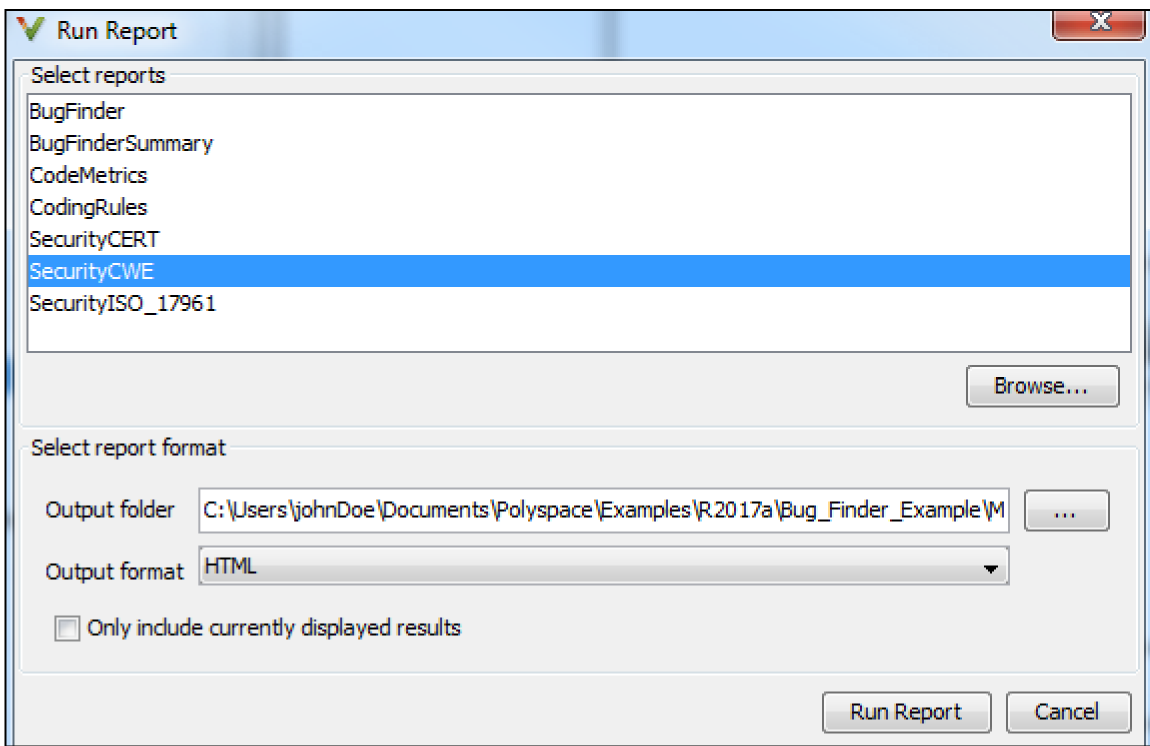
  If you ran analysis for all IDs from a standard but want to focus on specific IDs only:

  1. *Address each desired ID individually*: Use the custom filter to filter each ID that you want to focus on. Review the results for that ID. In other words, fix or justify the results. Assign the status, `To fix` or `Justified`. For results that you justified, enter comments with your rationale.
  2. *Filter out addressed IDs*: Filter out results with `To fix` or `Justified` status.
  3. *Assign common status to remaining IDs*: Assign a common status and comment to the remaining defects. To batch-edit these results, `Shift`-select them and add the status and comment.

     If you want to create a new status for these IDs, select **Tools > Preferences** and use the **Review Statuses** tab.

  In this way, you can make sure that a generated report shows your rationale for IDs that you did not fix.

Step 4: Generate Reports



If you rerun analysis, the results show only the results that you did not fix, along with your rationale for not fixing. Generate a report that shows how you addressed violations of the standard.

To create a report tailored for a security standard, use one of the following templates during report generation:

- CWE: `SecurityCWE`
- CERT C99: `SecurityCERT`
- ISO/IEC TS 17961: `SecurityISO_17961`

For more information, see “Generate Reports” on page 17-2.

Additional Information

- *How is a security standard report template different from other templates?*

  In the chapter on defects or coding rules, a separate column shows the security standard ID for each result.
- *If I did not choose a security standard before analysis, can I focus on that subset in the report?*

  If you ran analysis for all defects and coding rules, after analysis, narrow your review scope. Instead of `All results` in **Results List**, select `CWE checks`, `CERT checks` or `ISO-17961 checks`. Then, generate a filtered report.

  For information on filtered reports, see “Generate Reports” on page 17-2.
- *How do I ensure from the report that the analysis looked for violations of all supported security standard IDs?*

  The report appendix shows your options used. To make sure that Bug Finder looked for all supported IDs, check the appendix.

  See if the security standard subset or the `all` subset was used for the following options:

  - `Find defects (-checkers)`
  - `Check MISRA C:2012 (-misra3)`

CWE Coding Standard and Polyspace Results

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

CWE and Polyspace Bug Finder

The CWE dictionary assigns a unique identifier to each software weakness type. These identifiers serve 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 results can be mapped to CWE identifiers. Using Bug Finder, you can check and document if your software has weaknesses listed in the CWE dictionary. Bug Finder supports the following aspects of the CWE Compatibility and Effectiveness Program:

- **CWE Searchable**: For each supported CWE identifier, you can see all instances in your code that have weaknesses corresponding to the identifier.
- **CWE Output**: For each Polyspace Bug Finder defect:
  - You can view the associated CWE identifier.
  - You can report the associated CWE identifier.

Bug Finder results are mapped to CWE identifiers (IDs). Using the Bug Finder results, you can evaluate your code against the CWE standard. For instance, CWE ID 119 (Improper restriction of operations within the bounds of a memory buffer) maps to the Bug Finder defects, `Array access out of bounds` and `Pointer access out of bounds`.

For more information on the CWE Compatibility and Effectiveness Program, see CWE Compatibility.

Find CWE IDs from Polyspace Results

Use the following workflow if you want to focus your Bug Finder analysis on the CWE standard.

- *Analysis*: Check your code only for those Bug Finder defects that correspond to the standard.
- *Results*: See only the defects that correspond to the standard. Fix or justify each defect.

  Along with defects, you can see the standard IDs mapped to each defect.
- *Report*: When you generate a report, choose a template tailored for the CWE standard. The report shows the CWE ID-s corresponding to each result.

For the detailed workflow, see "Check C/C++ Code for Security Standards" on page 13-42.

Mapping Between CWE Identifiers and Polyspace Results

The following table lists the CWE IDs (version 2.8) addressed by Polyspace Bug Finder with its corresponding defect checkers. Using Polyspace Bug Finder defect checkers, you can check for 133 CWE IDs.

There are three types of CWE identifiers: Class, Base and Variant. Identifiers of type Class define security weaknesses at an abstract level independent of a specific language or technology, while identifiers of type Base and Variant are more concrete. On the other hand, Polyspace Bug Finder results are designed to be specific so that users can have a precise diagnosis of the defect in their code and understand the defect quickly. Therefore:

- The Bug Finder results are mapped to the specific identifiers of type Base and Variant rather than the generic identifiers of type Class.

  Only when a result covers more ground than a specific CWE identifier is the result mapped to its more general parent type. For instance, the defect checker `Array access out of bounds` covers many kinds of buffer overflows, while CWE-788 refers only to "Access of Memory Location After End of Buffer". Therefore, the defect checker is mapped to its parent, CWE-119, which refers to "Improper Restriction of Operations within the Bounds of a Memory Buffer". However, to keep the mapping precise, an attempt is made to map to specific CWE identifiers.
- Often, more than one Bug Finder result is mapped to a certain CWE identifier.

  For instance, CWE-908 refers to "Use of Uninitialized Resource". To highlights specific kinds of uninitialized resources, Bug Finder has three different checkers: `Member not initialized in constructor`, `Non-initialized pointer`, and `Non-initialized variable`.

For mapping to the subsets CWE-658 and CWE-659, see "Mapping Between CWE-658 or 659 and Polyspace Results" on page 13-79.

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

15	External control of system or configuration setting	`Host change using externally controlled elements`
		
		`Use of externally controlled environment variable`

20	Improper input validation	`Unsafe conversion from string to numerical value`
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`
		
		`Unsafe call to a system function`

78	Improper neutralization of special elements used in an OS command	`Execution of externally controlled command`
		
		`Unsafe call to a system function`

88	Argument injection or modification	`Execution of externally controlled command`
		
		`Unsafe call to a system function`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
114	Process control	Command executed from externally controlled path Execution of a binary from a relative path can be controlled by an external actor Execution of externally controlled command 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 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

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
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
126	Buffer over-read	Buffer overflow from incorrect string format specifier
127	Buffer under-read	Buffer overflow from incorrect string format specifier
128	Wrap-around error	Tainted sign change conversion Unsigned integer conversion overflow Memory allocation with tainted size Unsigned integer overflow Tainted size of variable length array Integer overflow Integer conversion overflow
129	Improper validation of array index	Array access with tainted index Pointer dereference with tainted offset

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
130	Improper handling of length parameter inconsistency	`Mismatch between data length and size`

131	Incorrect calculation of buffer size	`Pointer access out of bounds`
		`Tainted sign change conversion`
		`Unsigned integer conversion overflow`
		`Memory allocation with tainted size`
		`Unsigned integer overflow`
		`Array access out of bounds`
		`Tainted size of variable length array`

134	Uncontrolled format string	`Tainted string format`

170	Improper null termination	`Missing null in string array`
		`Misuse of readlink()`
		`Tainted NULL or non-null-terminated string`

188	Reliance on data/ memory layout	`Invalid assumptions about memory organization`
		`Memory comparison of padding data`
		`Memory comparison of strings`
		`Missing byte reordering when transferring data`
		`Pointer access out of bounds`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

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`

192	Integer coercion error	`Tainted sign change conversion`
		`Unsigned integer conversion overflow`
		`Unsigned integer overflow`
		`Sign change integer conversion overflow`
		`Integer overflow`
		`Integer conversion 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	CWE ID Description	Polyspace Bug Finder Defect Checker

197	Numeric truncation error	`Integer conversion overflow`
		
		`Float conversion overflow`
		
		`Unsigned integer conversion overflow`

198		`Missing byte reordering when transferring data`
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 integer 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`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

250	Execution with unnecessary privileges	`Bad order of dropping privileges`
		
		`Privilege drop not verified`

251	Often misused: string management	`Destination buffer overflow in string manipulation`
252	Unchecked return value	`Returned value of a sensitive function not checked`
273	Improper check for dropped privileges	`Privilege drop not verified`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
310	Cryptographic issues	Constant cipher key Weak cipher mode Context initialized incorrectly for cryptographic operation Nonsecure hash algorithm Constant block cipher initialization vector Context initialized incorrectly for digest operation Missing parameters for key generation Missing data for encryption, decryption or signing operation Nonsecure SSL/TLS protocol Missing peer key Missing cipher key Missing cipher algorithm Missing private key Missing public key Predictable block cipher initialization vector Nonsecure parameters for key generation Predictable cipher key Weak padding for RSA algorithm

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
		Nonsecure RSA public exponent Missing padding for RSA algorithm Missing blinding for RSA algorithm Incorrect key for cryptographic algorithm Missing block cipher initialization vector Weak cipher algorithm Incompatible padding for RSA algorithm operation
311	Missing encryption of sensitive data	Missing cipher final step Missing cipher data to process
320	Key management errors	Constant cipher key Missing peer key Missing cipher key Missing private key Missing public key
321	Use of hard-coded cryptographic key	Constant cipher key

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

325	Missing required cryptographic step	`Missing block cipher initialization vector`
		
		`Missing cipher algorithm`
		
		`Missing cipher data to process`
		
		`Missing cipher final step`
		
		`Missing cipher key`
		
		`Weak cipher algorithm`
		
		`Weak cipher mode`
		
		`Context initialized incorrectly for cryptographic operation`
		
		`Missing parameters for key generation`
		
		`Missing data for encryption, decryption or signing operation`
		
		`Incorrect key for cryptographic algorithm`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

326	Inadequate encryption strength	`Weak cipher algorithm`
		
		`Constant cipher key`
		
		`Weak cipher mode`
		
		`Constant block cipher initialization vector`
		
		`Nonsecure parameters for key generation`
		
		`Predictable cipher key`
		
		`Weak padding for RSA algorithm`
		
		`Nonsecure RSA public exponent`
		
		`Missing padding for RSA algorithm`
		
		`Missing blinding for RSA algorithm`
		
		`Missing block cipher initialization vector`

327	Use of a broken or risky cryptographic algorithm	`Unsafe standard encryption function`
		
		`Weak cipher algorithm`
		
		`Weak cipher mode`
		
		`Nonsecure hash algorithm`
		
		`Nonsecure SSL/TLS protocol`
		
		`Nonsecure parameters for key generation`
		
		`Weak padding for RSA algorithm`
		
		`Nonsecure RSA public exponent`
		
		`Missing padding for RSA algorithm`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
328	Reversible one-way hash	Nonsecure hash algorithm
329	Not using a random IV with CBC mode	Missing block cipher initialization vector Predictable block cipher initialization vector Constant block cipher initialization vector
330	Use of insufficiently random values	Deterministic random output from constant seed Predictable random output from predictable seed Vulnerable pseudo-random number generator Predictable block cipher initialization vector Predictable cipher key
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 Predictable block cipher initialization vector Predictable cipher key

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
362	Concurrent execution using shared resource with improper synchronization ('Race Condition')	File descriptor exposure to child process Opening previously opened resource
364	Signal handler race condition	Shared data access within signal handler Function called from signal handler not asynchronous-safe (strict) Function called from signal handler not asynchronous-safe
366	Race condition within a thread	Data race Data race including atomic operations Data race through standard library function call
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 floating point routine Invalid use of standard library integer routine Tainted division operand Tainted modulo operand

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

372	Incomplete internal state distinction	`Context initialized incorrectly for cryptographic operation`
		
		`Context initialized incorrectly for digest operation`
		
		`Missing parameters for key generation`
		
		`Missing data for encryption, decryption or signing operation`
		
		`Inconsistent cipher operations`
		
		`Missing cipher data to process`
		
		`Incompatible padding for RSA algorithm operation`
		
		`Missing cipher final step`

375	Returning a mutable object to an untrusted caller	`Return of non const handle to encapsulated data member`
377	Insecure temporary file	`Use of non-secure temporary file`

387	Signal errors	`Return from computational exception signal handler`
		
		`Signal call from within signal handler`
		
		`Function called from signal handler not asynchronous-safe`
		
		`Function called from signal handler not asynchronous-safe (strict)`

391	Unchecked error condition	`Errno not checked`
398	Indicator of poor code quality	`Write without a further read`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
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 Mismatched alloc/dealloc functions on Windows
415	Double free	Deallocation of previously deallocated pointer Missing reset of a freed pointer
416	Use after free	Missing reset of a freed pointer Use of previously freed pointer
426	Untrusted search path	Command executed from externally controlled path Library loaded from externally controlled path
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

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

456	Missing initialization of a variable	`Errno not reset`
		
		`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`

465	Pointer Issues	`Unsafe conversion between pointer and integer`

466	Return of pointer value outside of expected range	`Array access out of bounds`
		
		`Pointer access out of bounds`
		
		`Unsafe conversion between pointer and integer`

467	Use of sizeof() on a pointer type	`Possible misuse of sizeof`
		
		`Wrong type used in sizeof`

468	Incorrect pointer scaling	`Incorrect pointer scaling`
469	Use of pointer subtraction to determine size	`Subtraction or comparison between pointers to different arrays`
471	Modification of assumed-immutable data	`Writing to const qualified object`

474	Use of function with inconsistent implementations	`Signal call from within signal handler`
		
		`Use of obsolete standard function`

475	Undefined behavior for input to API	`Copy of overlapping memory`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
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
479	Signal handler use of a non-reentrant function	Function called from signal handler not asynchronous-safe Function called from signal handler not asynchronous-safe (strict)
480	Use of incorrect operator	Invalid use of == (equality) operator Invalid use of = (assignment) operator
481	Assigning instead of comparing	Invalid use of = (assignment) operator
482	Comparing instead of assigning	Invalid use of == (equality) operator
484	Omitted break statement in switch	Missing break of switch case
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

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
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`

562	Return of stack variable address	`Pointer or reference to stack variable leaving scope`
		
		`Use of automatic variable as putenv-family function argument`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

573	Improper following of specification by caller	`Missing cipher algorithm`
		
		`Missing cipher data to process`
		
		`Missing cipher final step`
		
		`Missing cipher key`
		
		`Modification of internal buffer returned from nonreentrant standard function`
		
		`Context initialized incorrectly for cryptographic operation`
		
		`Context initialized incorrectly for digest operation`
		
		`Missing parameters for key generation`
		
		`Missing data for encryption, decryption or signing operation`
		
		`Missing peer key`
		
		`Missing private key`
		
		`Missing public key`
		
		`Missing blinding for RSA algorithm`
		
		`Incorrect key for cryptographic algorithm`
		
		`Incompatible padding for RSA algorithm operation`

587	Assignment of a fixed address to a pointer	`Unsafe conversion between pointer and integer`
		
		`Function pointer assigned with absolute address`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
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`

658	See "Mapping Between CWE-658 or 659 and Polyspace Results" on page 13-79.	
659	See "Mapping Between CWE-658 or 659 and Polyspace Results" on page 13-79.	

663	Use of a non-reentrant function in a concurrent context	`Function called from signal handler not asynchronous-safe`
		
		`Function called from signal handler not asynchronous-safe (strict)`
		
		`Unsafe standard encryption function`
		
		`Unsafe standard function`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

664	Improper control of a resource through its lifetime	`Context initialized incorrectly for cryptographic operation`
		
		`Context initialized incorrectly for digest operation`
		
		`Missing peer key`
		
		`Missing cipher key`
		
		`Missing private key`
		
		`Missing public key`
		
		`Inconsistent cipher operations`
		
		`Missing cipher data to process`
		
		`Incorrect key for cryptographic algorithm`
		
		`Incompatible padding for RSA algorithm operation`
		
		`Missing cipher final step`

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`
		
		`Destruction of locked mutex`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

672	Operation on a resource after expiration or release	`Use of previously closed resource`
		
		`Closing a previously closed resource`

675	Duplicate operations on resource	`Opening previously opened resource`

676	Use of potentially dangerous function	`Unsafe conversion from string to numerical value`
		
		`Use of dangerous standard function`

681	Incorrect conversion between numeric types	`Float conversion overflow`

682	Incorrect calculation	`Absorption of float operand`
		
		`Float overflow`
		
		`Invalid use of standard library floating point routine`
		
		`Invalid use of standard library integer routine`
		
		`Tainted modulo operand`
		
		`Bitwise operation on negative value`
		
		`Use of plain char type for numerical value`

685	Function call with incorrect number of arguments	`Declaration mismatch`
		
		`Format string specifiers and arguments mismatch`
		
		`Standard function call with incorrect arguments`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

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`
		
		`Use of automatic variable as putenv-family function argument`
		
		`Writing to const qualified object`

687	Function call with incorrectly specified argument value	`Copy of overlapping memory`
		
		`Standard function call with incorrect arguments`
		
		`Variable length array with nonpositive size`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
690	Unchecked return value to null pointer dereference	Invalid use of standard library memory routine Use of tainted pointer Null pointer Returned value of a sensitive function not checked Invalid use of standard library string routine Unprotected dynamic memory allocation Tainted NULL or non-null-terminated string Standard function call with incorrect arguments Invalid use of standard library routine
691	Insufficient control flow management	Use of setjmp/longjmp
693	Protection mechanism failure	Nonsecure SSL/TLS protocol
696	Incorrect behavior order	Bad order of dropping privileges
703	Improper check or handling of exceptional conditions	Errno not reset Misuse of errno

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

704	Incorrect type conversion or cast	`Character value absorbed into EOF`
		
		`Qualifier removed in conversion`
		
		`Misuse of sign-extended character value`
		
		`Unreliable cast of pointer`
		
		`Wrong allocated object size for cast`

705	Incorrect control flow scoping	`Abnormal termination of exit handler`
710	Coding standard violation	`Bitwise and arithmetic operation on the same data`
732	Incorrect permission assignment for critical resource	`Vulnerable permission assignments`
754	Improper check for unusual or exceptional conditions	`Returned value of a sensitive function not checked`
755	Improper handling of exceptional conditions	`Exception handler hidden by previous handler`

758	Reliance on undefined, unspecified, or implementation-defined behavior	`Unsafe conversion between pointer and integer`
		
		`Use of plain char type for numerical value`
		
		`Bitwise operation on negative value`

762	Mismatched memory management routines	`Invalid free of pointer`
		
		`Mismatched alloc/dealloc functions on Windows`

764	Multiple locks of a critical resource	`Double lock`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
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`

780	Use of rsa algorithm without oaep	`Weak padding for RSA algorithm`
		
		`Missing padding for RSA algorithm`

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`

789	Uncontrolled memory allocation	`Memory allocation with tainted size`
		
		`Tainted size of variable length array`
		
		`Unprotected dynamic memory allocation`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
805	Buffer access with incorrect length value	`Hard-coded object size used to manipulate memory`

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`
826	Premature release of resource during expected lifetime	`Destruction of locked mutex`

828	Signal handler with functionality that is not asynchronous-safe	`Function called from signal handler not asynchronous-safe`
		
		`Function called from signal handler not asynchronous-safe (strict)`

832	Unlock of a resource that is not locked	`Missing lock`
833	Deadlock	`Deadlock`
843	Access of resource using incompatible type ('Type confusion')	`Unreliable cast of pointer`
872	CERT C++ Secure Coding Section 04 - Integers (INT)	`Invalid use of standard library integer routine`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
873	CERT C++ Secure Coding Section 05 - Floating point arithmetic (FLP)	Absorption of float operand Floating point comparison with equality operators 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
910	Use of expired file descriptor	Use of previously closed resource Closing a previously closed resource Standard function call with incorrect arguments

Mapping Between CWE-658 or 659 and Polyspace Results

CWE-658: Weaknesses in Software Written in C

CWE-658 is a subset of CWE IDs found in C programs that are not common to all languages. See CWE-658.

The following table lists the CWE IDs (version 2.8) from this subset that are addressed by Polyspace Bug Finder, with its corresponding defect checkers.

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

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`
		
		`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`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
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
126	Buffer over-read	Buffer overflow from incorrect string format specifier
127	Buffer under-read	Buffer overflow from incorrect string format specifier
128	Wrap-around error	Tainted sign change conversion Unsigned integer conversion overflow Memory allocation with tainted size Unsigned integer overflow Tainted size of variable length array Integer overflow Integer conversion overflow
129	Improper validation of array index	Array access with tainted index Pointer dereference with tainted offset

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
130	Improper handling of length parameter inconsistency	Mismatch between data length and size
131	Incorrect calculation of buffer size	Pointer access out of bounds Tainted sign change conversion Unsigned integer conversion overflow Memory allocation with tainted size Unsigned integer overflow Array access out of bounds Tainted size of variable length array
134	Uncontrolled format string	Tainted string format
170	Improper null termination	Missing null in string array Misuse of readlink() Tainted NULL or non-null-terminated string
188	Reliance on data/ memory layout	Invalid assumptions about memory organization Memory comparison of padding data Memory comparison of strings Missing byte reordering when transferring data Pointer access out of bounds

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

191	Integer underflow (Wrap or wraparound)	`Integer conversion overflow`
		`Integer overflow`
		`Unsigned integer conversion overflow`
		`Unsigned integer overflow`

192	Integer coercion error	`Tainted sign change conversion`
		`Unsigned integer conversion overflow`
		`Unsigned integer overflow`
		`Sign change integer conversion overflow`
		`Integer overflow`
		`Integer conversion 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`

197	Numeric truncation error	`Integer conversion overflow`
		`Float conversion overflow`
		`Unsigned integer conversion overflow`

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("/")`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
244	Improper clearing of heap memory before release	`Sensitive heap memory not cleared before release`

362	Concurrent execution using shared resource with improper synchronization ('Race Condition')	`File descriptor exposure to child process`
		
		`Opening previously opened resource`

364	Signal handler race condition	`Shared data access within signal handler`
		
		`Function called from signal handler not asynchronous-safe (strict)`
		
		`Function called from signal handler not asynchronous-safe`

366	Race condition within a thread	`Data race`
		
		`Data race including atomic operations`
		
		`Data race through standard library function call`

375	Returning a mutable object to an untrusted caller	`Return of non const handle to encapsulated data member`
401	Improper release of memory before removing last reference	`Memory leak`

415	Double free	`Deallocation of previously deallocated pointer`
		
		`Missing reset of a freed pointer`

416	Use after free	`Missing reset of a freed pointer`
		
		`Use of previously freed pointer`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

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`
		
		`Unsafe conversion between pointer and integer`

467	Use of sizeof() on a pointer type	`Possible misuse of sizeof`
		
		`Wrong type used in sizeof`

468	Incorrect pointer scaling	`Incorrect pointer scaling`
469	Use of pointer subtraction to determine size	`Subtraction or comparison between pointers to different arrays`

474	Use of function with inconsistent implementations	`Signal call from within signal handler`
		
		`Use of obsolete standard function`

476	NULL pointer dereference	`Null pointer`
		
		`Tainted NULL or non-null-terminated string`

478	Missing default case in switch statement	`Missing case for switch condition`

479	Signal handler use of a non-reentrant function	`Function called from signal handler not asynchronous-safe`
		
		`Function called from signal handler not asynchronous-safe (strict)`

480	Use of incorrect operator	`Invalid use of == (equality) operator`
		
		`Invalid use of = (assignment) operator`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
481	Assigning instead of comparing	`Invalid use of = (assignment) operator`
482	Comparing instead of assigning	`Invalid use of == (equality) operator`
484	Omitted break statement in switch	`Missing break of switch case`
558	Use of getlogin() in multithreaded application	`Unsafe standard function`
560	Use of umask() with chmod-style argument	`Umask used with chmod-style arguments`

562	Return of stack variable address	`Pointer or reference to stack variable leaving scope`
		
		`Use of automatic variable as putenv-family function argument`

587	Assignment of a fixed address to a pointer	`Unsafe conversion between pointer and integer`
		
		`Function pointer assigned with absolute address`

676	Use of potentially dangerous function	`Unsafe conversion from string to numerical value`
		
		`Use of dangerous standard function`

685	Function call with incorrect number of arguments	`Declaration mismatch`
		
		`Format string specifiers and arguments mismatch`
		
		`Standard function call with incorrect arguments`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
690	Unchecked return value to null pointer dereference	Invalid use of standard library memory routine Use of tainted pointer Null pointer Returned value of a sensitive function not checked Invalid use of standard library string routine Unprotected dynamic memory allocation Tainted NULL or non-null-terminated string Standard function call with incorrect arguments Invalid use of standard library routine
704	Incorrect type conversion or cast	Character value absorbed into EOF Qualifier removed in conversion Misuse of sign-extended character value Unreliable cast of pointer Wrong allocated object size for cast
762	Mismatched memory management routines	Invalid free of pointer Mismatched alloc/dealloc functions on Windows
783	Operator precedence logic error	Possibly unintended evaluation of expression because of operator precedence rules

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
785	Use of path manipulation function without maximum-sized buffer	`Use of path manipulation function without maximum sized buffer checking`

789	Uncontrolled memory allocation	`Memory allocation with tainted size`
		
		`Tainted size of variable length array`
		
		`Unprotected dynamic memory allocation`

805	Buffer access with incorrect length value	`Hard-coded object size used to manipulate memory`
843	Access of resource using incompatible type ('Type confusion')	`Unreliable cast of pointer`

910	Use of expired file descriptor	`Use of previously closed resource`
		
		`Closing a previously closed resource`
		
		`Standard function call with incorrect arguments`

CWE-659: Weaknesses in Software Written in C++

CWE-659 is a subset of CWE IDs found in C++ programs that are not common to all languages. See CWE-659.

The following table lists the CWE IDs (version 2.8) from this subset that are addressed by Polyspace Bug Finder, with its corresponding defect checkers.

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
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 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
126	Buffer over-read	Buffer overflow from incorrect string format specifier

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
127	Buffer under-read	`Buffer overflow from incorrect string format specifier`

128	Wrap-around error	`Tainted sign change conversion`
		`Unsigned integer conversion overflow`
		`Memory allocation with tainted size`
		`Unsigned integer overflow`
		`Tainted size of variable length array`
		`Integer overflow`
		`Integer conversion overflow`

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`

131	Incorrect calculation of buffer size	`Pointer access out of bounds`
		`Tainted sign change conversion`
		`Unsigned integer conversion overflow`
		`Memory allocation with tainted size`
		`Unsigned integer overflow`
		`Array access out of bounds`
		`Tainted size of variable length array`

134	Uncontrolled format string	`Tainted string format`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

170	Improper null termination	`Missing null in string array`
		
		`Misuse of readlink()`
		
		`Tainted NULL or non-null-terminated string`

188	Reliance on data/ memory layout	`Invalid assumptions about memory organization`
		
		`Memory comparison of padding data`
		
		`Memory comparison of strings`
		
		`Missing byte reordering when transferring data`
		
		`Pointer access out of bounds`

191	Integer underflow (Wrap or wraparound)	`Integer conversion overflow`
		
		`Integer overflow`
		
		`Unsigned integer conversion overflow`
		
		`Unsigned integer overflow`

192	Integer coercion error	`Tainted sign change conversion`
		
		`Unsigned integer conversion overflow`
		
		`Unsigned integer overflow`
		
		`Sign change integer conversion overflow`
		
		`Integer overflow`
		
		`Integer conversion overflow`

194	Unexpected sign extension	`Sign change integer conversion overflow`
		
		`Tainted sign change conversion`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

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`

197	Numeric truncation error	`Integer conversion overflow`
		
		`Float conversion overflow`
		
		`Unsigned integer conversion overflow`

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`

362	Concurrent execution using shared resource with improper synchronization ('Race Condition')	`File descriptor exposure to child process`
		
		`Opening previously opened resource`

364	Signal handler race condition	`Shared data access within signal handler`
		
		`Function called from signal handler not asynchronous-safe (strict)`
		
		`Function called from signal handler not asynchronous-safe`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

366	Race condition within a thread	`Data race`
		
		`Data race including atomic operations`
		
		`Data race through standard library function call`

375	Returning a mutable object to an untrusted caller	`Return of non const handle to encapsulated data member`
401	Improper release of memory before removing last reference	`Memory leak`

415	Double free	`Deallocation of previously deallocated pointer`
		
		`Missing reset of a freed pointer`

416	Use after free	`Missing reset of a freed pointer`
		
		`Use of previously freed pointer`

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`
		
		`Unsafe conversion between pointer and integer`

467	Use of sizeof() on a pointer type	`Possible misuse of sizeof`
		
		`Wrong type used in sizeof`

468	Incorrect pointer scaling	`Incorrect pointer scaling`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
469	Use of pointer subtraction to determine size	`Subtraction or comparison between pointers to different arrays`

476	NULL pointer dereference	`Null pointer`
		
		`Tainted NULL or non-null-terminated string`

478	Missing default case in switch statement	`Missing case for switch condition`

479	Signal handler use of a non-reentrant function	`Function called from signal handler not asynchronous-safe`
		
		`Function called from signal handler not asynchronous-safe (strict)`

480	Use of incorrect operator	`Invalid use of == (equality) operator`
		
		`Invalid use of = (assignment) operator`

481	Assigning instead of comparing	`Invalid use of = (assignment) operator`
482	Comparing instead of assigning	`Invalid use of == (equality) operator`
484	Omitted break statement in switch	`Missing break of switch case`
558	Use of getlogin() in multithreaded application	`Unsafe standard function`

562	Return of stack variable address	`Pointer or reference to stack variable leaving scope`
		
		`Use of automatic variable as putenv-family function argument`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

587	Assignment of a fixed address to a pointer	`Unsafe conversion between pointer and integer`
		
		`Function pointer assigned with absolute address`

676	Use of potentially dangerous function	`Unsafe conversion from string to numerical value`
		
		`Use of dangerous standard function`

690	Unchecked return value to null pointer dereference	`Invalid use of standard library memory routine`
		
		`Use of tainted pointer`
		
		`Null pointer`
		
		`Returned value of a sensitive function not checked`
		
		`Invalid use of standard library string routine`
		
		`Unprotected dynamic memory allocation`
		
		`Tainted NULL or non-null-terminated string`
		
		`Standard function call with incorrect arguments`
		
		`Invalid use of standard library routine`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
704	Incorrect type conversion or cast	Character value absorbed into EOF Qualifier removed in conversion Misuse of sign-extended character value Unreliable cast of pointer Wrong allocated object size for cast
762	Mismatched memory management routines	Invalid free of pointer Mismatched alloc/dealloc functions on Windows
767	Access to critical private variable via public method	Return of non const handle to encapsulated data member
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
789	Uncontrolled memory allocation	Memory allocation with tainted size Tainted size of variable length array Unprotected dynamic memory allocation
805	Buffer access with incorrect length value	Hard-coded object size used to manipulate memory
843	Access of resource using incompatible type ('Type confusion')	Unreliable cast of pointer

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

910	Use of expired file descriptor	`Use of previously closed resource`
		
		`Closing a previously closed resource`
		
		`Standard function call with incorrect arguments`

See Also

More About

- "CWE Coding Standard and Polyspace Results" on page 13-49
- "Check C/C++ Code for Security Standards" on page 13-42

CERT C Coding Standard and Polyspace Results

CERT C is a set of guidelines for software developers and is used for secure coding in C language. It was developed on the CERT community wiki following a community based development process, with the first edition released in 2008 and the second edition released in 2014.

The guidelines help eliminate constructs with undefined behavior that can lead to unexpected results at runtime and expose security weaknesses.

CERT C and Polyspace Bug Finder

Polyspace results can be mapped to CERT C rules and recommendations. Using the results, you can address 103 CERT C rules and 95 CERT C recommendations[5].

In some cases, despite the mapping, you might not see a defect on a noncompliant example from the CERT C documentation. For more information, see “Differences Between CERT C Standards and Defects” on page 13-161.

Find CERT C Guideline Violations from Polyspace Results

Use the following workflow if you want to focus your analysis on the CERT C standard.

- *Analysis*: Check your code only for those Bug Finder defects and coding rules that correspond to the standard.

  If you run a Code Prover analysis, all run-time checks are enabled. Enable the coding rule checkers only.
- *Results*: See only the defects and coding rules that correspond to the standard. Fix or justify each defect or coding rule violation.

  Along with results, you can see the standard IDs mapped to each Polyspace result.
- *Report*: When you generate a report, choose a template tailored for the CERT C standard. The report shows the CERT C rules or recommendations corresponding to each result.

5. The CERT C website, under continuous development, lists 118 rules and 188 recommendations as of 8th January, 2016.

For the detailed workflow, see “Check C/C++ Code for Security Standards” on page 13-42.

Mapping Between CERT C Rules and Polyspace Results

The following tables list the CERT C rules that Polyspace addresses and the corresponding analysis results.

Rule 01. Preprocessor (PRE)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
PRE30-C	Do not create a universal character name through concatenation	`Universal character name from token concatenation`		
PRE31-C	Avoid side effects in arguments to unsafe macros		`MISRA C:2012 Rule 13.2`	
PRE32-C	Do not use preprocessor directives in invocations of function-like macros	`Preprocessor directive in macro argument`		

Rule 02. Declarations and Initialization (DCL)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
DCL30-C	Declare objects with appropriate storage durations	`Pointer or reference to stack variable leaving scope`	`MISRA C:2012 Rule 18.6`	
DCL31-C	Declare identifiers before using them		`MISRA C:2012 Rule 8.1` `MISRA C:2012 Rule 17.3`	

DCL36-C	Do not declare an identifier with conflicting linkage classifications		`MISRA C:2012 Rule 8.2`
			`MISRA C:2012 Rule 8.4`
			`MISRA C:2012 Rule 8.8`
			`MISRA C:2012 Rule 17.3`	

DCL37-C	Do not declare or define a reserved identifier		`MISRA C:2012 Rule 21.1` `MISRA C:2012 Rule 21.2`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
DCL38-C	Use the correct syntax when declaring a flexible array member			`Out of bounds array index` `Illegally dereferenced pointer`
DCL39-C	Avoid information leakage when passing a structure across a trust boundary	`Information leak via structure padding`		
DCL40-C	Do not create incompatible declarations of the same function or object	`Declaration mismatch`	`MISRA C:2012 Rule 5.1` `MISRA C:2012 Rule 8.3`	
DCL41-C	Do not declare variables inside a switch statement before the first case label		`MISRA C:2012 Rule 16.1`	

Rule 03. Expressions (EXP)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
EXP30-C	Do not depend on the order of evaluation for side effects		MISRA C:2012 Rule 13.2	
EXP32-C	Do not access a volatile object through a nonvolatile reference	Qualifier removed in conversion	MISRA C:2012 Rule 11.8	
EXP33-C	Do not read uninitialized memory	Non-initialized pointer Non-initialized variable		Non-initialized local variable Non-initialized pointer Non-initialized variable

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
EXP34-C	Do not dereference null pointers	Arithmetic operation with NULL pointer Invalid use of standard library memory routine Null pointer Use of tainted pointer	MISRA C:2012 Dir 4.14	Illegally dereferenced pointer
EXP35-C	Do not modify objects with temporary lifetime	Accessing object with temporary lifetime		
EXP36-C	Do not cast pointers into more strictly aligned pointer types	Unreliable cast of pointer Wrong allocated object size for cast	MISRA C:2012 Rule 11.1 MISRA C:2012 Rule 11.2 MISRA C:2012 Rule 11.3 MISRA C:2012 Rule 11.5 MISRA C:2012 Rule 11.7	Illegally dereferenced pointer

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
EXP37-C	Call functions with the correct number and type of arguments	Bad file access mode or status Declaration mismatch Format string specifiers and arguments mismatch Qualifier removed in conversion Standard function call with incorrect arguments Unreliable cast of function pointer	MISRA C:2012 Rule 8.3 MISRA C:2012 Rule 11.1 MISRA C:2012 Rule 17.3	
EXP39-C	Do not access a variable through a pointer of an incompatible type	Pointer access out of bounds Unreliable cast of pointer	MISRA C:2012 Rule 11.3	Illegally dereferenced pointer
EXP40-C	Do not modify constant objects	Writing to const qualified object		
EXP42-C	Do not compare padding data	Memory comparison of padding data	MISRA C:2012 Rule 21.16	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
EXP43-C	Avoid undefined behavior when using restrict-qualified pointers	Copy of overlapping memory	MISRA C:2012 Rule 8.14	
EXP44-C	Do not rely on side effects in operands to sizeof, _Alignof, or _Generic	Side effect of expression ignored	MISRA C:2012 Rule 13.6	
EXP45-C	Do not perform assignments in selection statements	Invalid use of assignment operator		
EXP46-C	Do not use a bitwise operator with a Boolean-like operand		MISRA C:2012 Rule 10.1	
EXP47-C	Do not call va_arg with argument of the incorrect type	Incorrect data type passed to va_arg Too many va_arg calls for current argument list		

Rule 04. Integers (INT)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
INT30-C	Ensure that unsigned integer operations do not wrap	Unsigned integer overflow		Overflow
INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data	Integer conversion overflow Call to memset with unintended value Sign change integer conversion overflow Tainted sign change conversion Unsigned integer conversion overflow	MISRA C:2012 Rule 10.1 MISRA C:2012 Rule 10.3 MISRA C:2012 Rule 10.4 MISRA C:2012 Rule 10.6 MISRA C:2012 Rule 10.7	Overflow
INT32-C	Ensure that operations on signed integers do not result in overflow	Integer overflow Tainted division operand Tainted modulo operand		Overflow

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

INT33-C	Ensure that division and remainder operations do not result in divide-by-zero errors	`Integer division by zero`
		`Tainted division operand`
		`Tainted modulo operand`
				`Division by zero`

INT34-C	Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand	`Shift of a negative value` `Shift operation overflow`		`Invalid shift operations`
INT36-C	Converting a pointer to integer or integer to pointer	`Unsafe conversion between pointer and integer`	`MISRA C:2012 Rule 11.6`	

Rule 05. Floating Point (FLP)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
FLP30-C	Do not use floating-point variables as loop counters		MISRA C:2012 Rule 14.1	
FLP32-C	Prevent or detect domain and range errors in math functions	Invalid use of standard library floating point routine		Invalid use of standard library routine
FLP34-C	Ensure that floating-point conversions are within range of the new type	Float conversion overflow Integer conversion overflow Unsigned integer conversion overflow		Overflow
FLP37-C	Do not use object representations to compare floating-point values	Memory comparison of float-point values		

Rule 06. Arrays (ARR)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
ARR30-C	Do not form or use out-of-bounds pointers or array subscripts	Array access out of bounds Array access with tainted index Pointer access out of bounds Pointer dereference with tainted offset Use of tainted pointer	MISRA C:2012 Rule 18.1	Illegally dereferenced pointer Out of bounds array index
ARR32-C	Ensure size arguments for variable length arrays are in a valid range	Memory allocation with tainted size Tainted size of variable length array		
ARR36-C	Do not subtract or compare two pointers that do not refer to the same array	Subtraction or comparison between pointers to different arrays	MISRA C:2012 Rule 18.2	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
ARR37-C	Do not add or subtract an integer to a pointer to a non-array object	`Invalid assumptions about memory organization`		`Illegally dereferenced pointer`

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
ARR38-C	Guarantee that library functions do not form invalid pointers	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 Mismatch between data length and size Pointer access out of bounds Possible misuse of sizeof Use of tainted pointer		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

ARR39-C	Do not add or subtract a scaled integer to a pointer	`Incorrect pointer scaling`
		
		`Invalid use of standard library memory routine`
		
		`Pointer access out of bounds`
		
		`Possible misuse of sizeof`
			`MISRA C:2012 Rule 18.1`	

Rule 07. Characters and Strings (STR)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
STR30-C	Do not attempt to modify string literals	Writing to const qualified object		
STR31-C	Guarantee that storage for strings has sufficient space for character data and null terminator	Array access out of bounds Buffer overflow from incorrect string format specifier Destination buffer overflow in string manipulation Invalid use of standard library string routine Missing null in string array Pointer access out of bounds Tainted NULL or non-null-terminated string Use of dangerous standard function		Invalid use of standard library routine Out of bounds array index

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
STR32-C	Do not pass a non-null-terminated character sequence to a library function that expects a string	Invalid use of standard library string routine Standard function call with incorrect arguments Tainted NULL or non-null-terminated string	MISRA C:2012 Rule 21.17	Invalid use of standard library routine Out of bounds array index
STR34-C	Cast characters to unsigned char before converting to larger integer sizes	Misuse of sign-extended character value		
STR37-C	Arguments to character-handling functions must be representable as an unsigned char	Invalid use of standard library integer routine Misuse of sign-extended character value		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

STR38-C	Do not confuse narrow and wide characters strings and functions	`Unreliable cast of pointer`
		
		`Wrong allocated object size for cast`
		
		`Destination buffer overflow in string manipulation`		

Rule 08. Memory Management (MEM)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
MEM 30-C	Do not access freed memory	Deallocation of previously deallocated pointer Invalid use of standard library string routine Use of previously freed pointer	MISRA C:2012 Dir 4.13 MISRA C:2012 Rule 18.6 MISRA C:2012 Rule 22.1 MISRA C:2012 Rule 22.2	
MEM 31-C	Free dynamically allocated memory when no longer needed	Memory leak		
MEM 33-C	Allocate and copy structures containing a flexible array member dynamically	Misuse of structure with flexible array member		
MEM 34-C	Only free memory allocated dynamically	Invalid free of pointer	MISRA C:2012 Rule 22.2	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

MEM 35-C	Allocate sufficient memory for an object	`Memory allocation with tainted size`
		`Pointer access out of bounds`
		`Wrong type used in sizeof`
				`Illegally dereferenced pointer`

MEM 36-C	Do not modify the alignment of objects by calling realloc()	`Alignment changed after memory reallocation`		

Rule 09. Input Output (FIO)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
FIO30-C	Exclude user input from format strings	`Tainted string format`		
FIO34-C	Distinguish between characters read from a file and EOF or WEOF	`Character value absorbed into EOF`	`MISRA C:2012 Rule 22.7`	
FIO38-C	Do not copy a FILE object	`Misuse of a FILE object`		
FIO39-C	Do not alternately input and output from a stream without an intervening flush or positioning call	`Alternating input and output from a stream without flush or positioning call`		
FIO40-C	Reset strings on fgets() or fgetws() failure	`Use of indeterminate string`		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
FIO41-C	Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects	`Stream argument with possibly unintended side effects`		
FIO42-C	Close files when they are no longer needed	`Resource leak`	`MISRA C:2012 Rule 22.1`	
FIO44-C	Only use values for fsetpos() that are returned from fgetpos()	`Invalid file position`		
FIO45-C	Avoid TOCTOU race conditions while accessing files	`File access between time of check and use (TOCTOU)`		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
FIO46-C	Do not access a closed file	Closing a previously closed resource Standard function call with incorrect arguments Use of previously closed resource		
FIO47-C	Use valid format strings	Format string specifiers and arguments mismatch		

Rule 10. Environment (ENV)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
ENV30-C	Do not modify the object referenced by the return value of certain functions	`Modification of internal buffer returned from nonreentrant standard function`	`MISRA C:2012 Rule 21.19`	
ENV31-C	Do not rely on an environment pointer after following an operation that may invalidate it	`Environment pointer invalidated by previous operation`		
ENV32-C	All exit handlers must return normally	`Abnormal termination of exit handler`		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

ENV33-C	Do not call system()	`Execution of externally controlled command`
		
		`Command executed from externally controlled path`
		
		`Unsafe call to a system function`		

ENV34-C	Do not store pointers returned by certain functions	`Misuse of return value from nonreentrant standard function`		

Rule 11. Signals (SIG)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
SIG30-C	Call only asynchronous-safe functions within signal handlers	`Function called from signal handler not asynchronous-safe` `Function called from signal handler not asynchronous-safe (strict)`		
SIG31-C	Do not access shared objects in signal handlers	`Shared data access within signal handler`		
SIG34-C	Do not call signal() from within interruptible signal handlers	`Signal call from within signal handler`		
SIG35-C	Do not return from a computational exception signal handler	`Return from computational exception signal handler`		

Rule 12. Error Handling (ERR)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

ERR30-C	Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure	`Errno not reset`
		
		`Misuse of errno`
			`MISRA C:2012 Rule 22.8`
			
			`MISRA C:2012 Rule 22.10`	

ERR32-C	Do not rely on indeterminate values of errno	`Misuse of errno in a signal handler`		

ERR33-C	Detect and handle standard library errors	`Errno not checked`
		
		`Returned value of a sensitive function not checked`
		
		`Unprotected dynamic memory allocation`
			`MISRA C:2012 Rule 17.7`
			
			`MISRA C:2012 Rule 22.9`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
ERR34-C	Detect errors when converting a string to a number	`Unsafe conversion from string to numerical value`		

Rule 14. Concurrency (CON)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
CON31-C	Do not destroy a mutex while it is locked	`Destruction of locked mutex`		
CON32-C	Prevent data races when accessing bit-fields from multiple threads	`Data race`		
CON33-C	Avoid race conditions when using library functions	`Data race through standard library function call`		
CON35-C	Avoid deadlock by locking in a predefined order	`Deadlock`		
CON43-C	Do not allow data races in multithreaded code	`Data race`		

Rule 48. Miscellaneous (MSC)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
MSC30-C	Do not use the rand() function for generating pseudorandom numbers	`Vulnerable pseudo-random number generator`		
MSC32-C	Properly seed pseudorandom number generators	`Deterministic random output from constant seed` `Predictable random output from predictable seed`		
MSC33-C	Do not pass invalid data to the asctime() function	`Use of obsolete standard function`		
MSC37-C	Ensure that control never reaches the end of a non-void function	`Missing return statement`		`Function not reachable`

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
MSC38-C	Do not treat a predefined identifier as an object if it might only be implemented as a macro	`Predefined macro used as an object`		
MSC39-C	Do not call va_arg() on a va_list that has an indeterminate value	`Invalid va_list argument` `Non-initialized variable`		
MSC40-C	Do not violate constraints	`Inline constraint not respected`		

Rule 50. POSIX (POS)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
POS30-C	Use the readlink() function properly	`Misuse of readlink()`		
POS33-C	Do not use vfork()	`Use of obsolete standard function`		
POS34-C	Do not call putenv() with a pointer to an automatic variable as the argument	`Use of automatic variable as putenv-family function argument`		
POS35-C	Avoid race conditions while checking for the existence of a symbolic link	`File access between time of check and use (TOCTOU)`		
POS36-C	Observe correct revocation order while relinquishing privileges	`Bad order of dropping privileges`		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
POS37-C	Ensure that privilege relinquishment is successful	`Privilege drop not verified`		
POS38-C	Beware of race conditions when using fork and file descriptors	`File descriptor exposure to child process`		
POS39-C	Use the correct byte ordering when transferring data between systems	`Missing byte reordering when transfering data`		
POS48-C	Do not unlock or destroy another POSIX thread's mutex	`Destruction of locked mutex`		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
POS49-C	When data must be accessed by multiple threads, provide a mutex and guarantee no adjacent data is also accessed	`Data race`		
POS51-C	Avoid deadlock with POSIX threads by locking in predefined order	`Deadlock`		
POS54-C	Detect and handle POSIX library errors	`Returned value of a sensitive function not checked`		

Rule 51. Microsoft Windows (WIN)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
WIN30-C	Properly pair allocation and deallocation functions	`Mismatched alloc/dealloc functions on Windows`		

Mapping Between CERT C Recommendations and Polyspace Results

The following tables list the CERT C recommendations that Polyspace Bug Finder addresses and the corresponding defects.

Rec. 01. Preprocessor (PRE)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
PRE00-C	Prefer inline or static functions to function-like macros		MISRA C:2012 Dir 4.9	
PRE01-C	Use parenthesis within macros around parameter names		MISRA C:2012 Rule 20.7	
PRE06-C	Enclose header files in an inclusion guard		MISRA C:2012 Dir 4.10	
PRE07-C	Avoid using repeated question marks		MISRA C:2012 Rule 4.2	
PRE09-C	Do not replace secure functions with deprecated or obsolescent functions	Use of dangerous standard function Use of obsolete standard function		

Rec. 02. Declarations and Initialization (DCL)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
DCL01-C	Do not reuse variable names in subscopes	Variable shadowing	MISRA C:2012 Rule 5.3	
DCL02-C	Use visually distinct identifiers		MISRA C:2012 Dir 4.5	
DCL06-C	Use meaningful symbolic constants to represent literal values	Hard coded buffer size Hard coded loop boundary		
DCL07-C	Include the appropriate type information in function declarators		MISRA C:2012 Rule 8.2 MISRA C:2012 Rule 11.1	
DCL10-C	Maintain the contract between the writer and caller of variadic functions	Format string specifiers and arguments mismatch	MISRA C:2012 Rule 17.1	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
DCL12-C	Implement abstract data types using opaque types			
DCL11-C	Understand the type issues associated with variadic functions	`Format string specifiers and arguments mismatch`	`MISRA C:2012 Rule 17.1`	
DCL13-C	Declare function parameters that are pointers to values not changed by the function as const		`MISRA C:2012 Rule 8.13`	
DCL15-C	Declare file-scope objects or functions that do not need external linkage as static		`MISRA C:2012 Rule 8.7` `MISRA C:2012 Rule 8.8`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
DCL16-C	Use "L", not "l" to indicate a long value		MISRA C:2012 Rule 7.3	
DCL18-C	Do not begin integer constants with 0 when specifying a decimal value		MISRA C:2012 Rule 7.1	
DCL19-C	Minimize the scope of variables and functions		MISRA C:2012 Rule 8.7 MISRA C:2012 Rule 8.9	
DCL20-C	Explicitly specify void when a function accepts no arguments		MISRA C:2012 Rule 8.2	
DCL22-C	Use volatile for data that cannot be cached	Write without a further read	MISRA C:2012 Rule 2.2	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
DCL23-C	Guarantee that mutually visible identifiers are unique		MISRA C:2012 Rule 5.1 MISRA C:2012 Rule 5.2 MISRA C:2012 Rule 5.3 MISRA C:2012 Rule 5.4 MISRA C:2012 Rule 5.5	

Rec. 03. Expressions (EXP)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
EXP00-C	Use parentheses for precedence of operation	Possibly unintended evaluation of expression because of operator precedence rules	MISRA C:2012 Rule 12.1	
EXP05-C	Do not cast away a const qualification	Qualifier removed in conversion	MISRA C:2012 Rule 11.8	
EXP08-C	Ensure pointer arithmetic is used correctly	Incorrect pointer scaling Pointer access out of bounds Invalid use of standard library memory routine	MISRA C:2012 Rule 18.1 MISRA C:2012 Rule 18.2 MISRA C:2012 Rule 18.3	Illegally dereferenced pointer
EXP09-C	Use sizeof to determine the size of a type or variable	Hard-coded object size used to manipulate memory		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
EXP10-C	Do not depend on the order of evaluation of subexpressions or the order in which side effects take place		MISRA C:2012 Rule 13.2	
EXP12-C	Do not ignore values returned by functions	Returned value of a sensitive function not checked		
EXP13-C	Treat relational and equality operators as if they were nonassociative	Possibly unintended evaluation of expression because of operator precedence rules		
EXP19-C	Use braces for the body of an if, for, or while statement		MISRA C:2012 Rule 15.6	

Rec. 04. Integers (INT)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
INT00-C	Understand the data model used by your implementation(s)	Format string specifiers and arguments mismatch Integer overflow	MISRA C:2012 Dir 4.6	
INT02-C	Understand integer conversion rules	Integer conversion overflow Integer overflow Tainted sign change conversion Unsigned integer conversion overflow	MISRA C:2012 Rule 10.1 MISRA C:2012 Rule 10.3 MISRA C:2012 Rule 10.4 MISRA C:2012 Rule 10.6 MISRA C:2012 Rule 10.7 MISRA C:2012 Rule 10.8	Overflow
INT04-C	Enforce limits on integer values originating from tainted sources	Array access with tainted index Loop bounded with tainted value Memory allocation with tainted size Tainted size of variable length array	MISRA C:2012 Dir 4.14	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

INT07-C	Use only explicitly signed or unsigned char type for numeric values	`Use of plain char type for numerical value`	`MISRA C:2012 Rule 10.1`
			
			`MISRA C:2012 Rule 10.3`
			
			`MISRA C:2012 Rule 10.4`	

INT08-C	Verify that all integer values are in range	`Integer overflow`		`Overflow`
INT09-C	Ensure enumeration constants map to unique values		`MISRA C:2012 Rule 8.12`	
INT10-C	Do not assume a positive remainder when using the % operator	`Tainted modulo operand`	`MISRA C:2012 Dir 4.14`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
INT12-C	Do not make assumptions about the type of a plain int bit-field when used in an expression	`Integer conversion overflow`	`MISRA C:2012 Rule 6.1`	
INT13-C	Use bitwise operators only on unsigned operands	`Bitwise operation on negative value`	`MISRA C:2012 Rule 10.1`	
INT14-C	Avoid performing arithmetic and bitwise operations on the same data	`Bitwise and arithmetic operation on the same data`		
INT16-C	Do not make assumptions about representation of signed integers		`MISRA C:2012 Rule 10.1`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

INT18-C	Evaluate integer expressions in a larger size before comparing or assigning to that size	`Integer conversion overflow`
		
		`Integer overflow`
		
		`Unsigned integer conversion overflow`
		
		`Unsigned integer overflow`
			`MISRA C:2012 Rule 10.4`
			
			`MISRA C:2012 Rule 10.6`
			
			`MISRA C:2012 Rule 10.7`	

Rec. 05. Floating Point (FLP)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
FLP00-C	Understand the limitations of floating-point numbers	Absorption of float operand		
FLP02-C	Avoid using floating-point numbers when precise computation is needed	Floating point comparison with equality operators		
FLP03-C	Detect and handle floating-point errors	Float conversion overflow Float overflow Invalid use of standard library floating point routine Float division by zero		Overflow Division by zero
FLP06-C	Convert integers to floating point for floating-point operations	Float overflow	MISRA C:2012 Rule 10.3	

Rec. 06. Arrays (ARR)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

ARR00-C	Understand how arrays work	`Array access out of bounds`
		
		`Improper array initialization`
		
		`Possible misuse of sizeof`
		
		`Wrong type used in sizeof`		

ARR01-C	Do not apply the sizeof operator to a pointer when taking the size of an array	`Possible misuse of sizeof`
		
		`Wrong type used in sizeof`
			`MISRA C:2012 Rule 12.5`	

ARR02-C	Explicitly specify array bounds, even if implicitly defined by an initializer	`Improper array initialization`	`MISRA C:2012 Rule 8.11`
			
			`MISRA C:2012 Rule 9.5`	

Rec. 07. Characters and Strings (STR)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
STR02-C	Sanitize data passed to complex subsystems	Execution of externally controlled command Command executed from externally controlled path Library loaded from externally controlled path	MISRA C:2012 Dir 4.14	
STR03-C	Do not inadvertently truncate a string	Buffer overflow from incorrect string format specifier		Invalid use of standard library routine
STR04-C	Use plain char for characters in the basic character set		MISRA C:2012 Rule 10.1 MISRA C:2012 Rule 10.2 MISRA C:2012 Rule 10.3 MISRA C:2012 Rule 10.4	
STR05-C	Use pointers to const when referring to string literals	Writing to const qualified object		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

STR06-C	Do not assume strtok() leaves the parse string unchanged	`Modification of internal buffer returned from nonreentrant standard function`
		
		`Writing to const qualified object`		

STR07-C	Use the bounds-checking interface for string manipulation	`Use of dangerous standard function`
		
		`Destination buffer overflow in string manipulation`
				`Invalid use of standard library routine`

STR08-C	Use managed strings for development of new string manipulation code	`Use of dangerous standard function`
		
		`Destination buffer overflow in string manipulation`		

STR11-C	Do not specify the bound of a character array initialized with a string literal	`Missing null in string array`		

Rec. 08. Memory Management (MEM)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

MEM 00-C	Allocate and free memory in the same module, at the same level of abstraction	`Invalid free of pointer`
		
		`Deallocation of previously deallocated pointer`
		
		`Use of previously freed pointer`		

MEM 01-C	Store a new value in pointers immediately after free()	`Missing reset of a freed pointer`		

MEM 02-C	Immediately cast the result of a memory allocation function call into a pointer to the allocated type	`Wrong allocated object size for cast`
		
		`Wrong type used in sizeof`		

MEM 03-C	Clear sensitive information stored in reusable resources	`Sensitive heap memory not cleared before release`
		
		`Uncleared sensitive data in stack`		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

MEM 04-C	Beware of zero-length allocations	`Tainted sign change conversion`
		
		`Tainted size of variable length array`
		
		`Variable length array with nonpositive size`		

MEM 05-C	Avoid large stack allocations	`Tainted size of variable length array`
		
		`Variable length array with nonpositive size`
			`MISRA C:2012 Rule 17.2`	

MEM 06-C	Ensure that sensitive data is not written out to disk	`Sensitive data printed out`		
MEM 07-C	Ensure that arguments to `calloc()`, when multiplied, do not wrap	`Memory allocation with tainted size`		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
MEM 10-C	Define and use a pointer validation function	Memory allocation with tainted size Unprotected dynamic memory allocation Use of tainted pointer		Illegally dereferenced pointer
MEM 11-C	Do not assume infinite heap space	Memory leak Memory allocation with tainted size Tainted sign change conversion Unprotected dynamic memory allocation		
MEM 12-C	Consider a goto chain when leaving a function on error when using and releasing resources	Memory leak Missing unlock Resource leak		

Rec. 09. Input Output (FIO)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
FIO01-C	Be careful using functions that use file names for identification	`File access between time of check and use (TOCTOU)`		
FIO02-C	Canonicalize path names originating from tainted sources	`Vulnerable path manipulation`		
FIO03-C	Do not make assumptions about fopen() and file creation	`Use of non-secure temporary file`		

FIO06-C	Create files with appropriate access permissions	`Umask used with chmod-style arguments`
		
		`Vulnerable permission assignments`		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
FIO11-C	Take care when specifying the mode parameter of fopen()	`Bad file access mode or status`		
FIO17-C	Do not rely on an ending null character when using fread()	`Tainted NULL or non-null-terminated string`		
FIO21-C	Do not create temporary files in shared directories	`Use of non-secure temporary file`		
FIO24-C	Do not open a file that is already open	`Opening previously opened resource`		

Rec. 10. Environment (ENV)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
ENV01-C	Do not make assumptions about the size of an environment variable	Destination buffer overflow in string manipulation Tainted NULL or non-null-terminated string Use of dangerous standard function		

Rec. 12. Error Handling (ERR)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
ERR00-C	Adopt and implement a consistent and comprehensive error-handling policy		MISRA C:2012 Rule 17.1	

Rec. 13. Application Programming Interfaces (API)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
API00-C	Functions should validate their parameters	Array access with tainted index Command executed from externally controlled path Execution of externally controlled command Host change using externally controlled elements Invalid use of standard library memory routine Invalid use of standard library routine Invalid use of standard library string routine Library loaded from externally controlled path Loop bounded with tainted value Memory allocation with tainted size		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
		Pointer dereference with tainted offset Standard function call with incorrect arguments Tainted division operand Tainted modulo operand Tainted NULL or non-null-terminated string Tainted sign change conversion Tainted size of variable length array Tainted string format Use of externally controlled environment variable Use of tainted pointer		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

API02-C	Functions that read or write to or from an array should take an argument to specify the source or target size	`Array access out of bounds`
		`Array access with tainted index`
		`Pointer access out of bounds`
		`Pointer dereference with tainted offset`
		`Use of dangerous standard function`
		`Use of tainted pointer`		

API03-C	Create consistent interfaces and capabilities across related functions		`MISRA C:2012 Rule 21.3`	

Rec. 14. Concurrency (CON)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

CON01-C	Acquire and release synchronization primitives in the same module, at the same level of abstraction	`Double lock`
		`Double unlock`
		`Missing lock`
		`Missing unlock`		

CON09-C	Avoid the ABA problem when using lock-free algorithms	`Data race`		

Rec. 48. Miscellaneous (MSC)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
MSC01-C	Strive for logical completeness	Dead code Missing case for switch condition Unreachable code		
MSC04-C	Use comments consistently and in a readable fashion		MISRA C:2012 Rule 1.2 MISRA C:2012 Rule 3.1	
MSC07-C	Detect and remove dead code	Dead code Missing case for switch condition Unreachable code	MISRA C:2012 Rule 2.1	Unreachable code
MSC12-C	Detect and remove code that has no effect or is never executed	Dead code Unreachable code Use of memset with size argument zero	MISRA C:2012 Rule 2.1 MISRA C:2012 Rule 2.2	Function not reachable Unreachable code
MSC13-C	Detect and remove unused values	Unused parameter Write without a further read		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
MSC15-C	Do not depend on undefined behavior	Array access out of bounds Copy of overlapping memory Declaration mismatch Format string specifiers and arguments mismatch Integer overflow Invalid use of standard library memory routine Invalid use of standard library routine Invalid use of standard library string routine Non-initialized pointer Non-initialized variable Null pointer Overlapping assignment		Illegally dereferenced pointer Non-initialized variable Out of bounds array index Overflow

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

		`Pointer access out of bounds`
		
		`Standard function call with incorrect arguments`
		
		`Unreliable cast of function pointer`
		
		`Unreliable cast of pointer`
		
		`Use of tainted pointer`
		
		`Writing to const qualified object`		

MSC17-C	Finish every set of statements associated with a case label with a break statement	`Missing break of switch case`		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

MSC18-C	Be careful while handling sensitive data, such as passwords, in program code	`Constant block cipher initialization vector`
		`Constant cipher key`
		`Predictable block cipher initialization vector`
		`Predictable cipher key`
		`Sensitive heap memory not cleared before release`
		`Uncleared sensitive data in stack`
		`Unsafe standard encryption function`		

MSC20-C			`MISRA C:2012 Rule 16.2`	
MSC21-C	Use robust loop termination conditions	`Loop bounded with tainted value` `Tainted sign change conversion`		`Non-terminating loop`
MSC22-C	Use the setjmp(), longjmp() facility securely	`Use of setjmp/longjmp`		

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
MSC24-C	Do not use deprecated or obsolescent functions	`Use of obsolete standard function`		

Rec. 50. POSIX (POS)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check
POS05-C	Limit access to files by creating a jail	`File manipulation after chroot() without chdir("/")`		

Rec. 51. Microsoft Windows (WIN)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule	Polyspace Code Prover Check

WIN00-C	Be specific when dynamically loading libraries	`Library loaded from externally controlled path`
		
		`Load of library from a relative path can be controlled by an external actor`		

Differences Between CERT C Standards and Defects

Despite the mapping, if you do not see a Bug Finder defect in a noncompliant example on the CERT C document, it might be because:

- The Bug Finder defect covers only a certain aspect of the CERT C rule or recommendation. Your code can violate a rule or recommendation in multiple ways, but a specific defect covers a specific violation pattern.

  From the name of the Bug Finder defect and the description, you can understand which aspect of the rule or recommendation is covered by the defect.
- In certain cases, Bug Finder issues a defect only if a run-time error can occur due to not following a rule or recommendation. Not following the rule or recommendation alone does not trigger the defect.

  For instance, in the following noncompliant code example from the CERT-C documentation on `ARR30-C` (Do not form or use out-of-bounds pointers or array subscripts), the array index is not checked for negative values.

  ```
  enum { TABLESIZE = 100 };
  static int table[TABLESIZE];
  int *f(int index) {
      if (index < TABLESIZE) {
          return table + index;
      }
      return NULL;
  }
  ```

  If you analyze this example, Polyspace Bug Finder does not show the defect **Array access out of bounds** or **Pointer access out of bounds** because the array index is not used to access the array. The return value `table + index` is not used anywhere in the code.

  You can see the defect if you use the return value `table + index`. For instance, if the code contains the following call to the function `f` with a negative value, the defect appears when the return value of `f` is dereferenced.

  ```
  int main () {
      int *p = f(-2);
      return *p;
  }
  ```

ISO/IEC TS 17961 Coding Standard and Polyspace Results

ISO/IEC TS 17961 is a set of rules for developing secure code. The rules are designed such that they can enforced by static analysis tools without excessive false positives.

Polyspace® Bug Finder™ results can be mapped to ISO/IEC TS 17961 rules.

Find ISO/IEC TS 17961 Rule Violations from Polyspace Results

Use the following workflow if you want to focus your Bug Finder analysis on the ISO/IEC TS 17961 standard.

- *Analysis*: Check your code only for those Bug Finder defects and coding rules that correspond to the standard.
- *Results*: See only the defects and coding rules that correspond to the standard. Fix or justify each defect or coding rule violation.

  Along with results, you can see the standard IDs mapped to each Polyspace result.
- *Report*: When you generate a report, choose a template tailored for the ISO/IEC TS 17961 standard. The report shows the ISO/IEC TS 17961 rules corresponding to each result.

For the detailed workflow, see “Check C/C++ Code for Security Standards” on page 13-42.

Mapping Between ISO/IEC TS 17961 Rules and Polyspace Results

The following tables list the ISO/IEC TS 17961 rules that Polyspace Bug Finder addresses and the corresponding defects or MISRA C: 2012 rule.

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
ptrcomp	Accessing an object through a pointer to an incompatible type	Pointer access out of bounds Unreliable cast of pointer	MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 10.8 MISRA C:2012 Rule 11.2 MISRA C:2012 Rule 11.3
accfree	Accessing freed memory	Deallocation of previously deallocated pointer Invalid use of standard library string routine Use of previously freed pointer	MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 21.3
accsig	Accessing shared objects in signal handlers	Shared data access within signal handler	MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 21.5
boolasgn	No assignment in conditional expressions	Invalid use of = (assignment) operator	MISRA C:2012 Rule 13.4

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
asyncsig	Calling functions in the C Standard Library other than abort, _Exit, and signal from within a signal handler	Function called from signal handler not asynchronous-safe Function called from signal handler not asynchronous-safe (strict)	MISRA C:2012 Rule 21.5
argcomp	Calling functions with incorrect arguments	Declaration mismatch Format string specifiers and arguments mismatch Qualifier removed in conversion Standard function call with incorrect arguments Unreliable cast of function pointer	MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 8.2 MISRA C:2012 Rule 8.3 MISRA C:2012 Rule 17.3
sigcall	Calling signal from interruptible signal handlers	Signal call from within signal handler	MISRA C:2012 Rule 21.5

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule

`syscall`	Calling `system`	`Command executed from externally controlled path`
		
		`Execution of externally controlled command`
		
		`Unsafe call to a system function`
			`MISRA C:2012 Rule 21.8`

`padcomp`	Comparison of padding data	`Memory comparison of padding data`	

`intptrconv`	Converting a pointer to integer or integer to pointer	`Unsafe conversion between pointer and integer`	`MISRA C:2012 Rule 1.3`
			
			`MISRA C:2012 Rule 11.4`

`alignconv`	Converting pointer values to more strictly aligned pointer types	`Unreliable cast of pointer`
		
		`Wrong allocated object size for cast`
			`MISRA C:2012 Rule 11.3`

`filecpy`	Copying a `FILE` object	`Misuse of a FILE object`	`MISRA C:2012 Rule 22.5`

`funcdecl`	Declaring the same function or object in incompatible ways	`Declaration mismatch`	`MISRA C:2012 Rule 1.3`
			
			`MISRA C:2012 Rule 5.1`
			
			`MISRA C:2012 Rule 8.3`
			
			`MISRA C:2012 Rule 8.4`

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
nullref	Dereferencing an out-of-domain pointer	Arithmetic operation with NULL pointer Invalid use of standard library memory routine Invalid use of standard library string routine Null pointer Use of tainted pointer	MISRA C:2012 Dir 4.1 MISRA C:2012 Rule 18.1
addrescape	Escaping of the address of an automatic object	Pointer or reference to stack variable leaving scope Use of automatic variable as putenv-family function argument	MISRA C:2012 Rule 18.6
signconv	Conversion of signed characters to wider integer types before a check for EOF	Misuse of sign-extended character value	
swtchdflt	Use of an implied default in a switch statement	Dead code Missing case for switch condition Unreachable code	MISRA C:2012 Rule 16.4

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule

`fileclose`	Failing to close files or free dynamic memory when they are no longer needed	`Memory leak`
		
		`Resource leak`
			`MISRA C:2012 Rule 22.1`

`liberr`	Failing to detect and handle standard library errors	`Returned value of a sensitive function not checked`
		
		`Standard function call with incorrect arguments`
		
		`Unprotected dynamic memory allocation`
			`MISRA C:2012 Dir 4.7`
			
			`MISRA C:2012 Rule 17.7`

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
libptr	Forming invalid pointers by library function	Destination buffer overflow in string manipulation Incorrect pointer scaling Invalid use of standard library memory routine Invalid use of standard library string routine Possible misuse of sizeof Use of path manipulation function without maximum-sized buffer checking	MISRA C:2012 Dir 4.1 MISRA C:2012 Dir 4.11 MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 21.18
insufmem	Allocating insufficient memory	Pointer access out of bounds Possible misuse of sizeof Wrong allocated object size for cast Wrong type used in sizeof	MISRA C:2012 Rule 21.3

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
invptr	Forming or using out-of-bounds pointers or array subscripts	Array access out of bounds Array access with tainted index Pointer access out of bounds Pointer dereference with tainted offset Use of tainted pointer	MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 18.1
dblfree	Freeing memory multiple times	Deallocation of previously deallocated pointer Use of previously freed pointer	MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 21.3 MISRA C:2012 Rule 22.2
usrfmt	Including tainted or out-of-domain input in a format string	Tainted string format	MISRA C:2012 Dir 4.1 MISRA C:2012 Dir 4.11 MISRA C:2012 Dir 4.14 MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 21.6

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
inverrno	Incorrectly setting and using errno	Errno not checked Errno not reset Misuse of errno	MISRA C:2012 Dir 4.1 MISRA C:2012 Dir 4.7 MISRA C:2012 Rule 22.8 MISRA C:2012 Rule 22.9 MISRA C:2012 Rule 22.10
diverr	Integer division errors	Integer division by zero Tainted division operand Tainted modulo operand	MISRA C:2012 Dir 4.1 MISRA C:2012 Dir 4.14 MISRA C:2012 Rule 1.3
ioileave	Interleaving stream inputs and outputs without a flush or positioning call	Alternating input and output from a stream without flush or positioning call	MISRA C:2012 Dir 4.1 MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 21.6
strmod	Modifying string literals	Writing to const qualified object	MISRA C:2012 Rule 7.4

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule

`libmod`	Modifying the string returned by `getenv`, `localeconv`, `setlocale`, and `strerror`	`Modification of internal buffer returned from non-reentrant standard function`	`MISRA C:2012 Rule 1.3`
			`MISRA C:2012 Rule 21.8`
			`MISRA C:2012 Rule 21.19`

`intoflow`	Overflowing signed integers	`Integer overflow`
		`Tainted modulo operand`
			`MISRA C:2012 Dir 4.1`
			`MISRA C:2012 Rule 1.3`
			`MISRA C:2012 Rule 10.3`
			`MISRA C:2012 Rule 10.4`

`nonnullcs`	Passing a non-null-terminated character sequence to a library function that expects a string	`Invalid use of standard library string routine`
		`Missing null in string array`
		`Standard function call with incorrect arguments`
		`Tainted NULL or non-null-terminated string`
			`MISRA C:2012 Dir 4.1`
			`MISRA C:2012 Dir 4.11`
			`MISRA C:2012 Rule 1.3`

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
chrsgnext	Passing arguments to character-handling functions that are not representable as unsigned char	Invalid use of standard library integer routine	MISRA C:2012 Dir 4.1 MISRA C:2012 Dir 4.11 MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 21.13
restrict	Passing pointers into the same object as arguments to different restrict-qualified parameters	Copy of overlapping memory	MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 8.14
xfree	Reallocating or freeing memory that was not dynamically allocated	Invalid free of pointer	MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 22.2

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
uninitref	Referencing uninitialized memory	Member not initialized in constructor Non-initialized pointer Non-initialized variable Pointer to non initialized value converted to const pointer	MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 9.1
ptrobj	Subtracting or comparing two pointers that do not refer to the same array	Subtraction or comparison between pointers to different arrays	MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 18.2 MISRA C:2012 Rule 18.3
taintstrcpy	Tainted strings are passed to a string copying function	Tainted NULL or non-null-terminated string	MISRA C:2012 Dir 4.1 MISRA C:2012 Dir 4.11 MISRA C:2012 Dir 4.14
sizeofptr	Taking the size of a pointer to determine the size of the pointed-to type	Possible misuse of sizeof	

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
taintnoproto	Using a tainted value as an argument to an unprototyped function pointer	Call through non-prototyped function pointer	MISRA C:2012 Dir 4.14 MISRA C:2012 Rule 8.2
taintformatio	Using a tainted value to write to an object using a formatted input or output function	Buffer overflow from incorrect string format specifier Destination buffer overflow in string manipulation Invalid use of standard library string routine Missing null in string array Pointer access out of bounds Tainted NULL or non-null-terminated string Use of dangerous standard function	MISRA C:2012 Dir 4.1 MISRA C:2012 Dir 4.11 MISRA C:2012 Dir 4.14 MISRA C:2012 Rule 21.6

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
xfilepos	Using a value for fsetpos other than a value returned from fgetpos	Invalid file position	MISRA C:2012 Dir 4.1 MISRA C:2012 Dir 4.11 MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 21.6
libuse	Using an object overwritten by getenv, localeconv, setlocale, and strerror	Misuse of return value from nonreentrant standard function	MISRA C:2012 Rule 21.8
chreof	Using character values that are indistinguishable from EOF	Character value absorbed into EOF	MISRA C:2012 Rule 22.7
resident	Using identifiers that are reserved for the implementation		MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 20.4 MISRA C:2012 Rule 21.1 MISRA C:2012 Rule 21.2

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
invfmtstr	Using invalid format strings	Format string specifiers and arguments mismatch	MISRA C:2012 Dir 4.1 MISRA C:2012 Dir 4.11 MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 21.6
taintsink	Tainted, potentially mutilated, or out-of-domain integer values are used in a restricted sink	Loop bounded with tainted value Memory allocation with tainted size Tainted sign change conversion Tainted size of variable length array	MISRA C:2012 Dir 4.1 MISRA C:2012 Dir 4.11 MISRA C:2012 Dir 4.14 MISRA C:2012 Rule 1.3

14

Interpret Polyspace Bug Finder Results

- "Interpret Polyspace Bug Finder Results" on page 14-2
- "Investigate the Cause of Empty Results List" on page 14-9
- "Dashboard" on page 14-11
- "Concurrency Modeling" on page 14-17
- "Results List" on page 14-19
- "Source" on page 14-22
- "Result Details" on page 14-28

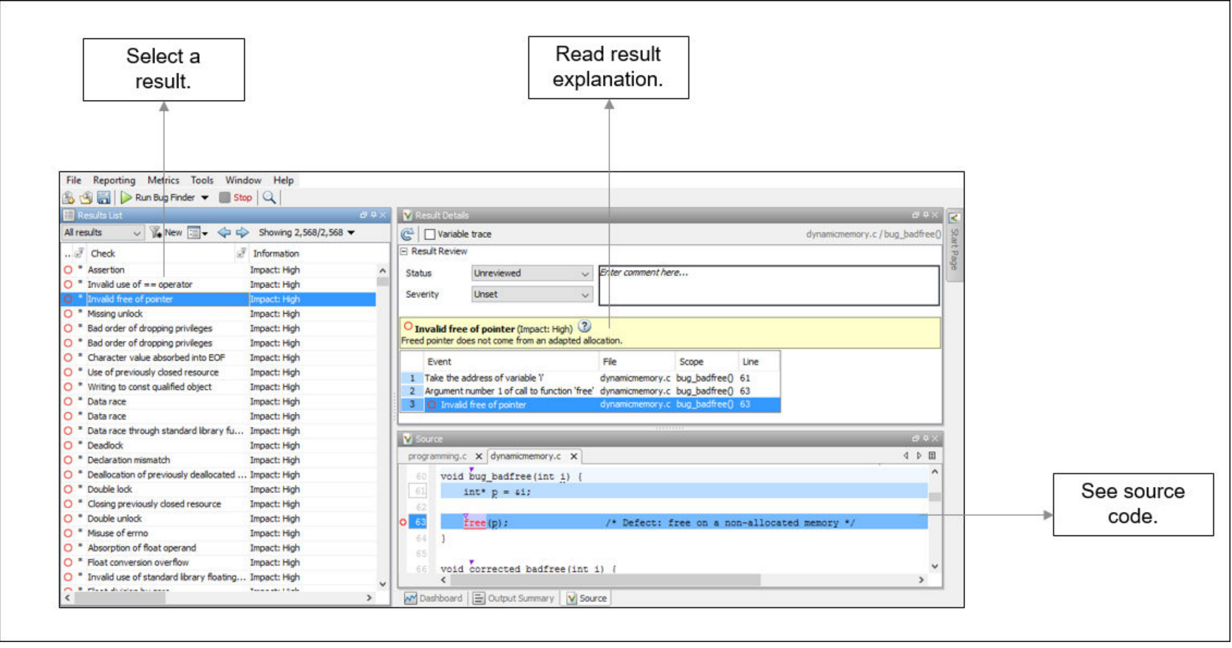
Interpret Polyspace Bug Finder Results

When you open the results of a Polyspace Bug Finder analysis, you see a list on the **Results List** pane. The results consist of defects, coding rule violations or code metrics.

You can first narrow down the focus of your review:

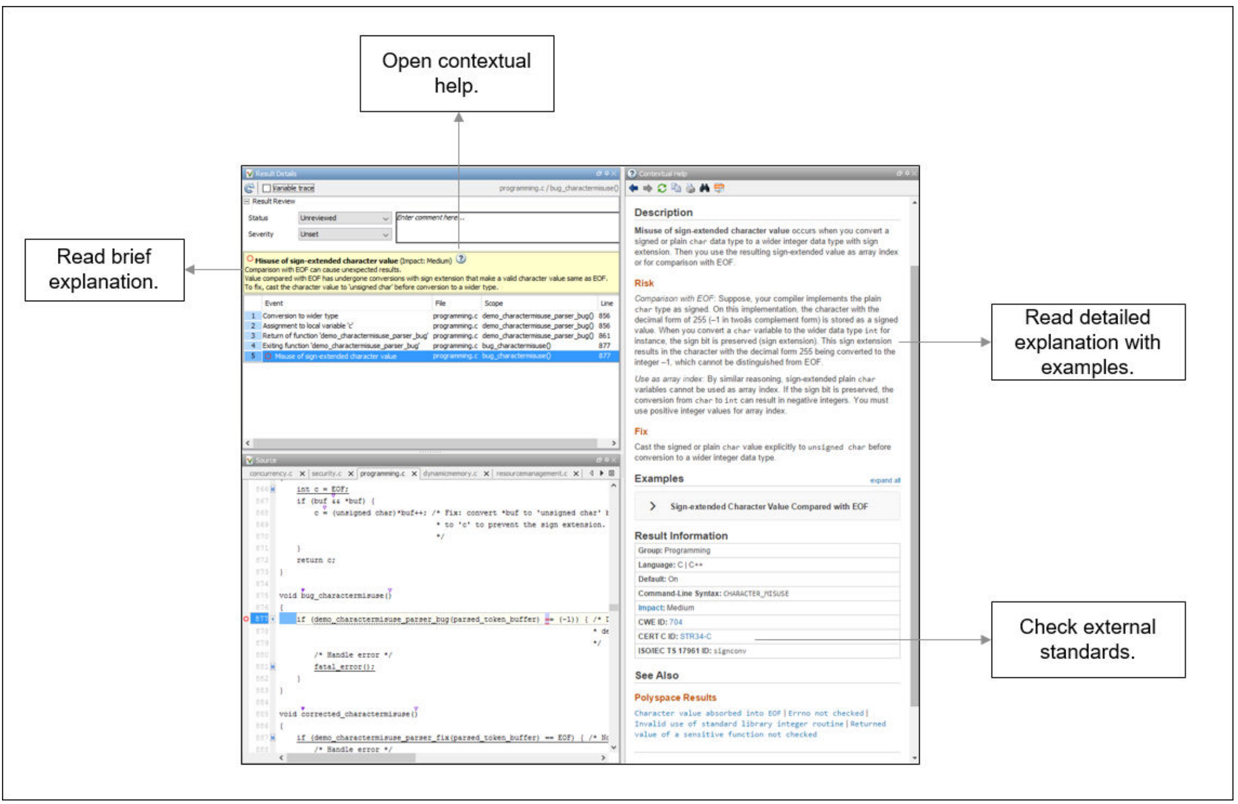
- Use filters on the results list columns to narrow down the list. For instance, you can focus on the high-impact defects.
- Organize results by file or result family. Use the icon above the list.

Once you narrow down the list, you can begin reviewing individual results. This topic describes how to review a result.



To begin your review, select a result in the list.

Interpret Result Details Message



Interpret Message

The first step is to understand what is wrong. Read the message on the **Result Details** pane and the related line of code on the **Source** pane.

Seek Additional Resources for Help

Sometimes, you need additional help for certain results. Click the icon to open a help page for the selected result. See code examples illustrating the result. Check external standards such as CWE or CERT-C that provide additional rationale for fixing the issue.

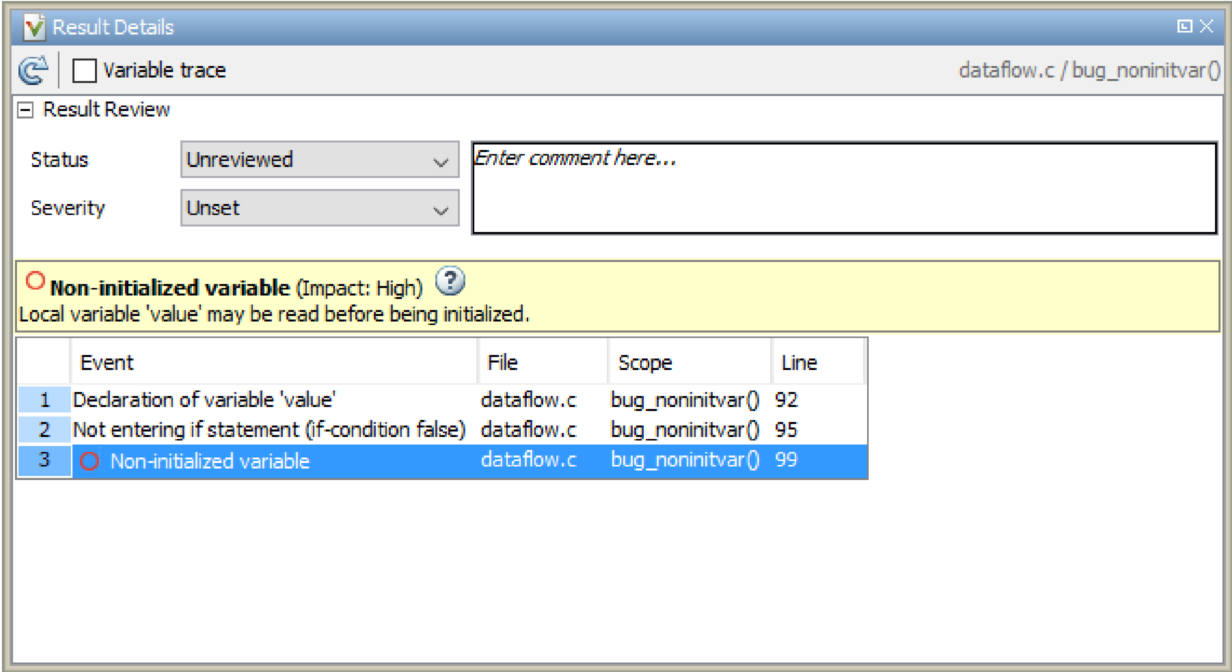
At this point, you might be ready to decide whether to fix the issue or not. Once you identify a fix, it might help to review all results of that type together.

Find Root Cause of Result

Sometimes, the root cause might be far from the actual location where the result is displayed. For instance, a variable that you read might be non-initialized because the initialization is not reachable. The defect is shown when you read the variable, but the root cause is perhaps a previous `if` or `while` condition that is always false.

Navigate to Related Events

Typically, the **Result Details** pane shows one sequence of events that leads to the result. The **Source** pane also highlights these events.



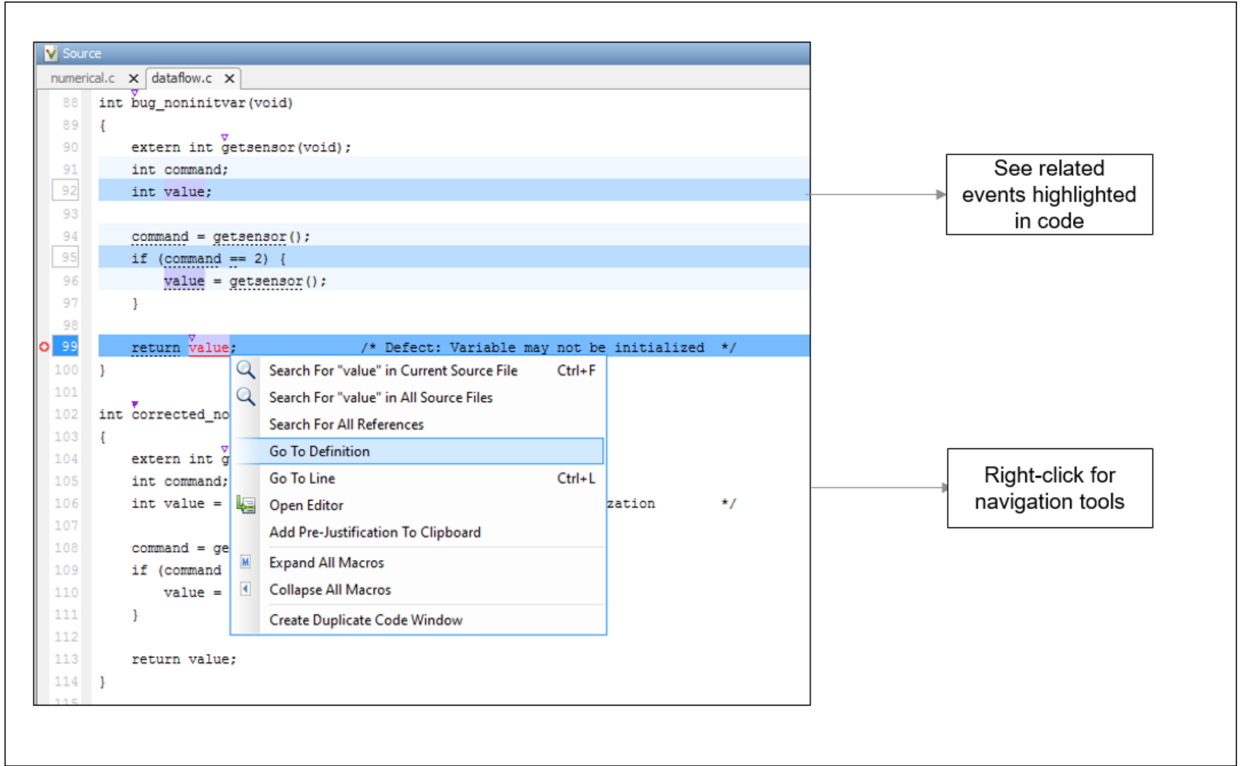
In the above event traceback, this sequence is shown:

1. A variable `value` is declared.
2. The execution path bypasses an `if` statement. This information might be relevant if the variable is initialized inside the `if` block.
3. Location of the current defect: **Non-initialized variable**

Typically, the traceback shows major points in the control flow: entering or bypassing conditional statements or loops, entering a function, and so on. For specific defects, the traceback shows other kinds of events relevant to the defect. For instance, for a **Declaration mismatch** defect, the traceback shows the two locations with conflicting declarations.

Create Your Own Navigation Path

If the event traceback is not available, use other navigation tools to trace your own path through the code.



Before you begin navigating through pathways in your code, ask the question: What am I looking for? Based on your answer, choose the appropriate navigation tool. For instance:

- To investigate a **Non-initialized variable** defect, you might want to make sure that the variable is not initialized at all. To look for previous instances of the variable, on the **Source** pane, right-click the variable and select **Search For All References**.

Alternatively, double-click the variable. These options show only instances of a specific variable and not other variables with the same name in other scopes.

- To investigate a violation of **MISRA C:2012 Rule 17.7**:

  ```
  The value returned by a function having non-void return type shall be used.
  ```

  you might want to navigate from a function call to the function definition. Right-click the function and select **Go To Definition**.

After you navigate away from the current result, use the icon on the **Result Details** pane to come back.

Navigate in Separate Window

If reviewing a result requires deeper navigation in your source code, you can create a duplicate source code window that focuses on the result while you navigate in the original source code window.

Right-click on the **Source** pane and select **Create Duplicate Code Window**. Right-click on the tab showing the duplicate file name (ending with `-spawn 1`) and select **New Vertical Group**.

Perform the navigation steps in the duplicate file window while the defect still appears on the original file window. After the investigation is over, close the duplicate window.

See Also

More About

- "Address Polyspace Results Through Bug Fixes or Comments" on page 15-2
- "Filter and Group Results" on page 16-2

Investigate the Cause of Empty Results List

When you run an analysis with Polyspace Bug Finder, the **Results List** pane can be empty or it can display this message:

```
Polyspace Bug Finder did not find any defects or coding rule violations in your code.
```

The message can indicate that your code has no defect or coding rule violation. However, before you reach this conclusion, check the following.

Possible Cause	Action to Take
Did all your source files compile?	In the **Output Summary** pane, look for warning messages that start with: `Failed compilation.` If a file does not compile, Bug Finder can return some results, but only files with no compilation errors are fully analyzed.
Did you include all your source files in your project?	In the **Project Browser** pane, make sure that all the files that you want to analyze are included in the **Project Source Files** folder.
Did you configure your project correctly?	In the **Configuration** pane: • Under **Coding Rules & Code Metrics**, verify that you have selected the appropriate options if you want to check **Coding Rules** and compute **Code Metrics**. • Under **Bug Finder Analysis**, confirm that you have selected all the defects that you want to check during the analysis. • Under **Run Settings**, see if **Use fast analysis mode for Bug Finder** is selected. In this mode, Bug Finder checks for only a subset of defects and coding rules.

Possible Cause	Action to Take
Are you applying any filters to the results?	In the **Results List** pane header, make sure that there are no **Hidden results** in the **Showing** drop-down list. To clear all applied filters, click **Clear active filters**.

See Also

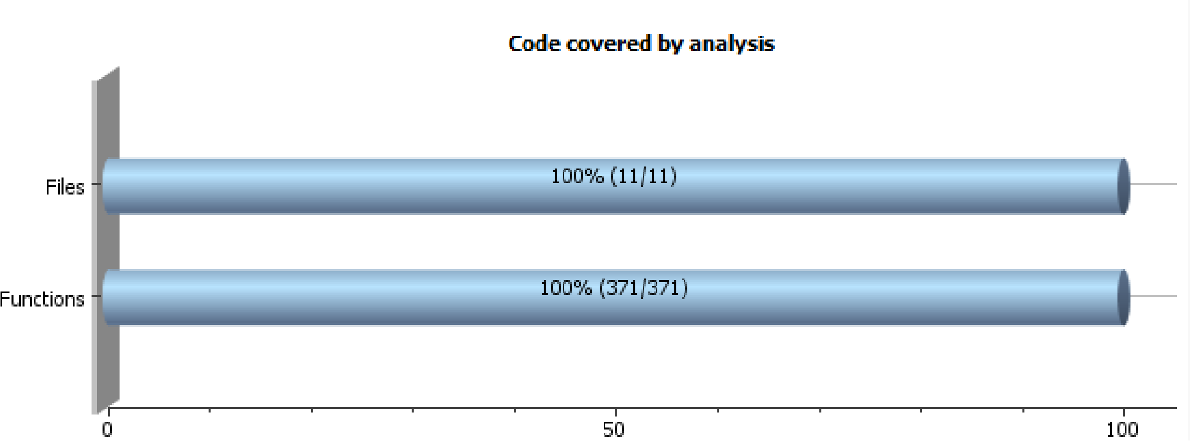
"Address Polyspace Results Through Bug Fixes or Comments" on page 15-2 |
"Troubleshooting in Polyspace Bug Finder"

Dashboard

The **Dashboard** pane provides statistics on the analysis results in a graphical format.

When you open a results file in Polyspace, this pane is displayed by default. You can view the following graphs:

- **Code covered by analysis**

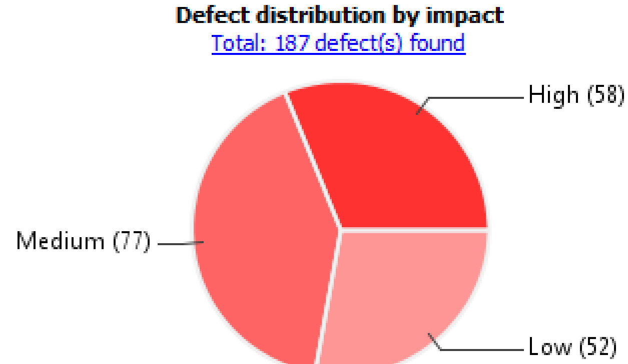


From this graph you can obtain the following information:

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

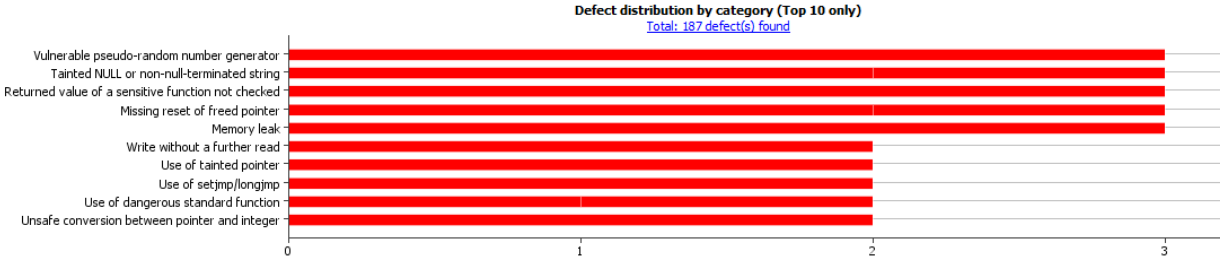
  If some of your files were only partially analyzed because of compilation errors, this pane contains a link stating that some files failed to compile. To see the compilation errors, click the link and navigate to the **Output Summary** pane.
- **Functions**: Ratio of analyzed functions to total number 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.

- **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 16-11.

- **Defect distribution by category or file**



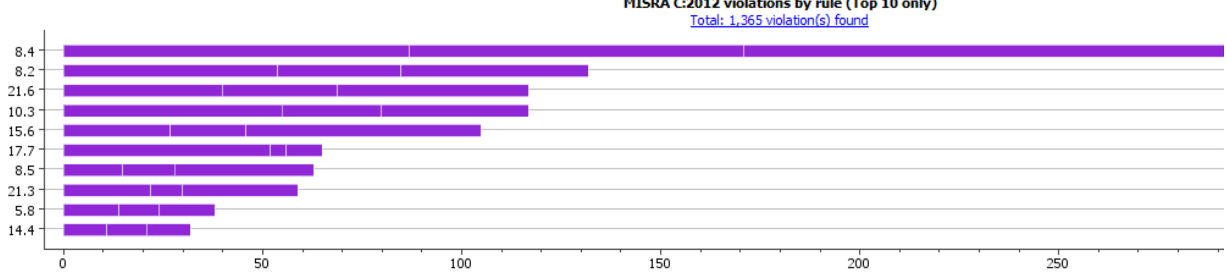
From this graph you can obtain the following information.

<table>
<tr><th></th><th>Category</th><th>File</th></tr>
<tr><td>Top 10</td><td>The ten defect types with the highest number of individual defects.
Each column represents a defect type and is divided into the:
File with highest number of defects of this type.
File with second highest number of defects of this type.
All other files with defects of this type.
Place your cursor on a column to see the file name and number of defects of this type in this file.
The x-axis represents the number of defects.
Use this view to organize your check review starting at defect types with more individual defects.</td><td>The ten source files with the highest number of defects.
Each column represents a file and is divided into the:
Defect type with highest number of defects in this file.
Defect type with second highest number of defects in this file.
All other defect types in this file.
Place your cursor on a column to see the defect type name and number of defects of this type in this file.
The x-axis represents the number of defects.
Use this view to organize your check review starting at files with more defects.</td></tr>
<tr><td>Bottom 10</td><td>The ten defect types with the lowest number of individual defects. Each column on the graph is divided the same way as the Top 10 defect types.

Use this view to organize your check review starting at defect types with fewer individual defects.</td><td>The ten source files with the lowest number of defects. Each column on the graph is divided the same way as the Top 10 files.

Use this view to organize your check review starting at files with fewer defects.</td></tr>
</table>

- **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.

<table>
<tr><th></th><th>Category</th><th>File</th></tr>
<tr><td>Top 10</td><td>The ten rules with the highest number of violations.
Each column represents a rule number and is divided into the:
File with highest number of violations of this rule.
File with second highest number of violations of this rule.
All other files with violations of this rule.
Place your cursor on a column to see the file name and number of violations of this rule in the file.
The x-axis represents the number of rule violations.
Use this view to organize your review starting at rules with more violations.</td><td>The ten source files containing the highest number of violations.
Each column represents a file and is divided into the:
Rule with highest number of violations in this file.
Rule with second highest number of violations in this file.
All other rules violated in this file.
Place your cursor on a column to see the rule number and number of violations of the rule in this file.
The x-axis represents the number of rule violations.
Use this view to organize your review starting at files with more rule violations.</td></tr>
<tr><td>Bottom 10</td><td>The ten rules with the lowest number of violations. Each column on the graph is divided in the same way as the Top 10 rules.

Use this view to organize your review starting at rules with fewer violations.</td><td>The ten source files containing the lowest number of rule violations. Each column on the graph is divided in the same way as the Top 10 files.

Use this view to organize your review starting at files with fewer rule violations.</td></tr>
</table>

For a list of supported coding rules, see “Supported MISRA C:2004 and MISRA AC AGC Rules” on page 12-3, “Supported MISRA C++ Coding Rules” on page 12-86 and “Supported JSF C++ Coding Rules” on page 12-124.

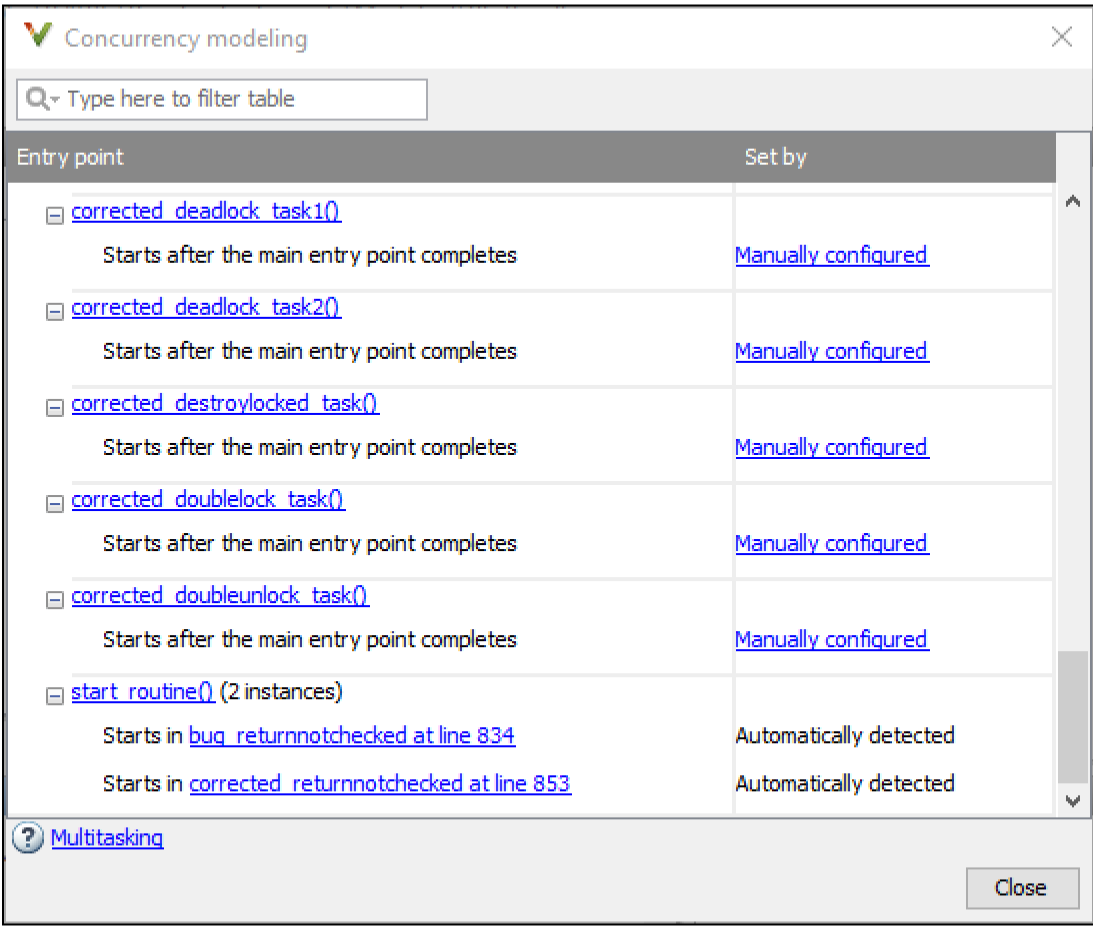
You can also perform the following actions on this pane:

- Select elements on the graphs to filter results from the **Results List** pane. See “Filter and Group Results” on page 16-2.

- View the configuration used to obtain the result. Select the link **Configuration**.
- View the modeling of the multitasking configuration of your code. Select the link **Concurrency modeling on page 14-17**.

Concurrency Modeling

The **Concurrency Modeling** view displays all the tasks and interrupts that the analysis extracts from your code and your Polyspace multitasking configuration.



in the table, the functions are listed in the first column by order of decreasing priority. The second column shows how Polyspace detects each task or interrupt: automatically, manually from the Polyspace configuration, or from an external file.

From this view, you can:

- Click a function name to go to its definition in the source code.
- Click an event to go to the corresponding call to the concurrency primitive in the source code, for instance `pthread_create`.
- Click **Manually configured**, for functions that are manually configured, to go to the **Multitasking** node on the **Configuration** pane.

Results List

The **Results List** pane lists all results along with their attributes. To organize your results review, from the 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 13-3.
- **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 result, the **Results List** pane contains the result attributes, listed in columns:

Attribute	Description
Family	Group to which the result belongs.
ID	Unique identification number of the result.
Type	Defect or coding rule violation.
Group	Category of the result, for instance: • For defects: Groups such as static memory, numerical, control flow, concurrency, etc. • For coding rule violations: Groups defined by the coding rule standard. For instance, MISRA C: 2012 defines groups related to code constructs such as functions, pointers and arrays, etc.

Attribute	Description
Check	Result name, for instance: • For defects: Defect name • For coding rule violations: Coding rule number
Detail	Additional information about a result. The column shows the first line of the **Result Details** pane. For an example of how to use this column, see the result `MISRA C:2012 Dir 1.1`.
File	File containing the instruction where the result occurs
Class	Class containing the instruction where the result occurs. If the result is not inside a class definition, then this column contains the entry, **Global Scope**.
Function	Function containing the instruction where the result occurs. If the function is a method of a class, it appears in the format *`class_name`*`::`*`function_name`*.
Folder	Path to the folder that contains the source file with the result
CWE ID, **CERT ID** or **ISO-17961 ID**	CWE, CERT C99 or ISO/IEC TS 17961 IDs corresponding to the Bug Finder results. See: • "CWE Coding Standard and Polyspace Results" on page 13-49 • "CERT C Coding Standard and Polyspace Results" on page 13-97 • "ISO/IEC TS 17961 Coding Standard and Polyspace Results" on page 13-163

Attribute	Description
Severity	Level of severity you have assigned to the result. The possible levels are: • `Unset` • `High` • `Medium` • `Low`
Status	Review status you have assigned to the result. The possible statuses are: • `Unreviewed` (default status) • `To investigate` • `To fix` • `Justified` • `No action planned` • `Not a defect` • `Other`
Comments	Comments you have entered about the result

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 results.
- Organize your result review using filters on the columns. For more information, see "Filter and Group Results" on page 16-2.

Source

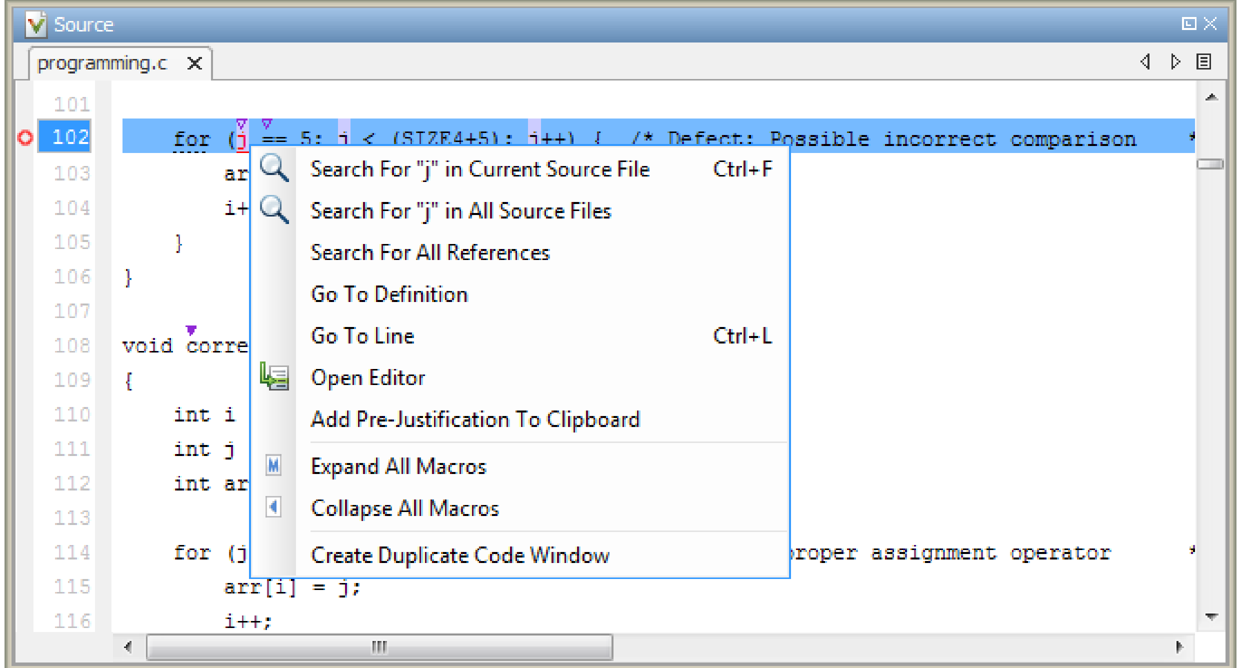
The **Source** pane shows the source code with the defects colored in red and the corresponding line number marked by ⭘.

Tooltips

Placing your cursor over a result 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 "j" in Current Source** — List occurrences of the string within the current source file on the **Search** pane.
- **Search "j" 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. If a definition is not available to Polyspace, selecting the option takes you to the declaration.
- **Go To Line** — Open the Go to line dialog box. If you specify a line number and click **Enter**, the software displays the specified line of code.
- **Expand All Macros** or **Collapse All Macros** — Display or hide the content of macros in current source file.

Expand Macros

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

Source

numerical.c

```
329
330 div_t corrected_intstdlib(int num, int denom)
331 {
332     div_t test = { .quot=0, .rem=0};
333
334     if ( denom != 0 && !(num == INT_MIN ) && ( denom == -1 ) )
335         test = div(num, denom); /*Fix: Exclude problematic values */
336
337     return test;
338 }
339
340
```

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

Source
numerical.c

```
329
330 div_t corrected_intstdlib(int num, int denom)
331 {
332     div_t test = { .quot=0, .rem=0};
333
334     if ( denom != 0 && !(num == (-0x7fffffffL - 1) ) && ( denom == -1 ) )
335         test = div(num, denom); /*Fix: Exclude problematic values */
336
337     return test;
338 }
339
340
```

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**.

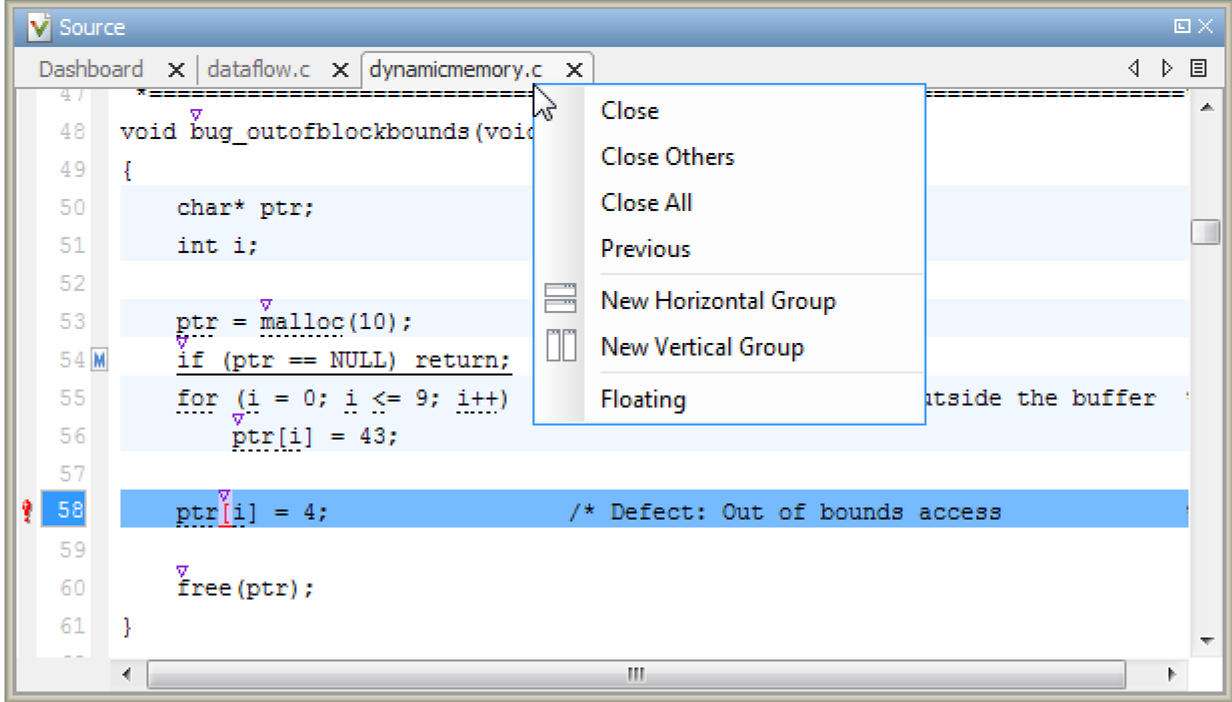
Note

1. The **Result Details** pane also allows you to view the contents of a macro if the check you select lies within a macro.
2. You cannot expand OSEK API macros in the **Source** pane.

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 χ 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

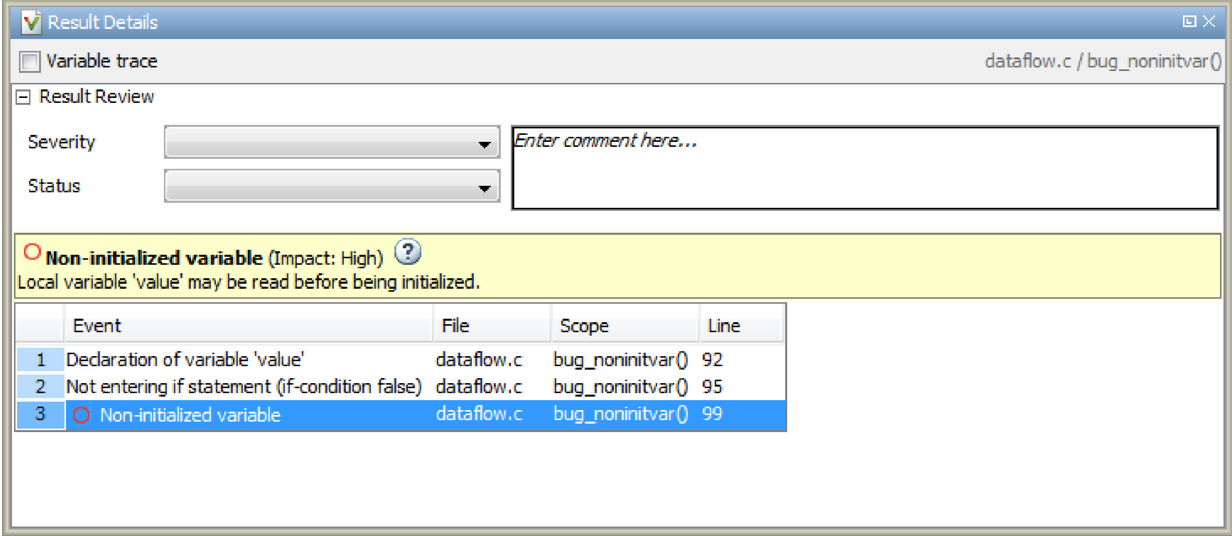
On the **Source** pane, to highlight a block of code, click either its opening or closing brace. If the brace itself is highlighted, click the brace twice.

```
196  *=========================================================================
197 int do_anything(void);
198
199 int bug_deactivatedcode(int val)
200 {
201     int retval;
202     retval = do_anything();
203
204     if (0 /*|| retval < 0*/) {       /* Defect: Deactivated (debug) code
205         retval = do_anything();
206     }
207
208     return retval;
209 }
210
```

Result Details

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



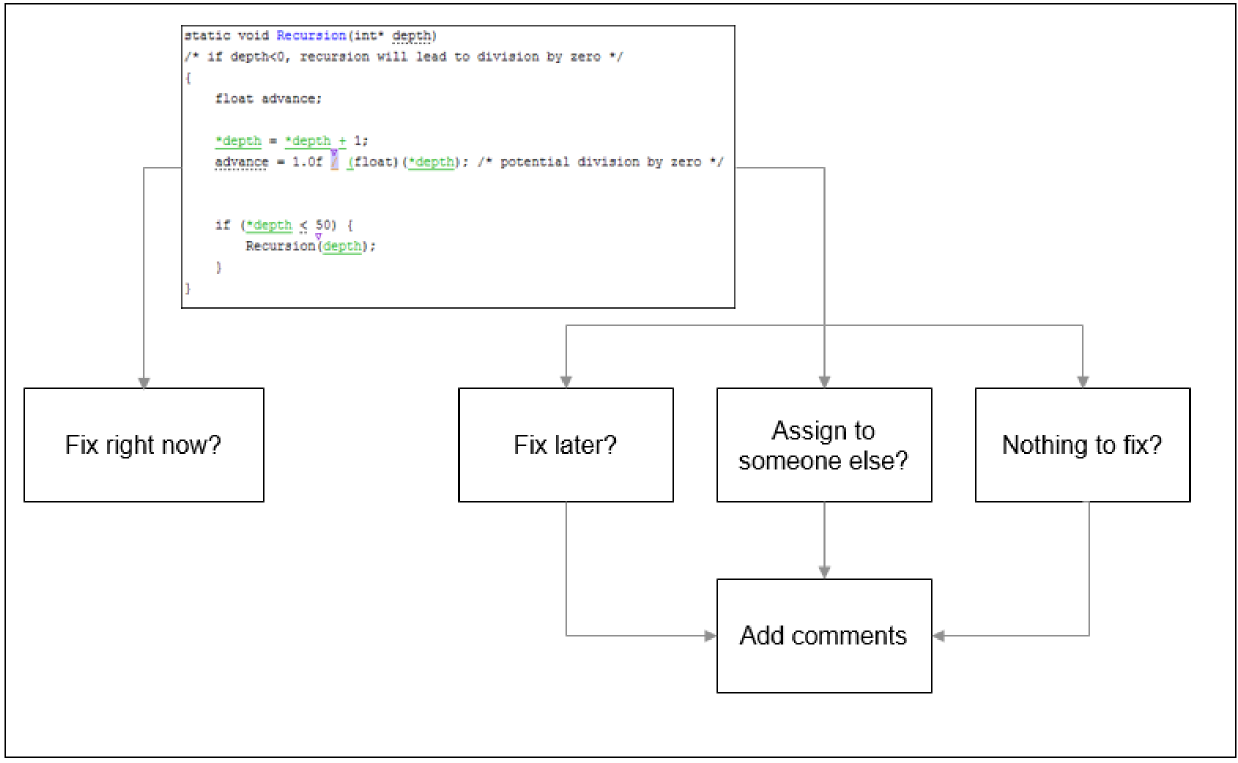
- The top right corner shows the file and function containing the defect, in the format *file_name/function_name*.
- The yellow box contains the name of the defect with an explanation of why the defect occurs.
- The **Event** column lists the sequence of code instructions causing the defect. The **Scope** column lists the function containing the instructions. If the instructions are not in a function, the column lists the file 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.

Fix or Comment Polyspace Results

- “Address Polyspace Results Through Bug Fixes or Comments” on page 15-2
- “Annotate Code and Hide Known or Acceptable Results” on page 15-6
- “Short Names of Bug Finder Defect Checkers” on page 15-12
- “Annotate Code for Known or Acceptable Results (Deprecated)” on page 15-27
- “Define Custom Annotation Format” on page 15-32
- “Annotation Description Full XML Template” on page 15-42
- “Import Comments from Previous Polyspace Analysis” on page 15-49
- “Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results” on page 15-52

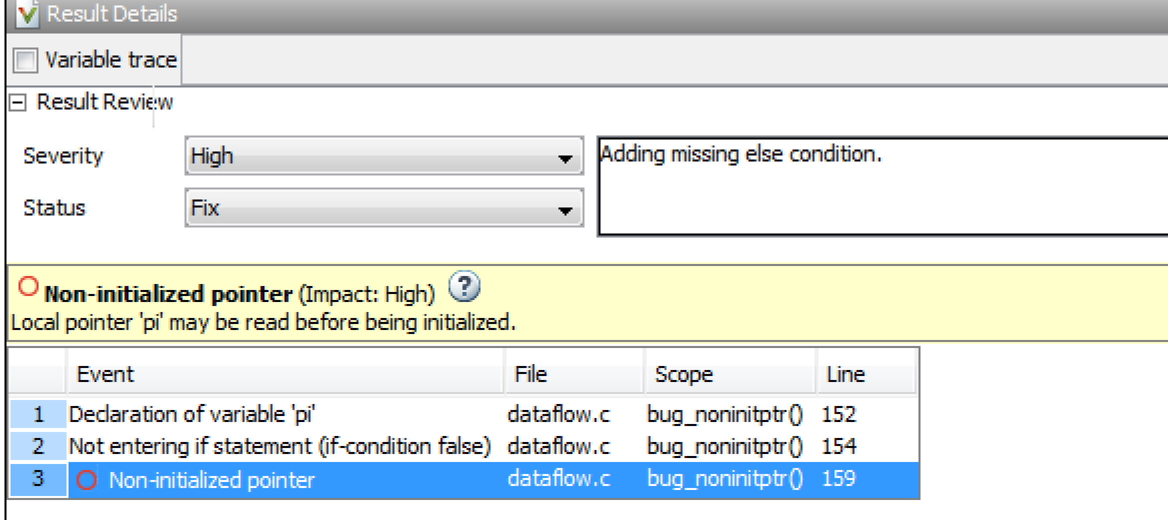
Address Polyspace Results Through Bug Fixes or Comments

Once you understand the root cause of a Polyspace finding, you can fix your code. Otherwise, add comments to your Polyspace results to fix the code later or to justify the result. You can use the comments to keep track of your review progress.



If you add comments to your results file, they carry over to the next analysis on the same project. If you add comments to your code (annotate), they carry over to any subsequent analysis of the code, whether in the same project or not. You can also hide results using code annotations.

Comment in Results File

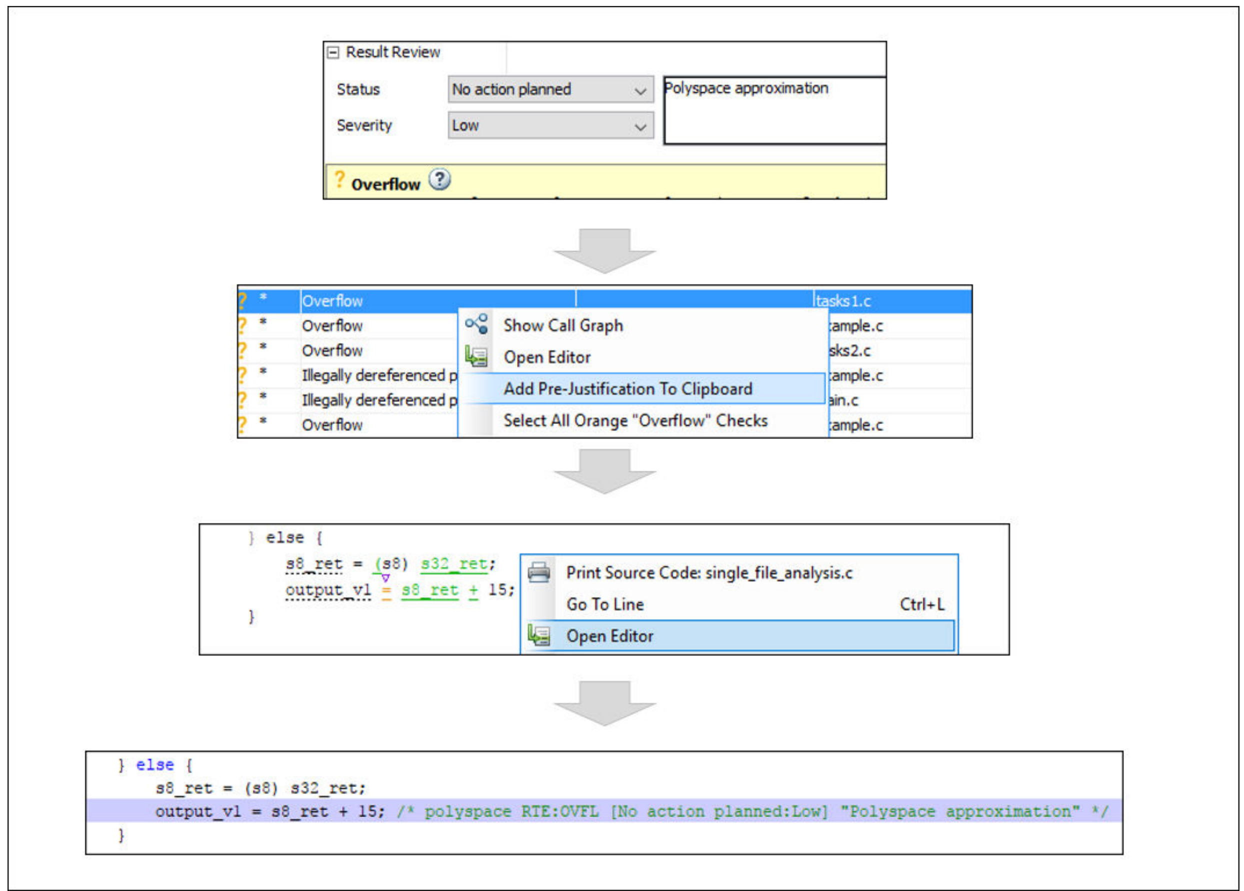


You can comment either on the **Results List** or **Result Details** pane. To comment, select a result, then set the **Severity** and **Status** fields, and optionally, enter notes with more explanations. The status indicates your response to the Polyspace result. If you do not plan to fix your code in response to a result, assign one of the following statuses:

- `Justified`
- `No Action Planned`
- `Not a Defect`

Based on the status, Polyspace considers that you have given due consideration and justified that result (retained the code despite the result).

Comment or Annotate in Code



If you enter code comments or annotations in a specific syntax, the software can read them and populate the **Severity**, **Status** and **Comment** fields in the next analysis of the code.

You can either type the annotation directly or copy it from the user interface. In the user interface, to copy annotations, right-click a result and select **Add Pre-Justification To**

Clipboard. Open your source code in an editor and paste *on the same line as* the result. If you follow this workflow, Polyspace assumes that you have set a status of `No Action Planned`. The software hides the result from all places (except reports needed for certification). The only exceptions are the safety-critical Code Prover run-time checks, which are hidden from the results list but not the source code.

If you want to explicitly set a status, first fill the **Status** field for a result and then copy to your code. Paste on the line containing the result.

If you want to directly type the annotation, see the annotation syntax in “Annotate Code and Hide Known or Acceptable Results” on page 15-6.

See Also

More About

- “Annotate Code and Hide Known or Acceptable Results” on page 15-6
- “Import Comments from Previous Polyspace Analysis” on page 15-49

Annotate Code and Hide Known or Acceptable Results

If a Polyspace analysis of your code finds known or acceptable defects or coding rule violations, you can suppress the defects or violations in subsequent analyses. Add code annotations indicating that you have reviewed the issues and that you do not intend to fix them.

You can add annotations through the Polyspace user interface or by typing them directly in your code. For the general workflow, see “Address Polyspace Results Through Bug Fixes or Comments” on page 15-2. This topic shows the annotation syntax.

Code Annotation Syntax

To add comments directly to your source file, use the Polyspace annotation syntax. The syntax is not case sensitive, and has this format:

- Annotation for current line of code:

  ```
  line of code; /* polyspace Family:Result_name */
  ```

- Annotation for current line of code and `n` following lines:

  ```
  code; /* polyspace +n Family:Result_name */
  ```

- Annotation for block of code:

  ```
  /* polyspace-begin Family:Result_name */
  code;
  /* polyspace-end Family:Result_name */
  ```

Annotations begin with the keyword `polyspace` and must include *`Family`* and *`Result_name`* field values. You can optionally specify *`Status`*, *`Severity`*, and *`Comment`* field values.

```
polyspace Family:Result_name [Status:Severity] "Comment"
```

If you do not specify a status, Polyspace considers the result justified, and assigns the status `No action planned` to the result.

To replace the different annotation fields with their allowed values, use the values in this table or see the examples on page 15-9.

Field	Allowed Value
Family	Type of analysis result: • `DEFECT` (Polyspace Bug Finder) • `RTE`, for run-time checks (Polyspace Code Prover) • `VARIABLE`, for global variables (Polyspace Code Prover) • `MISRA-C` or `MISRA2004` • `MISRA-C3` or `MISRA2012` • `MISRA-AC-AGC` • `JSF` • `CUSTOM` To specify all analysis results, use the asterisk character `*:*`.
Result_name	For `DEFECT`, use short names of checkers. See: “Short Names of Bug Finder Defect Checkers” on page 15-12. For `RTE`, use short names of run-time checks. See: “Short Names of Code Prover Run-Time Checks” (Polyspace Code Prover). For `VARIABLE`, the only allowed value is the asterisk character " * ". For coding rule violations, specify the rule number or numbers. To specify all parts of a result name `[MISRA2012:17.*]` or all result names in a family `[DEFECT:*]`, use the asterisk character.

Field	Allowed Value
Status	Text to indicate how you intend to address the error in your code. This value populates the **Status** column in the **Results List** pane as: • `Unreviewed` • `To investigate` • `To fix` • `Justified` • `No action planned` • `Not a defect` • `Other` Polyspace suppresses results annotated with status `Justified`, `No action planned`, or `Not a defect` in subsequent analyses. If you specify a status that is not an allowed value, Polyspace stores it as a custom status.

Field	Allowed Value
Severity	Text to indicate how critical you consider the error in your code. This value populates the **Severity** column in the **Results List** pane as: • `Unset` • `High` • `Medium` • `Low` If you specify a severity that is not an allowed value, Polyspace appends it to the status field and stores it as a custom status. For example, `[To investigate:sporadic]` is displayed in the **Status** column of the **Results List** pane as `To investigate sporadic`.
Comment	Additional text, such as a keyword or an explanation for the status and severity. This value populates the **Comment** column in the **Results List** pane.

Syntax Examples

Suppress a Single Defect

Enter an annotation on the same line as the defect and specify the *`Family`* (`DEFECT`) and the *`Result_name`* (`INT_OVFL`). When you do not specify a status, Polyspace assigns the status `No action planned`, and then suppresses the result in subsequent analyses.

```
int var = INT_MAX;
var++;/* polyspace DEFECT:INT_OVFL */
```

Suppress All MISRA C: 2012 Violations Over Multiple Lines

Enter an annotation with `+n` between `polyspace` and the *`Family`*:*`Result_name`* entries. The annotation applies to the same line and the `n` following lines.

This annotation applies to lines 4–7. The line count includes code, comments, and blank lines.

```
4. code ; // polyspace +3 MISRA2012:*
5. //comment
6.
7. code;
8. code;
```

Specify Multiple Families in the Same Annotation

Enter each family separated by a space.This annotation applies to all MISRA C:2012 rules 17 and to all run-time checks.

```
some code; /* polyspace MISRA2012:17.* RTE:* */
```

Specify Multiple Result Names in the Same Annotation

After you specify the *Family* (DEFECT), enter each *Result_name* separated by a comma.

```
  system("rm ~/.config"); /* polyspace DEFECT:UNSAFE_SYSTEM_CALL,RETURN_NOT_CHECKED */
```

Add Explanatory Comments to Annotation

After you specify a *Family* and a *Result_name*, you can add a *Comment* with additional information for your justification. You can provide a comment for all families and result names, or a comment for each family or result name.

```
//Single comment

code; /* polyspace DEFECT:BAD_FREE MISRA2004:* "OK Defect and MISRA" */
//Multiple comments incorrect syntax:

code; /* polyspace DEFECT:* "OK defect" MISRA2004:5.2 "OK MISRA" */

//Multiple comments correct syntax:
code; /* polyspace DEFECT:* "OK defect" polyspace MISRA2004:5.2 "OK MISRA" */
```

In annotations, Polyspace ignores all text following double quotes. To specify additional *Family*:*Result_name*, [*Status*:*Severity*] or *Comment* entries, you must reenter the keyword `polyspace` after text in double quotes.

Set Status and Severity

You can specify allowed values on page 15-6 or enter custom values for status and severity. A custom severity entry is appended to the status and stored as a custom **Status** in the user interface.

```
//Set Status only
code; /* polyspace DEFECT:* [To fix] "some comment" */

//Set Status 'To fix' and Severity 'High'
code; /* polyspace VARIABLE:* [To fix: High] "some comment"*/

//Set custom status 'Assigned' and Severity 'Medium'
code; /* polyspace MISRA2012:12.* [Assigned: Medium] */
```

See Also

`-xml-annotations-description`

More About

- “Define Custom Annotation Format” on page 15-32
- “Short Names of Bug Finder Defect Checkers” on page 15-12

Short Names of Bug Finder Defect Checkers

To justify defects through code annotations, use the command-line names, or short names, listed in the following table.

You can also enable the detection of a specific defect by using its short name as argument of the `-checkers` option. To specify groups of defects, see `Find defects (-checkers)`.

Defect	Command-line Name
`*this not returned in copy assignment operator`	`RETURN_NOT_REF_TO_THIS`
`Abnormal termination of exit handler`	`EXIT_ABNORMAL_HANDLER`
`Absorption of float operand`	`FLOAT_ABSORPTION`
`Accessing object with temporary lifetime`	`TEMP_OBJECT_ACCESS`
`Alignment changed after memory reallocation`	`ALIGNMENT_CHANGE`
`Alternating input and output from a stream without flush or positioning call`	`IO_INTERLEAVING`
`Arithmetic operation with NULL pointer`	`NULL_PTR_ARITH`
`Array access out of bounds`	`OUT_BOUND_ARRAY`
`Array access with tainted index`	`TAINTED_ARRAY_INDEX`
`Assertion`	`ASSERT`
`Bad file access mode or status`	`BAD_FILE_ACCESS_MODE_STATUS`
`Bad order of dropping privileges`	`BAD_PRIVILEGE_DROP_ORDER`

Defect	Command-line Name
Base class assignment operator not called	MISSING_BASE_ASSIGN_OP_CALL
Base class destructor not virtual	DTOR_NOT_VIRTUAL
Bitwise and arithmetic operation on the same data	BITWISE_ARITH_MIX
Bitwise operation on negative value	BITWISE_NEG
Buffer overflow from incorrect string format specifier	STR_FORMAT_BUFFER_OVERFLOW
Call through non-prototyped function pointer	UNPROTOTYPED_FUNC_CALL
Call to memset with unintended value	MEMSET_INVALID_VALUE
Character value absorbed into EOF	CHAR_EOF_CONFUSED
Closing a previously closed resource	DOUBLE_RESOURCE_CLOSE
Code deactivated by constant false condition	DEACTIVATED_CODE
Command executed from externally controlled path	TAINTED_PATH_CMD
Constant block cipher initialization vector	CRYPTO_CIPHER_CONSTANT_IV
Constant cipher key	CRYPTO_CIPHER_CONSTANT_KEY
Context initialized incorrectly for cryptographic operation	CRYPTO_PKEY_INCORRECT_INIT

Defect	Command-line Name
Context initialized incorrectly for digest operation	CRYPTO_MD_BAD_FUNCTION
Copy constructor not called in initialization list	MISSING_COPY_CTOR_CALL
Copy of overlapping memory	OVERLAPPING_COPY
Data race	DATA_RACE
Data race including atomic operations	DATA_RACE_ALL
Data race through standard library function call	DATA_RACE_STD_LIB
Dead code	DEAD_CODE
Deadlock	DEADLOCK
Deallocation of previously deallocated pointer	DOUBLE_DEALLOCATION
Declaration mismatch	DECL_MISMATCH
Delete of void pointer	DELETE_OF_VOID_PTR
Destination buffer overflow in string manipulation	STRLIB_BUFFER_OVERFLOW
Destination buffer underflow in string manipulation	STRLIB_BUFFER_UNDERFLOW
Destruction of locked mutex	DESTROY_LOCKED
Deterministic random output from constant seed	RAND_SEED_CONSTANT

Defect	Command-line Name
Double lock	DOUBLE_LOCK
Double unlock	DOUBLE_UNLOCK
Environment pointer invalidated by previous operation	INVALID_ENV_POINTER
Errno not checked	ERRNO_NOT_CHECKED
Errno not reset	MISSING_ERRNO_RESET
Exception caught by value	EXCP_CAUGHT_BY_VALUE
Exception handler hidden by previous handler	EXCP_HANDLER_HIDDEN
Execution of a binary from a relative path can be controlled by an external actor	RELATIVE_PATH_CMD
Execution of externally controlled command	TAINTED_EXTERNAL_CMD
File access between time of check and use (TOCTOU)	TOCTOU
File descriptor exposure to child process	FILE_EXPOSURE_TO_CHILD
File manipulation after chroot without chdir	CHROOT_MISUSE
Float conversion overflow	FLOAT_CONV_OVFL
Float division by zero	FLOAT_ZERO_DIV
Floating point comparison with equality operators	BAD_FLOAT_OP
Float overflow	FLOAT_OVFL

Defect	Command-line Name
Format string specifiers and arguments mismatch	STRING_FORMAT
Function called from signal handler not asynchronous-safe	SIG_HANDLER_ASYNC_UNSAFE
Function called from signal handler not asynchronous-safe (strict)	SIG_HANDLER_ASYNC_UNSAFE_STRICT
Function pointer assigned with absolute address	FUNC_PTR_ABSOLUTE_ADDR
Hard-coded buffer size	HARD_CODED_BUFFER_SIZE
Hard-coded loop boundary	HARD_CODED_LOOP_BOUNDARY
Hard-coded object size used to manipulate memory	HARD_CODED_MEM_SIZE
Host change using externally controlled elements	TAINTED_HOSTID
Improper array initialization	IMPROPER_ARRAY_INIT
Incompatible padding for RSA algorithm operation	CRYPTO_RSA_BAD_PADDING
Incompatible types prevent overriding	VIRTUAL_FUNC_HIDING
Inconsistent cipher operations	CRYPTO_CIPHER_BAD_FUNCTION
Incorrect data type passed to va_arg	VA_ARG_INCORRECT_TYPE
Incorrect key for cryptographic algorithm	CRYPTO_PKEY_INCORRECT_KEY

Defect	Command-line Name
Incorrect order of network connection operations	BAD_NETWORK_CONNECT_ORDER
Incorrect pointer scaling	BAD_PTR_SCALING
Information leak via structure padding	PADDING_INFO_LEAK
Inline constraint not respected	INLINE_CONSTRAINT_NOT_RESPECTED
Integer conversion overflow	INT_CONV_OVFL
Integer division by zero	INT_ZERO_DIV
Integer overflow	INT_OVFL
Invalid assumptions about memory organization	INVALID_MEMORY_ASSUMPTION
Invalid deletion of pointer	BAD_DELETE
Invalid file position	INVALID_FILE_POS
Invalid free of pointer	BAD_FREE
Invalid use of = (assignment) operator	BAD_EQUAL_USE
Invalid use of == (equality) operator	BAD_EQUAL_EQUAL_USE
Invalid use of standard library floating point routine	FLOAT_STD_LIB
Invalid use of standard library integer routine	INT_STD_LIB
Invalid use of standard library memory routine	MEM_STD_LIB

Defect	Command-line Name
Invalid use of standard library routine	OTHER_STD_LIB
Invalid use of standard library string routine	STR_STD_LIB
Invalid va_list argument	INVALID_VA_LIST_ARG
Large pass-by-value argument	PASS_BY_VALUE
Library loaded from externally controlled path	TAINTED_PATH_LIB
Line with more than one statement	MORE_THAN_ONE_STATEMENT
Load of library from a relative path can be controlled by an external actor	RELATIVE_PATH_LIB
Loop bounded with tainted value	TAINTED_LOOP_BOUNDARY
Member not initialized in constructor	NON_INIT_MEMBER
Memory allocation with tainted size	TAINTED_MEMORY_ALLOC_SIZE
Memory comparison of float-point values	MEMCMP_FLOAT
Memory comparison of padding data	MEMCMP_PADDING_DATA
Memory comparison of strings	MEMCMP_STRINGS
Memory leak	MEM_LEAK
Mismatch between data length and size	DATA_LENGTH_MISMATCH

Defect	Command-line Name
Mismatched alloc/dealloc functions on Windows	WIN_MISMATCH_DEALLOC
Missing blinding for RSA algorithm	CRYPTO_RSA_NO_BLINDING
Missing block cipher initialization vector	CRYPTO_CIPHER_NO_IV
Missing break of switch case	MISSING_SWITCH_BREAK
Missing byte reordering when transferring data	MISSING_BYTESWAP
Missing case for switch condition	MISSING_SWITCH_CASE
Missing cipher algorithm	CRYPTO_CIPHER_NO_ALGORITHM
Missing cipher data to process	CRYPTO_CIPHER_NO_DATA
Missing cipher final step	CRYPTO_CIPHER_NO_FINAL
Missing cipher key	CRYPTO_CIPHER_NO_KEY
Missing data for encryption, decryption or signing operation	CRYPTO_PKEY_NO_DATA
Missing explicit keyword	MISSING_EXPLICIT_KEYWORD
Missing lock	BAD_UNLOCK
Missing null in string array	MISSING_NULL_CHAR
Missing padding for RSA algorithm	CRYPTO_RSA_NO_PADDING
Missing parameters for key generation	CRYPTO_PKEY_NO_PARAMS
Missing peer key	CRYPTO_PKEY_NO_PEER
Missing private key	CRYPTO_PKEY_NO_PRIVATE_KEY

Defect	Command-line Name
Missing public key	CRYPTO_PKEY_NO_PUBLIC_KEY
Missing reset of a freed pointer	MISSING_FREED_PTR_RESET
Missing return statement	MISSING_RETURN
Missing unlock	BAD_LOCK
Missing virtual inheritance	MISSING_VIRTUAL_INHERITANCE
Misuse of a FILE object	FILE_OBJECT_MISUSE
Misuse of errno	ERRNO_MISUSE
Misuse of errno in a signal handler	SIG_HANDLER_ERRNO_MISUSE
Misuse of readlink()	READLINK_MISUSE
Misuse of return value from nonreentrant standard function	NON_REENTRANT_STD_RETURN
Misuse of sign-extended character value	CHARACTER_MISUSE
Misuse of structure with flexible array member	FLEXIBLE_ARRAY_MEMBER_STRUCT_MISUSE
Modification of internal buffer returned from nonreentrant standard function	WRITE_INTERNAL_BUFFER_RETURNED_FROM_STD_FUNC
Non-initialized pointer	NON_INIT_PTR
Non-initialized variable	NON_INIT_VAR
Nonsecure hash algorithm	CRYPTO_MD_WEAK_HASH
Nonsecure parameters for key generation	CRYPTO_PKEY_WEAK_PARAMS
Nonsecure RSA public exponent	CRYPTO_RSA_LOW_EXPONENT

Defect	Command-line Name
Nonsecure SSL/TLS protocol	CRYPTO_SSL_WEAK_PROTOCOL
Null pointer	NULL_PTR
Object slicing	OBJECT_SLICING
Opening previously opened resource	DOUBLE_RESOURCE_OPEN
Overlapping assignment	OVERLAPPING_ASSIGN
Partially accessed array	PARTIALLY_ACCESSED_ARRAY
Partial override of overloaded virtual functions	PARTIAL_OVERRIDE
Pointer access out of bounds	OUT_BOUND_PTR
Pointer dereference with tainted offset	TAINTED_PTR_OFFSET
Pointer or reference to stack variable leaving scope	LOCAL_ADDR_ESCAPE
Pointer to non-initialized value converted to const pointer	NON_INIT_PTR_CONV
Possible misuse of sizeof	SIZEOF_MISUSE
Possibly unintended evaluation of expression because of operator precedence rules	OPERATOR_PRECEDENCE
Predefined macro used as an object	MACRO_USED_AS_OBJECT
Predictable block cipher initialization vector	CRYPTO_CIPHER_PREDICTABLE_IV

Defect	Command-line Name
Predictable cipher key	CRYPTO_CIPHER_PREDICTABLE_KEY
Predictable random output from predictable seed	RAND_SEED_PREDICTABLE
Preprocessor directive in macro argument	PRE_DIRECTIVE_MACRO_ARG
Privilege drop not verified	MISSING_PRIVILEGE_DROP_CHECK
Qualifier removed in conversion	QUALIFIER_MISMATCH
Resource leak	RESOURCE_LEAK
Returned value of a sensitive function not checked	RETURN_NOT_CHECKED
Return from computational exception signal handler	SIG_HANDLER_COMP_EXCP_RETURN
Return of non const handle to encapsulated data member	BREAKING_DATA_ENCAPSULATION
Self assignment not tested in operator	MISSING_SELF_ASSIGN_TEST
Sensitive data printed out	SENSITIVE_DATA_PRINT
Sensitive heap memory not cleared before release	SENSITIVE_HEAP_NOT_CLEARED
Shared data access within signal handler	SIG_HANDLER_SHARED_OBJECT
Shift of a negative value	SHIFT_NEG
Shift operation overflow	SHIFT_OVFL

Defect	Command-line Name
Side effect of expression ignored	SIDE_EFFECT_IGNORED
Signal call from within signal handler	SIG_HANDLER_CALLING_SIGNAL
Sign change integer conversion overflow	SIGN_CHANGE
Standard function call with incorrect arguments	STD_FUNC_ARG_MISMATCH
Static uncalled function	UNCALLED_FUNC
Stream argument with possibly unintended side effects	STREAM_WITH_SIDE_EFFECT
Subtraction or comparison between pointers to different arrays	PTR_TO_DIFF_ARRAY
Tainted division operand	TAINTED_INT_DIVISION
Tainted modulo operand	TAINTED_INT_MOD
Tainted NULL or non-null-terminated string	TAINTED_STRING
Tainted sign change conversion	TAINTED_SIGN_CHANGE
Tainted size of variable length array	TAINTED_VLA_SIZE
Tainted string format	TAINTED_STRING_FORMAT
Too many va_arg calls for current argument list	TOO_MANY_VA_ARG_CALLS
Typedef mismatch	TYPEDEF_MISMATCH
Umask used with chmod-style arguments	BAD_UMASK

Defect	Command-line Name
Uncleared sensitive data in stack	SENSITIVE_STACK_NOT_CLEARED
Universal character name from token concatenation	PRE_UCNAME_JOIN_TOKENS
Unprotected dynamic memory allocation	UNPROTECTED_MEMORY_ALLOCATION
Unreachable code	UNREACHABLE
Unreliable cast of function pointer	FUNC_CAST
Unreliable cast of pointer	PTR_CAST
Unsafe call to a system function	UNSAFE_SYSTEM_CALL
Unsafe conversion between pointer and integer	BAD_INT_PTR_CAST
Unsafe conversion from string to numerical value	UNSAFE_STR_TO_NUMERIC
Unsafe standard encryption function	UNSAFE_STD_CRYPT
Unsafe standard function	UNSAFE_STD_FUNC
Unsigned integer conversion overflow	UINT_CONV_OVFL
Unsigned integer overflow	UINT_OVFL
Unused parameter	UNUSED_PARAMETER
Useless if	USELESS_IF
Use of automatic variable as putenv-family function argument	PUTENV_AUTO_VAR

Defect	Command-line Name
Use of dangerous standard function	DANGEROUS_STD_FUNC
Use of externally controlled environment variable	TAINTED_ENV_VARIABLE
Use of indeterminate string	INDETERMINATE_STRING
Use of memset with size argument zero	MEMSET_INVALID_SIZE
Use of non-secure temporary file	NON_SECURE_TEMP_FILE
Use of obsolete standard function	OBSOLETE_STD_FUNC
Use of path manipulation function without maximum sized buffer checking	PATH_BUFFER_OVERFLOW
Use of plain char type for numerical value	BAD_PLAIN_CHAR_USE
Use of previously closed resource	CLOSED_RESOURCE_USE
Use of previously freed pointer	FREED_PTR
Use of setjmp/longjmp	SETJMP_LONGJMP_USE
Use of tainted pointer	TAINTED_PTR
Variable length array with nonpositive size	NON_POSITIVE_VLA_SIZE
Variable shadowing	VAR_SHADOWING
Vulnerable path manipulation	PATH_TRAVERSAL
Vulnerable permission assignments	DANGEROUS_PERMISSIONS

Defect	Command-line Name
Vulnerable pseudo-random number generator	VULNERABLE_PRNG
Weak cipher algorithm	CRYPTO_CIPHER_WEAK_CIPHER
Weak cipher mode	CRYPTO_CIPHER_WEAK_MODE
Weak padding for RSA algorithm	CRYPTO_RSA_WEAK_PADDING
Write without a further read	USELESS_WRITE
Writing to const qualified object	CONSTANT_OBJECT_WRITE
Writing to read-only resource	READ_ONLY_RESOURCE_WRITE
Wrong allocated object size for cast	OBJECT_SIZE_MISMATCH
Wrong type used in sizeof	PTR_SIZEOF_MISMATCH

Annotate Code for Known or Acceptable Results (Deprecated)

Note Starting R2017b, Polyspace uses a simpler annotation format. See "Annotate Code and Hide Known or Acceptable Results" on page 15-6.

If Polyspace finds defects in your code that you cannot or will not fix, you can add annotations to your code. These annotations are code comments that indicate known or acceptable defects or coding rule violations. By using these annotations, you can:

- Avoid rereviewing defects or coding rule violations from previous analyses.
- Preserve review comments and classifications.

Note Source code annotations do not apply to code comments. You cannot annotate these rules:

- MISRA C:2004 Rules 2.2 and 2.3
- MISRA C:2012 Rules 3.1 and 3.2
- MISRA-C++ Rule 2-7-1
- JSF++ Rules 127 and 133

Add Annotations from the Polyspace Interface

This method shows you how to convert review comments and classifications in the Polyspace interface into code annotations.

1. On the **Results List** or **Result Details** pane, assign a **Severity**, **Status**, and **Comment** to a result.
   a. Click a result.
   b. From the **Severity** and **Status** dropdown lists, select an option.
   c. In the **Comment** field, enter a comment or keyword that helps you easily recognize the result.

2. On the **Results List** pane, right-click the commented result and select **Add Pre-Justification to Clipboard**. The software copies the severity, status, and comment to the clipboard.
3. Right-click the result again and select **Open Editor**. The software opens the source file to the location of the defect.
4. Paste the contents of your clipboard on the line immediately before the line containing the defect or coding rule violation.

   You can see your review comments as a code comment in the Polyspace annotation syntax, which Polyspace uses to repopulate review comments on your next analysis.
5. Save your source file and rerun the analysis.

   On the **Results List** pane, the software populates the **Severity**, **Status**, and **Comment** columns for the defect or rule violation that you annotated. These fields are read only because they are populated from your code annotation. If you use a specific keyword or status for your annotations, you can filter your results to hide or show your annotated results. For more information on filtering, see “Filter and Group Results” on page 16-2.

Add Annotations Manually

This method shows you how to enter comments directly into your source files by using the Polyspace code annotation syntax. The syntax is not case-sensitive and applies to the first uncommented line of C/C++ code following the annotation.

1. Open your source file in an editor and locate the line or section of code that you want to annotate.
2. Add one of the following annotations:

   - For a single line of code, add the following text directly before the line of code that you want to annotate.

     ```
     /* polyspace<Type:Kind1[,Kind2] : [Severity] : [Status] > [Notes] */
     ```
   - For a section of code, use the following syntax.

     ```
     /* polyspace:begin<Type:Kind1[,Kind2] : [Severity] : [Status] > [Notes] */

     ... Code section ...

     /* polyspace:end<Type:Kind1[,Kind2] : [Severity] : [Status] >  */
     ```

If a macro expands to multiple lines, use the syntax for code sections even though the macro itself covers one line. The single-line syntax applies only to results that appear in the first line of the expanded macro.

3. Replace the words *`Type`*, *`Kind1`*, *`[Kind2]`*, *`[Severity]`*, *`[Status]`*, and *`[Additional text]`* with allowed values, indicated in the following table. The text with square brackets *`[]`* is optional and you can delete it. See "Syntax Examples" on page 15-30.

Word	Allowed Values
`Type`	The type of results: • `Defect` (Polyspace Bug Finder) • `RTE`, for run-time checks (Polyspace Code Prover) • `VARIABLE`, for global variables (Polyspace Code Prover) • `MISRA-C`, for MISRA C:2004 • `MISRA-AC-AGC` • `MISRA-C3`, for MISRA C:2012 • `MISRA-CPP` • `JSF` • `Custom`, for custom coding rule violations.
`Kind1, [Kind2],...`	For defects and run-time checks, use the short names of checkers. See: • "Short Names of Bug Finder Defect Checkers" on page 15-12 • "Short Names of Code Prover Run-Time Checks" (Polyspace Code Prover) For coding rule violations, specify the rule number or numbers. For global variables, the only allowed value is `ALL`. If you want the comment to apply to all possible defects or coding rules, specify `ALL`.

Word	Allowed Values
Severity	Text that indicates how critical you consider the defect. Enter one of the following: • `Unset` • `High` • `Medium` • `Low` This text populates the **Severity** column on the **Results List** pane.
Status	Text that indicates how you intend to correct the error in your code. Enter one of the following or any other text: • `Unreviewed` • `To investigate` • `To fix` • `Justified` • `No action planned` • `Not a defect` • `Other` This text populates the **Status** column on the **Results List** pane. The status is also used in Polyspace Metrics to determine whether a result is justified. To justify a result, use `Justified`, `No action planned` or `Not a defect`.
Notes	Additional comments, such as a keyword or an explanation for the status and severity.

Syntax Examples

- A single defect:

```
/* polyspace<Defect:HARD_CODED_BUFFER_SIZE:Medium:To investigate> Known issue */
int table[100];
```

- A single run-time check:

```
/* polyspace<RTE: ZDV : High : To Fix > Denominator cannot be zero */
y=1/x;
```

- A MISRA C:2012 rule violation:

```
/* polyspace<MISRA-C3: 13.1 : Low : Justified> Known issue */
int arr[2] = {x++,y};
```

- Unused global variable:

```
/* polyspace<VARIABLE: ALL : Low : Justified> Variable to use later*/
int var_unused;
```

- Multiple defects:

```
polyspace<Defect:USELESS_WRITE,DEAD_CODE:Low:No Action Planned> OK issue
```

- Multiple JSF rule violations:

```
polyspace<JSF:9,13:Low:Justified> Known issue
```

Define Custom Annotation Format

This example shows how to create and edit an XML file to define an annotation format and map it to the Polyspace annotation syntax.

To get started, copy the following code to a text editor and save it on your machine as `annotations_description.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>

<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
             Group="example XML">


 <Expressions Search_For_Keywords="myKeyword"
              Separator_Result_Name="," >
    <!-- Define annotation format in this
    section by adding <Expression/> elements -->

   <Expression Mode="SAME_LINE"
               Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)"
               Rule_Identifier_Position="1"
               />

   <Expression Mode="GOTO_INCREMENT"
               Regex="myKeyword\s+(\+\d+\s)(\w+(\s*,\s*\w+)*)"
               Increment_Position="1"
          Rule_Identifier_Position="2"
               />

  <Expression Mode="BEGIN"
               Regex="myKeyword\s*(\w+(\s*,\s*\w+)*)\s*Block_on"
               Rule_Identifier_Position="1"
          Case_Insensitive="true"
               />

   <Expression Mode="END"
               Regex="myKeyword\s*(\w+(\s*,\s*\w+)*)\s*Block_off"
               Rule_Identifier_Position="1"
               />
   <Expression Mode="END_ALL"
               Regex="myKeyword\sBlock_off_all"
               />


   <Expression Mode="SAME_LINE"


Regex="myKeywords\s+(\w+(\s*,\s*\w+)*)
(\s*\[(\w+\s*)*([:]\s*(\w+\s*)+)*\])*(\s*-*\s*)*([^-]*)(\s*-*)*"
Rule_Identifier_Position="1"
```

```
Status_Position="4"
Severity_Position="6"
Comment_Position="8"
               />
<!  -- Put the regular expression on a single line instead of two line
when you copy it to a text editor -->

    <!-- SAME_LINE example with more complex regular expression.
        Matches the following annotations:
        //myKeywords 50 [my_status:my_severity] -Additional comment-
        //myKeywords 50 [my_status]
        //myKeywords 50 [:my_severity]
        //myKeywords 50 -Additional comment-
            -->

 </Expressions>

 <Mapping>
    <!-- Map your annotation syntax to the Polyspace annotation
    syntax by adding <Result_Name_Mapping /> elements in this section -->



<Result_Name_Mapping Rule_Identifier="100" Family="DEFECT" Result_Name="INT_ZERO_DIV"/>

<Result_Name_Mapping Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
<Result_Name_Mapping Rule_Identifier="51" Family="MISRA-C3" Result_Name="8.7"/>
<Result_Name_Mapping Rule_Identifier="ALL_MISRA" Family="MISRA-C3" Result_Name="*"/>
 </Mapping>
</Annotations>
```

The XML file consists of two parts:

- `<Expressions>...</Expressions>` where you define the format of your annotation syntax.
- `<Mapping>...</Mapping>` where you map your syntax to the Polyspace annotation syntax.

After you edit this file, Polyspace can interpret your custom code annotation when you invoke the option `-xml-annotations-description`.

Define Annotation Syntax Format

To define an annotation syntax in Polyspace, your syntax must follow a pattern that you can represent with a regular expression. See “Regular Expressions” (MATLAB). It is recommended that you include a keyword in the pattern of your annotation syntax to help identify it. In this example, the keyword is `myKeyword`. Set the attribute `Search_For_Keywords` equal to this keyword.

Once you know the pattern of your annotation, you can define it in the XML by adding an `<Expression/>` element and specifying at least the attributes `Mode`, `Regex`, and `Rule_Identifier_Position`. For instance, the first `<Expression/>` element in `annotations_description.xml` defines an annotation with these attributes:

- `Mode="SAME_LINE"`. The annotation applies to code on the same line.
- `Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)"`. Polyspace uses the regular expression to search for a string that begins with `myKeyword`, followed by a space `\s+`. Polyspace then searches for a capturing group `(\w+(\s*,\s*\w+)*)` that includes an alphanumeric rule identifier `\w+` and, optionally, additional comma-separated rule identifiers `(\s*,\s*\w+)*`.
- `Rule_Identifier_Position="1"`. The integer value of this attribute corresponds to the number of opening parentheses preceding the relevant capturing group in the regular expression. In `myKeyword\s+(\w+(\s*,\s*\w+)*)`, one opening parenthesis precedes the capturing group of the rule identifier `(\w+(\s*,\s*\w+)*)`. If you want to match rule identifiers captured by `(\s*,\s*\w+)`, then you set `Rule_Identifier_Position="2"` because two opening parentheses precede this capturing group.

The list of attributes and their values are listed in this table. The example column refers to the format defined in `annotations_description.xml`.

Attribute	Use	Value	Example
`Mode`	Required	`SAME_LINE`	Applies only on the same line as the annotation. `code; //myKeyword 100`

Attribute	Use	Value	Example

		`GOTO_INCREMENT`	Applies on the same line as the annotation and the following n lines:
			`3. code; // myKeyword +3 ALL_MISRA`
			`4. /*commments */`
			`5.`
			`6. code;`
			`7. code;`
			The preceding annotation applies to lines 3–6 only.

		`BEGIN`	Applies to the same line and all following lines until a corresponding expression with attribute `Mode="END"` or `"END_ALL"`, or until the end of the file.
			`//myKeyword 50, 51 Block_on`
			`Code block 1;`
			`...`

		`END`	Stops the application of a rule identifier declared by a corresponding expression with attribute `Mode="BEGIN"`.
			`//myKeyword 50, 51 Block_on`
			`Code block 1;`
			`...`
			`More code;`
			`//myKeyword 50 Block_off`
			Only rule identifier 50 is turned off. Rule identifier 51 still applies.

Attribute	Use	Value	Example

		`END_ALL`	Stops all rule identifiers declared by an expression with attribute `Mode="BEGIN"`.
			
			`//myKeyword 50, 51 Block_on`
			`Code block 1;`
			`...`
			`More code;`
			`//myKeyword Block_off_all`
			
			Rule identifiers 50 and 51 are turned off.

`Regex`	Required	Regular expression search string	See “Regular Expressions” (MATLAB). `Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)"` matches these expressions:
			
			`// myKeyword 50, 51`
			`/* myKeyword ALL_MISRA, 100 */`

Attribute	Use	Value	Example
Rule_Identifier_Position	Required, except when you set Mode="END_ALL"	Integer	The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression. <Expression Mode="GOTO_INCREMENT" Regex="myKeyword\s+(\+\d+\s) (\w+(\s*,\s*\w+)*)" Increment_Position="1" Rule_Identifier_Position="2"/> **Note** Enter the regex expression on a single line when you edit your XML file. The search expression for the rule identifier \w+(\s*,\s*\w+)* is after the second opening parenthesis of the regular expression.

Attribute	Use	Value	Example
Increment_Position	Required only when you set Mode="GOTO_INCREMENT"	Integer	The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression. <Expression Mode="GOTO_INCREMENT" Regex="myKeyword\s+(\+\d+\s) (\w+(\s*,\s*\w+)*)" Increment_Position="1" Rule_Identifier_Position="2"/> **Note** Enter the regex expression on a single line when you edit your XML file. The search expression for the increment \+\d+\s is after the first opening parenthesis of the regular expression.
Status_Position	Optional	Integer	See Increment_Position example. When you use this attribute, the entry in your annotation is displayed in the **Status** column on the **Results List** pane of the user interface.
Severity_Position	Optional	Integer	See Increment_Position example. When you use this attribute, the entry in your annotation is displayed in the **Severity** column on the **Results List** pane of the user interface.

Attribute	Use	Value	Example
`Comment_Position`	Optional	Integer	See `Increment_Position` example. When you use this attribute, the entry in your annotation is displayed in the **Comment** column on the **Results List** pane of the user interface. Your comment is appended to the string `Justified by annotation in source:`
`Case_Insensitive`	Optional	True or false	When you set this attribute to "true", the regular expression is case insensitive, otherwise it is case sensitive. If you do not declare this attribute in your expression, the regular expression is case sensitive. For `Case_Insensitive="true"`, these annotations are equivalent: `//MYKEYWORD ALL_MISRA BLOCK_ON` `//mykeyword all_misra block_on`

Map Your Annotation to the Polyspace Annotation Syntax

After you define your annotation format, you can map the rule identifiers you are using to their corresponding Polyspace annotation syntax. You can do this mapping by adding an `<Result_Name_Mapping/>` element and specifying attributes `Rule_Identifier`, `Family`, and `Result_Name`. For instance, if rule identifier 50 corresponds to MISRA C: 2012 rule 8.4, map it to the Polyspace syntax `MISRA-C3:8.4` by using this element:

```
<Result_Name_Mapping Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
```

The list of attributes and their values are listed in this table. The example column refers to the format defined in `annotations_description.xml`.

Attribute	Use	Value	Example
Rule_Identifier	Required	User defined	See the mapping section of annotations_description.xml
Family	Required	Corresponds to Polyspace results family. For a list of allowed values, see allowed values on page 15-6.	See the mapping section of annotations_description.xml
Result_Name	Required	Corresponds to Polyspace result names. For a list of allowed values, see allowed values on page 15-6.	See the mapping section of annotations_description.xml

See Also

"Annotation Description Full XML Template" on page 15-42 | -xml-annotations-description

More About

- "Annotate Code and Hide Known or Acceptable Results" on page 15-6

Annotation Description Full XML Template

This table lists all the elements, attributes, and values of the XML that you can use to define an annotation format and map it to the Polyspace annotation syntax. For an example of how to edit an XML to define annotation syntax, see “Define Custom Annotation Format” on page 15-32.

Element	Attribute	Use	Value
Annotations	Group	Required	User defined string. For example, "Custom Annotations"
Expressions	Search_For_Keywords	Required	User defined string. This string is a keyword you include in the pattern of your annotation syntax to help identify it. For example, "myKeyword"
	Separator_Result_Name	Required	User defined string. This string is a separator when you list multiple Polyspace result names in the same annotation. For example ","
	Separator_Family_And_Result_Name	Optional	User defined string. This string is a separator when you list multiple Polyspace results families in the same annotation. For example, " "

Element	Attribute	Use	Value
	Separator_Family	Optional	User defined string. This string is a separator when you list a Polyspace results family and results name in the same annotation. For example, ":"
Expression	Mode	Required	SAME_LINE
			GOTO_INCREMENT
			BEGIN
			END
			END_ALL
			NEXT_CODE_LINE The annotation applies to the next line of code. Comments and blank lines are ignored.
			GOTO_LABEL
			LABEL
			XML_START
			XML_CONTENT The annotation for this expression must be on a single line.
			XML_END
	Regex	Required	Regular expression search string that matches the pattern of your annotation.

Element	Attribute	Use	Value
	Rule_Identifier_Position	Required, except when you set Mode="END_ALL" or "LABEL"	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Increment_Position	Required only when you set Mode="GOTO_INCREMENT"	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Status_Position	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Severity_Position	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.

Element	Attribute	Use	Value
	`Comment_Position`	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	`Label_Position`	Required only when you set `Mode="GOTO_LABEL" or "LABEL"`	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	`Case_Insensitive`	Optional	True or false. When you do not declare this attribute, the default value is false.
	`Is_Pragma`	Optional	True or false. When you do not declare this attribute, the default value is false. Set this attribute to true if you want to declare your annotation using a pragma instead of a comment.

Element	Attribute	Use	Value
	Applies_Also_On_Same_Line	Optional	True or false. When you do not declare this attribute, the default value is true. Use this attribute to enable annotations with the old Polyspace syntax to apply on the same line of code.
Mapping	None	None	None
Result_Name_Mapping	Rule_Identifier	Required	User defined
	Family	Required	Corresponds to Polyspace results family. For a list of allowed values, see allowed values on page 15-6.
	Result_Name	Required	Corresponds to Polyspace result names. For a list of allowed values, see allowed values on page 15-6.

Example

This example code covers some of the less commonly used attributes for defining annotations in XML.

```
<?xml version="1.0" encoding="UTF-8"?>


<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
             Group="XML Template">

  <Expressions Separator_Result_Name=","
               Search_For_Keywords="myKeyword">

    <Expression Mode="GOTO_LABEL"
                Regex="(\A|\W)myKeyword\s+S\s+(\d+(\s*,\s*\d+)*)\s+([a-zA-Z_-]\w+)"
                Rule_Identifier_Position="2"
                Label_Position="4"

                />

     <Expression Mode="LABEL"
                Regex="(\A|\W)myKeyword\s+L:(\w+)"
                Label_Position="2"

                />
     <!-- Annotation applies starting current line until
        next declaration of label word "myLabel"
         Example:

         code; // myKeyword S 100 myLabel
         ...
         more code;
         // myKeyword L myLabel
    -->

    <Expression Mode="BEGIN"
                Regex="#\s*pragma\s+myKeyword_MESSAGES_ON\s+(\w+)"
                Rule_Identifier_Position="1"
                Is_Pragma="true"
                />
    <!-- Annotation declared with pragma instead of comment
        Example:#pragma myKeyword_MESSAGES_ON 100 -->


    <!-- Comment declaration with XML format-->

    <!-- XML_START must be declared before XML_CONTENT -->
```

```
    <Expression Mode="XML_START"
                Regex="<\s*myKeyword_COMMENT\s*>"

                />
    <!-- Example: <myKeyword_COMMENT>  -->

    <Expression Mode="XML_CONTENT"
                Regex="<\s*(\d*)\s*>(((?![*]/)(?!<).)*)</\s*(\d*)\s*>"
                Rule_Identifier_Position="1"
                Comment_Position="2"

                />
    <!-- Example: <100>This is my comment</100>
       XML_CONTENT must be declare on a single line.

       <100>This is my comment
       </100>
       is incorrect.
       -->

    <Expression Mode="XML_END"
                Regex="</\s*myKeyword_COMMENT\s*>"

                />
    <!--    Example: </myKeyword_COMMENT> -->
  </Expressions>

  <Mapping>

    <Result_Name_Mapping Rule_Identifier="100" Family="MISRA-C" Result_Name="4.1"/>
      </Mapping>
</Annotations>
```

See Also

-xml-annotations-description

More About

- “Annotate Code and Hide Known or Acceptable Results” on page 15-6

Import Comments from Previous Polyspace Analysis

After you have reviewed analysis results, you can reuse your review comments for subsequent analyses. If you add comments to your results file, they carry over to the next analysis on the same project. If you add comments to your code (annotate), they carry over to any subsequent analysis of the code, whether in the same project or not. You can also hide results using code annotations. For more information on commenting, see “Address Polyspace Results Through Bug Fixes or Comments” on page 15-2.

This topic shows how to import comments from one result file to another. By default, Polyspace imports comments from the most recent analysis on the module

After you import comments, on the **Results List** pane, clicking the icon skips justified checks. Using this icon, you can browse through unreviewed checks. You can also filter the justified checks from display.

Import Comments from Another Analysis Result

You can import comments directly from another Polyspace result to the current result.

If a result is found in both a Bug Finder and Code Prover analysis, you can comment on the Bug Finder result and import the comment to Code Prover. For instance, most coding rule checkers are common to Bug Finder and Code Prover. You can add comments to coding rule violations in Bug Finder and import the comments to the same violations in Code Prover.

To import comments from another set of results:

1. Open the current analysis results.
2. Select **Tools** > **Import Comments**.
3. Navigate to the folder containing your previous results.
4. Select the other results file (with extension `.psbf` or `.pscp`) and then click **Open**.

   The review comments from the previous results are imported into the current results.

View Imported Comments That Do Not Apply

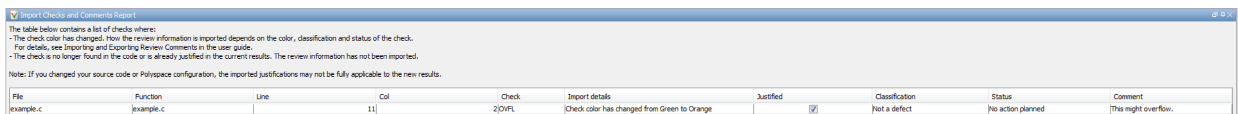
You can directly import review information from another set of results into the current results. However, it is possible that part of your review information are not imported to a subsequent analysis because:

- You have changed your source code so that the result is no longer present.
- You have changed your source code so that the Code Prover result color has changed.
- You have already entered different review comments for the same result.

The Import Checks and Comments Report highlights differences between two analysis results. When you import comments from a previous analysis, you can see this report. If you have closed the report after an import, to review the report again:

1. Select **Window** > **Show/Hide View** > **Import Comments Report**.

   The Import Checks and Comments Report opens, highlighting differences in the two results.



2. Review the differences between the two results.

Your review information can differ between two results because of the following reasons:

- In Code Prover, if the check color changes, Polyspace imports the **Comment** field but not the **Status** field. In addition, Polyspace imports the **Severity** and **Justified** fields depending on the color change.

Color Change	Severity	Justified
Orange or red to green	Not imported	Imported
Gray to green	Not imported	Imported, if the **Severity** was set to `High`, `Medium` or `Low`.
Red to orange or vice versa	Imported	Imported
Green to red/orange/gray	Not imported	Not imported

- If a result no longer appears in the code, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review comments from the previous result.
- If you have already entered different review comments for the same check, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review comments from the previous result.

Disable Automatic Comment Import from Last Analysis

By default, comments are imported automatically from the most recent analysis on the project module. You can disable this default behavior.

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**.

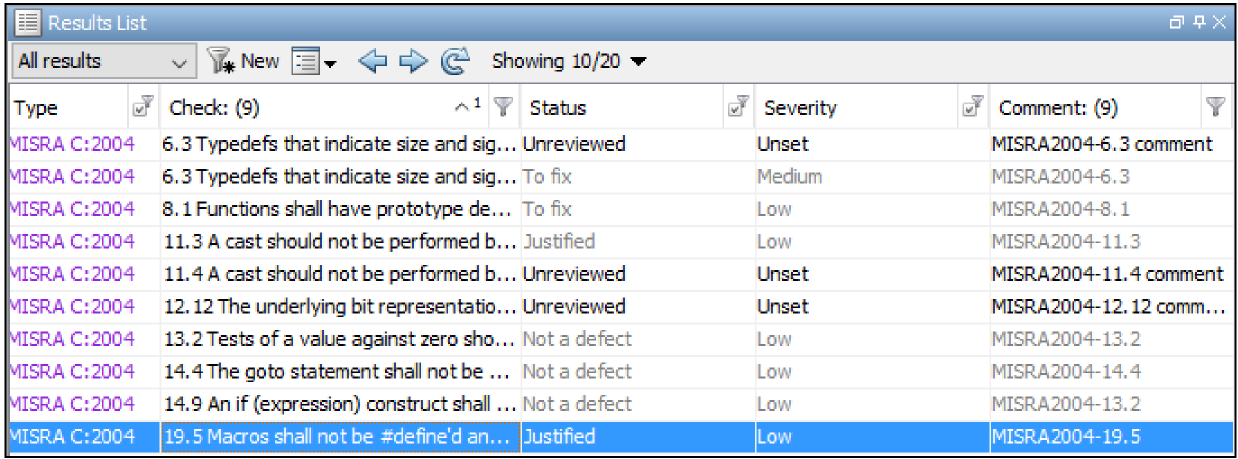
See Also

`-import-comments`

Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results

When you check your code for MISRA C: 2012 violations, Polyspace imports justifications of MISRA C: 2004 violations from previous analyses.

The software maps MISRA C: 2004 **Status**, **Severity**, and **Comment** values that you added through the user interface or code annotations to the corresponding MISRA C: 2012 results, if the results exist. For more information about mapping, consult addendum one of the MISRA C: 2012 publication.



If you are transitioning from MISRA C: 2004 to MISRA C: 2012, you do not have to review results that you have already justified.

Results List

All results | New | Showing 8/14

Type	Check	Status	Severity	Comment: (7)
MISRA C:2012	Dir 4.6 typedefs that indicate size and...	Unreviewed	Unset	MISRA2004-6.3 comment
MISRA C:2012	Dir 4.6 typedefs that indicate size and...	To fix	Medium	MISRA2004-6.3
MISRA C:2012	8.4 A compatible declaration shall be v...	To fix	Low	MISRA2004-8.1
MISRA C:2012	11.3 A cast shall not be performed bet...	Unreviewed	Unset	MISRA2004-11.4 comment
MISRA C:2012	11.4 A conversion should not be perfo...	Justified	Low	MISRA2004-11.3
MISRA C:2012	14.4 The controlling expression of an i...	Not a defect	Low	MISRA2004-13.2
MISRA C:2012	15.1 The goto statement should not b...	Not a defect	Low	MISRA2004-14.4
MISRA C:2012	15.6 The body of an iteration-stateme...	Not a defect	Low	MISRA2004-13.2

Mapping Multiple MISRA C: 2004 Annotations to the Same MISRA C: 2012 Result

When you justify MISRA C: 2004 violations by using code block syntax or multiple line annotation syntax, and multiple violations map to the same MISRA C: 2012 rule, Polyspace does not import each result justification. Instead, the software imports only one set of **Status**, **Severity**, and **Comment** values and applies these values to all the instances of that particular MISRA C: 2012 rule violation.

For example, suppose that you analyze your code and find violations of MISRA C: 2004 rules 16.3 and 16.5. You can justify these results by using the annotation syntax where you enter a different status and explanatory comment for each rule.

```
//polyspace-begin misra2004:16.3 [Status 1] "Explanatory comment 1"
//polyspace-begin misra2004:16.5 [Status 2] "Explanatory comment 2"

code block start;
/* This block of code contains violations of
MISRA C:2004 rules 16.3 and 16.5 */
code block end;

//polyspace-end misra2004:16.3
//polyspace-end misra2004:16.5
```

The previous violations map to MISRA C: 2012 rule 8.2. When you check your annotated code against MISRA C: 2012 rules, Polyspace imports only the first line of annotations (for rule 16.3) and applies it to all rule 8.2 results. The second line of annotations for rule 16.5 is ignored. In the **Results List** pane, all violations of rule 8.2 have the **Status** column set to **`Status 1`** and the **Comment** column set to **`"Explanatory comment 1"`**.

Note The **Output Summary** pane displays a warning message for every result where the imported annotation conflicts with the original annotation. After you import your MISRA C: 2004 annotations, check that a justified status has not been assigned to results you intend to investigate or fix.

See Also

`Check MISRA C:2004 (-misra2)` | `Check MISRA C:2012 (-misra3)`

More About

- "Annotate Code and Hide Known or Acceptable Results" on page 15-6

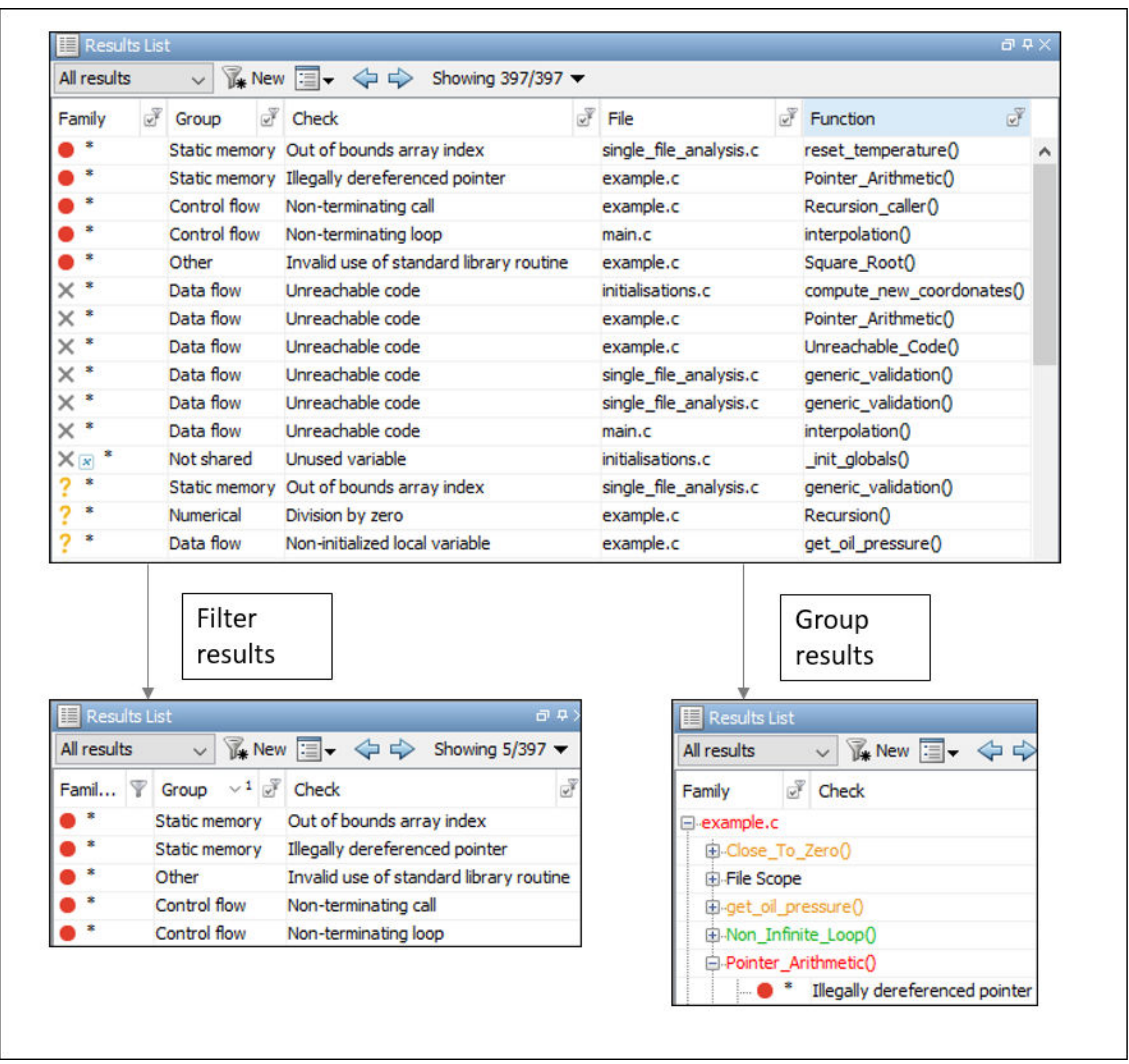
16

Manage Results

- "Filter and Group Results" on page 16-2
- "Classification of Defects by Impact" on page 16-11

Filter and Group Results

When you open the results of a Polyspace analysis, you see a flat list of defects (Bug Finder), run-time checks (Code Prover), coding rule violations or other results. To organize your review, you can narrow down the list or group results by file or result type.



Some of the ways you can use filtering are:

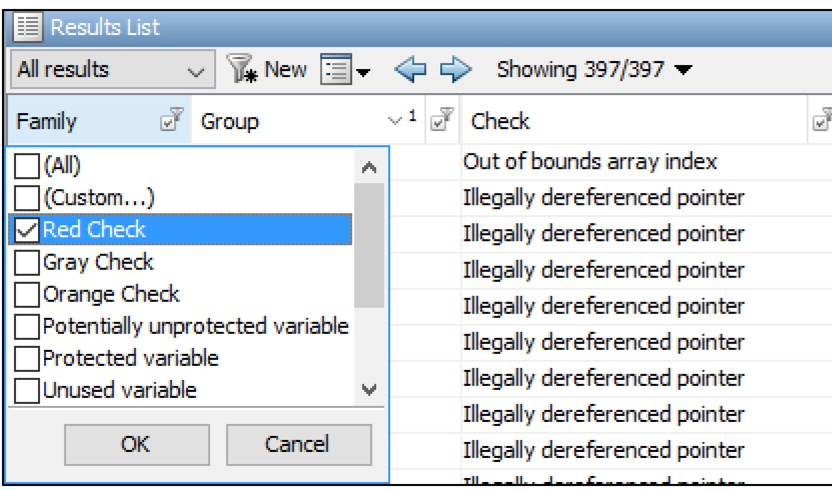
- You can display certain types of defects or run-time checks only.

  For instance, in Bug Finder, you can display only high-impact defects. See “Classification of Defects by Impact” on page 16-11.
- You can display only new results found since the last analysis.
- You can display only the results that have not justified.

  For information on justification, see “Address Polyspace Results Through Bug Fixes or Comments” on page 15-2.

Filter Results

Filter Using Results List



You can filter using the columns on the **Results List** pane. Click the icon on the column headers to see the available filters. For information on the columns, see:

- “Results List” on page 14-19
- “Results List” (Polyspace Code Prover)

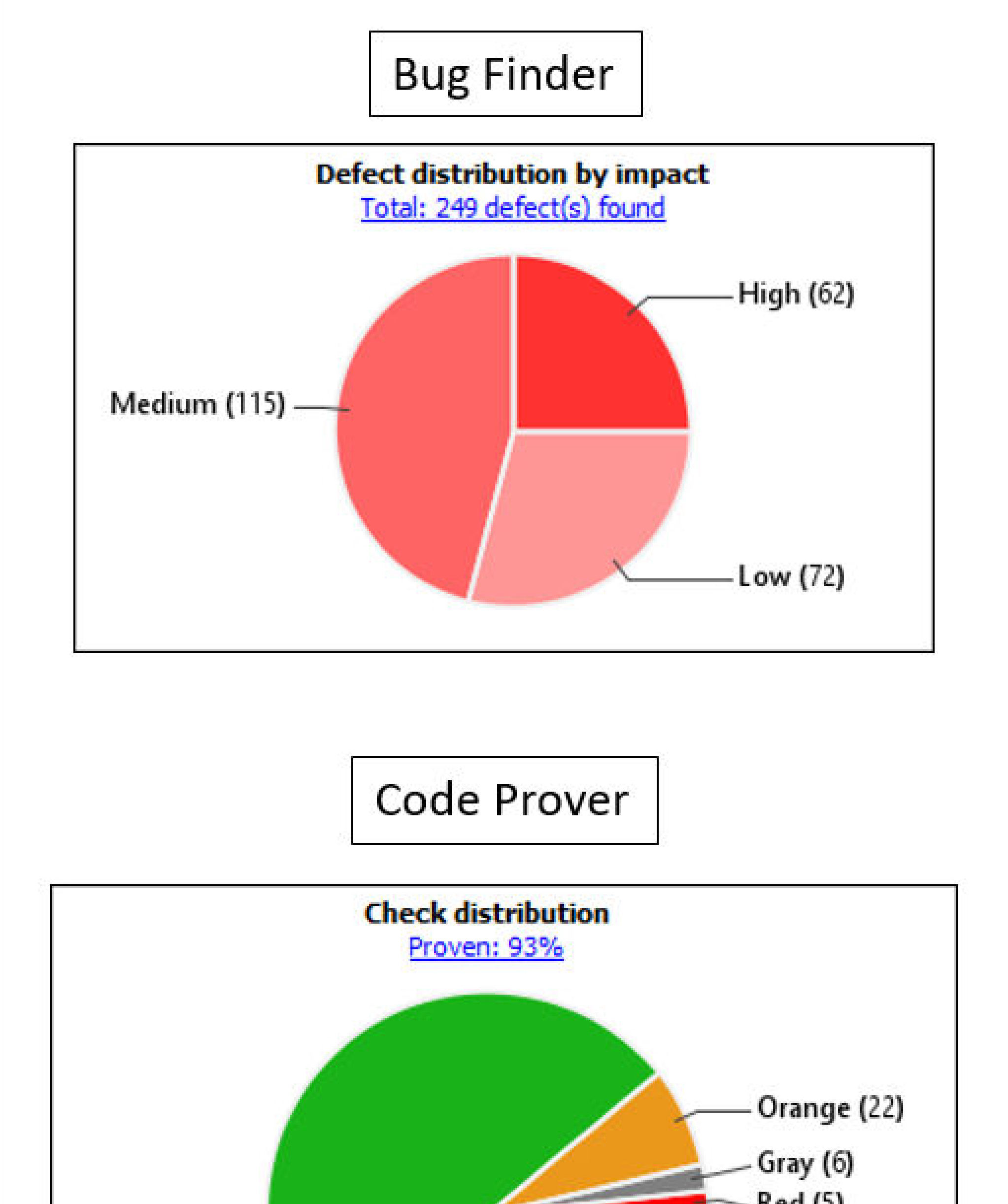
To see only results found since the last analysis, click the **New** button.

If you do not want to filter by the exact contents of a column, you can use a custom filter instead. For instance, you want to filter out subfolders of a specific folder. Instead of filtering out each subfolder in the **Folder** column, select **Custom** from the filter dropdown. Specify the root folder name for the `doesn’t contain` filter.

You can use wildcard characters for the custom filter. The wildcard `?` represents 0 or 1 character and `*` represents 0 or more characters.

If you apply filters in this way, they carry over to the next analysis. You can also name and save a subset of filters for use in multiple projects. To apply the named set of filters, pick this filter set from the **All results** list. To create a new entry in this list, select **Tools > Preferences** and create your own set of filters on the **Review Scope** tab.

Filter Using Dashboard


Bug Finder
Defect distribution by impact
Total: 249 defect(s) found
High (62)
Medium (115)
Low (72)
Code Prover
Check distribution
Proven: 93%
Orange (22)
Gray (6)
Red (5)
Green (269)

You can click graphs on the **Dashboard** pane to filter results. For instance:

- To see only high-impact defects in Bug Finder, click the corresponding section of the **Defect distribution by impact** chart.
- To see only red checks in Code Prover, click the corresponding section of the **Check distribution** chart.

To see all results again, click the link **View all results in this scope**.

Filter Using Orange Sources

An orange source can cause multiple orange checks in Code Prover. You can display all orange checks from the same source and review them together.

For instance, in this code, the unknown value `input` can cause an overflow and a division by zero. The variable `input` is an orange source that causes two orange checks.

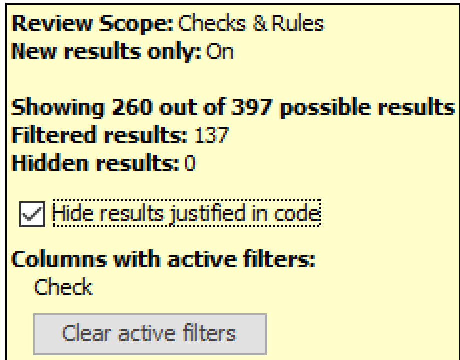
```
void func (int input) {
int val1;
double val2;
val1 = input++;
val2 = 1.0/input;
}
```

To begin, select **Window** > **Show/Hide View** > **Orange Sources**. You see the list of orange sources. Select an orange source to see all orange checks coming from this source.

Orange Sources

Source Type	Name	File	Line	Max Oranges
stubbed function	get_bus_status()		-1	1
stubbed function	random_float()		-1	3
stubbed function	random_int()		-1	1
local volatile variable	get_oil_pressure.vol_i	example.c	27	2
local volatile variable	all_values_s32.tmps32	single_file_analysis.c	29	2

See Filters Used



On the **Results List** header, you see the number of results displayed in the format **Showing *x*/*y***, for instance **Showing 100/250**. Click the dropdown beside this number to see the filters that are currently active. You can also clear the active filters from this dropdown (all except the named set of filters that you picked from the **All results** dropdown).

You see this information about the filters:

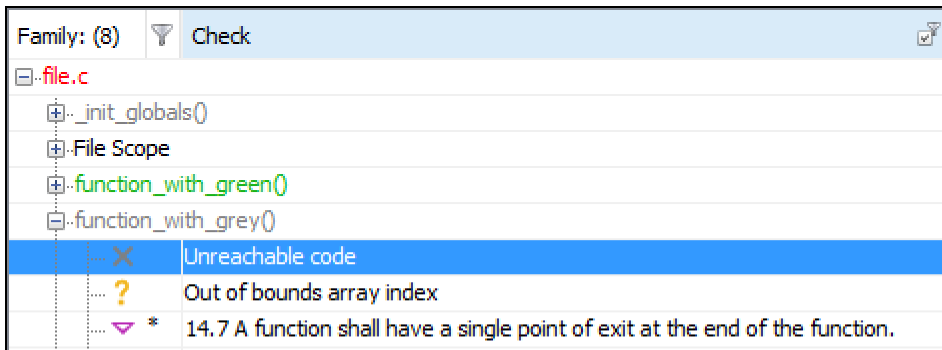
- **Review Scope**: If you pick a named set of filters from the **All results** dropdown, you see this filter set.
- **New results only**: If you use the **New** button to see only new results, you see this filter enabled.
- **Filtered results**: You see the number of results filtered in the Polyspace user interface (by any means: results list, dashboard or orange sources).
- **Hidden results**: You see the number of results hidden using code annotations. To unhide these results, clear **Hide results justified in code**.

  For information on hiding results through code annotations, see "Address Polyspace Results Through Bug Fixes or Comments" on page 15-2.

- **Columns with active filters**: You see the columns in the **Results List** pane (or columns corresponding to graphs in the **Dashboard** pane) that you used to filter results.

Group Results

On the **Results List** pane, from the list, select an option, for instance, grouping by file. Alternatively, you can click a column header to sort the column contents alphabetically.



The available options for grouping are:

- **None**: Shows results without grouping.
- **Family**: Shows results grouped by result type.

  The results are organized by type: checks (Code Prover), defects (Bug Finder), global variables (Code Prover), coding rule violations, code metrics. Within each type, they are grouped further.

  - The defects (Bug Finder) are organized by the defect groups. For more information on the groups, see “Defects”.

  - The checks (Code Prover) are grouped by color. Within each color, the checks are organized by check group. For more information on the groups, see “Run-Time Checks” (Polyspace Code Prover).
  - The global variables (Code Prover) are grouped by their usage. For more information, see “Global Variables” (Polyspace Code Prover).
  - 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”.
- **File**: Show results grouped 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**.

  In Code Prover, the file or function name shows the worst check color in the file or function. The severity of a check color decreases in the order: red, gray, orange, green.
- **Class** (for C++ code only): Shows results grouped by class.

  Within each class, the results are grouped by method. The results that are not associated with a particular class are grouped under **Global Scope**.

See Also

More About

- “Classification of Defects by Impact” on page 16-11

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. You can also select elements on the chart to filter out low or medium impact defects from the **Results List** pane. See "Filter and Group Results" on page 16-2.
- The **Results List** pane, in the **Information** column. When you select **None** from the list, 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. See "Filter and Group Results" on page 16-2.
- 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.

Concurrency

- Data race
- Data race through standard library function call
- Deadlock
- Double lock
- Double unlock
- Missing unlock

Data Flow

- Non-initialized pointer
- Non-initialized variable

Dynamic Memory

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

Numerical

- Absorption of float operand
- 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

Object Oriented

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

Programming

- Assertion
- Character value absorbed into EOF
- Declaration mismatch
- Errno not reset
- Invalid use of == (equality) operator
- Invalid use of standard library routine
- Invalid va_list argument
- Misuse of errno
- Misuse of return value from nonreentrant standard function
- Possible misuse of sizeof
- Possibly unintended evaluation of expression because of operator precedence rules
- Typedef mismatch
- Variable length array with nonpositive size
- Writing to const qualified object
- Wrong type used in sizeof

Resource Management

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

Security

- Bad order of dropping privileges
- Privilege drop not verified
- Returned value of a sensitive function not checked
- Unsafe call to a system function
- Use of non-secure temporary file

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`
- `Subtraction or comparison between pointers to different arrays`
- `Use of automatic variable as putenv-family function argument`
- `Use of path manipulation function without maximum sized buffer checking`
- `Wrong allocated object size for cast`

Medium Impact Defects

The following list shows the medium-impact defects.

Concurrency

- `Data race including atomic operations`
- `Destruction of locked mutex`
- `Missing lock`

Cryptography

- `Constant block cipher initialization vector`
- `Constant cipher key`
- `Context initialized incorrectly for cryptographic operation`
- `Context initialized incorrectly for digest operation`
- `Incompatible padding for RSA algorithm operation`

- Inconsistent cipher operations
- Incorrect key for cryptographic algorithm
- Missing blinding for RSA algorithm
- Missing block cipher initialization vector
- Missing cipher algorithm
- Missing cipher data to process
- Missing cipher final step
- Missing cipher key
- Missing data for encryption, decryption or signing operation
- Missing padding for RSA algorithm
- Missing parameters for key generation
- Missing peer key
- Missing private key
- Missing public key
- Nonsecure hash algorithm
- Nonsecure parameters for key generation
- Nonsecure RSA public exponent
- Nonsecure SSL/TLS protocol
- Predictable block cipher initialization vector
- Predictable cipher key
- Weak cipher algorithm
- Weak cipher mode
- Weak padding for RSA algorithm

Data Flow

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

Dynamic Memory

- Memory leak

Numerical

- Bitwise operation on negative value
- Integer overflow
- Sign change integer conversion overflow
- Use of plain char type for numerical value

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

Programming

- Abnormal termination of exit handler
- Bad file access mode or status
- Call through non-prototyped function pointer
- Copy of overlapping memory
- Environment pointer invalidated by previous operation
- Exception caught by value
- Exception handler hidden by previous handler
- Floating point comparison with equality operators
- Function called from signal handler not asynchronous-safe
- Function called from signal handler not asynchronous-safe (strict)
- Improper array initialization
- Incorrect data type passed to va_arg
- Incorrect pointer scaling
- Inline constraint not respected

- Invalid assumptions about memory organization
- Invalid file position
- Invalid use of = (assignment) operator
- Memory comparison of padding data
- Memory comparison of strings
- Missing byte reordering when transfering data
- Misuse of errno in a signal handler
- Misuse of sign-extended character value
- Shared data access within signal handler
- Signal call from within signal handler
- Standard function call with incorrect arguments
- Too many va_arg calls for current argument list
- Unsafe conversion between pointer and integer
- Use of indeterminate string
- Use of memset with size argument zero

Resource Management

- Opening previously opened resource

Security

- Deterministic random output from constant seed
- Errno not checked
- 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 descriptor exposure to child process
- 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

- Misuse of readlink()
- 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

Static Memory

- Unreliable cast of function pointer
- Unreliable cast of pointer

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

Low Impact Defects

The following list shows the low-impact defects.

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

Dynamic Memory

- Alignment changed after memory reallocation
- Mismatched alloc/dealloc functions on Windows
- Unprotected dynamic memory allocation

Good Practice

- Bitwise and arithmetic operation on a same data
- Delete of void pointer
- Hard coded buffer size
- Hard coded loop boundary
- Hard-coded object size used to manipulate memory
- Large pass-by-value argument
- Line with more than one statement
- Missing break of switch case
- Missing reset of a freed pointer
- Unused parameter
- Use of setjmp/longjmp

Numerical

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

Object Oriented

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

Programming

- Accessing object with temporary lifetime
- Alternating input and output from a stream without flush or positioning call
- Call to memset with unintended value
- Format string specifiers and arguments mismatch
- Memory comparison of float-point values
- Missing null in string array
- Misuse of a FILE object
- Misuse of structure with flexible array member
- Modification of internal buffer returned from nonreentrant standard function
- Overlapping assignment
- Predefined macro used as an object
- Preprocessor directive in macro argument
- Qualifier removed in conversion
- Return from computational exception signal handler
- Side effect of expression ignored
- Stream argument with possibly unintended side effects
- Universal character name from token concatenation
- Unsafe string to numeric value conversion

Security

- Function pointer assigned with absolute address
- Information leak via structure padding
- Missing case for switch condition
- Umask used with chmod-style arguments

- `Use of dangerous standard function`
- `Use of obsolete standard function`
- `Vulnerable path manipulation`

Static Memory

- `Arithmetic operation with NULL pointer`

Tainted Data

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

See Also

More About

- "Filter and Group Results" on page 16-2

Generate Reports

- "Generate Reports" on page 17-2
- "Export Polyspace Analysis Results" on page 17-6
- "Visualize Bug Finder Analysis Results in MATLAB" on page 17-10
- "Customize Existing Bug Finder Report Template" on page 17-15

Generate Reports

This example shows how to generate reports from Polyspace Bug Finder analysis results.

To generate reports, you can do one of the following:

- Run a Polyspace Bug Finder analysis and create a report from the analysis results. See the workflow described here.
- Specify that a report will be automatically generated after analysis. For more information on the options, see "Reporting".
- Export your results to a text file and generate graphs and statistics. See "Export Polyspace Analysis Results" on page 17-6.

Depending on the template you use, the report contains information about certain types of results from the **Results List** pane. You can see the following information about a result:

- ID: Unique number for a result for the current analysis

  To identify the result in your source code, you can use the ID in the **Results List** pane of the Polyspace user interface or in your IDE if you are using a Polyspace plugin.
- Check: Defect names, MISRA C:2012 coding rule number, and so on.
- File and function
- Status, Severity, Comment: Information that you enter about a result.

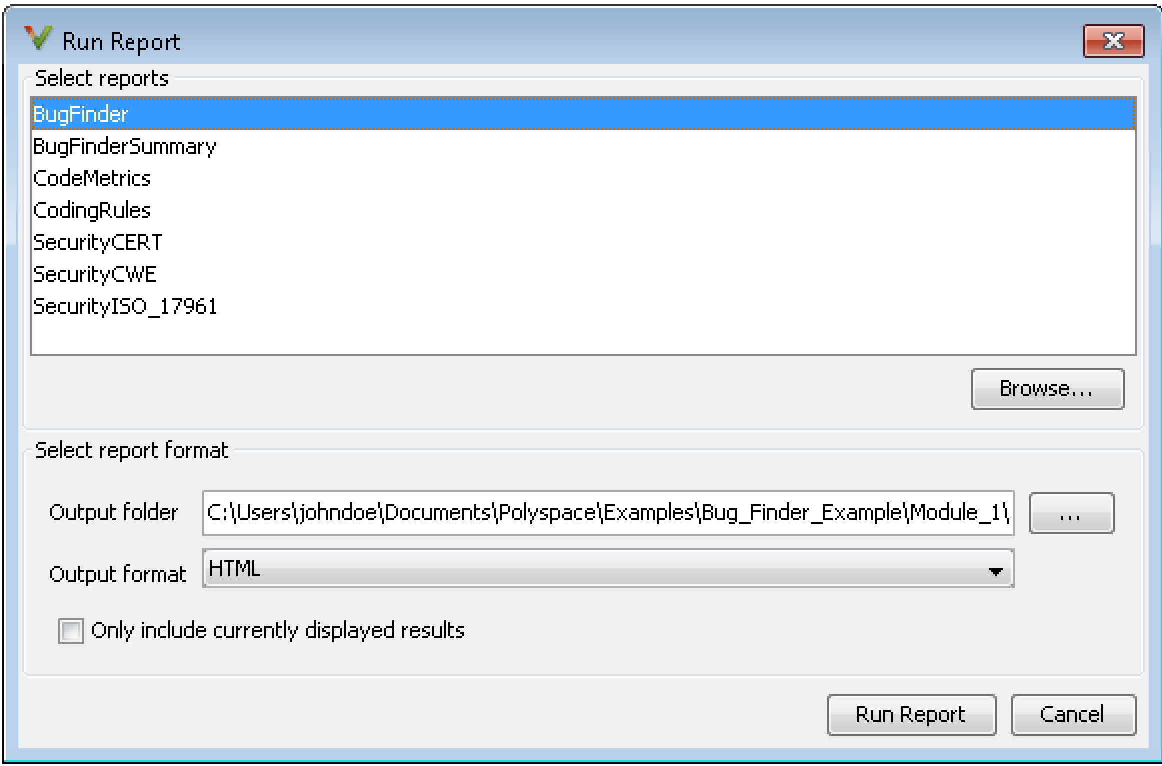
The report does not contain the line or column number for a result. Use the report for archiving, gathering statistics and checking whether results have been reviewed and addressed (for certification purposes or otherwise). To review a result in your source code, use the Polyspace user interface or your IDE if you are using a Polyspace plugin.

Generate Reports from User Interface

You can generate a report from your analysis results. Using a customizable template, the report presents your results in a concise manner for managerial review or other purposes.

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

   The Run Report dialog box opens.



3. Select the following options:

   - 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**.
   - Select the **Output folder** in which to save the report.
   - Select an **Output format** for the report.
   - If the display language (Windows) or locale (Linux) of your operating system is set to another language, you see an option to generate English reports. Select this option if you want an English report, otherwise the report is in another language.
   - If you want to filter results from your report, use filters on the **Results List** pane to display only the results that you want to report. Then, when generating reports, select **Only include currently displayed results**.

     For more information on filtering, see “Filter and Group Results” on page 16-2.

4 Click **Run Report**.

The software creates the specified report and opens it.

Generate Reports from Command Line

You can script the generation of reports using the `polyspace-report-generator` command.

Use the following options with the command:

- `-template` *`path`*: Path to report template file. For more information, see `Bug Finder and Code Prover report (-report-template)`.

  The predefined report templates are in *`matlabroot`*`\toolbox\polyspace\psrptgen\templates\Developer.rpt`. Here, *`matlabroot`* is the MATLAB installation folder such as `C:\Program Files\MATLAB\R2015a`.
- `-format` *`type`*: Output format of report. The allowed *`type`*s are`HTML`, `PDF` and `WORD`.
- `-output-name` *`filename`*: Name of report.
- `-results-dir` *`folder_paths`*: Path to folder containing your analysis results.

  To generate a single report for multiple analyses, specify *`folder_paths`* as follows:

  ```
  "folder1, folder2, ..., folderN"
  ```

  where *`folder1`*, *`folder2`*, `...` are paths to the folders that contain analysis results. For example,

  ```
  "C:\Recent_project\Results,C:\Old_project\Results"
  ```

  If you do not specify a folder path, the software uses analysis results from the current folder.
- `-set-language-english`: Use this option to generate English reports if the default report is in another language. The display language (Windows) or locale (Linux) of your operating system determines the default language in the report.

See Also

`Generate report` | `Bug Finder and Code Prover report (-report-template)` | `Output format (-report-output-format)`

More About

- “Customize Existing Bug Finder Report Template” on page 17-15
- “Export Polyspace Analysis Results” on page 17-6

Export Polyspace Analysis Results

You can export your analysis results to a tab delimited text file or a MATLAB table (MATLAB). Using the text file or table, you can:

- Generate graphs or statistics about your results that you cannot readily obtain from the user interface by using MATLAB or Microsoft Excel®. For instance, for each Code Prover check type (**Division by zero**, **Overflow**), you can calculate how many checks are red, orange, or green.
- Integrate the analysis results with other checks you perform on your code.

Export Results to Text File

You can export results from the user interface or command line.

<table>
<tr><th>User Interface</th><th>Command Line</th></tr>
<tr><td>
1. Open your analysis results.
2. Export all results or only a subset of the results.
• To export all results, select Reporting > Export > Export All Results.
• If you want to filter results from your report, use filters on the Results List pane to display only the results that you want to report. Then, when exporting results, select Reporting > Export > Export Currently Displayed Results.
For more information on filtering, see “Filter and Group Results” on page 16-2.
3. Select a location to save the text file and click OK.
</td><td>
Use appropriate options with the <code>polyspace-report-generator</code> command.
The available options are:
• <code>-generate-results-list-file</code>: Specifies that a text file must be generated.
• <code>-results-dir folder_paths</code>: Path to folder containing your analysis results. If you do not specify a folder path, the software uses analysis results from the current folder.
To generate text files for multiple analyses, specify <code>folder_paths</code> as follows:
<code>"folder1, folder2, ..., folderN"</code>
<code>folder1, folder2, ...</code> are paths to the folders that contain analysis results. For example:
<code>"C:\My_project\Module_1\results, C:\My_project\Module_2\Results"</code>
To merge the text files, use the <code>join</code> function.
• <code>-set-language-english</code>: Use this option to generate English reports if the default report is in another language. The display language (Windows) or locale (Linux) of your operating system determines the default language in the report.
</td></tr>
</table>

The exported text file uses the character encoding on your operating system. If special characters from your comments are not exported correctly in the text file, change the character encoding on your operating system before exporting.

Export Results to MATLAB Table

Instead of a text file, you can read your Polyspace analysis results into a MATLAB table. See:

- "Visualize Bug Finder Analysis Results in MATLAB" on page 17-10
- "Visualize Code Prover Analysis Results in MATLAB" (Polyspace Code Prover)

View Exported Results

The text file or the table contains the result information available on the **Results List** pane in the user interface (except for line and column information). See:

- "Results List" on page 14-19
- "Results List" (Polyspace Code Prover)

Some differences in presentation between the **Results List** pane and the text file are listed below.

- The text file has a **New** column that shows whether the result is new compared to the last analysis on the same code.
- The text file or the table also contains a **Key** column. The entry in this column is unique to a result across multiple analyses. When you merge multiple analysis results that might contain common files, use this entry to eliminate copies of a result. For instance, if you run coding-rule checking on multiple modules and merge the results, header files and coding rule violations in them appear in multiple module results. To eliminate copies of a coding rule violation, use the entry in the **Key** column.

You cannot identify the location of a Bug Finder result in your source code via the text file. However, you can still parse the file and generate graphs or statistics about your results.

See Also

Related Examples

- "Visualize Bug Finder Analysis Results in MATLAB" on page 17-10

Visualize Bug Finder Analysis Results in MATLAB

After a Polyspace analysis, you can read your results to a MATLAB table (MATLAB). Using the table, you can generate graphs or statistics about your results. If you have MATLAB Report Generator, you can include these tables and graphs in a PDF or HTML report.

Export Results to MATLAB Table

To read existing Polyspace analysis results into a MATLAB table, use a `polyspace.BugFinderResults` object associated with the results.

For instance, if the folder `C:\MyResults` contains results of a Polyspace analysis, enter the following:

```
resObj = polyspace.BugFinderResults('C:\MyResults')
resSummary = getSummary(resObj)
resTable = getResults(resObj)
```

`resSummary` and `resTable` are two MATLAB tables containing summary and details of the Polyspace results.

Alternatively, you can run a Polyspace analysis on C/C++ source files using a `polyspace.Project` object. After analysis, the `Results` property of the object contains the results. See “Run Polyspace Analysis by Using MATLAB Scripts” on page 3-2.

Generate Graphs from Results and Include in Report

After reading the results to a MATLAB table, you can visualize them in a convenient format. If you have MATLAB Report Generator, you can create a PDF or HTML report that contains your visualizations.

This example creates a pie chart showing the distribution of showing the distribution of defects by defect groups on page 13-3, and includes the chart in a report.

```
%% This example shows how to create a pie chart from your
% results and append it to a report.

%% Generate Pie Chart from Polyspace Results

% Copy a demo result set to a temporary folder.
```

```
resPath = fullfile(matlabroot,'polyspace','examples','cxx', ...
  'Bug_Finder_Example','Module_1','BF_Result');
userResPath = tempname;
copyfile(resPath,userResPath);

% Read results into a table.
resObj = polyspace.BugFinderResults(userResPath);
resTable = getResults(resObj);

% Eliminate results that are not defects.
matches = (resTable.Family == 'Defect');
defectTable = resTable(matches ,:);

% Create a pie chart showing distribution of defects.
defectGroupList = removecats(defectTable.Group);
pieDefects = pie(defectGroupList);
labels = get(pieDefects(2:2:end),'String');
set(pieDefects(2:2:end),'String','');
legend(labels,'Location','bestoutside')

% Save the pie chart.
print('file','-dpng');


%% Append Pie Chart to Report
% Requires MATLAB Report Generator

% Create a report object.
import mlreportgen.dom.*;
report = Document('PolyspaceReport','html');

% Add a heading and paragraph to the report.
append(report, Heading(1,'Bug Finder Run-Time Errors Graph'));
paragraphText = ['The following graph shows the distribution of ' ...
               'run-time errors in your code.'];
append(report, Paragraph(paragraphText));

% Add the image to the report.
chartObj = Image('file.png');
append(report, chartObj);

% Add another heading and paragraph to the report.
append(report, Heading(1,'Bug Finder Run-Time Errors Details'));
paragraphText = ['The following table shows the run-time errors ' ...
```

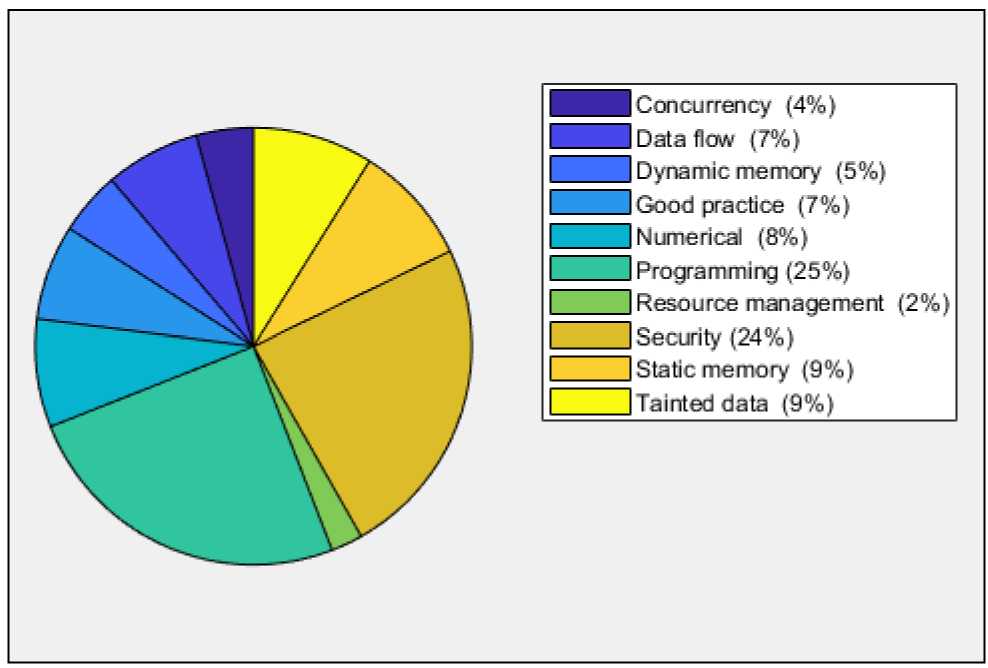
```
                'in your code.'];
append(report, Paragraph(paragraphText));

% Add the table of defects to the report.
reducedInfoTable = defectTable(:,{'File','Function','Check',...
    'Status','Severity','Comment'});
reducedInfoTable = sortrows(reducedInfoTable,[1 2]);
tableObj = MATLABTable(reducedInfoTable);
tableObj.Style = {Border('solid','black'),ColSep('solid','black'),...
    RowSep('solid','black')};
append(report, tableObj);

% Close and view the report in a browser.
close(report);
rptview(report.OutputPath);
```

The key functions used in the example are:

- `polyspace.BugFinderResults`: Read Bug Finder results into table (MATLAB).
- `pie`: Create pie chart from a categorical array (MATLAB). You can alternatively use the function `histogram` or `heatmap`.

  To create histograms, replace `pie` with `histogram` in the script and remove the pie chart legends.
- `mlreportgen.dom.Document`: Create a report object that specifies the report format and where to store the report.
- `mlreportgen.dom.Document.append`: Append contents to the existing report.

When you execute the script, you see a distribution of defects by defect group. The script also creates an HTML report that contains the graph and table of Polyspace defects.



You can use any criteria to remove rows from the results table before reporting. The preceding example uses the criteria that the result must be from the defect family. See also Bug Finder result families.

```
matches = (resTable.Family == 'Defect');
defectTable = resTable(matches ,:);
```

Instead, you can use another criteria. For instance, you can remove results in header files and retain the results from source files only.

```
sourceExtensions = [".c",".cpp",".cxx"];
fileNameStrings = string(resTable.File);
matches = endsWith(fileNameStrings,sourceExtensions);
sourceTable = resTable(matches ,:);
```

See Also

Related Examples

- “Export Polyspace Analysis Results” on page 17-6

Customize Existing Bug Finder Report Template

In this example, you learn how to customize an existing report template to suit your requirements. A report template allows you to generate a report from your analysis results in a specific format. If an existing report template does not suit your requirements, you can change certain aspects of the template.

For more information on the existing templates, see `Bug Finder and Code Prover report (-report-template)`.

Prerequisites

Before you customize a report template:

- See whether an existing report template meets your requirements. Identify the template that produces reports in a format close to what you need. You can adapt this template.

  To test a template, generate a report from sample results using the template. See “Generate Reports” on page 17-2.
- Make sure you have MATLAB Report Generator installed on your system.

In this example, you modify the **BugFinder** template that is available in Polyspace Bug Finder.

View Components of Template

A report template can be broken into components in MATLAB Report Generator. Each component represents some of the information that is included in a report generated using the template. For example, the component **Title Page** represents the information in the title page of the report.

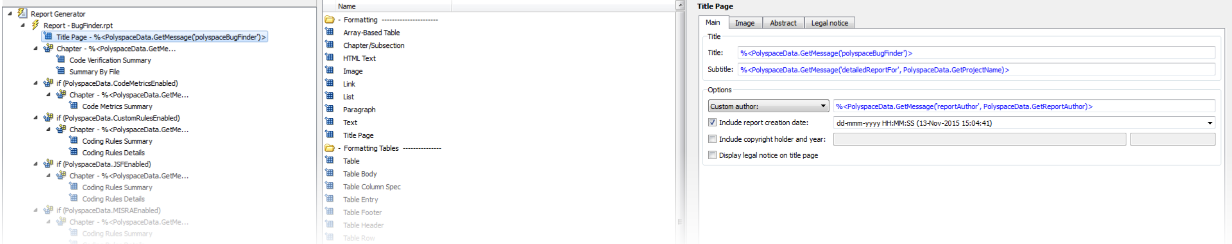
In this example, you view the components of the **BugFinder** template.

1. Open the Report Explorer interface of Simulink Report Generator. At the MATLAB command prompt, enter:

   ```
   report
   ```
2. Open the **BugFinder** template in the Report Explorer interface.

> The **BugFinder** template is in *`matlabroot`*`/toolbox/polyspace/psrptgen/templates/bug_finder` where *`matlabroot`* is the MATLAB installation folder. Run `matlabroot` in MATLAB to find the installation folder location.

Your template opens in the Report Explorer. On the left pane, you can see the components of the template. You can click each component and view the component properties on the right pane.



Some components of the **BugFinder** template and their purpose are described below.

Component	Purpose
`Title Page`	Inserts title page in the beginning of report
`Chapter/ Subsection`	Groups portions of report into sections with titles
`Code Verification Summary`	Inserts summary table of Polyspace analysis results
`Logical If`	Executes child components only if a condition is satisfied
`Run-time Checks Summary Ordered by File`	Inserts a table with Polyspace Bug Finder defects grouped by file

To understand how the template works, compare the components in the template with a report generated using the template.

For more information on the components, see “Update Reports Created Interactively” (Simulink Report Generator). For information on Polyspace-specific components, see “Generate Reports”.

Note Some of the component properties are set using internal expressions. Although you can view the expressions, do not change them. For instance, the conditions specified in the **Logical If** components in the **BugFinder** template are specified using internal expressions.

Change Components of Template

In the Report Explorer interface, you can:

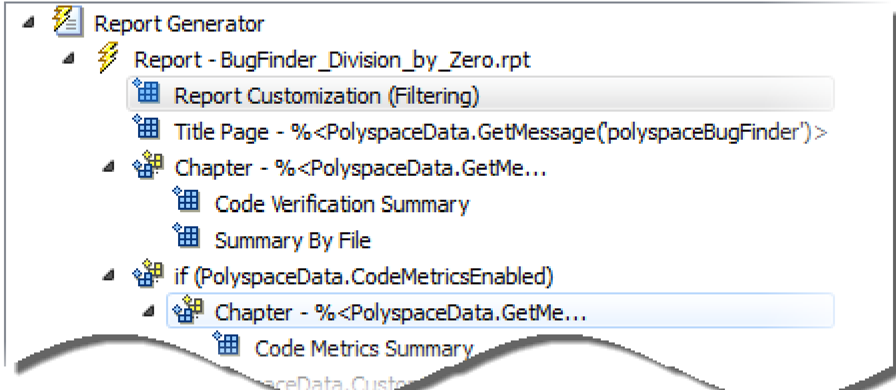
- Change properties of existing components of your template.
- Add new components to your template or remove existing components.

In this example, you add a component to the **BugFinder** template so that the template includes only **Integer division by zero** and **Float division by zero** defects in a report.

1. Open the **BugFinder** template in the Report Explorer interface and save it elsewhere with a different name, for instance, **BugFinder_Division_by_Zero**.
2. Add a new global component that filters every defect except division by zero from the **BugFinder_Division_by_Zero** template. The component is global because it applies to the full report and not one chapter of the report.

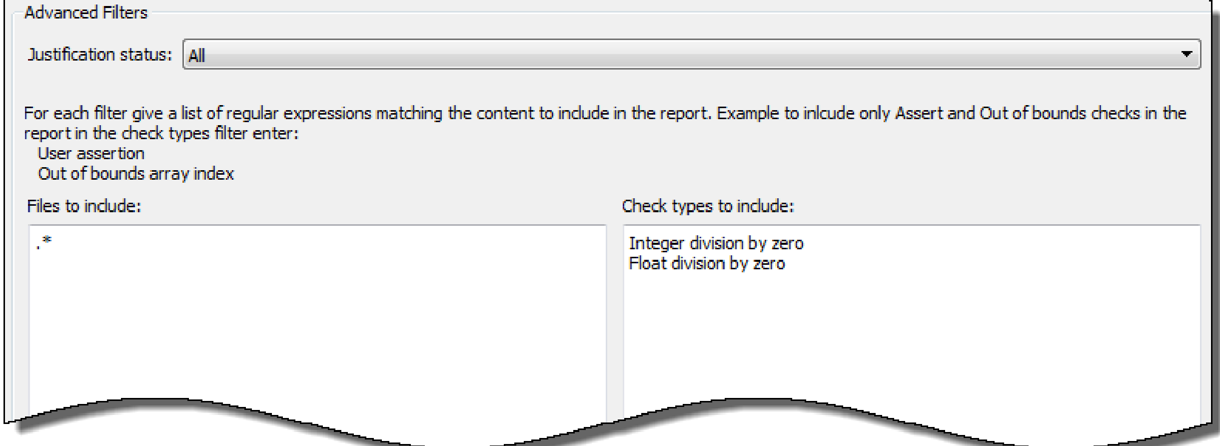
   To perform this action:

   a. Drag the component `Report Customization (Filtering)` from the middle pane and drop it above the **Title Page** component. The positioning of the component ensures that the filters apply to the full report and not one chapter of the report.



b Select the **Report Customization (Filtering)** component. On the right pane, you can set the properties of this component. By default, the properties are set such that all results are included in the report.

To include only **Integer division by zero** and **Float division by zero** defects, under the **Advanced Filters** group, enter `Integer division by zero` and `Float division by zero` in the **Check types to include** field.



You can also use MATLAB regular expressions in this field to exclude defects. See “Regular Expressions” (MATLAB).

You can toggle between activating and deactivating this component. Right-click the component and select **Activate/Deactivate Component**.

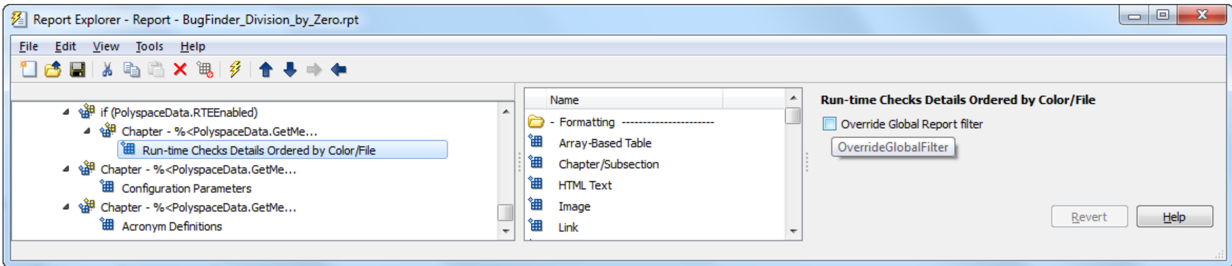
3. Change an existing chapter-specific component so that it does not override the global filter you applied in the previous step. If you prevent the overriding, the chapter-specific component follows the filtering specifications in the global component.

   To perform this action:

   a. On the left pane, select the `Run-time Checks Details Ordered by Color/File` component. This component produces tables in the report with details of run-time checks found in Polyspace Bug Finder.

      The right pane shows the properties of this component.

   b. Clear the **Override Global Report** filter box.



4. In the Polyspace user interface, create a report using both the **BugFinder** and **BugFinder_Division_by_Zero** template from results containing division by zero defects. Compare the two reports.

   For instance:

   a. Open **Help > Examples > Bug_Finder_Example.psprj**.

      The demo result contains **Integer division by zero** and **Float division by zero** defects.

   b. Create a PDF report using the **BugFinder** template. See “Generate Reports” on page 17-2.

      In the report, open **Chapter 4. Defects**. *You can see all defects from the example result.* Close the report.

c Create a PDF report using the **BugFinder_Division_by_Zero** template. In the Run Report window, use the **Browse** button to add the **BugFinder_Division_by_Zero** template to the existing template list.

In the report, open **Chapter 5. Defects**. *You see only **Integer division by zero** and **Float division by zero** defects.*

Note After you add the template to the existing list of templates, before generating the report, make sure to select the newly added template.

Software Quality with Polyspace Metrics

- “Upload Results to Polyspace Metrics” on page 18-2
- “View Projects in Polyspace Metrics” on page 18-5
- “Compare Metrics Against Software Quality Objectives” on page 18-13
- “Web Browser Requirements for Polyspace Metrics” on page 18-24
- “View Results List in Polyspace Metrics” on page 18-25

Upload Results to Polyspace Metrics

After analysis, you can upload results to the Polyspace Metrics web interface. The web interface displays a summary of your analysis results. You can share this summary with others even if they do not have Polyspace installed locally. You can also compare the results against previous analyses on the same project or measure them against predefined software quality objectives.

For more information, see "Polyspace Metrics Interface" on page 18-8.

Before you generate code quality metrics, set up Polyspace Metrics. See "Set Up Polyspace Metrics".

Manually Upload Results

To upload your results to the Polyspace Metrics web interface,

1. Select your results in the Project Browser pane.
2. Select **Metrics** > **Upload to Metrics**.
3. When you upload results to Polyspace Metrics, you are prompted to enter a password. Leave the field blank if you do not want to specify one.

   If you specify a password, you must enter it every time you open your project in Polyspace Metrics. The session lasts for 30 minutes even if you close and reopen your web browser. After 30 minutes, enter your password again.

   You can also specify a password later. On the Polyspace Metrics web interface, right-click your project and select **Change/Set Password**.

   Note The password for a Polyspace Metrics project is encrypted. The web data transfer is not encrypted. The password feature minimizes unintentional data corruption, but it does not provide data security. However, data transfers between a Polyspace local host and the remote analysis MJS host are always encrypted. To use a secure web data transfer with HTTPS, see "Configure Web Server for HTTPS" (Polyspace Code Prover).

Command Line: Use the command `polyspace-results-repository`. For a quick review of the command options, use the `-h` flag. At the command line, enter:

```
matlabroot\polyspace\bin\polyspace-results-repository -h
```

Here, *`matlabroot`* is the MATLAB installation folder, for instance, `C:\Program Files\MATLAB`.

Automatically Upload Results (Batch Analysis Only)

If you perform a remote analysis, you can specify for the results to be uploaded automatically to the web interface after analysis. Otherwise, upload the results after analysis manually.

1. On the **Configuration** pane, select **Run Settings**.
2. Along with **Run Bug Finder analysis on a remote cluster**, select **Upload results to Polyspace Metrics**.

   After analysis, the results are automatically uploaded to the web interface.
3. When you upload results to Polyspace Metrics, you are prompted to enter a password. Leave the field blank if you do not want to specify one.

   If you specify a password, you must enter it every time you open your project in Polyspace Metrics. The session lasts for 30 minutes even if you close and reopen your web browser. After 30 minutes, enter your password again.

   You can also specify a password later. On the Polyspace Metrics web interface, right-click your project and select **Change/Set Password**.

   Note The password for a Polyspace Metrics project is encrypted. The web data transfer is not encrypted. The password feature minimizes unintentional data corruption, but it does not provide data security. However, data transfers between a Polyspace local host and the remote analysis MJS host are always encrypted. To use a secure web data transfer with HTTPS, see “Configure Web Server for HTTPS” (Polyspace Code Prover).

Command Line: Use the option `Upload results to Polyspace metrics (-add-to-results-repository)`.

See Also

`polyspace-results-repository`

Related Examples

- "View Projects in Polyspace Metrics" on page 18-5
- "Compare Metrics Against Software Quality Objectives" on page 18-13

View Projects in Polyspace Metrics

Polyspace Metrics is a web dashboard that displays code quality metrics from your analysis results. Using this dashboard, you can:

- Track improvements or regression in code quality over time.
- Provide managers a high-level overview of your code quality.
- Compare your code against quality objectives.
- Narrow your analysis review to critical results.

Upload Results

Before you can review your project in Polyspace Metrics, you must “Upload Results to Polyspace Metrics” on page 18-2.

Open Metrics Interface

You can open the metrics interface in one of the following ways:

- If you have Polyspace installed, select **Metrics > Open Metrics**.
- If you do not have Polyspace installed, open a web browser and enter the following URL:

  `protocol:// ServerName: PortNumber`

  - `protocol` is either `http` (default) or `https`.

    To use HTTPS, additional configuration is required. See “Configure Web Server for HTTPS” (Polyspace Code Prover).
  - `ServerName` is the name or IP address of your Polyspace Metrics server.
  - `PortNumber` is the web server port number (default 8080).

When the Polyspace Metrics web interface opens you are presented with a list of results in your repository. You can view these results by project or by run.

The **Projects** tab lists the uploaded results by projects. On this tab, you can:

- See the number of project runs and overall statistics about the project by hovering your cursor over the project name.

- See project-level metrics by right-clicking the column headers and adding additional columns: Bug Finder Checks, Coding Rules, Code Metrics, Run-Time Errors, or Review Progress.
- Create project groups by right-clicking a project and selecting **Create Project Category**. Drag projects to your new category.
- Filter projects using the column headers.
- Delete projects from the Metrics repository by right-clicking the project and selecting **Delete Project from Repository**.
- Assign or change the password for a project by right-clicking the project and selecting **Change/Set Password**.
- Review into code quality metrics for a project by clicking the project. For details, see “Polyspace Metrics Interface” on page 18-8.

The **Runs** tab lists the individual runs for allprojects. On this tab, you can:

- Delete a run from the repository by right-clicking the run and selecting **Delete Run from Repository**.
- Assign password to run by right-clicking the run and selecting **Change/Set Password**.
- See runs between two specific dates by selecting the starting date in the **From** field and the end date in the **To** field.
- See only the last *n* runs by changing the value of the **Maximum number of runs** field.
- See code quality metrics for a run by right-clicking the run and selecting **Go to Metrics Page**.
- Download results of run to Polyspace user interface by clicking the run name.

Review Metrics

For each project or analysis, you can view the code quality metrics spread over four tabs, at project, file, and function level. Select a project and you see four tabs:

- The **Summary** tab on page 18-9 provides a high-level overview of the verification results.
- The **Code Metrics** tab on page 18-10 provides the details of the code complexity metrics in your results.
- The **Coding Rules** tab on page 18-10 provides the details of the coding rule violations in your results.

- The **Bug Finder** tab on page 18-11 provides details of code defects in your results.

If you want to "Compare Metrics Against Software Quality Objectives" on page 18-13, before reviewing your results, you can turn on quality objectives.

1. Click an entry on the **Summary** tab. Clicking on an entry brings you to the respective tab for more details.
2. On the **Code Metrics**, **Coding Rules** or **Run-Time Errors** tabs, select an entry to download the result to the Polyspace user interface.

   The results appear on the **Results List** pane in the Polyspace user interface. The filter **Show** > **Web checks** on this pane indicate that you have downloaded the results from Polyspace Metrics.
3. In the Polyspace user interface, review the particular result, investigate the root cause in your source code, and assign review comments and justifications.
4. To upload your comments and justifications to the Polyspace Metrics repository, select **Metrics** > **Upload to Metrics**.

   Tip To upload automatically your comments and justifications to the Polyspace Metrics repository when you save them:

   a. Select **Tools** > **Preferences**.
   b. On the **Server Configuration** tab, select **Save justifications in the Polyspace Metrics repository**.

   ---
5. In the Polyspace Metrics interface, click Refresh to view your newly uploaded metrics.

Compare Metrics Between Results

Using the Polyspace Metrics interface, you can track improvements or regression in code quality metrics over various runs on the same source code.

To view trends in metrics, upload the various versions of your results to the Polyspace Metrics repository.

1. Open the Polyspace Metrics interface.

   For more information, see "Open Metrics Interface" on page 18-5.

2 On the **Projects** tab, select the project for which you want to view trends.

   The code quality metrics for all versions of the project appear on the **Summary**, **Code Metrics**, **Coding Rules**, and **Bug-Finder** tabs.

3 To compare two versions of the same project:

   a In the **From** and **To** lists on the upper left of the web dashboard, select the two versions that you want to compare.

   b Select the **Compare** box.

   On each tab, new columns appear and existing columns display improvement or regression in a metric. For example, in the figure below, you see a new **All Metrics Trend** column that appears on the **Summary** tab. This column describes how the metrics in the **Bug-Finder** group compare over two versions of a project.

   - A means that the metrics is better.
   - A means that the metric is worse.
   - A mixed in the **All Metrics Trend** column means some metrics improved and some did not improve.

4 To see only the new findings in a version compared to a previous version:

   a In the **From** and **To** lists on the upper left of the web dashboard, select the two versions that you want to compare.

   b Select the **New Findings Only** box.

   The existing columns display only the new findings. In addition, you also see two new columns:

   - The **Newly Confirmed** column shows those new findings to which you assign a **Severity** of `High`, `Medium`, or `Low` in the Polyspace user interface.
   - The **Newly Fixed** column shows those findings to which you had assigned a **Severity** of `High`, `Medium`, or `Low` in the previous run. However, the assignment does not exist in the current run, either because a red or orange check turned green, or because you changed the **Severity** to `Unset`.

Polyspace Metrics Interface

- "Summary Tab" on page 18-9

- “Code Metrics Tab” on page 18-10
- “Coding Rules Tab” on page 18-10
- “Bug-Finder Tab” on page 18-11

If you turn on Software Quality Objectives, each tab also specifies how your project or run compares against those objectives. See “Compare Metrics Against Software Quality Objectives” on page 18-13.

Summary Tab

The **Summary** tab summarizes the analysis results for a project or run.

Column Name		Description
Verification		**Version number of the results and the source files.**
Verification Status		Analysis level completed.
Code Metrics	**Files**	Number of files in project.
	Lines of code	Number of lines of code, broken down by file.
Coding Rules	**Confirmed Defects**	Number of coding rule violations assigned a **Severity** of `High`, `Medium`, or `Low` in the Polyspace user interface.
	Violations	Total number of coding rule violations.
Bug-Finder Checks	**Confirmed Defects**	Number of defects assigned a **Severity** of `High`, `Medium`, or `Low` in the Polyspace user interface.
	Checks	Total number of defects.
Software Quality Objectives	**Overall Status**	A status of **PASS** or **FAIL** based on whether your code satisfies the software quality objectives you specified.
	Level	The software quality objectives that you specify. You can either use a predefined set of objectives or specify your own objectives.
	Review Progress	Number or percent of justified results. To justify a result, you must assign a **Status** in the Polyspace user interface.

Column Name		Description
Verification		**Version number of the results and the source files.**
	Justified Code Metrics	Number or percent of code metric threshold violations that you have justified. To justify a result, you must assign a **Status** in the Polyspace user interface.
	Justified Coding Rules	Number or percent of coding rule violations that you have justified. To justify a result, you must assign a **Status** in the Polyspace user interface.
	Justified Bug-Finder Checks	Number or percent of defects that you have justified. To justify a result, you must assign a **Status** in the Polyspace user interface.

Code Metrics Tab

The **Code Metrics** tab lists the code complexity metrics for your project or run.

Some metrics are calculated at the project level, while others are calculated at file or function level. For metrics calculated at the function level, the entry displayed for a file is either an aggregate or a maximum over the functions in the file.

For more information, see “Code Metrics”.

Coding Rules Tab

The **Coding Rules** tab lists the coding rule violations in your project or run. For more information on the coding rules, see “Coding Rules”.

You can group the information in the columns by **Files** or **Coding Rules**.

Column Name		Description
Coding Rules	**Confirmed Defects**	Number of coding rule violations assigned a **Severity** of `High`, `Medium`, or `Low` in the Polyspace user interface.

Column Name		Description
	Justified	Number of coding rule violations that you have justified. To justify a result, you must assign a **Status** in the Polyspace user interface.
	Violations	Total number of coding rule violations.
Software Quality Objectives	**Quality Status**	A status of **PASS** or **FAIL** based on whether your code satisfies the software quality objectives you specified.
	Level	The software quality objectives that you specify. You can either use a predefined set of objectives, or specify your own objectives.
	Review Progress	Number or percent of justified coding rule violations. To justify a result, you must assign a **Status** in the Polyspace user interface.

Bug-Finder Tab

The **Bug-Finder** tab lists the “Defects” in your project or run.

You can group the information in the columns by **Files** or **Bug-Finder Checkers**.

Column Name		Description
Confirmed Defects		Number or percent of defects assigned a **Severity** of `High`, `Medium`, or `Low` in the Polyspace user interface. See “Address Polyspace Results Through Bug Fixes or Comments” on page 15-2.
Bug-Finder Checks	**Justified**	Number or percent of justified defects. To justify a result, you must assign a **Status** in the Polyspace user interface.
	Checks	Total number of checks.
	High	Total number of “High Impact Defects” on page 16-11.
	Medium	Total number of “Medium Impact Defects” on page 16-14.
	Low	Total number of “Low Impact Defects” on page 16-18.

Column Name		Description
Software Quality Objectives	**Quality Status**	A status of **PASS** or **FAIL** based on whether your code satisfies the software quality objectives you specified.
	Level	The software quality objectives that you specify. You can either use a predefined set of objectives or specify your own objectives.
	Review Progress	Number or percent of justified defects. To justify a result, you must assign a **Status** in the Polyspace user interface.

See Also

Related Examples

- "Address Polyspace Results Through Bug Fixes or Comments" on page 15-2
- "Upload Results to Polyspace Metrics" on page 18-2
- "Compare Metrics Against Software Quality Objectives" on page 18-13
- "Compare Metrics Between Results" on page 18-7

Compare Metrics Against Software Quality Objectives

After generating and viewing metrics from your analysis results, you can review the results in greater detail.

To focus your review, you can:

1. Define quality objectives that you or developers in your organization must meet.
2. Apply the quality objectives to your analysis results.
3. Review only those results that fail to meet those objectives.

Apply Predefined Objectives to Metrics

By default, the software quality objectives are turned off. To apply quality objectives:

1. In the Polyspace Metrics interface, open the metrics page for a project.
2. From the **Quality Objectives** list in the upper left, select `ON`.

   A new group of **Software Quality Objectives** columns appears.

   - The **Overall Status** column shows the last used quality objective level to generate a status of **PASS** or **FAIL** for your results.
   - The **Level** column shows the quality objective level.

     To change your quality objective level, in this column, select a cell. From the drop-down list, select a quality level. For more information, see "Bug Finder Quality Objective Levels" on page 18-14.
3. For files with an **Overall Status** of **FAIL**, to see what causes the failure, view the entries in the other **Software Quality Objectives** columns. The failing levels are marked red.

   If the ⚠ icon appears next to the status, it means that Polyspace does not have enough information to compute the status. For instance, if you specify `BF-QO-1`, certain coding rules must be review. But, if you do not check coding rules during the analysis, Polyspace cannot determine whether your project satisfies the coding rule objectives specified in `BF-QO-1`.
4. To investigate the failing quality objectives, select the entries marked red for more details.

5 On the **Code Metrics**, **Coding Rules**, or **Bug-Finder** tab,

   a Select the red column entries to download the results.

   b Review the violations and fix or justify the results.

   See “Address Polyspace Results Through Bug Fixes or Comments” on page 15-2.

   c Upload your new justifications to the Polyspace Metrics web dashboard.

6 After your review, in the Polyspace Metrics interface, click Refresh to view the updated metrics.

   If you change your code, to update the metrics, rerun your analysis and upload the results to the Polyspace Metrics repository. If you have justifications in your previous results, import them to the new results before uploading to the repository.

Bug Finder Quality Objective Levels

The Bug Finder Quality Objectives or BF-QOs are a set of thresholds against which you can compare your Bug Finder analysis results. These objectives are adapted from the Polyspace Code Prover “Software Quality Objectives” (Polyspace Code Prover). You can develop a review process based on the Quality Objectives.

You can use a predefined BF-QO level or define your own. Following are the predefined quality thresholds specified by each BF-QO.

BF-QO Level 1

Metric	Threshold Value
Comment density of a file	20
Number of paths through a function	80
Number of `goto` statements	0
Cyclomatic complexity	10
Number of calling functions	5
Number of calls	7
Number of parameters per function	5
Number of instructions per function	50

<table>
<tr><th>Metric</th><th>Threshold Value</th></tr>
<tr><td>Number of call levels in a function</td><td>4</td></tr>
<tr><td>Number of <code>return</code> statements in a function</td><td>1</td></tr>
<tr><td>Language scope, an indicator of the cost of maintaining or changing functions. Calculated as follows:

<code>(N1+N2) / (n1+n2)</code>

• n1 — Number of different operators
• N1 — Total number of operators
• n2 — Number of different operands
• N2 — Total number of operands</td><td>4</td></tr>
<tr><td>Number of recursions</td><td>0</td></tr>
<tr><td>Number of direct recursions</td><td>0</td></tr>
<tr><td>Number of unjustified violations of the following MISRA C:2004 rules:

• 5.2
• 8.11, 8.12
• 11.2, 11.3
• 12.12
• 13.3, 13.4, 13.5
• 14.4, 14.7
• 16.1, 16.2, 16.7
• 17.3, 17.4, 17.5, 17.6
• 18.4
• 20.4</td><td>0</td></tr>
</table>

Metric	Threshold Value
Number of unjustified violations of the following MISRA C:2012 rules: • 8.8, 8.11, and 8.13 • 11.1, 11.2, 11.4, 11.5, 11.6, and 11.7 • 14.1 and 14.2 • 15.1, 15.2, 15.3, and 15.5 • 17.1 and 17.2 • 18.3, 18.4, 18.5, and 18.6 • 19.2 • 21.3	0
Number of unjustified violations of the following MISRA C++ rules: • 2-10-2 • 3-1-3, 3-3-2, 3-9-3 • 5-0-15, 5-0-18, 5-0-19, 5-2-8, 5-2-9 • 6-2-2, 6-5-1, 6-5-2, 6-5-3, 6-5-4, 6-6-1, 6-6-2, 6-6-4, 6-6-5 • 7-5-1, 7-5-2, 7-5-4 • 8-4-1 • 9-5-1 • 10-1-2, 10-1-3, 10-3-1, 10-3-2, 10-3-3 • 15-0-3, 15-1-3, 15-3-3, 15-3-5, 15-3-6, 15-3-7, 15-4-1, 15-5-1, 15-5-2 • 18-4-1	0

BF-QO Level 2 and 3

In addition to all the requirements of BF-QO Level 1, these levels includes the following thresholds:

Metric	Threshold Value
Number of "High Impact Defects" on page 16-11	0

BF-QO Level 4

In addition to all the requirements of BF-QO Level 2 and 3, this level includes the following thresholds:

Metric	Threshold Value
Number of “Medium Impact Defects” on page 16-14	0

BF-QO Level 5

In addition to all the requirements of BF-QO Level 4, this level includes the following thresholds:

Metric	Threshold Value
Number of unjustified violations of the following MISRA C:2004 rules: • 6.3 • 8.7 • 9.2, 9.3 • 10.3, 10.5 • 11.1, 11.5 • 12.1, 12.2, 12.5, 12.6, 12.9, 12.10 • 13.1, 13.2, 13.6 • 14.8, 14.10 • 15.3 • 16.3, 16.8, 16.9 • 19.4, 19.9, 19.10, 19.11, 19.12 • 20.3	0

Metric	Threshold Value
Number of unjustified violations of the following MISRA C:2012 rules: • 11.8 • 12.1 and 12.3 • 13.2 and 13.4 • 14.4 • 15.6 and 15.7 • 16.4 and 16.5 • 17.4 • 20.4, 20.6, 20.7, 20.9, and 20.11	0
Number of unjustified violations of the following MISRA C++ rules: • 3-4-1, 3-9-2 • 4-5-1 • 5-0-1, 5-0-2, 5-0-7, 5-0-8, 5-0-9, 5-0-10, 5-0-13, 5-2-1, 5-2-2, 5-2-7, 5-2-11, 5-3-3, 5-2-5, 5-2-6, 5-3-2, 5-18-1 • 6-2-1, 6-3-1, 6-4-2, 6-4-6, 6-5-3 • 8-4-3, 8-4-4, 8-5-2, 8-5-3 • 11-0-1 • 12-1-1, 12-8-2 • 16-0-5, 16-0-6, 16-0-7, 16-2-2, 16-3-1	0

BF-QO Level 6

In addition to all the requirements of BF-QO Level 5, this level includes the following thresholds:

Metric	Threshold Value
Number of "Low Impact Defects" on page 16-18	0

BF-QO Exhaustive

In addition to all the requirements of BF-QO Level 1, this level includes the following thresholds. The thresholds for coding rule violations apply only if you check for coding rule violations.

Metric	Threshold Value
Number of unjustified MISRA C and MISRA C++ coding rule violations	0
Number of unjustified defects	0

Customize Software Quality Objectives

Instead of using a predefined objective, you can define your own quality objectives and apply them to your project.

1 Save the following content in an XML file. Name the file `Custom-BF-QO-Definitions.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<MetricsDefinitions>

    <SQO ID="Custom-BF-QO-Level" ApplicableProduct="Bug Finder"
                                 ApplicableProject="My_Project">
        <comf>20</comf>
        <path>80</path>
        <goto>0</goto>
        <vg>10</vg>
        <calling>5</calling>
        <calls>7</calls>
        <param>5</param>
        <stmt>50</stmt>
        <level>4</level>
        <return>1</return>
        <vocf>4</vocf>
        <ap_cg_cycle>0</ap_cg_cycle>
        <ap_cg_direct_cycle>0</ap_cg_direct_cycle>
        <Num_Unjustified_Violations>Custom_MISRA_Rules_Set
                                   </Num_Unjustified_Violations>
        <Num_Unjustified_BF_Checks>Custom_BF_Checks_Set
                                    </Num_Unjustified_BF_Checks>
    </SQO>
```

```
    <CodingRulesSet ID="Custom_MISRA_Rules_Set">
        <Rule Name="MISRA_C_5_2">0</Rule>
        <Rule Name="MISRA_C_17_6">0</Rule>
    </CodingRulesSet>

    <BugFinderChecksSet ID="Custom_BF_Checks_Set">
        <Check Name="UNREACHABLE">0</Check>
        <Check Name="USELESS_IF">0</Check>
    </BugFinderChecksSet>

</MetricsDefinitions>
```

2. Save this XML file in the folder where remote analysis data is stored, for example, `C:\Users\JohnDoe\AppData\Roaming\Polyspace_RLDatas`.

   If you want to change the folder location, select **Metrics > Metrics and Remote Server Settings**.

3. To make the quality level `Custom-BF-QO-Level` applicable to a certain project, replace the value of the `ApplicableProject` attribute with the project name.

   If you want the quality objectives to apply to all projects, use `ApplicableProject=""`.

4. For specifying coding rules, begin the rule name with the appropriate string followed by the rule number. Use _ instead of a decimal point in the rule number.

Rule	String	Rule numbers
MISRA C: 2004	`MISRA_C_`	"MISRA C:2004 and MISRA AC AGC Coding Rules" on page 12-3
MISRA C: 2012	`MISRA_C3_`	"MISRA C:2012 Directives and Rules"
MISRA C++	`MISRA_Cpp_`	"MISRA C++ Coding Rules" on page 12-86
JSF C++	`JSF_Cpp_`	"JSF C++ Coding Rules" on page 12-124
Custom coding rules	`Custom_`	"Custom Coding Rules"

5. For specifying defects, use the defect acronym. For defect acronyms, see the individual defect reference pages.

6. After you have made your modifications, in the Polyspace Metrics interface, open the metrics for your project. From the **Quality Objectives** list in the upper left, select `ON`.

7. On the **Summary** tab, select an entry in the **Level** column. For the project name that you specified, your new quality objective **Custom-BF-QO-Level** appears in the drop-down list.
8. Select your new quality objective.

   The software compares the thresholds you had specified against your results and updates the **Overall Status** column with **PASS** or **FAIL**.
9. To define another set of custom quality objectives, add the following content to the `Custom-BF-QO-Definitions.xml` file:

```
<SQO ID="Custom-BF-QO-Level_2" ParentID="Custom-BF-QO-Level"
         ApplicableProduct="Bug Finder" ApplicableProject="My_Project">

 ...
</SQO>
```

   - `ID` represents the name of the new set.

     You cannot have the same values of `ID` and `ApplicableProject` for two different sets of quality objectives. For example, if you use `ID="Custom-BF-QO-Level"` for two different custom sets, and `ApplicableProject` is either `My_Project` or `""` for both sets, you see the following error:

     ```
     The SQO level 'Custom-BF-QO-Level' is multiply defined.
     ```
   - `ParentID` specifies another level from which the current level inherits its quality objectives. In the preceding example, the level `Custom-BF-QO-Level_2` inherits its quality objectives from the level `Custom-BF-QO-Level`.

     If you do not want to inherit quality objectives from another level, omit this attribute.
   - `...` represents the additional quality thresholds that you specify for the level `Custom-BF-QO-Level_2`.

     The quality thresholds that you specify override the thresholds that `Custom-BF-QO-Level_2` inherits from `Custom-BF-QO-Level`. For instance, if you specify `<goto>1</goto>`, this objective overrides the threshold specification `<goto>0</goto>` of `Custom-BF-QO-Level`.

Elements in Custom Quality Objective Files

• "HIS Metrics" on page 18-22

- "Non-HIS Metrics" on page 18-22

The following tables list the XML elements that can be added to the custom BF-QO file. The content of each element specifies a threshold against which the software compares analysis results. For each element, the table lists the metric to which the threshold applies. Here, HIS refers to the Hersteller Initiative Software.

HIS Metrics

Element	Metric
comf	Comment Density
path	Number of Paths
goto	Number of Goto Statements
vg	Cyclomatic Complexity
calling	Number of Calling Functions
calls	Number of Called Functions
param	Number of Function Parameters
stmt	Number of Instructions
level	Number of Call Levels
return	Number of Return Statements
vocf	Language Scope
ap_cg_cycle	Number of Recursions
ap_cg_direct_cycle	Number of Direct Recursions
Num_Unjustified_Violations	Number of unjustified violations of coding rules specified by entries under the element CodingRulesSet
Num_Unjustified_BF_Checks	Number of unjustified defects of types specified by entries under the element BugFinderChecksSet

Non-HIS Metrics

Element	Description of metric
fco	Estimated Function Coupling
flin	Number of Lines Within Body

Element	Description of metric
fxln	Number of Executable Lines
ncalls	Number of Call Occurrences

Web Browser Requirements for Polyspace Metrics

Polyspace Metrics supports the following web browsers:

- Internet Explorer® version 7.0, or later
- Firefox® version 3.6, or later
- Google® Chrome version 12.0, or later
- Safari for Mac version 6.1.4 and 7.0.4

To use Polyspace Metrics, install Java®, version 1.4 or later on your computer.

For the Firefox web browser, manually install the required Java plug-in. For example, if your computer uses the Linux operating system:

1. Create a Firefox folder for plug-ins:

   ```
   mkdir ~/.mozilla/plugins
   ```

2. Go to this folder:

   ```
   cd ~/.mozilla/plugins
   ```

3. Create a symbolic link to the Java plug-in, which is available in the Java Runtime Environment folder of your MATLAB installation:

   ```
   ln -s MATLAB_Install/sys/java/jre/glnxa64/jre/lib/amd64/libnpjp2.so
   ```

View Results List in Polyspace Metrics

This example shows how to use Polyspace Metrics to view the Results List in and download results. Your results appear in Polyspace Metrics if,

- Before analyzing your code in batch mode, you selected the **Add to results repository** analysis option.
- After analyzing your code, batch or local mode, you selected **Metrics > Upload to Metrics**.

Open Polyspace Metrics

1 From the Polyspace interface, select **Metrics > Open Metrics**.

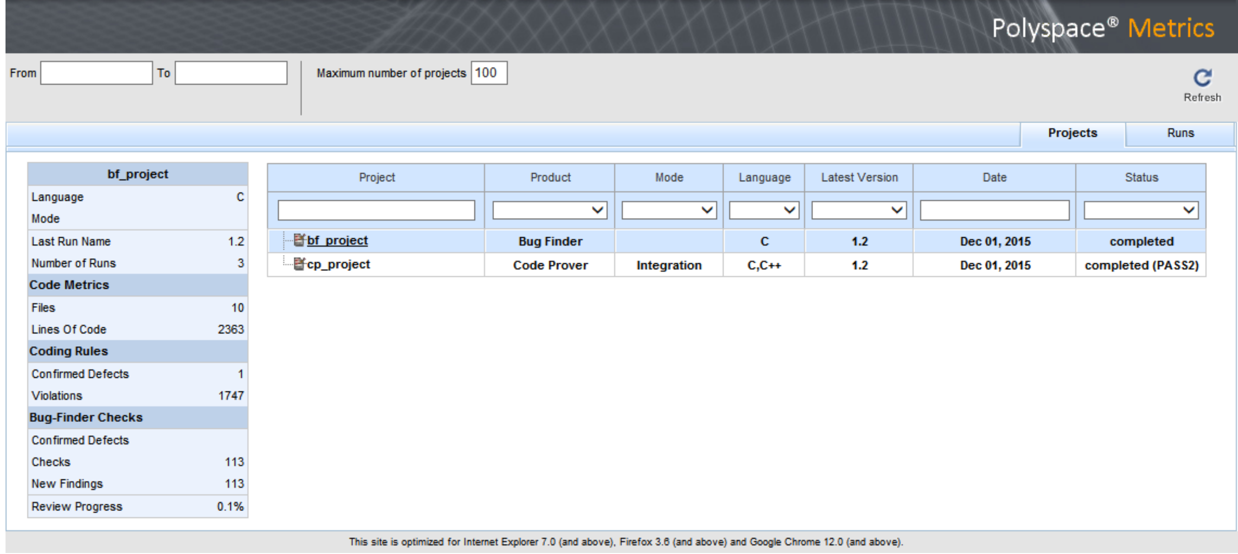
   You can also open the Polyspace Metrics Web UI using the URL:

   `protocol://ServerName:PortNumber`

   - *`protocol`* is either `http` (default) or `https`.

     To use HTTPS, you must configure the web server for HTTPS.
   - *`ServerName`* is the name or IP address of your Polyspace Metrics server.
   - *`PortNumber`* is the web server port number (default 8080).

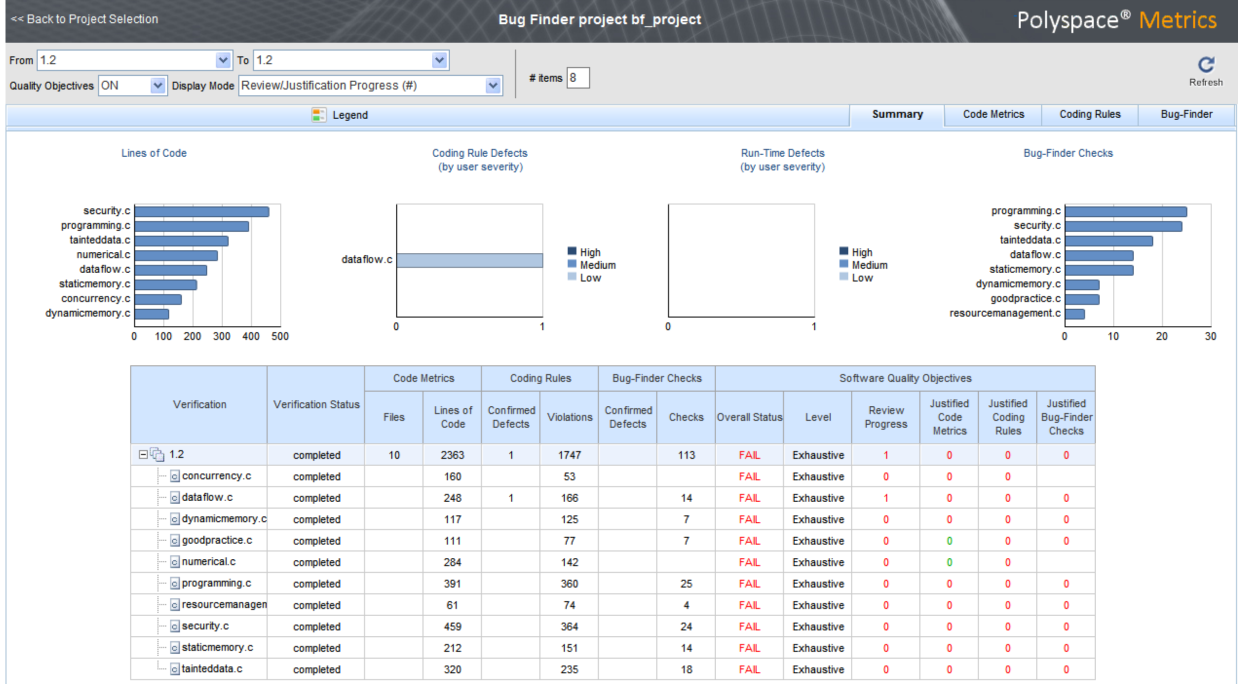
   On the Metrics homepage, you can see all projects uploaded to your Polyspace Metrics repository.



View Results List

1. Select the **Projects** tab.
2. Hover over the project name to see a summary of the project results.
3. To see more details, select the project name.

   The project opens to a Results List for the project.



Verification	Verification Status	Code Metrics		Coding Rules		Bug-Finder Checks		Software Quality Objectives					
		Files	Lines of Code	Confirmed Defects	Violations	Confirmed Defects	Checks	Overall Status	Level	Review Progress	Justified Code Metrics	Justified Coding Rules	Justified Bug-Finder Checks
1.2	completed	10	2363	1	1747		113	FAIL	Exhaustive	1	0	0	0
concurrency.c	completed		160		53			FAIL	Exhaustive	0	0	0	
dataflow.c	completed		248	1	166		14	FAIL	Exhaustive	1	0	0	0
dynamicmemory.c	completed		117		125		7	FAIL	Exhaustive	0	0	0	0
goodpractice.c	completed		111		77		7	FAIL	Exhaustive	0	0	0	0
numerical.c	completed		284		142			FAIL	Exhaustive	0	0	0	
programming.c	completed		391		360		25	FAIL	Exhaustive	0	0	0	0
resourcemanagen	completed		61		74		4	FAIL	Exhaustive	0	0	0	0
security.c	completed		459		364		24	FAIL	Exhaustive	0	0	0	0
staticmemory.c	completed		212		151		14	FAIL	Exhaustive	0	0	0	0
tainteddata.c	completed		320		235		18	FAIL	Exhaustive	0	0	0	0

Polyspace Metrics shows the summary graphically

Confirmed Defects column lists the number of coding rule violations or checks that you have reviewed.

4 To view the results in more detail, select the tabs:

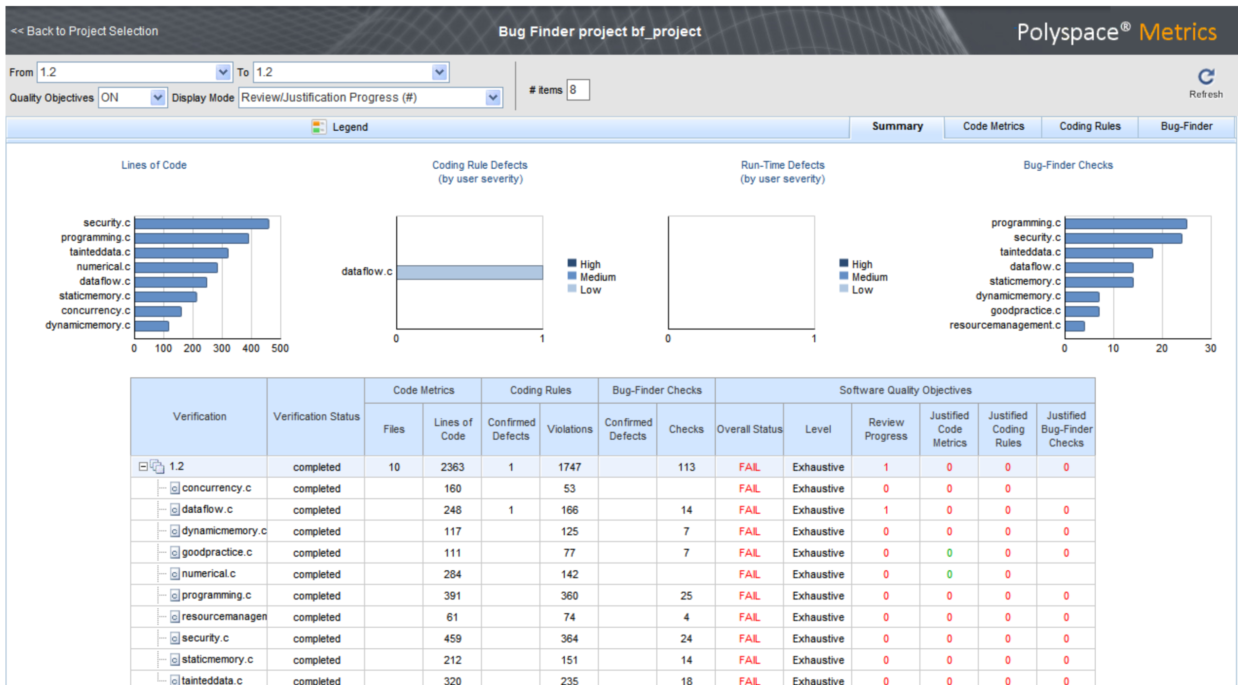
- **Code Metrics**: Statistics about your project such as number of lines, header files, and function calls. To see code metrics, you must enable the analysis option `Calculate code metrics (-code-metrics)`.
- **Coding Rules**: Description of coding rule violations.
- **Bug-Finder**: Description of defects detected by Polyspace Bug Finder.

Download Results

1 Select the **Projects** tab.

2 To view the Results List for your project, on the **Projects** column, select the project name.

The Results List for the project appears on the web page under the **Summary** tab.



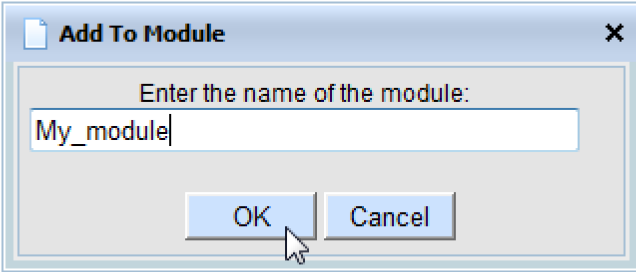
Verification	Verification Status	Code Metrics: Files	Code Metrics: Lines of Code	Coding Rules: Confirmed Defects	Coding Rules: Violations	Bug-Finder Checks: Confirmed Defects	Bug-Finder Checks: Checks	Software Quality Objectives: Overall Status	Level	Review Progress	Justified Code Metrics	Justified Coding Rules	Justified Bug-Finder Checks
1.2	completed	10	2363	1	1747		113	FAIL	Exhaustive	1	0	0	0
concurrency.c	completed		160		53			FAIL	Exhaustive	0	0	0	
dataflow.c	completed		248	1	166		14	FAIL	Exhaustive	1	0	0	0
dynamicmemory.c	completed		117		125		7	FAIL	Exhaustive	0	0	0	0
goodpractice.c	completed		111		77		7	FAIL	Exhaustive	0	0	0	0
numerical.c	completed		284		142			FAIL	Exhaustive	0	0	0	
programming.c	completed		391		360		25	FAIL	Exhaustive	0	0	0	0
resourcemanagen	completed		61		74		4	FAIL	Exhaustive	0	0	0	0
security.c	completed		459		364		24	FAIL	Exhaustive	0	0	0	0
staticmemory.c	completed		212		151		14	FAIL	Exhaustive	0	0	0	0
tainteddata.c	completed		320		235		18	FAIL	Exhaustive	0	0	0	0

3 To download results:

- Individual file — click a file name in the **Verification** column.
- Whole project — click a version number in the **Verification** column.
- Group of files —

  a Right-click 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 Click the name of the module.

Note If you download results using Internet Explorer 11, it may take a minute or two to open the Java plug-in and load the Polyspace interface.

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

See Also

Related Examples

- "Set Up Polyspace Metrics"
- "Address Polyspace Results Through Bug Fixes or Comments" on page 15-2

Troubleshooting in Polyspace Bug Finder

- “License Error –4,0” on page 19-3
- “View Error Information When Analysis Stops” on page 19-4
- “Contact Technical Support” on page 19-7
- “Compiler Not Supported for Project Creation from Build Systems” on page 19-9
- “Slow Build Process When Polyspace Traces the Build” on page 19-19
- “Check if Polyspace Supports Build Scripts” on page 19-20
- “Troubleshooting Project Creation from MinGW Build” on page 19-22
- “Troubleshooting Project Creation from Visual Studio Build” on page 19-23
- “Polyspace Cannot Find the Server” on page 19-25
- “Job Manager Cannot Write to Database” on page 19-26
- “Undefined Identifier Error” on page 19-28
- “Unknown Function Prototype Error” on page 19-32
- “Error Related to #error Directive” on page 19-34
- “Large Object Error” on page 19-36
- “Errors Related to Generic Compiler” on page 19-39
- “Errors Related to Keil or IAR Compiler” on page 19-41
- “Errors Related to Diab Compiler” on page 19-42
- “Errors Related to TASKING Compiler” on page 19-45
- “Conflicting Declarations in Different Translation Units” on page 19-47
- “Errors from Conflicts with Polyspace Header Files” on page 19-53
- “Errors from Assertion or Memory Allocation Functions” on page 19-55
- “Error from Special Characters” on page 19-56
- “Errors from In-Class Initialization (C++)” on page 19-57
- “Errors from Double Declarations of Standard Template Library Functions (C++)” on page 19-58

- “Errors Related to GNU Compiler” on page 19-59
- “Errors Related to Visual Compilers” on page 19-60
- “Eclipse Java Version Incompatible with Polyspace Plug-in” on page 19-62
- “Coding Rule Violations Not Displayed” on page 19-64
- “Insufficient Memory During Report Generation” on page 19-66
- “Error from Disk Defragmentation and Antivirus Software” on page 19-67
- “Errors with Temporary Files” on page 19-68

License Error –4,0

Issue

When you try to run Polyspace, you get this error message:

```
License Error -4,0
```

Cause

You can open multiple instances of Polyspace, but you can only run one code analysis at a time.

If you try to run Polyspace processes from multiple windows, you will get a `License Error –4,0` error.

Solution

Only run one analysis at a time, including any command-line or plugin analyses.

View Error Information When Analysis Stops

If the analysis stops, you can view error information on the screen, either in the user interface or at the command-line terminal. Alternatively, you can view error information in a log file generated during analysis. Based on the error information, you can either fix your source code, add missing files or change analysis options to get past the error.

View Error Information in User Interface

1 View the errors on the **Output Summary** tab.

   The messages on this tab appear with the following icons.

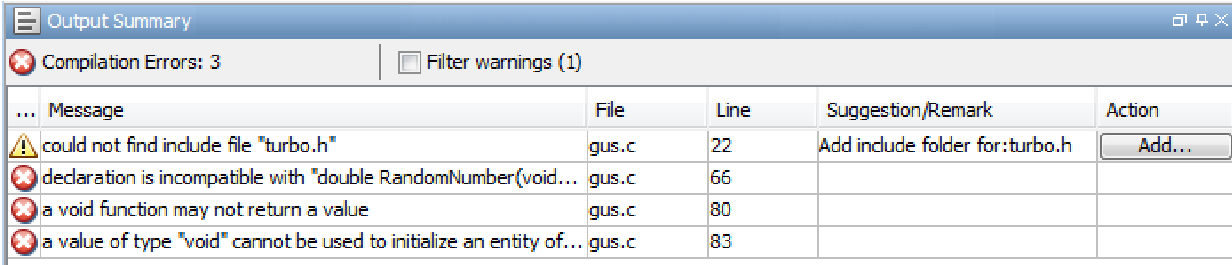
Icon	Meaning
(error icon)	Error that blocks analysis. For instance, the analysis cannot find a variable declaration or definition and therefore cannot determine the variable type.
(warning icon)	Warning about an issue that does not block analysis by itself, but could be related to a blocking error. For instance, the analysis cannot find an include file that is `#include`-d in your code. The issue does not block the analysis by itself, but if the include file contains the definition of a variable that you use in your source code, you can face an error later.
(information icon)	Additional information about the analysis.

2 To diagnose and fix each error, you can do the following:

   - To see further details about the error, select the error message. The details appear in a **Detail** window below the **Output Summary** tab.
   - To open the source code at the line containing the error, double-click the message.

3 If you enable the Compilation Assistant, to fix an error, you can perform certain actions on the **Output Summary** tab.

   The following figure shows an error due to a missing include file `turbo.h`. You can add the missing file by clicking the **Add** button on the **Output Summary** tab.



To turn on the Compilation Assistant, select **Tools > Preferences**. On the **Project and Results Folder** tab, select **Use Compilation Assistant**.

Note the following:

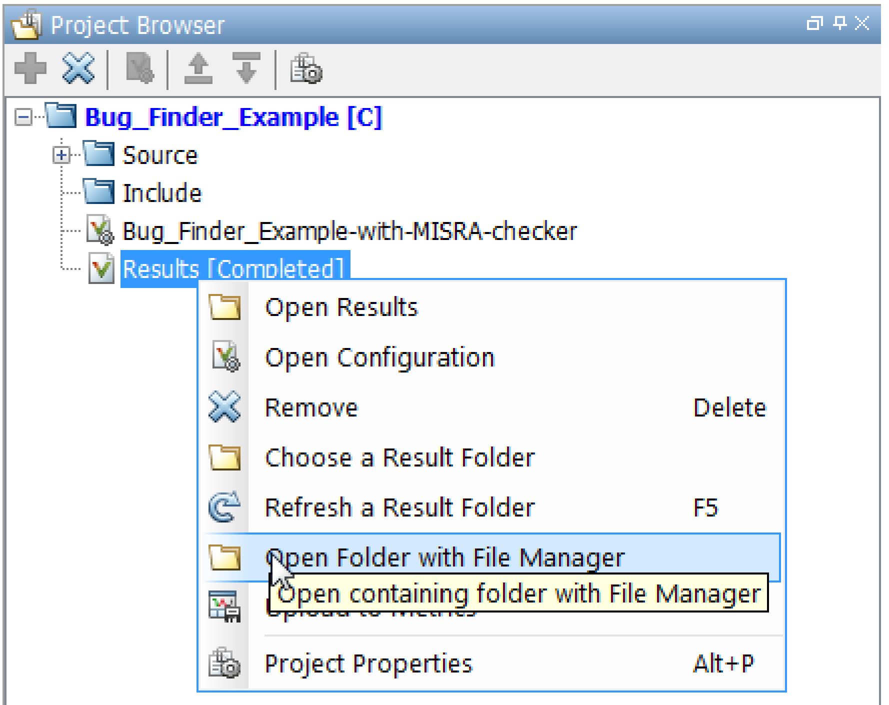
- By default, if some files do not compile, Bug Finder analyzes the remaining files. If you turn on Compilation Assistant, all files must compile. You do not get analysis results even if there is a single compilation error.
- The Compilation Assistant is disabled if you specify the option `Command/script to apply to preprocessed files (-post-preprocessing-command)`

Tip To search the error messages for a specific term, 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 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...
```

Contact Technical Support

To contact MathWorks Technical Support, use this page. You will need a MathWorks Account login and password. For faster turnaround with an issue in Polyspace, besides the required system information, provide appropriate code that reproduces the issue or the verification log file.

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, replacing `matlabroot` with your MATLAB installation folder:
  - UNIX — `matlabroot/polyspace/bin/polyspace-code-prover-nodesktop -ver`
  - Windows — `matlabroot\polyspace\bin\polyspace-code-prover-nodesktop -ver`

Provide Information About the Issue

If you face compilation issues with your project, see “Troubleshooting in Polyspace Bug Finder”. If you are still having issues, contact technical support with the following information:

- The analysis log.

  The analysis 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 analysis 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 analysis log.

  The analysis log is a text file generated in your results folder and titled `Polyspace_`*`version_project_date_time`*`.txt`. It contains the options used for the analysis 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.

Compiler Not Supported for Project Creation from Build Systems

Issue

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

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 8-8.

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\`. Select the configuration that most closely corresponds to your compiler using the mapping between the configuration files and compiler names on page 19-17.
2. Save the file as *my_compiler*.xml. *my_compiler* can be a 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.
4. After saving the edited XML file to *matlabroot*\polyspace\configure\compiler_configuration\, create a project automatically using your build command.

If you see errors, for instance, compilation errors, contact MathWorks Technical Support. After tracing your build command, the software compiles certain files using the compiler specifications detected from your configuration file and build command. Compilation errors might indicate issues in the configuration file.

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 with your compiler configuration file, you can use this file for larger builds.

Elements of Compiler Configuration File

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

XML Element	Content Description	Content Example for GNU C Compiler

`<compiler_names><name> ...`
 	
`</name><compiler_names>`
	Name of the compiler executable. This executable transforms your `.c` files into object files. You can add several binary names, each in a separate `<name>...</name>` element. The software checks for each of the provided names and uses the compiler name for which it finds a match.
	
	You must not specify the linker binary inside the `<name>...</name>` elements.
	
	If the name that you specify is present in an existing compiler configuration file, an error occurs. To avoid the error, use the additional option `-compiler-config` *`my_compiler`*`.xml` when tracing the build so that the software explicitly uses your compiler configuration file.
		• `gcc`
		• `gpp`

XML Element	Content Description	Content Example for GNU C Compiler

`<include_options><opt> ...`
 	
`</opt></include_options>`
	The option that you use with your compiler to specify include folders.
	
	To specify options where the argument immediately follows the option, use an `isPrefix` attribute for `opt` and set it to `true`.
		`-I`

`<system_include_options>`
 	
`<opt> ... </opt>`
 	
`</system_include_options>`
	The option that you use with your compiler to specify system headers.
	
	To specify options where the argument immediately follows the option, use an `isPrefix` attribute for `opt` and set it to `true`.
		`-isystem`

`<preinclude_options><opt> ...`
 	
`</opt></preinclude_options>`
	The option that you use with your compiler to force inclusion of a file in the compiled object.
	
	To specify options where the argument immediately follows the option, use an `isPrefix` attribute for `opt` and set it to `true`.
		`-include`

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

XML Element	Content Description	Content Example for GNU C Compiler

`<semantic_options><opt> ...`
 	
`</opt></semantic_options>`
	The options that you use to modify the compiler behavior. These options specify the language settings to which the code must conform.
	
	You can use the `isPrefix` attribute to specify multiple options that have the same prefix and the `numArgs` attribute to specify options with multiple arguments. For instance:
	
	• Instead of
	`<opt>-m32</opt>`
	`<opt>-m64</opt>`
	You can write `<opt isPrefix="true">-m</opt>`.
	• Instead of
	`<opt>-std=c90</opt>`
	`<opt>-std=c99</opt>`
	You can write `<opt numArgs="1">-std</opt>`. If your makefile uses `-std c90` instead of `-std=c90`, this notation also supports that usage.
		• `-ansi`
		• `-std =C90`
		• `-std =c++11`
		• `-fun signed -char`

XML Element	Content Description	Content Example for GNU C Compiler
`<dialect> ... </dialect>`	The Polyspace dialect that corresponds to or closely matches your compiler dialect. The content of this element directly translates to the option **Dialect** in your Polyspace project or options file. For the complete list of dialects, on the **Configuration** pane, select **Target & Compiler**.	`gnu4.7`

`<preprocess_options_list>`
 	
`<opt> ... </opt>`
 	
`</preprocess_options_list>`
	The options that specify how your compiler generates a preprocessed file.
	
	You can use the macro `$(OUTPUT_FILE)` if your compiler does not allow sending the preprocessed file to the standard output. Instead it defines the preprocessed file internally.
		`-E`
		
		For an example of the `$(OUTPUT_FILE)` macro, see the existing compiler configuration file `cl2000.xml`.

XML Element	Content Description	Content Example for GNU C Compiler
<preprocessed_output_file> ... </ preprocessed_output_file>	The name of file where the preprocessed output is stored. You can use the following macros when the name of the preprocessed output file is adapted from the source file: • $(SOURCE_FILE): Source file name • $(SOURCE_FILE_EXT): Source file extension • $(SOURCE_FILE_NO_EXT): Source file name without extension For instance, use $(SOURCE_FILE_NO_EXT).pre when the preprocessor file name has the same name as the source file, but with extension .pre.	For an example of this element, see the existing compiler configuration file xc8.xml.
<src_extensions><ext> ... </ext></src_extensions>	The file extensions for source files.	• c • cpp • c++
<obj_extensions><ext> ... </ext></obj_extensions>	The file extensions for object files.	
<precompiled_header_extensions> ... </precompiled_header_extensions>	The file extensions for precompiled headers (if available).	

XML Element	Content Description	Content Example for GNU C Compiler

`<polyspace_c_extra_options_list>`
`<opt> ... </opt>`
`</polyspace_c_extra_options_list>`
	Additional options that will be added to your project configuration	To avoid compilation errors due to non-ANSI extension keywords, enter `-D` *`keyword`*. For more information, see `Preprocessor definitions (-D)`.

`<polyspace_cpp_extra_options_list>`
`<opt> ... </opt>`
`</polyspace_cpp_extra_options_list>`
	Additional options that will be added to your C++ project configuration	To avoid compilation errors due to non-ANSI extension keywords, enter `-D` *`keyword`*. For more information, see `Preprocessor definitions (-D)`.

Mapping Between Existing Configuration Files and Compiler Names

Select the configuration file in *`matlabroot`*`\polyspace\configure\compiler_configuration\` that most closely resembles the configuration of your compiler. Use the following table to map compilers to their configuration files.

Vendor	Compiler Name	XML File
Microsoft	Visual C++	cl.xml
Texas Instruments	TM320 and its derivatives	cl2000.xml

Vendor	Compiler Name	XML File
Not applicable	Clang	clang.xml
Cosmic	`cx6808`	cx6808.xml
Wind River	Diab	diab.xml
Not applicable	gcc	gcc.xml
Green Hills Software	Green Hills	ghs_arm.xml
		ghs_arm64.xml
		ghs_i386.xml
		ghs_ppc.xml
		ghs_rh850.xml
		ghs_tricore.xml
IAR	IAR Embedded Workbench	iar.xml
		iar-arm.xml
		iar-avr.xml
		iar-msp430.xml
		iar-rh850.xml
		iar-rl78.xml
Altium	TASKING®	tasking.xml
		tasking-166.xml
		tasking-850.xml
		tasking-arm.xml
Not applicable	Tiny C	tcc.xml
NXP	CodeWarrior	ti_arm.xml
		ti_c28x.xml
		ti_c6000.xml
		ti_msp430.xml
Microchip	`xc8 (PIC)`	xc8.xml

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 `polyspace-configure`.
- If you trace your build from the DOS, UNIX or MATLAB command line, use this flag with the `polyspace-configure` command or `polyspaceConfigure` function.

Check if Polyspace Supports Build Scripts

Issue

This topic is relevant only if you are creating a Polyspace project in Windows from your build scripts.

When Polyspace traces your build script in a Windows console application other than `cmd.exe`, the command fails. However, the build command by itself executes to completion.

For instance, your build script executes to completion from the Cygwin shell. However, when Polyspace traces the build, the build script throws an error.

Possible Cause

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

Polyspace attempts to trace your build script with the assumption that the script runs to completion in `cmd.exe`. Therefore, even if your script runs to completion in the alternate console application, when Polyspace traces the build, the script can fail.

Solution

Make sure that your build script executes to completion in the `cmd.exe` interface. If the build executes successfully, create a wrapper `.bat` file around your script and trace this file.

For instance, before you trace a build command that executes to completion in the Cygwin shell, do one of the following:

- Launch the Cygwin shell from `cmd.exe` and then run your build script. For instance, if you use a script `build.sh` to build your code, enter the following command at the DOS command line:

  ```
  cmd.exe /C "C:\cygwin64\bin\bash.exe" -c build.sh
  ```
- Find the full path to your build script and then run this script from `cmd.exe`.

For instance, enter the following command at the DOS command line:

```
cmd.exe /C path_to_script
```

path_to_script is the full path to your build script. For instance, `C:\my_scripts\build.sh`.

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

If the steps complete successfully, trace the build command after launching it from `cmd.exe`. For instance, on the command-line, do the following to create a Polyspace options file.

1. Enter your build commands in a `.bat` file.

   ```
   rem @echo off
   cmd.exe /C "C:\cygwin64\bin\bash.exe" -c build.sh
   ```

   Name the file, for instance, `launching.bat`.
2. Trace the build commands in the `.bat` file and create a Polyspace options file.

   ```
   "C:\Program Files\MATLAB\R2017b\polyspace\bin\polyspace-configure.exe"
             -output-options-file myOptions.txt launching.bat
   ```

You can now run `polyspace-bug-finder-nodesktop` on the options file.

Troubleshooting Project Creation from MinGW Build

Issue

You create a project from a MinGW build, but get an error when running an analysis on the project. The error message comes from using one of these keywords: `__declspec`, `__cdecl`, `__fastcall`, `__thiscall` or `__stdcall`.

Cause

When you create a project from a MinGW build, the project uses a GNU compiler. Polyspace does not recognize these keywords for the GNU compilers.

Solution

Replace these keywords with equivalent keywords just for the purposes of analysis.

Before analysis, for the option `Preprocessor definitions (-D)`, enter:

- `__declspec(x)=__attribute__((x))`
- `__cdecl=__attribute__((__cdecl__))`
- `__fastcall=__attribute__((__fastcall__))`
- `__thiscall=__attribute__((__thiscall__))`
- `__stdcall=__attribute__((__stdcall__))`

Troubleshooting Project Creation from Visual Studio Build

In this section...

"Cannot Create Project from Visual Studio Build" on page 19-23
"Compilation Error After Creating Project from Visual Studio Build" on page 19-23

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 version of the compiler, `visual12.0` to your project. This assignment can cause compilation errors. For more information on the option to specify compilers, see `Compiler (-compiler)`.

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 setting for **Compiler**. If it is set to `visual12.0`, change it to `visual10`.

Note If you are creating an options file from your Visual Studio 2010 build, check the `-compiler` argument. If it is set to `visual12.0`, 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>visual12.0</dialect>
     ```

     to

     ```
     <dialect>visual10</dialect>
     ```
  3. Create your project or options file. The compiler is already assigned to `visual10`.

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".

Job Manager Cannot Write to Database

Message

`Unable to write data to the job manager database`

Possible Cause

If the job scheduler cannot send data to the localhost, Polyspace returns this error. The most likely reasons for the MJS being unable to connect to the client computer are:

- Firewalls do not allow traffic from the MJS to the client.
- The MJS cannot resolve the short hostname of the client computer.

Workaround

Add localhost IP to configuration.

1. Select **Tools** > **Preferences**.
2. Select the **Server Configuration** tab.
3. In the **Localhost IP address** field, enter the IP address of your local computer.

   To retrieve your IP address:

   - Windows
     a. Open **Control Panel** > **Network and Sharing Center**.
     b. Select your active network.
     c. In the Status window, click **Details**. Your IP address is listed under **IPv4 address**.
   - Linux — Run the `ifconfig` command and find the `inet addr` corresponding to your network connection.
   - Mac — Open **System Preferences** > **Network**.

See Also

Related Examples

- “Set Up Server for Metrics and Remote Analysis”
- “Connection Problems Between the Client and MJS” (Parallel Computing Toolbox)

Undefined Identifier Error

Issue

Polyspace verification fails during the compilation phase with a message about undefined identifiers.

The message indicates that Polyspace cannot find a variable definition. Therefore, it cannot identify the variable type.

Possible Cause: Missing Files

The source code you provided does not contain the variable definition. For instance, the variable is defined in an include file that Polyspace cannot find.

If you `#include`-d the include file in your source code but did not add it to your Polyspace project, you see a previous warning:

```
Warning: could not find include file "my_include.h"
```

Solution

If the variable definition occurs in an include file, add the folder that contains the include file.

- In the user interface, add the folder to your project.

  For more information, see “Add Source Files for Analysis in Polyspace User Interface” on page 1-2.
- At the command line, use the flag `-I` with the `polyspace-bug-finder-nodesktop` command.

  For more information, see `-I`.

Possible Cause: Unrecognized Keyword

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

If the variable represents a keyword that Polyspace does not recognize, replace or remove the keyword from your source code or preprocessed code.

If you remove or replace the keyword from the preprocessed code, you can avoid the compilation error while keeping your source code intact. You can do one of the following:

- Replace or remove each individual unknown keyword using an analysis option. Replace the compiler-specific keyword with an equivalent keyword from the ANSI C Standard.

  For information on the analysis option, see `Preprocessor definitions (-D)`.
- Declare the 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 (-include)`.

Possible Cause: Declaration Embedded in #ifdef Statements

The variable is declared in a branch of an `#ifdef` *`macro_name`* preprocessor directive. For instance, the declaration of a variable `max_power` occurs as follows:

```
#ifdef _WIN32
  #define max_power 31
#endif
```

Your compilation toolchain might consider the macro *`macro_name`* as implicitly defined and execute the `#ifdef` branch. However, the Polyspace compilation might not consider the macro as defined. Therefore, the `#ifdef` branch is not executed and the variable `max_power` is not declared.

Solution

To work around the compilation error, do one of the following:

- Use **Target & Compiler** options to directly specify your compiler. For instance, to emulate a Visual C++ compiler, set the **Compiler** to `visual12.0`. See “Target and Compiler”.
- Define the macro explicitly using the option `Preprocessor definitions (-D)`.

Note If you create a Polyspace by tracing your build commands, most **Target & Compiler** options are automatically set.

Possible Cause: Project Created from Non-Debug Build

This can be a possible cause only if the undefined identifier occurs in an `assert` statement (or an equivalent Visual C++ macro such as `ASSERT` or `VERIFY`).

Typically, you come across this error in the following way. You create a Polyspace project from a build system in non-debug mode. When you run an analysis on the project, you face a compilation error from an undefined identifier in an `assert` statement. You find that the identifier `my_identifier` is defined in a `#ifndef NDEBUG` statement, for instance as follows:

```
#ifndef NDEBUG
int my_identifier;
#endif
```

The C standard states that when the `NDEBUG` macro is defined, all assert statements must be disabled.

Most IDEs define the `NDEBUG` macro in their build systems. When you build your source code in your IDE in non-debug mode, code in a `#ifndef NDEBUG` statement is removed during preprocessing. For instance, in the preceding example, `my_identifier` is not defined. If `my_identifier` occurs only in assert statements, it is not used either, because `NDEBUG` disables assert statements. You do not have compilation errors from undefined identifiers and your build system executes successfully.

Polyspace does not disable `assert` statements even if `NDEBUG` macro is defined because the software uses `assert` statements internally to enhance verification.

When you create a Polyspace project from your build system, if your build system defines the `NDEBUG` macro, it is also defined for your Polyspace project. Polyspace removes code in a `#ifndef NDEBUG` statement during preprocessing, but does not disable `assert` statements. If `assert` statements in your code rely on the code in a `#ifndef NDEBUG` statement, compilation errors can occur.

In the preceding example:

- The definition of `my_identifier` is removed during preprocessing.

- `assert` statements are not disabled. When `my_identifier` is used in an `assert` statement, you get an error because of undefined identifier `my_identifier`.

Solution

To work around this issue, create a Polyspace project from your build system in debug mode. When you execute your build system in debug mode, `NDEBUG` is not defined. When you create a Polyspace project from this build, `NDEBUG` is not defined for your Polyspace project.

Depending on your project settings, use the option that enables building in debug mode. For instance, if your build system is `gcc`-based, you can define the `DEBUG` macro and undefine `NDEBUG`:

```
gcc -DDEBUG=1 -UNDEBUG *.c
```

Alternatively, you can disable the `assert` statements in your preprocessed code using the option `Disabled preprocessor definitions (-U)`. However, Polyspace will not be able to emulate the `assert` statements.

Unknown Function Prototype Error

Issue

During the compilation phase, the software displays a warning or error message about unknown function prototype.

```
the prototype for function 'myfunc' is unknown
```

The message indicates that Polyspace cannot find a function prototype. Therefore, it cannot identify the data types of the function argument and return value, and has to infer them from the calls to the function.

To determine the data types for such functions, Polyspace follows the C99 Standard (ISO/IEC 9899:1999, Chapter 6.5.2.2: Function calls).

- The return type is assumed to be `int`.
- The number and type of arguments are determined by the first call to the function. For instance, if the function takes one `double` argument in the first call, for subsequent calls, the software assumes that it takes one `double` argument. If you pass an `int` argument in a subsequent call, a conversion from `int` to `double` takes place.

During the linking phase, if a mismatch occurs between the number or type of arguments or the return type in different compilation units, the analysis follows an internal algorithm to resolve this mismatch and determine a common prototype.

Cause

The source code you provided does not contain the function prototype. For instance, the function is declared in an include file that Polyspace cannot find.

If you `#include`-d the include file in your source code but did not add it to your Polyspace project, you see a previous warning:

```
Warning: could not find include file "my_include.h"
```

Solution

Search for the function declaration in your source repository.

If you find the function declaration in an include file, add the folder that contains the include file.

- In the user interface, add the folder to your project.

  For more information, see “Add Source Files for Analysis in Polyspace User Interface” on page 1-2.
- At the command line, use the flag `-I` with the `polyspace-bug-finder-nodesktop` command.

  For more information, see `-I`.

Error Related to #error Directive

Issue

The analysis stops with a message containing a #error directive. For instance, the following message appears: #error directive: !Unsupported platform; stopping!.

Cause

You typically use the #error directive in your code to trigger a fatal error in case certain macros are not defined. Your compiler implicitly defines the macros, therefore the error is not triggered when you compile your code. However, the default Polyspace compilation does not consider the macros as defined, therefore, the error occurs.

For instance, in the following example, the #error directive is reached only if the macros __BORLANDC__, __VISUALC32__ or __GNUC__ are not defined. If you use a GNU C compiler, for instance, the compiler considers the macro __GNUC__ as defined and the error does not occur. However, if you use the default Polyspace compilation, it does not consider the macros as defined.

```
#if defined(__BORLANDC__) || defined(__VISUALC32__)
#define MYINT int
#elif defined(__GNUC__)
#define MYINT long
#else
#error !Unsupported platform; stopping!
#endif
```

Solution

For successful compilation, do one of the following:

- Specify a compiler such as visual12.0 or gnu4.9. Specifying a compiler defines some of the compilation flags for the analysis.

  For more information, see Compiler (-compiler).
- If the available compiler options do not match your compiler, explicitly define one of the compilation flags __BORLANDC__, __VISUALC32__, or __GNUC__.

For more information, see `Preprocessor definitions (-D)`.

Large Object Error

Issue

The analysis stops during compilation with a message indicating that an object is too large.

Cause

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.

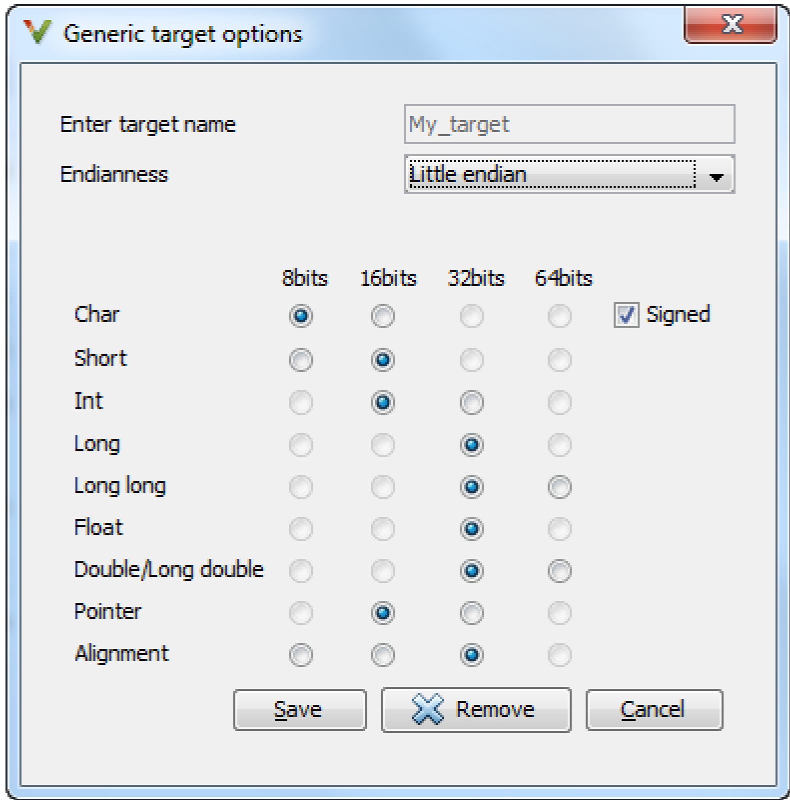
For instance, you get the message, `Limitation: struct or union is too large` in the following example. You specify a pointer size of 16 bits. The maximum object size allocated to a pointer, and therefore the maximum allowed size for an object, can be 2^{16}-1 bytes. However, you declare a structure 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 (-target)`.

   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`.

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 Related to Generic Compiler

If you use a generic compiler, you can encounter this issue. For more information, see `Compiler (-compiler)`.

Issue

The analysis stops with an error message related to a non-ANSI C keyword, for instance, `data` or attributes such as `__attribute__((weak))`.

Depending on the location of the keyword, the error message can vary. For instance, this line causes the error message: `expected a ";"`.

```
data int tab[10];
```

Cause

The generic Polyspace compiler supports only ANSI C keywords. If you use a language extension, the generic compiler does not recognize it and treats the keyword as a regular identifier.

Solution

Specify your compiler by using the option `Compiler (-compiler)`.

If your compiler is not directly supported or is not based on a supported compiler, you can use the generic compiler. To work around the compilation errors:

- If the keyword is related to memory modelling, remove it from the preprocessed code. For instance, to remove the `data` keyword, enter `data=` for the option `Preprocessor definitions (-D)`.
- If the keyword is related to an attribute, remove attributes from the preprocessed code. Enter `__attribute__(x)=` for the option `Preprocessor definitions (-D)`.

  If your code has this line:

  ```
  void __attribute__ ((weak)) func(void);
  ```

  And you remove attributes, the analysis reads the line as:

```
void func(void);
```

When you use these workarounds, your source code is not altered.

Errors Related to Keil or IAR Compiler

If you use the compiler, Keil or IAR, you can encounter this issue. For more information, see `Compiler (-compiler)`.

Missing Identifiers

Issue

The analysis stops with the error message, `expected an identifier`, as if an identifier is missing. However, in your source code, you can see the identifier.

Cause

If you select Keil or IAR as your compiler, the software removes certain keywords during preprocessing. If you use these keywords as identifiers such as variable names, a compilation error occurs.

For a list of keywords that are removed, see "Supported Keil or IAR Language Extensions" on page 8-13.

Solution

Specify that Polyspace must not remove the keywords during preprocessing. Define the macros `__PST_KEIL_NO_KEYWORDS__` or `__PST_IAR_NO_KEYWORDS__`.

For more information, see `Preprocessor definitions (-D)`.

Errors Related to Diab Compiler

If you choose `diab` for the option `Compiler (-compiler)`, you can encounter this issue.

Issue

During Polyspace analysis, you see an error related to a keyword specific to the Diab compiler. For instance, you see an error related to the `restrict` keyword.

Cause

You typically use a compiler flag to enable the keyword. The Polyspace analysis does not enable these keywords by default. You have to make Polyspace aware of your compiler flags.

The Polyspace analysis does not enable these keywords by default to prevent compilation errors. Another user might not enable the keyword and instead use the keyword name as a regular identifier. If Polyspace treats the identifier as a keyword, a compilation error will occur.

Solution

Use the command-line option `-compiler-parameter` in your Polyspace analysis as follows. You use this command-line option to make Polyspace aware of your compiler flags. In the user interface, you can enter the command-line option in the field `Other`. You can enter the option multiple times.

The argument of `-compiler-parameter` depends on the keyword that causes the error. Once you enable the keyword, do not use the keyword name as a regular identifier. For instance, once you enable the keyword `pixel`, do not use `pixel` as a variable name. The statement `int pixel = 1` causes a compilation error.

- `restrict` keyword:

  You typically use the compiler flag `-Xlibc-new` or `-Xc-new`. For your Polyspace analysis, use

  ```
  -compiler-parameter -Xc-new
  ```

The following code will not compile with Polyspace unless you specify the compiler flag.

```
int sscanf(const char *restrict, const char *restrict, ...);
```

- PowerPC AltiVec vector extensions such as the `vector` type qualifier:

  You typically use the compiler flag `-tPPCALLAV:`. For your Polyspace analysis, use

  ```
  -compiler-parameter -tPPCALLAV:
  ```

  The following code will not compile with Polyspace unless you specify the compiler flag.

  ```
  vector unsigned char vbyte;
  vector bool vbool;
  vector pixel vpx;

  int main(int argc, char** argv)
  {
    return 0;
  }
  ```

- Extended keywords such as `pascal`, `inline`, `packed`, `interrupt`, `extended`, `__X`, `__Y`, `vector`, `pixel`, `bool` and others:

  You typically use the compiler flag `-Xkeywords=`. For your Polyspace analysis, use

  ```
  -compiler-parameter -Xkeywords=0xFFFFFFFF
  ```

  The following code will not compile with Polyspace unless you specify the compiler flag.

  ```
  packed(4) struct s2_t {
      char b;
      int i;
  } s2;

  packed(4,2) struct s3_t {
      char b;
  } s3;

  int pascal foo = 4;

  int main(int argc, char** argv) {
      foo++;
  ```

```
    return 0;
}
```

Errors Related to TASKING Compiler

If you choose `tasking` for the option `Compiler (-compiler)`, you can encounter this issue.

Issue

During Polyspace analysis, you see an error related to a Special Function Register data type.

Cause

When compiling with the TASKING compiler, you typically use the following compiler flags to specify where Special Function Register (SFR) data types are declared:

- `--cpu=`*`xxx`*: The compiler implicitly `#include`s the file `sfr/reg`*`xxx`*`.sfr` in your source files. Once `#include`-ed, you can use Special Function Registers (SFR-s) declared in that `.sfr` file.
- `--alternative-sfr-file`: The compiler uses an alternative SFR file instead of the regular SFR file. You can use Special Function Registers (SFR-s) declared in that alternative SFR file.

If you specify the TASKING compiler for your Polyspace analysis, the analysis makes the following assumptions about these compiler flags:

- `--cpu=`*`xxx`*: The analysis chooses a specific value of *`xxx`*. If you use a different value with your TASKING compiler, you can encounter an error during Polyspace analysis.

  The *`xxx`* value that the Polyspace analysis uses depends on your choice of `Target processor type (-target)`:

  - `tricore`: `tc1793b`
  - `c166`: `xc167ci`
  - `rh850`: `r7f701603`
  - `arm`: `ARMv7M`
- `--alternative-sfr-file`: The analysis assumes that you do not use an alternative SFR file. If you use one, you can encounter an error.

Solution

Use the command-line option `-compiler-parameter` in your Polyspace analysis as follows. You use this command-line option to make Polyspace aware of your compiler flags. In the user interface, you can enter the command-line option in the field `Other`. You can enter the option multiple times.

- `--cpu=xxx`: For your Polyspace analysis, use

  ```
  -compiler-parameter --cpu=xxx
  ```

  Here, *xxx* is the value that you use when compiling with your compiler.
- `--alternative-sfr-file`: For your Polyspace analysis, use

  ```
  -compiler-parameter --alternative-sfr-file
  ```

  If you still encounter an error because Polyspace is not able to locate your `.asfr` file, explicitly `#include` your `.asfr` file in the preprocessed code using the option `Include (-include)`.

  Typically, the path to the file is `Tasking_C166_INSTALL_DIR\include\sfr\regCPUNAME.asfr`. For instance, if your TASKING compiler is installed in `C:\Program Files\Tasking\C166-VX_v4.0r1\` and you use the CPU-related flag `-Cxc2287m_104f` or `--cpu=xc2287m_104f`, the path is `C:\Program Files\Tasking\C166-VX_v4.0r1\include\sfr\regxc2287m.asfr`.

  You can also encounter the same issue with alternative sfr files when you trace your build command. For more information, see “Requirements for Project Creation from Build Systems” on page 8-8.

Conflicting Declarations in Different Translation Units

Issue

The analysis shows a warning:

```
C++ compilation and linking of translation units failed. This can
lead to incomplete results for project-wide C++ coding rule violations.
Relaunch with option -verbose to obtain more details.
```

If you rerun the analysis with the option `-verbose`, you see an error or warning similar to one of these error messages:

- `Declaration of [...] is incompatible with a declaration in another translation unit ([...])`
- `Declaration of [...] had a different meaning during compilation of [...] ([...])`

The error indicates that the same variable or function or data type is declared differently in different translation units. The conflicting declarations violate the One Definition Rule (cf. C++Standard, ISO/IEC 14882:2003, Section 3.2). When conflicting declarations occur, Polyspace does not choose a declaration and continue analysis.

Common compilation toolchains often do not store data type information during the linking process. The conflicting declarations do not cause errors with your compiler. Polyspace Bug Finder follows stricter standards for linking to detect violations of system-wide coding rules.

To identify the root cause of the error:

1. From the error message, identify the two source files with the conflicting declarations.

   For instance, an error message looks like this message:

   ```
   C:\field.h, line 1: declaration of class "a_struct" had
         a  different meaning during compilation of "file1.cpp"
   | struct a_struct {
   |
   | Detected during compilation of secondary translation unit "file2.cpp"
   ```

   The message shows that the structure `a_struct` has a conflicting declaration in `file1.cpp` and `file2.cpp`.

2 Try to identify the conflicting declarations in the source files.

Otherwise, open the translation units containing these files. Sometimes, the translation units or preprocessed files show the conflicting declarations more clearly than the source files because the preprocessor directives, such as `#include` and `#define` statements, are replaced appropriately and the macros are expanded.

a Rerun the analysis with the flag `-keep-relaunch-files` so that all translation units are saved. In the user interface, enter the flag for the option `Other`.

The translation units or preprocessed files are stored in a zipped file `ci.zip` in a subfolder `.relaunch` of the results folder.

b Unzip the contents of `ci.zip`.

The preprocessed files have the same name as the source files. For instance, the preprocessed file with `file1.cpp` is named `file1.ci`.

When you open the preprocessed files at the line numbers stated in the error message, you can spot the conflicting declarations.

Possible Cause: Variable Declaration and Definition Mismatch

A variable declaration does not match its definition. For instance:

- The declaration and definition use different data types.
- The variable is declared as signed, but defined as unsigned.
- The declaration and definition uses different type qualifiers.
- The variable is declared as an array, but defined as a non-array variable.
- For an array variable, the declaration and definition use different array sizes.

In this example, the code shows a linking error because of a mismatch in type qualifiers. The declaration in `file1.c` does not use type qualifiers, but the definition in `file2.c` uses the `volatile` qualifier.

<table>
<tr><th>file1.c</th><th>file2.c</th></tr>
<tr><td><pre>extern int x;

void main(void)
{/* Variable x used */}</pre></td><td><pre> volatile int x;</pre></td></tr>
</table>

In these cases, you can typically spot the difference by looking at the source files. You do not need to see the preprocessed files.

Solution

Make sure that the variable declaration matches its definition.

Possible Cause: Function Declaration and Definition Mismatch

A function declaration does not match its definition. For instance:

- The declaration and definition use different data types for arguments or return values.
- The declaration and definition use a different number of arguments.
- A variable-argument or varargs function is declared in one function, but it is called in another function without a previous declaration.

  In this case, the error message states that the required prototype for the function is missing.

In this example, the code shows a linking error because of a mismatch in the return type. The declaration in `file1.c` has return type `int`, but the definition in `file2.c` has return type `float`.

file1.c	file2.c

`int input(void);`
 	
`void main() {`
`  int val = input();`
`}`
	`float input(void) {`
	`  float x = 1.0;`
	`  return x;`
	`}`

In these cases, you can typically find the difference by looking at the source files. You do not need to see the preprocessed files.

Solution

Make sure that the function declaration matches its definition.

Even if your build process allows these errors, you can have unexpected results during run time. If a function declaration and definition with conflicting prototypes exist in your code, when you call the function, the result can be unexpected.

For a variable-argument or varargs function, declare the function before you call it. If you do not want to change your source code, you can work around this linking error.

1. Add the function declaration in a separate file.
2. Only for the purposes of verification, `#include` this file in every source file by using the option `Include (-include)`.

Possible Cause: Macro-dependent Definitions

A variable definition is dependent on a macro being defined earlier. One source file defines the macro while another does not, causing conflicts in variable definitions.

In this example, `file1.cpp` and `file2.cpp` include a header file `field.h`. The header file defines a structure `a_struct` that is dependent on a macro definition. Only one of the two files, `file2.cpp`, defines the macro `DEBUG`. The definition of `a_struct` in the translation unit with `file1.cpp` differs from the definition in the unit with `file2.cpp`.

file1.cpp

```
#include "field.h"

int main()
{
    a_struct s;
    init_a_struct(&s);
    return 0;
}
```

file2.cpp

```
#define DEBUG

#include <string.h>
#include "field.h"

void init_a_struct(a_struct* s)
{
    memset(s, 0, sizeof(*s));
}
```

field.h:

```
struct a_struct {
    int n;
#ifdef DEBUG
    int debug;
#endif
};
```

When you open the preprocessed files `file1.ci` and `file2.ci`, you see the conflicting declarations.

<table>
<tr><th>file1.ci</th><th>file2.ci</th></tr>
<tr><td><pre>struct a_struct {
    int n;

};</pre></td><td><pre>struct a_struct {
    int n;

    int debug;

};</pre></td></tr>
</table>

Solution

Avoid macro-dependent definitions. Otherwise, fix the linking errors. Make sure that the macro is either defined or undefined on all paths that contain the variable definition.

Possible Cause: Keyword Redefined as Macro

A keyword is redefined as a macro, but not in all files.

In this example, `bool` is a keyword in `file1.cpp`, but it is redefined as a macro in `file2.cpp`.

<table>
<tr><th>file1.cpp</th><th>file2.cpp</th></tr>
<tr><td><pre>#include "bool.h"

int main()
{
    return 0;
}</pre></td><td><pre>#define false 0
#define true (!false)

#include "bool.h"</pre></td></tr>
<tr><td colspan="2">bool.h:
<pre>template <class T>
struct a_struct {
    bool flag;
    T t;
    a_struct() {
        flag = true;
    }
};</pre></td></tr>
</table>

Solution

Be consistent with your keyword usage throughout the program. Use the keyword defined in a standard library header or use your redefined version.

Possible Cause: Differences in Structure Packing

A `#pragma pack(n)` statement changes the structure packing alignment, but not in all files. See also “#pragma Directives” (Polyspace Code Prover).

In this example, the default packing alignment is used in `file1.cpp`, but a `#pragma pack(1)` statement enforces a packing alignment of 1 byte in `file2.cpp`.

<table>
<tr><th>file1.cpp</th><th>file2.cpp</th></tr>
<tr><td><pre>int main()
{
    return 0;
}</pre></td><td><pre>#pragma pack(1)

#include "pack.h"</pre></td></tr>
<tr><td colspan="2">pack.h:<pre>struct a_struct {
    char ch;
    short sh;
};</pre></td></tr>
</table>

Solution

Enter the `#pragma pack(n)` statement in the header file so that it applies to all source files that include the header.

Errors from Conflicts with Polyspace Header Files

Issue

You see compilation errors from header files included by Polyspace.

For instance, the error message refers to one of the subfolders of `matlabroot\polyspace\verifier\cxx\include`.

Typically, the error message is related to a standard library function.

Cause

If your compiler defines a standard library function or another construct and you do not provide the path to your compiler header files, Polyspace uses its own implementation of the function.

If your compiler definitions differ from the corresponding Polyspace definitions, the verification stops with an error.

Solution

Specify the folder containing your compiler header files.

- In the user interface, add the folder to your project.

  For more information, see "Add Source Files for Analysis in Polyspace User Interface" on page 1-2.
- At the command line, use the flag `-I` with the `polyspace-bug-finder-nodesktop` command.

  For more information, see `-I`.

For compilation with GNU C on UNIX-based platforms, use `/usr/include`. On embedded compilers, the header files are typically in a subfolder of the compiler installation folder. Examples of include folders are given for some compilers.

- Wind River Diab: For instance, `/apps/WindRiver/Diab/5.9.4/diab/5.9.4.8/include/`.

- IAR Embedded Workbench: For instance, `C:\Program Files\IAR Systems\Embedded Workbench 7.5\arm\inc`.
- Microsoft Visual Studio: For instance, `C:\Program Files\Microsoft Visual Studio 14.0\VC\include`.

Consult your compiler documentation for the path to your compiler header files. Alternatively, see “Provide Standard Library Headers for Polyspace Analysis” (Polyspace Code Prover).

Errors from Assertion or Memory Allocation Functions

Issue

Polyspace uses its own implementation of standard library functions for more efficient analysis. If you redefine a standard library function and provide the function body to Polyspace, the analysis uses your definition.

However, for certain standard library functions, Polyspace continues to use its own implementations, even if you redefine the function and provide the function body. The functions include `assert` and memory allocation functions such as `malloc`, `calloc` and `alloca`.

You see a warning message like the following:

```
Body of routine "malloc" was discarded.
```

Cause

These functions have special meaning for the Polyspace analysis, so you are not allowed to redefine them. For instance:

- The Polyspace implementation of the `malloc` function allows the software to check if memory allocated using `malloc` is freed later.
- The Polyspace implementation of `assert` is used internally to enhance analysis.

Solution

Unless you particularly want your own redefinitions to be used, ignore the warning. The analysis results are based on Polyspace implementations of the standard library function, which follow the original function specifications.

If you want your own redefinitions to be used and you are sure that your redefined function behaves the same as the original function, rename the functions. You can rename the function only for the purposes of analysis using the option `Preprocessor definitions (-D)`. For instance, to rename a function `malloc` to `my_malloc`, use `malloc=my_malloc` for the option argument.

Error from Special Characters

Issue

Your file or folder names contain extended ASCII characters, such as accented letters or Kanji characters. You face file access errors during analysis. Error messages you might see include:

- `No source files to analyze`
- `Control character not valid`
- `Cannot create directory` *`Folder_Name`*

Cause

Polyspace does not fully support these characters. If you use extended ASCII in your file or folder names, your Polyspace analysis may fail due to file access errors.

Workaround

Change the unsupported ASCII characters to standard US-ASCII characters.

Errors from In-Class Initialization (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:

<table>
<tr><th>in file Test.h</th><th>in file Test.cpp</th></tr>
<tr><td><pre>class Test
{
public:
static int m_number;
};</pre></td><td><pre>int Test::m_number = 0;</pre></td></tr>
</table>

Errors from Double Declarations of Standard Template Library Functions (C++)

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 (-no-stl-stubs)`.
- 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 analysis 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 (-D)`.

Errors Related to GNU Compiler

If you choose `gnu` for the option `Compiler (-compiler)`, you can encounter this issue.

Issue

The Polyspace analysis stops with a compilation error.

Cause

You are using certain advanced compiler-specific extensions that Polyspace does not support. See “Limitations”.

Solution

For easier portability of your code, avoid using the extensions.

If you want to use the extensions and still analyze your code, wrap the unsupported extensions in a preprocessor directive. For instance:

```
#ifdef POLYSPACE
    // Supported syntax
#else
    // Unsupported syntax
#endif
```

For regular compilation, do not define the macro `POLYSPACE`. For Polyspace analysis, enter `POLYSPACE` for the option `Preprocessor definitions (-D)`.

If the compilation error is related to assembly language code, see “Assembly Code” (Polyspace Code Prover).

Errors Related to Visual Compilers

The following messages appear if the compiler is based on a Visual compiler. For more information, see `Compiler (-compiler)`.

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.

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

#pragma pack(4)
 	
#include "type.h"
	struct A
	{
	    char c ;
	    int i ;
	} ;
		#pragma pack(2)
		
		#include "type.h"

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 (-ignore-pragma-pack)`.

C++/CLI

Polyspace does not support Microsoft C++/CLI, a set of language extensions for .NET programming.

You can get errors such as:

```
error: name must be a namespace name
|          using namespace System;
```

Or:

```
error: expected a declaration
|          public ref class Form1 : public System::Windows::Forms::Form
```

Eclipse Java Version Incompatible with Polyspace Plug-in

In this section...
"Issue" on page 19-62
"Cause" on page 19-62
"Solution" on page 19-62

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
matlabroot\sys\java\jre\arch\jre\bin\javaw.exe
```

Here, *`matlabroot`* is your product installation folder, for instance, `C:\MATLAB\R2015b\` and *`arch`* is `win32` or `win64` depending on the product platform.

Coding Rule Violations Not Displayed

Issue

You expect a coding rule violation on a line of code but the Polyspace analysis does not show the violation.

Possible Cause: Rule Checker Not Enabled

You might be looking for a reduced subset of coding rules.

For instance, if you check for MISRA C: 2012 rules, by default, the analysis looks for the `mandatory-required` subset only.

Solution

Check the coding rules options that you use. See:

- `Check MISRA C:2004 (-misra2)`
- `Check MISRA C:2012 (-misra3)`
- `Check MISRA C++ rules (-misra-cpp)`
- `Check JSF C++ rules (-jsf-coding-rules)`

Possible Cause: Rule Violations in Header Files

All coding rule violations in the file might be suppressed.

For instance, by default, coding rule violations are suppressed from header files that are not in the same location as the source files.

Solution

Check the files where you suppress coding rule violations. See `Do not generate results for (-do-not-generate-results-for)`.

Possible Cause: Rule Violations in Macros

The rule violation occurs in a macro expansion. To save you from reviewing the same violation multiple times, the violation is shown on the macro definition instead of the

macro usage. If the definition occurs in a header file, it might be suppressed from the results.

On the **Source** pane, you can tell if a line contains a macro expansion. Look for the M icon.

```
110 M     s8_ret = (s8) NA_VALUE;
```

Solution

Find the macro definition and see if it occurs in a header file. Determine if you are suppressing coding rule violations from header files. See `Do not generate results for (-do-not-generate-results-for)`.

Possible Cause: Compilation Errors

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

Check for compilation errors. See "View Error Information When Analysis Stops" on page 19-4.

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

Insufficient Memory During Report Generation

Message

```
....
Exporting views...
Initializing...
Polyspace Report Generator
Generating Report
 .....
    Converting report
Opening log file:  C:\Users\auser\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`*`\`.

Error from Disk Defragmentation and Antivirus Software

Issue

The analysis stops with an error message like the following:

```
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.

Errors with Temporary Files

Polyspace produces some temporary files during analysis. The following issues are related to storage of temporary files.

No Access Rights

When running verification, you get an error message that Polyspace could not create a folder for writing temporary files. For instance, the error message can be as follows:

```
Unable to create folder "C:\Temp\Polyspace\foldername
```

Cause

Polyspace produces some temporary files during analysis. If you do not have write permissions for the folder used to store the files, you can encounter the error.

Solution

There are two possible solutions to this error:

- Change the permissions of your temporary folder so you have full read and write privileges.

  To learn how Polyspace determines the temporary folder location, see "Storage of Temporary Files" on page 1-15.
- Use the option `-tmp-dir-in-results-dir`. Instead of the standard temporary folder, Polyspace uses a subfolder of the results folder.

No Space Left on Device

When running verification, you get an error message that there is no space on a device.

Cause

If you do not have sufficient space on for the folder used to store the files, you can encounter the error.

Solution

There are two possible solutions to this error:

- Change the temporary folder to a drive that has enough disk space.

  To learn how Polyspace determines the temporary folder location, see “Storage of Temporary Files” on page 1-15.
- Use the option `-tmp-dir-in-results-dir`. Instead of the standard temporary folder, Polyspace uses a subfolder of the results folder.

Cannot Open Temporary File

When running verification, you get an error message that Polyspace could not open a temporary file.

Cause

You defined the path for storing temporary files by using the environment variable `RTE_TMP_DIR`. You either used a relative path for the temporary folder, the folder does not exist or you do not have access rights to the folder.

Solution

There are two possible solutions to this error:

- Instead of defining a temporary folder specific to Polyspace through `RTE_TMP_DIR`, use a standard temporary folder.

  To learn how Polyspace determines the temporary folder location, see “Storage of Temporary Files” on page 1-15.
- If you continue to use `RTE_TMP_DIR`, make sure you specify an absolute path to an existing folder and you have access rights to the folder.