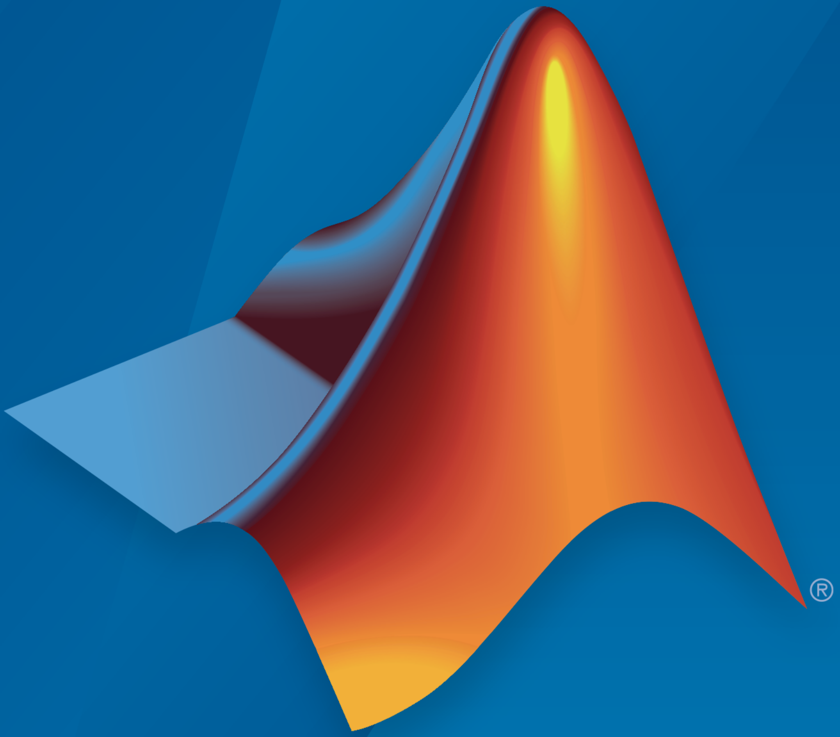
Polyspace® Bug Finder™

User's Guide



MATLAB® & SIMULINK®

R2017a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

Polyspace® Bug Finder™ User's Guide

Trademarks

Patents

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)

Contents

1 Project Configuration

2 Coding Rule Sets and Concepts

3 Check Coding Rules from the Polyspace Environment

4 Find Bugs From the Polyspace Environment

5 View Results in the Polyspace Environment

6 Command-Line Analysis

7 Polyspace Bug Finder Analysis in Simulink

8 Configure Model for Code Analysis

9 Configure Code Analysis Options

10 Run Polyspace on Generated Code

11 Check Coding Rules from Eclipse

12 Find Bugs from Eclipse

13 View Results in Eclipse

14 Troubleshooting in Polyspace Bug Finder

15 Software Quality with Polyspace Metrics

Project Configuration

Create New Project Manually

To analyze your sources files in the user interface, you must create a Polyspace® project. The project consists of your source files, include folders and one or more modules. You add all or some of your source files to a module, change the default analysis options if you want, and run analysis on the module.

If you do not use build automation scripts to build your source code, you can create a Polyspace project manually.

Otherwise, see "Create Project Automatically" (Polyspace Code Prover). If automatic project creation is not supported for your compiler, see the suggestions in "Requirements for Project Creation from Build Systems" (Polyspace Code Prover). If the suggestions do not work, create a project manually.

Tip: In the Polyspace user interface, you can quickly change to an arrangement of panes dedicated to project setup. Select **Window** > **Reset Layout** > **Project Setup**.

Create Project

When creating a new project, you must know:

- Location of source files
- Location of include files
- Location where you want to store analysis results

1. Select **File** > **New Project**.
2. In the Project – Properties window, specify properties for your project:

   - **Project name**
   - **Location**: Folder where you will store the project file (`.psprj` file) and the results unless you specify otherwise. You can use the `.psprj` file to reopen the project.

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

- Clear the **Use template** check box unless you have a template you want to use.

3 On the next screen, add source folders to your project.

   a Use the **Browse** button to navigate to the folder containing the source files you want to analyze.

   By default, Polyspace looks for `.c`, `.cpp`, `.cxx`, or `.cc` files. If you use other file extensions, before closing the dialog box, change the **Files of types** option.

   b If you chose a source folder with subfolders but do not want to analyze files in the subfolders, clear the check box **Add recursively**.

   c (Linux® only) Often, compilers add symbolic links in your source folders during compilation. If your folder contains symbolic links to other folders but you do not want to add source files from the other folders, select **Exclude symbolic links**.

   d Click **Add Source Folder**. All source files found under this folder are added to your Polyspace project.

   Tip: To see the full path of your files, toggle the ☰ button.

   ---

   e If you do not want to analyze all the files under your source folder, 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**.

4 On the next screen, add include folders to your project.

   a Use the **Browse** button to navigate to your folder containing the include files needed for compilation.

   By default, Polyspace looks for `.h`, `.hpp`, or `.hxx` files. If you use other file extensions, before closing the dialog box, change the **Files of types** option.

   b If you chose an include folder that contains subfolder and you want to add those include folders as well, select the check box **Include all subfolders**.

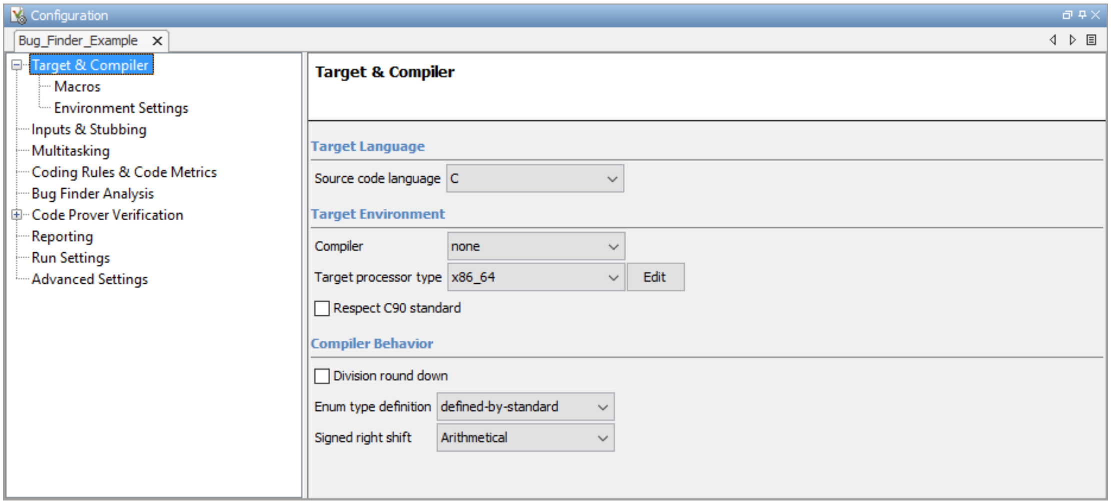
   c (Linux only) Often, compilers add symbolic links in your folders during compilation. If your folder contains symbolic links to other folders but you do not want to add includes from the other folders, select **Exclude symbolic links**.

   d Click **Add Include Folders**. The include folder is added to your Polyspace project.

Specify Analysis Options

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

1. On the **Project Browser**, select the configuration file .
2. Change the options on the **Configuration** pane.



Source code language: C

Compiler: none

Target processor type: x86_64

☐ Respect C90 standard

☐ Division round down

Enum type definition: defined-by-standard

Signed right shift: Arithmetical

Some options to consider looking at are:

- **Target & Compiler** > **Compiler**, enables different language extensions.
- **Target & Compiler** > **Target processor type**, sets the size of your data types for the analysis.
- **Macros** > **Preprocessor definitions**, a location to enter your compilation flags.
- **Multitasking** options, for analyzing multitasking code.
- **Bug Finder Analysis** options, to change which defects Polyspace checks for.
- **Coding Rules & Code Metrics** options, to check for predefined coding rules or calculate metrics about your project.

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

If you specify an option multiple times, only the last setting is used. For example, in the user interface, on the Target and Compiler pane you can specify the target as **c18** and in the **Advanced options** box enter `-target i386`. These two targets count as multiple analysis option specifications. Polyspace uses the target specified in the Advanced options dialog box, `i386`.

For more information on the options, see “Analysis Options”.

Related Examples

- “Update Project” on page 1-15
- “Create Project Automatically” on page 1-6

Create Project Automatically

If you use build automation scripts to build your source code, you can automatically setup a Polyspace project from your scripts. The automatic project setup runs your automation scripts to determine:

- Source files.
- Includes.
- Target & compiler options. For more information on these options, see "Target & Compiler".

1. Select **File** > **New Project**.
2. On the Project – Properties dialog box, after specifying the project name and location, under **Project configuration**, select **Create from build command**.
3. On the next window, enter the following information:

Field	Description
Specify command used for building your source files	If you use an IDE such as Visual Studio® or Eclipse™ to build your project, specify the full path to your IDE executable or navigate to it using the 📁 button. For a tutorial using Visual Studio, see "Create Project Using Visual Studio Information" on page 1-12. **Example:** `"C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\VCExpress.exe"` If you use command-line tools to build your project, specify the appropriate command. **Example:** • `make -B -f` *`makefileName`* or `make -W` *`makefileName`* • `"mingw32-make.exe -B -f makefilename"`
Specify working directory for running build command	Specify the folder from which you run your build automation script.

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

4. Click ▷ Run.

   - If you entered your build command, Polyspace runs the command and sets up a project.
   - If you entered the path to an executable, the executable runs. Build your source code and close the executable. Polyspace traces your build and sets up a project.

     For example, in Visual Studio 2010, use **Tools** > **Rebuild Solution** to build your source code. Then close Visual Studio.

   If a failure occurs, see if your build command meets the requirements for automatic project setup. In some cases, you can modify your build command to work around the limitations. For more information, see “Requirements for Project Creation from Build Systems” on page 1-9.

5. Click **Finish**.

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

6. If you updated your build command, you can recreate the Polyspace project from the updated command. To recreate an existing project, on the **Project Browser**, right-click the project name and select **Update Project**.

Note:

- In the Polyspace interface, it is possible that the created project will not show implicit defines or includes. The configuration tool does include them. However, they are not visible.
- By default, Polyspace assigns the latest version of the compiler to your project. If you have compilation errors in your project, check the setting for Compiler (-compiler). If it does not apply to you, change it to a more appropriate one.

For instance, if the compiler setting is `visual12` but you are using Microsoft® Visual C++® 2010, change the setting to `visual10`.

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

Related Examples

- "Create Project Using Visual Studio Information" on page 1-12

More About

- "Compiler Not Supported for Project Creation from Build Systems" on page 14-48
- "Slow Build Process When Polyspace Traces the Build" on page 14-56
- "Check if Polyspace Supports Build Scripts" on page 14-57

Requirements for Project Creation from Build Systems

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

For more information on automatic project creation, see:

- "Create Project Automatically" on page 1-6
- "Create Project Automatically at Command Line" on page 6-2
- "Create Project Automatically from MATLAB Command Line" on page 6-23

The requirements for your build command are as follows:

- Your compiler must be called locally.

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

  If your compiler performs only an incremental build, use appropriate options to build all your source files. For example, if you use `gmake`, append the `-B` or `-W` *`makefileName`* option to force a clean build. For the list of options allowed with the GNU® `make`, see make options.
- Your compiler configuration must be available to Polyspace. The compilers currently supported include the following:

  - Visual C++ compiler
  - `gcc`
  - `clang`
  - `MinGW` compiler
  - `IAR` compiler

  If your compiler configuration is not available to Polyspace:

  - Write a compiler configuration file for your compiler in a specific format. For more information, see "Compiler Not Supported for Project Creation from Build Systems" on page 14-48.
  - Contact MathWorks Technical Support. For more information, see "Contact Technical Support" (Polyspace Code Prover).

- 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 14-57.
- 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.
- You cannot trace your build command on the operating system OS X El Capitan if the security feature System Integrity Protection (SIP) is active. Before tracing your build command, disable this 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.
- 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\reg*CPUNAME*.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`.

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

See Also

polyspaceConfigure

Related Examples

- “Create Project Automatically” on page 1-6

More About

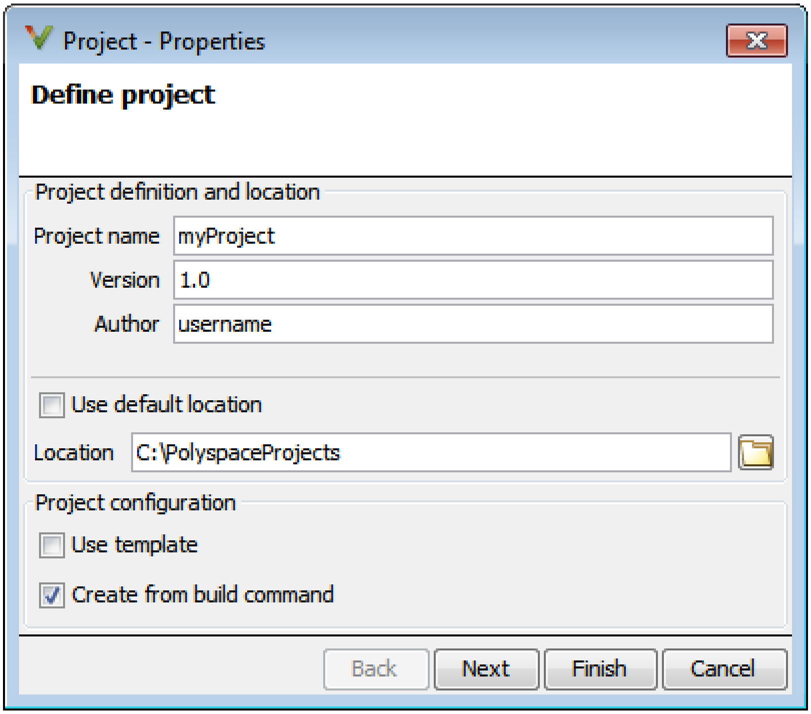
- “Slow Build Process When Polyspace Traces the Build” on page 14-56

Create Project Using Visual Studio Information

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

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

2 In the Project – Properties window, under **Project Configuration**, select **Create from build command** and click **Next**.



Project - Properties

Define project

Project definition and location

Project name: myProject

Version: 1.0

Author: username

□ Use default location

Location: C:\PolyspaceProjects

Project configuration

□ Use template

☑ Create from build command

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

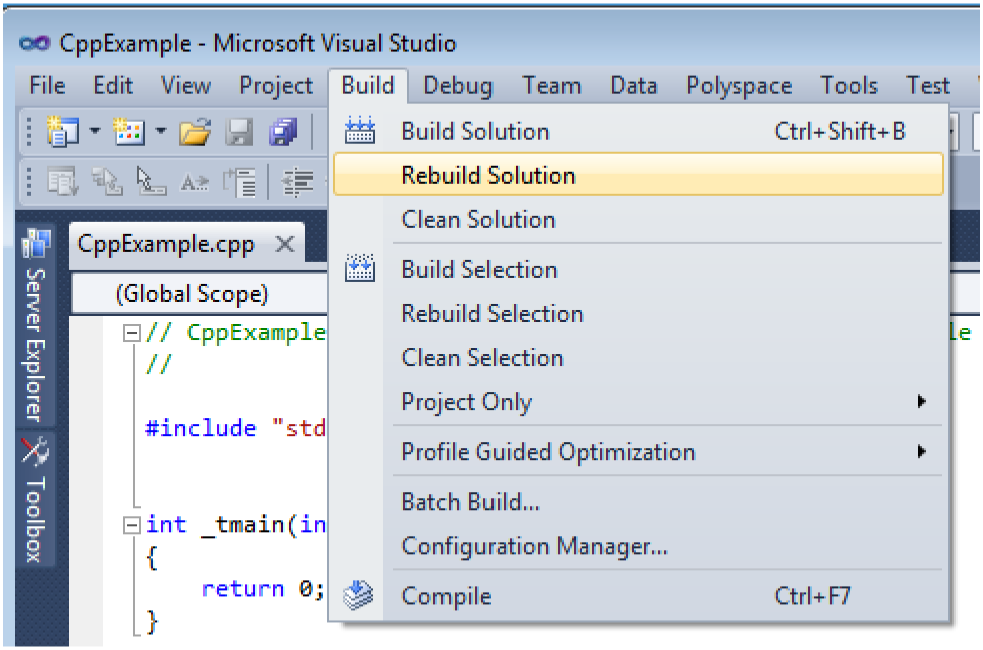
4 In the field **Specify working directory for running build command**, enter `C:\`.

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**.



```
// CppExample
//

#include "std


int _tmain(in
{
    return 0;
}
```

6 After the project builds, close Visual Studio.

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

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

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

More About

- “Troubleshooting Project Creation from Visual Studio Build” on page 14-46

Update Project

In this section…

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.

1. In the **Project Browser**, right-click the top sources folder 📂 and select **Modify Path**.
2. In the dialog box, in the text box, change the path to the new location.
3. Click **Save Changes**.
4. Click **Finish**.
5. 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.

1. 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:

1 On the **Project Browser** pane, right-click the project folder and select **Update Project**.

2 Enter the same information you did when creating the original project. For more information, see “Create Project Automatically” on page 1-6.

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:

1 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, follow these steps:

1 In the **Project Browser**, right-click your project or the **Source** or **Include** folder in your project.

2 Select **Add Source Folder** or **Add Include Folder**.

3 Add source folders to your project:

   a Use the **Browse** button to navigate to the folder containing the source files you want to analyze.

      By default, Polyspace looks for `.c`, `.cpp`, `.cxx`, or `.cc` files. If you use other file extensions, before closing the dialog box, change the **Files of types** option.

   b If you chose a source folder that contains subfolder and you do not want to analyze source files in those subfolders, clear the check box **Add recursively**.

   c (Linux only) Often, compilers add symbolic links in your source folders during compilation. If your folder contains symbolic links to other folders but you do not want to add source files from the other folders, select **Exclude symbolic links**.

   d Click **Add Source Folder**. All source files found under the folder are added to your Polyspace project.

Tip: To see the full path of your files, click the ☰ button.

   e. If you do not want to analyze all the files under your source folder, 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**.

   Repeat these steps as many times as necessary, then click **Next**.

4. Add include folders to your project.

   a. Use the **Browse** button to navigate to your folder containing the include files needed for compilation.

      By default, Polyspace looks for `.h`, `.hpp`, or `.hxx` files. If you use other file extensions, before closing the dialog box, change the **Files of types** option.

   b. If you chose an include folder that contains subfolder and you want to add those include folders as well, select the check box **Include all subfolders**.

   c. (Linux only) Often, compilers add symbolic links in your folders during compilation. If your folder contains symbolic links to other folders but you do not want to add includes from the other folders, select **Exclude symbolic links**.

   d. Click **Add Include Folders**. The include folder is added to your Polyspace project.

   Repeat these steps as many times as necessary, then click **Finish**. The new project opens in the **Project Browser** pane.

5. Click **Finish**.

6. Before running an analysis, you must copy the source files to a module.

   a. Select the source files that you want to copy. To select multiple files together, press the **Ctrl** key while selecting the files.

   b. Right-click your selection.

   c. Select **Copy to** > **Module_*n***. *n* is the module number.

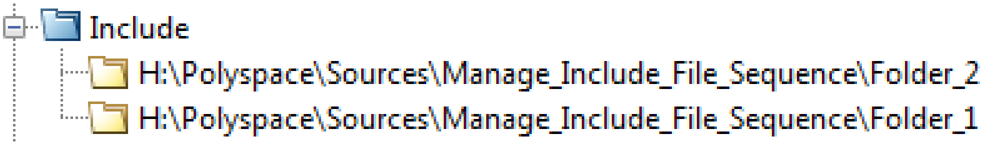
Manage Include File Sequence

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`.



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

To change the order of include folders:

1 In your project, expand the **Include** folder.
2 Select the include folder or folders that you want to move.
3 To move the folder, click either ⇧ or ⇩.

Related Examples

- "Specify Results Folder" on page 4-6
- "Create New Project Manually" on page 1-2

Specify Analysis Options

In this section...

About Analysis Options

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

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

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

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

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

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

Gather Compilation Options Efficiently

The code is often tuned for the target (as discussed in "Analyze Keil or IAR Compiled Code" on page 1-58). Rather than applying minor changes to the code, create a single `polyspace.h` file which contains target specific functions and options. The `-include` option can then be used to force the inclusion of the `polyspace.h` file in the source files.

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

Direct benefits:

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

Indirect benefits:

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

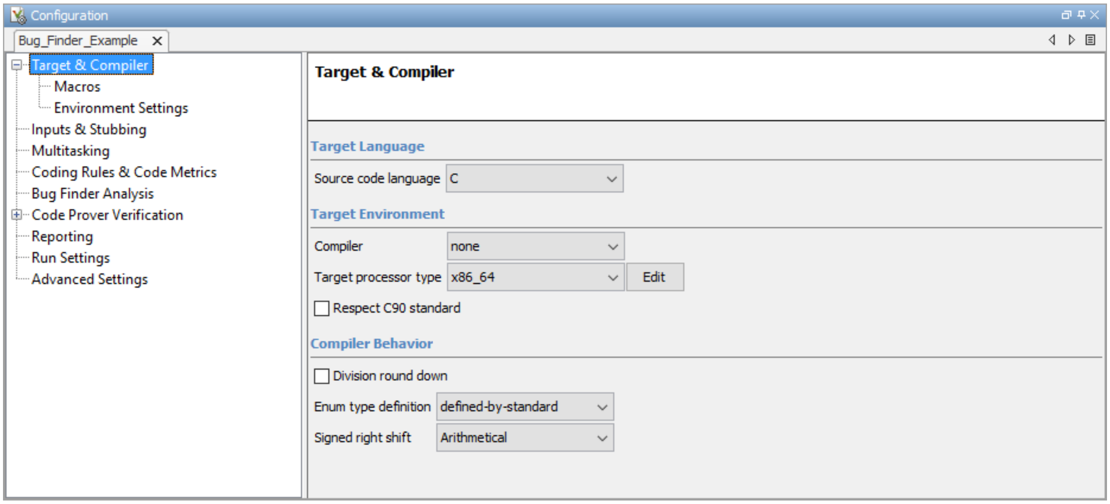
Example

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

```
/* Standard Includes Used by Cross Compiler */
#include <stdlib.h>
#include "another_file.h"

/* Generic definitions, reusable from one project to another /*
#define far
#define at(x)

/* Function prototype to detect declaration mismatches earlier */
void f(int);

/* Variable prototype to detect declaration mismatches earlier */
extern int x;
```

Specify Options in User Interface

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



Target & Compiler

Target Language

Source code language: C

Target Environment

Compiler: none

Target processor type: x86_64

□ Respect C90 standard

Compiler Behavior

□ Division round down

Enum type definition: defined-by-standard

Signed right shift: Arithmetical

For instance:

- To specify the target processor, select **Target & Compiler** in the **Configuration** tree view. Select your processor from the **Target processor type** dropdown list.
- To check for violation of MISRA C$^{®}$ rules, select **Coding Rules**. Check the **Check MISRA C Rules** box. To check for a subset of rules, select an option from the dropdown list.

See Also

polyspaceBugFinder

Related Examples

- “Create Project Using Configuration Template” on page 1-22

More About

- “Analysis Options”

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

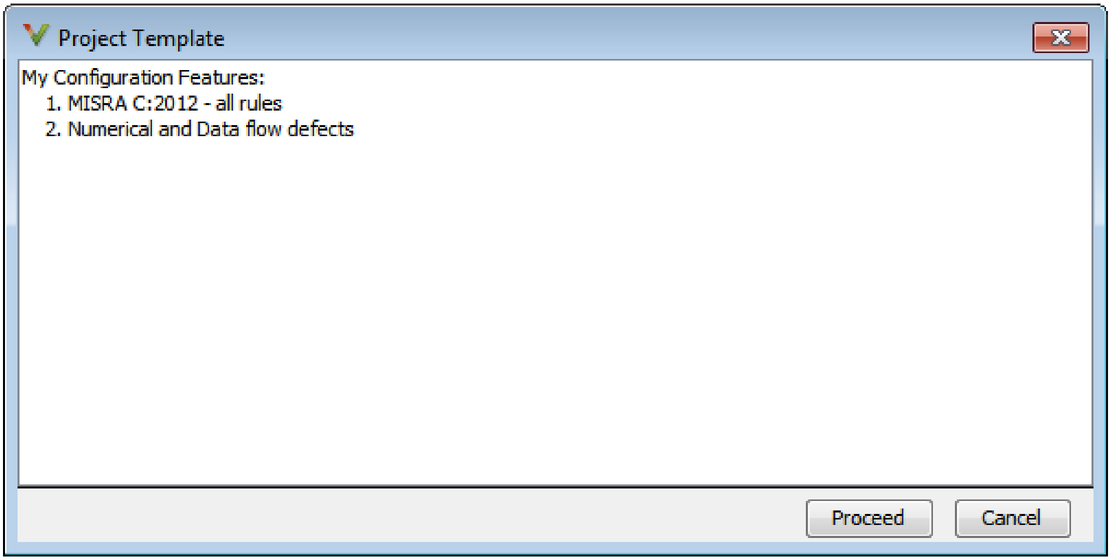
1. Select **File** > **New Project**.
2. On the Project – Properties dialog box, after specifying the project name and location, under **Project configuration**, select **Use template**.
3. On the next screen, select the template that corresponds to your compiler. For further details on a template, select 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. For more information, see “Create New Project Manually” on page 1-2.

Create Template

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

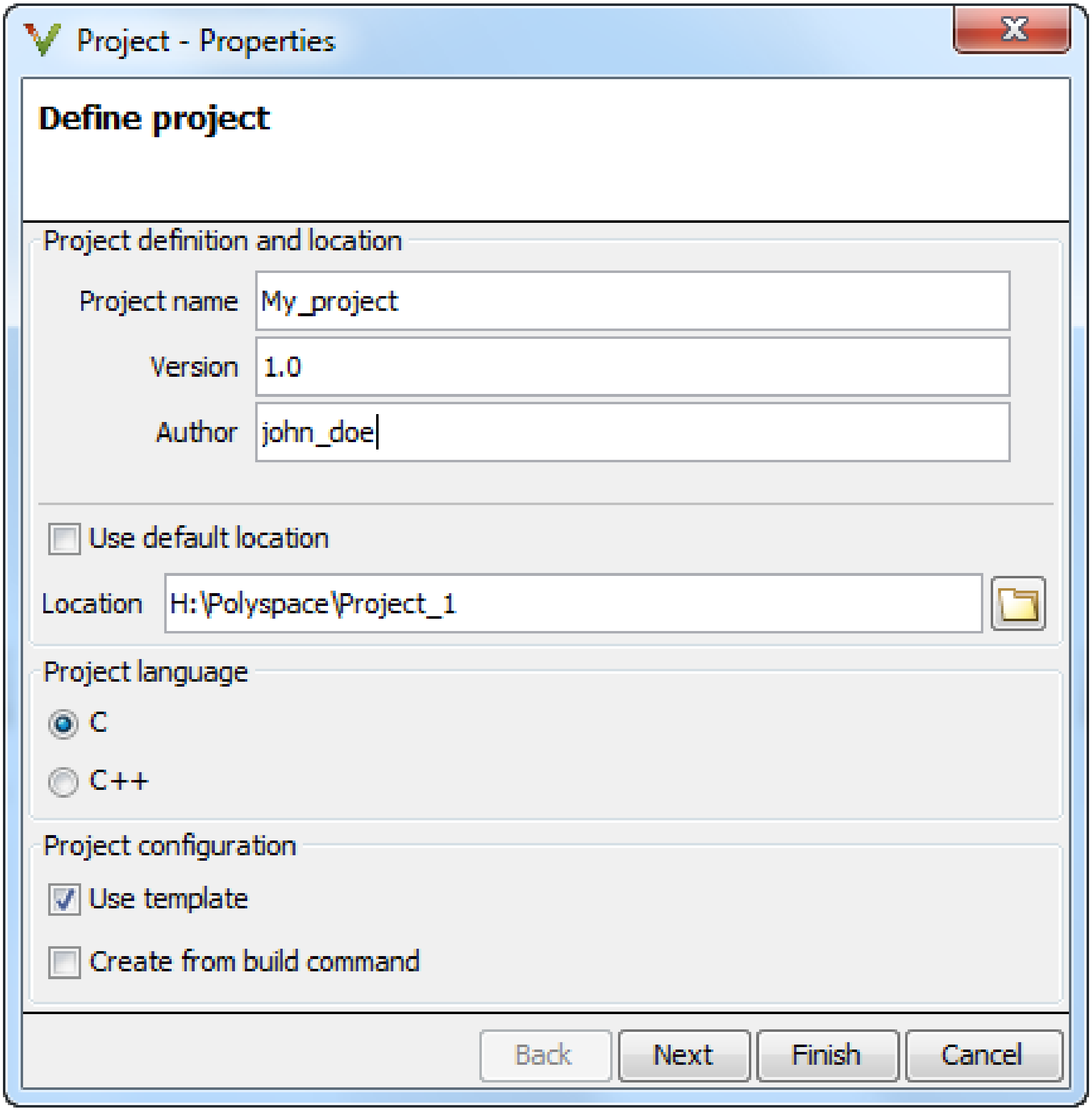
  1. Right-click the configuration that you want to use, and then select **Save As Template**.
  2. Enter a description for the template, then click **Proceed**. Save your Template file.



My Configuration Features:
1. MISRA C:2012 - all rules
2. Numerical and Data flow defects

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

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



Project - Properties

Define project

Project definition and location

Project name: My_project

Version: 1.0

Author: john_doe

□ Use default location

Location: H:\Polyspace\Project_1

Project language

● C

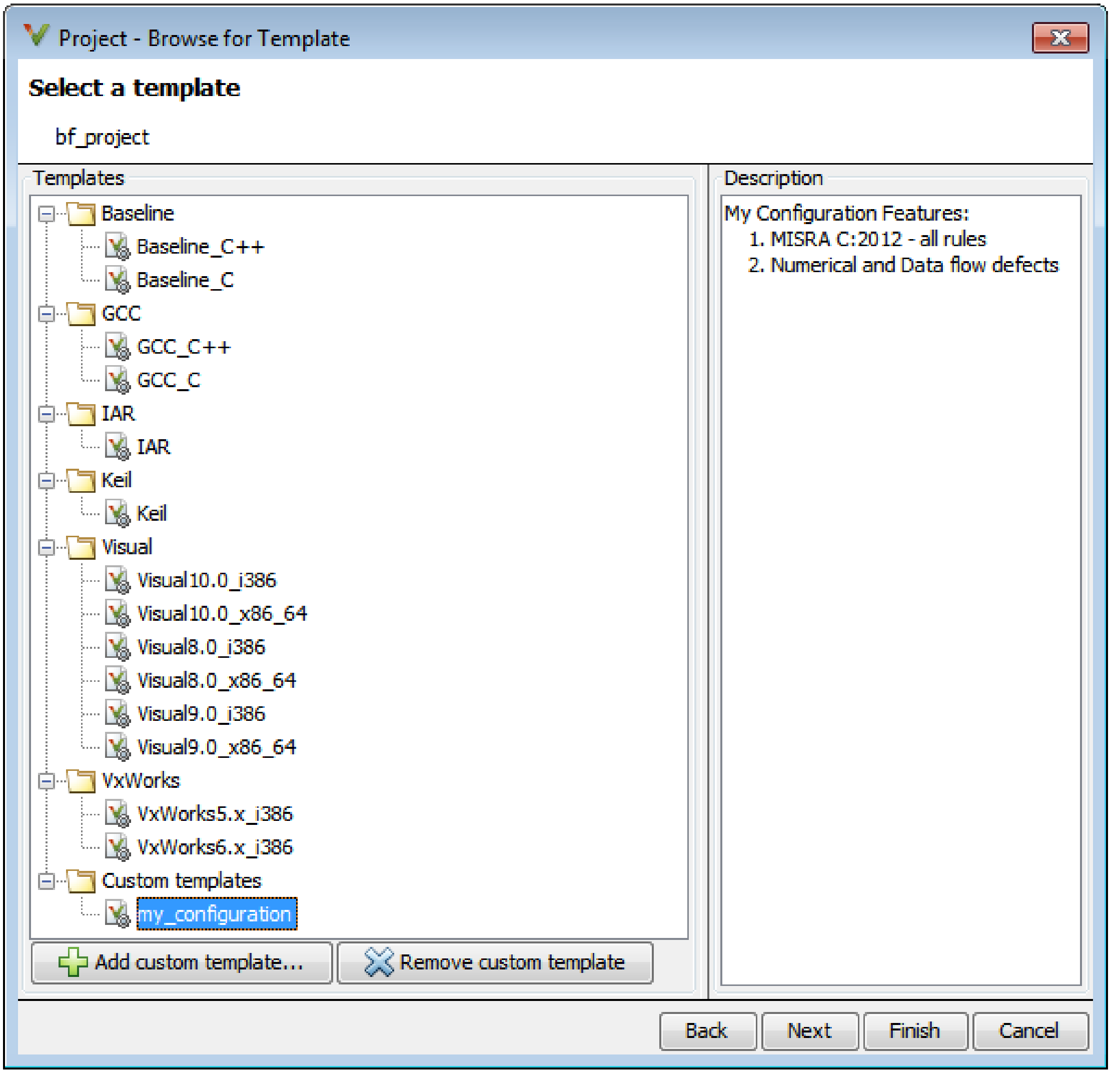
○ C++

Project configuration

☑ Use template

□ Create from build command

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



Project - Browse for Template

Select a template

bf_project

Description

My Configuration Features:
1. MISRA C:2012 - all rules
2. Numerical and Data flow defects

Related Examples

- "Specify Analysis Options" on page 1-19

More About

- "Analysis Options"

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.

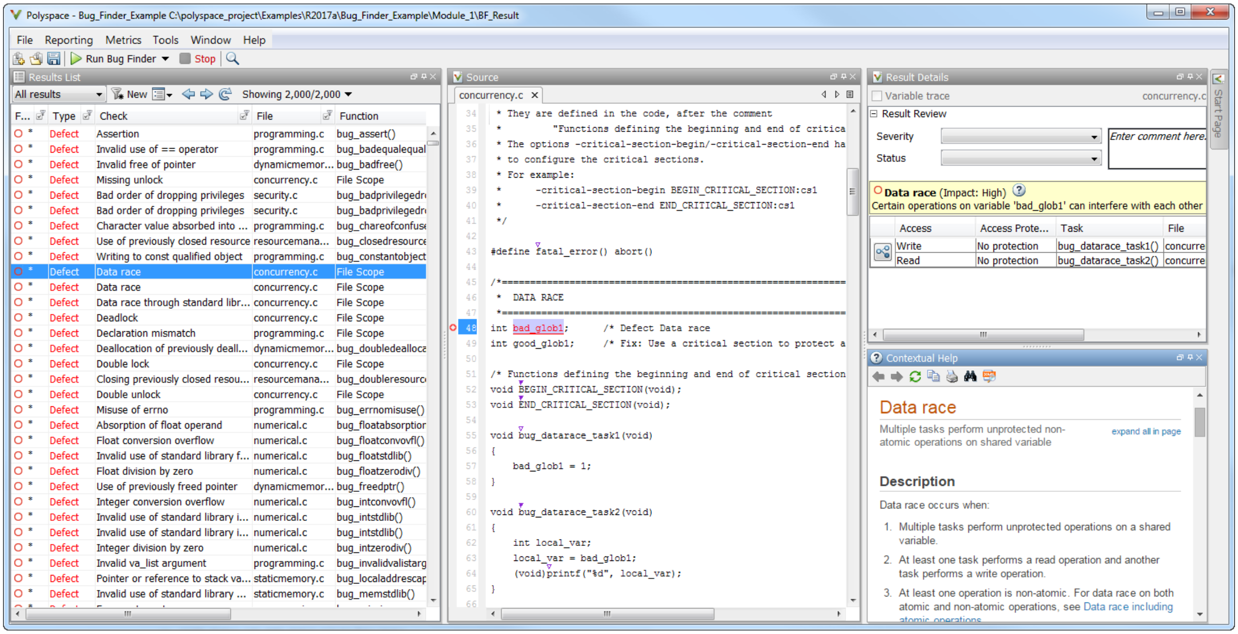
  For instance, if **Project Browser** and **Results List** are tabbed on the same pane, dragging out **Project Browser** creates a pane with only **Project Browser** on it and floats this new pane.

- 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.



F...	Type	Check	File	Function
○ *	Defect	Assertion	programming.c	bug_assert()
○ *	Defect	Invalid use of == operator	programming.c	bug_badequalequal
○ *	Defect	Invalid free of pointer	dynamicmemor...	bug_badfree()
○ *	Defect	Missing unlock	concurrency.c	File Scope
○ *	Defect	Bad order of dropping privileges	security.c	bug_badprivilegedr
○ *	Defect	Bad order of dropping privileges	security.c	bug_badprivilegedr
○ *	Defect	Character value absorbed into ...	programming.c	bug_chareofconfuse
○ *	Defect	Use of previously closed resource	resourcemana...	bug_closedresource
○ *	Defect	Writing to const qualified object	programming.c	bug_constantobject
○ *	Defect	Data race	concurrency.c	File Scope
○ *	Defect	Data race	concurrency.c	File Scope
○ *	Defect	Data race through standard libr...	concurrency.c	File Scope
○ *	Defect	Deadlock	concurrency.c	File Scope
○ *	Defect	Declaration mismatch	programming.c	File Scope
○ *	Defect	Deallocation of previously deall...	dynamicmemor...	bug_doubledealloca
○ *	Defect	Double lock	concurrency.c	File Scope
○ *	Defect	Closing previously closed resou...	resourcemana...	bug_doubleresourc
○ *	Defect	Double unlock	concurrency.c	File Scope
○ *	Defect	Misuse of errno	programming.c	bug_errnomisuse()
○ *	Defect	Absorption of float operand	numerical.c	bug_floatabsorptior
○ *	Defect	Float conversion overflow	numerical.c	bug_floatconvovfl()
○ *	Defect	Invalid use of standard library f...	numerical.c	bug_floatstdlib()
○ *	Defect	Float division by zero	numerical.c	bug_floatzerodiv()
○ *	Defect	Use of previously freed pointer	dynamicmemor...	bug_freedptr()
○ *	Defect	Integer conversion overflow	numerical.c	bug_intconvovfl()
○ *	Defect	Invalid use of standard library i...	numerical.c	bug_intstdlib()
○ *	Defect	Invalid use of standard library i...	numerical.c	bug_intstdlib()
○ *	Defect	Integer division by zero	numerical.c	bug_intzerodiv()
○ *	Defect	Invalid va_list argument	programming.c	bug_invalidvalistarg
○ *	Defect	Pointer or reference to stack va...	staticmemory.c	bug_localaddrescap
○ *	Defect	Invalid use of standard library ...	staticmemory.c	bug_memstdlib()

```
 * They are defined in the code, after the comment
 *         "Functions defining the beginning and end of critica
 * The options -critical-section-begin/-critical-section-end ha
 * to configure the critical sections.
 * For example:
 *      -critical-section-begin BEGIN_CRITICAL_SECTION:cs1
 *      -critical-section-end END_CRITICAL_SECTION:cs1
 */

#define fatal_error() abort()

/*=================================================================
 *  DATA RACE
 *=================================================================
int bad_glob1;       /* Defect Data race
int good_glob1;      /* Fix: Use a critical section to protect a

/* Functions defining the beginning and end of critical section
void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void bug_datarace_task1(void)
{
    bad_glob1 = 1;
}

void bug_datarace_task2(void)
{
    int local_var;
    local_var = bad_glob1;
    (void)printf("%d", local_var);
}
```

Result Review

Severity:

Status:

Enter comment here.

○ **Data race** (Impact: High)
Certain operations on variable 'bad_glob1' can interfere with each other

	Access	Access Prote...	Task	File
	Write	No protection	bug_datarace_task1()	concurre
	Read	No protection	bug_datarace_task2()	concurre

Data race

Multiple tasks perform unprotected non-atomic operations on shared variable

expand all in page

Description

Data race occurs when:

1. Multiple tasks perform unprotected operations on a shared variable.
2. At least one task performs a read operation and another task performs a write operation.
3. At least one operation is non-atomic. For data race on both atomic and non-atomic operations, see Data race including atomic operations

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***.

Specify External Text Editor

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

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

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

5. To make sure that your source code opens at the correct line and column in your text editor, specify command-line arguments for the editor using Polyspace macros, `$FILE`, `$LINE` and `$COLUMN`. Once you specify the arguments, when you right-click a check on the **Results 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 the following editors:

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

   If you are using one of these editors, select it from the **Arguments** drop-down list.

   If you are using another text editor, select `Custom` from the drop-down list, and enter the command-line options in the field provided.

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

   a. In the **Text Editor** field, enter `/usr/bin/xterm`.

   b. From the **Arguments** drop-down list, select `Custom`.
   c. In the field to the right, enter `-e /usr/bin/vi $FILE`.

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

Change Default Font Size

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

1 Select **Tools** > **Preferences**.

2 On the **Miscellaneous** tab:

   - To increase the font size of labels on the user interface, select a value for **GUI font size**.

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

3 Click **OK**.

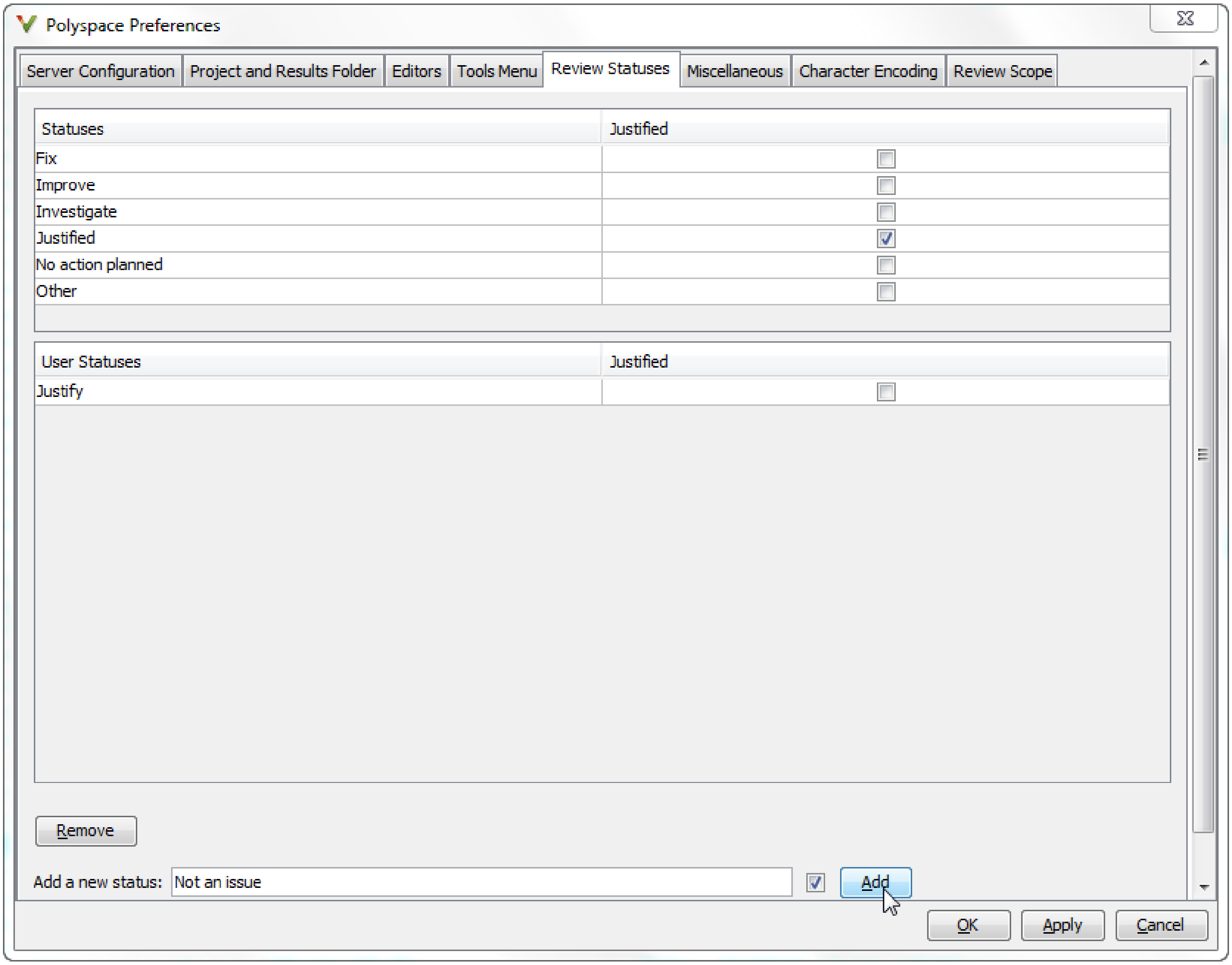
   When you restart Polyspace, you see the increased font size.

Define Custom Review Status

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

Define Custom Status

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



Statuses	Justified
Fix	☐
Improve	☐
Investigate	☐
Justified	☑
No action planned	☐
Other	☐

User Statuses	Justified
Justify	☐

Add a new status: Not an issue ☑

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

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

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

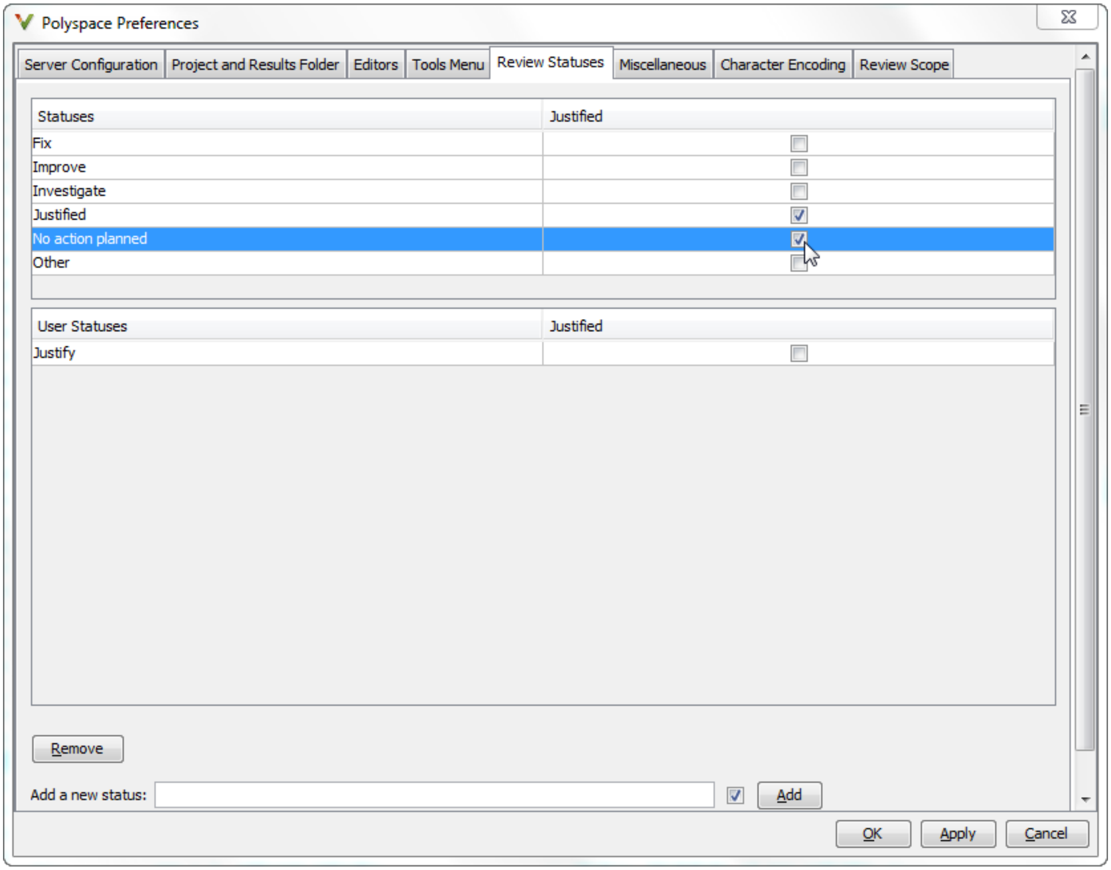
Add Justification to Existing Status

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

To add justification to existing status `Improve`:

1 Select **Tools** > **Preferences**.

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



Statuses	Justified
Fix	□
Improve	□
Investigate	□
Justified	☑
No action planned	☑
Other	□

User Statuses	Justified
Justify	□

Add a new status: ☑

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

Modeling Multitasking Code

In this section...
"Example" on page 1-37 "Limitations" on page 1-40

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

If you use POSIX®, VxWorks®, Windows, µC/OS II or C++11 functions for multitasking, the analysis automatically detects your multitasking model from your code.

The supported functions are the following:

Family	Thread Creation	Critical Section Begins	Critical Section Ends
POSIX	`pthread_create`	`pthread_mutex_lock`	`pthread_mutex_unlock`
VxWorks	`taskSpawn`	`semTake`	`semGive`
Windows	`CreateThread`	`EnterCriticalSection`	`LeaveCriticalSection`
µC/OS II	`OSTaskCreate`	`OSMutexPend`	`OSMutexPost`
C++11	`std::thread::thread`	`std::mutex::lock`	`std::mutex::unlock`

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 1-22.

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 "Set Up Multitasking Analysis Manually" on page 1-41.

Note: There are some aspects of multitasking that Polyspace cannot model. See "Limitations" on page 1-40.

Example

The following multitasking code models five philosophers sharing five forks.

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

pthread_mutex_t forks[4];

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

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

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

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

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

int main(void)
{
  pthread_t ph[5];
  pthread_create(&ph[0],NULL,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.

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

To run this example in Polyspace Bug Finder:

1 Copy this code into a `.c` file.

2. Create a Polyspace Bug Finder project with that `.c` file.
3. Run the analysis.

Limitations

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

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

See Also

Configure multitasking manually | Entry points (-entry-points) | Critical section details (-critical-section-begin -critical-section-end) | Temporally exclusive tasks (-temporal-exclusions-file) | Find defects (-checkers)

Related Examples

- "Set Up Multitasking Analysis Manually" on page 1-41

More About

- "Concurrency" on page 5-56

Set Up Multitasking Analysis Manually

In this section...

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

Polyspace Bug Finder automatically sets up the multitasking configuration for some types of multitasking functions. For information about the supported concurrency functions, see “Modeling Multitasking Code” on page 1-36.

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

Prerequisites

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

```
int a;

begin_critical_section();
end_critical_section();

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

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

```
void task2(void) {
    performTaskCycle();
}

void task3() {
    a=0;
}
```

Set Up Multitasking Analysis in User Interface

1. Specify your entry points and protection mechanisms.

   a. On the **Configuration** pane, select the **Multitasking** node.
   b. Select **Configure multitasking manually**.
   c. For **Cyclic tasks**, specify `task1`, `task2`, and `task3`, each on its own line.
   d. For **Critical section details**, specify `begin_critical_section` as **Starting routine** and `end_critical_section` as **Ending routine**.
   e. For **Temporally exclusive tasks**, specify `task1 task3` and `task2 task3`, each on its own line.

2. Specify the concurrency defects that you want Polyspace Bug Finder to detect. For more information, see “Concurrency Defects”.

   a. On the **Configuration** pane, select the **Bug Finder Analysis** node.
   b. From the **Find defects** list, select `custom`.
   c. Under the **Concurrency** node, select **Data race** and **Deadlock**.

Set Up Multitasking Analysis at Command Line

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

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

Set Up Multitasking Analysis at MATLAB Command Line

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

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

See Also

Configure multitasking manually | Entry points (-entry-points) | Critical section details (-critical-section-begin -critical-section-end) | Temporally exclusive tasks (-temporal-exclusions-file) | Find defects (-checkers)

More About

- “Concurrency” on page 5-56
- “Modeling Multitasking Code” on page 1-36

Annotate Code for Known or Acceptable Results

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 Rules 2.2 and 2.3
- 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.

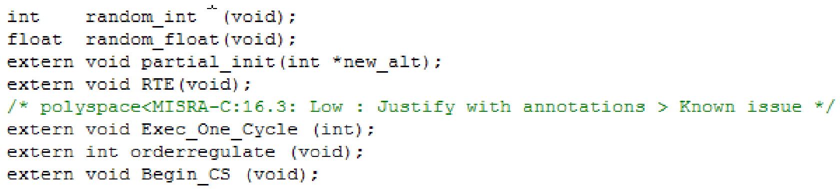
   c In the **Comment** field, enter a comment or keyword that helps you easily recognize the result.

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.



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

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 5-4.

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] >
    [Additional comments] */
```

- For a section of code, use the following syntax. (Polyspace Code Prover™ ignores this type of code annotation.)

```
/* polyspace:begin<Type:Kind1[,Kind2] : [Severity] : [Status] >
    [Additional text] */

... 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 1-47.

Word	Allowed Values
Type	The type of results: • Defect • 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, use specific defect abbreviations such as MEM_LEAK, FREED_PTR. See the full list of defects and abbreviations on page 6-10. For coding rule violations, specify the rule number or numbers. If you want the comment to apply to all possible defects or coding rules, specify ALL.
Severity	• Unset • High • Medium • Low • Not a defect
Status	Action for correcting the defect in your code. Possible values are: • Fix

Word	Allowed Values
	• `Improve` • `Investigate` • `Justified` • `No action planned` • `Other`
`Additional text`	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:Investigate>
    Known issue, why is this buffer hard coded?
int table[100];
```

- A MISRA C:2012 rule violation:

```
/* polyspace<MISRA-C3: 13.1 : Low : Justified> Known issue */
int arr[2] = {x++,y};
```

- Multiple defects:

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

- Multiple JSF® rule violations:

```
polyspace<JSF:9,13:Low:Justified> Known issue
```

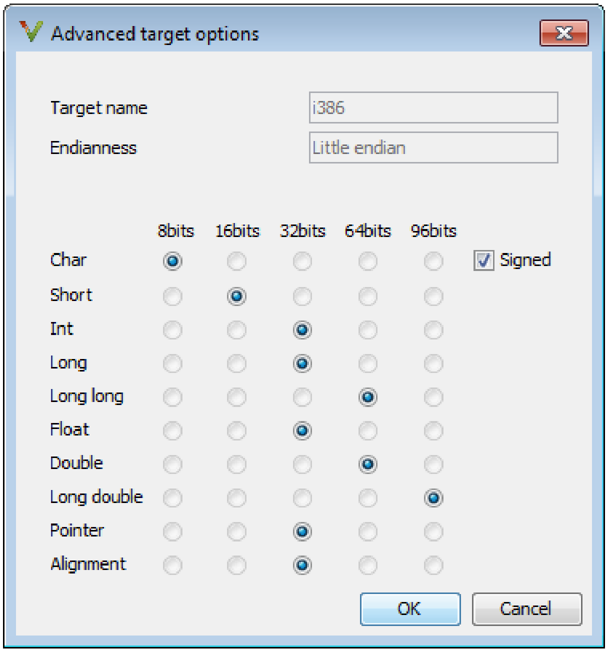
Modify Predefined Target Processor Attributes

You can modify certain attributes of the predefined target processors. If your specific processor is not listed, you may be able to specify a similar processor and modify its characteristics to match your processor. The settings that you can modify for each target are shown in [brackets] in the target processor settings. See Target processor type (-target).

To modify target processor attributes:

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

   The Advanced target options dialog box opens.



Target name: i386

Endianness: Little endian

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

4. Modify the attributes as required.

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

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

Specify Generic Target Processors

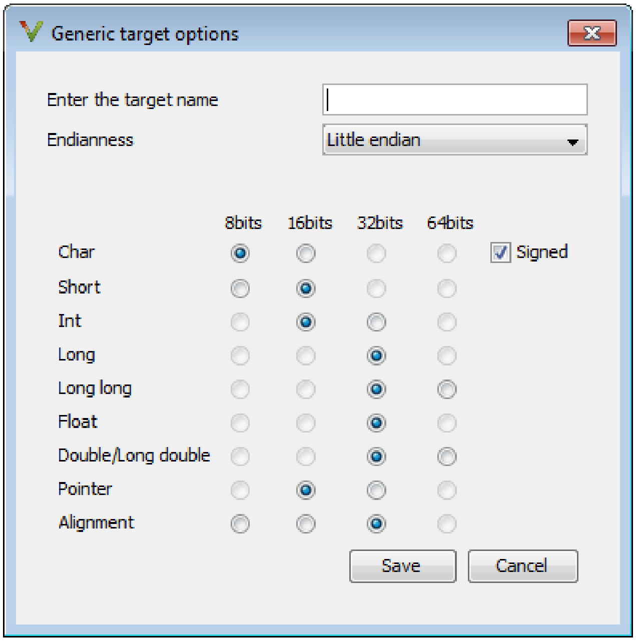
Define Generic Target

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

To configure a generic target:

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

   The Generic target options dialog box opens.



Enter the target name

Endianness: Little endian

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

3. In the **Enter the target name** field, enter a name, for example, `MyTarget`.
4. Specify the parameters for your target, such as the size of basic types, and alignment with arrays and structures.

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

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

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

Common Generic Targets

The following tables describe the characteristics of common generic targets.

ST7 (Hiware C compiler : HiCross for ST7)

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

ST9 (GNU C compiler : gcc9 for ST9)

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

Hitachi H8/300, H8/300L

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

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

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

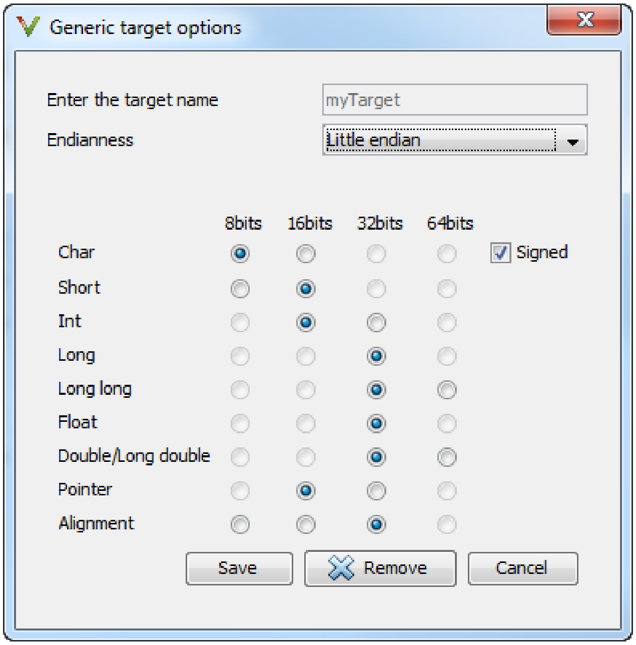
View or Modify Existing Generic Targets

To view or modify generic targets that you previously created:

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

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

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



Enter the target name: myTarget

Endianness: Little endian

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

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

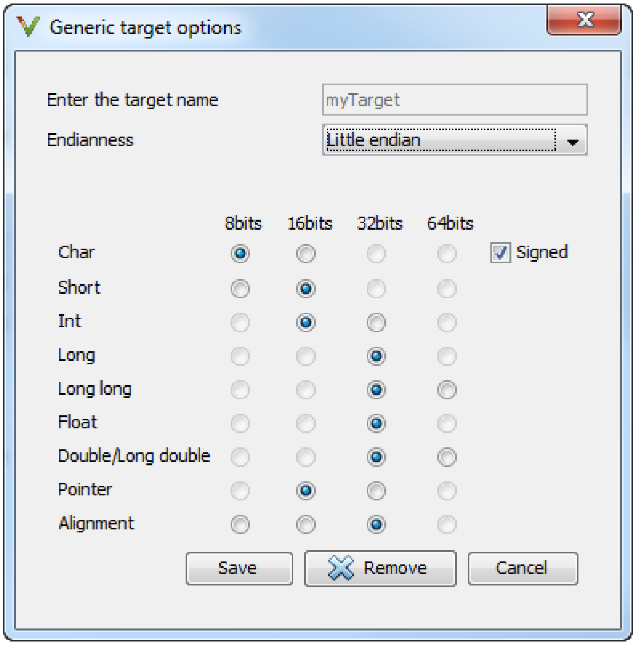
5 Otherwise, click **Cancel**.

Delete Generic Target

To delete a generic target:

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

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



Enter the target name: myTarget

Endianness: Little endian

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

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

Address Alignment

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

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

Consider the example:

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

- Each field must be aligned; that is, the starting offset of a field must be a multiple of its own size[1]
- So in the example, `char a` begins at offset 0 and its size is 8 bits. `int b` cannot begin at 8 (the end of the previous field) because the starting offset must be a multiple of its own size (32 bits). Consequently, `int b` begins at offset=32. The size of the `struct foo` before global alignment is therefore 64 bits.
- The global alignment of a structure is the maximum of the individual alignments of each of its fields;
- In the example, `global_alignment = max (alignment char a, alignment int b) = max (8, 32) = 32`
- The size of a struct must be a multiple of its global alignment. In our case, `b` begins at 32 and is 32 long, and the size of the struct (64) is a multiple of the `global_alignment` (32), so `sizeof` is not adjusted.

[1] except in the cases of “double” and “long” on some targets.

Ignore or Replace Keywords Before Compilation

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

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

For more information, see Command/script to apply to preprocessed files (-post-preprocessing-command).

Content of myTpl.pl file

```
#!/usr/bin/perl

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

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

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

Perl Regular Expression Summary

```
#########################################################
# Metacharacter What it matches
#########################################################
# Single Characters
# . Any character except newline
# [a-z0-9] Any single character in the set
# [^a-z0-9] Any character not in set
# \d A digit same as
# \D A non digit same as [^0-9]
# \w An Alphanumeric (word) character
# \W Non Alphanumeric (non-word) character
#
# Whitespace Characters
# \s Whitespace character
# \S Non-whitespace character
# \n newline
# \r return
# \t tab
# \f formfeed
# \b backspace
#
```

```
# Anchored Characters
# \B word boundary when no inside []
# \B non-word boundary
# ^ Matches to beginning of line
# $ Matches to end of line
#
# Repeated Characters
# x? 0 or 1 occurrence of x
# x* 0 or more x's
# x+ 1 or more x's
# x{m,n} Matches at least m x's and no more than n x's
# abc Exactly "abc"
# to|be|great One of "to", "be" or "great"
#
# Remembered Characters
# (string) Used for back referencing see below
# \1 or $1 First set of parentheses
# \2 or $2 First second of parentheses
# \3 or $3 First third of parentheses
##########################################################
# Back referencing
#
# e.g. swap first two words around on a line
# red cat -> cat red
# s/(\w+) (\w+)/$2 $1/;
#
##########################################################
```

Analyze Keil or IAR Compiled Code

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

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

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

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

```
int status,P0;

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

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

The following table summarizes the supported Keil C language extensions:

Example: `-compiler keil -sfr-types sfr=8`

Type/Language	Description	Example	Restrictions

Type `bit`	• An expression to type bit gives values in range [0,1].	`bit x = 0, y = 1,`
		`z = 2;`
		`assert(x == 0);`
		`assert(y == 1);`
		`assert(z == 1);`
			pointers to bits and arrays of bits are not allowed

Type/Language	Description	Example	Restrictions

	• Converting an expression in the type, gives 1 if it is not equal to 0, else 0. This behavior is similar to c ++ `bool`type.	`assert(sizeof(bit)`
		` == sizeof(int));`	

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

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

Type/Language	Description	Example	Restrictions

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

Interrupt functions	A warnings in the log file is displayed when an interrupt function has been found: "interrupt handler detected : <name>" or "task entry point detected : <name>"	`void foo1 (void)`
		`interrupt XX = YY`
		`using 99 {…}`
		`void foo2 (void) _`
		`task_ 99 _priority_`
		`2 {…}`
			Entry points and interrupts are not taken into account as `-entry-points`.

Keywords ignored	alien, bdata, far, idata, ebdata, huge, sdata, small, compact, large, reentrant. Defining -D __C51__, keywords large code, data, xdata, pdata and xhuge are ignored.		

The following table summarize the IAR compiler support:

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

Type/Language	Description	Example	Restrictions

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

Type/Language	Description	Example	Restrictions
	variable (mapped with a bit or a sfr type)		

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

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

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

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

Type/Language	Description	Example	Restrictions
	• A monitor function is a function that disables interrupts while it is executing, and then restores the previous interrupt state at function exit.		
Keywords ignored	`saddr, reentrant, reentrant_idata, non_banked, plm, bdata, idata, pdata, code, data, xdata, xhuge, interrupt, __interrupt and __intrinsic`		

Unnamed struct/ union	• Fields of unions/ structs without a tag or a name can be accessed without naming their parent struct.
	• On a conflict between a field of an anonymous struct with other identifiers:
	    • with a variable name, field name is hidden
	    • with a field of another anonymous struct at different scope, closer scope is chosen
	    • with a field of another anonymous struct at same scope: an error "anonymous struct field name <name> conflict“ is
		`union { int x; };`
		`union { int y; struct { int z; }; } @ 0xF0;`	

Type/Language	Description	Example	Restrictions
	displayed in the log file.		

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

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

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

For example:

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

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

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

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

Supported C++ 2011 Extensions

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

Standard	Description	Supported
C++2011-N2118	Rvalue references	Yes
C++2011-N2439	Rvalue references for *this	Yes
C++2011-N1610	Initialization of class objects by rvalues	Yes
C++2011-N2756	Non-static data member initializers	Yes
C++2011-N2242	Variadic templates	Yes
C++2011-N2555	Extending variadic template 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 declaration syntax	Yes
C++2011-N2927	New wording for C++0x lambdas	Yes
C++2011-N2343	Declared type of an expression	Yes

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

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

Standard	Description	Supported
C++2011-N2427	Concurrency - Atomic operations	No
C++2011-N2748	Concurrency - Strong Compare and Exchange	No
C++2011-N2752	Concurrency - Bidirectional Fences	No
C++2011-N2429	Concurrency - Memory model	No
C++2011-N2664	Concurrency - Data-dependency ordering: atomics and memory model	No
C++2011-N2179	Concurrency - Propagating exceptions	No
C++2011-N2440	Concurrency - Abandoning a process and 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)

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.

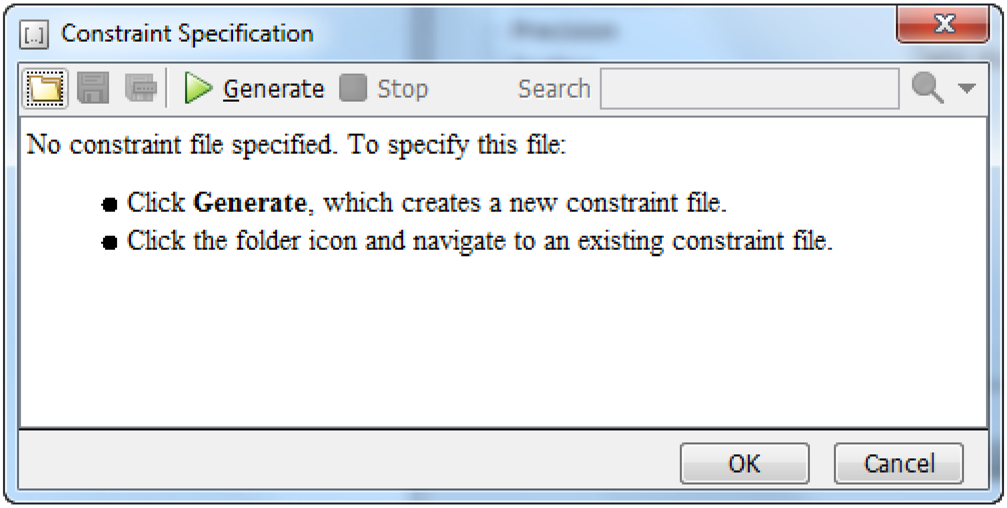
To reduce the number of such false positives, you can specify additional constraints on global variables, function inputs, and return values of stubbed functions. After you specify your constraints, you can save them as an XML file to use them for subsequent analyses. If your source code changes, you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

In this section...

Create Constraint Template

1 On the **Configuration** pane, select **Inputs & Stubbing**.

2 To the right of **Constraint setup**, click the **Edit** button.



No constraint file specified. To specify this file:

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

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 “Constraints” on page 1-72.

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 . 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 User assertion (Polyspace Code Prover).

See Also

Constraint setup (-data-range-specifications) (Polyspace Code Prover)

Related Examples

- “Constrain Global Variable Range” (Polyspace Code Prover)

Constraints

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

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

For more information, see “Specify External Constraints” on page 1-68.

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.
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.

Column	Settings
	• **`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).

Column	Settings

Init Range	Specifies the minimum and maximum values for the variable.
	You can use the keywords `min` and `max` to denote the minimum and maximum values of the variable type. For example, for the type long, `min` and `max` correspond to -2^31 and 2^31-1 respectively.
	You can also use hexadecimal values. For example: `0x12..0x100`
	For `enum` variables, you cannot specify ranges directly using the enumerator constants. Instead use the values represented by the constants.
	For `enum` variables, you can also use the keywords `enum_min` and `enum_max` to denote the minimum and maximum values that the variable can take. For example, for an `enum` variable of the type defined below, `enum_min` is 0 and `enum_max` is 5:
	`enum week{ sunday, monday=0, tuesday,`
	`     wednesday, thursday, friday, saturday};`

Initialize Pointer	Applicable only to pointers. Enabled only when you specify **Init Mode**:`INIT`. Specifies whether the pointer should be NULL: • **`May-be NULL`** – The pointer could potentially be a NULL pointer (or not). • **`Not Null`** – The pointer is never initialized as a null pointer. • **`Null`** – The pointer is initialized as NULL. **Note:** Not applicable for C++ projects.

Column	Settings
Init Allocated	Applicable only to pointers. Enabled only when you specify **Init Mode**:`INIT`. Specifies how the pointed object is allocated: • **`MAIN GENERATOR`** – The pointed object is allocated by the main generator. • **`None`** – Pointed object is not written. • **`SINGLE`** – Write the pointed object or the first element of an array. (This setting is useful for stubbed function parameters.) • **`MULTI`** – All objects (or array elements) are initialized. **Note:** Not applicable for C++ projects.
# Allocated Objects	Applicable only to pointers. Specifies how many objects are pointed to by the pointer (the pointed object is considered as an array). **Note:** The Init Allocated parameter specifies how many allocated objects are actually initialized. **Note:** Not applicable for C++ projects.
Global Assert	Specifies whether to perform an assert check on the variable at global initialization, and after each assignment.
Global Assert Range	Specifies the minimum and maximum values for the range you want to check.
Comment	Remarks that you enter, for example, justification for your DRS values.

Storage of Polyspace Preferences

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

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 14-44
- “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

Coding Rule Sets and Concepts

Rule Checking

Polyspace Coding Rule Checker

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

- MISRA C 2004
- MISRA C 2012
- MISRA® C++:2008
- JSF++:2005

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

While creating a project, you specify both the coding standard, and which rules to enforce. Polyspace software performs rule checking before and during the analysis. Violations appear in the **Results List** pane.

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

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

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

Differences Between Bug Finder and Code Prover

Coding rule checker results can differ between Polyspace Bug Finder and Polyspace Code Prover. The rule checking engines are identical in Bug Finder and Code Prover, but the context in which the checkers execute is not the same. If a project is launched from Bug Finder and Code Prover with the same source files and same configuration options, the coding rule results can differ. For example, the main generator used in Code Prover activates global variables, which causes the rule checkers to identify such global variables as initialized. The Bug Finder does not have a main generator, so handles the initialization of the global variables differently. Another difference is how violations are reported. The coding rules violations found in header files are not reported to the user in Bug Finder, but these violations are visible in Code Prover.

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

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

Polyspace MISRA C 2004 and MISRA AC AGC Checkers

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

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

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

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

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

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

Note: The Polyspace MISRA checker is based on MISRA C:2004, which also incorporates MISRA C Technical Corrigendum.

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

Software Quality Objective Subsets (C:2004)

In this section...

Rules in `SQO-Subset1`

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.

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

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

Rules in `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; }
```

Software Quality Objective Subsets (AC AGC)

In this section...
"Rules in SQO-Subset1" on page 2-10
"Rules in SQO-Subset2" on page 2-11

Rules in `SQO-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*.

MISRA C:2004 and MISRA AC AGC Coding Rules

In this section...

Supported MISRA C:2004 and MISRA AC AGC Rules

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

Note: The Polyspace coding rules checker:

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

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

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

Environment

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	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
		• ANSI C does not allow ‘#assert’ • ANSI C does not allow '#unassert' • ANSI C does not allow testing assertions • ANSI C does not allow '#ident' • ANSI C does not allow '#sccs' • text following '#else' violates ANSI standard. • text following '#endif' violates ANSI standard. • text following '#else' or '#endif' violates ANSI standard.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
1.1 (cont.)		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

N.	MISRA Definition	Messages in report file	Polyspace Specification
5.5	No object or function identifier with a static storage duration should be reused.	{static identifier/parameter name}'%s' should not be reused. (already used as {static identifier/parameter name} with static storage duration at %s:%d)	Warning when a static name is reused as another identifier name Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
5.6	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.	{member name}'%s' should not be reused. (already used as {member name} at %s:%d)	Warning when an `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.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
		- 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
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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	
8.7	Objects shall be defined at block scope if they are only accessed from within a single function	Object 'XX' should be declared at block scope.	Restricted to static objects.
8.8	An external object or function shall be declared in one file and only one file	Function/Object 'XX' has external declarations in multiples files.	Restricted to explicit extern declarations (tentative 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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
8.9	Definition: An identifier with external linkage shall have exactly one external definition.	• Procedure/Global variable XX multiply defined. • Forbidden multiple tentative definitions for object XX • Global variable has multiple tentative definitions • Undefined global variable XX	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 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.	
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	• 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	ANSI C base types order (signed char, short, int, long) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1. The same interpretation is applied on the unsigned version of base types.

N.	MISRA Definition	Messages in report file	Polyspace Specification
	• the expression is not constant and is a function argument, or • the expression is not constant and is a return expression	• Implicit conversion of the binary right hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary left hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary right hand operand of underlying type XX to XX that is not a wider integer type of the same signedness or Implicit conversion of the binary ? left hand operand of underlying type XX to XX, but it is a complex expression. • Implicit conversion of complex integer expression of underlying type XX to XX. • Implicit conversion of non-constant integer expression of underlying type XX in function return whose expected type is XX. • Implicit conversion of non-constant integer expression of underlying type XX as argument	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 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

N.	MISRA Definition	Messages in report file	Polyspace Specification
		of function whose corresponding parameter type is XX.	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.	• ANSI C base types order (signed char, short, int, long) defines that T1 is narrower than T2 if T2 is on the right hand of T1 or T1 = T2. The same methodology is applied on the unsigned version of base types. • An expression of bool or enum types has int as underlying type. • Plain char may have signed or unsigned underlying type (depending on target configuration or option setting). • The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width is not token into account and it assumes that only signed, unsigned int are used for bitfield (Rule 6.4).
10.4	The value of a complex expression of float type may only be cast to narrower floating type	Complex expression of XX type may only be cast to narrower floating type, however the destination type is XX.	ANSI C base types order (float, double) defines that T1 is narrower than T2 if T2 is on the right hand of T1 or T2 = T1.

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.5	If the bitwise operator ~ and << are applied to an operand of underlying type *unsigned char* or *unsigned short*, the result shall be immediately cast to the underlying type of the operand	Bitwise [<<|~] is applied to the operand of underlying type [unsigned char| unsigned short], the result shall be immediately cast to the underlying type.	
10.6	The "U" suffix shall be applied to all constants of *unsigned* types	No explicit 'U suffix on constants of an unsigned type.	Warning when the type determined from the value and the base (octal, decimal or hexadecimal) is unsigned and there is no suffix `u` or `U`. For example, when the size of the `int` and `long int` data types is 32 bits, the coding rule checker will 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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.	There is also a warning on qualifier loss
11.3	A cast should not be performed between a pointer type and an integral type	A cast should not be performed between a pointer type and an integral type.	Exception on zero constant. Extended to all conversions
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.	A cast should not be performed between a pointer to object type and a different pointer to object type.	
11.5	A cast shall not be performed that removes any *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.

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

N.	MISRA Definition	Messages in report file	Polyspace Specification

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

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

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.	
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	No warning is given on integer constants. Example: if (2) The use of the option `-boolean-types` may

N.	MISRA Definition	Messages in report file	Polyspace Specification
			increase or decrease the number of warnings generated.
13.3	Floating-point expressions shall not be tested for equality or inequality.	Floating-point expressions shall not be tested for equality or inequality.	Warning on directs tests only.
13.4	The controlling expression of a *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 lest one side effect however executed, or cause control flow to change	• All non-null statements shall either: • have at lest one side effect however executed, or • cause control flow to change	
14.3	All non-null statements shall either • have at lest one side effect however executed, or	A null statement shall appear on a line by itself	We assume that a ';' is a null statement when it is the first character on a line (excluding comments). The rule is violated when:

N.	MISRA Definition	Messages in report file	Polyspace Specification
	- cause control flow to change		- 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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a *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	
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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
16.4	The identifiers used in the declaration and definition of a function shall be identical.	The identifiers used in the declaration and definition of a function shall be identical.	Assumes that rules **8.8**, **8.1** and **16.3** are not violated. All occurrences are detected.
16.5	Functions with no parameters shall be declared with parameter type *void*.	Functions with no parameters shall be declared with parameter type void.	Definitions are also checked.
16.6	The number of arguments passed to a function shall match the number of parameters.	• Too many arguments to XX. • Insufficient number of arguments to XX.	Assumes that rule **8.1** is not violated.
16.7	A pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.	Pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.	Warning if a non-`const` pointer parameter is either not used to modify the addressed object or is passed to a call of a function that is declared with a `const` pointer parameter.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Missing return value for non-void function XX.	Warning when a non-void function is not terminated with an unconditional return with an expression.
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.	Function identifier XX should be preceded by a & or followed by a parameter list.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
16.10	If a function returns error information, then that error information shall be tested.	If a function returns error information, then that error information shall be tested.	Warning if a non-`void` function is called and the returned value is ignored. No warning if the result of the call is cast to `void`. No check performed for calls of `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.	
17.2	Pointer subtraction shall only be applied to pointers that address elements of the same array	Pointer subtraction shall only be applied to pointers that address elements of the same array.	
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	
17.4	Array indexing shall be the only allowed form of pointer arithmetic.	Array indexing shall be the only allowed form of pointer arithmetic.	Warning on operations on pointers. (`p+I`, `I+p` and `p-I`, where `p` is a pointer and `I` an integer).
17.5	A type should not contain more than 2 levels of pointer indirection	A type should not contain more than 2 levels of pointer indirection	
17.6	The address of an object with automatic storage shall not	Pointer to a parameter is an illegal return value. Pointer	Warning when assigning address to a global variable,

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

Structures and Unions

N.	MISRA Definition	Messages in report file	Polyspace Specification
18.1	All structure or union types shall be complete at the end of a translation unit.	All structure or union types shall be complete at the end of a translation unit.	Warning for all incomplete declarations of structs or unions.
18.2	An object shall not be assigned to an overlapping object.	• 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"	

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

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.7	A function should be used in preference to a function like-macro.	A function should be used in preference to a function like-macro	Message on all function-like macro definitions.
19.8	A function-like macro shall not be invoked without all of its arguments	• 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.	
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**.

N.	MISRA Definition	Messages in report file	Polyspace Specification
			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
20.3	The validity of values passed to library functions shall be checked.	Validity of values passed to library functions shall be checked	Warning for argument in library function call if the following are all true: • Argument is a local variable • Local variable is not tested between last assignment and call to the library function • Library function is a common mathematical function • Corresponding parameter of the library function has a restricted input domain. The library function can be one of the following : `sqrt`, `tan`, `pow`, `log`, `log10`, `fmod`, `acos`, `asin`, `acosh`, `atanh`, or `atan2`.

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

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.
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

Rule	Description	Polyspace Specification
		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.
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.

Polyspace MISRA C:2012 Checker

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

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

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 4.13 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 2-56.

See Also

Check MISRA C:2012 (-misra3) | Use generated code requirements (-misra3-agc-mode)

Related Examples

- “Activate Coding Rules Checker” on page 3-2
- “Set Up Coding Rules Checking” (Polyspace Code Prover)

More About

- “MISRA C:2012 Directives and Rules”

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

- “Software Quality Objective Subsets (C:2012)” on page 2-56

Software Quality Objective Subsets (C:2012)

In this section...
"Guidelines in SQO-Subset1" on page 2-56
"Guidelines in SQO-Subset2" on page 2-57

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
11.1	Conversions shall not be performed between a pointer to a function and any other type

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

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

See Also

Check MISRA C:2012 (-misra3) | Use generated code requirements (-misra3-agc-mode)

Related Examples

- "Activate Coding Rules Checker" on page 3-2

- “Set Up Coding Rules Checking” (Polyspace Code Prover)

More About

- “MISRA C:2012 Directives and Rules”

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.

To detect only coding rule violations, see “Find Coding Rule Violations” on page 3-15.

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.

Rule	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.

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.

Rule	Description
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.

Rule	Description
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.

Rule	Description
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.

Rule	Description
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.

Rule	Description
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.

Rule	Description
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.

Rule	Description
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.

Rule	Description
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.

Rule	Description
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 `[ ]`.

Rule	Description
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.

Rule	Description
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 coding rules. These rules cannot be enforced because they are outside the scope of Polyspace software. These guidelines concern documentation, dynamic aspects, or functional aspects of MISRA rules.

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

Polyspace MISRA C++ Checker

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

When MISRA C++ rules are violated, the Polyspace 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 192 of the 228 MISRA C++ coding rules.

There are subsets of MISRA C++ coding rules that can have a direct or indirect impact on the selectivity (reliability percentage) of your results. When you set up rule checking, you can select these subsets directly. These subsets are defined in “Software Quality Objective Subsets (C++)” on page 2-81.

Note: The Polyspace MISRA C++ checker is based on MISRA C++:2008 – “Guidelines for the use of the C++ language in critical systems."

[4] MISRA is a registered trademark of MISRA Ltd., held on behalf of the MISRA Consortium.

Software Quality Objective Subsets (C++)

SQO Subset 1 – Direct Impact on Selectivity

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

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	The One Definition Rule shall not be violated.
3-9-3	The underlying bit representations of floating-point values shall not be used.
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.

MISRA C++ Rule	Description
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
7-5-4	Functions should not call themselves, either directly or indirectly.
8-4-1	Functions shall not be defined using the ellipsis notation.
9-5-1	Unions shall not be used.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.

MISRA C++ Rule	Description
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
18-4-1	Dynamic heap memory allocation shall not be used.

SQO Subset 2 – Indirect Impact on Selectivity

Good design practices generally lead to less code complexity, which can improve the selectivity of your results. The following set of coding rules may help to address design issues that impact selectivity. The `SQO-subset2` option checks the rules in `SQO-subset1` and `SQO-subset2`.

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.
3-9-2	typedefs that indicate size and signedness should be used in place of the basic numerical types.
3-9-3	The underlying bit representations of floating-point values shall not be used.
4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, `||`, !, the

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

MISRA C++ Rule	Description
5-2-11	The comma operator, && operator and the || operator shall not be overloaded.
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
5-3-3	The unary & operator shall not be overloaded.
5-18-1	The comma operator shall not be used.
6-2-1	Assignment operators shall not be used in sub-expressions.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.
6-4-2	All if ... else if constructs shall be terminated with an else clause.
6-4-6	The final clause of a switch statement shall be the default-clause.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

MISRA C++ Rule	Description
7-5-4	Functions should not call themselves, either directly or indirectly.
8-4-1	Functions shall not be defined using the ellipsis notation.
8-4-3	All exit paths from a function with non- void return type shall have an explicit return statement with an expression.
8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.
8-5-2	Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.
8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.
11-0-1	Member data in non- POD class types shall be private.
12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.
12-8-2	The copy assignment operator shall be declared protected or private in an abstract class.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.

MISRA C++ Rule	Description
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.
16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.
16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.
16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.
18-4-1	Dynamic heap memory allocation shall not be used.

MISRA C++ Coding Rules

In this section...

Supported MISRA C++ Coding Rules

Language Independent Issues

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

N.	Category	MISRA Definition	Polyspace Specification
			analyses can produce different results.
0-1-2	Required	A project shall not contain infeasible paths.	
0-1-7	Required	The value returned by a function having a non- void return type that is not an overloaded operator shall always be used.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
0-1-9	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.
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

N.	Category	MISRA Definition	Polyspace Specification
			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.	

N.	Category	MISRA Definition	Polyspace Specification
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.	
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	

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

N.	Category	MISRA Definition	Polyspace Specification
5-0-4	Required	An implicit integral conversion shall not change the signedness of the underlying type.	Assumes that `ptrdiff_t` is signed integer If the conversion is to a narrower integer with a different sign then MISRA C++ 5-0-4 takes precedence over MISRA C++ 5-0-6.
5-0-5	Required	There shall be no implicit floating-integral conversions.	This rule takes precedence over 5-0-4 and 5-0-6 if they apply at the same time.
5-0-6	Required	An implicit integral or floating-point conversion shall not reduce the size of the underlying type.	If the conversion is to a narrower integer with a different sign then MISRA C++ 5-0-4 takes precedence over MISRA C++ 5-0-6.
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.

N.	Category	MISRA Definition	Polyspace Specification
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.	
5-0-15	Required	Array indexing shall be the only form of pointer arithmetic.	Warning on operations on pointers. (p+I, I+p and p-I, where p is a pointer and I an integer, p[i] accepted).
5-0-18	Required	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.	Report when relational operator are used on pointers types (casts ignored).
5-0-19	Required	The declaration of objects shall contain no more than two levels of pointer indirection.	
5-0-20	Required	Non-constant operands to a binary bitwise operator shall have the same underlying type.	
5-0-21	Required	Bitwise operators shall only be applied to operands of unsigned underlying type.	
5-2-1	Required	Each operand of a logical && or || shall be a postfix - expression.	During preprocessing, violations of this rule are detected on the expressions in #if directives. Allowed exception on associativity (a && b && c), (a || b || c).
5-2-2	Required	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.	

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

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.	
6-2-3	Required	Before preprocessing, a null statement shall only occur on a	

N.	Category	MISRA Definition	Polyspace Specification
		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.	
6-4-7	Required	The condition of a switch statement shall not have bool type.	

N.	Category	MISRA Definition	Polyspace Specification
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.	
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.

N.	Category	MISRA Definition	Polyspace Specification
6-6-4	Required	For any iteration statement there shall be no more than one break or goto statement used for loop termination.	
6-6-5	Required	A function shall have a single point of exit at the end of the function.	At most one return not necessarily as last statement for void functions.

Declarations

N.	Category	MISRA Definition	Polyspace Specification
7-3-1	Required	The global namespace shall only contain main, namespace declarations and extern "C" declarations.	
7-3-2	Required	The identifier main shall not be used for a function other than the global function main.	
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.	

N.	Category	MISRA Definition	Polyspace Specification
7-5-1	Required	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.	
7-5-2	Required	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	
7-5-3	Required	A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.	
7-5-4	Advisory	Functions should not call themselves, either directly or indirectly.	

Declarators

N.	Category	MISRA Definition	Polyspace Specification
8-0-1	Required	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.	
8-3-1	Required	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.	
8-4-1	Required	Functions shall not be defined using the ellipsis notation.	
8-4-2	Required	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.	

N.	Category	MISRA Definition	Polyspace Specification
8-4-3	Required	All exit paths from a function with non- void return type shall have an explicit return statement with an expression.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8-4-4	Required	A function identifier shall either be used to call the function or it shall be preceded by &.	
8-5-1	Required	All variables shall have a defined value before they are used.	Non-initialized variable in results and error messages for obvious cases
8-5-2	Required	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	
8-5-3	Required	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	

Classes

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

Derived Classes

N.	Category	MISRA Definition	Polyspace Specification
10-1-1	Advisory	Classes should not be derived from virtual bases.	
10-1-2	Required	A base class shall only be declared virtual if it is used in a diamond hierarchy.	Assumes 10.1.1 not required
10-1-3	Required	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.	
10-2-1	Required	All accessible entity names within a multiple inheritance hierarchy should be unique.	No detection between entities of different kinds (member functions against data members, ...).
10-3-1	Required	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.	Member functions that are virtual by inheritance are also detected.
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.	

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

Templates

N.	Category	MISRA Definition	Polyspace Specification
14-5-2	Required	A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.	
14-5-3	Required	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.	
14-6-1	Required	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->	
14-6-2	Required	The function chosen by overload resolution shall resolve to a	

N.	Category	MISRA Definition	Polyspace Specification
		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.	
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".

N.	Category	MISRA Definition	Polyspace Specification
			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.	
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.

N.	Category	MISRA Definition	Polyspace Specification
15-5-2	Required	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).	Limit detection to throw that are internals to the function; rethrows are partially processed; no detections in nested handlers.

Preprocessing Directives

N.	Category	MISRA Definition	Polyspace Specification
16-0-1	Required	#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.	

N.	Category	MISRA Definition	Polyspace Specification
16-1-1	Required	The defined preprocessor operator shall only be used in one of the two standard forms.	
16-1-2	Required	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	
16-2-1	Required	The preprocessor shall only be used for file inclusion and include guards.	The rule is raised for #ifdef/#define if the file is not an include file.
16-2-2	Required	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.	
16-2-3	Required	Include guards shall be provided.	
16-2-4	Required	The ', ", /* or // characters shall not occur in a header file name.	
16-2-5	Advisory	The \ character should not occur in a header file name.	
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.
17-0-2	Required	The names of standard library macros and objects shall not be reused.	
17-0-5	Required	The setjmp macro and the longjmp function shall not be used.	

Language Support Library

N.	Category	MISRA Definition	Polyspace Specification
18-0-1	Required	The C library shall not be used.	
18-0-2	Required	The library functions atof, atoi and atol from library <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 2-110
- “General” on page 2-111
- “Lexical Conventions” on page 2-112
- “Standard Conversions” on page 2-112
- “Expressions” on page 2-112
- “Declarations” on page 2-113
- “Classes” on page 2-113
- “Templates” on page 2-114
- “Exception Handling” on page 2-114
- “Library Introduction” on page 2-115

Language Independent Issues

N.	Category	MISRA Definition	Polyspace Specification
0–1–3	Required	A project shall not contain unused variables.	
0-1-4	Required	A project shall not contain non-volatile POD variables having only one use.	

N.	Category	MISRA Definition	Polyspace Specification
0-1-5	Required	A project shall not contain unused type declarations.	
0-1-6	Required	A project shall not contain instances of non-volatile variables being given values that are never subsequently used.	
0-1-8	Required	All functions with void return type shall have external side effects.	
0-3-1	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.
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.

Standard Conversions

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

Expressions

N.	Category	MISRA Definition	Polyspace Specification
5-0-16	Required	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	
5-0-17	Required	Subtraction between pointers shall only be applied to pointers that address elements of the same array.	

N.	Category	MISRA Definition	Polyspace Specification
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
7-1-1	Required	A variable which is not modified shall be const qualified.	
7-1-2	Required	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.	
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.
15-1-1	Required	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	
15-3-1	Required	Exceptions shall be raised only after start-up and before termination of the program.	
15-3-4	Required	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	
15-5-3	Required	The terminate() function shall not be called implicitly.	

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.

Polyspace JSF C++ Checker

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

[5]

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

Note: The Polyspace JSF C++ checker is based on JSF++:2005.

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

JSF C++ Coding Rules

In this section...

Supported JSF C++ Coding Rules

- “Fault Handling” on page 2-139
- “Portable Code” on page 2-139

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.	
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.	

N.	JSF++ Definition	Polyspace Specification
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:** 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".

N.	JSF++ Definition	Polyspace Specification
81	The assignment operator shall handle self-assignment correctly	Reports when copy assignment body does not begin with "`if (this != arg)`" A violation is not raised if an empty `else` statement follows the `if`, or the body contains only a return statement. A violation is raised when the `if` statement is followed by a statement other than the return statement.
82	An assignment operator **shall** return a reference to `*this`.	The following operators should return `*this` on method, and `*first_arg` on plain function. `operator=operator+=operator-=operator*=operator >>=operator <<=operator /=operator %=operator |=operator &=operator ^=prefix operator++ prefix operator--` Does not report when no return exists. No special message if type does not match. Messages in report file: • An assignment operator shall return a reference to `*this`. • An assignment operator shall return a reference to its first arg.
83	An assignment operator shall assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).	Reports when a copy assignment does not assign all data members. In a derived class, it also reports when a copy assignment does not call inherited copy assignments.

N.	JSF++ Definition	Polyspace Specification
88	Multiple inheritance **shall** only be allowed in the following restricted form: `n` interfaces plus `m` private implementations, plus at most one protected implementation.	Messages in report file: • 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.	
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.

N.	JSF++ Definition	Polyspace Specification
119	Functions **shall** not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	Direct recursion is reported statically. Indirect recursion reported through the software. Message in report file: Function <*F*> shall not call directly itself.
121	Only functions with 1 or 2 statements **should** be considered candidates for inline functions.	Reports inline functions with more than 2 statements.

Comments

N.	JSF++ Definition	Polyspace Specification
126	Only valid C++ style comments (`//`) **shall** be used.	
133	Every source file will be documented with an introductory comment that provides information on the file name, its contents, and any program-required information (e.g. legal statements, copyright information, etc).	Reports when a file does not begin with two comment lines. **Note**: This rule cannot be annotated in the source code.

Declarations and Definitions

N.	JSF++ Definition	Polyspace Specification
135	Identifiers in an inner scope **shall not** use the same name as an identifier in an outer scope, and therefore hide that identifier.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
136	Declarations should be at the smallest feasible scope.	Reports when: • 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

N.	JSF++ Definition	Polyspace Specification
		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.
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.
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

<table>
<tr><th>N.</th><th>JSF++ Definition</th><th>Polyspace Specification</th></tr>
<tr><td>157</td><td>The right hand operand of a && or || operator shall not contain side effects.</td><td>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.</td></tr>
<tr><td>158</td><td>The operands of a logical && or || <b>shall</b> be parenthesized if the operands contain binary operators.</td><td>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</td></tr>
<tr><td>159</td><td>Operators ||, &&, and unary & <b>shall not</b> be overloaded.</td><td>Messages in report file:
• Unary operator & shall not be overloaded.
• Operator || shall not be overloaded.
• Operator && shall not be overloaded.</td></tr>
</table>

N.	JSF++ Definition	Polyspace Specification
160	An assignment expression **shall** be used only as the expression in an expression statement.	Only simple assignment, not +=, ++, etc.
162	Signed and unsigned values **shall not** be mixed in arithmetic or comparison operations.	
163	Unsigned arithmetic **shall not** be used.	
164	The right hand operand of a shift operator **shall** lie between zero and one less than the width in bits of the left-hand operand (inclusive).	
164.1	The left-hand operand of a right-shift operator **shall not** have a negative value.	Detects constant case +. Found by the software for dynamic cases.
165	The unary minus operator **shall not** be applied to an unsigned expression.	
166	The `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	Reports when relational operator are used on pointer types (casts ignored).

N.	JSF++ Definition	Polyspace Specification
	• elements of the same array (including one past the end of the same array).	
173	The address of an object with automatic storage **shall not** be assigned to an object which persists after the object has ceased to exist.	
174	The null pointer **shall not** be de-referenced.	Done with checks in software.
175	A pointer **shall not** be compared to `NULL` or be assigned `NULL`; use plain `0` instead.	Reports usage of `NULL` macro in pointer contexts.
176	A `typedef` **will** be used to simplify program syntax when declaring function pointers.	Reports non-`typedef` function pointers, or pointers to member functions for types of variables, fields, parameters. Returns type of function, cast, and exception specification.

Type Conversions

N.	JSF++ Definition	Polyspace Specification
177	User-defined conversion functions **should** be avoided.	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.)

N.	JSF++ Definition	Polyspace Specification
179	A pointer to a virtual base class **shall not** be converted to a pointer to a derived class.	Reports this specific down cast. Allows dynamic_cast.

180	Implicit conversions that may result in a loss of information **shall not** be used.	Reports the following implicit casts :
		`integer => smaller integer`
		`unsigned => smaller or eq signed`
		`signed => smaller or eq un-signed`
		`integer => float float => integer`
		Does not report for cast to `bool` reports for implicit cast on constant done with the options `-scalar-overflows-checks signed-and-unsigned` or `-ignore-constant-overflows`
		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.

N.	JSF++ Definition	Polyspace Specification
		Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
187	All non-null statements **shall** potentially have a side-effect.	
188	Labels **will not** be used, except in switch statements.	
189	The `goto` statement **shall** not be used.	
190	The `continue` statement **shall not** be used.	
191	The `break` statement **shall not** be used (except to terminate the cases of a switch statement).	
192	All `if`, `else if` constructs will contain either a final `else` clause or a comment indicating why a final `else` clause is not necessary.	`else if` should contain an `else` clause.
193	Every non-empty `case` clause in a switch statement **shall** be terminated with a `break` statement.	
194	All `switch` statements that do not intend to test for every enumeration value **shall** contain a final `default` clause.	Reports only for missing `default`.
195	A `switch` expression **will** not represent a Boolean value.	
196	Every `switch` statement **will** have at least two cases and a potential `default`.	
197	Floating point variables **shall not** be used as loop counters.	Assumes 1 loop parameter.
198	The initialization expression in a `for` loop **will** perform no actions other than to initialize the value of a single `for` loop parameter.	Reports if `loop` parameter cannot be determined. Assumes Rule 200 is not violated. The `loop variable` parameter is assumed to be a variable.

N.	JSF++ Definition	Polyspace Specification
199	The increment expression in a `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.
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

N.	JSF++ Definition	Polyspace Specification
		- 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`.
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.

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

Unsupported JSF++ Rules

- "Code Size and Complexity" on page 2-140
- "Rules" on page 2-141
- "Environment" on page 2-141
- "Libraries" on page 2-141
- "Header Files" on page 2-141
- "Style" on page 2-142
- "Classes" on page 2-142
- "Namespaces" on page 2-144
- "Templates" on page 2-144
- "Functions" on page 2-144
- "Comments" on page 2-145
- "Initialization" on page 2-145
- "Types" on page 2-146
- "Unions and Bit Fields" on page 2-146
- "Operators" on page 2-146
- "Type Conversions" on page 2-146
- "Expressions" on page 2-146
- "Memory Allocation" on page 2-147
- "Portable Code" on page 2-147
- "Efficiency Considerations" on page 2-147
- "Miscellaneous" on page 2-147
- "Testing" on page 2-147

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.

N.	JSF++ Definition
37	Header (include) files should include only those header files that are required for them to successfully compile. Files that are only used by the associated .cpp file should be placed in the .cpp file — not the .h file.
38	Declarations of classes that are only accessed via pointers (*) or references (&) should be supplied by forward headers that contain only forward declarations.

Style

N.	JSF++ Definition
45	All words in an identifier will be separated by the '_' character.
49	All acronyms in an identifier will be composed of uppercase letters.
55	The name of a header file should reflect the logical entity for which it provides declarations.
56	The name of an implementation file should reflect the logical entity for which it provides definitions and have a ".cpp" extension (this name will normally be identical to the header file that provides the corresponding declarations.) At times, more than one .cpp file for a given logical entity will be required. In these cases, a suffix should be appended to reflect a logical differentiation.

Classes

N.	JSF++ Definition
64	A class interface should be complete and minimal.
65	A structure should be used to model an entity that does not require an invariant.
66	A class should be used to model an entity that maintains an invariant.
69	A member function that does not affect the state of an object (its instance variables) will be declared const. Member functions should be const by default. Only when there is a clear, explicit reason should the const modifier on member functions be omitted.
70	A class will have friends only when a function or object requires access to the private elements of the class, but is unable to be a member of the class for logical or efficiency reasons.
70.1	An object shall not be improperly used before its lifetime begins or after its lifetime ends.
71	Calls to an externally visible operation of an object, other than its constructors, shall not be allowed until the object has been fully initialized.

N.	JSF++ Definition
72	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.
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.

N.	JSF++ Definition
120	Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal parameters.
122	Trivial accessor and mutator functions should be inlined.
123	The number of accessor and mutator functions should be minimized.
124	Trivial forwarding functions should be inlined.
125	Unnecessary temporary objects should be avoided.

Comments

N.	JSF++ Definition
127	Code that is not used (commented out) shall be deleted. **Note**: This rule cannot be annotated in the source code.
128	Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented will not be allowed.
129	Comments in header files should describe the externally visible behavior of the functions or classes being documented.
130	The purpose of every line of executable code should be explained by a comment, although one comment may describe more than one line of code.
131	One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).
132	Each variable declaration, typedef, enumeration value, and structure member will be commented.
134	Assumptions (limitations) made by functions should be documented in the function's preamble.

Initialization

N.	JSF++ Definition
143	Variables will not be introduced until they can be initialized with meaningful values. (See also AV Rule 136, AV Rule 142, and AV Rule 73 concerning declaration scope, initialization before use, and default constructors respectively.)

Types

N.	JSF++ Definition
146	Floating point implementations shall comply with a defined floating point standard. The standard that will be used is the ANSI/IEEE® Std 754 [1].

Unions and Bit Fields

N.	JSF++ Definition
155	Bit-fields will not be used to pack data into a word for the sole purpose of saving space.

Operators

N.	JSF++ Definition
167	The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.

Type Conversions

N.	JSF++ Definition
183	Every possible measure should be taken to avoid type casting.

Expressions

N.	JSF++ Definition
204	A single operation with side-effects shall only be used in the following contexts: **1** by itself **2** the right-hand side of an assignment **3** a condition **4** the only argument expression with a side-effect in a function call **5** condition of a loop **6** switch condition **7** single part of a chained operation

Memory Allocation

N.	JSF++ Definition
207	Unencapsulated global data will be avoided.

Portable Code

N.	JSF++ Definition
210	Algorithms shall not make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.).
210.1	Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.
211	Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.
212	Underflow or overflow functioning shall not be depended on in any special way.
214	Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.

Efficiency Considerations

N.	JSF++ Definition
216	Programmers should not attempt to prematurely optimize code.

Miscellaneous

N.	JSF++ Definition
217	Compile-time and link-time errors should be preferred over run-time errors.
218	Compiler warning levels will be set in compliance with project policies.

Testing

N.	JSF++ Definition
219	All tests applied to a base class interface shall be applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions /invariants shall be substituted in the derived class tests.

N.	JSF++ Definition
220	Structural coverage algorithms shall be applied against flattened classes.
221	Structural coverage of a class within an inheritance hierarchy containing virtual functions shall include testing every possible resolution for each set of identical polymorphic references.

Check Coding Rules from the Polyspace Environment

Activate Coding Rules Checker

This example shows how to activate the coding rules checker before you start an analysis. This activation enables Polyspace Bug Finder to search for coding rule violations. You can view the coding rule violations in your analysis results.

1 Open project configuration.

2 On the **Configuration** pane, select **Coding Rules & Code Metrics**.

3 Select the check box for the type of coding rules that you want to check.

   For C code, you can check compliance with:

   - MISRA C:2004
   - MISRA AC AGC
   - MISRA C:2012

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

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

   - MISRA C++: 2008
   - JSF C++
   - Custom coding rules

4 For each rule type that you select, from the drop-down list, select the subset of rules to check.

   ### MISRA C:2004

Option	Description
`required-rules`	All required MISRA C:2004 coding rules.
`all-rules`	AllMISRA C:2004 coding rules (required and advisory).
`SQO-subset1`	A small subset of MISRA C:2004 rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.

Option	Description
`SQO-subset2`	A second subset of rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA C:2004 coding rules that you specify.

MISRA AC AGC

Option	Description
`OBL-rules`	All required MISRA AC AGC coding rules.
`OBL-REC-rules`	All required and recommended MISRA AC AGC coding rules.
`all-rules`	All required, recommended, and readability coding rules.
`SQO-subset1`	A small subset of MISRA AC AGC rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of MISRA AC AGC rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA AC AGC coding rules that you specify.

MISRA C:2012

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

Option	Description
`SQO-subset1`	A small subset of MISRA C rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA C:2012 coding rules that you specify.

MISRA C++

Option	Description
`required-rules`	All required MISRA C++ coding rules.
`all-rules`	All required and advisory MISRA C++ coding rules.
`SQO-subset1`	A small subset of MISRA C++ rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of rules with indirect impact on the selectivity in addition to `SQO-subset1`. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A specified set of MISRA C++ coding rules.

JSF C++

Option	Description
`shall-rules`	**Shall** rules are mandatory requirements. These rules require verification.
`shall-will-rules`	All **Shall** and **Will** rules. **Will** rules are intended to be mandatory requirements. However, these rules do not require verification.
`all-rules`	All **Shall**, **Will**, and **Should** rules. **Should** rules are advisory rules.
`custom`	A set of JSF C++ coding rules that you specify.

5 If you select **Check custom rules**, specify the path to your custom rules file or click **Edit** to create one.

When rules checking is complete, the software displays the coding rule violations in purple on the **Results List** pane.

Related Examples

- "Select Specific MISRA or JSF Coding Rules" on page 3-6
- "Create Custom Coding Rules" on page 3-9
- "Exclude Files from Analysis" on page 3-12

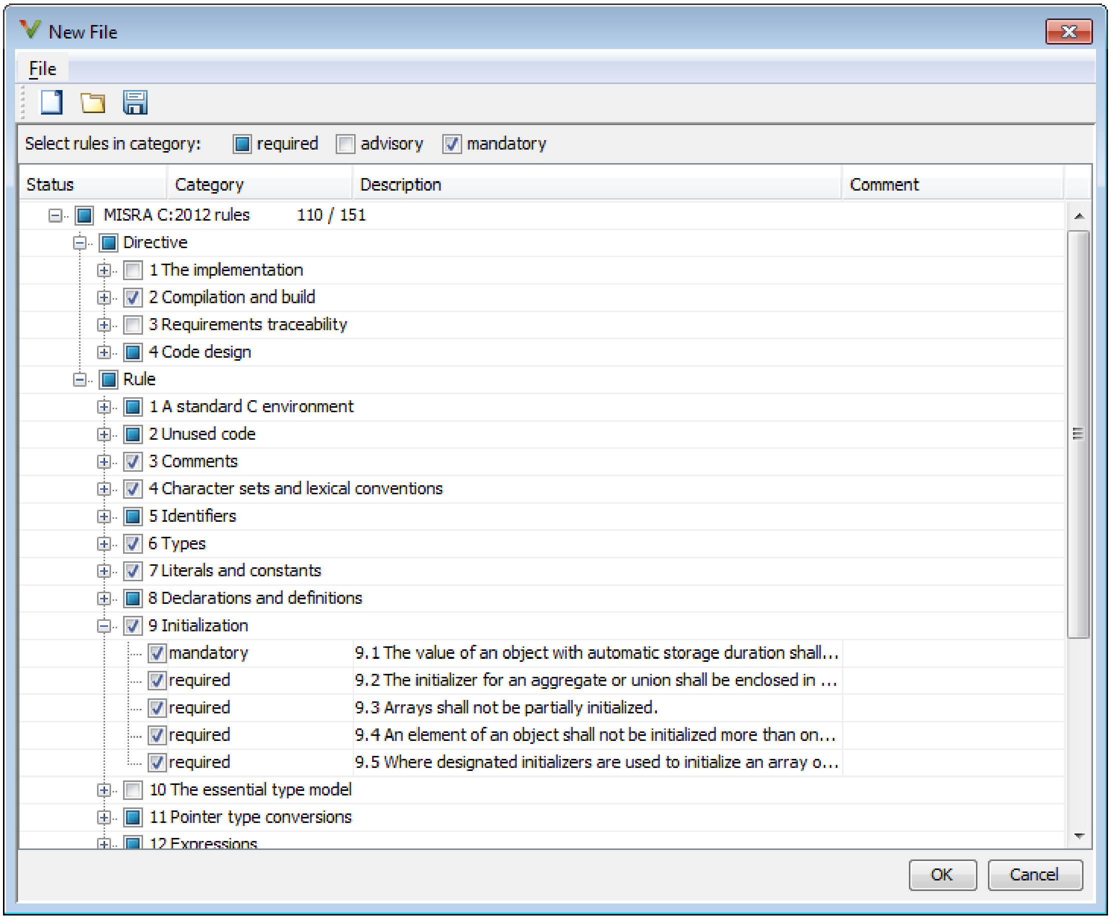
More About

- "Rule Checking" on page 2-2

Select Specific MISRA or JSF Coding Rules

This example shows how to specify a subset of MISRA or JSF rules for the coding rules checker. If you select `custom` from the MISRA or JSF drop-down list, you must provide a file that specifies the rules to check.

1. Open the project configuration.
2. In the **Configuration** tree view, select **Coding Rules & Code Metrics**.
3. Select the check box for the type of coding rules you want to check.
4. From the corresponding drop-down list, select `custom`. The software displays a new field for your custom file.
5. To the right of this field, click **Edit**. A New File window opens, displaying a table of rules.



Select rules in category: ■ required □ advisory ☑ mandatory

Status	Category	Description	Comment
⊟ ■ MISRA C:2012 rules	110 / 151		
⊟ ■ Directive			
⊞ □ 1 The implementation			
⊞ ☑ 2 Compilation and build			
⊞ □ 3 Requirements traceability			
⊞ ■ 4 Code design			
⊟ ■ Rule			
⊞ ■ 1 A standard C environment			
⊞ ■ 2 Unused code			
⊞ ☑ 3 Comments			
⊞ ☑ 4 Character sets and lexical conventions			
⊞ ■ 5 Identifiers			
⊞ ☑ 6 Types			
⊞ ☑ 7 Literals and constants			
⊞ ■ 8 Declarations and definitions			
⊟ ☑ 9 Initialization			
☑	mandatory	9.1 The value of an object with automatic storage duration shall...	
☑	required	9.2 The initializer for an aggregate or union shall be enclosed in ...	
☑	required	9.3 Arrays shall not be partially initialized.	
☑	required	9.4 An element of an object shall not be initialized more than on...	
☑	required	9.5 Where designated initializers are used to initialize an array o...	
⊞ □ 10 The essential type model			
⊞ ■ 11 Pointer type conversions			
⊞ ■ 12 Expressions			

6 If you already have a customized rule file you want to edit, reload your customization using the 📁 button.

7 Select the rules you want to check.

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

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

9 For new files, use the Save As dialog box the opens to save your customization as a rules file.

10 In the Configuration window, the full path to the rules file appears in the `custom` field. To reuse this customized set of rules for other projects, enter this path name in the dialog box.

Related Examples

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

More About

- "Rule Checking" on page 2-2

Create Custom Coding Rules

This example shows how to create a custom coding rules file. You can use this file to check names or text patterns in your source code against custom rules that you specify. For each rule, you specify a pattern in the form of a regular expression. The software compares the pattern against identifiers in the source code and determines whether the custom rule is violated.

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)

Related Examples

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

More About

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

Format of Custom Coding Rules File

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

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

- *N.n* — Custom rule number, for example, 1.2.
- `off` — Rule is not considered.
- `on` — The software checks for violation of the rule. After 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_.*
```

Related Examples

- "Create Custom Coding Rules" on page 3-9

Exclude Files from Analysis

This example shows how to specify files that you do not want analyzed. For instance, sometimes, you have to add header files from a third-party library to your Polyspace project for a precise analysis, but you cannot address defects in those header files. Therefore, you do not want analysis results on those files.

By default:

- Results are generated for all source files and header files in the same folders as source files.
- Results are not generated for the remaining header files in your project.

You can change this default behavior and specify your own set of files on which you do not want results.

1. Open the project configuration.
2. In the **Configuration** tree view, select **Inputs & Stubbing**.
3. Use a combination of the following options to suppress results from files in which you are not interested.

   - Do not generate results for (-do-not-generate-results-for)
   - Generate results for sources and (-generate-results-for)

   For instance, you can suppress results from certain folders and unsuppress them only for certain files in those folders.

Related Examples

- "Activate Coding Rules Checker" on page 3-2

More About

- "Rule Checking" on page 2-2

Allow Custom Pragma Directives

This example shows how to exclude custom pragma directives from coding rules checking. MISRA C rule 3.4 requires checking that pragma directives are documented within the documentation of the compiler. However, you can allow undocumented pragma directives to be present in your code.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules & Code Metrics**.
3. To the right of **Allowed pragmas**, click ➕.

   In the **Pragma** view, the software displays an active text field.
4. In the text field, enter a pragma directive.
5. To remove a directive from the **Pragma** list, select the directive. Then click 🗑.

Related Examples

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

More About

- “Rule Checking” on page 2-2

Specify Boolean Types

This example shows how to specify data types you want Polyspace to consider as Boolean during MISRA C rules checking. The software applies this redefinition only to data types defined by `typedef` statements.

The use of this option is related to checking of the following rules:

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

  For more information, see "MISRA C:2004 and MISRA AC AGC Coding Rules" on page 2-14.
- MISRA C:2012 — 10.1, 10.3, 10.5, 14.4 and 16.7

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules & Code Metrics**.
3. To the right of **Effective boolean types**, click ✚.

   In the **Type** view, the software displays an active text field.
4. In the text field, specify the data type that you want Polyspace to treat as Boolean.
5. To remove a data type from the **Type** list, select the data type. Then click 🗑.

Related Examples

- "Activate Coding Rules Checker" on page 3-2

More About

- "Rule Checking" on page 2-2

Find Coding Rule Violations

This example shows how to check for coding rule violations alone.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules & Code Metrics**. Activate the desired coding rule checker.

   For more information, see “Activate Coding Rules Checker” on page 3-2.
3. If you select certain rules, the analysis can complete quicker than checking other rules.

   For more information, see “Coding Rule Subsets Checked Early in Analysis” on page 2-61.
4. Specify that the analysis must not look for defects.

   - In the **Configuration** tree view, select **Bug Finder Analysis**.
   - Clear the **Find defects** check box.
5. Click ▶ to run the coding rules checker without checking defects.

Related Examples

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

More About

- “Rule Checking” on page 2-2

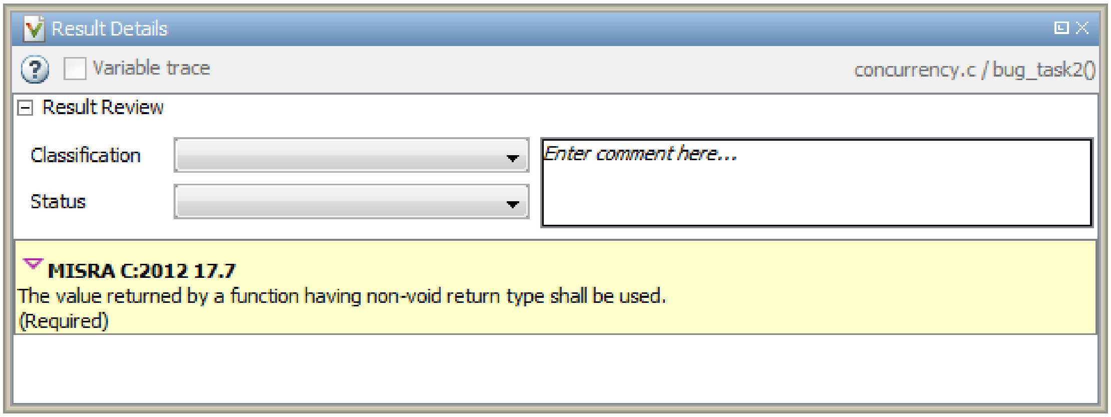
Review Coding Rule Violations

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

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

1 Select a coding-rule violation on the **Results List** pane.

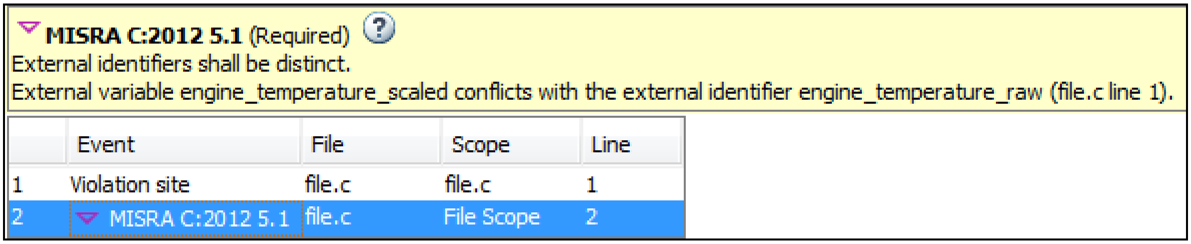
2 On the **Result Details** pane, view the location and description of the violated rule. In the source code, the line containing the violation appears highlighted.



▽ **MISRA C:2012 17.7**
The value returned by a function having non-void return type shall be used.
(Required)

3 For certain rules, use additional information available for investigating the rule violation.

- For MISRA C: 2012 rules, on the **Result Details** pane, click the ⓘ icon to see the rationale for the rule. In some cases, you can also see code examples illustrating the violation.
- For MISRA C: 2012 and MISRA C++ rules that involve more than one location in the code, the **Result Details** pane shows both locations as an event history. To navigate to a location in the source code, click the corresponding event.



MISRA C:2012 5.1 (Required)
External identifiers shall be distinct.
External variable engine_temperature_scaled conflicts with the external identifier engine_temperature_raw (file.c line 1).

	Event	File	Scope	Line
1	Violation site	file.c	file.c	1
2	MISRA C:2012 5.1	file.c	File Scope	2

4 Review the violation in your code.

   a Determine whether you must fix the code to avoid the violation.

   b If you choose to retain the code, on the **Result Details** pane, add a comment explaining why you retain the code. This comment helps you or other reviewers avoid reviewing the same coding rule violation twice.

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

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

Related Examples

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

Filter and Group Coding Rule Violations

This example shows how to use filters in the **Results List** pane to focus on specific kinds of coding rule violations. By default, the software displays both coding rule violations and defects.

In this section...

Filter Coding Rules

1. On the **Results List** pane, select the ☑ icon on the **Check** column header.
2. From the context menu, clear the **All** check box.
3. Select the violated rule numbers that you want to focus on.
4. Click **OK**.

To filter out all results other than coding rule violations, use the filters on the **Type** or **Family** column header.

You can also filter rule violations using the ***`Coding rule`*** **violations by rule (Top 10 only)** graph on the **Dashboard** pane in the Polyspace user interface. See “Filter and Group Results” on page 5-4.

Group Coding Rules

1. On the **Results List** pane, from the ☰▾ list, select **Family**.

   The rules are grouped by numbers. Each group corresponds to a certain code construct.
2. Expand the group nodes to select an individual coding rule violation.

Suppress Certain Rules from Display in One Click

Instead of filtering individual rules from display each time you open your results, you can limit the display of rule violations in one click. Use the drop-down list in the left of the **Results List** pane toolbar. You can add some predefined options to this list or create your own options. You can share the option file to help developers in your organization review violations of at least certain coding rules.

1. In the Polyspace user interface, select **Tools** > **Preferences**.
2. On the **Review Scope** tab, do one of the following:

   - To add predefined options to the drop-down list on the **Results List** pane, select **Include Quality Objectives Scopes**.

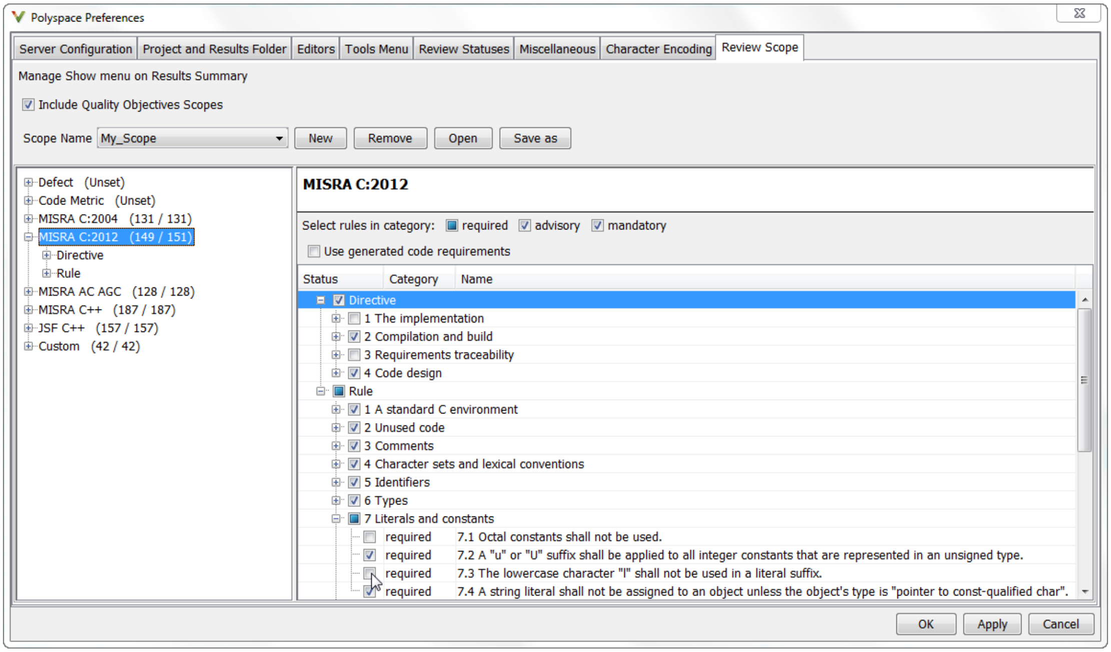
     The **Scope Name** list shows additional options, `HIS`, `SQO-4`, `SQO-5`, and `SQO-6`. Select an option to see which rules are suppressed from display.

     In addition to coding rule violations, the options impose limits on the display of code metrics and defects.
   - To create your own option in the drop-down list on the **Results List** pane, select **New**. Save your option file.

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

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

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



Manage Show menu on Results Summary

☑ Include Quality Objectives Scopes

Scope Name My_Scope

- Defect (Unset)
- Code Metric (Unset)
- MISRA C:2004 (131 / 131)
- MISRA C:2012 (149 / 151)
  - Directive
  - Rule
- MISRA AC AGC (128 / 128)
- MISRA C++ (187 / 187)
- JSF C++ (157 / 157)
- Custom (42 / 42)

MISRA C:2012

Select rules in category: ■ required ☑ advisory ☑ mandatory

□ Use generated code requirements

Status	Category	Name
☑ Directive		
□ 1 The implementation		
☑ 2 Compilation and build		
□ 3 Requirements traceability		
☑ 4 Code design		
■ Rule		
☑ 1 A standard C environment		
☑ 2 Unused code		
☑ 3 Comments		
☑ 4 Character sets and lexical conventions		
☑ 5 Identifiers		
☑ 6 Types		
■ 7 Literals and constants		
□	required	7.1 Octal constants shall not be used.
☑	required	7.2 A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.
□	required	7.3 The lowercase character "l" shall not be used in a literal suffix.
☑	required	7.4 A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

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

On the **Results List** pane, the drop-down list on the **Results List** pane displays the additional options.

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

Related Examples

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

Find Bugs From the Polyspace Environment

Choose Specific Defects

There are two preset configurations for Bug Finder defects, but you can also customize which defects to check for during the analysis.

1 On the **Configuration** pane, select **Bug Finder Analysis**.
2 From the **Find defects** menu, select a set of defects. The options are:

   - `default` for the default list of defects. This list contains defects that are applicable to most coding projects. To see the defects in the default list, expand the nodes.
   - `all` for all defects.
   - `CWE`, `CERT-rules`, `CERT-all`, or `ISO-17961`, for defects related to a security standard.

     For more information, see “Check Code for Security Standards” on page 5-92.
   - `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\R2017a`.

Run Local Analysis

Before running an analysis from the Polyspace interface, you must set up your project's source files and analysis options. For more information, see "Create New Project Manually" on page 1-2.

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

   During a Polyspace Bug Finder analysis, the software first compiles the project and looks for coding rule errors. If the files have compilation errors, a message appears in the **Output Summary** pane and the offending files are ignored during the later analysis stages. Files with compilation problems do not appear in the results.
4. Once some results are available, start reviewing the results by selecting the link in the Output Summary window or by clicking the Running (51) button on the toolbar. This button reactivates as more results are ready.
5. If you viewed some of the results during the analysis, click the toolbar button Completed (5) to load the rest of your results.

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

   After analysis is over, the **Dashboard** pane shows the number of files analyzed. 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.

Related Examples

- "Run Remote Batch Analysis" on page 4-4
- "Create New Project Manually" on page 1-2
- "Open Results" on page 5-2
- "Review and Fix Results" on page 5-32

Run Remote Batch Analysis

Before running a batch analysis, you must set up your project's source files, analysis options, and remote analysis settings. If you have not done so, see "Create New Project Manually" on page 1-2 and "Set Up Polyspace Metrics".

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**.

   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`.

Related Examples

- "Open Results" on page 5-2
- "View Results List in Polyspace Metrics" on page 15-23

Monitor Analysis

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

- **Output Summary** — Displays progress of analysis, compile phase messages and errors. To search for a term, in the **Search** field, enter the required term. Click the up or down arrow to move sequentially through occurrences of the term.
- **Full Log** — This tab displays messages, errors, and statistics for the phases of the analysis. To search for a term, in the **Search** field, enter the required term. Click the up arrow or down arrow to move sequentially through occurrences of this term.

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

To monitor the progress of a remote analysis:

1. From the Polyspace interface, select **Tools** > **Open Job Monitor**.
2. In the Polyspace Job Monitor, follow your queued job to monitor progress.

Specify Results Folder

This example shows how to specify the folder that contains your Polyspace results.

- In the **Project Browser** pane, the folder appears as a node under the **Result** node of your project module.
- This node points to the results folder in your file explorer. The results folder is located in `*Project_folder*` / `*Module_name*`. `*Project_folder*` is the project location that you specified when you created a project.

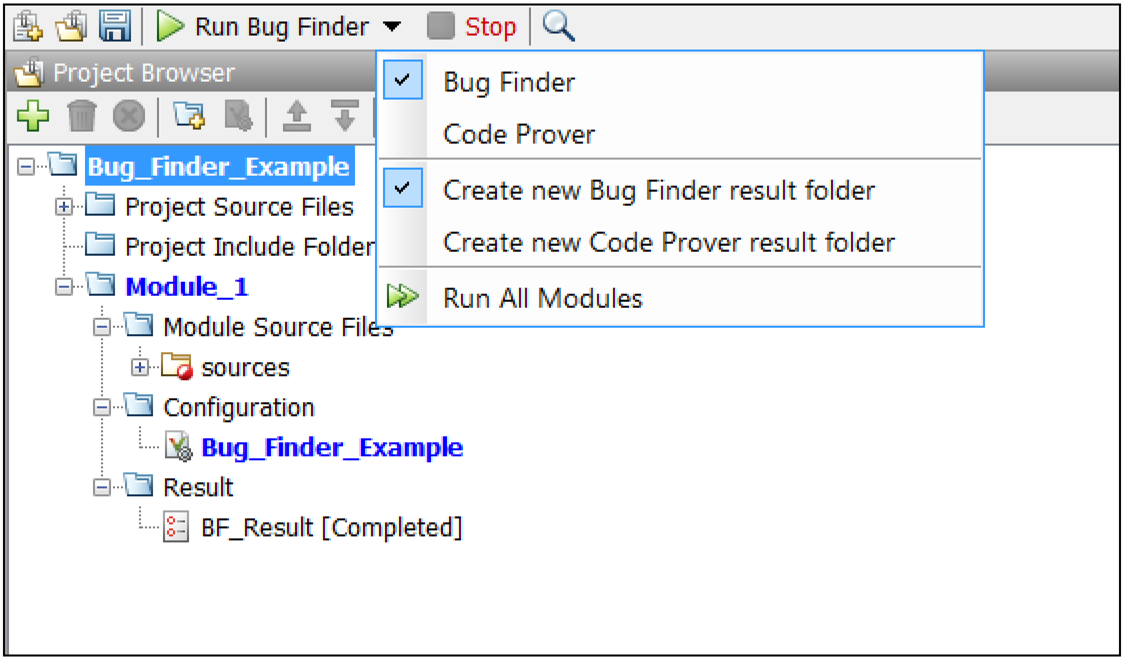
By default, the result folder is named using the convention `BF_Result_*n*`. `*n*` is the run number. To rename the folder, right-click the `Result_*n*` node. To changing the naming convention, select **Tools** > **Preferences**. Create a different naming convention on the **Project and Results Folder** tab.

Create New Result Folder for Each Run

By default, if you rerun an analysis, the previous results are overwritten.

In some cases, you want to store results of a second run in a separate folder from the previous run. For instance, if you change an analysis option and want to see the effect of this change, you want the new results to appear in a separate folder.

To create a new result folder for every run, from the **Run** button drop-down list, select **Create new Bug Finder result folder**.



If you run an incremental analysis by using the option Use fast analysis mode for Bug Finder (-fast-analysis), you cannot create a new results folder for each run. Even if you choose to create a new result folder, each new run overwrites the previous results.

Change Parent Folder of Results Folders

You can also create a parent folder for storing your results. Select **Tools** > **Preferences** and enter the parent folder location on the **Project and Results Folder** tab. If you enter a parent folder location, any new result folder is created under this parent folder.

View Results in the Polyspace Environment

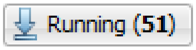
Open Results

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

Open Results From Active Project

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

-

— The analysis is running. No results to load.
- 

— The analysis is running and new results are available to start reviewing. Click this button to load the new results in the Results List. This button reactivates every time more results are available.
-

— The analysis is complete, but you have not loaded all results. Click this button to load the last set of results.

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

After analysis is over, the **Dashboard** pane shows the number of files analyzed. If some of your files were not analyzed because of compilation errors, to see which files were not analyzed, click the **Code covered by analysis** graph. For more information, see “Dashboard” on page 5-43.

Open Results File From File Browser

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

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

More About

- “Results Folder Contents” on page 5-42
- “Windows Used to Review Results” on page 5-43

Filter and Group Results

This example shows how to filter and group defects on the **Results List** pane. To organize your review of results, use filters and groups when you want to:

- Review only high-impact defects.

  For more information on impact, see "Classification of Defects by Impact" on page 5-9.
- Review certain types of defects in preference to others.

  For instance, you first want to address the defects resulting from **Missing or invalid return statement**.
- Review only new results found since the last analysis.
- Not address the full set of coding rule violations detected by the coding rules checker.
- Review only those defects that you have already assigned a certain status.

  For instance, you want to review only those defects to which you have assigned the status, `Investigate`.
- Review defects from a particular file or function. Because of continuity of code, reviewing these defects together can help you organize your review process.

  If you have written the code for a particular source file, you can review the defects only in that file.

Filter Results

You can filter results using graphs on the **Dashboard** pane or filters on the **Results List** pane. You can generate reports using only the results that are currently on display. See "Generate Reports" on page 5-20.

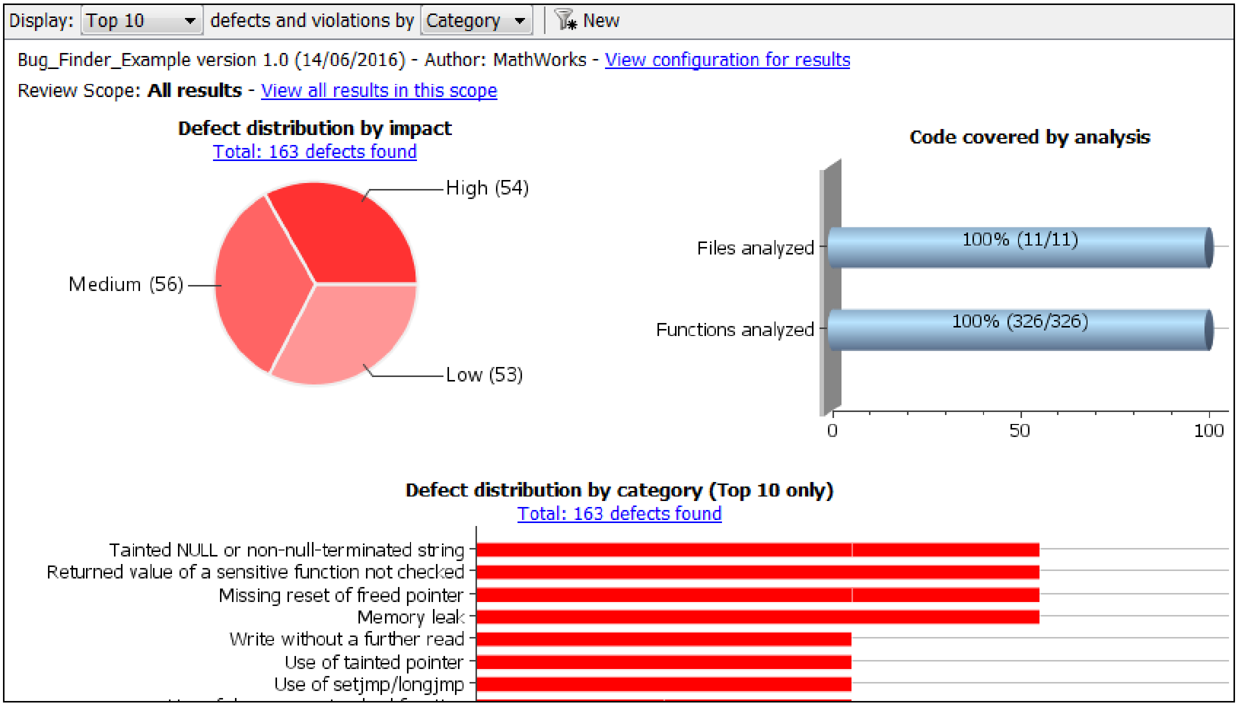
Filter Using Dashboard

The **Dashboard** pane provides a graphical overview of the results. You can click the elements on the graphs to filter results. For instance, you can use the following graphs:

- **Defect distribution by impact**: If you click a region on this pie chart that corresponds to the impact **High**, the **Results List** pane shows high-impact defects only.

- **Defect distribution by category (Top 10 only)**: If you click a column corresponding to a defect, the **Results List** pane shows instances of that defect only.
- ***`Coding rule`* violations by rule (Top 10 only)**: If you click a column corresponding to a coding rule, the **Results List** pane shows violations of that rule only.

To clear filters from the **Dashboard** pane, select the link **View results in this scope**. This action clears filters and displays the available results in the scope that you choose in the upper left menu of the **Results List** toolbar.



Display: Top 10 defects and violations by Category New

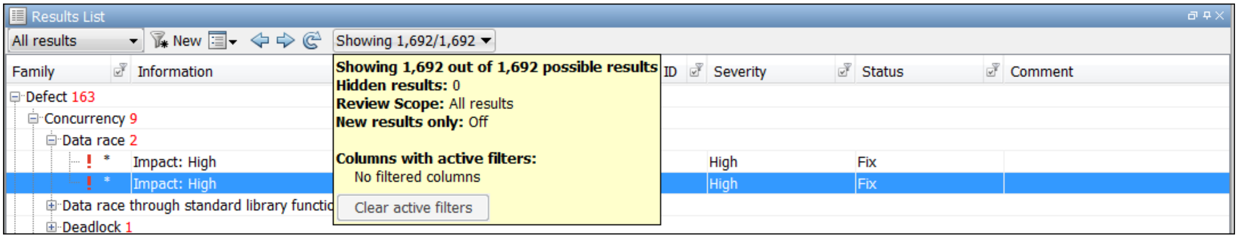
Bug_Finder_Example version 1.0 (14/06/2016) - Author: MathWorks - View configuration for results

Review Scope: **All results** - View all results in this scope

Defect distribution by impact
Total: 163 defects found
High (54)
Medium (56)
Low (53)
Code covered by analysis
Files analyzed
100% (11/11)
Functions analyzed
100% (326/326)
0
50
100
Defect distribution by category (Top 10 only)
Total: 163 defects found
Tainted NULL or non-null-terminated string
Returned value of a sensitive function not checked
Missing reset of freed pointer
Memory leak
Write without a further read
Use of tainted pointer
Use of setjmp/longjmp

Filter Using Results List

For the other filtering mechanisms, use filters on the **Results List** pane itself. To clear filters from the **Results List** pane, use the button **Clear active filters** in the **Showing** dropdown.



Showing 1,692 out of 1,692 possible results
Hidden results: 0
Review Scope: All results
New results only: Off

Columns with active filters:
No filtered columns

- To filter results from the **Results List** pane, click the ☑ icon on the appropriate column. Do one of the following:
  - Clear the boxes for the results that you want filtered from display.
  - Clear **All**. Select the boxes for the results that you want displayed.

Item to Filter	Column
Results in a certain file or function	**File** or **Function**
Results in files belonging to specific folders	**Folder**
Defects of a certain type, for instance, **Integer division by zero**	**Check** The column does not appear if you group checks by family. See “Group Results” on page 5-7.
Results with a certain severity or status	**Severity** or **Status**
Results in a certain group such as numerical or data flow	**Group**
Results with a certain impact	**Information**
Results that correspond to certain CWE IDs.	**CWE ID** For more information, see “CWE Coding Standard and Polyspace Results” on page 5-99.

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.

- To review only new results found since the last , on the **Results List** pane, select New.
- To suppress code metrics from your results, from the drop-down list in the left of the **Results List** pane toolbar, select **Defects & Rules**.

  You can increase the options on this list or create your own options. For examples, see:

  - “Suppress Certain Rules from Display in One Click” on page 3-18
  - “Limit Display of Defects” on page 5-18
  - “Review Code Metrics” on page 5-35

Note: You can also apply multiple filters. Once you apply a set of filters to your results, they are preserved for subsequent runs on the same project module. The **Results List** pane shows the number of results filtered from display. If you place your cursor on the number, you can see which filters have been applied.

Group Results

On the **Results List** pane, from the ☰▾ list, select an appropriate option.

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

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

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

  Within each class, the results are grouped by method.

Related Examples

- "Review and Fix Results" on page 5-32

More About

- "Windows Used to Review Results" on page 5-43

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 5-4.
- 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 5-4.
- 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 == 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

Resources

- 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
- 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
- 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

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
- Copy of overlapping memory
- Exception caught by value
- Exception handler hidden by previous handler
- Improper array initialization
- Incorrect pointer scaling
- Invalid assumptions about memory organization
- Invalid use of = operator
- Invalid use of floating point operation
- Memory comparison of padding data
- Memory comparison of strings
- Misuse of sign-extended character value
- Overlapping assignment
- Standard function call with incorrect arguments
- Unsafe conversion between pointer and integer
- Use of memset with size argument zero

Resources

- Opening previously opened resource

Security

- Constant block cipher initialization vector

- Constant cipher key
- 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 manipulation after chroot() without chdir(“/”)
- Inconsistent cipher operations
- 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
- Missing block cipher initialization vector
- Missing cipher final step
- Missing cipher final step
- Missing cipher key
- Misuse of readlink()
- Predictable block cipher initialization vector
- Predictable cipher key
- 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
- Weak cipher algorithm
- Weak cipher mode

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

- Unprotected dynamic memory allocation

Good practice

- Bitwise and arithmetic operation on the 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

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

Security

- Function pointer assigned with absolute address
- 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

Limit Display of Defects

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

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

If you do not want to change your analysis configuration, you can still change which defects are displayed in your results. There are two ways to filter defects from your results:

- Filter individual defects from display after each run.

  For more information, see “Filter and Group Results” on page 5-4.
- Create a set of filters that you can apply in one click.

This example shows the second approach.

1 Select **Tools** > **Preferences**.

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

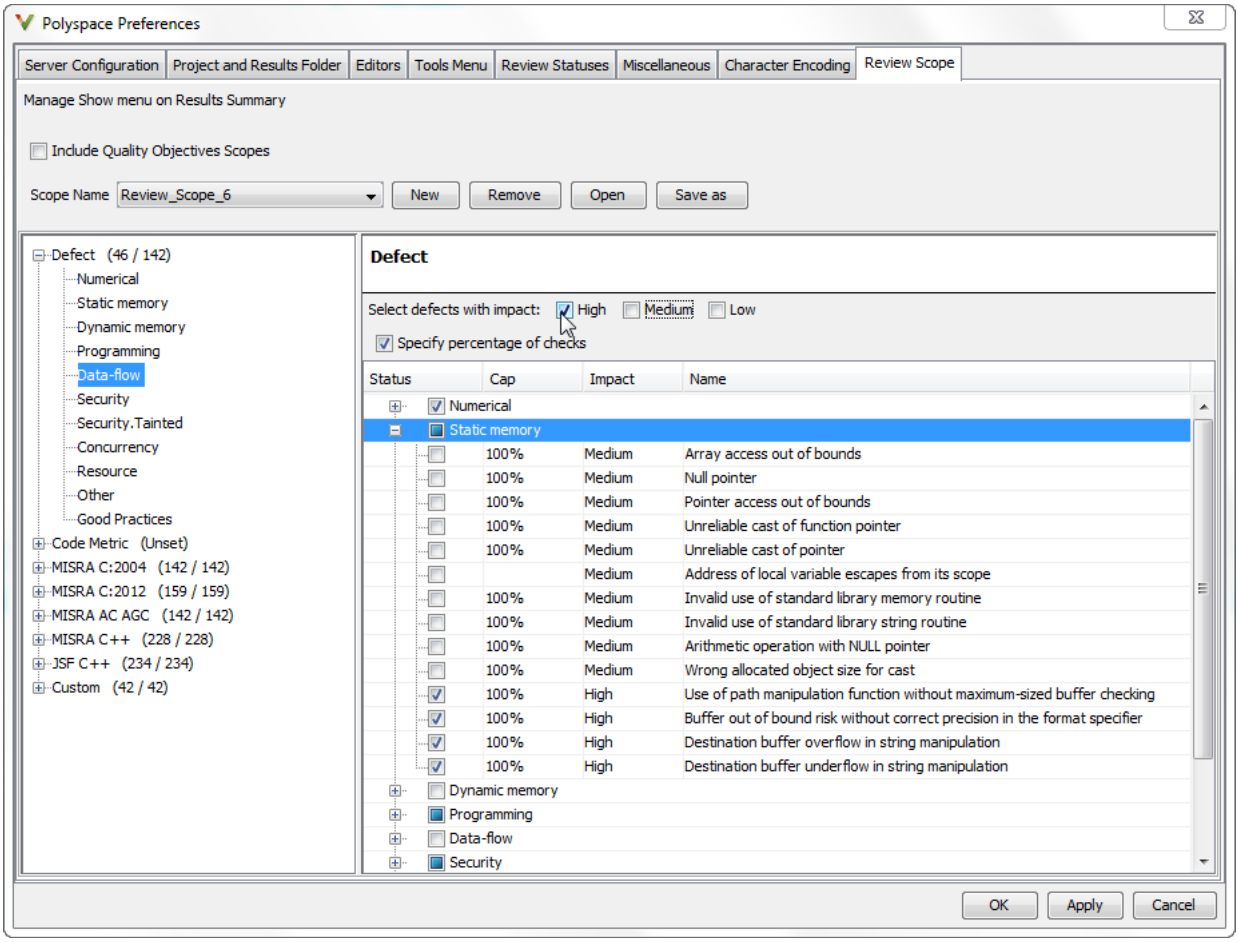
   a Select **New**. Save your filter file.

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

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

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

   To suppress all defects with a certain impact such as **Low**, clear the box next to the impact. For more information on impacts, see “Classification of Defects by Impact” on page 5-9. If only a fraction of defects with a certain impact are selected, the check box next to the impact displays a ▣ symbol.



Manage Show menu on Results Summary

□ Include Quality Objectives Scopes

Scope Name: Review_Scope_6

- Defect (46 / 142)
  - Numerical
  - Static memory
  - Dynamic memory
  - Programming
  - Data-flow
  - Security
  - Security.Tainted
  - Concurrency
  - Resource
  - Other
  - Good Practices
- Code Metric (Unset)
- MISRA C:2004 (142 / 142)
- MISRA C:2012 (159 / 159)
- MISRA AC AGC (142 / 142)
- MISRA C++ (228 / 228)
- JSF C++ (234 / 234)
- Custom (42 / 42)

Defect

Select defects with impact: ☑ High □ Medium □ Low

☑ Specify percentage of checks

Status	Cap	Impact	Name
☑ Numerical			
▪ Static memory			
□	100%	Medium	Array access out of bounds
□	100%	Medium	Null pointer
□	100%	Medium	Pointer access out of bounds
□	100%	Medium	Unreliable cast of function pointer
□	100%	Medium	Unreliable cast of pointer
□		Medium	Address of local variable escapes from its scope
□	100%	Medium	Invalid use of standard library memory routine
□	100%	Medium	Invalid use of standard library string routine
□	100%	Medium	Arithmetic operation with NULL pointer
□	100%	Medium	Wrong allocated object size for cast
☑	100%	High	Use of path manipulation function without maximum-sized buffer checking
☑	100%	High	Buffer out of bound risk without correct precision in the format specifier
☑	100%	High	Destination buffer overflow in string manipulation
☑	100%	High	Destination buffer underflow in string manipulation
□ Dynamic memory			
▪ Programming			
□ Data-flow			
▪ Security			

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

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

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

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

Generate Reports

This example shows how to generate reports 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 Results to Text File or MATLAB Table” on page 5-23

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.

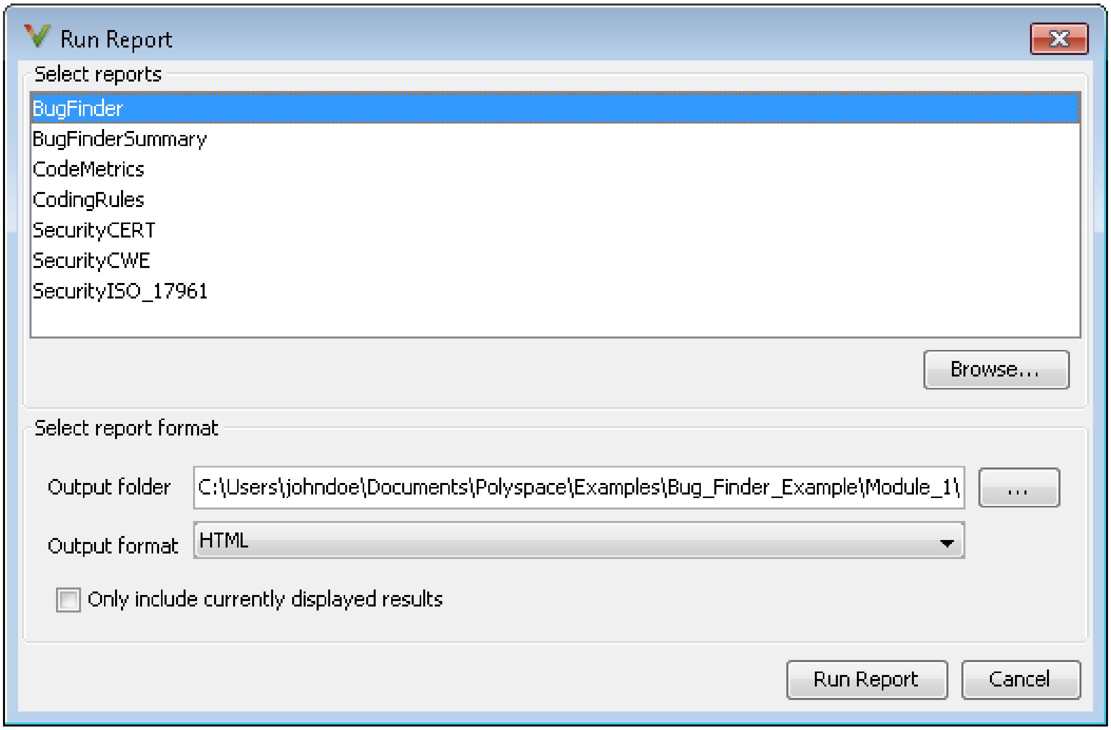
The report does not contain the line or column number for a result. Use the report for archiving, gathering statistics and checking whether results have been reviewed and addressed (for certification purposes or otherwise). To review a result in your source code, use the Polyspace user interface or your 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.



Select reports

- BugFinder
- BugFinderSummary
- CodeMetrics
- CodingRules
- SecurityCERT
- SecurityCWE
- SecurityISO_17961

Select report format

Output folder: C:\Users\johndoe\Documents\Polyspace\Examples\Bug_Finder_Example\Module_1\

Output format: HTML

□ Only include currently displayed results

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 5-4.

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)

Export Results to Text File or MATLAB Table

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 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.

User Interface	Command Line
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 5-4. **3** Select a location to save the text file and click **OK**.	Use appropriate options with the `polyspace-report-generator` command. The available options are: • `-generate-results-list-file`: Specifies that a text file must be generated. • `-results-dir` *`folder_paths`*: 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 *`folder_paths`* as follows: `"folder1, folder2, ..., folderN"` *`folder1`*`,` *`folder2`*`, ...` are paths to the folders that contain analysis results. For example:

User Interface	Command Line
	`"C:\My_project \Module_1\results, C: \My_project\Module_2\Results"` To merge the text files, use the `join` function.

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

You can read your Polyspace analysis results into a MATLAB table. 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.

See also polyspace.BugFinderResults.

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). Some of the result information includes:

- ID: Unique number for a result for the current analysis
- Family: Defect, Code Metric, MISRA C:2012, and so on.
- Group: Defect groups, MISRA C:2012 groups, etc.
- Check: Defect names, MISRA C:2012 coding rule number, and so on.
- New: Whether the result is new compared to the last analysis on the same code
- Full path to file
- CWE, CERT C99 or ISO/IEC TS 17961 IDs corresponding to the Bug Finder results.

See “CWE Coding Standard and Polyspace Results” on page 5-99, “CERT C Coding Standard and Polyspace Results” on page 5-130 and “ISO/IEC TS 17961 Coding Standard and Polyspace Results” on page 5-166.

- Function
- Status, Severity, Comment: Information that you enter about a result.

For more information, see “Results List” on page 5-47. Though you cannot identify the location of a result in your source code via the text file, you can parse the file and generate graphs or statistics about your results.

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.

Generate Graphs from Results

This example shows how to create a pie chart showing the distribution of defects by defect groups on page 5-56.

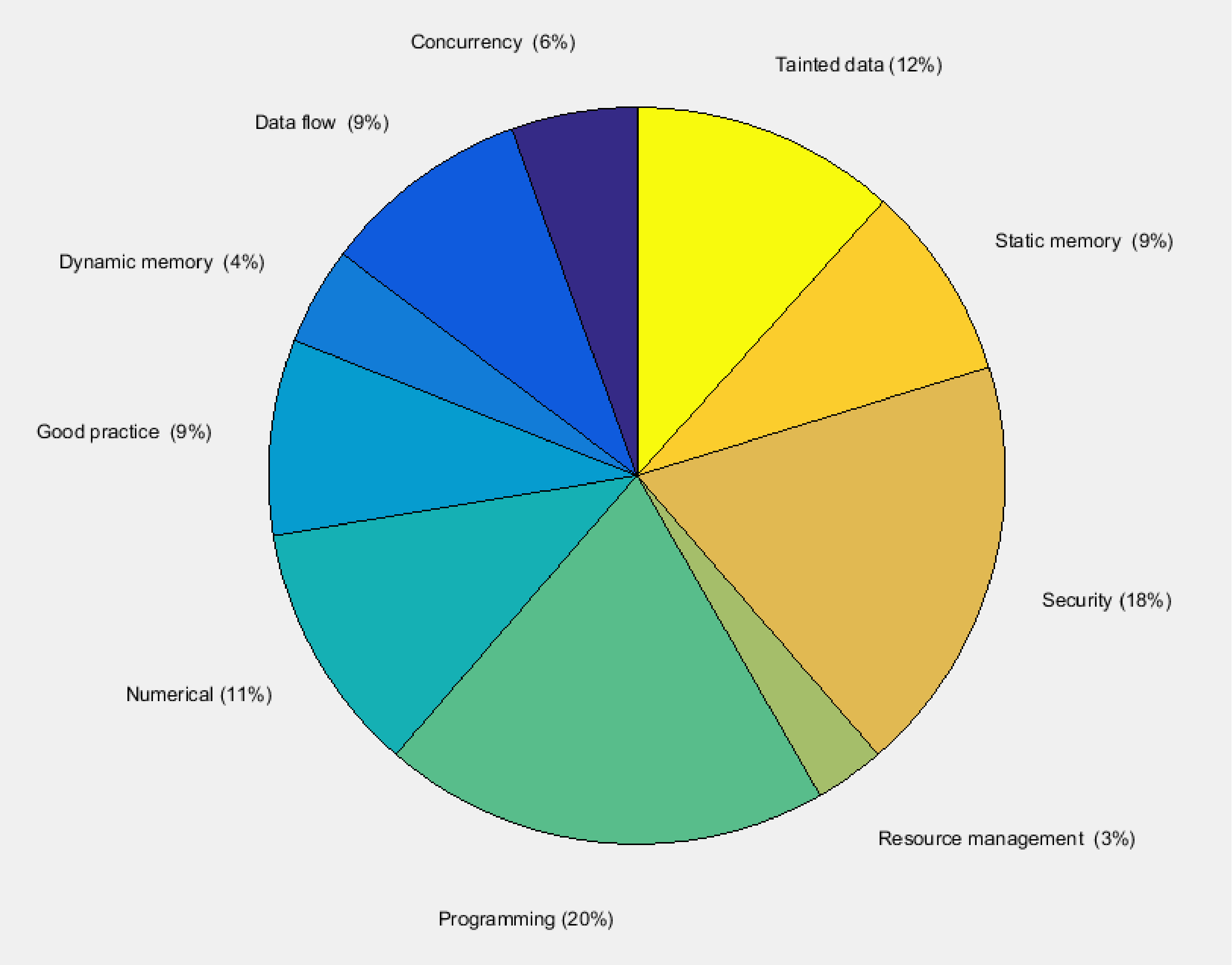
```
% Copy a demo result set to a temporary folder
resPath = fullfile(matlabroot,'polyspace','examples','cxx','Bug_Finder_Example', ...
'Module_1','BF_Result');
userResPath = tempname;
copyfile(resPath,userResPath);

% 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
pie(defectTable.Group)
```

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 create a histogram (see `histogram`) or a heatmap chart (see `heatmap`).

When you execute the script, you see a distribution of defects by defect group.



Customize Existing 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 5-20.
- 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.

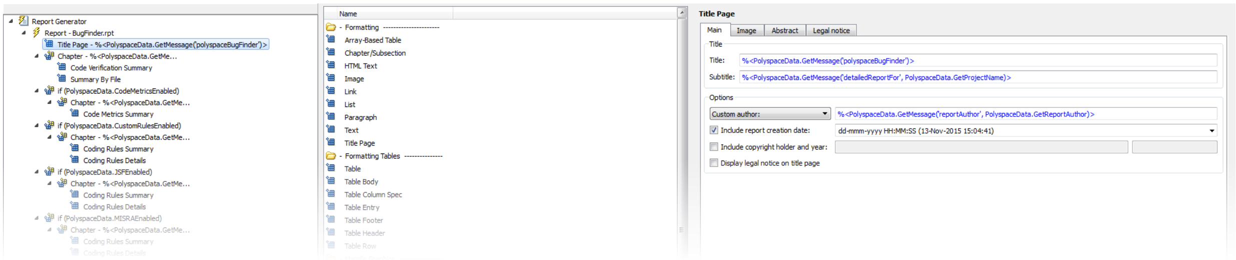
In this example, you view the components of the **BugFinder** template.

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.



```
Report Generator
  Report - BugFinder.rpt
    Title Page - %<PolyspaceData.GetMessage('polyspaceBugFinder')>
    Chapter - %<PolyspaceData.GetMe...
      Code Verification Summary
      Summary By File
    if (PolyspaceData.CodeMetricsEnabled)
      Chapter - %<PolyspaceData.GetMe...
        Code Metrics Summary
    if (PolyspaceData.CustomRulesEnabled)
      Chapter - %<PolyspaceData.GetMe...
        Coding Rules Summary
        Coding Rules Details
    if (PolyspaceData.JSFEnabled)
      Chapter - %<PolyspaceData.GetMe...
        Coding Rules Summary
        Coding Rules Details
    if (PolyspaceData.MISRAEnabled)
      Chapter - %<PolyspaceData.GetMe...
        Coding Rules Summary
        Coding Rules Details
```

```
Name
- Formatting ---------------------
Array-Based Table
Chapter/Subsection
HTML Text
Image
Link
List
Paragraph
Text
Title Page
- Formatting Tables ---------------
Table
Table Body
Table Column Spec
Table Entry
Table Footer
Table Header
Table Row
```

Title Page

Title

Title: %<PolyspaceData.GetMessage('polyspaceBugFinder')>

Subtitle: %<PolyspaceData.GetMessage('detailedReportFor', PolyspaceData.GetProjectName)>

Options

Custom author: %<PolyspaceData.GetMessage('reportAuthor', PolyspaceData.GetReportAuthor)>

☑ Include report creation date: dd-mmm-yyyy HH:MM:SS (13-Nov-2015 15:04:41)

☐ Include copyright holder and year:

☐ Display legal notice on title page

Some components of the **BugFinder** template and their purpose are described below.

Component	Purpose
Title Page (MATLAB Report Generator)	Inserts title page in the beginning of report
Chapter/Subsection (MATLAB Report Generator)	Groups portions of report into sections with titles
Code Verification Summary	Inserts summary table of Polyspace analysis results
Logical If (MATLAB Report Generator)	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 “Create Reports 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.

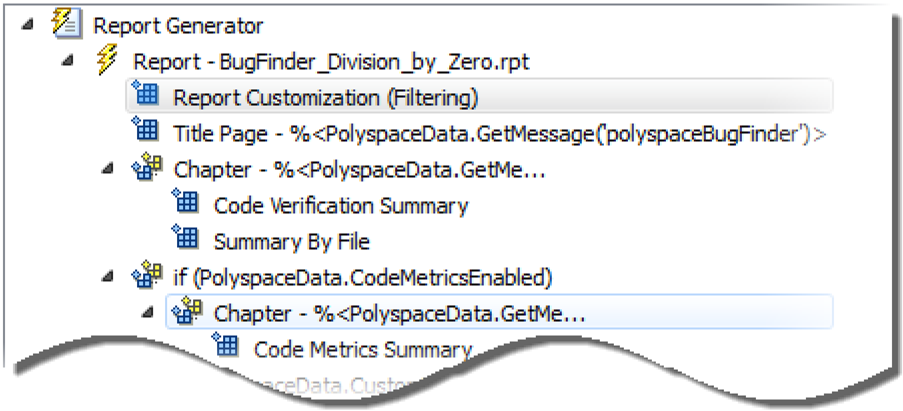
In this example, you add a component to the **BugFinder** template so that the template includes only **Integer division by zero** and **Float division by zero** defects in a report.

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:

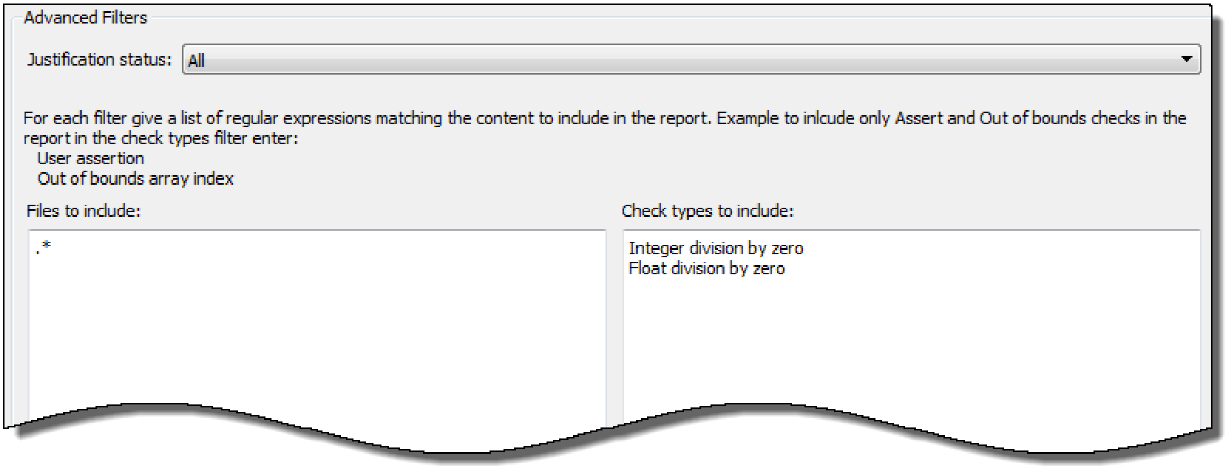
   a Drag the component Report Customization (Filtering) from the middle pane and drop it above the **Title Page** component. The positioning of the component ensures that the filters apply to the full report and not one chapter of the report.



```
Report Generator
  Report - BugFinder_Division_by_Zero.rpt
    Report Customization (Filtering)
    Title Page - %<PolyspaceData.GetMessage('polyspaceBugFinder')>
    Chapter - %<PolyspaceData.GetMe...
      Code Verification Summary
      Summary By File
    if (PolyspaceData.CodeMetricsEnabled)
      Chapter - %<PolyspaceData.GetMe...
        Code Metrics Summary
        aceData.Custo
```

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.



Advanced Filters

Justification status: All

For each filter give a list of regular expressions matching the content to indude in the report. Example to inlcude only Assert and Out of bounds checks in the report in the check types filter enter:
User assertion
Out of bounds array index

Files to indude:
.*

Check types to include:
Integer division by zero
Float division by zero

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**.

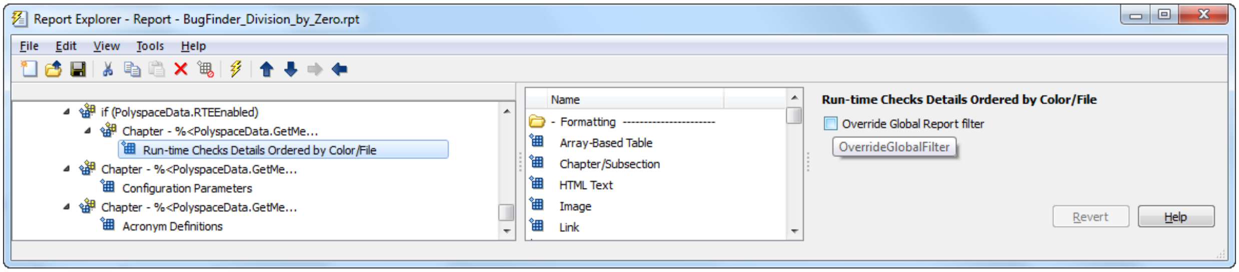
3 Change an existing chapter-specific component so that it does not override the global filter you applied in the previous step. If you prevent the overriding, the chapter-specific component follows the filtering specifications in the global component.

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.



Run-time Checks Details Ordered by Color/File

□ Override Global Report filter

Report Explorer - Report - BugFinder_Division_by_Zero.rpt
File
Edit
View
Tools
Help
if (PolyspaceData.RTEEnabled)
Chapter - %<PolyspaceData.GetMe...
Run-time Checks Details Ordered by Color/File
Chapter - %<PolyspaceData.GetMe...
Configuration Parameters
Chapter - %<PolyspaceData.GetMe...
Acronym Definitions
Name
- Formatting --------------------
Array-Based Table
Chapter/Subsection
HTML Text
Image
Link
OverrideGlobalFilter
Revert
Help

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 5-20.

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.

Review and Fix Results

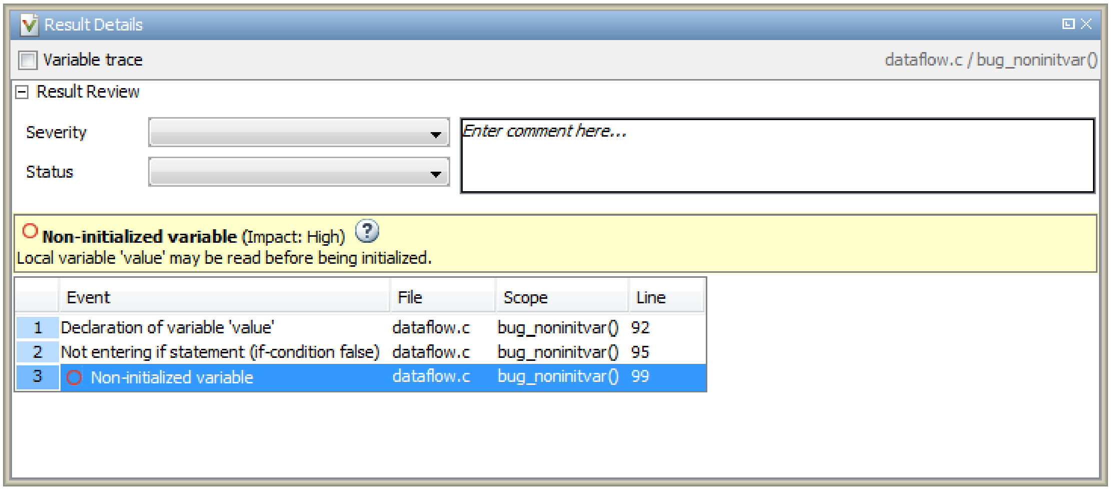
This example shows how to review and comment your Bug Finder results. When reviewing results, you can assign a status to the defects and enter comments to describe the results of your review. These actions help you to track the progress of your review and avoid reviewing the same defect twice.

In this section...
"Assign and Save Comments" on page 5-32
"Import Review Comments from Previous Analysis" on page 5-33

Assign and Save Comments

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

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



○ **Non-initialized variable** (Impact: High)
Local variable 'value' may be read before being initialized.

	Event	File	Scope	Line
1	Declaration of variable 'value'	dataflow.c	bug_noninitvar()	92
2	Not entering if statement (if-condition false)	dataflow.c	bug_noninitvar()	95
3	○ Non-initialized variable	dataflow.c	bug_noninitvar()	99

2 Investigate the result further. Determine whether to fix your code, review the result later, or retain the code but provide some explanation.

3 On the **Results List** or **Result Details** pane, provide the following review information for the result:

- **Severity** to describe how critical you consider the issue.

- **Status** to describe how you intend to address the issue.

  You can also create your own status or associate justification with an existing status. Select **Tools** > **Preferences** and create or modify statuses on the **Review Statuses** tab.
- **Comment** to describe any other information about the result.

4 To provide review information for several results together, select the results. Then, provide review information for a single result.

  To select the results in a group:

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

    To group certain results together, use the column headers on the **Results List** pane.
  - If the results are not contiguous, **Ctrl**-left click each result.
  - If the results belong to the same group and have the same color, right-click one result. From the context menu, select **Select All *Type* Results**.

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

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

Import Review Comments from Previous Analysis

After you have reviewed analysis results, you can reuse your review comments for subsequent analyses. By default, Polyspace Bug Finder automatically imports comments from the last analysis on the module.

Import Comments from Another Analysis

You can import comments directly from another Bug Finder 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 the corresponding Code Prover result. 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.

1 Open your results.

2 Select **Tools** > **Import Comments**.

3 Navigate to the folder containing your previous results.

4 Select the results file and then click **Open**.

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

Disable Automatic Comment Import from Last Analysis

1 Select **Tools** > **Preferences**, which opens the Polyspace Preferences dialog box.

2 Select the **Project and Results Folder** tab.

3 Under **Import Comments**, clear **Automatically import comments from last** .

4 Click **OK**.

   After you set this preference, for every run, the software does not import review comments from the last run.

Related Examples

- "Filter and Group Results" on page 5-4
- "Add Annotations from the Polyspace Interface" on page 1-44

More About

- "Windows Used to Review Results" on page 5-43

Review Code Metrics

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

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

- **User interface**: On the **Configuration** pane, select **Coding Rules & Code Metrics**. Select **Calculate Code Metrics**.
- **Command line**: 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**.

In this section...
“Impose Limits on Metrics” on page 5-35
“Comment and Justify Limit Violations” on page 5-38

Impose Limits on Metrics

1. Select **Tools** > **Preferences**.
2. On the **Review Scope** tab, do one of the following:

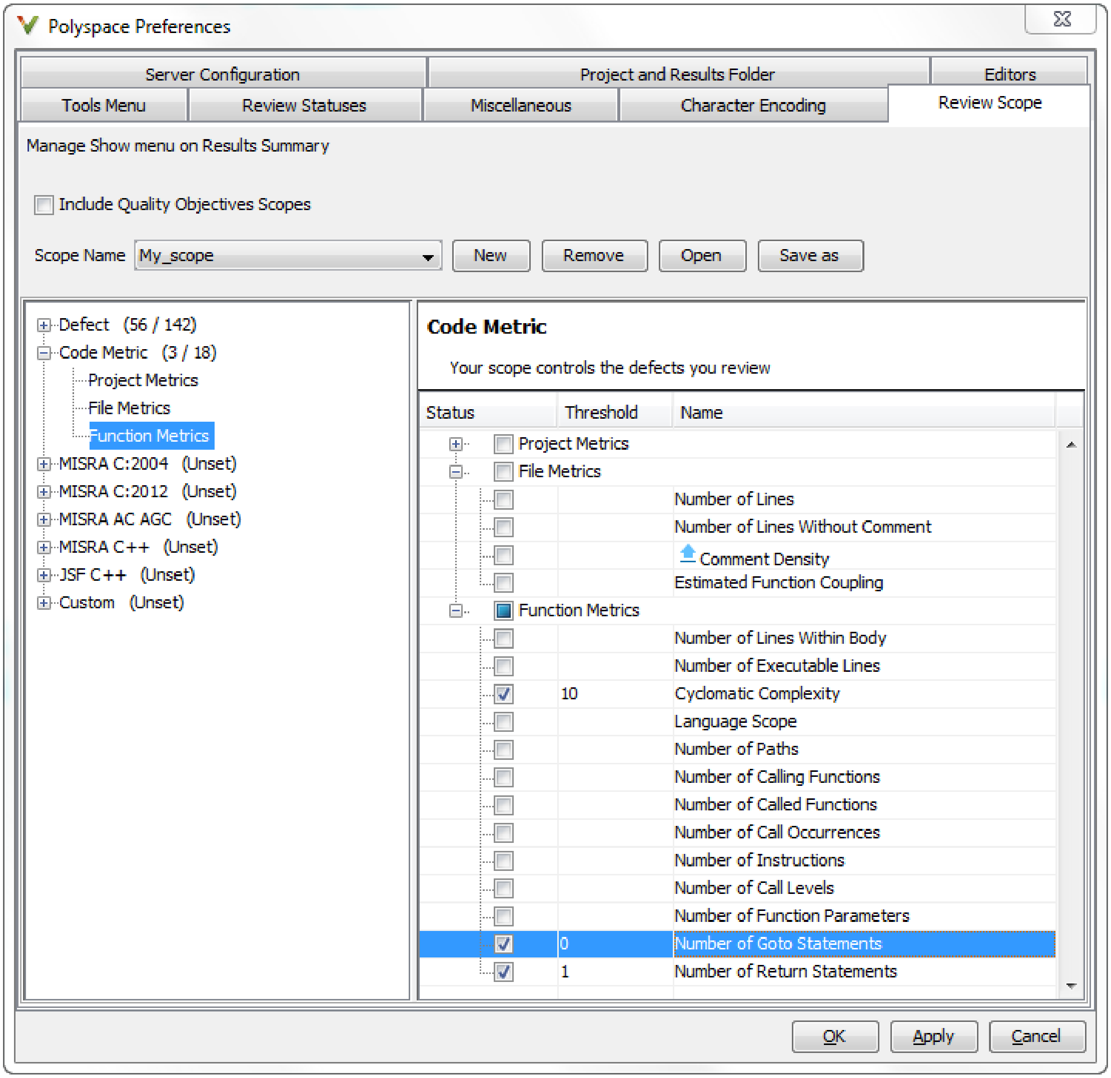
   - To use a predefined limit, select **Include Quality Objectives Scopes**.

The **Scope Name** list shows the additional option `HIS`. The option `HIS` displays the HIS code metrics on page 5-90 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.



Manage Show menu on Results Summary

□ Include Quality Objectives Scopes

Scope Name: My_scope

- Defect (56 / 142)
- Code Metric (3 / 18)
  - Project Metrics
  - File Metrics
  - Function Metrics
- MISRA C:2004 (Unset)
- MISRA C:2012 (Unset)
- MISRA AC AGC (Unset)
- MISRA C++ (Unset)
- JSF C++ (Unset)
- Custom (Unset)

Code Metric

Your scope controls the defects you review

Status	Threshold	Name
□ Project Metrics		
□ File Metrics		
□		Number of Lines
□		Number of Lines Without Comment
□		Comment Density
□		Estimated Function Coupling
■ Function Metrics		
□		Number of Lines Within Body
□		Number of Executable Lines
☑	10	Cyclomatic Complexity
□		Language Scope
□		Number of Paths
□		Number of Calling Functions
□		Number of Called Functions
□		Number of Call Occurrences
□		Number of Instructions
□		Number of Call Levels
□		Number of Function Parameters
☑	0	Number of Goto Statements
☑	1	Number of Return Statements

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 “Review and Fix Results” on page 5-32.

Navigate to Root Cause of Defect

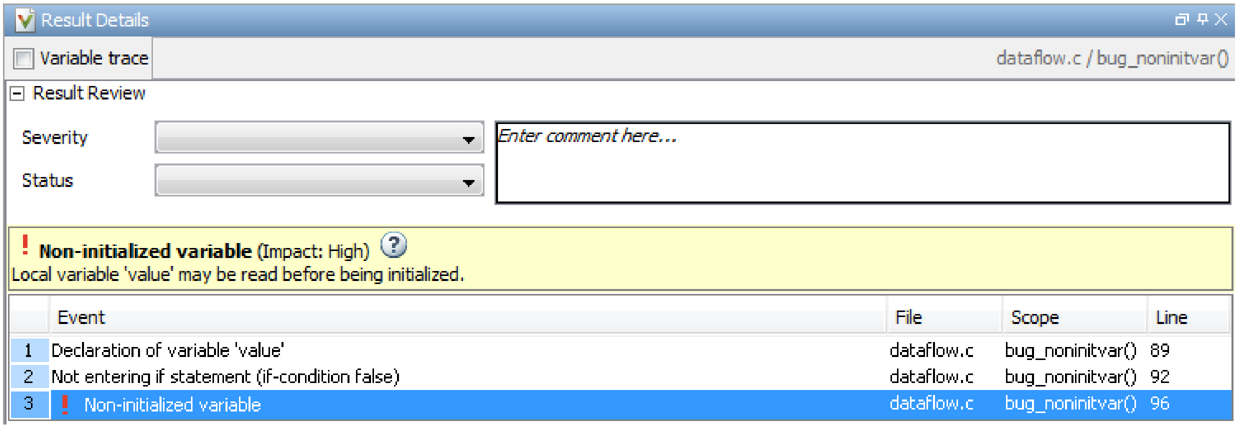
Through the Polyspace Bug Finder user interface, you can navigate to the root cause of a defect in your source code. If you select a result on the **Results List** pane, you see the immediate location of the defect on the **Source** pane. However, the defect can be related to previous statements in your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

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

Follow Code Sequence Causing Defect

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

For instance, if an **Array Access Out of Bounds** error occurs in a loop, the **Result Details** pane shows updates to the array index that occur inside the loop. The update statements might physically occur in your code before or after the array access, but because the statements occur in a loop, they are related to the array access.



! **Non-initialized variable** (Impact: High)
Local variable 'value' may be read before being initialized.

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

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

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

Navigate to Identifier Definition

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

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

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

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

If a definition is not available to Polyspace, selecting the option takes you to the declaration. For instance, Polyspace Bug Finder displays results in real time as they are produced. The software displays results on some files while others are not yet analyzed. In your results, if you select a function and then select **Go To Definition**, and the function definition is not yet analyzed, selecting the option takes you to the function declaration.

Navigate to Identifier References

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

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

To navigate to previous locations where an identifier is used:

1 Right-click the identifier name and select **Search For All References**.

   The search results appear on the **Search** pane with the current location highlighted.

2. Click each search result, starting backward from the highlighted result.
3. The option **Search for All References** is not available in some cases. For instance, if you right-click a C++ `virtual` function, this option is not available.

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

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

   a. Right-click on the **Source** pane and select **Create Duplicate Code Window**.

   b. Right-click on the tab showing the duplicate file name and select **New Vertical Group**.

   c. Perform the navigation steps in the original file window while the defect still appears on the duplicate file window.

   d. After reviewing the defect, click the button on the **Results List** pane to return to the defect location in the original file window. Close the duplicate window.

Related Examples

- “Review and Fix Results” on page 5-32

More About

- “Source” on page 5-49
- “Result Details” on page 5-55

Results Folder Contents

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

By default, your results are saved in your project folder in a folder called `Result`. To use a different folder, see "Specify Results Folder" on page 4-6.

Files in the Results Folder

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

- `Polyspace_`*`release_project_name_date-time`*`.log` — A log file associated with each analysis.
- `ps_results.psbf` — An encrypted file containing your Polyspace results. Open this file in the Polyspace environment to view your results.
- `ps_sources.db` — A non-encrypted database file listing source files and macros.
- `drs-template.xml` — A template generated when you use constraint specification.
- `ps_comments.db` — An encrypted database file containing your comments and justifications.
- `comments_bak` — A subfolder used to import comments between results.
- `.status` and `.settings` — Two folders 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

Related Examples

- "Specify Results Folder" on page 4-6
- "Open Results" on page 5-2

Windows Used to Review Results

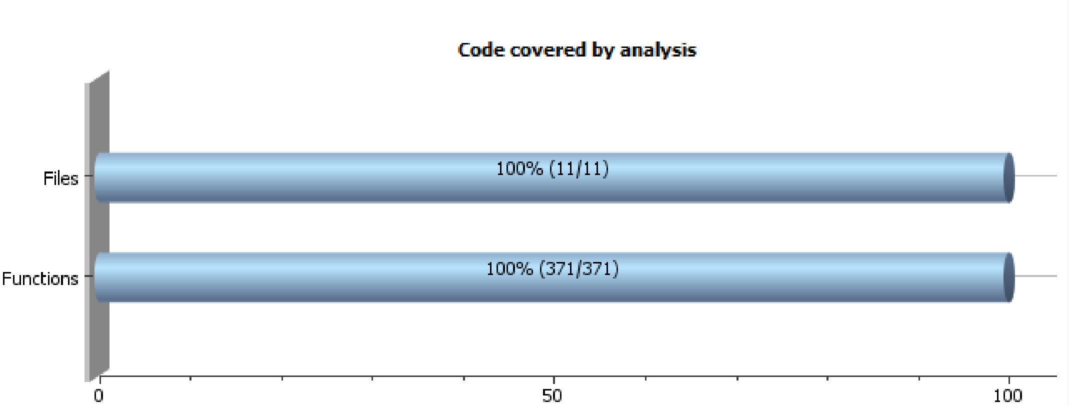
In this section...

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**



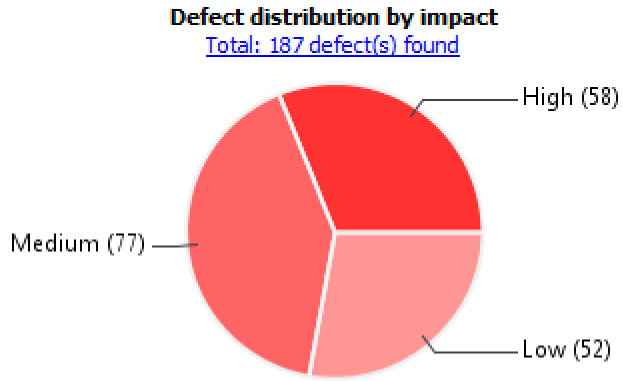
From this graph you can obtain the following information:

- **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.

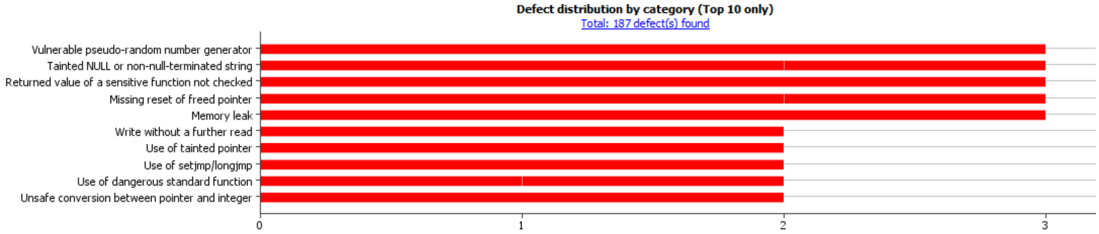
- **Functions**: Ratio of analyzed functions to total number of functions *in the analyzed files*. If the analysis of a function takes longer than a certain threshold value, Polyspace Bug Finder does not analyze the function.

- **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 5-9.

- **Defect distribution by category or file**

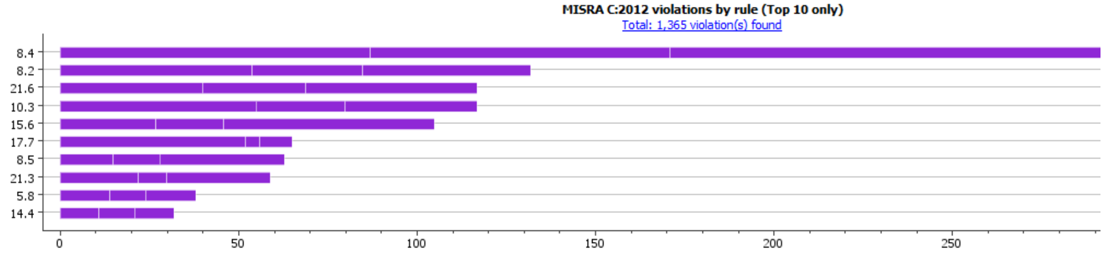


From this graph you can obtain the following information.

	Category	File
Top 10	The ten defect types with the highest number of individual defects.	The ten source files with the highest number of defects.

	Category	File
	• 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.	• Each column represents a file and is divided into the: • Defect type with highest number of defects in this file. • Defect type with second highest number of defects in this file. • All other defect types in this file. Place your cursor on a column to see the defect type name and number of defects of this type in this file. • The x-axis represents the number of defects. Use this view to organize your check review starting at files with more defects.
Bottom 10	The ten defect types with the lowest number of individual defects. Each column on the graph is divided the same way as the **Top 10** defect types. Use this view to organize your check review starting at defect types with fewer individual defects.	The ten source files with the lowest number of defects. Each column on the graph is divided the same way as the **Top 10** files. Use this view to organize your check review starting at files with fewer defects.

- **Coding rule violations by rule or file**



For every type of coding rule that you check (MISRA, JSF, or custom), the **Dashboard** contains a graph of the rule violations.

From this graph you can obtain the following information.

	Category	File
Top 10	The ten rules with the highest number of violations. • Each column represents a rule number and is divided into the: • File with highest number of violations of this rule. • File with second highest number of violations of this rule. • All other files with violations of this rule. Place your cursor on a column to see the file name and number of violations of this rule in the file. • The x-axis represents the number of rule violations. Use this view to organize your review starting at rules with more violations.	The ten source files containing the highest number of violations. • Each column represents a file and is divided into the: • Rule with highest number of violations in this file. • Rule with second highest number of violations in this file. • All other rules violated in this file. Place your cursor on a column to see the rule number and number of violations of the rule in this file. • The x-axis represents the number of rule violations. Use this view to organize your review starting at files with more rule violations.

	Category	File
Bottom 10	The ten rules with the lowest number of violations. Each column on the graph is divided in the same way as the **Top 10** rules. Use this view to organize your review starting at rules with fewer violations.	The ten source files containing the lowest number of rule violations. Each column on the graph is divided in the same way as the **Top 10** files. Use this view to organize your review starting at files with fewer rule violations.

For a list of supported coding rules, see “Supported MISRA C:2004 and MISRA AC AGC Rules” on page 2-14, “Supported MISRA C++ Coding Rules” on page 2-88 and “Supported JSF C++ Coding Rules” on page 2-117.

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 5-4.
- View the configuration used to obtain the result. Select the link **View configuration for results**.

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 5-56.
- **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.
Check	Result name, for instance: • For defects: Defect name • For coding rule violations: Coding rule number
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

Attribute	Description
Severity	Level of severity you have assigned to the result. The possible levels are: • `High` • `Medium` • `Low` • `Not a defect`
Status	Review status you have assigned to the result. The possible statuses are: • `Fix` • `Improve` • `Investigate` • `Justified` • `No action planned` • `Other`
Comments	Comments you have entered about the result

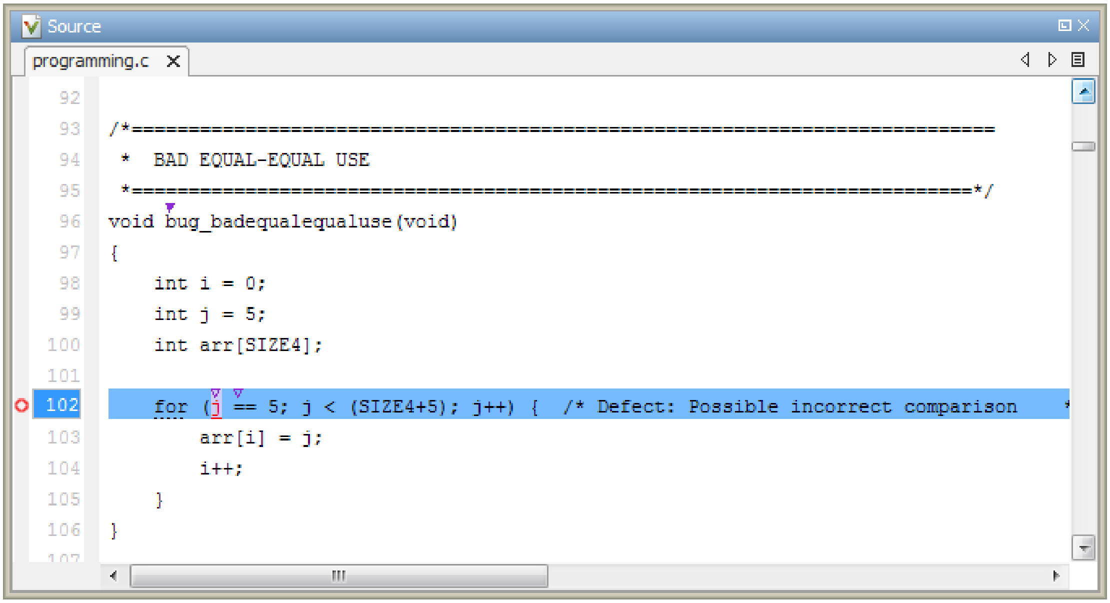
To show or hide any of the columns, right-click anywhere on the column titles. From the context menu, select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through the results. For more information, see “Review and Fix Results” on page 5-32.
- Organize your result review using filters on the columns. For more information, see “Filter and Group Results” on page 5-4.

Source

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



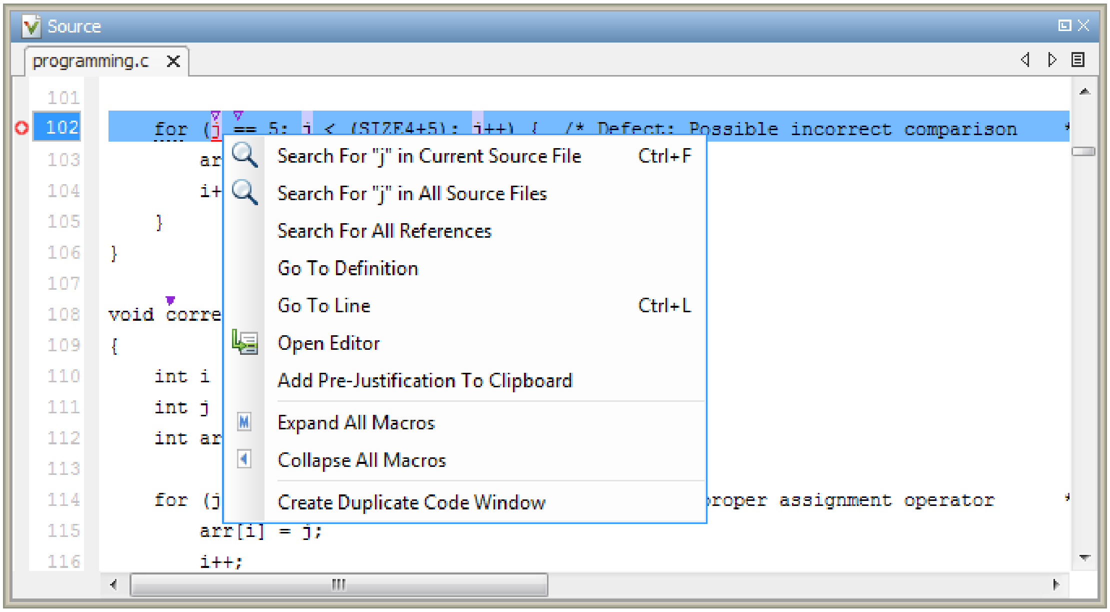
```

/*=========================================================================
 *  BAD EQUAL-EQUAL USE
 *========================================================================*/
void bug_badequalequaluse(void)
{
    int i = 0;
    int j = 5;
    int arr[SIZE4];

    for (j == 5; j < (SIZE4+5); j++) {  /* Defect: Possible incorrect comparison  *
        arr[i] = j;
        i++;
    }
}
```

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:


Source
programming.c
Search For "j" in Current Source File Ctrl+F
Search For "j" in All Source Files
Search For All References
Go To Definition
Go To Line Ctrl+L
Open Editor
Add Pre-Justification To Clipboard
Expand All Macros
Collapse All Macros
Create Duplicate Code Window

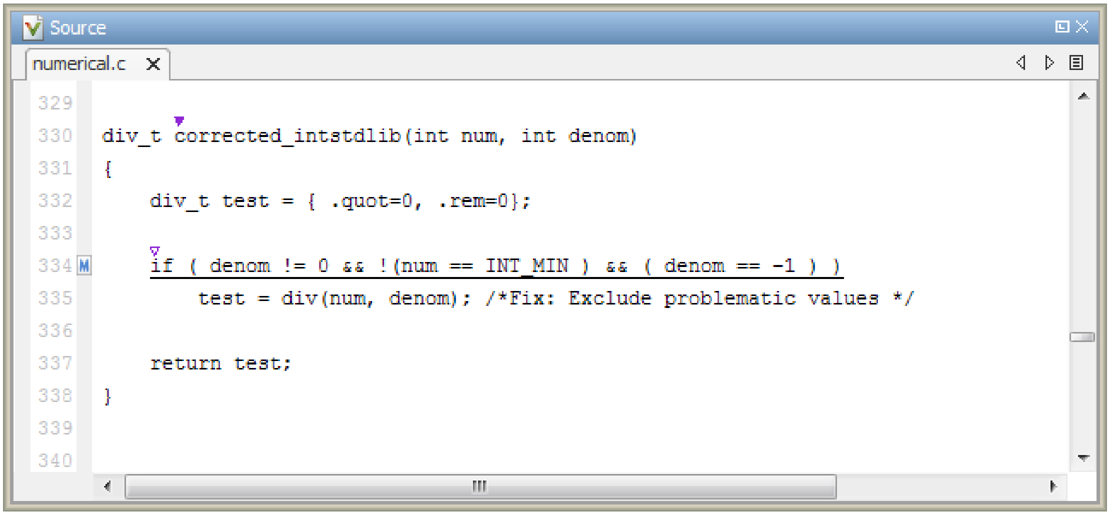
```
    for (j == 5; i < (SIZE4+5); i++) {   /* Defect: Possible incorrect comparison
        ar
        i+
    }
}

void corre
{
    int i
    int j
    int ar

    for (j                                             proper assignment operator
        arr[i] = j;
        i++;
```

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.

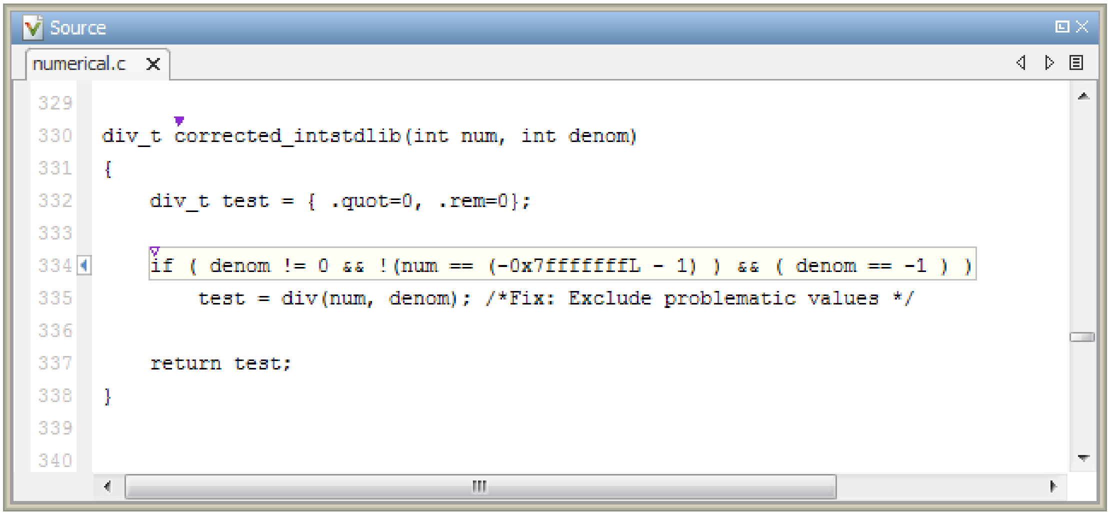
Expand Macros

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.



```
div_t corrected_intstdlib(int num, int denom)
{
    div_t test = { .quot=0, .rem=0};

    if ( denom != 0 && !(num == INT_MIN ) && ( denom == -1 ) )
        test = div(num, denom); /*Fix: Exclude problematic values */

    return test;
}
```

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



```
div_t corrected_intstdlib(int num, int denom)
{
    div_t test = { .quot=0, .rem=0};

    if ( denom != 0 && !(num == (-0x7fffffffL - 1) ) && ( denom == -1 ) )
        test = div(num, denom); /*Fix: Exclude problematic values */

    return test;
}
```

To display the normal source code again, click the line away from the box, for example, on the ◀ icon.

To display or hide the content of *all* macros:

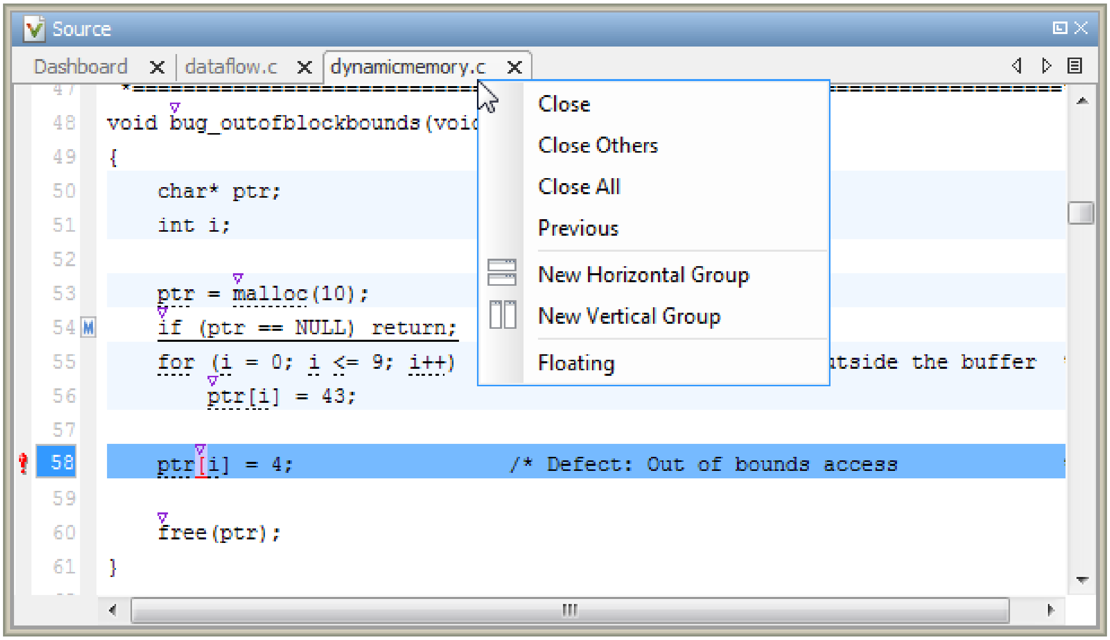
1 Right-click anywhere on the source.
2 From the context menu, select either **Expand All Macros** or **Collapse All Macros**.

Note: The **Result Details** pane also allows you to view the contents of a macro if the result you select lies within a macro.

Manage Multiple Files in Source Pane

You can view multiple source files in the **Source** pane.

Right-click on the **Source** pane toolbar.



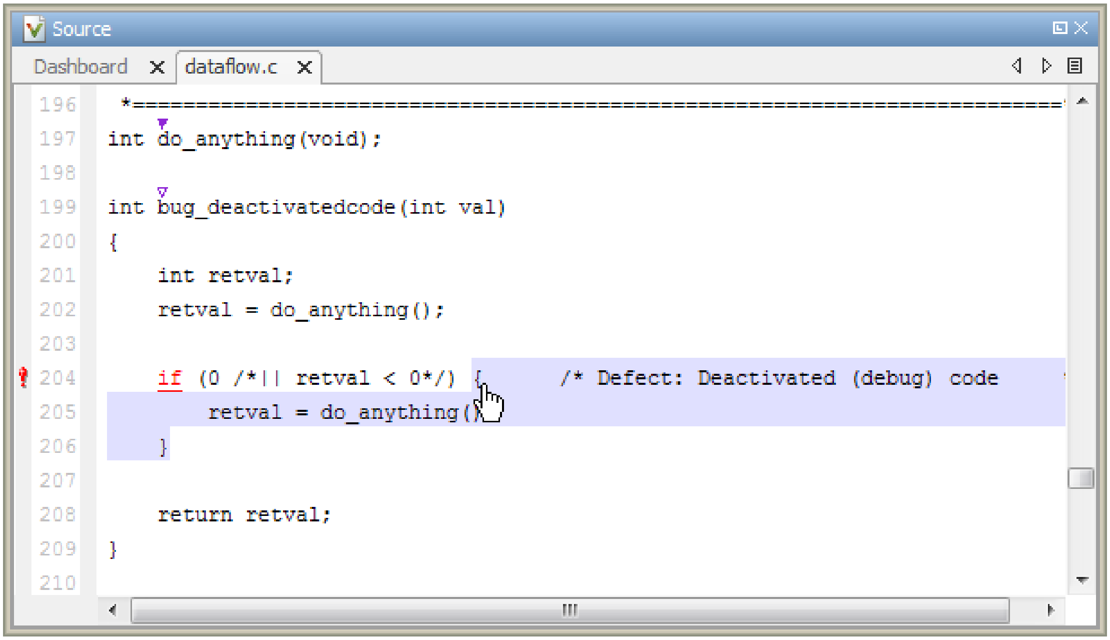
```
 *===========================                              ====================
void bug_outofblockbounds(voi
{
    char* ptr;
    int i;

    ptr = malloc(10);
    if (ptr == NULL) return;
    for (i = 0; i <= 9; i++)                              tside the buffer
        ptr[i] = 43;

    ptr[i] = 4;                /* Defect: Out of bounds access

    free(ptr);
}
```

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.

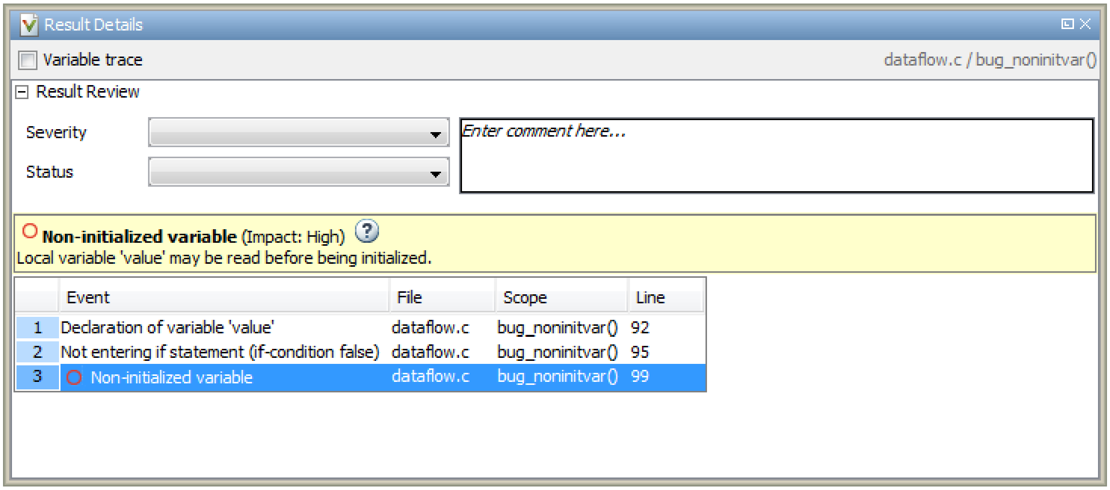


```
 *==========================================================================
int do_anything(void);

int bug_deactivatedcode(int val)
{
    int retval;
    retval = do_anything();

    if (0 /*|| retval < 0*/) {       /* Defect: Deactivated (debug) code
        retval = do_anything();
    }

    return retval;
}
```

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.



Result Details

□ Variable trace

dataflow.c / bug_noninitvar()

⊟ Result Review

Severity

Status

Enter comment here...

○ **Non-initialized variable** (Impact: High) ?

Local variable 'value' may be read before being initialized.

	Event	File	Scope	Line
1	Declaration of variable 'value'	dataflow.c	bug_noninitvar()	92
2	Not entering if statement (if-condition false)	dataflow.c	bug_noninitvar()	95
3	○ Non-initialized variable	dataflow.c	bug_noninitvar()	99

- 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.

For more information, see “Navigate to Root Cause of Defect” on page 5-39.

Bug Finder Defect Groups

In this section...

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`

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. In R2017a, 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

Name	Description
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

Name	Description
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	Statement with mixed bitwise and arithmetic operations

Name	Description
(bitwise_arith_mix)	
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.

Rule	Description
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.

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.

Rule	Description
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.

Rule	Description
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.

Rule	Description
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.

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.

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.

Rule	Description
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.

Rule	Description
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.
8.8	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.

Rule	Description
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.

Rule	Description
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.

Rule	Description
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.

Rule	Description
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.

Rule	Description
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.

Rule	Description
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.
5-2-1	Each operand of a logical && or || shall be a postfix - expression.
5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.
5-2-3	Casts from a base class to a derived class should not be performed on polymorphic types.
5-2-4	C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.
5-2-5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.
5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.
5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.

Rule	Description
5-2-9	A cast should not convert a pointer type to an integral type.
5-2-10	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.
5-2-11	The comma operator, && operator and the || operator shall not be overloaded.
5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer.
5-3-1	Each operand of the ! operator, the logical && or the logical || operators shall have type bool.
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
5-3-3	The unary & operator shall not be overloaded.
5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects.
5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.
5-14-1	The right hand operand of a logical && or || operator shall not contain side effects.
5-18-1	The comma operator shall not be used.
5-19-1	Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Statements

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.

Rule	Description
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.
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.

Rule	Description
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.

Rule	Description
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.

Rule	Description
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.

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.
15-5-1	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.

Rule	Description
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.
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.

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 "Review Code Metrics" on page 5-35.

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 (Polyspace Code Prover), 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

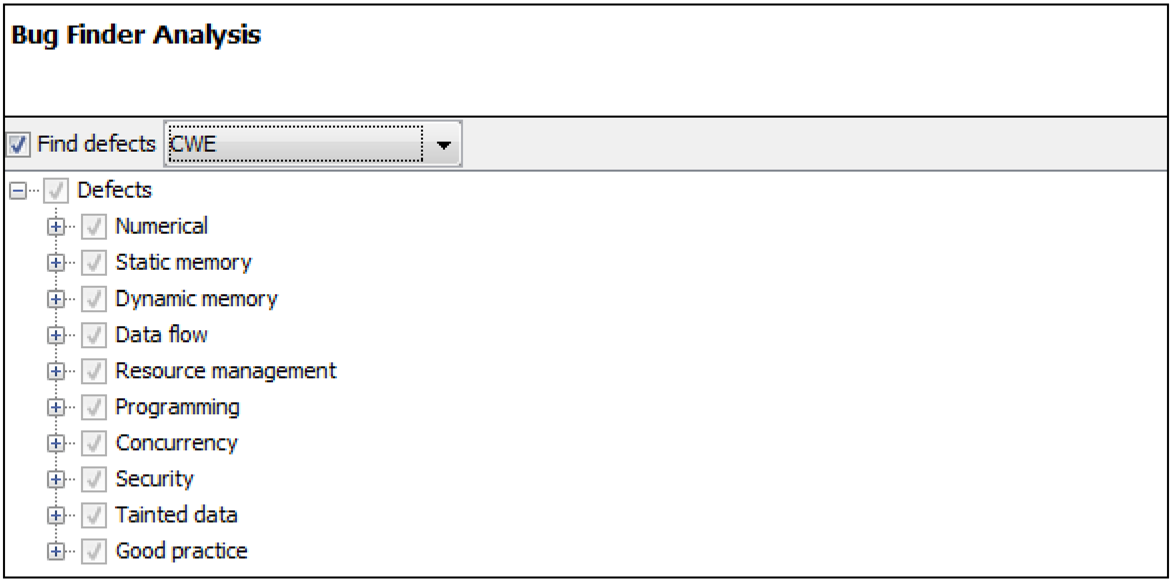
Check Code for Security Standards

Using results of a Bug Finder analysis, you can check your code for the following security standards:

- CWE: See also "CWE Coding Standard and Polyspace Results" on page 5-99.
- CERT C99: See also "CERT C Coding Standard and Polyspace Results" on page 5-130.
- ISO/IEC TS 17961: See also "ISO/IEC TS 17961 Coding Standard and Polyspace Results" on page 5-166.

To adhere to a security standard, follow this workflow.

Step 1: Check Code Against Standard



Bug Finder Analysis

☑ Find defects CWE

- ☑ Defects
  - ☑ Numerical
  - ☑ Static memory
  - ☑ Dynamic memory
  - ☑ Data flow
  - ☑ Resource management
  - ☑ Programming
  - ☑ Concurrency
  - ☑ Security
  - ☑ Tainted data
  - ☑ Good practice

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`.
- 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?*

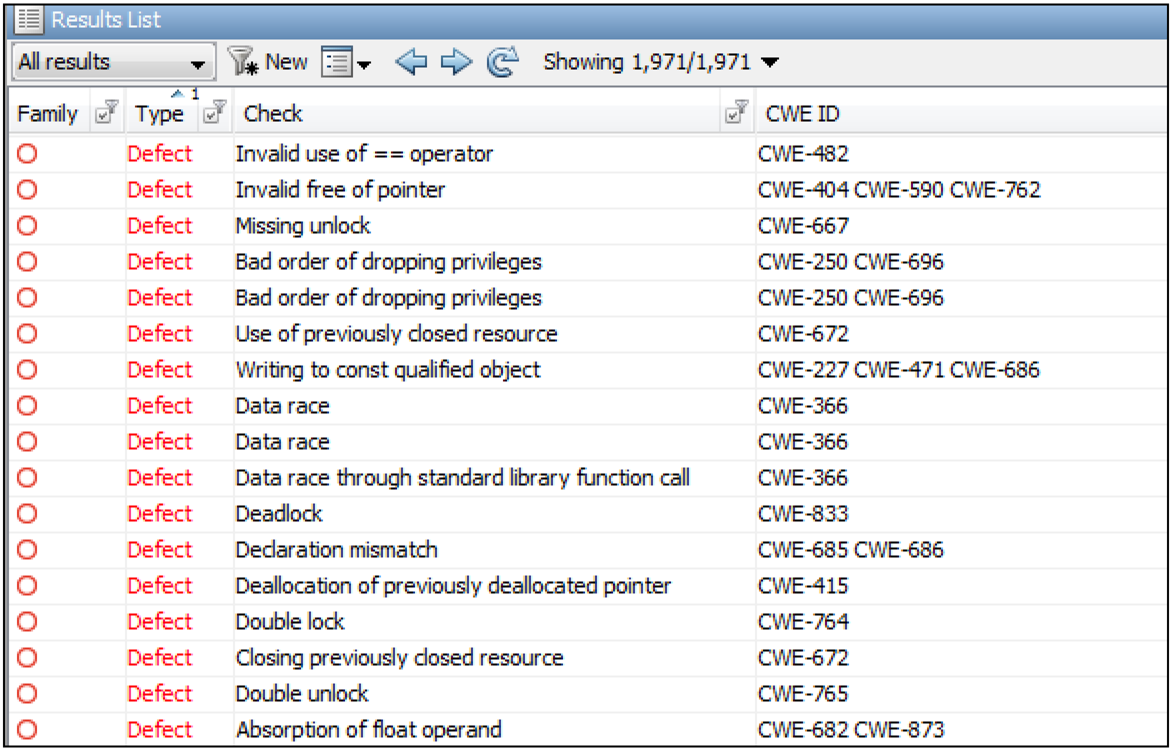
  Choose `custom` for the options to find defects and coding rules. Select defects and coding rules corresponding to specific IDs only.

  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 5-99, CERT C99 on page 5-130 or ISO/IEC TS 17961 on page 5-166.
  - Using configuration templates, see “Create Project Using Configuration Template” on page 1-22.

Step 2: See Results with IDs from Standard



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?*

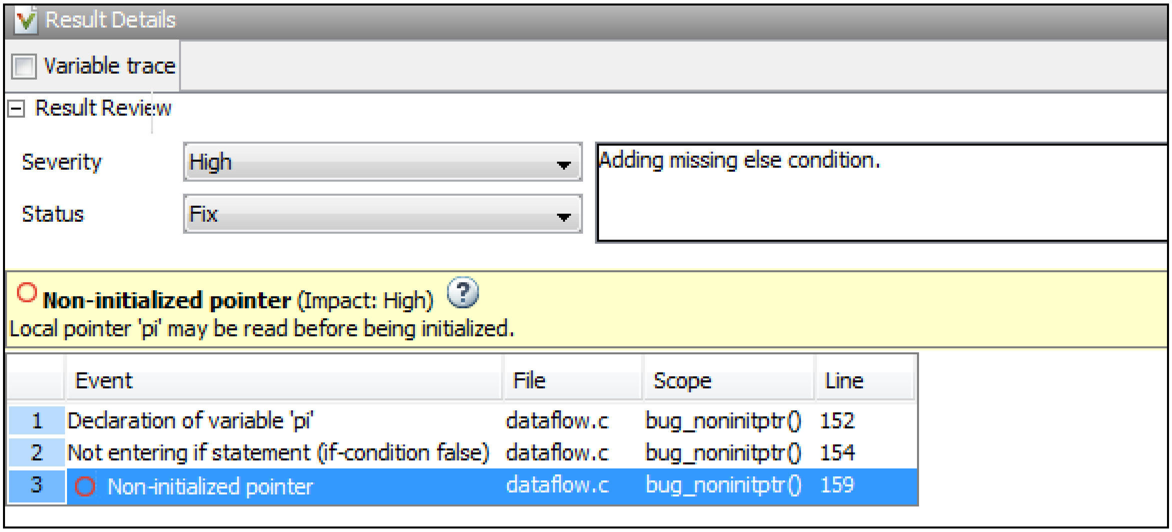
  Narrow your review scope only to results that correspond to a security standard. Instead of `All results` in **Results List**, select `CWE checks`, `CERT checks` or `ISO-17961 checks`.

- *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



☐ Variable trace

⊟ Result Review

Severity: High

Status: Fix

Adding missing else condition.

○ **Non-initialized pointer** (Impact: High)

Local pointer 'pi' may be read before being initialized.

	Event	File	Scope	Line
1	Declaration of variable 'pi'	dataflow.c	bug_noninitptr()	152
2	Not entering if statement (if-condition false)	dataflow.c	bug_noninitptr()	154
3	○ Non-initialized pointer	dataflow.c	bug_noninitptr()	159

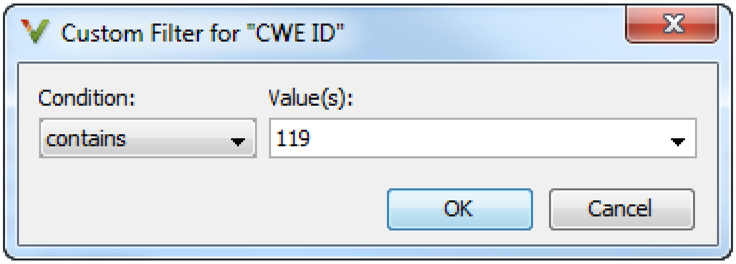
Fix or justify each result. To keep track of your progress, assign the status,`Fix` or `Justified`. For results that you justified, enter comments with your rationale.

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.



Condition: contains

Value(s): 119

- *Can I review only specific IDs*?

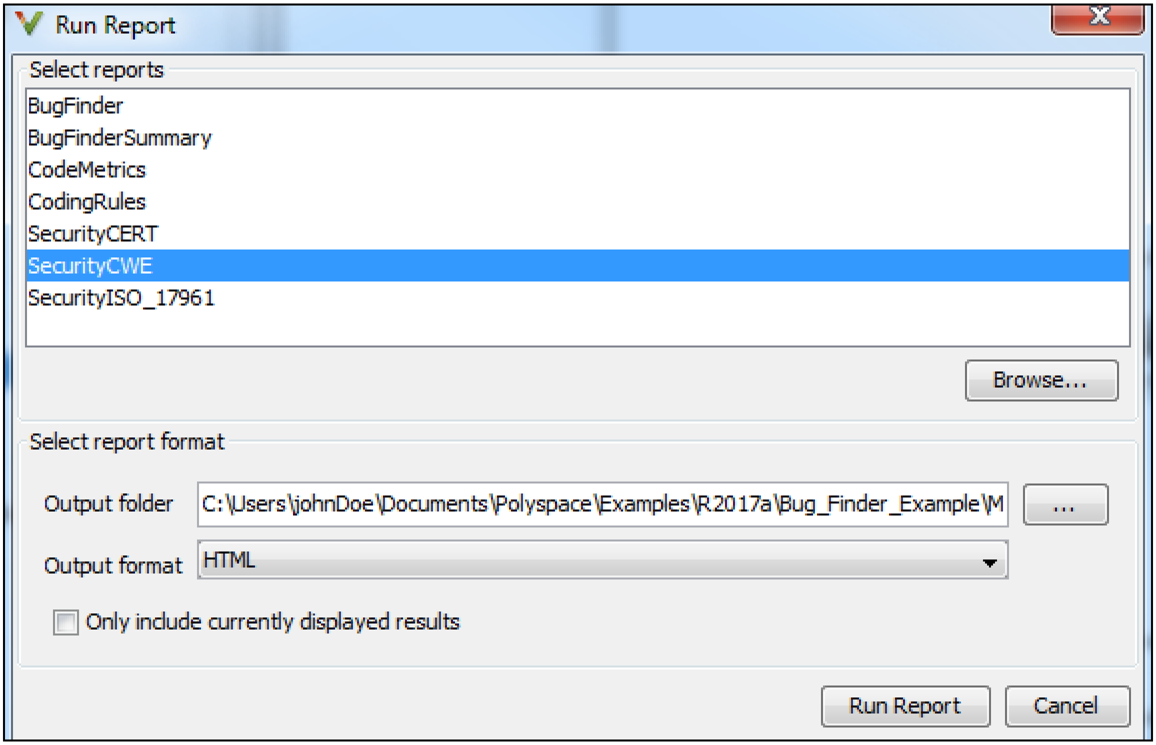
If you ran analysis for all IDs from a standard but want to focus on specific IDs only:

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, `Fix` or `Justified`. For results that you justified, enter comments with your rationale.
2. *Filter out addressed IDs*: Filter out results with `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



BugFinder
BugFinderSummary
CodeMetrics
CodingRules
SecurityCERT
SecurityCWE
SecurityISO_17961

Output folder: C:\Users\johnDoe\Documents\Polyspace\Examples\R2017a\Bug_Finder_Example\M

Output format: HTML

□ Only include currently displayed results

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 5-20.

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 5-20.
- *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 Code for Security Standards" on page 5-92.

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 5-118.

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`
78	Improper neutralization of special elements used in an OS command	`Execution of externally controlled command`
88	Argument injection or modification	`Execution of externally controlled command`

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`

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`
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`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

188	Reliance on data/ memory layout	`Invalid assumptions about memory organization`
		`Memory comparison of padding data`
		`Memory comparison of strings`
		`Pointer access out of bounds`

190	Integer overflow or wraparound	`Integer conversion overflow`
		`Integer overflow`
		`Shift operation overflow`
		`Tainted division operand`
		`Unsigned integer conversion overflow`
		`Unsigned integer overflow`

191	Integer underflow (Wrap or wraparound)	`Integer conversion overflow`
		`Integer overflow`
		`Unsigned integer conversion overflow`
		`Unsigned integer overflow`

194	Unexpected sign extension	`Sign change integer conversion overflow`
		`Tainted sign change conversion`

195	Signed to unsigned conversion error	`Sign change integer conversion overflow`
		`Tainted sign change conversion`

196	Unsigned to signed conversion error	`Sign change integer conversion overflow`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

197	Numeric truncation error	`Integer conversion overflow`
		`Float conversion overflow`
		`Unsigned integer conversion overflow`

226	Sensitive information uncleared before release	`Uncleared sensitive data in stack`

227	Improper fulfillment of API contract	`Invalid use of standard library floating point routine`
		`Invalid use of standard library 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`
311	Missing encryption of sensitive data	`Missing cipher final step`

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`

326	Inadequate encryption strength	`Weak cipher algorithm`

327	Use of a broken or risky cryptographic algorithm	`Unsafe standard encryption function`
		`Weak cipher algorithm`
		`Weak cipher mode`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

329	Not using a random IV with CBC mode	`Missing block cipher initialization vector`
		`Predictable 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`

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`
362	Concurrent execution using shared resource with improper synchronization ('Race Condition')	`Opening previously opened resource`

366	Race condition within a thread	`Data race`
		`Data race including atomic operations`
		`Data race through standard library function call`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
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`

377	Insecure temporary file	`Use of non-secure temporary file`
391	Unchecked error condition	`Errno not checked`
398	Indicator of poor code quality	`Write without a further read`
401	Improper release of memory before removing last reference	`Memory leak`

404	Improper resource shutdown or release	`Invalid deletion of pointer`
		`Invalid free of pointer`
		`Memory leak`

415	Double free	`Deallocation of previously deallocated pointer`
		`Missing reset of a freed pointer`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

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`

427	Uncontrolled search path element	`Execution of a binary from a relative path can be controlled by an external actor`
		`Library loaded from externally controlled path`
		`Load of library from a relative path can be controlled by an external actor`
		`Use of externally controlled environment variable`

456	Missing initialization of a variable	`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`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

467	Use of sizeof() on a pointer type	`Possible misuse of sizeof`
		`Wrong type used in sizeof`

468	Incorrect pointer scaling	`Incorrect pointer scaling`
471	Modification of assumed-immutable data	`Writing to const qualified object`
475	Undefined behavior for input to API	`Copy of overlapping memory`

476	NULL pointer dereference	`Null pointer`
		`Tainted NULL or non-null-terminated string`

477	Use of obsolete functions	`Use of obsolete standard function`
478	Missing default case in switch statement	`Missing case for switch condition`
481	Assigning instead of comparing	`Invalid use of = (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`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

547	Use of hard-coded, security-relevant constants	`Hard coded buffer size`
		`Hard coded loop boundary`

558	Use of getlogin() in multithreaded application	`Unsafe standard function`
560	Use of umask() with chmod-style argument	`Umask used with chmod-style arguments`

561	Dead code	`Dead code`
		`Static uncalled function`
		`Unreachable code`

562	Return of stack variable address	`Pointer or reference to stack variable leaving scope`

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`

587	Assignment of a fixed address to a pointer	`Unsafe conversion between pointer and integer`
		`Function pointer assigned with absolute address`

590	Free of memory not on the heap	`Invalid free of pointer`
606	Unchecked input for loop condition	`Loop bounded with tainted value`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
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 5-118.	
659	See “Mapping Between CWE-658 or 659 and Polyspace Results” on page 5-118.	
663	Use of a non-reentrant function in a concurrent context	Unsafe standard encryption function Unsafe standard function
665	Improper initialization	Call to memset with unintended value Improper array initialization Overlapping assignment Use of memset with size argument zero
666	Operation on resource in wrong phase of lifetime	Incorrect order of network connection operations
667	Improper locking	Missing unlock Destruction of locked mutex
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

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

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`
		`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`

691	Insufficient control flow management	`Use of setjmp/longjmp`
696	Incorrect behavior order	`Bad order of dropping privileges`

703	Improper check or handling of exceptional conditions	`Errno not reset`
		`Misuse of errno`

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`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
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`
764	Multiple locks of a critical resource	`Double lock`
765	Multiple unlocks of a critical resource	`Double unlock`
767	Access to critical private variable via public method	`Return of non const handle to encapsulated data member`
770	Allocation of resources without limits or throttling	`Tainted size of variable length array`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
772	Missing release of resource after effective lifetime	`Resource leak`
783	Operator precedence logic error	`Possibly unintended evaluation of expression because of operator precedence rules`
785	Use of path manipulation function without maximum-sized buffer	`Use of path manipulation function without maximum sized buffer checking`
786	Access of memory location before start of buffer	`Destination buffer underflow in string manipulation`

787	Out-of-bounds write	`Destination buffer overflow in string manipulation`
		`Destination buffer underflow in string manipulation`

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`
822	Untrusted pointer dereference	`Tainted NULL or non-null-terminated string`

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`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
826	Premature release of resource during expected lifetime	`Destruction of locked mutex`
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`

873	CERT C++ Secure Coding Section 05 - Floating point arithmetic (FLP)	`Absorption of float operand`
		`Invalid use of floating point operation`
		`Invalid use of standard library floating point routine`
		`Float overflow`

908	Use of uninitialized resource	`Member not initialized in constructor`
		`Non-initialized pointer`
		`Non-initialized variable`

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`

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`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

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`

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`
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`
		`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`

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("/")`
244	Improper clearing of heap memory before release	`Sensitive heap memory not cleared before release`
251	Often misused: string management	`Destination buffer overflow in string manipulation`
362	Concurrent execution using shared resource with improper synchronization ('Race Condition')	`Opening previously opened resource`

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`

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

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`
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`

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`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

685	Function call with incorrect number of arguments	`Declaration mismatch`
		`Format string specifiers and arguments mismatch`
		`Standard function call with incorrect arguments`

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`
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`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
843	Access of resource using incompatible type ('Type confusion')	`Unreliable cast of pointer`

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`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
		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
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
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

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

188	Reliance on data/ memory layout	`Invalid assumptions about memory organization`
		`Memory comparison of padding data`
		`Memory comparison of strings`
		`Pointer access out of bounds`

191	Integer underflow (Wrap or wraparound)	`Integer conversion overflow`
		`Integer overflow`
		`Unsigned integer conversion overflow`
		`Unsigned integer overflow`

194	Unexpected sign extension	`Sign change integer conversion overflow`
		`Tainted sign change conversion`

195	Signed to unsigned conversion error	`Sign change integer conversion overflow`
		`Tainted sign change conversion`

196	Unsigned to signed conversion error	`Sign change integer conversion overflow`

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`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
251	Often misused: string management	`Destination buffer overflow in string manipulation`
362	Concurrent execution using shared resource with improper synchronization ('Race Condition')	`Opening previously opened resource`

366	Race condition within a thread	`Data race`
		`Data race including atomic operations`
		`Data race through standard library function call`

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`

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker

467	Use of sizeof() on a pointer type	`Possible misuse of sizeof`
		`Wrong type used in sizeof`

468	Incorrect pointer scaling	`Incorrect pointer scaling`

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`
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`

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`

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`
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`

More About

- “CWE Coding Standard and Polyspace Results” on page 5-99
- “Check Code for Security Standards” on page 5-92

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® Bug Finder™ results can be mapped to CERT C rules and recommendations. Using Bug Finder results (defects and coding rule violations), you can address 71 CERT C rules and 94 CERT C recommendations[6]

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 5-164.

Find CERT C Guideline Violations from Polyspace Results

Use the following workflow if you want to focus your Bug Finder analysis on the CERT C 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 CERT C standard. The report shows the CERT C rules or recommendations corresponding to each result.

For the detailed workflow, see “Check Code for Security Standards” on page 5-92.

[6] For comparison, the print version of the CERT C rules (second edition) lists 98 rules. The CERT C website, under continuous development, lists 118 rules and 188 recommendations as of 8th January, 2016.

Mapping Between CERT C Rules and Polyspace Results

The following tables list the CERT C rules that Polyspace Bug Finder addresses and the corresponding defects or MISRA C: 2012 rule.

Rule 01. Preprocessor (PRE)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
PRE31-C	Declare objects with appropriate storage durations		`MISRA C:2012 Rule 13.2`

Rule 02. Declarations and Initialization (DCL)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
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`

DCL40-C	Do not create incompatible	`Declaration mismatch`	`MISRA C:2012 Rule 5.1`

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
	declarations of the same function or object		`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
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`	

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`	

EXP36-C	Do not cast pointers	`Unreliable cast of pointer`	`MISRA C:2012 Rule 11.1`

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule

	into more strictly aligned pointer types	`Wrong allocated object size for cast`	`MISRA C:2012 Rule 11.2`
			`MISRA C:2012 Rule 11.3`
			`MISRA C:2012 Rule 11.5`
			`MISRA C:2012 Rule 11.7`

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`	

EXP40-C	Do not modify constant objects	`Writing to const qualified object`	
EXP43-C	Avoid undefined behavior when using restrict-qualified pointers	`Copy of overlapping memory`	`MISRA C:2012 Rule 8.14`

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
EXP45-C	Do not perform assignments in selection statements	`Invalid use of assignment operator`	

Rule 04. Integers (INT)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
INT30-C	Ensure that unsigned integer operations do not wrap	`Unsigned integer 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`

INT32-C	Ensure that operations on signed integers do not result in overflow	`Integer overflow`
		`Tainted division operand`
		`Tainted modulo operand`	

INT33-C	Ensure that division and remainder operations do	`Integer division by zero`
		`Tainted division operand`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
	not result in divide-by-zero errors	Tainted modulo operand	
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	
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
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	
FLP34-C	Ensure that floating-point conversions are within range of the new type	Float conversion overflow Integer conversion overflow	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
		Unsigned integer conversion overflow	

Rule 06. Arrays (ARR)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
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
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	
ARR37-C	Do not add or subtract an integer to a pointer to a non-array object	Invalid assumptions about memory organization	
ARR38-C	Guarantee that library functions do not form invalid pointers	Array access out of bounds Buffer overflow from incorrect string format specifier	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule

		`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`	

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
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`	

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`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
STR34-C	Cast characters to unsigned char before converting to larger integer sizes	Misuse of sign-extended character value	
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
MEM30-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 Directive 4.13 MISRA C:2012 Rule 18.6 MISRA C:2012 Rule 22.1
MEM31-C	Free dynamically allocated memory when no longer needed	Memory leak	
MEM34-C	Only free memory	Invalid free of pointer	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
	allocated dynamically		

MEM35-C	Allocate sufficient memory for an object	`Memory allocation with tainted size`
		`Pointer access out of bounds`
		`Wrong type used in sizeof`	

Rule 09. Input Output (FIO)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
FIO24–C	Do not open a file that is already open	`Opening previously opened resource`	
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`	
FIO42-C	Close files when they are no longer needed	`Resource leak`	
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

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
ENV30-C	Do not modify the object referenced by the return value of certain functions	`Modification of internal buffer returned from nonreentrant standard function`	
ENV32-C	All exit handlers must return normally	`Abnormal termination of exit handler`	

ENV33-C	Do not call system()	`Execution of externally controlled command`
		`Command executed from externally controlled path`	

ENC34-C	Do not store pointers returned by certain functions	`Misuse of return value from nonreentrant standard function`	

Rule 12: Error Handling (ERR)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule

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`	

ERR33-C	Detect and handle standard library errors	`Errno not checked`
		`Returned value of a sensitive function not checked`
		`Unprotected dynamic memory allocation`	

Rule 14. Concurrency (CON)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
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`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
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
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`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
MSC37-C	Ensure that control never reaches the end of a non-void function	`Missing return statement`	

MSC39-C	Do not call va_arg() on a va_list that has an indeterminate value	`Invalid va_list argument`
		`Non-initialized variable`	

Rule 50. POSIX (POS)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
POS30-C	Use the readlink() function properly	`Misuse of readlink()`	
POS33-C	Do not use vfork()	`Use of obsolete standard function`	
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`	
POS37-C	Ensure that privilege	`Privilege drop not verified`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
	relinquishment is successful		
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`	

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
PRE00-C	Prefer inline or static functions to function-like macros		`MISRA C:2012 Directive 4.9`

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
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 Directive 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
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 Directive 4.5
DCL06-C	Use meaningful symbolic constants	Hard coded buffer size Hard coded loop boundary	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
	to represent literal values		
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
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
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
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
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`

EXP09-C	Use sizeof to determine the size of a type or variable	`Hard-coded object size used to manipulate memory`	
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	`Possibly unintended evaluation of expression because of operator precedence rules`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
	if they were nonassociative		
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
INT00-C	Understand the data model used by your implementation	Format string specifiers and arguments mismatch Integer overflow	
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
INT04-C	Enforce limits on integer values originating from tainted sources	Loop bounded with tainted value Memory allocation with tainted size Tainted size of variable length array	
INT04-C			
INT07-C	Use only explicitly signed or	Use of plain char type for numerical value	MISRA C:2012 Rule 10.1 MISRA C:2012 Rule 10.3

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
	unsigned char type for numeric values		`MISRA C:2012 Rule 10.4`
INT08-C	Verify that all integer values are in range	`Integer 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`	
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		`MISRA C:2012 Rule 10.1`

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
	about representation of signed integers		

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
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	`Invalid use of floating point operation`	

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`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
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

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`	

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

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`	

STR03-C	Do not inadvertently truncate a string	`Buffer overflow from incorrect string format specifier`	

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`	

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	`Use of dangerous standard function`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
	for string manipulation	`Destination buffer overflow in string manipulation`	

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

MEM00-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`	

MEM01-C	Store a new value in pointers immediately after free()	`Missing reset of a freed pointer`	
MEM02-C	Immediately cast the result of a memory	`Wrong allocated object size for cast`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
	allocation function call into a pointer to the allocated type	Wrong type used in sizeof	
MEM03-C	Clear sensitive information stored in reusable resources	Sensitive heap memory not cleared before release Uncleared sensitive data in stack	
MEM04-C	Beware of zero-length allocations	Tainted sign change conversion Tainted size of variable length array Variable length array with nonpositive size	
MEM05-C	Avoid large stack allocations	Tainted size of variable length array Variable length array with nonpositive size	MISRA C:2012 Rule 17.2
MEM06-C	Ensure that sensitive data is not written out to disk	Sensitive data printed out	
MEM07-C	Ensure that arguments to calloc(), when multiplied, do not wrap	Memory allocation with tainted size	
MEM10-C	Define and use a pointer	Memory allocation with tainted size	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule

	validation function	`Unprotected dynamic memory allocation`
		`Use of tainted pointer`	

MEM11-C	Do not assume infinite heap space	`Memory leak`
		`Memory allocation with tainted size`
		`Tainted sign change conversion`
		`Unprotected dynamic memory allocation`	

MEM12-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
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`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
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`	

FIO11-C	Take care when specifying the mode parameter of fopen()	`Bad file access mode or status`	
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

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
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
API00-C	Functions should validate their parameters	Invalid use of standard library memory routine Invalid use of standard library routine Invalid use of standard library string routine Standard function call with incorrect arguments "Tainted Data Defects"	
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	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
		`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
CON01-C	Acquire and release synchronizatio primitives in the same module, at the same level of abstraction	`Missing lock`	
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

MSC01-C	Strive for logical completeness	`Dead code`
		`Missing case for switch condition`
		`Unreachable code`	

MSC04-C	Use comments consistently		`MISRA C:2012 Rule 1.2`
			`MISRA C:2012 Rule 3.1`

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
	and in a readable fashion		
MSC07-C	Detect and remove dead code	Dead code Missing case for switch condition Unreachable code	MISRA C:2012 Rule 2.1
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
MSC13-C	Detect and remove unused values	Unused parameter Write without a further read	
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	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule

		`Invalid use of standard library string routine`
		`Non-initialized pointer`
		`Non-initialized variable`
		`Null pointer`
		`Overlapping assignment`
		`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`	
MSC18-C	Be careful while handling sensitive	`Sensitive heap memory not cleared before release`	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
	data, such as passwords, in program code	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	
MSC22-C	Use the setjmp(), longjmp() facility securely	Use of setjmp/longjmp	
MSC24-C	Do not use deprecated or obsolescent functions	Use of obsolete standard function	

Rec. 05. POSIX (POS)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
POS05-C	Limit access to files by creating a jail	File manipulation after chroot() without chdir("/")	

Rec. 05. Microsoft Windows (WIN)

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
WIN00-C	Be specific when dynamically loading libraries	Library loaded from externally controlled path Load of library from a relative path can	

CERT C	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
		`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 Code for Security Standards" on page 5-92.

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`

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
			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		MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 21.3
boolasgn	No assignment in conditional expressions	Invalid use of = (assignment) operator	MISRA C:2012 Rule 13.4
asyncsig	Calling functions in the C Standard Library other than abort, _Exit, and signal from within a signal handler		MISRA C:2012 Rule 21.5

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule

`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 17.3`

`sigcall`	Calling `signal` from interruptible signal handlers		`MISRA C:2012 Rule 21.5`

`syscall`	Calling `system`	`Command executed from externally controlled path`
		`Execution of externally controlled command`
			`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`

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
`alignconv`	Converting pointer values to more strictly aligned pointer types	`Unreliable cast of pointer`	`MISRA C:2012 Rule 11.3`
`filecpy`	Copying 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 8.3`
			`MISRA C:2012 Rule 8.4`

`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 Directive 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`	`MISRA C:2012 Rule 18.6`

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
`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`

`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 Directive 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 Directive 4.1 MISRA C:2012 Directive 4.11 MISRA C:2012 Rule 1.3
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`

`usrfmt`	Including tainted or out-of-domain input in a format string	`Tainted string format`	`MISRA C:2012 Directive 4.1`
			`MISRA C:2012 Directive 4.11`
			`MISRA C:2012 Rule 1.3`
			`MISRA C:2012 Rule 21.6`

`inverrno`	Incorrectly setting and using `errno`	`Errno not checked`
		`Errno not reset`
		`Misuse of errno`
			`MISRA C:2012 Directive 4.1`
			`MISRA C:2012 Directive 4.7`

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
diverr	Integer division errors	Integer division by zero Tainted division operand Tainted modulo operand	MISRA C:2012 Directive 4.1 MISRA C:2012 Rule 1.3
ioileave	Interleaving stream inputs and outputs without a flush or positioning call		MISRA C:2012 Directive 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
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
intoflow	Overflowing signed integers	Integer overflow Tainted modulo operand	MISRA C:2012 Directive 4.1 MISRA C:2012 Rule 1.3 MISRA C:2012 Rule 10.3 MISRA C:2012 Rule 10.4

ISO/IEC TS 17961 Rule ID	Description	Polyspace Bug Finder Defect	MISRA C: 2012 Rule
nonnullcs	Passing 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 Directive 4.1 MISRA C:2012 Directive 4.11 MISRA C:2012 Rule 1.3
chrsgnext	Passing arguments to character-handling functions that are not representable as unsigned char		MISRA C:2012 Directive 4.1 MISRA C:2012 Directive 4.11 MISRA C:2012 Rule 1.3
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		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 Directive 4.1 MISRA C:2012 Directive 4.11
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		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 Directive 4.1 MISRA C:2012 Directive 4.11 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		MISRA C:2012 Directive 4.1 MISRA C:2012 Directive 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
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 Directive 4.1 MISRA C:2012 Directive 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 Directive 4.1 MISRA C:2012 Directive 4.11 MISRA C:2012 Rule 1.3

6

Command-Line Analysis

Create Project Automatically at Command Line

If you use build automation scripts to build your source code, you can automatically setup a Polyspace project from your scripts. The automatic project setup runs your automation scripts to determine:

- Source files.
- Includes.
- Target & compiler options. For more information on these options, see “Target & Compiler”.

Use the `polyspace-configure` command to trace your build automation scripts. You can use the trace information to:

- Create a Polyspace project. You can then open the project in the user interface.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following command to create a Polyspace project `myProject.psprj` from your makefile:

  ```
  polyspace-configure -prog myProject make targetName buildOptions
  ```

  For the list of options allowed with the GNU `make`, see make options.
- Create an options file. You can then use the options file to run analysis on your source code from the command-line.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following commands to create an options file `myOptions` from your makefile:

  ```
  polyspace-configure -no-project -output-options-file myOptions ...
              make targetName buildOptions
  ```

  Use the options file to run analysis:

  ```
  polyspace-bug-finder-nodesktop -options-file myOptions
  ```

You can also use advanced options to modify the default behavior of `polyspace-configure`. For more information, see the `-options value` argument for `polyspaceConfigure`.

Note:

- In the Polyspace interface, it is possible that the created project will not show implicit defines or includes. The configuration tool does include them. However, they are not visible.

- By default, Polyspace assigns the latest version of the compiler to your project. If you have compilation errors in your project, check the setting for Compiler (-compiler). If it does not apply to you, change it to a more appropriate one.
  For instance, if the compiler setting is `visual12` but you are using Microsoft Visual C ++ 2010, change the setting to `visual10`.
- If your build process requires user interaction, you cannot run the build command from the Polyspace user interface and do an automatic project setup.

More About

- "Requirements for Project Creation from Build Systems" on page 1-9
- "Compiler Not Supported for Project Creation from Build Systems" on page 14-48
- "Slow Build Process When Polyspace Traces the Build" on page 14-56
- "Check if Polyspace Supports Build Scripts" on page 14-57

Run Local Analysis from DOS or UNIX Command Line

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

Note: To run Bug Finder from the MATLAB Command Window, use the command `polyspaceBugFinder` *`[options]`*

In this section...
"Specify Sources and Analysis Options Directly" on page 6-4
"Specify Sources and Analysis Options in Text File" on page 6-4
"Create Options File from Build System" on page 6-5

Specify Sources and Analysis Options Directly

At the Windows, Linux or Mac OS X command-line, append sources and analysis options to the `polyspace-bug-finder-nodesktop` command.

For instance:

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

  ```
  polyspace-bug-finder-nodesktop -sources "file.c" -lang c -target m68k
  ```
- To check for violation of MISRA C rules, use the `-misra2` option. For instance, to check for only the required MISRA C rules on your source file `file.c`, use the command:

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

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

You can also enter the following at the command line:

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

Specify Sources and Analysis Options in Text File

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

```
#These are the options for MyBugFinderProject
-lang c
-prog MyBugFinderProject
-author jsmith
-sources "mymain.c,funAlgebra.c,funGeometry.c"
-OS-target no-predefined-OS
-target x86_64
-compiler none
-dos
-misra2 required-rules
-do-not-generate-results-for all-headers
-checkers default
-disable-checkers concurrency
-results-dir C:\Polyspace\MyBugFinderProject
```

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

```
polyspace-bug-finder-nodesktop -options-file listofoptions.txt
```

Create Options File from Build System

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

```
polyspace-configure -c -no-project -output-options-file \
        myOptions make -B myCode
```

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

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

3 Open the results in the Bug Finder interface.

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

Run Remote Analysis at the Command Line

Before you run a remote analysis, you must set up a server for this purpose. For more information, see “Set Up Server for Metrics and Remote Analysis”.

In this section...

Run Remote Analysis

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.
- *options* are the analysis options. These options are the same as that of a local analysis. For more information, see “Run Local Analysis from DOS or UNIX Command Line” on page 6-4.

After compilation, the software submits the analysis job to the cluster and provides you a job ID. 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 6-7.

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 6-4.

To specify your sources, in the options file, instead of `-sources`, use -sources-list-file. This option is available only for remote analysis and allows you to specify your sources in a separate text file.

2. Create a file `launcher.bat` in a text editor like Notepad.
3. Enter the following commands in the file.

```
echo off
set POLYSPACE_PATH=C:\Program Files\MATLAB\R2015a\polyspace\bin
set RESULTS_PATH=C:\Results
set OPTIONS_FILE=C:\Options\listofoptions.txt
"%POLYSPACE_PATH%\polyspace-bug-finder-nodesktop.exe" -batch -scheduler localhost
                     -results-dir "%RESULTS_PATH%" -options-file "%OPTIONS_FILE%"
pause
```

4. Replace the definitions of the following variables in the file:
   - `POLYSPACE_PATH`: Enter the actual location of the `.exe` file.
   - `RESULTS_PATH`: Enter the path to a folder. The files generated during compilation are saved in the folder.
   - `OPTIONS_FILE`: Enter the path to the file `listofoptions.txt`.

   Replace `localhost` with the name of the computer that hosts the head node of your MATLAB Distributed Computing Server cluster.
5. Double-click `launcher.bat` to run the 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`*. 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 “Open Results” (Polyspace Code Prover).
`getlog`	`-job` *`ID`*	Open log for job with specified ID.
`remove`	`-job` *`ID`*	Remove job with specified ID.

Enable Defect Checkers from Command Line

To enable detection of a specific defect, you must specify its command-line name as argument of the `-checkers` option. You can also specify groups of defects. For information on the command-line names of the groups, see `Find defects (-checkers)`.

The following table lists the command-line names for individual defects.

Defect	Command-line name
`*this not returned in copy assignment operator`	`return_not_ref_to_this`
`Absorption of float operand`	`float_absorption`
`Arithmetic operation with NULL pointer`	`null_ptr_arith`
`Array access out of bounds`	`out_bound_array`
`Assertion`	`assert`
`Bad file access mode or status`	`bad_file_access_mode_status`
`Base class assignment operator not called`	`missing_base_assign_op_call`
`Base class destructor not virtual`	`dtor_not_virtual`
`Buffer overflow from incorrect string format specifier`	`str_format_buffer_overflow`
`Call to memset with unintended value`	`memset_invalid_value`
`Closing a previously closed resource`	`double_resource_close`
`Code deactivated by constant false condition`	`deactivated_code`

Defect	Command-line name
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
Deadlock	deadlock
Dead code	dead_code
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
Double lock	double_lock
Double unlock	double_unlock
Exception caught by value	excp_caught_by_value
Exception handler hidden by previous handler	excp_handler_hidden

Defect	Command-line name
Float overflow	float_ovfl
Float conversion overflow	float_conv_ovfl
Float division by zero	float_zero_div
Format string specifiers and arguments mismatch	string_format
Hard-coded buffer size	hard_coded_buffer_size
Hard-coded loop boundary	hard_coded_loop_boundary
Improper array initialization	improper_array_init
Incompatible types prevent overriding	virtual_func_hiding
Incorrect pointer scaling	bad_ptr_scaling
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 free of pointer	bad_free
Invalid use of == (equality) operator	bad_equal_equal_use
Invalid use of = (assignment) operator	bad_equal_use
Invalid use of floating point operation	bad_float_op

Defect	Command-line name
Invalid use of standard library routine	other_std_lib
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
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
Line with more than one statement	more_than_one_statement
Member not initialized in constructor	non_init_member
Memory leak	mem_leak
Missing case for switch condition	missing_switch_case
Missing explicit keyword	missing_explicit_keyword
Missing lock	bad_unlock
Missing null in string array	missing_null_char
Missing unlock	bad_lock
Missing return statement	missing_return
Missing virtual inheritance	missing_virtual_inheritance

Defect	Command-line name
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
Null pointer	null_ptr
Object slicing	object_slicing
Overlapping assignment	overlapping_assign
Partial override of overloaded virtual functions	partial_override
Pointer access out of bounds	out_bound_ptr
Pointer or reference to stack variable leaving scope	local_addr_escape
Pointer to non-initialized value converted to const pointer	non_init_ptr_conv
Partially accessed array	partially_accessed_array
Possible misuse of sizeof	sizeof_misuse
Possibly unintended evaluation of expression because of operator precedence rules	operator_precedence
Resource leak	resource_leak
Return of non const handle to encapsulated data member	breaking_data_encapsulation

Defect	Command-line name
Self assignment not tested in operator	missing_self_assign_test
Qualifier removed in conversion	qualifier_mismatch
Shift of a negative value	shift_neg
Shift operation overflow	shift_ovfl
Sign change integer conversion overflow	sign_change
Standard function call with incorrect arguments	std_func_arg_mismatch
Static uncalled function	uncalled_func
Unprotected dynamic memory allocation	unprotected_memory_allocation
Unreachable code	unreachable
Unreliable cast of function pointer	func_cast
Unreliable cast of pointer	ptr_cast
Unsigned integer conversion overflow	uint_conv_ovfl
Unsigned integer overflow	uint_ovfl
Unused parameter	unused_parameter
Useless if	useless_if
Use of memset with size argument zero	memset_invalid_size
Use of path manipulation function without maximum sized buffer checking	path_buffer_overflow

Defect	Command-line name
Use of previously closed resource	closed_resource_use
Use of previously freed pointer	freed_ptr
Use of setjmp/longjmp	setjmp_longjmp_use
Variable length array with nonpositive size	non_positive_vla_size
Variable shadowing	var_shadowing
Writing to const qualified object	constant_object_write
Writing to read-only resource	read_only_resource_write
Write without a further read	useless_write
Wrong allocated object size for cast	object_size_mismatch
Wrong type used in sizeof	ptr_sizeof_mismatch
Incorrect order of network connection operations	bad_network_connect_order
Umask used with chmod-style arguments	bad_umask
File manipulation after chroot without chdir	chroot_misuse
Vulnerable permission assignments	dangerous_permissions
Use of dangerous standard function	dangerous_std_func
Mismatch between data length and size	data_length_mismatch

Defect	Command-line name
Function pointer assigned with absolute address	func_ptr_absolute_addr
Use of non-secure temporary file	non_secure_temp_file
Use of obsolete standard function	obsolete_std_func
Vulnerable path manipulation	path_traversal
Deterministic random output from constant seed	rand_seed_constant
Predictable random output from predictable seed	rand_seed_predictable
Execution of a binary from a relative path can be controlled by an external actor	relative_path_cmd
Load of library from a relative path can be controlled by an external actor	relative_path_lib
Sensitive data printed out	sensitive_data_print
Sensitive heap memory not cleared before release	sensitive_heap_not_cleared
Uncleared sensitive data in stack	sensitive_stack_not_cleared
Array access with tainted index	tainted_array_index

Defect	Command-line name
Use of externally controlled environment variable	tainted_env_variable
Execution of externally controlled command	tainted_external_cmd
Host change using externally controlled elements	tainted_hostid
Tainted division operand	tainted_int_division
Tainted modulo operand	tainted_int_mod
Loop bounded with tainted value	tainted_loop_boundary
Memory allocation with tainted size	tainted_memory_alloc_size
Command executed from externally controlled path	tainted_path_cmd
Library loaded from externally controlled path	tainted_path_lib
Use of tainted pointer	tainted_ptr
Pointer dereference with tainted offset	tainted_ptr_offset
Tainted sign change conversion	tainted_sign_change
Tainted NULL or non-null-terminated string	tainted_string
Tainted string format	tainted_string_format
Tainted size of variable length array	tainted_vla_size

Defect	Command-line name
File access between time of check and use (TOCTOU)	toctou
Unsafe standard encryption function	unsafe_std_crypt
Unsafe standard function	unsafe_std_func
Vulnerable pseudo-random number generator	vulnerable_prng
Unsafe conversion between pointer and integer	bad_int_ptr_cast
Use of plain char type for numerical value	bad_plain_char_use
Bad order of dropping privileges	bad_privilege_drop_order
Bitwise and arithmetic operation on the same data	bitwise_arith_mix
Bitwise operation on negative value	bitwise_neg
Opening previously opened resource	double_resource_open
Abnormal termination of exit handler	exit_abnormal_handler
Hard-coded object size used to manipulate memory	hard_coded_mem_size
Missing reset of a freed pointer	missing_freed_ptr_reset
Missing break of switch case	missing_switch_break

Defect	Command-line name
Privilege drop not verified	missing_privilege_drop_check
Returned value of a sensitive function not checked	return_not_checked
Typedef mismatch	typedef_mismatch
Unsafe conversion from string to numerical value	unsafe_str_to_numeric
Predictable block cipher initialization vector	crypto_cipher_predictable_iv
Constant block cipher initialization vector	crypto_cipher_constant_iv
Missing block cipher initialization vector	crypto_cipher_no_iv
Predictable cipher key	crypto_cipher_predictable_key
Constant cipher key	crypto_cipher_constant_key
Missing cipher key	crypto_cipher_predictable_iv
Inconsistent cipher operations	crypto_cipher_bad_function
Missing cipher data to process	crypto_cipher_no_data
Missing cipher algorithm	crypto_cipher_no_algorithm
Weak cipher mode	crypto_cipher_weak_cipher
Weak cipher algorithm	crypto_cipher_weak_cipher

Create Command-Line Script from Project File

In this section...
"Generate Scripting Files" on page 6-21
"Run an Analysis" on page 6-22

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. For this example, you use the example shipped with Polyspace.

Generate Scripting Files

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/R2017a/Bug_Finder_Example
```

4 Run the script generation command . (*`matlabroot`* is your installed program folder, for example `C:\Program Files\MATLAB\R2017a`.)

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

   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 6-21, 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 Local Analysis from DOS or UNIX Command Line” on page 6-4

External Websites

- How do I use Polyspace with Jenkins?

Create Project Automatically from MATLAB Command Line

If you use build automation scripts to build your source code, you can automatically setup a Polyspace project from your scripts. The automatic project setup runs your automation scripts to determine:

- Source files.
- Includes.
- Target & compiler options. For more information on these options, see "Target & Compiler".

Use the `polyspaceConfigure` command to trace your build automation scripts. You can use the trace information to:

- Create a Polyspace project. You can then open the project in the user interface.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following command to create a Polyspace project `myProject.psprj` from your makefile:

  ```
  polyspaceConfigure  -prog myProject ...
                make targetName buildOptions
  ```
- Create an options file. You can then use the options file to run analysis on your source code from the command-line.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following commands to create an options file `myOptions` from your makefile:

  ```
  polyspaceConfigure  -no-project -output-options-file myOptions ...
              make targetName buildOptions
  ```
  Use the options file to run analysis:

  ```
  polyspaceBugFinder -options-file myOptions
  ```

You can also use advanced options to modify the default behavior of `polyspaceConfigure`. For more information, see `polyspaceConfigure`.

Note:

- In the Polyspace interface, it is possible that the created project will not show implicit defines or includes. The configuration tool does include them. However, they are not visible.

- By default, Polyspace assigns the latest version of the compiler to your project. If you have compilation errors in your project, check the setting for Compiler (-compiler). If it does not apply to you, change it to a more appropriate one.
  For instance, if the compiler setting is `visual12` but you are using Microsoft Visual C ++ 2010, change the setting to `visual10`.
- If your build process requires user interaction, you cannot run the build command from the Polyspace user interface and do an automatic project setup.

More About

- "Requirements for Project Creation from Build Systems" on page 1-9
- "Compiler Not Supported for Project Creation from Build Systems" on page 14-48
- "Slow Build Process When Polyspace Traces the Build" on page 14-56

Run Polyspace in MATLAB

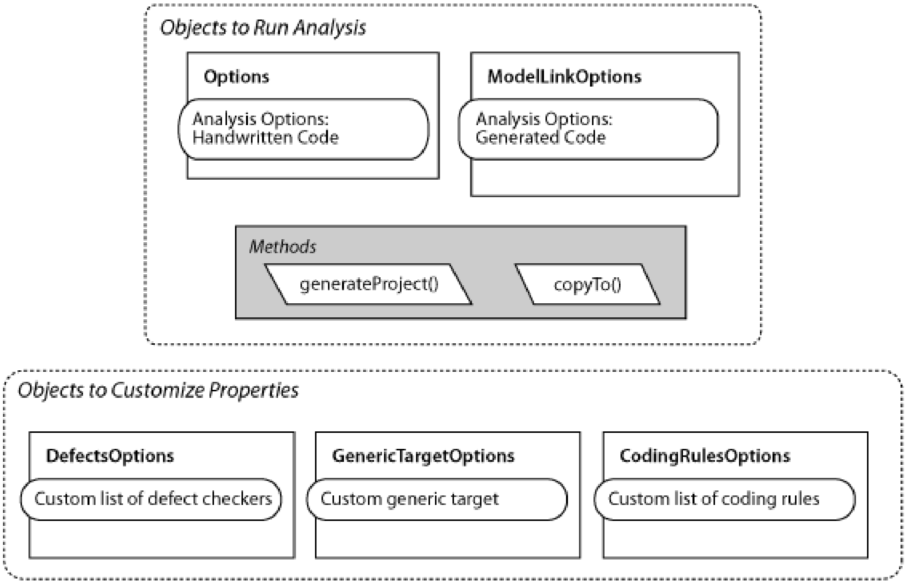
There are several different ways to analyze C/C++ code in MATLAB. Choose the method that best fits your needs:

- “MATLAB Objects” on page 6-25 — Recommended for pure MATLAB scripting.
- “Project Files” on page 6-27 — Recommended for running projects created in the Polyspace interface. You can continue to view and edit the project from the Polyspace interface.
- “UNIX/DOS Command-Line Analysis Options and Values” on page 6-27 — Recommended if adapting a UNIX/DOS script directly. Uses the syntax from UNIX and DOS scripts.

MATLAB Objects

Using a combination of objects, methods, and functions, this method is best for scripting in the MATLAB language only.

For Bug Finder, there are two main classes, with two methods, and three additional helper classes.


Objects to Run Analysis
Options
Analysis Options: Handwritten Code
ModelLinkOptions
Analysis Options: Generated Code
Methods
generateProject()
copyTo()
Objects to Customize Properties
DefectsOptions
Custom list of defect checkers
GenericTargetOptions
Custom generic target
CodingRulesOptions
Custom list of coding rules

To run analysis in MATLAB, follow this workflow:

1 Create a Polyspace options object.

   You can either create an object with the class that fits best:

   - polyspace.Options — for handwritten code.
   - polyspace.ModelLinkOptions — for model-generated code.

   You can create an object in other ways:

   - Copy an existing object using polyspace.options.copyTo.
   - Create an options object from an existing Polyspace project using `polyspace.loadProject`:

     ```
     optsObject = polyspace.loadProject('C:\projects\myProject.psprj')
     ```

2 Customize the properties of your options object.

   It can take some trial-and-error to find the optimal set of analysis options. At minimum, you must add source files to the options object and any related include file. To create further customizations, use the following classes. These options objects are added to the Bug Finder options object:

- polyspace.DefectsOptions — Custom list of active defects to check.
- polyspace.GenericTargetOptions — Custom target processor settings.
- polyspace.CodingRulesOptions — Custom list of coding rules to check.

3 Run an analysis on your object with the function `polyspaceBugFinder`.

4 If you want to view and modify your project in the Polyspace environment, create a project from your options object with the method polyspace.options.generateProject.

For an example script, see “Examples”.

Project Files

This method uses a `.psprj` project file to run the analysis. When you create a project in the Polyspace interface, the project file is saved as a `.psprj` file.

1 In the Polyspace interface, create a project on page 1-2.

   Unless you specify a different location, the project is saved as a `.bf.psprj` file in your Polyspace Workspace.

2 In MATLAB, run an analysis on your project file with this line of code:

```
polyspaceBugFinder(path to .psprj project,'nodesktop')
```

3 To make changes to the project, open the project in the Polyspace interface.

UNIX/DOS Command-Line Analysis Options and Values

This method uses the analysis option name and values that are used in UNIX or DOS. Unless you are adapting a UNIX/DOS script to MATLAB, try one of the previous methods first.

In MATLAB, enter analysis options and their values as character vector arguments to the function `polyspaceBugFinder`.

Examples:

- To specify the target processor, use the `-target` option and a supported target.

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

- To add MISRA C:2012 rule checking to your analysis, use the `-misra3` option.

  ```
  polyspaceBugFinder('-sources','file.c','-misra3')
  ```

To see the full list of analysis options, enter:

```
polyspaceBugFinder('-help')
```

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

See Also

`polyspaceBugFinder`

Polyspace Bug Finder Analysis in Simulink

Embedded Coder Considerations

In this section...

Default Options

For Embedded Coder® code, the software sets certain analysis options by default.

Default options for C:

```
-sources path_to_source_code
-results-dir results
-D PST_ERRNO
-D main=main_rtwec __restrict__=
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
-ignore-constant-overflows true
-scalar-overflows-behavior wrap-around
-allow-negative-operand-in-shift true
-boolean-types boolean_T
-functions-to-stub=[rtIsNaN,rtIsInf,rtIsNaNF,rtIsInfF]
```

Default options for C++:

```
-sources path_to_source_code
-results-dir results
-D PST_ERRNO
-D main=main_rtwec __restrict__=
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
```

```
-OS-target no-predfined-OS
-dialect iso
-ignore-constant-overflows true
-scalar-overflows-behavior wrap-around
-allow-negative-operand-in-shift true
-functions-to-stub=[rtIsNaN,rtIsInf,rtIsNaNF,rtIsInfF]
```

Note: *matlabroot* is the MATLAB installation folder.

Recommended Polyspace Bug Finder Options for Analyzing Generated Code

For Embedded Coder code, you can specify other analysis options for your Polyspace Project through the Polyspace **Configuration** pane. To open this pane:

1. In the Simulink model window, select **Code** > **Polyspace** > **Options**. The **Polyspace** pane opens.
2. Click **Configure**. The Polyspace **Configuration** pane opens.

The following table describes options that you should specify in your Polyspace project before analyzing code generated by Embedded Coder software.

Option	Recommended Value	Comments
Macros > **Preprocessor definitions** `-D`	See comments	Defines macro compiler flags used during compilation. Some defines are applied by default, depending on your `-OS-target`. Use one `-D` for each line of the Embedded Coder generated `defines.txt` file. Polyspace does not do this by default.
Environment Settings > **Code from DOS or Windows file system** `-dos`	On	You must select this option if the contents of the include or source directory comes from a DOS or Windows file system. The option allows the analysis to deal with upper/lower case sensitivity and control characters issues. Concerned files are:

Option	Recommended Value	Comments
		• **Header files** – All include folders specified (`-I` option) • **Source files** – All source files selected for the analysis (`-sources` option)

Hardware Mapping Between Simulink and Polyspace

The software automatically imports target word lengths and byte ordering (endianess) from Simulink model hardware configuration settings. The software maps **Device vendor** and **Device type** settings on the Simulink **Configuration Parameters** > **Hardware Implementation** pane to **Target processor type** settings on the Polyspace **Configuration** pane.

The software creates a generic target for the analysis.

TargetLink Considerations

In this section...

TargetLink Support

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 Signal Ranges” on page 9-15.

The software automatically creates a Polyspace constraints file using the dSPACE Data Dictionary for each global variable. The DRS 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 DRS file in the Polyspace user interface. If you define a DRS file, the software appends the automatically generated information to the DRS file you create. Manually defined DRS information overrides automatically generated information for all variables.

DRS 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).

Related Examples

- "Run Analysis for TargetLink" on page 10-5

External Websites

- dSPACE – TargetLink

Generate and Analyze Code

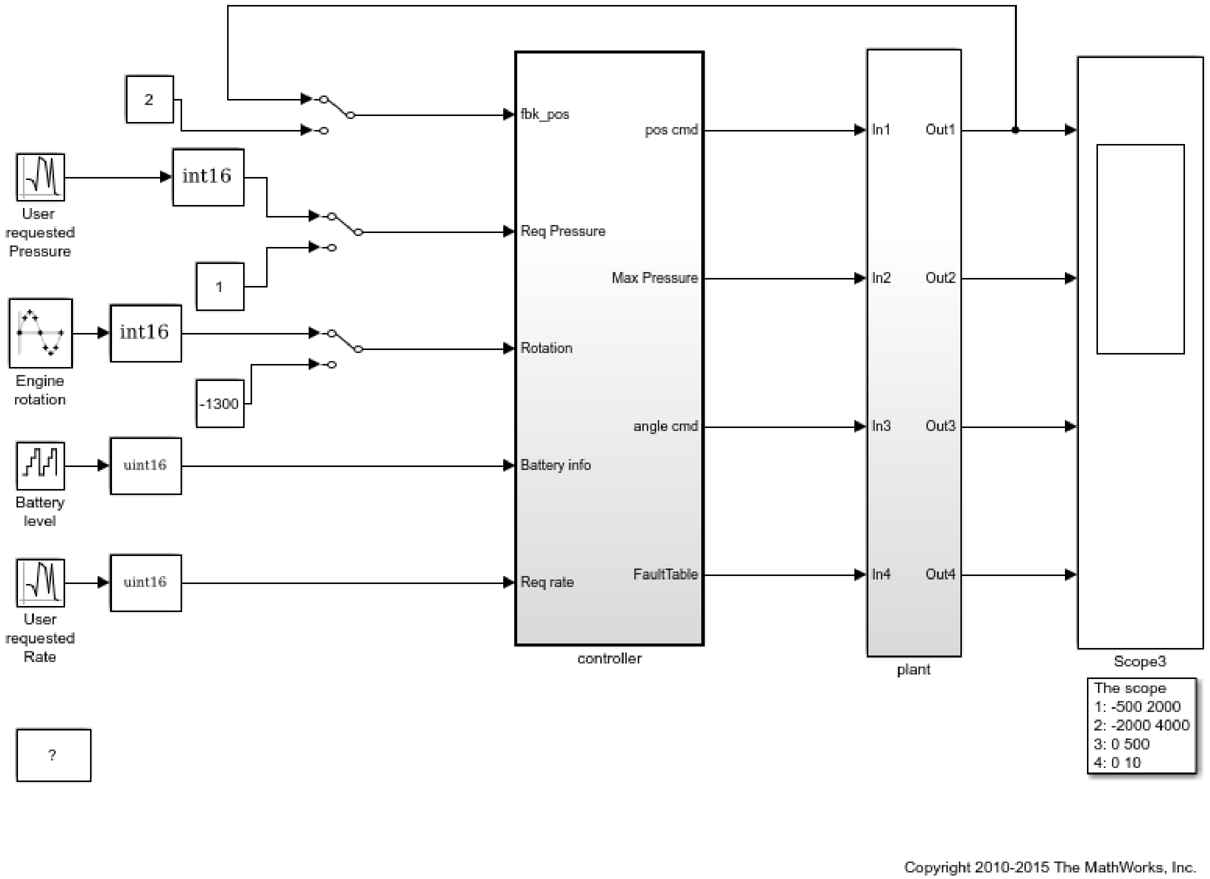
Generate code from referenced models and S-Functions, run a Polyspace analysis from Simulink, and find code defects and MISRA-C:2012 rule violations.

Generate Code and Run Analysis

Before running Polyspace on models, define the scope of your analysis and generate code in Embedded Coder.

1. Open the example model.

```
psdemo_model_link_sl
```



2. Right-click the `controller` subsystem.

3. From the context menu, select **C/C++ Code** > **Build This Subsystem**.

4. In the dialog box, select **Build**.

5. After the build is completed, right-click the `controller` subsystem.

6. From the context menu, select **Polyspace** > **Options**

7. In the Configuration Parameters window, select **Product Mode** > **Bug Finder**.

8. Apply your changes and close the Configuration Parameters window.

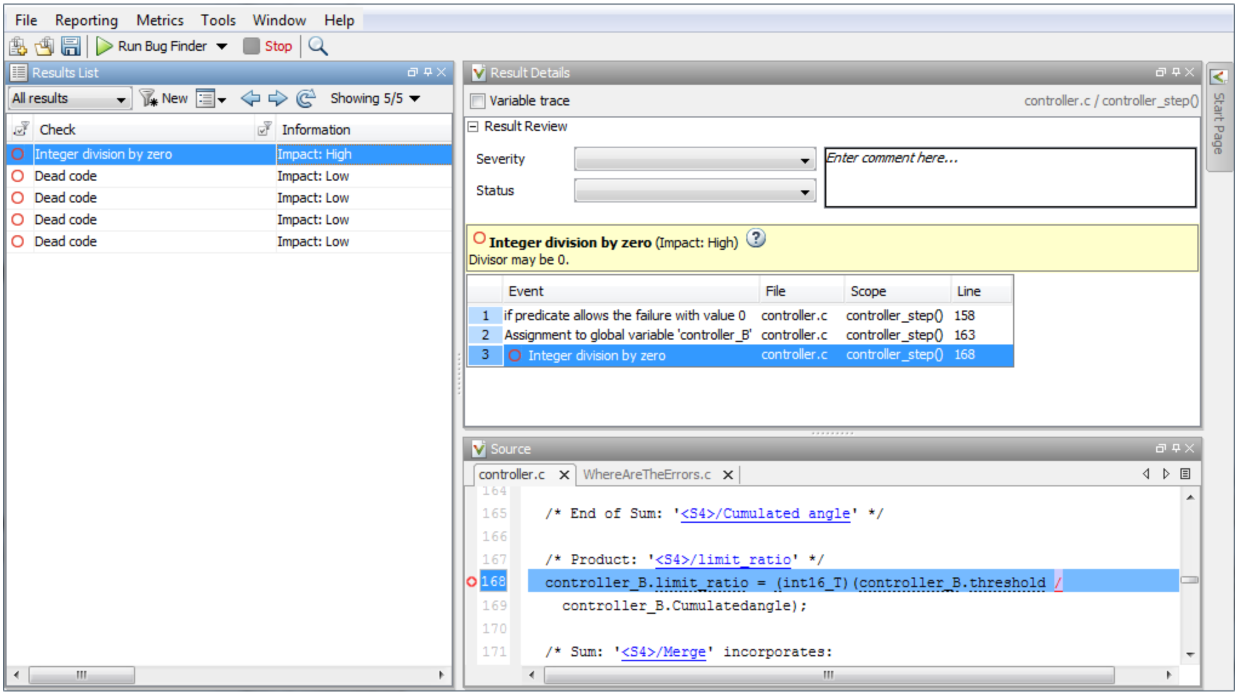
9. Right-click the `controller` subsystem.

10. Select **Polyspace** > **Verify code generated for** > **Selected subsystem**.

You can monitor progress from the Command Window. The results are displayed in the Polyspace environment.

Review Results

In the Polyspace Environment, explore your results and link back to the model.



Check	Information
Integer division by zero	Impact: High
Dead code	Impact: Low
Dead code	Impact: Low
Dead code	Impact: Low
Dead code	Impact: Low

Integer division by zero (Impact: High)
Divisor may be 0.

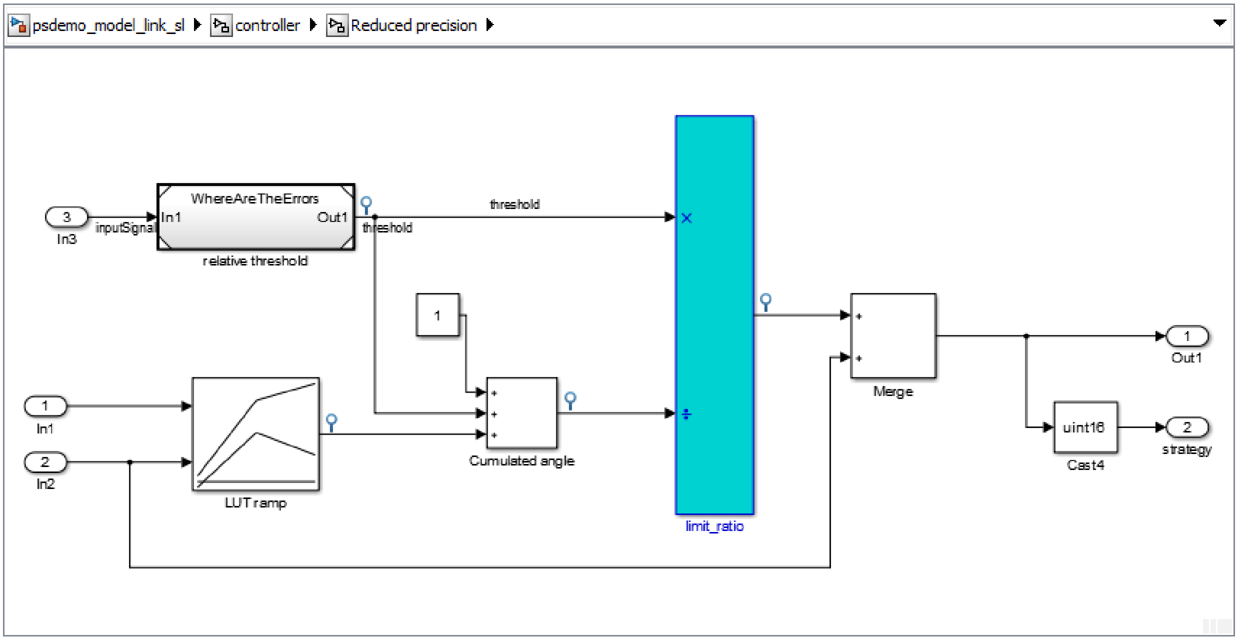
	Event	File	Scope	Line
1	if predicate allows the failure with value 0	controller.c	controller_step()	158
2	Assignment to global variable 'controller_B'	controller.c	controller_step()	163
3	Integer division by zero	controller.c	controller_step()	168

```
  /* End of Sum: '<S4>/Cumulated angle' */

  /* Product: '<S4>/limit_ratio' */
  controller_B.limit_ratio = (int16_T)(controller_B.threshold /
    controller_B.Cumulatedangle);

  /* Sum: '<S4>/Merge' incorporates:
```

1. Select the first result `Integer division by zero`.

This result shows a possible division by zero. The Source pane shows the division operation between variables `controller_B.threshold` and `controller_B.Cumulatedangle`.

2. To see this division operation in your model, select the link `<S4>/limit_ratio`. In your model, the related block is highlighted in blue.



Related Examples

- "Polyspace Configuration for Generated Code" on page 9-2
- "Run Analysis for Embedded Coder" on page 10-3
- "Run Analysis for TargetLink" on page 10-5
- "Configure Model for Code Generation Objectives by Using Code Generation Advisor" (Embedded Coder)

More About

- "Recommended Model Settings for Code Analysis" on page 8-3
- "Troubleshoot Back to Model" on page 7-16

Main Generation for Model Analysis

When you run an analysis, the software automatically reads the following information from the model:

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

The software then uses this information to generate a `main` function that:

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

If the `codeInfo` for the model does not contain the names of the inputs, the software considers all variables as entries, except for parameters and outputs.

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

- For class methods in the generated code, the variables that are written before and in the loop refer to the class members.
- For functions with global scope, the associated variables are also in the global scope.

`main` for Generated Code

The following example shows the `main` generator options that the software uses to generate the `main` function for code generated from a Simulink model.

```
init parameters    \\ -variables-written-before-loop
init_fct()          \\ -functions-called-before-loop
  while(1){        \\ start main loop
  init inputs      \\ -variables-written-in-loop
  step_fct()       \\ -functions-called-in-loop
}
terminate_fct()    \\ -functions-called-after-loop
```

Review Generated Code Results

After you run a Polyspace analysis on generated code, you review the results from the Polyspace environment. From the results you can link back to the related blocks in your model.

1. Open the results using one of the following methods.

   - If you analyzed the whole model, from the Simulink toolbar, select **Code** > **Polyspace** > **Open Results**.

     If you set **Model reference verification depth** to `All` and selected **Model by model verification**. The **Select the Result Folder to Open in Polyspace** dialog box opens showing a hierarchy of referenced models from which the software generates code. To view the analysis results for a specific model, select the model from the hierarchy. Then click **OK**.
   - If you want to open results for a Model block or subsystem, right-click the Model block or subsystem, and from the context menu, select **Polyspace** > **Open Results**.
   - From the Polyspace Interface, select **File** > **Open** and navigate to your results.
   - If you selected **Add to results repository** the results are stored on the Polyspace Metrics server. See “View Results List in Polyspace Metrics” on page 15-23.
2. On the **Results List** tab, select a result.

   When you select a result, the **Result Details** pane shows additional information about the defect, including traceback information (if available).
3. Look at the result in the **Source** pane. Your select result is highlighted in the source code.
4. Hover over the result in the source code. The tooltip can provide additional information including variable ranges.
5. Above the defect, click a blue underlined link. For example, `<Root>/Relational Operator`.

   The Simulink model opens, highlighting the block related to the nearby source code. This back-to-model linking allows you to fix defects in the model instead of the generated code.

6 To investigate a defect, sometimes you have to trace an instance of a variable in generated code back to your model.

Right-click an identifier and select **Go To Model**. The model shows the corresponding block highlighted in blue. If the block is in a subsystem, both the subsystem and the block are highlighted in blue.

Related Examples

- “Result Review Process for Generated Code”
- “Polyspace Bug Finder Results”

More About

- “Troubleshoot Back to Model” on page 7-16

Troubleshoot Back to Model

In this section...
"Back-to-Model Links Do Not Work" on page 7-16
"Your Model Already Uses Highlighting" on page 7-16

Back-to-Model Links Do Not Work

You may encounter issues with the back-to-model feature if:

- Your operating system is Windows Vista™ or Windows 7; and User Account Control (UAC) is enabled or you do not have administrator privileges.
- You have multiple versions of MATLAB installed.

To reconnect MATLAB and Polyspace:

1. Close Polyspace.
2. At the MATLAB command-line, enter `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. To change the color of blocks when they are linked to Polyspace results use this command:

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

Where *`color`* is one of the following:

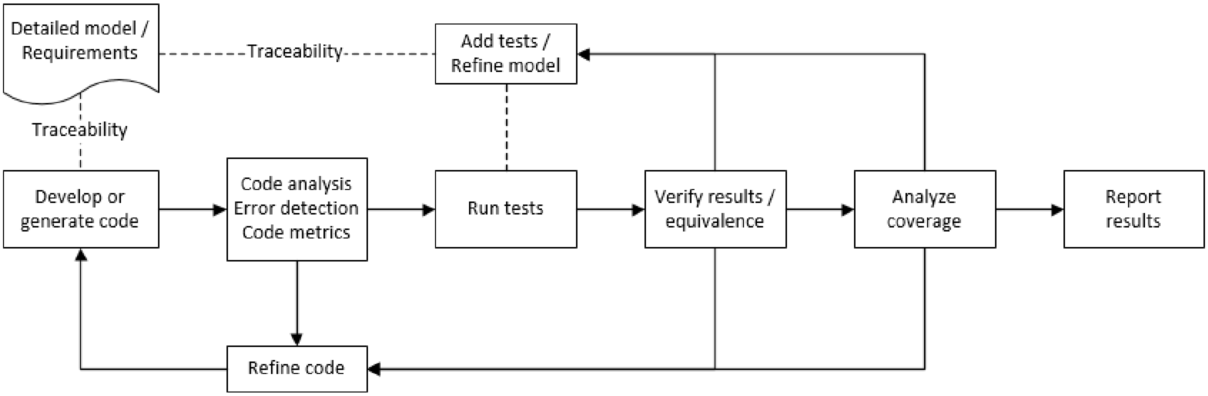
- `'cyan'`
- `'magenta'`
- `'orange'`
- `'lightBlue'`

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

Analyze Code and Test Software-in-the-Loop

Code Analysis and Testing Software-in-the-Loop Overview

Analyze code to detect errors, check standards compliance, and evaluate key metrics such as length and cyclomatic complexity. Typically for handwritten code, you check for run-time errors with static code analysis and run test cases that evaluate the code against requirements and evaluate code coverage. Based on the results, refine the code and add tests. For generated code, demonstrate that code execution produces equivalent results to the model by using the same test cases and baseline results. Compare the code coverage to the model coverage. Based on test results, add tests and modify the model to regenerate code.



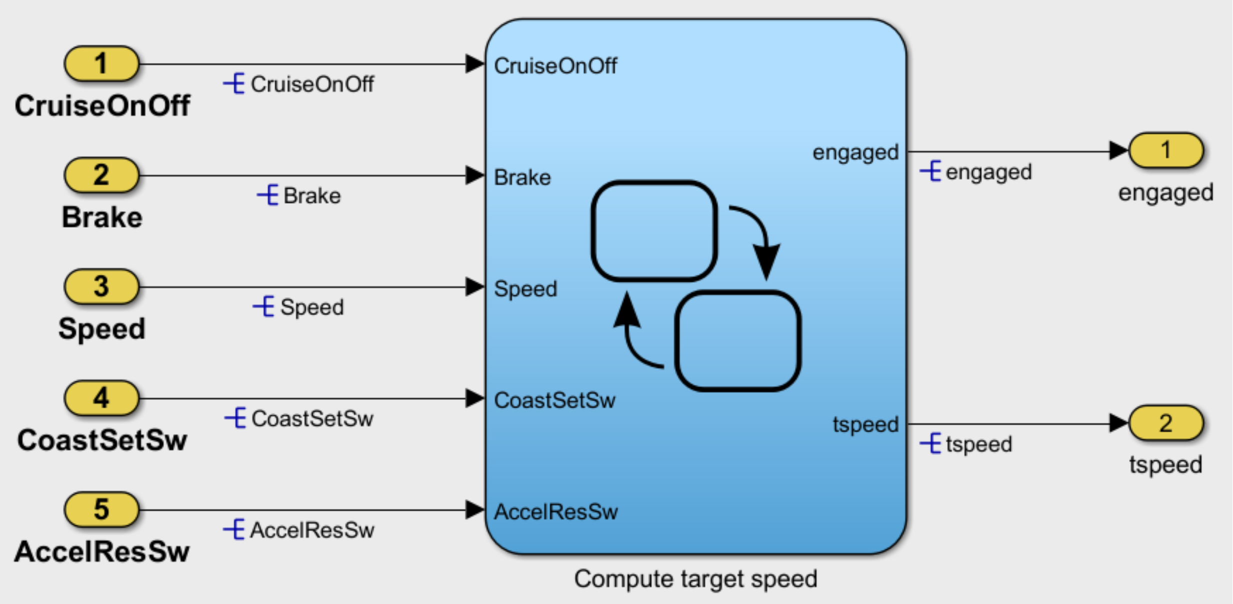
Analyze Code for Defects, Metrics, and MISRA C:2012

This workflow describes how to check if your model produces MISRA C:2012 compliant code and how to check your generated code for code metrics, code defects, and MISRA compliance. To produce more MISRA compliant code from your model, you use the code generation and model advisors. To check whether the code is MISRA compliant, you use the Polyspace MISRA C:2012 checker and report generation capabilities. For this example, you use the model `simulinkCruiseErrorAndStandardsExample`. To open the model:

1 Open the Simulink project:

```
slVerificationCruiseStart
```

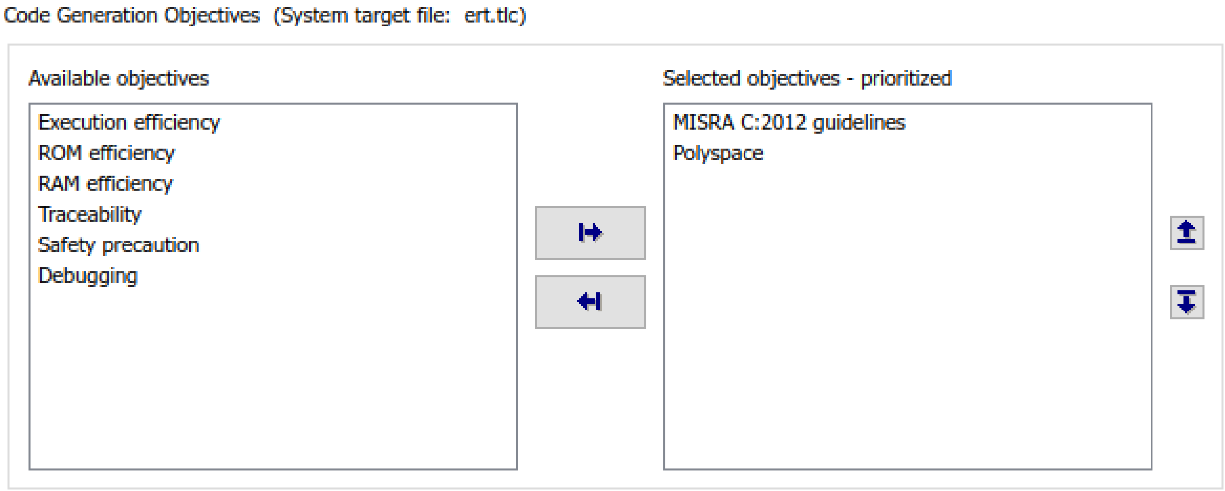
2. From the Simulink project, open the model `simulinkCruiseErrorAndStandardsExample`.



Run Code Generator Checks

Before you generate code from your model, there are steps that you can take to generate code more compliant with MISRA C and more compatible with Polyspace. This example shows how to use the Code Generation Advisor to check your model before generating code.

1. Right-click Compute target speed and select **C/C++** > **Code Generation Advisor**.
2. Select the Code Generation Advisor folder. Add the `Polyspace` objective. The `MISRA C:2012 guidelines` objective is already selected.



Code Generation Objectives (System target file: ert.tlc)

Available objectives

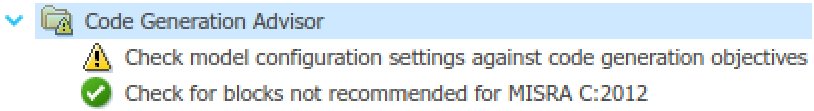
Execution efficiency
ROM efficiency
RAM efficiency
Traceability
Safety precaution
Debugging

Selected objectives - prioritized

MISRA C:2012 guidelines
Polyspace

3 Click **Run Selected Checks**.

The Code Generation Advisor checks whether there are any blocks or configuration settings that are not recommended for MISRA C:2012 compliance and Polyspace code analysis. For this mode, the check for incompatible blocks passes, but there are some configuration settings that are incompatible with MISRA compliance and Polyspace checking.



Code Generation Advisor
⚠ Check model configuration settings against code generation objectives
✓ Check for blocks not recommended for MISRA C:2012

4 Click on check that was not passed. Accept the parameter changes by selecting **Modify Parameters**.

5 Rerun the check by selecting **Run This Check**.

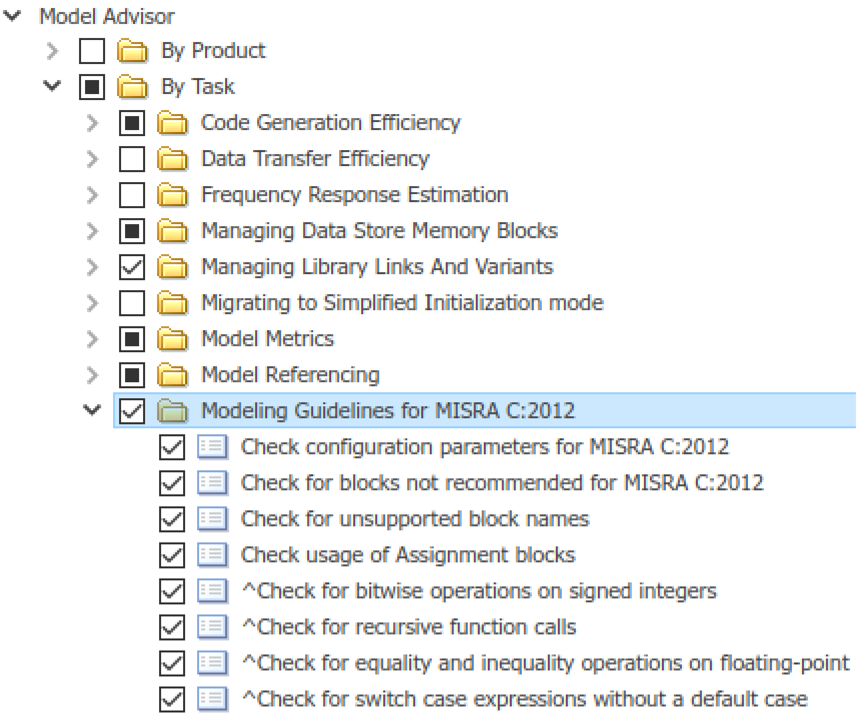
For your own model, you might not want to use all the recommended configuration settings. Using nonrecommended settings can generate less MISRA compliant code.

Run Model Advisor Checks

Before you generate code from your model, there are steps you can take to generate code more compliant with MISRA C and more compatible with Polyspace. This example shows you how to use the Model Advisor to check your model further before generating code.

For more checking before generating code, you can also run the Modeling Guidelines for MISRA C:2012.

1 At the bottom of the Code Generation Advisor window, select **Model Advisor**.
2 Under the **By Task** folder, select the **Modeling Guidelines for MISRA C:2012** advisor checks.



- Model Advisor
  - ☐ By Product
  - ▣ By Task
    - ▣ Code Generation Efficiency
    - ☐ Data Transfer Efficiency
    - ☐ Frequency Response Estimation
    - ▣ Managing Data Store Memory Blocks
    - ☑ Managing Library Links And Variants
    - ☐ Migrating to Simplified Initialization mode
    - ▣ Model Metrics
    - ▣ Model Referencing
    - ☑ Modeling Guidelines for MISRA C:2012
      - ☑ Check configuration parameters for MISRA C:2012
      - ☑ Check for blocks not recommended for MISRA C:2012
      - ☑ Check for unsupported block names
      - ☑ Check usage of Assignment blocks
      - ☑ ^Check for bitwise operations on signed integers
      - ☑ ^Check for recursive function calls
      - ☑ ^Check for equality and inequality operations on floating-point
      - ☑ ^Check for switch case expressions without a default case

3 Click **Run Selected Checks** and review the results.
4 If any of the tasks fail, make the suggested modifications and rerun the checks until the MISRA modeling guidelines pass.

For your own model, you might not want to use all the recommendations. Using nonrecommended settings or blocks can generate less MISRA compliant code.

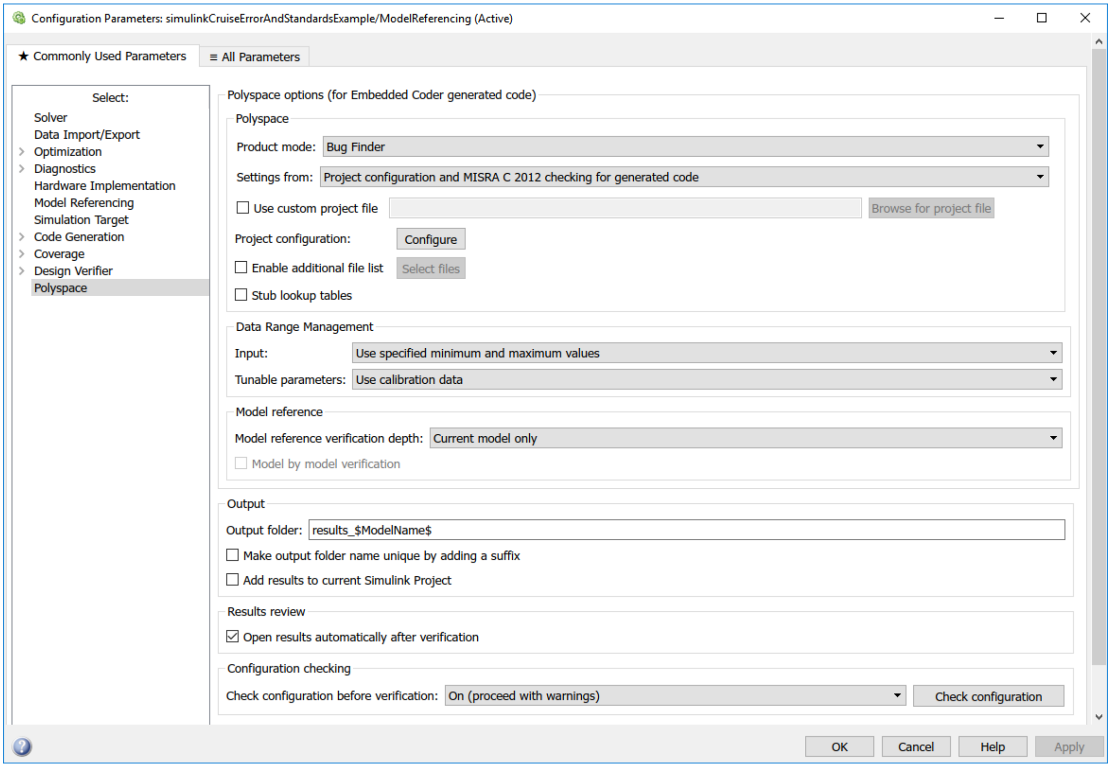
Generate and Analyze Code

After you have done the model compliance checking, you can now generate code. With Polyspace, you can check your code for compliance with MISRA C:2012 and generate reports to demonstrate compliance with MISRA C:2012.

1 In the Simulink editor, right-click Compute target speed and select **C/C++ > Build This Subsystem**.

2 Use the default settings for the tunable parameters and select **Build**.

3 After the code is generated, right-click Compute target speed and select **Polyspace > Options**.



Polyspace options (for Embedded Coder generated code)

Polyspace

Product mode: Bug Finder

Settings from: Project configuration and MISRA C 2012 checking for generated code

□ Use custom project file [Browse for project file]

Project configuration: [Configure]

□ Enable additional file list [Select files]

□ Stub lookup tables

Data Range Management

Input: Use specified minimum and maximum values

Tunable parameters: Use calibration data

Model reference

Model reference verification depth: Current model only

□ Model by model verification

Output

Output folder: results_$ModelName$

□ Make output folder name unique by adding a suffix

□ Add results to current Simulink Project

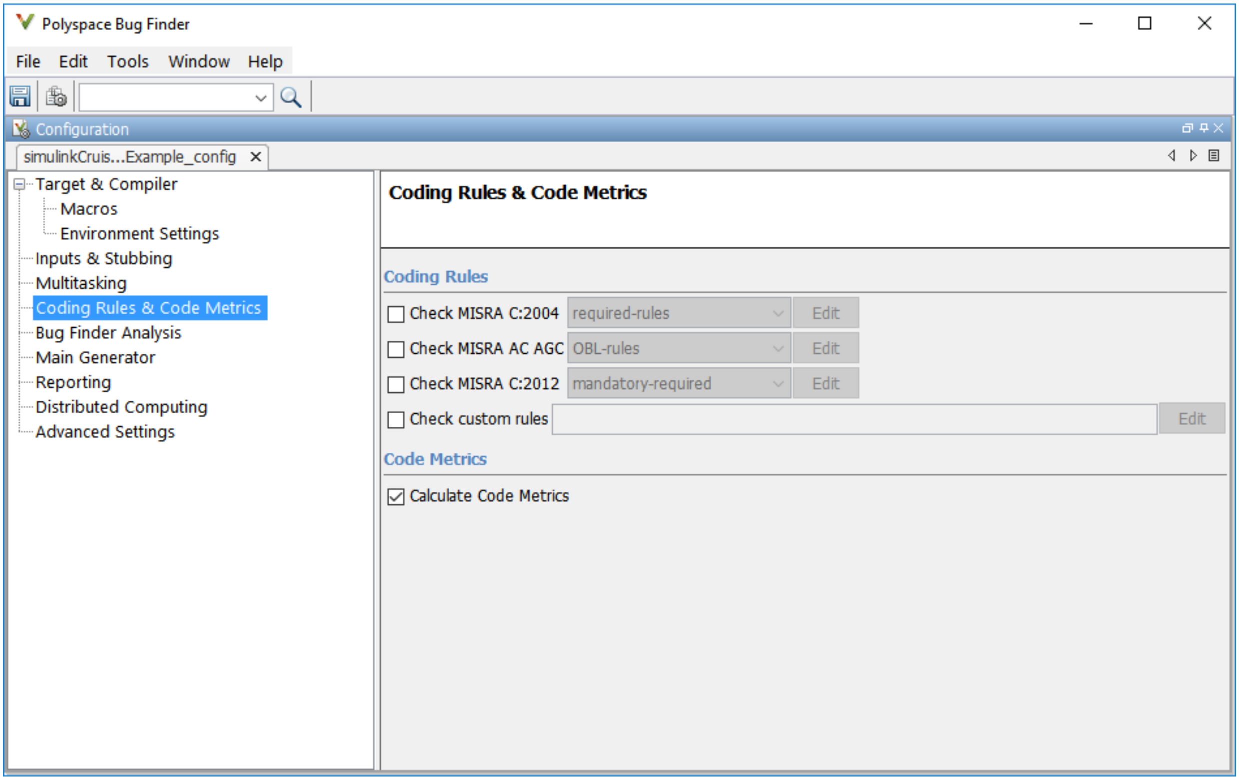
Results review

☑ Open results automatically after verification

Configuration checking

Check configuration before verification: On (proceed with warnings) [Check configuration]

4 Click the **Configure** button. This option allows you to choose more advanced Polyspace analysis options in the Polyspace configuration window.



Coding Rules & Code Metrics

Coding Rules

□ Check MISRA C:2004 required-rules Edit

□ Check MISRA AC AGC OBL-rules Edit

□ Check MISRA C:2012 mandatory-required Edit

□ Check custom rules Edit

Code Metrics

☑ Calculate Code Metrics

5 On the same pane, select **Calculate Code Metrics**. This option turns on code metric calculations for your generated code.

6 Save and close the Polyspace configuration window.

7 From your model, right-click Compute target speed and select **Polyspace** > **Verify Code Generated For** > **Selected Subsystem**.

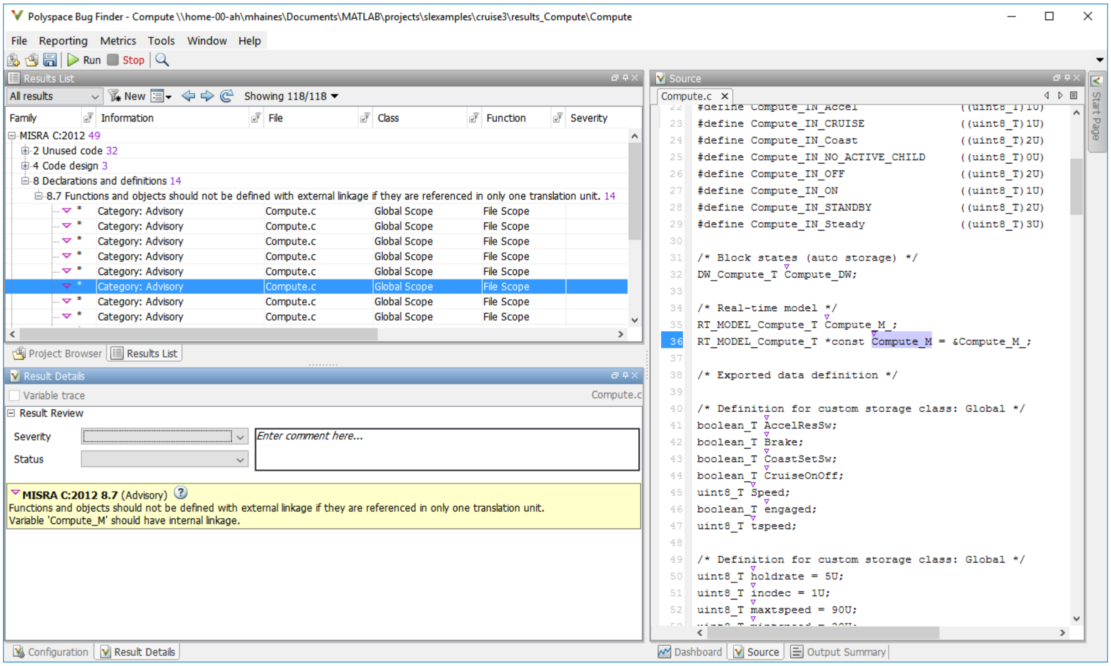
Polyspace Bug Finder analyzes the generated code for a subset of MISRA checks and defect checks. You can see the progress of the analysis in the MATLAB Command Window. Once the analysis is finished, the Polyspace environment opens.

Review Results

After you run a Polyspace analysis of your generated code, the Polyspace environment shows you the results of the static code analysis. There are 50 MISRA C:2012 coding rule violations in your generated code.

1 Expand the tree for rule 8.7 and click through the different results.

Rule 8.7 states that functions and objects should not be global if the function or object is local. As you click through the 8.7 violations, you can see that these results refer to variables that other components also use, such as `CruiseOnOff`. You can annotate your code or your model to justify every result. But, because this model is a unit in a larger program, you can also change the configuration of the analysis to check only a subset of MISRA rules.



Polyspace Bug Finder - Compute \\home-00-ah\mhaines\Documents\MATLAB\projects\slexamples\cruise3\results_Compute\Compute

Family	Information	File	Class	Function	Severity
MISRA C:2012 49					
2 Unused code 32					
4 Code design 3					
8 Declarations and definitions 14					
8.7 Functions and objects should not be defined with external linkage if they are referenced in only one translation unit. 14					
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	

```
#define Compute_IN_Accel                  ((uint8_T)1U)
#define Compute_IN_CRUISE                 ((uint8_T)1U)
#define Compute_IN_Coast                  ((uint8_T)2U)
#define Compute_IN_NO_ACTIVE_CHILD        ((uint8_T)0U)
#define Compute_IN_OFF                    ((uint8_T)2U)
#define Compute_IN_ON                     ((uint8_T)1U)
#define Compute_IN_STANDBY                ((uint8_T)2U)
#define Compute_IN_Steady                 ((uint8_T)3U)

/* Block states (auto storage) */
DW_Compute_T Compute_DW;

/* Real-time model */
RT_MODEL_Compute_T Compute_M_;
RT_MODEL_Compute_T *const Compute_M = &Compute_M_;

/* Exported data definition */

/* Definition for custom storage class: Global */
boolean_T AccelResSw;
boolean_T Brake;
boolean_T CoastSetSw;
boolean_T CruiseOnOff;
uint8_T Speed;
boolean_T engaged;
uint8_T tspeed;

/* Definition for custom storage class: Global */
uint8_T holdrate = 5U;
uint8_T incdec = 1U;
uint8_T maxtspeed = 90U;
```

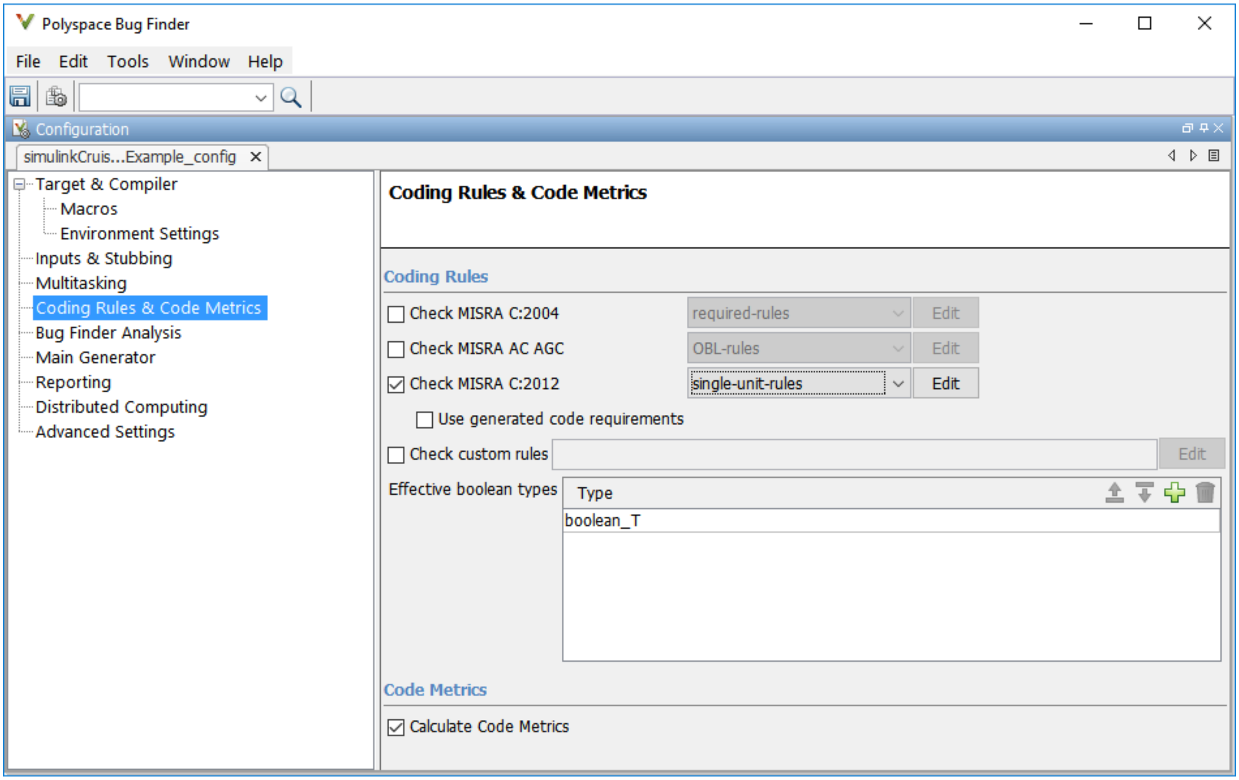
MISRA C:2012 8.7 (Advisory)
Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.
Variable 'Compute_M' should have internal linkage.

2 In your model, right-click Compute target speed and select **Polyspace** > **Options**.

3 Set the **Settings from** option to `Project configuration`. This option allow you to choose a subset of MISRA rules in the Polyspace configuration.

4 Click the **Configure** button.

5 In the Polyspace Configuration window, on the **Coding Rules & Code Metrics** pane, select the check box **Check MISRA C:2012** and from the drop-down list, select `single-unit-rules`. Now, Polyspace checks only the MISRA C:2012 rules that are applicable to a single unit.



Coding Rules & Code Metrics

Coding Rules

☐ Check MISRA C:2004 required-rules

☐ Check MISRA AC AGC OBL-rules

☑ Check MISRA C:2012 single-unit-rules

☐ Use generated code requirements

☐ Check custom rules

Effective boolean types

Type
boolean_T

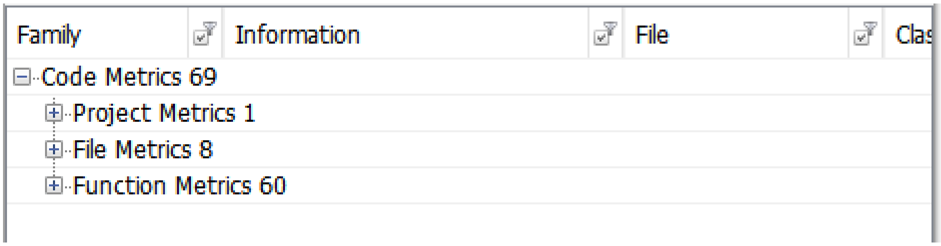
Code Metrics

☑ Calculate Code Metrics

6 Save and close the Polyspace configuration window.

7 Rerun the analysis with the new configuration.

When the Polyspace environment reopens, there are no MISRA results, only code metric results. The rules Polyspace showed previously were found because the model was analyzed by itself. When you limited the rules Polyspace checked to the single-unit subset, no violations were found.



Family	Information	File	Clas
⊟ Code Metrics 69			
⊞ Project Metrics 1			
⊞ File Metrics 8			
⊞ Function Metrics 60			

When this model is integrated with its parent model, you can add the rest of the MISRA C:2012 rules.

Generate Report

To demonstrate compliance with MISRA C:2012 and report on your generated code metrics, you must export your results. This section shows you how to generate a report after the analysis. If you want to generate a report every time you run an analysis, see Generate report.

1. If they are not open already, open your results in the Polyspace environment.
2. From the toolbar, select **Reporting** > **Run Report**.
3. Select **BugFinderSummary** as your report type.
4. Click **Run Report**.

   The report is saved in the same folder as your results.
5. To open the report, select **Reporting** > **Open Report**.

Related Examples

- “Generate and Analyze Code” on page 7-8
- (Simulink Test)
- “Export Test Results and Generate Reports” (Simulink Test)

Configure Model for Code Analysis

Configure Simulink Model

Before analyzing your generated code, there are certain settings that you should apply to your model. Use the following workflow to prepare your model for code analysis.

- If you know of results ahead of time, annotate your blocks with Polyspace annotations.
- Set the recommended configuration parameters.
- Double-check your model settings.
- Generate code.
- Set up your Polyspace options.

Recommended Model Settings for Code Analysis

For Polyspace analyses, set the following parameter configurations before generating code. If you do not use the recommended value for `SystemTargetFile`, you get an error. For 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: **All Parameters** 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: **All Parameters** Name: **Code-to-model** Value: ☑ Selected
	Name: `GenerateSampleERTMain (Embedded Coder)` Value: `'off'`	Location: **Code Generation > Templates** Name: **Generate an example main program** Value: □ Not selected

Grouping	Command-Line	Name and Location in Configuration
	Name: `GenerateComments (Simulink Coder)` Value: `'on'`	Location: **Code Generation > Comments** Name: **Include comments** Value: ☑ Selected
Optimization	Name: `DefaultParameterBehavior (Simulink)` Value: `'Inlined'`	Location: **Optimization > Signals and Parameters** Name: **Default parameter behavior** Value: `Inlined`
	Name: `InitFltsAndDblsToZero (Simulink)` Value: `'on'`	Location: **All Parameters** Name: **Use memset to initialize floats and doubles to 0.0** Value: □ Not selected
	Name: `ZeroExternalMemoryAtStartup (Simulink)` 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`
	Name: `Solver (Simulink)` Value: `'FixedStepDiscrete'`	Location: **Solver** Name: **Solver** Value: `discrete (no continuous states)`

Check Simulink Model Settings

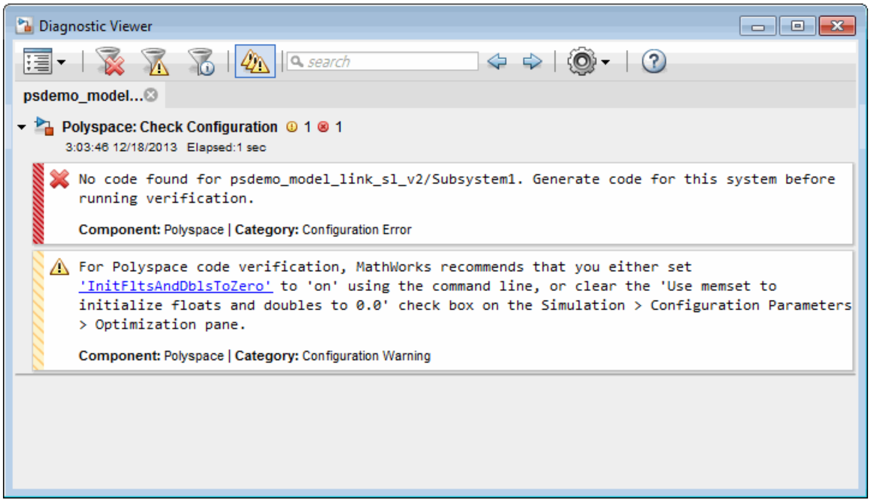
With the Polyspace plug-in, you can check your model settings before generating code or before starting an analysis. If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, warnings appear when you run the analysis.

Check Simulink Model Settings Using the Code Generation Advisor

Before generating code, you can check your model settings against the "Recommended Model Settings for Code Analysis" on page 8-3. If you do not use the recommended model settings, the back-to-model linking will not work correctly.

1. From the Simulink model window, select **Code** > **C/C++ Code** > **Code Generation Options**. The Configuration Parameters dialog box opens, displaying the **Code Generation** pane.
2. Select **Set Objectives**.
3. From the **Set Objective – Code Generation Advisor** window, add the `Polyspace` objective and any others that you want to check.
4. In the **Check model before generating code** drop-down list, select either:
   - `On (stop for warnings)`, the process stops for either errors or warnings without generating code.
   - `On (proceed with warnings)`, the process stops for errors, but continues generating code if the configuration only has warnings.
5. Select **Check Model**.

   The software runs a configuration check. If your configuration check finds errors or warnings, the **Diagnostics Viewer** displays the issues and recommendations.


Diagnostic Viewer
search
psdemo_model...
Polyspace: Check Configuration 1 1
3:03:46 12/18/2013 Elapsed:1 sec

No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before running verification.

Component: Polyspace | **Category:** Configuration Error

For Polyspace code verification, MathWorks recommends that you either set 'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters > Optimization pane.

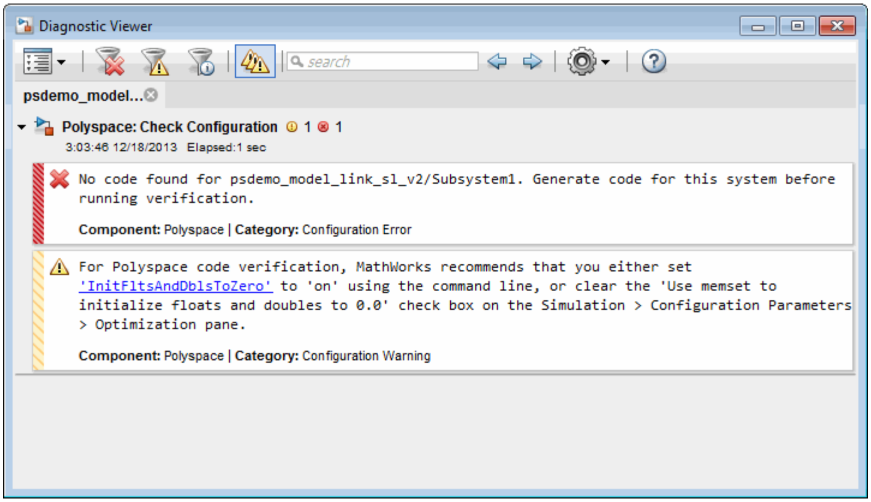
Component: Polyspace | **Category:** Configuration Warning

Check Simulink Model Settings Before Analysis

With the Polyspace plug-in, you can check your model settings before starting an analysis:

1 From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.

2 Click **Check configuration**. If your model settings are not optimal for Polyspace, the software displays warning messages with recommendations.


Diagnostic Viewer
search
psdemo_model...
Polyspace: Check Configuration 1 1
3:03:46 12/18/2013 Elapsed:1 sec

No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before running verification.

Component: Polyspace | **Category:** Configuration Error

For Polyspace code verification, MathWorks recommends that you either set 'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters > Optimization pane.

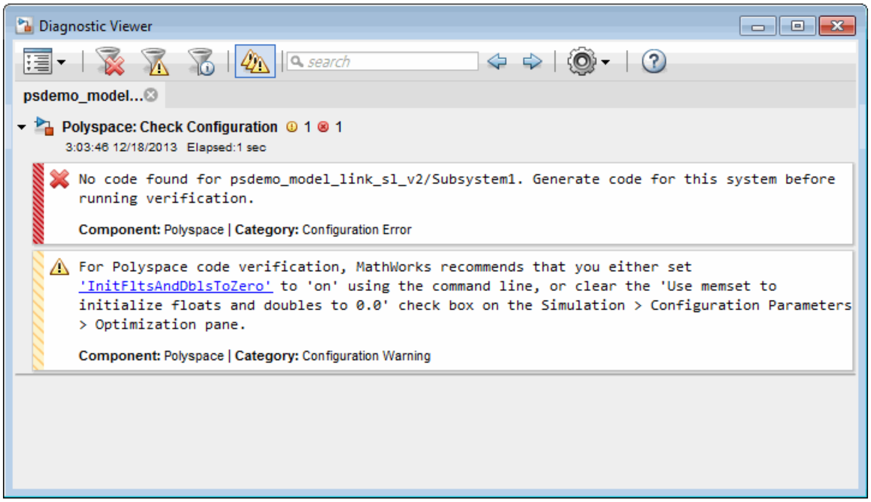
Component: Polyspace | **Category:** Configuration Warning

3 From the **Check configuration before verification** menu, select either:

- `On (stop for warnings)`, the analysis stops for either errors or warnings.
- `On (proceed with warnings)`, the analysis stops for errors, but continues the code analysis if the configuration only has warnings.

4 Select **Run verification**.

The software runs a configuration check. If your configuration check finds errors or warnings, the **Diagnostics Viewer** displays the issues and recommendations.



Polyspace: Check Configuration ⚠ 1 ⊗ 1

3:03:46 12/18/2013 Elapsed:1 sec

```
No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before
running verification.
```

Component: Polyspace | **Category:** Configuration Error

```
For Polyspace code verification, MathWorks recommends that you either set
'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to
initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters
> Optimization pane.
```

Component: Polyspace | **Category:** Configuration Warning

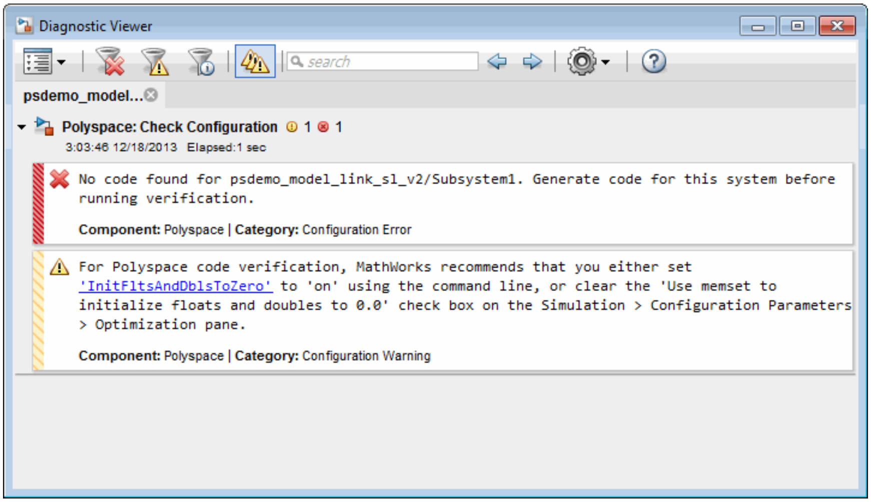
If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, the software produces warnings when you run the analysis.

Check Simulink Model Settings Automatically

With the Polyspace plug-in, you can check your model settings before starting an analysis:

1 From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.

2 Click **Check configuration**. If your model settings are not optimal for Polyspace, the software displays warning messages with recommendations.


Diagnostic Viewer
search
psdemo_model...
Polyspace: Check Configuration ⚠ 1 ⊗ 1
3:03:46 12/18/2013 Elapsed:1 sec

```
No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before
running verification.
```

Component: Polyspace | **Category:** Configuration Error

```
For Polyspace code verification, MathWorks recommends that you either set
'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to
initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters
> Optimization pane.
```

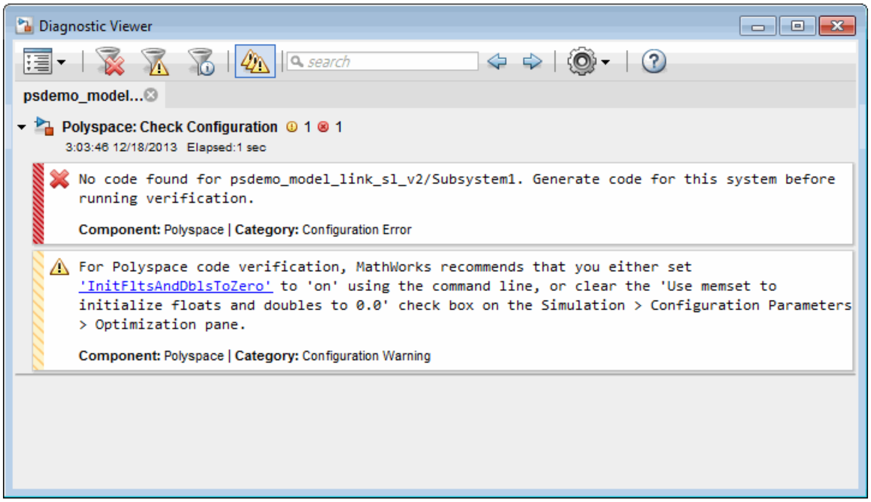
Component: Polyspace | **Category:** Configuration Warning

3 From the **Check configuration before verification** menu, select either:

- `On (stop for warnings)` — will
- `On (proceed with warnings)`

4 Select **Run verification**.

The software runs a configuration check. If your configuration check finds errors or warnings, the **Diagnostics Viewer** displays the issues and recommendations.



Diagnostic Viewer
search
psdemo_model...
Polyspace: Check Configuration 1 1
3:03:46 12/18/2013 Elapsed:1 sec

```
No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before
running verification.
```

Component: Polyspace | **Category:** Configuration Error

```
For Polyspace code verification, MathWorks recommends that you either set
'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to
initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters
> Optimization pane.
```

Component: Polyspace | **Category:** Configuration Warning

If you select:

- `On (stop for warnings)`, the analysis stops for either errors or warnings.
- `On (proceed with warnings)` — the analysis stops for errors, but continues the code analysis if the configuration only has warnings.

If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, the software produces warnings when you run the analysis.

More About

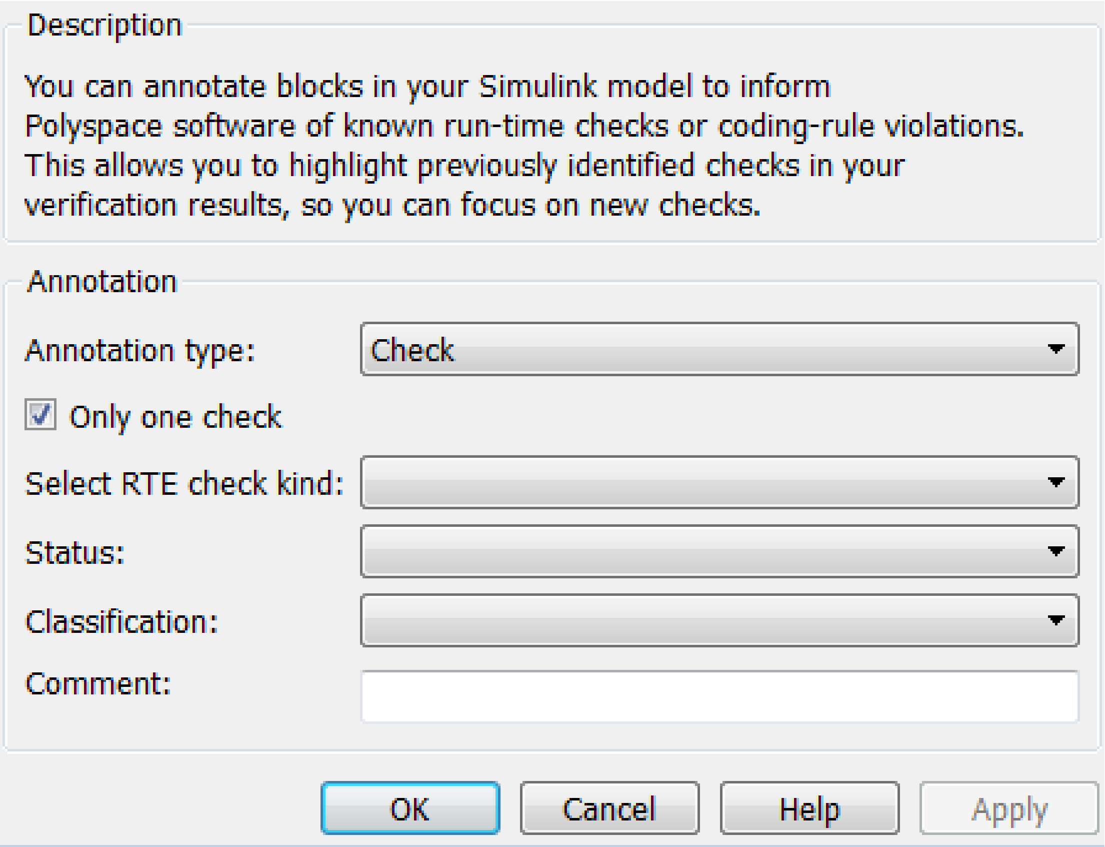
- "Recommended Model Settings for Code Analysis" on page 8-3

Annotate Blocks for Known Results

You can annotate individual blocks in your Simulink model to inform Polyspace software of known defects, run-time checks, or coding-rule violations. These annotations allow you to highlight and categorize previously identified results, so you can focus on reviewing new results.

Your Polyspace results displays the information that you provide with block annotations. To annotate blocks:

1 Right-click the block you want to annotate and select **Polyspace** > **Annotate Selected Block** > **Edit**.



Description

You can annotate blocks in your Simulink model to inform Polyspace software of known run-time checks or coding-rule violations. This allows you to highlight previously identified checks in your verification results, so you can focus on new checks.

Annotation

Annotation type: Check

☑ Only one check

Select RTE check kind:

Status:

Classification:

Comment:

2 In the Polyspace Annotation dialog box, select an **Annotation type**:

- `Check` — Code Prover run-time error
- `Defect` — Bug Finder defect
- `MISRA-C` — MISRA C 2004 coding rule violation
- `MISRA-AC-AGC` — MISRA AC AGC coding rule violation
- `MISRA-C-2012` — MISRA C 2012 coding rule violation
- `MISRA-C++` — MISRA C++ coding rule violation
- `JSF` — JSF C++ coding rule violation

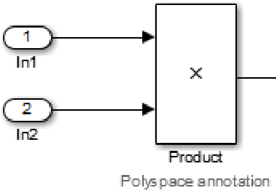
3 If you want to highlight only one kind of result, select **Only 1 check** and the relevant error or coding rule from the **Select *`result`* kind** drop-down list.

4 If you want to highlight a list of checks, clear **Only 1 check**. In the **Enter a list of *`results`*** field, specify the errors or rules that you want to highlight.

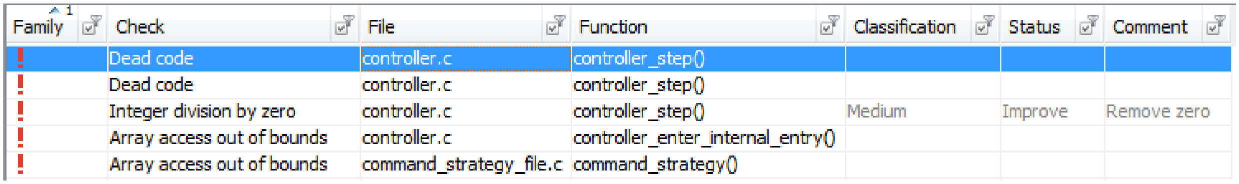
5 Set any of the following options as desired:

Option	Values
Status – describe how you intend to address the issue	`Fix`, `Improve`, `Investigate`, `Other`, `Justified` (This status marks the result as justified), `No action planned` (This status also marks the result as justified.)
Severity — describe the severity of the issue	`High`, `Medium`, `Low`, `Not a defect`
Comment	Any additional information about the check.

6 Click **OK**. You annotation is added to the block.



When you run an analysis, the **Results List** pane pre-populates the results with your annotation.



Family	Check	File	Function	Classification	Status	Comment
!	Dead code	controller.c	controller_step()			
!	Dead code	controller.c	controller_step()			
!	Integer division by zero	controller.c	controller_step()	Medium	Improve	Remove zero
!	Array access out of bounds	controller.c	controller_enter_internal_entry()			
!	Array access out of bounds	command_strategy_file.c	command_strategy()			

See Also

`pslinkfun`

Configure Code Analysis Options

Polyspace Configuration for Generated Code

You do not have to manually create a Polyspace project or specify Polyspace options before running an analysis for your generated code. By default, Polyspace automatically creates a project and extracts the required information from your model. You can modify this configuration and or specify additional options for your analysis with the Polyspace configuration options:

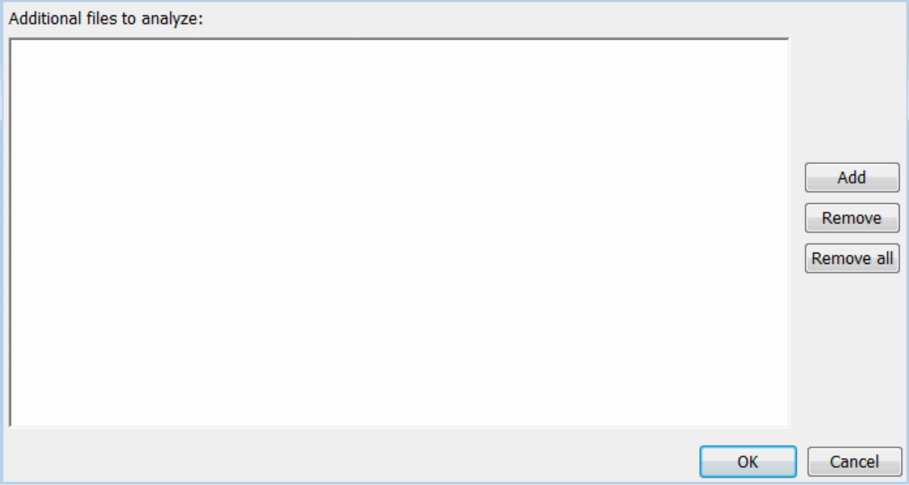
- You may incorporate separately created code within the code generated from your Simulink model. See “Include Handwritten Code” on page 9-3.
- You may customize the options for your analysis. For example, to specify the target environment or adjust precision settings. See “Configure Advanced Polyspace Analysis Options” on page 9-5 and “Recommended Polyspace Bug Finder Options for Analyzing Generated Code” on page 7-3.
- You may create specific configurations for batch runs. See “Use a Saved Polyspace Configuration File Template” on page 9-6.
- If you want to analyze code generated for a 16-bit target processor, you must specify header files for your 16-bit compiler. See “Set Custom Target Settings” on page 9-8.

Include Handwritten Code

Files such as S-Function wrappers are, by default, not part of the Polyspace analysis. However, you can add these files to your generated code analysis manually. You can also analyze your S-Functions separately.

1 From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.

2 Select the **Enable additional file list** check box. Then click **Select files**. The Files Selector dialog box opens.



3 Click **Add**. The Select files to add dialog box opens.

4 Use the Select files to add dialog box to:

- Navigate to the relevant folder
- Add the required files.

The software displays the selected files as a list under **Additional files to analyze**.

Note: To remove a file from the list, select the file and click **Remove**. To remove all files from the list, click **Remove all**.

5 Click **OK**.

Configure Analysis Depth for Referenced Models

From the **Polyspace** pane, you can specify the analysis of generated code with respect to model reference hierarchy levels:

- **Model reference verification depth** — From the drop-down list, select one of the following:
  - `Current model only` — Default. The Polyspace runs code from the top level only. The software creates stubs to represent code from lower hierarchy levels.
  - `1` — The software analyzes code from the top level and the next level. For subsequent hierarchy levels, the software creates stubs.
  - `2` — The software analyzes code from the top level and the next two hierarchy levels. For subsequent hierarchy levels, the software creates stubs.
  - `3` — The software analyzes code from the top level and the next three hierarchy levels. For subsequent hierarchy levels, the software creates stubs.
  - `All` — The software analyzes code from the top level and all lower hierarchy levels.
- **Model by model verification** — Select this check box if you want the software to analyze code from each model separately.

Note: The same configuration settings apply to all referenced models within a top model. The options and parameters are the same whether you open the **Polyspace** pane (**Polyspace** > **Options**) from the toolbar or through the right-click context menu. However, you can run analyses for code generated from specific Model blocks. See “Run Analysis for Embedded Coder” on page 10-3.

Configure Advanced Polyspace Analysis Options

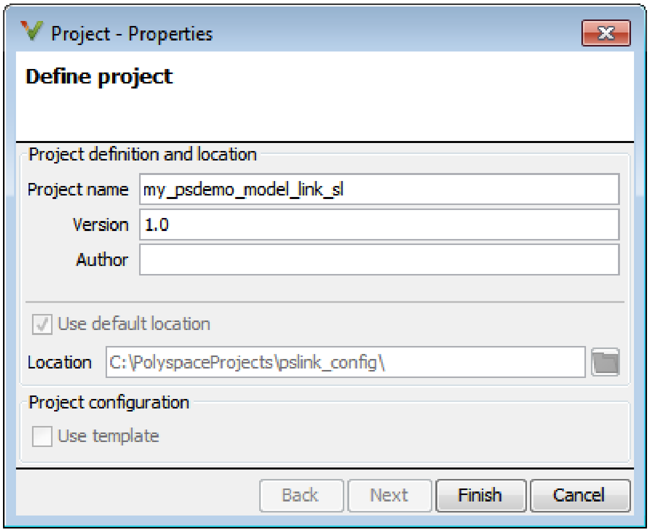
From Simulink, you can specify Polyspace options to change the configuration of the Polyspace analysis. For example, you can specify the processor type and operating system of your target device. For descriptions of options, see “Analysis Options”.

In this section...

Set Advanced Analysis Options

1. From Simulink, select **Code** > **Polyspace** > **Options**.
2. In the Polyspace parameter configuration pane, select **Configure**.
3. In the Polyspace Configuration window that opens, set the options required by your application.

   The first time you open the configuration, the software sets certain options by default depending on your code generator. See Default Embedded Coder Options on page 7-2 or Default TargetLink Options on page 7-5.
4. To change the project name or other project properties, on the toolbar, click the **Project properties** icon



Project - Properties

Define project

Project definition and location

Project name my_psdemo_model_link_sl

Version 1.0

Author

☑ Use default location

Location C:\PolyspaceProjects\pslink_config\

Project configuration

□ Use template

5 Save your changes and close.

6 To use your configuration with other projects, copy the `.psprj` file and rename the updated project configuration file. For example, you can call your cross-compilation configuration `my_cross_compiler.psprj`.

Use a Saved Polyspace Configuration File Template

If you want to reuse a Polyspace configuration for multiple project, you need to add the configuration to the model parameters. This workflow shows how to add a previously created configuration. To create a configuration file template, see "Set Advanced Analysis Options" on page 9-5.

In the Simulink user interface:

1 From Simulink, select **Code** > **Polyspace** > **Options**.

2 In the Polyspace parameter configuration pane, select **Use custom project file**.

3 In the text box, enter the full path to a `.psprj` file, or click **Browse for project file** to browse instead.

At the MATLAB command line:

- Use `pslinkfun('settemplate',...)` to apply a configuration defined by a configuration file template.

  For example:

  ```
  pslinkfun('settemplate','C:\Work\my_cross_compiler.psprj')
  ```

Reset Polyspace Options for a Simulink Model

If you want to reset the Polyspace configuration information to the default, you can remove your custom options from your Simulink model.

1. To remove options from a top model, select **Code** > **Polyspace** > **Remove Options from Current Configuration**.
2. To remove options from a Model block or subsystem, right-click the block or subsystem and select **Polyspace** > **Remove Options from Current Configuration**.
3. Save the model.

See Also

`pslinkfun` | `pslinkoptions`

Related Examples

- "Use a Saved Polyspace Configuration File Template" on page 9-6

More About

- "Embedded Coder Considerations" on page 7-2
- "TargetLink Considerations" on page 7-5
- "Recommended Polyspace Bug Finder Options for Analyzing Generated Code" on page 7-3

Set Custom Target Settings

If your target has specific setting, you can analyze your code in context of those settings. For example, if you want to analyze code generated for a 16-bit target processor, you must specify header files for your 16-bit compiler. The software automatically identifies the compiler from the Simulink model. If the compiler is 16-bit and you do not specify the relevant header files, the Polyspace will produce an error when you try to run an analysis.

Note: For a 32-bit or 64-bit target processor, the software automatically specifies the default header file.

1 In the Simulink model window, select **Code** > **Polyspace** > **Options**.

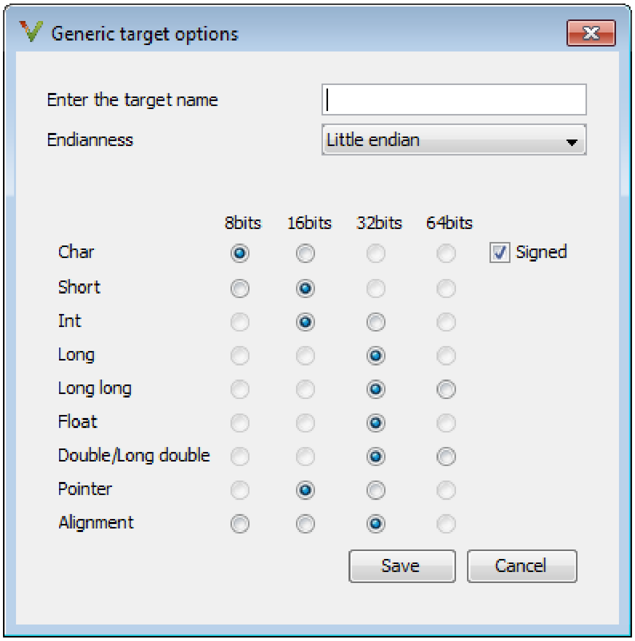
2 Click **Configure**.

   The Polyspace Configuration window opens. Use this pane to customize the target and cross compiler.

3 From the **Configuration** tree, expand the **Target & Compiler** node.

4 In the **Target Environment** section, use the **Target processor type** option to define the size of data types.

   a From the drop-down list, select `mcpu...(Advanced)`. The Generic target options dialog box opens.



Generic target options

Enter the target name

Endianness: Little endian

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

Use this dialog box to create a new target and specify data types for the target. Then click **Save**.

5 From the Configuration tree, select **Target & Compiler** > **Macros**. Use the **Preprocessor definitions** section to define preprocessor macros for your cross-compiler.

To add a macro, in the **Macros** table, select ➕. In the new line, enter the required text.

To remove a macro, select the macro and click 🗑.

Note: If you use the LCC cross-compiler, then you must specify the `MATLAB_MEX_FILE` macro.

---

6 Select **Target & Compiler** > **Environment Settings**.

7 In the **Include folders** (or **Include**) section, specify a folder (or header file) path by doing one of the following:

- Select ✚ and enter the folder or file path.
- Select 📁 and use the dialog box to navigate to the required folder (or file).

You can remove an item from the displayed list by selecting the item and then clicking 🗑.

8 Save your changes and close.

To use your configuration settings in other projects, see “Use a Saved Polyspace Configuration File Template” on page 9-6.

Set Up Remote Analysis

By default, the Polyspace software runs locally. To specify a remote analysis:

1. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.
2. Select **Configure**.
3. In the Polyspace Configuration window, select the **Run Settings** pane.
4. Select the **Run Bug Finder analysis on a remote cluster** check box.
5. If you use Polyspace Metrics as a results repository, select **Upload results to Polyspace Metrics**.
6. If you have not already connected to a server, from the toolbar, select **Options** > **Preferences**. For help filling in this dialog, see "Configure Polyspace Preferences".
7. Close the configuration window and apply your changes.

Manage Results

In this section...

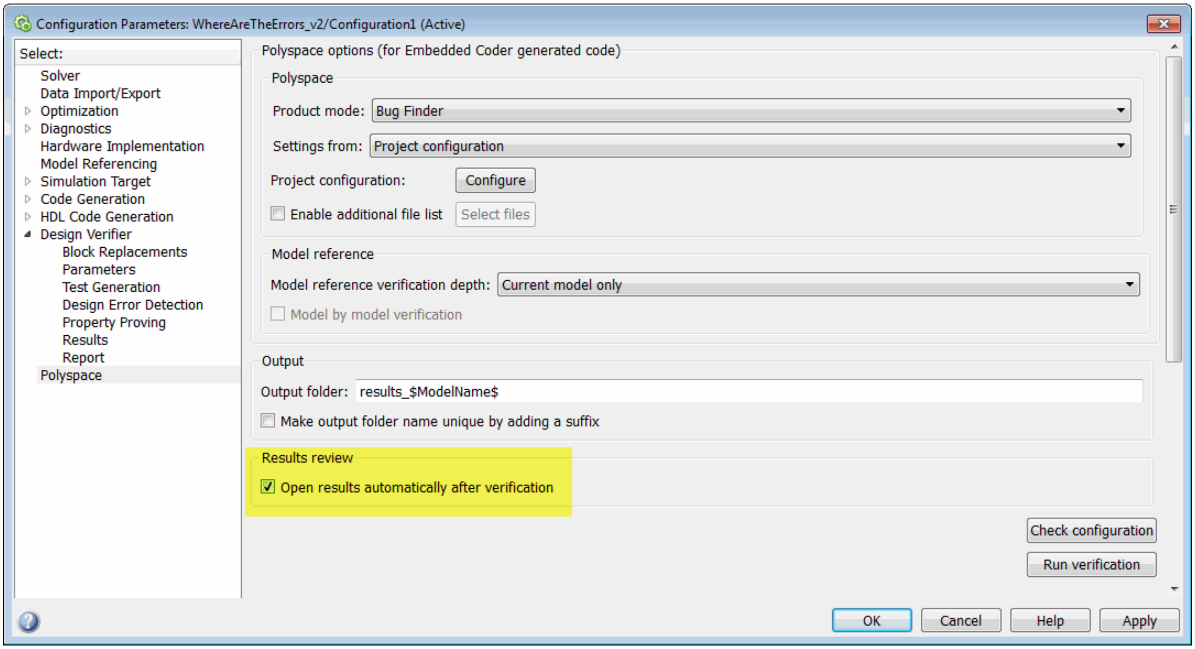
Open Polyspace Results Automatically

You can configure the software to automatically open your Polyspace results after you start the analysis. If you are doing a remote analysis, the Polyspace Metrics web page opens. When the remote job is complete, you can download your results from Polyspace Metrics. If you are doing a local analysis, when the local job is complete, the Polyspace environment opens the results in the Polyspace interface.

To configure the results to open automatically:

1 From the model window, select **Code** > **Polyspace** > **Options**.

The Polyspace pane opens.



Configuration Parameters: WhereAreTheErrors_v2/Configuration1 (Active)

Polyspace options (for Embedded Coder generated code)

Polyspace

Product mode: Bug Finder

Settings from: Project configuration

Project configuration: Configure

☐ Enable additional file list Select files

Model reference

Model reference verification depth: Current model only

☐ Model by model verification

Output

Output folder: results_$ModelName$

☐ Make output folder name unique by adding a suffix

Results review

☑ Open results automatically after verification

2 In the Results review section, select **Open results automatically after verification**.

3 Click **Apply** to save your settings.

Specify Location of Results

By default, the software stores your results in *Current Folder*\results_*model_name*. Every time you rerun, your old results are over written. To customize these options:

1 From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens to the Polyspace pane.

2 In the **Output folder** field, specify a full path for your results folder. By default, the software stores results in the current folder.

3 If you want to avoid overwriting results from previous analyses, select **Make output folder name unique by adding a suffix**.

Instead of overwriting an existing folder, the software specifies a new location for the results folder by appending a unique number to the folder name.

Save Results to a Simulink Project

By default, the software stores your results in `Current Folder\results_model_name`. If you use a Simulink project for your model work, you can store your Polyspace results there as well for better organization. To add your results to a Simulink Project:

1. Open your Simulink project.
2. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens with the Polyspace pane displayed.
3. Select **Add results to current Simulink Project**.
4. Run your analysis.

   Your results are saved to the Simulink project you opened in step 1.

Specify Signal Ranges

If you constrain signals in your Simulink model to lie within specified ranges, Polyspace software automatically applies these constraints during verification of the generated code. Using signal rages can improve the precision of your results.

You can specify a range for a model signal by:

- Applying constraints through source block parameters. See “Specify Signal Range Through Source Block Parameters” on page 9-15.
- Constraining signals through the base workspace. See “Specify Signal Range Through Base Workspace” on page 9-17.

Note: You can also manually define data ranges using the constraint setup feature in the Polyspace user interface. If you manually define a constraint specification file, the software automatically appends any signal range information from your model to the constraint specification file. However, manually defined constraint information overrides information generated from the model for all variables.

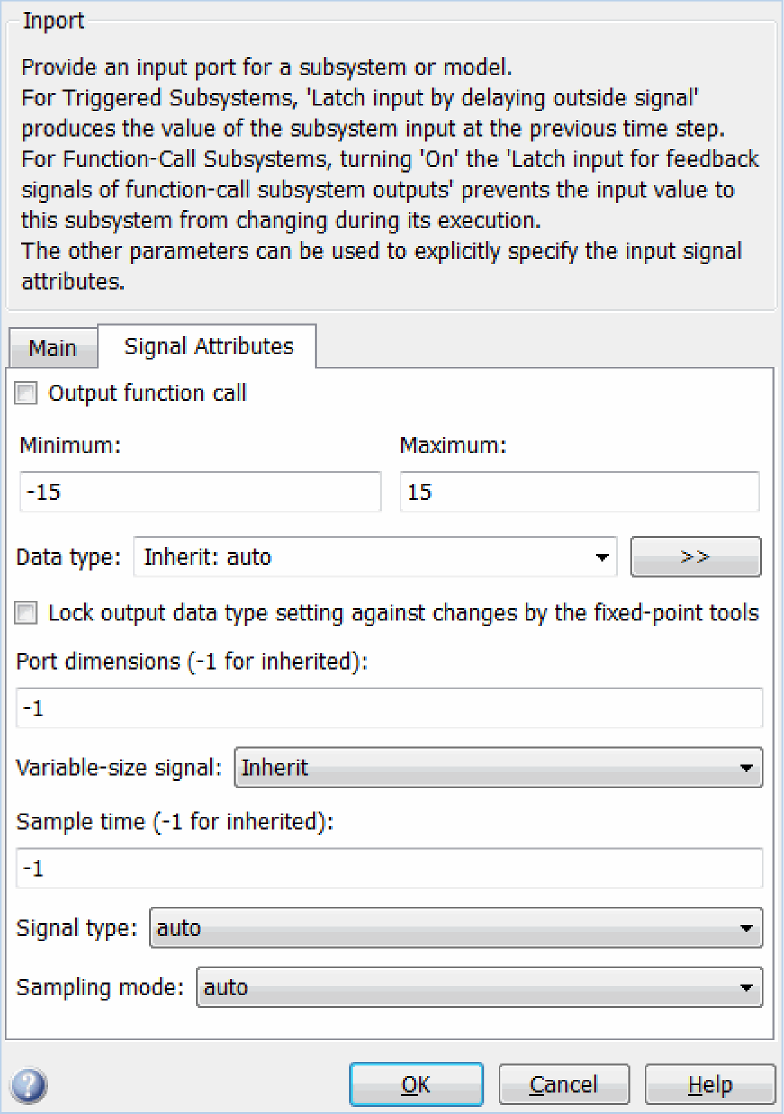
Specify Signal Range Through Source Block Parameters

You can specify a signal range by applying constraints to source block parameters.

Specifying a range through source block parameters is often easier than creating signal objects in the base workspace, but must be repeated for each source block. For information on using the base workspace, see “Specify Signal Range Through Base Workspace” on page 9-17.

To specify a signal range using source block parameters:

1. Double-click the source block in your model. The Source Block Parameters dialog box opens.
2. Select the **Signal Attributes** tab.
3. Specify the **Minimum** value for the signal, for example, -15.
4. Specify the **Maximum** value for the signal, for example, 15.



Inport

Provide an input port for a subsystem or model.
For Triggered Subsystems, 'Latch input by delaying outside signal' produces the value of the subsystem input at the previous time step.
For Function-Call Subsystems, turning 'On' the 'Latch input for feedback signals of function-call subsystem outputs' prevents the input value to this subsystem from changing during its execution.
The other parameters can be used to explicitly specify the input signal attributes.

□ Output function call

Minimum: -15

Maximum: 15

Data type: Inherit: auto

□ Lock output data type setting against changes by the fixed-point tools

Port dimensions (-1 for inherited): -1

Variable-size signal: Inherit

Sample time (-1 for inherited): -1

Signal type: auto

Sampling mode: auto

5 Click **OK**.

Specify Signal Range Through Base Workspace

You can specify a signal range by creating signal objects in the MATLAB workspace. This information is used to initialize each global variable to the range of valid values, as defined by the min-max information in the workspace.

Note: You can also specify a signal range by applying constraints to individual source block parameters. This method can be easier than creating signal objects in the base workspace, but must be repeated for each source block. For more information, see "Specify Signal Range Through Source Block Parameters" on page 9-15.

To specify an input signal range through the base workspace:

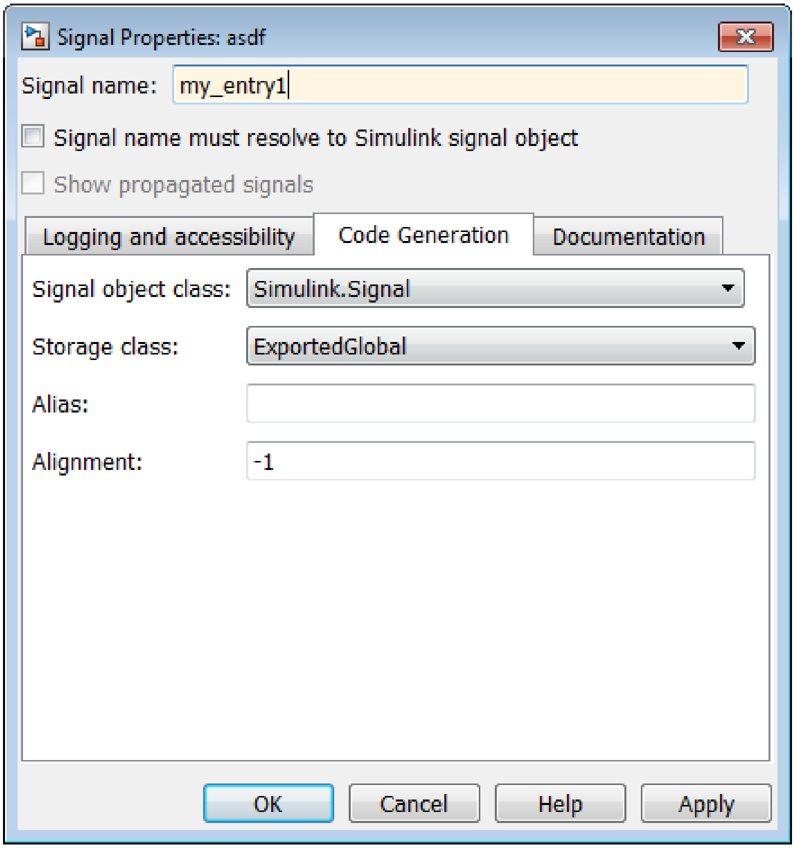
1 Configure the signal to use, for example, the `ExportedGlobal` storage class:

   a Right-click the signal. From the context menu, select **Properties**. The Signal Properties dialog box opens.

   b In the **Signal name** field, enter a name, for example, `my_entry1`.

   c Select the **Code Generation** tab.

   d In the **Storage class** drop-down list, select `ExportedGlobal`.



Signal Properties: asdf

Signal name: my_entry1

☐ Signal name must resolve to Simulink signal object

☐ Show propagated signals

Signal object class: Simulink.Signal

Storage class: ExportedGlobal

Alias:

Alignment: -1

e Click **OK**, which applies your changes and closes the dialog box.

2 Using Model Explorer, specify the signal range:

a Select **Tools** > **Model Explorer** to open Model Explorer.

b From the **Model Hierarchy** tree, select **Base Workspace**.

c Create a signal by clicking the `Add Simulink Signal` button. Rename this signal, for example, `my_entry1`.

d Set the **Minimum** value for the signal, for example, to `-15`.

e Set the **Maximum** value for the signal, for example, to `15`.

f From the **Storage class** drop-down list, select `ExportedGlobal`.

g Click **Apply**.

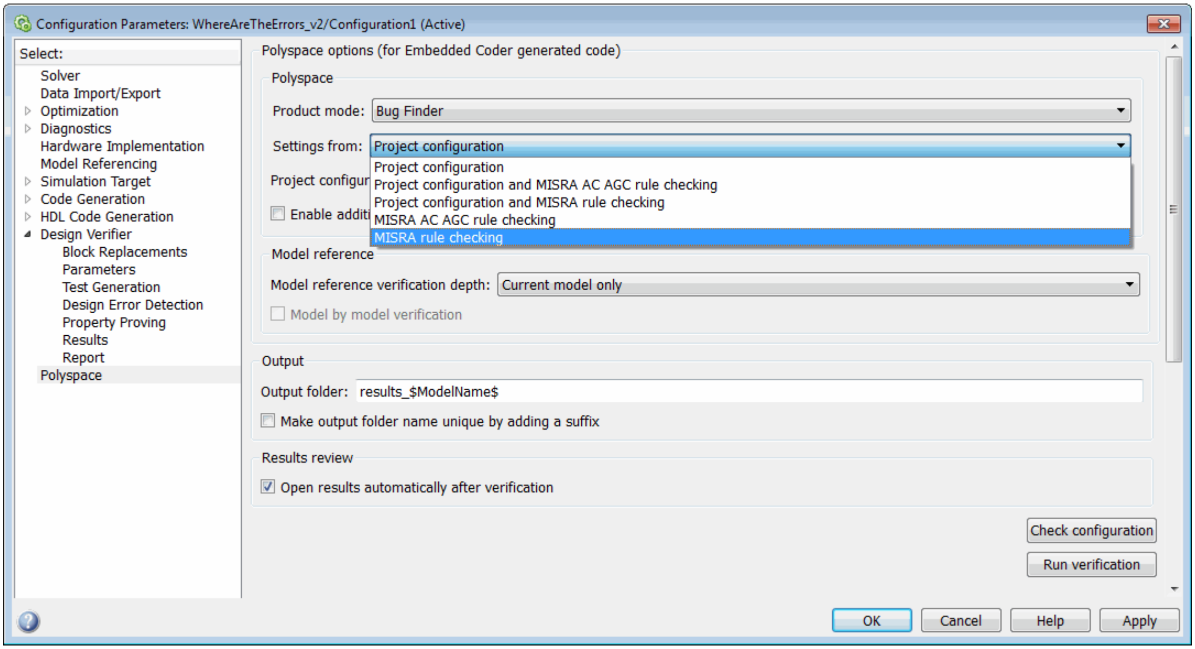
Run Polyspace on Generated Code

Specify Type of Analysis to Perform

Before running Polyspace, you can specify what type of analysis you want to run. You can choose to run code analysis, coding rules checking, or both. You can check compliance with MISRA AC AGC, MISRA C:2004, MISRA C:2012, MISRA C++, and JSF C++ coding rules.

To specify the type of analysis to run:

1 From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameter window opens to the **Polyspace** options pane.



Configuration Parameters: WhereAreTheErrors_v2/Configuration1 (Active)

Polyspace options (for Embedded Coder generated code)

Polyspace

Product mode: Bug Finder

Settings from: Project configuration

Project configur

□ Enable additi

Model reference

Model reference verification depth: Current model only

□ Model by model verification

Output

Output folder: results_$ModelName$

□ Make output folder name unique by adding a suffix

Results review

☑ Open results automatically after verification

2 In the **Settings from** drop-down menu, select the type of analysis you want to perform. For descriptions of the different settings, see “Settings from (C)” or “Settings from (C++)”.

Run Analysis for Embedded Coder

Start the Analysis

There are several different types of analyses you can run on code generated with Embedded Coder. Use the table below to figure out how to start the type of analysis you want.

Code Type to Analyze	What To Select
Code generated from the top model	From the toolbar, select **Code** > **Polyspace** > **Verify Code Generated for** > **Model**.
All code generated as model referenced code	From the toolbar, select **Code** > **Polyspace** > **Verify Code Generated for** > **Referenced Model**.
Model reference code associated with a specific block or subsystem	Right-click the Model block or subsystem and select **Verify Code Generated for** > **Selected Subsystem**

Note: You can also start the Polyspace software from the **Polyspace** configuration parameter pane by clicking **Run verification**.

When the Polyspace software starts, messages appear in the MATLAB Command Window:

```
### Starting Polyspace verification for Embedded Coder
### Creating results folder C:\PolySpace_Results\results_my_first_code
                                           for system my_first_code
### Checking Polyspace Model-Link Configuration:
### Parameters used for code verification:
 System                : my_first_code
 Results Folder        : C:\PolySpace_Results\results_my_first_code
 Additional Files      : 0
 Remote                : 0
 Model Reference Depth : Current model only
 Model by Model        : 0
 DRS input mode        : DesignMinMax
 DRS parameter mode    : None
 DRS output mode       : None
...
```

Follow the progress of the analysis in the MATLAB Command Window. If you are running a remote, batch, analysis you can follow the later stages through the Polyspace Job Monitor.

The software writes status messages to a log file in the results folder.

Monitor Progress

Local Analyses

For a local Polyspace runs, you can follow the progress of the software in the MATLAB Command Window. The software also saves the status messages to a log file in the results folder.

Remote Batch Analyses

For a remote analysis, you can follow the initial stages of the analysis in the MATLAB Command Window.

Once the compilation phase is complete, you can follow the progress of the software using the Polyspace Job Monitor.

From Simulink, select **Code** > **Polyspace** > **Open Job Monitor**.

Run Analysis for TargetLink

To start the Polyspace software on TargetLink generated code:

Start the Analysis

1 In your model, select the Target Link subsystem.

2 In the Simulink model window select **Code** > **Polyspace** > **Verify Code Generated for** > **Selected Target Link Subsystem**.

   Messages appear in the MATLAB Command Window:

```
### Starting Polyspace verification for Embedded Coder
### Creating results folder results_WhereAreTheErrors_v2
                         for system WhereAreTheErrors_v2
### Parameters used for code verification:
    System               : WhereAreTheErrors_v2
    Results Folder       : H:\Desktop\Test_Cases\ModelLink_Testers
                                    \results_WhereAreTheErrors_v2
    Additional Files     : 0
    Verifier settings    : PrjConfig
    DRS input mode       : DesignMinMax
    DRS parameter mode   : None
    DRS output mode      : None
    Model Reference Depth : Current model only
    Model by Model       : 0
```

   The exact messages depend on the code generator you use and the Polyspace product. The software writes status messages to a log file in the results folder.

Follow the progress of the software in the MATLAB Command Window. If you are running a remote, batch analysis, you can follow the later stages in the Polyspace Job Monitor.

Monitor Progress

Local Analyses

For a local Polyspace runs, you can follow the progress of the software in the MATLAB Command Window. The software also saves the status messages to a log file in the results folder.

Remote Batch Analyses

For a remote analysis, you can follow the initial stages of the analysis in the MATLAB Command Window.

Once the compilation phase is complete, you can follow the progress of the software using the Polyspace Job Monitor.

From Simulink, select **Code** > **Polyspace** > **Open Job Monitor**.

Verify 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.

In this section...

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 5-4.
   - 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 Verification and Validation™ function `slcovmex`, with the option `-sldv`. See (Simulink Design Verifier).

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 = { [matlabroot ...
   '\toolbox\polyspace\pslink\pslinkdemos\psdemo_model_link_sl'] };
def.SrcPaths = def.IncPaths;
def.Options.supportCoverageAndDesignVerifier = true;
legacy_code('compile',def);
```

   In Linux, replace \ in the paths with `/`.
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.

Related Examples

- "Include Handwritten Code" on page 9-3
- "Configure Advanced Polyspace Analysis Options" on page 9-5
- "Polyspace Bug Finder Results"
- (Simulink Design Verifier)

Check Coding Rules from Eclipse

Activate Coding Rules Checker

This example shows how to activate the coding rules checker before you start an analysis. This activation enables the Polyspace Bug Finder plug-in to search for coding rule violations. You can view the coding rule violations in your analysis results.

1 Open project configuration.

2 On the **Configuration** pane, select **Coding Rules & Code Metrics**.

3 Select the check box for the type of coding rules that you want to check.

For C code, you can check compliance with:

- MISRA C:2004
- MISRA AC AGC
- MISRA C:2012

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

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

- MISRA C++: 2008
- JSF C++
- Custom coding rules

4 For each rule type that you select, from the drop-down list, select the subset of rules to check.

MISRA C:2004

Option	Description
`required-rules`	All required MISRA C:2004 coding rules.
`all-rules`	AllMISRA C:2004 coding rules (required and advisory).
`SQO-subset1`	A small subset of MISRA C:2004 rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.

Option	Description
`SQO-subset2`	A second subset of rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA C:2004 coding rules that you specify.

MISRA AC AGC

Option	Description
`OBL-rules`	All required MISRA AC AGC coding rules.
`OBL-REC-rules`	All required and recommended MISRA AC AGC coding rules.
`all-rules`	All required, recommended, and readability coding rules.
`SQO-subset1`	A small subset of MISRA AC AGC rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of MISRA AC AGC rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA AC AGC coding rules that you specify.

MISRA C:2012

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

Option	Description
`SQO-subset1`	A small subset of MISRA C rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA C:2012 coding rules that you specify.

MISRA C++

Option	Description
`required-rules`	All required MISRA C++ coding rules.
`all-rules`	All required and advisory MISRA C++ coding rules.
`SQO-subset1`	A small subset of MISRA C++ rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of rules with indirect impact on the selectivity in addition to `SQO-subset1`. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A specified set of MISRA C++ coding rules.

JSF C++

Option	Description
`shall-rules`	**Shall** rules are mandatory requirements. These rules require verification.
`shall-will-rules`	All **Shall** and **Will** rules. **Will** rules are intended to be mandatory requirements. However, these rules do not require verification.
`all-rules`	All **Shall**, **Will**, and **Should** rules. **Should** rules are advisory rules.
`custom`	A set of JSF C++ coding rules that you specify.

5 If you select **Check custom rules**, specify the path to your custom rules file or click **Edit** to create one.

When rules checking is complete, the software displays the coding rule violations in purple on the **Results List** pane.

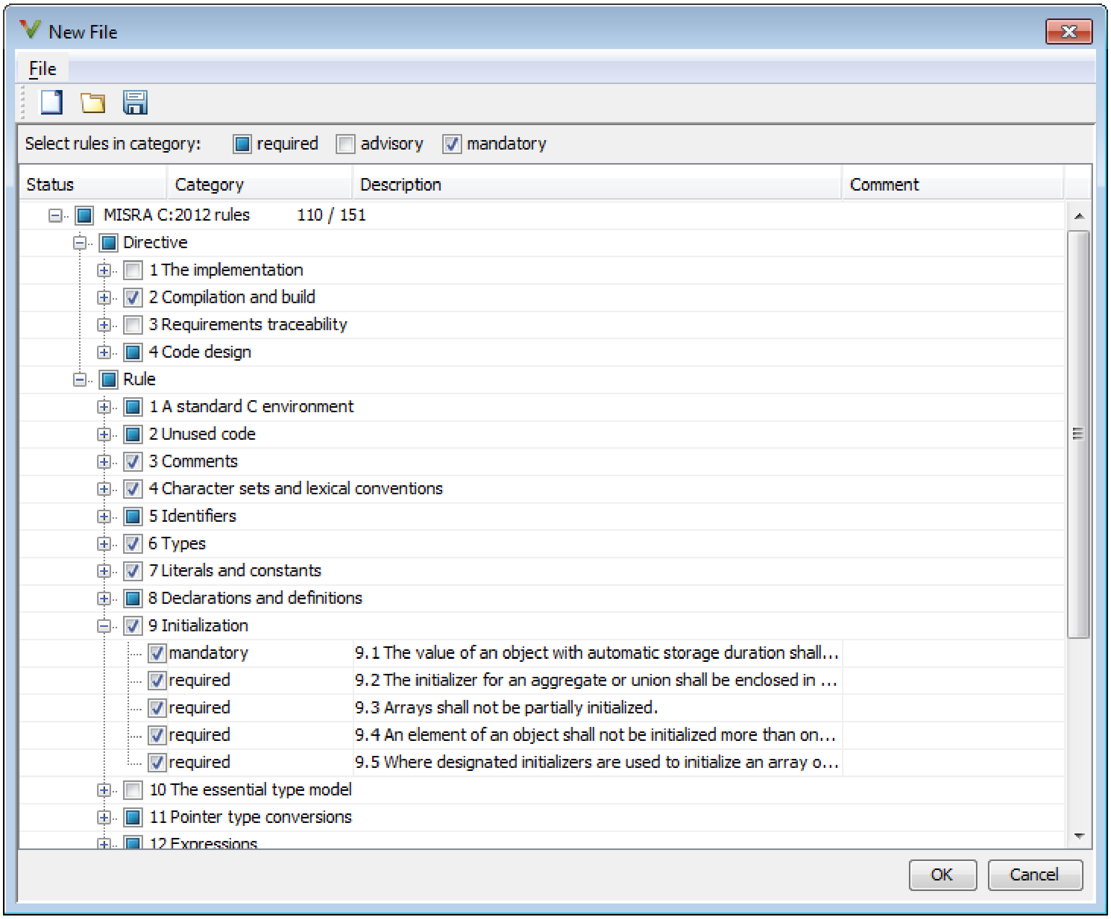
Related Examples

- "Select Specific MISRA or JSF Coding Rules" on page 11-6
- "Create Custom Coding Rules" on page 11-9

Select Specific MISRA or JSF Coding Rules

This example shows how to specify a subset of MISRA or JSF rules for the coding rules checker. If you select `custom` from the MISRA or JSF drop-down list, you must provide a file that specifies the rules to check.

1. Open the project configuration.
2. In the **Configuration** tree view, select **Coding Rules & Code Metrics**.
3. Select the check box for the type of coding rules you want to check.
4. From the corresponding drop-down list, select `custom`. The software displays a new field for your custom file.
5. To the right of this field, click **Edit**. A New File window opens, displaying a table of rules.



New File

File

Select rules in category: ■ required □ advisory ☑ mandatory

Status	Category	Description	Comment
⊟ ■ MISRA C:2012 rules	110 / 151		
⊟ ■ Directive			
⊞ □ 1 The implementation			
⊞ ☑ 2 Compilation and build			
⊞ □ 3 Requirements traceability			
⊞ ■ 4 Code design			
⊟ ■ Rule			
⊞ ■ 1 A standard C environment			
⊞ ■ 2 Unused code			
⊞ ☑ 3 Comments			
⊞ ☑ 4 Character sets and lexical conventions			
⊞ ■ 5 Identifiers			
⊞ ☑ 6 Types			
⊞ ☑ 7 Literals and constants			
⊞ ■ 8 Declarations and definitions			
⊟ ☑ 9 Initialization			
☑	mandatory	9.1 The value of an object with automatic storage duration shall...	
☑	required	9.2 The initializer for an aggregate or union shall be enclosed in ...	
☑	required	9.3 Arrays shall not be partially initialized.	
☑	required	9.4 An element of an object shall not be initialized more than on...	
☑	required	9.5 Where designated initializers are used to initialize an array o...	
⊞ □ 10 The essential type model			
⊞ ■ 11 Pointer type conversions			
⊞ ■ 12 Expressions			

6 If you already have a customized rule file you want to edit, reload your customization using the 📁 button.

7 Select the rules you want to check.

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

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

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

Related Examples

- "Activate Coding Rules Checker" on page 11-2
- "Create Custom Coding Rules" on page 11-9

Create Custom Coding Rules

This example shows how to create a custom coding rules file. You can use this file to check names or text patterns in your source code against custom rules that you specify. For each rule, you specify a pattern in the form of a regular expression. The software compares the pattern against identifiers in the source code and determines whether the custom rule is violated.

1. Create 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 ☑.
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.

Related Examples

- "Activate Coding Rules Checker" on page 11-2
- "Select Specific MISRA or JSF Coding Rules" on page 11-6

More About

- "Contents of Custom Coding Rules File" on page 11-11

Contents of Custom Coding Rules File

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

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

- *N.n* — Custom rule number, for example, 1.2.
- off — Rule is not considered.
- on — The software checks for violation of the rule. After 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-ZO-9_]*
9.1  on    # Non-adherence to custom rule 9.1 produces a warning
convention=Global variables should begin by g_.
pattern=g_.*
```

Related Examples

- “Create Custom Coding Rules” on page 11-9

Exclude Files from Analysis

This example shows how to specify files that you do not want analyzed. For instance, sometimes, for a precise analysis, you have to add header files from a third-party library to your Polyspace project, but you cannot address defects in those header files. Therefore, you do not want analysis results on those files.

By default:

- Results are generated for all source files and header files in the same folders as source files.
- Results are not generated for the remaining header files in your project.

You can change this default behavior and specify your own set of files on which you do not want results.

1. Open the project configuration.
2. In the **Configuration** tree view, select **Inputs & Stubbing**.
3. Use a combination of the following options to suppress results from files in which you are not interested.

    - Do not generate results for (-do-not-generate-results-for)
    - Generate results for sources and (-generate-results-for)

    For instance, you can suppress results from certain folders and unsuppress them only for certain files in those folders.

Related Examples

- "Customize Analysis Options" on page 12-3

Allow Custom Pragma Directives

This example shows how to exclude custom pragma directives from coding rules checking. MISRA C rule 3.4 requires checking that pragma directives are documented within the documentation of the compiler. However, you can allow undocumented pragma directives to be present in your code.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules & Code Metrics**.
3. To the right of **Allowed pragmas**, click ✚.

   In the **Pragma** view, the software displays an active text field.
4. In the text field, enter a pragma directive.
5. To remove a directive from the **Pragma** list, select the directive. Then click 🗑.

Related Examples

- “Activate Coding Rules Checker” on page 11-2

Specify Boolean Types

This example shows how to specify data types you want Polyspace to consider as Boolean during MISRA C rules checking. The software applies this redefinition only to data types defined by `typedef` statements.

The use of this option is related to checking of the following rules:

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

  For more information, see “MISRA C:2004 and MISRA AC AGC Coding Rules” on page 2-14.
- MISRA C:2012 — 10.1, 10.3, 10.5, 14.4 and 16.7

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules & Code Metrics**.
3. To the right of **Effective boolean types**, click ➕.

   In the **Type** view, the software displays an active text field.
4. In the text field, specify the data type that you want Polyspace to treat as Boolean.
5. To remove a data type from the **Type** list, select the data type. Then click 🗑.

Related Examples

- “Activate Coding Rules Checker” on page 11-2

Find Coding Rule Violations

This example shows how to check for coding rule violations alone.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules & Code Metrics**. Activate the desired coding rule checker.

   For more information, see “Activate Coding Rules Checker” on page 3-2.
3. If you select certain rules, the analysis can complete quicker than checking other rules.

   For more information, see “Coding Rule Subsets Checked Early in Analysis” on page 2-61.
4. Specify that the analysis must not look for defects.

   - In the **Configuration** tree view, select **Bug Finder Analysis**.
   - Clear the **Find defects** check box.
5. Click ▶ to run the coding rules checker without checking defects.

Related Examples

- “Activate Coding Rules Checker” on page 11-2
- “Select Specific MISRA or JSF Coding Rules” on page 11-6
- “Review Coding Rule Violations” on page 11-16

Review Coding Rule Violations

This example shows how to review coding rule violations once code analysis is complete. After analysis, the **Results List - Bug Finder** tab displays the rule violations with a

- ▽ symbol for predefined coding rules such as MISRA C:2004.
- ▼ symbol for custom coding rules.

In addition, Polyspace Bug Finder highlights defects in your source code in the following ways:

- A ▽ or ▼ mark appears before the line number on the left.
- A ▭ icon appears in the overview ruler to the right of the line containing the rule violation.

To further review a coding rule violation and determine your course of action:

1. Select the coding rule violation on the **Results List - Bug Finder** tab.
2. On the **Result Details** pane, view the location and description of the violated rule. In the source code, the line containing the violation appears highlighted.
3. For MISRA C: 2012 rules, on the **Result Details** pane, click the ⓘ icon to see the rationale for the rule. In some cases, you can also see code examples illustrating the violation.
4. Review the violation in your code.

   a. Determine whether you must fix the code to avoid the violation.

   b. If you choose to retain the code, on the **Result Details** pane, add a comment explaining why you retain the code. This comment helps you or other reviewers avoid reviewing the same coding rule violation twice.

      You can also assign a **Severity** and **Status** to the coding rule violation.
5. After you have fixed or justified the coding rule violations, run the analysis again.

Related Examples

- "Activate Coding Rules Checker" on page 11-2
- "Find Coding Rule Violations" on page 11-15
- "Filter and Group Coding Rule Violations" on page 11-17

Filter and Group Coding Rule Violations

This example shows how to use filters in the **Results List** pane to focus on specific kinds of coding rule violations. By default, the software displays both coding rule violations and defects.

In this section...

Filter Coding Rules

1. On the **Results List** pane, select the icon on the **Check** column header.
2. From the context menu, clear the **All** check box.
3. Select the violated rule numbers that you want to focus on.
4. Click **OK**.

To filter out all results other than coding rule violations, use the filters on the **Type** or **Family** column header.

You can also filter rule violations using the ***`Coding rule`*** **violations by rule (Top 10 only)** graph on the **Dashboard** pane in the Polyspace user interface. See “Filter and Group Results” on page 5-4.

Group Coding Rules

1. On the **Results List** pane, from the list, select **Family**.

   The rules are grouped by numbers. Each group corresponds to a certain code construct.
2. Expand the group nodes to select an individual coding rule violation.

Suppress Certain Rules from Display in One Click

Instead of filtering individual rules from display each time you open your results, you can limit the display of rule violations in one click. Use the drop-down list in the left of the **Results List** pane toolbar. You can add some predefined options to this list or create your own options. You can share the option file to help developers in your organization review violations of at least certain coding rules.

1 In the Polyspace user interface, select **Tools** > **Preferences**.

2 On the **Review Scope** tab, do one of the following:

   - To add predefined options to the drop-down list on the **Results List** pane, select **Include Quality Objectives Scopes**.

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

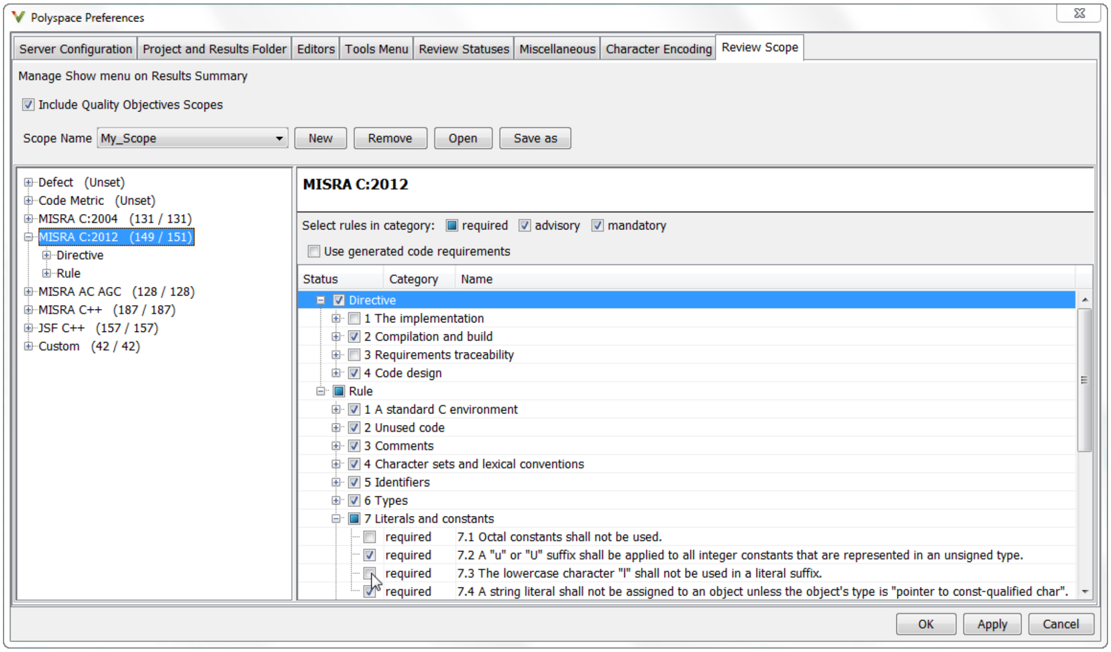
     In addition to coding rule violations, the options impose limits on the display of code metrics and defects.

   - To create your own option in the drop-down list on the **Results List** pane, select **New**. Save your option file.

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

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

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



Manage Show menu on Results Summary

☑ Include Quality Objectives Scopes

Scope Name: My_Scope

- Defect (Unset)
- Code Metric (Unset)
- MISRA C:2004 (131 / 131)
- MISRA C:2012 (149 / 151)
  - Directive
  - Rule
- MISRA AC AGC (128 / 128)
- MISRA C++ (187 / 187)
- JSF C++ (157 / 157)
- Custom (42 / 42)

MISRA C:2012

Select rules in category: ■ required ☑ advisory ☑ mandatory

□ Use generated code requirements

Status	Category	Name
☑ Directive		
□ 1 The implementation		
☑ 2 Compilation and build		
□ 3 Requirements traceability		
☑ 4 Code design		
■ Rule		
☑ 1 A standard C environment		
☑ 2 Unused code		
☑ 3 Comments		
☑ 4 Character sets and lexical conventions		
☑ 5 Identifiers		
☑ 6 Types		
■ 7 Literals and constants		
□	required	7.1 Octal constants shall not be used.
☑	required	7.2 A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.
□	required	7.3 The lowercase character "l" shall not be used in a literal suffix.
☑	required	7.4 A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

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

On the **Results List** pane, the drop-down list on the **Results List** pane displays the additional options.

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

Related Examples

- "Activate Coding Rules Checker" on page 11-2
- "Review Coding Rule Violations" on page 11-16

Find Bugs from Eclipse

Run Analysis

1 Switch to the Polyspace perspective.

   a Select **Window** > **Open Perspective** > **Other**.

   b In the Open Perspective dialog box, select **Polyspace**.

   This allows you to view only the information related to a Polyspace analysis.

2 If you previously ran a Polyspace Code Prover verification, open the **Polyspace Run - Code Prover** view. In the dropdown list beside the ✓ icon, select **Bug Finder**.

3 To start an analysis, do one of the following:

   - In the **Project Explorer**, right-click the project containing the files that you want to verify and select **Run Polyspace Bug Finder**.
   - In the **Project Explorer**, select the project containing the files that you want to verify. From the global menu, select **Polyspace** > **Run**.

   You can follow the progress of the analysis in the **Polyspace Run - Bug Finder** view. If you see an error or warning during the compilation phase, double-click it to go to the corresponding location in the source code. Once the analysis is over, the results are displayed in the **Results List - Bug Finder** view.

4 If results are available, the ⟳ icon in the **Polyspace Run - Bug Finder** view turns to ⤓. Click the ⤓ icon to load available results.

   With your results open, if additional results are available, the ⤓ icon is still visible. Click the ⤓ icon to load all available results.

5 To stop an analysis, select **Polyspace** > **Stop**. Alternatively you can use the ■ button in the **Polyspace Run - Bug Finder** view.

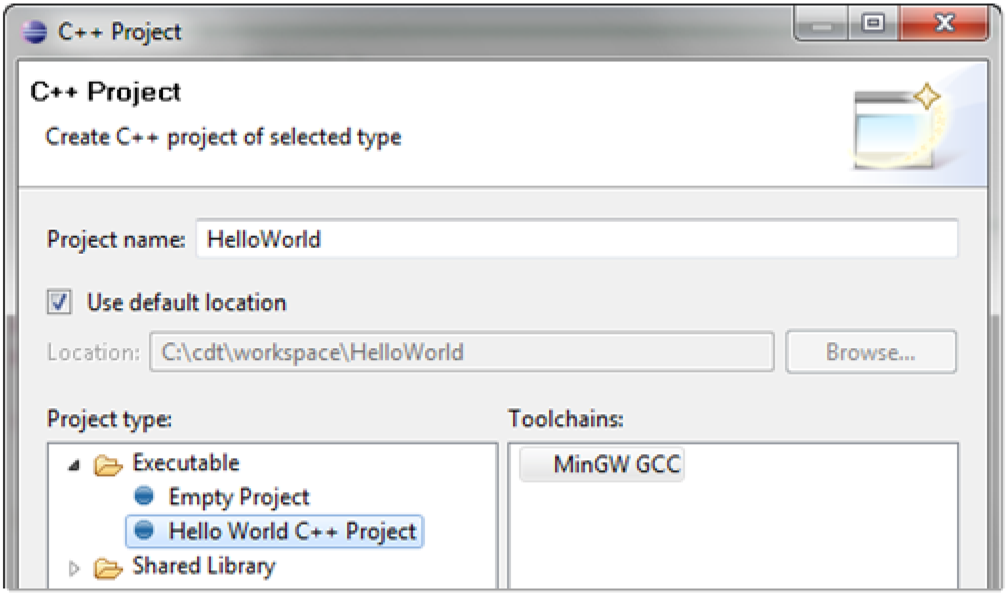
Customize Analysis Options

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. 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.
- Other options: Through the other options, you specify which analysis results you want and how precise you want them to be. If the default values of the options are not optimal for your situation, change them. For instance, the default analysis checks for certain defects only. If you also want coding rule violations, specify the type of rules that you want checked.

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. If you directly refer to your compilation toolchain in the Eclipse project, configure the analysis as follows.



C++ Project

Create C++ project of selected type

Project name: HelloWorld

☑ Use default location

Location: C:\cdt\workspace\HelloWorld

Project type:

- Executable
  - Empty Project
  - Hello World C++ Project
- Shared Library

Toolchains:

MinGW GCC

Compiler Options

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).

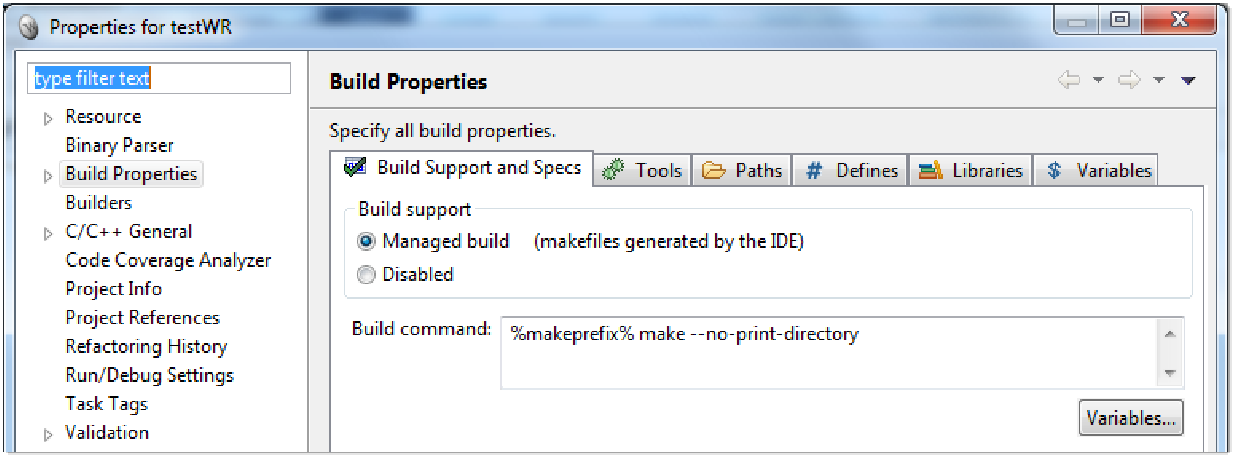
Other Options

You specify other options not related to your compiler directly in the Polyspace configuration. To open the Polyspace Bug Finder Configuration window, select

Polyspace > Configure Project. Specify the analysis options and save them. For more information, see “Analysis Options”.

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 refer to your compilation toolchain through a build command, configure the analysis as follows.



Specify all build properties.

Build support

● Managed build (makefiles generated by the IDE)

○ Disabled

Build command: %makeprefix% make --no-print-directory

Compiler Options

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).

- *`projectName`* 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.

Other Options

You can specify other options not related to your compiler directly in the Polyspace configuration. Select **Polyspace** > **Configure Project** to open the Polyspace Bug Finder Configuration window. Specify the analysis options and save them. For more information, see For more information, see “Analysis Options”.

Related Examples

- “Run Analysis” on page 12-2

View Results in Eclipse

View Results

This example shows how to view Polyspace Bug Finder results. After you run an analysis, you can view the results either in Eclipse or from the Polyspace Bug Finder interface.

In this section...
“View Results in Eclipse” on page 13-2
“View Results in Polyspace Environment” on page 13-2
“Results Location” on page 13-2

View Results in Eclipse

After you run an analysis in Eclipse, your results automatically appear on the **Results List - Bug Finder** tab.

- If you closed the **Results List - Bug Finder** tab, select **Polyspace** > **Show View** > **Show Results List view** to reopen the tab.
- If you need to reload the results, select **Polyspace** > **Reload results**.

  This option is useful when you reopen Eclipse or when you are switching between Polyspace projects.

View Results in Polyspace Environment

To view your results in the Polyspace Bug Finder interface, select **Polyspace** > **Open Results in PVE**.

Note: You can view defects, coding rule violations and code metrics from the Eclipse environment. However, you can impose limits on metrics only from the Polyspace environment. For more information, see “Review Code Metrics” on page 5-35.

Results Location

Polyspace stores your results from Eclipse in the following folder:

```
Polyspace_Workspace\EclipseProjects\Project_Name
```

Where *`Project_Name`* is the name of your Eclipse project and *`Polyspace_Workspace`* is the default Polyspace project location. You can change the *`Polyspace_Workspace`* in the Polyspace interface preferences.

1. In the Polyspace interface, select **Tools** > **Preferences**.
2. On the **Project and Results Folder** tab, change the value of the **Default project location**.

If you prefer to store your results within your Eclipse project, inside your Eclipse project folder, create a folder named `polyspace`. Polyspace will save your analysis results inside this folder.

Related Examples

- "Run Analysis" on page 12-2
- "Open Results" on page 5-2

Review and Fix Results

This example shows how to review and comment results obtained from a Polyspace Bug Finder analysis. When reviewing results, you can assign a status and severity to the defects and enter comments to describe the results of your review. These actions help you to track the progress of your review and avoid reviewing the same defect twice. If you run successive analyses on the same project, the review status, severity and comments from the previous analysis will be automatically imported into the next.

After analysis, the results appear on the **Results List - Bug Finder** tab. In addition, Polyspace Bug Finder highlights defects in your source code in the following ways:

- A ○ mark appears before the line number on the left.
- The operation containing the defect has a wavy red underlining.
- A▭ icon appears in the overview ruler to the right of the line containing the defect.
- A 🟥 icon appears at the top of the overview ruler. If you place your cursor on the icon, a tooltip states the total number of defects in the file.

To further review a defect and determine your course of action:

1. On the **Results List - Bug Finder** tab, select the defect that you want to review.

   The **Result Details** pane displays information about the current defect.
2. On the **Result Details** pane, click the ⓘ icon to see a brief description and code examples for the defect. In some cases, you can also see risks associated with not fixing the defect and a suggested fix.
3. Investigate the result further. Determine whether to fix your code, review the result later, or retain the code but provide some explanation.
4. On the **Result Details** pane, provide the following review information for the result:

   - **Severity** to describe how critical you consider the issue.
   - **Status** to describe how you intend to address the issue.

     You can also create your own status or associate justification with an existing status from the Polyspace user interface. Select **Tools** > **Preferences** and create or modify statuses on the **Review Statuses** tab.
   - **Comment** to describe any other information about the result.

5 To provide review information for several results together, select the results. Then, provide review information for a single result.

To select the results in a group:

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

  To group certain results together, use the column headers on the **Results List - Bug Finder** tab.
- If the results are not contiguous, **Ctrl**-left click each result.
- If the results belong to the same group and have the same color, right-click one result. From the context menu, select **Select All *Type* Results**.

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

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

Related Examples

- "View Results" on page 13-2
- "Filter and Group Results" on page 13-8

Limit Display of Defects

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

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

If you do not want to change your analysis configuration, you can still change which defects are displayed in your results. There are two ways to filter defects from your results:

- Filter individual defects from display after each run.

  For more information, see “Filter and Group Results” on page 13-8.
- Create a set of filters that you can apply in one click, called a scope.

This example shows how to create a scope:

1. Select **Polyspace** > **Configure Project**.
2. On the configuration window, select **Tools** > **Preferences**.
3. On the **Review Scope** tab, create your filter file.

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

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

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

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

      To suppress all defects with a certain impact such as **Low**, clear the box next to the impact. For more information on impacts, see “Classification of Defects by Impact” on page 5-9. If only a fraction of defects with a certain impact are selected, the check box next to the impact displays a ▣ symbol.

4 Select **Apply** or **OK**.

On the **Results List** pane, a menu displays the list of review scopes. The default scopes are `All` and `Defects & Rules`.

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

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

Related Examples

- “Filter and Group Results” on page 13-8

Filter and Group Results

This example shows how to filter and group defects on the **Results List - Bug Finder** tab. To organize your review of results, use filters and groups when you want to:

- Review only high-impact defects.

  For more information on impact, see "Classification of Defects by Impact" on page 5-9.
- Review certain types of defects in preference to others.

  For instance, you first want to address the defects resulting from **Missing or invalid return statement**.
- Review only new results found since the last analysis.
- Not address the full set of coding rule violations detected by the coding rules checker.
- Review only those defects that you have already assigned a certain status.

  For instance, you want to review only those defects to which you have assigned the status, `Investigate`.
- Review defects from a particular file or function. Because of continuity of code, reviewing these defects together can help you organize your review process.

  If you have written the code for a particular source file, you can review the defects only in that file.

In this section...

Filter Results

- To filter results from the **Results List - Bug Finder** tab, click the ☑ icon on the appropriate column. Clear **All**. Select the boxes for the results that you want displayed.

Item to Filter	Column
Results in a certain file or function	**File** or **Function**

Item to Filter	Column
Defects of a certain type, for instance, **Integer division by zero**	**Check** The column does not appear if you group checks by family. See “Group Results” on page 13-9.
Results with a certain severity or status	**Severity** or **Status**
Results in a certain group such as numerical or data flow	**Group**
Results with a certain impact	**Information**
Results that correspond to certain CWE IDs.	**CWE ID** For more information, see “CWE Coding Standard and Polyspace Results” on page 5-99.

- To review only new results found since the last analysis, on the **Results List - Bug Finder** tab, select New.
- To suppress code metrics from your results, from the drop-down list in the left of the **Results List - Bug Finder** tab, select **Defects & Rules**.

  You can increase the options on this list or create your own options from the Polyspace user interface. For examples, see:

  - “Suppress Certain Rules from Display in One Click” on page 3-18
  - “Limit Display of Defects” on page 5-18
  - “Review Code Metrics” on page 5-35

Note: You can also apply multiple filters.

Group Results

On the **Results List - Bug Finder** tab, from the ☰▾ list, select an option.

- To show results without grouping, select **None**.

- To show results grouped by result type, select **Family**.
  - The defects are organized by the defect groups. For more information on the groups, see “Defects”.
  - The coding rule violations are grouped by type of coding rule. For more information, see “Coding Rules”.
  - The code metrics are grouped by scope of metric. For more information, see “Code Metrics”.
- To show results grouped by file, select **File**.

  Within each file, the results are grouped by function. The results that are not associated with a particular function are grouped under **File Scope**.
- For C++ code, to show results grouped by class, select **Class**. The results that are not associated with a particular class are grouped under **Global Scope**.

  Within each class, the results are grouped by method.

Related Examples

- “View Results” on page 13-2
- “Review and Fix Results” on page 13-4

Understanding the Results Views

In this section...

Results List

The **Results List - Bug Finder** tab lists the defects and coding rule violations along with their attributes. To organize your results review, from the ☰▾ list on this tab, select one of the following options:

- **None**: Lists defects and coding rule violations without grouping. By default the results are listed in order of severity.
- **Family**: Lists results grouped by defect group. For more information on the defect groups, see "Bug Finder Defect Groups" on page 5-56.
- **Class**: Lists results grouped by class. Within each class, the results are grouped by method. The first group, **Global Scope**, lists results not occurring in a class definition.

  This option is available for `C++` code only.
- **File**: Lists results grouped by file. Within each file, the results are grouped by function.

For each defect, the **Results List** pane contains the defect attributes, listed in columns:

Attribute	Description
Family	Grouping to which the defect belongs. For example, if you choose the `Checks by File/Function` grouping, this column contains the name of the file and function containing the defect.
ID	Unique identification number of the defect. In the default view on the **Results List - Bug Finder** tab, the defects appear sorted by this number.
Type	Defect or coding rule violation.

Attribute	Description
Group	Category of the defect. For more information on the defect groups, see "Bug Finder Defect Groups" on page 5-56.
Check	Description of the defect
CWE ID	CWE ID-s that correspond to the defect. For more information, see "CWE Coding Standard and Polyspace Results" on page 5-99.
File	File containing the instruction where the defect occurs
Class	Class containing the instruction where the defect occurs. If the defect is not inside a class definition, then this column contains the entry, `Global Scope`.
Function	Function containing the instruction where the defect occurs. If the function is a method of a class, it appears in the format *`class_name`*`::`*`function_name`*.
Severity	Level of severity you have assigned to the defect. The possible levels are: • `High` • `Medium` • `Low` • `Not a defect`
Status	Review status you have assigned to the check. The possible statuses are: • `Fix` • `Improve` • `Investigate` • `Justified` • `No action planned` • `Other`
Comments	Comments you have entered about the check

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

Using this pane, you can:

- Navigate through the checks. For more information, see “Review and Fix Results” on page 13-4.
- Organize your check review using filters on the columns. For more information, see “Filter and Group Results” on page 13-8.

Result Details

The **Result Details** pane contains detailed information about a specific defect. Select a defect on the **Results List - Bug Finder** tab to reveal further information about the defect on the **Result Details** pane.

- The top right hand corner shows the file and function containing the defect, in the format `file_name/function_name`.
- The yellow box contains the name of the defect, along with an explanation.
- The **Event** column lists the sequence of code instructions causing the defect. The **Scope** column lists the name of the function containing the instructions. The **Line** column lists the line number of the instructions.
- The **Variable trace** check box when selected reveals an additional set of instructions that are related to the defect.
- The ℹ button allows you to access documentation for the defect.

Troubleshooting in Polyspace Bug Finder

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.

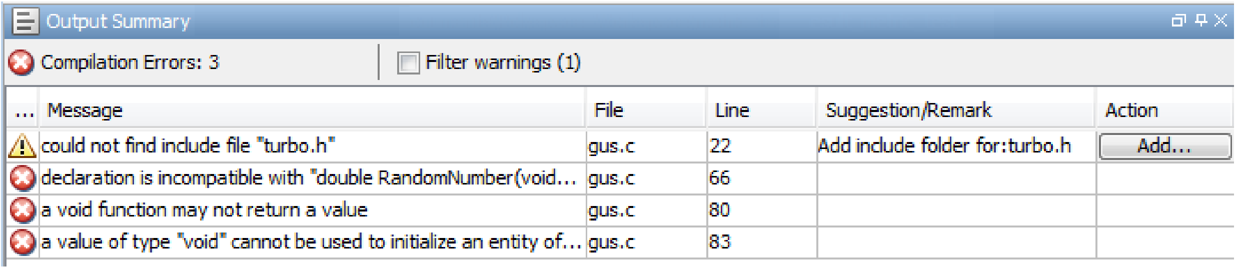
Icon	Meaning
⊗	Error that blocks analysis. For instance, the analysis cannot find a variable declaration or definition and therefore cannot determine the variable type.
⚠	Warning about an issue that does not block analysis by itself, but could be related to a blocking error. For instance, the analysis 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.
ℹ	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.



Output Summary

Compilation Errors: 3 | □ Filter warnings (1)

...	Message	File	Line	Suggestion/Remark	Action
⚠	could not find include file "turbo.h"	gus.c	22	Add include folder for:turbo.h	Add...
⊗	declaration is incompatible with "double RandomNumber(void...	gus.c	66		
⊗	a void function may not return a value	gus.c	80		
⊗	a value of type "void" cannot be used to initialize an entity of...	gus.c	83		

To turn on the Compilation Assistant, select **Tools** > **Preferences**. On the **Project and Results Folder** tab, select **Use Compilation Assistant**.

Note the following:

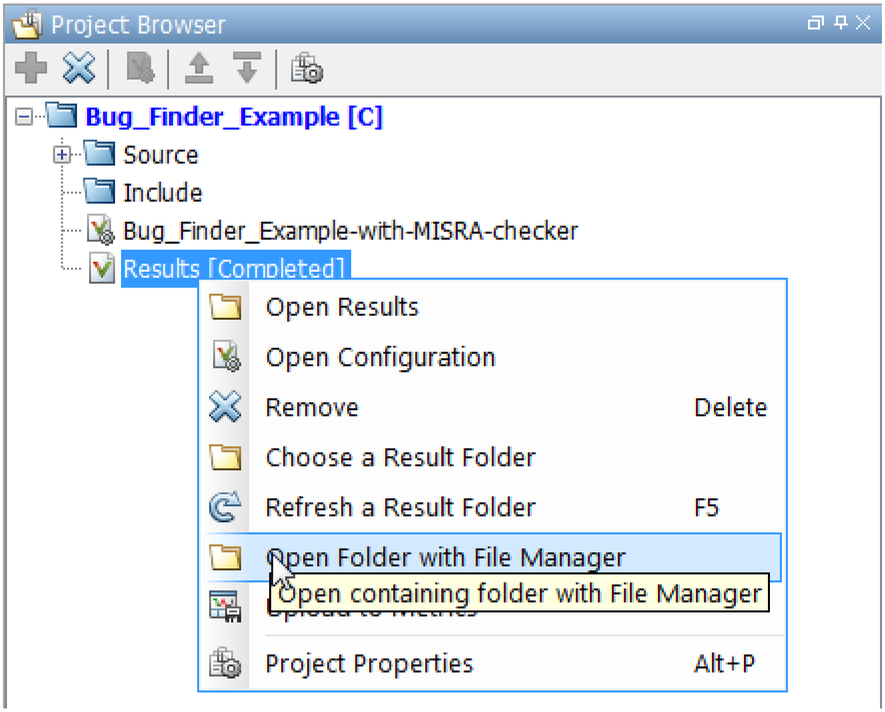
- By default, if some files do not compile, Bug Finder analyzes the remaining files. If you turn on Compilation Assistant, all files must compile. You do not get analysis results even if there is a single compilation error.
- The Compilation Assistant is disabled if you specify the option Command/script 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.

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**.

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 “Update Project” on page 1-15.
- 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 & 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.

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
```

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 “Update Project” on page 1-15.
- 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:

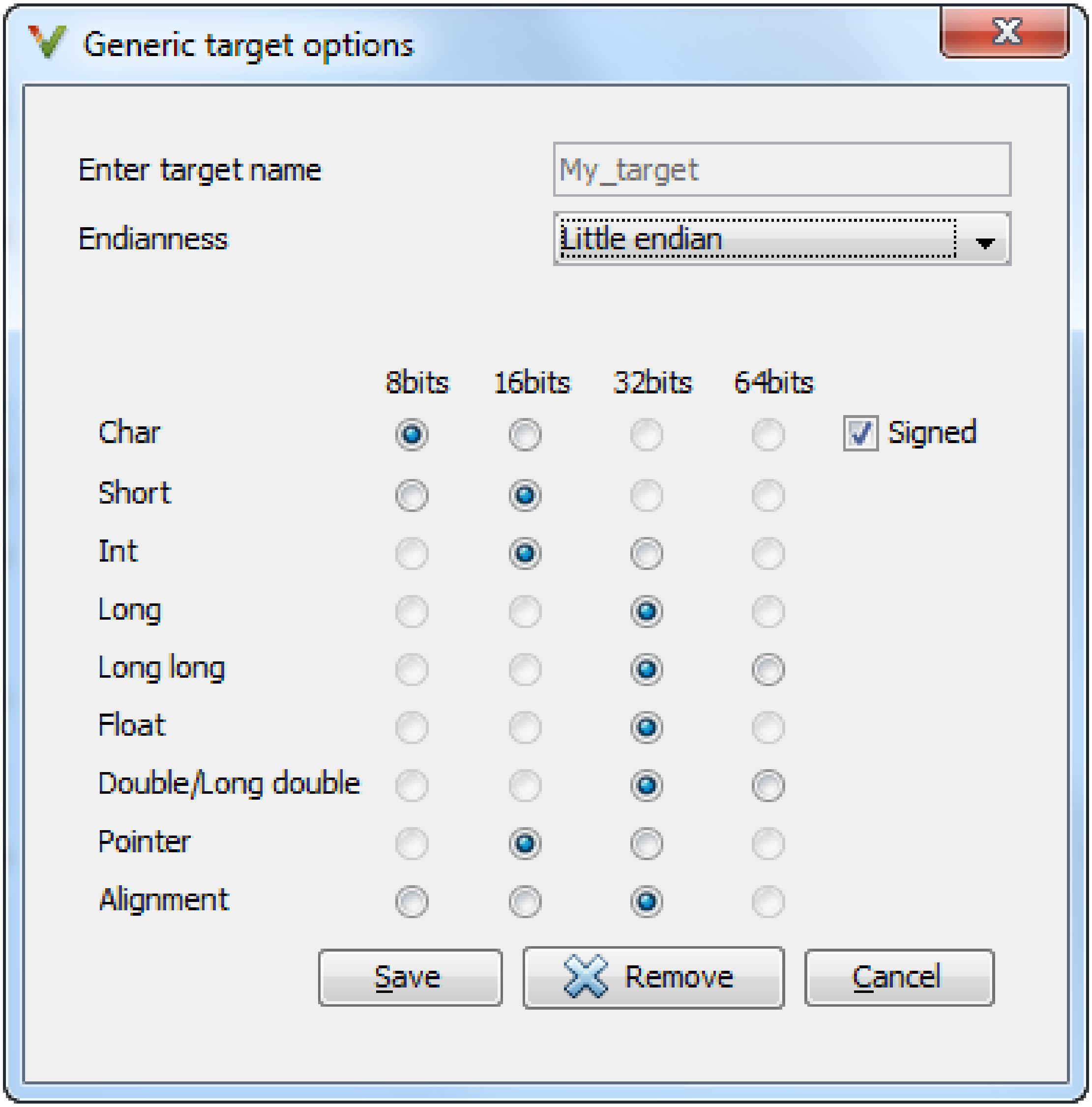
-
  ```
  struct S
  {
    char tab[65536];
  }s;
  ```

-
  ```
  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.



Generic target options

Enter target name: My_target

Endianness: Little endian

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

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 Keil or IAR Compiler

The following issue occurs if you use the compiler, Keil or IAR. 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 “Analyze Keil or IAR Compiled Code” on page 1-58.

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

The following issues can occur if you choose `diab` for the option Compiler (-compiler).

Errors from Language Extensions

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

The following issues can occur if you choose `tasking` for the option Compiler (-compiler).

Errors from SFRs

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 1-9.

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:

in file Test.h	in file Test.cpp

`class Test`
`{`
`public:`
`static int m_number;`
`};`
	`int Test::m_number = 0;`

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) (Polyspace Code Prover).
- 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 compile your code using a GNU C++ compiler, specify one of the GNU compilers for the Polyspace analysis. For more information, see Compiler (-compiler).

If you specify one of the GNU compilers, Polyspace does not produce an error during the **Compile** phase because of assembly language keywords such as `__asm__ __volatile__`. However, Polyspace ignores the content of the assembly-language code for the analysis.

Polyspace software supports the following GNU elements:

- Variable length arrays
- Anonymous structures:

```
void f(int n) { char tmp[n] ; /* ... */ }

union A {
    struct {
        double x;
        double y;
        double z;
    };
    double tab[3];
} a;


void main(void) {

    assert(&(a.tab[0]) == &(a.x));

}
```

- Other syntactic constructions allowed by GCC, except as noted below.
- Statement expressions:

```
int i = ({ int tmp ; tmp = f() ; if (tmp > 0 ) { tmp = 0 ; } tmp ; })
```

Partial Support

Zero-length arrays have the same support as in Visual Mode. They are allowed when used through a pointer, but not in a local variable.

Syntactic Support Only

Polyspace software provides syntactic support for the following options, but not semantic support:

- `__attribute__(...)` is allowed, but generally not taken into account.
- No special stubs are computed for predeclared functions such as `__builtin_cos`, `__builin_exit`, and `__builtin_fprintf`).

Not Supported

The following options are not supported:

- Taking the address of a label:

  ```
  { L : void *a = &&L ; goto *a ; }
  ```
- General C99 features supported by default in GCC, such as complex built-in types (`__complex__`, `__real__`, etc.).
- Extended designators initialization:

  ```
  struct X { double a; int b[10] } x = { .b = { 1, [5] =2 },
  .b[3] = 1, .a = 42.0 };
  ```
- Nested functions

Examples

Example 1: `_asm_volatile_` keyword

In the following example, for the `inb_p` function to manage the return of the local variable `_v`, the `__asm__ __volatile__` keyword is used as follows:

```
extern inline unsigned char
inb_p (unsigned short port)
{
  unsigned char _v;

  __asm__ __volatile__ ("inb %w1,%0\noutb %%al,$0x80":"=a"
                                   (_v):"Nd" (port));
  return _v;
}
```

```
...
```

Although Polyspace does not produce an error during the **Compile** phase, it ignores the assembly code. An orange **Non-initialized local variable** error appears on the `return` statement after verification.

Example 2: Anonymous Structure

The following example shows an unnamed structure supported by GNU:

```
class x
{
public:

  struct {
  unsigned int a;
  unsigned int b;
  unsigned int c;
  };
  unsigned short pcia;
  enum{
  ea = 0x1,
  eb = 0x2,
  ec = 0x3
  };

  struct {
  unsigned int z1;
  unsigned int z2;
  unsigned int z3;
  unsigned int z4;
  };
};

int main(int argc, char *argv[])
{
  class x myx;

  myx.a = 10;
  myx.z1 = 11;
  return(0);
}
```

Errors Related to ISO Compiler

The ISO setting for Compiler (-compiler) strictly follows the ISO/IEC 14882:1998 ANSI C++ standard. If you specify the setting `iso`, the Polyspace compiler might produce permissiveness errors. The following code contains five common permissiveness errors that occur if you specify the option. These errors are explained in detail following the code.

By default, the Polyspace compiler uses compilation standards that many C++ compilers use. The default compilation is more permissive with regard to the C++ standard.

Original code (file `permissive.cpp`):

```
class B {} ;
class A
{
    friend B ;
    enum e ;
    void f() {
        long float ff = 0.0 ;
    }
    enum e { OK = 0, KO } ;
};
template <class T>
struct traits
{
    typedef T * pointer ;
    typedef T * pointer ;
} ;
template<class T>
struct C
{
    typedef traits<T>::pointer pointer ;
} ;

void main()
{
    C<int> c;
}
```

If you use `iso` for Compiler (-compiler), the following errors occur.

Error message	Original code	Corrected code
`error: omission of`	`friend B;`	`friend class B;`

Error message	Original code	Corrected code

`"class"`
`is nonstandard`		

`forward declaration of enum type is nonstandard`	`enum e;`	The line must be removed.
`invalid combination of type specifiers`	`long float ff = 0.0;`	`double ff = 0.0;`
`class member typedef may not be redeclared`	Second instance of `typedef T * pointer;`	The line must be removed.

`nontype "traits<T>::pointer [with T=T]" is not a type name`	`typedef \`
	`traits<T>::pointer\`
	`pointer;`
		`typedef`
		`typename`
		`traits<T>::pointer`
		`pointer;`

The error messages disappear if you specify `none` instead of `iso`.

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).

Eclipse Java Version Incompatible with Polyspace Plug-in

In this section...
"Issue" on page 14-40
"Cause" on page 14-40
"Solution" on page 14-40

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.

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 1-77.
- 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 1-77.
- 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 1-77.
- 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.

Troubleshooting Project Creation from Visual Studio Build

In this section...

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`.

Compiler Not Supported for Project Creation from Build Systems

Issue

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

For more information on automatic project creation, see:

- “Create Project Automatically” on page 1-6
- “Create Project Automatically at Command Line” on page 6-2
- “Create Project Automatically from MATLAB Command Line” on page 6-23

Cause

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

For information on which compilers are supported, see “Requirements for Project Creation from Build Systems” on page 1-9.

Solution

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

1. Copy one of the existing configuration files from `*matlabroot*\polyspace\configure\compiler_configuration\`.
2. Save the file as `*my_compiler*.xml`. `*my_compiler*` can be 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.

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

XML Element	Content Description	Content Example for GNU C Compiler

`<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`

`<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`

XML Element	Content Description	Content Example for GNU C Compiler

`<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`

`<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`

XML Element	Content Description	Content Example for GNU C Compiler

`<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 Code Prover).

`<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) (Polyspace Code Prover).

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

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

Slow Build Process When Polyspace Traces the Build

Issue

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

Cause

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

Solution

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

- If you trace your build from the Polyspace user interface, enter this flag in the field **Add advanced configure options**. For more information, see "Create Project Automatically" on page 1-6.
- 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.

For more information on automatic project creation from build commands, see:

- "Create Project Automatically" on page 1-6
- "Create Project Automatically at Command Line" on page 6-2
- "Create Project Automatically from MATLAB Command Line" on page 6-23

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\R2017a\polyspace\bin\polyspace-configure.exe"
             -output-options-file myOptions.txt launching.bat
   ```

You can now run polyspace-bug-finder-nodesktop on the options file.

Software Quality with Polyspace Metrics

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 15-7.

Before you generate code quality metrics, set up Polyspace Metrics. See “Set Up Polyspace Metrics”.

In this section...

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).

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).

Related Examples

- “View Projects in Polyspace Metrics” on page 15-4
- “Compare Metrics Against Software Quality Objectives” on page 15-12

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 15-2.

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 15-7.

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 provides a high-level overview of the verification results.
- The **Code Metrics** tab provides the details of the code complexity metrics in your results.
- The **Coding Rules** tab provides the details of the coding rule violations in your results.

- The **Bug Finder** tab provides details of code defects in your results.

If you want to “Compare Metrics Against Software Quality Objectives” on page 15-12, 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 15-4.

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 `Not a defect`.

Polyspace Metrics Interface

- “Coding Rules Tab” on page 15-9
- “Bug-Finder Tab” on page 15-10

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 15-12.

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.
	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.

Column Name		Description
Verification		**Version number of the results and the source files.**
	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.
	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.

Column Name		Description
	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 "Review and Fix Results" on page 5-32.
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 5-9.
	Medium	Total number of "Medium Impact Defects" on page 5-12.
	Low	Total number of "Low Impact Defects" on page 5-15.
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.

Related Examples

- "Assign and Save Comments" on page 5-32
- "Upload Results to Polyspace Metrics" on page 15-2
- "Compare Metrics Against Software Quality Objectives" on page 15-12
- "Compare Metrics Between Results" on page 15-6

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 15-13.
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.

   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
Number of call levels in a function	4
Number of `return` statements in a function	1

Metric	Threshold Value
Language scope, an indicator of the cost of maintaining or changing functions. Calculated as follows: `(N1+N2) / (n1+n2)` • *n1* — Number of different operators • *N1* — Total number of operators • *n2* — Number of different operands • *N2* — Total number of operands	4
Number of recursions	0
Number of direct recursions	0
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	0

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 5-9	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 5-12	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
Number of unjustified violations of the following MISRA C:2012 rules: • 11.8 • 12.1 and 12.3 • 13.2 and 13.4	0

Metric	Threshold Value
• 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	
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 5-15	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

Metric	Threshold Value
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 2-14
MISRA C: 2012	`MISRA_C3_`	“MISRA C:2012 Directives and Rules”
MISRA C++	`MISRA_Cpp_`	“MISRA C++ Coding Rules” on page 2-88
JSF C++	`JSF_Cpp_`	“JSF C++ Coding Rules” on page 2-117
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 15-21
- “Non-HIS Metrics” on page 15-21

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 (Polyspace Code Prover)
`path`	Number of Paths (Polyspace Code Prover)
`goto`	Number of Goto Statements (Polyspace Code Prover)
`vg`	Cyclomatic Complexity (Polyspace Code Prover)
`calling`	Number of Calling Functions (Polyspace Code Prover)
`calls`	Number of Called Functions (Polyspace Code Prover)
`param`	Number of Function Parameters (Polyspace Code Prover)
`stmt`	Number of Instructions (Polyspace Code Prover)
`level`	Number of Call Levels (Polyspace Code Prover)
`return`	Number of Return Statements (Polyspace Code Prover)
`vocf`	Language Scope (Polyspace Code Prover)
`ap_cg_cycle`	Number of Recursions (Polyspace Code Prover)
`ap_cg_direct_cycle`	Number of Direct Recursions (Polyspace Code Prover)
`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 (Polyspace Code Prover)
`flin`	Number of Lines Within Body (Polyspace Code Prover)
`fxln`	Number of Executable Lines (Polyspace Code Prover)
`ncalls`	Number of Call Occurrences (Polyspace Code Prover)

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**.

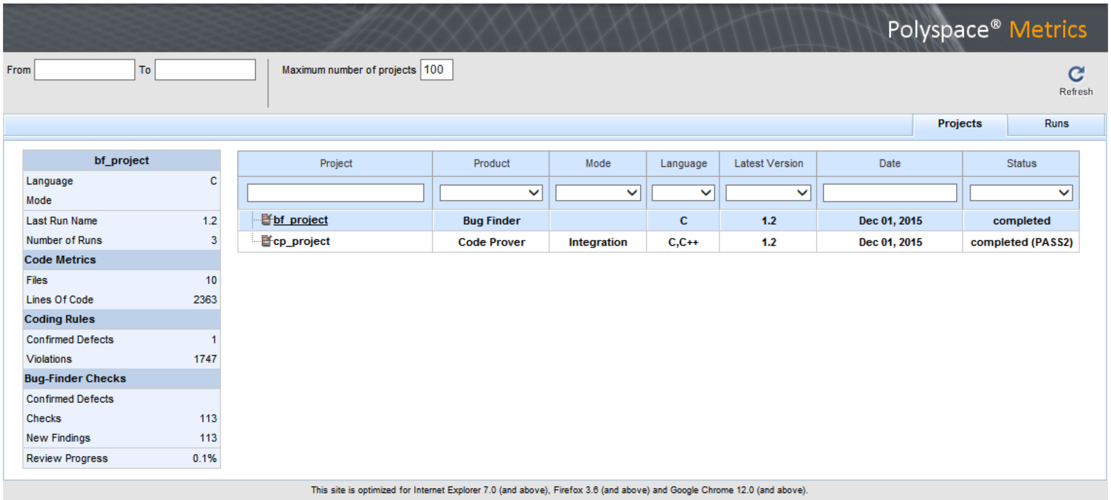
   You can also open the Polyspace Metrics Web UI using the URL:

   `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.



bf_project	
Language	C
Mode	
Last Run Name	1.2
Number of Runs	3
Code Metrics	
Files	10
Lines Of Code	2363
Coding Rules	
Confirmed Defects	1
Violations	1747
Bug-Finder Checks	
Confirmed Defects	
Checks	113
New Findings	113
Review Progress	0.1%

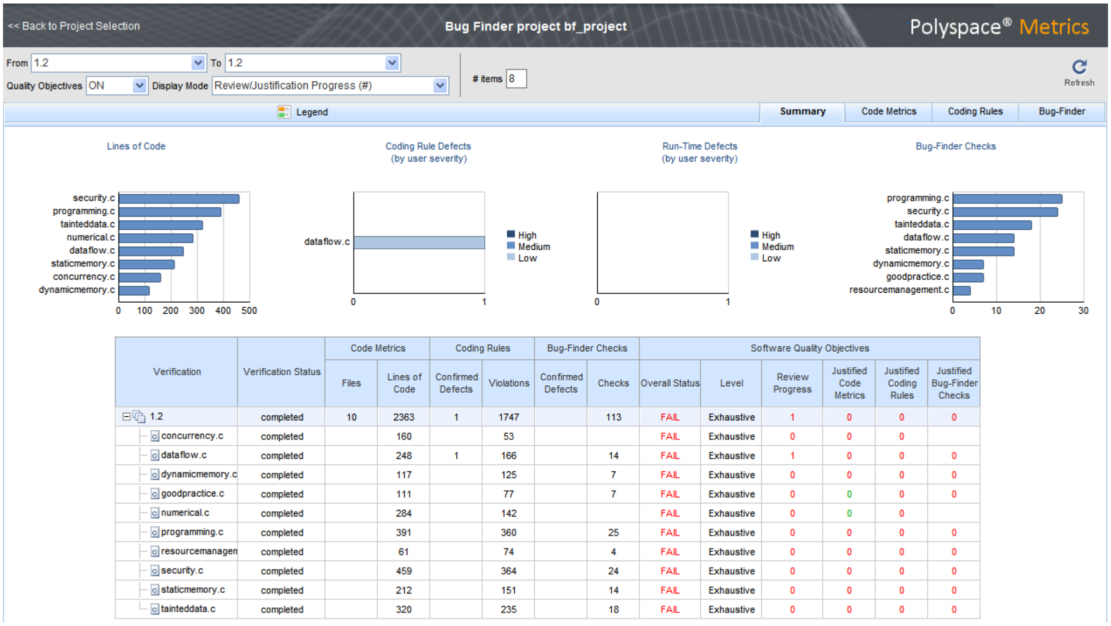
Project	Product	Mode	Language	Latest Version	Date	Status
bf_project	Bug Finder		C	1.2	Dec 01, 2015	completed
cp_project	Code Prover	Integration	C,C++	1.2	Dec 01, 2015	completed (PASS2)

This site is optimized for Internet Explorer 7.0 (and above), Firefox 3.6 (and above) and Google Chrome 12.0 (and above).

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.



From 1.2 To 1.2

Quality Objectives ON Display Mode Review/Justification Progress (#)

items 8

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:

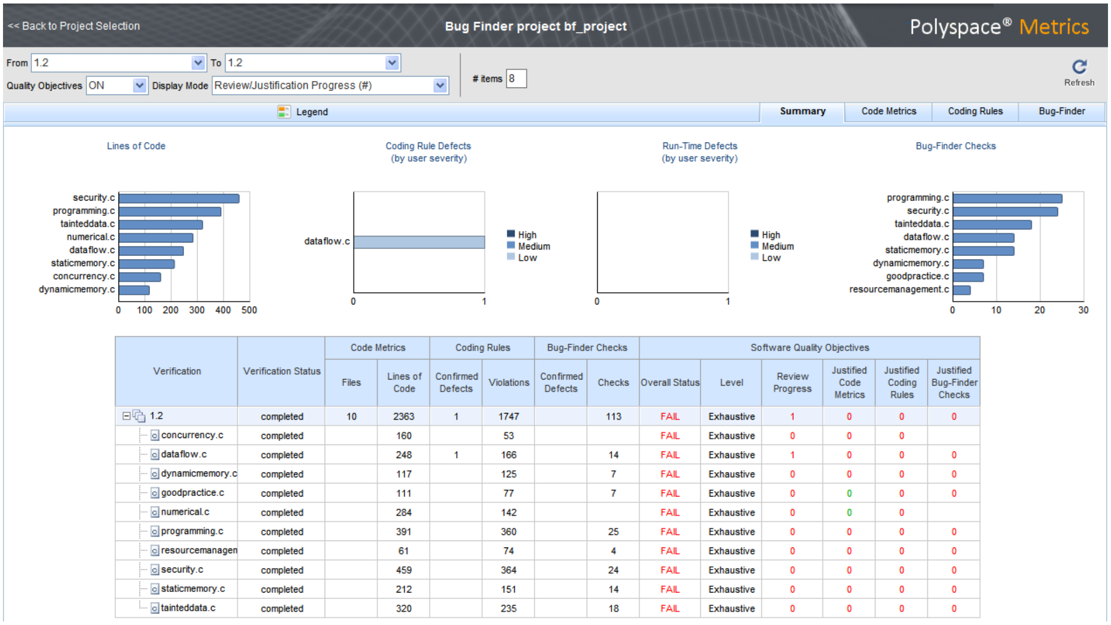
- **Code Metrics**: Statistics about your project such as number of lines, header files, and function calls. To see code metrics, you must enable the analysis option Calculate code metrics (-code-metrics).
- **Coding Rules**: Description of coding rule violations.
- **Bug-Finder**: Description of defects 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.



From 1.2 To 1.2

Quality Objectives ON Display Mode Review/Justification Progress (#)

items 8

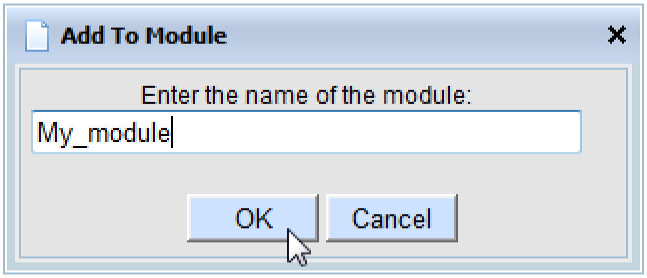
Verification	Verification Status	Code Metrics		Coding Rules		Bug-Finder Checks		Software Quality Objectives					
		Files	Lines of Code	Confirmed Defects	Violations	Confirmed Defects	Checks	Overall Status	Level	Review Progress	Justified Code Metrics	Justified Coding Rules	Justified Bug-Finder Checks
1.2	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

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**.



Enter the name of the module:

My_module

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.

Related Examples

- “Set Up Polyspace Metrics”
- “Review and Fix Results” on page 5-32